Dive into Deep Learning

Release 0.7.1

Aston Zhang, Zachary C. Lipton, Mu Li, and Alexander J. Smola

Contents

Preface

Just a few years ago, there were no legions of deep learning scientists developing intelligent products and services at major companies and startups. When the youngest among us (the authors) entered the field, machine learning did not command headlines in daily newspapers. Our parents had no idea what machine learning was, let alone why we might prefer it to a career in medicine or law. Machine learning was a forward-looking academic discipline with a narrow set of real-world applications. And those applications, e.g., speech recognition and computer vision, required so much domain knowledge that they were often regarded as separate areas entirely for which machine learning was one small component. Neural networks then, the antecedents of the deep learning models that we focus on in this book, were regarded as outmoded tools.

In just the past five years, deep learning has taken the world by surprise, driving rapid progress in fields as diverse as computer vision, natural language processing, automatic speech recognition, reinforcement learning, and statistical modeling. With these advances in hand, we can now build cars that drive themselves with more autonomy than ever before (and less autonomy than some companies might have you believe), smart reply systems that automatically draft the most mundane emails, helping people dig out from oppressively large inboxes, and software agents that dominate the world's best humans at board games like Go, a feat once thought to be decades away. Already, these tools exert ever-wider impacts on industry and society, changing the way movies are made, diseases are diagnosed, and playing a growing role in basic sciences—from astrophysics to biology.

About This Book

This book represents our attempt to make deep learning approachable, teaching you both the *concepts*, the *context*, and the *code*.

One Medium Combining Code, Math, and HTML

For any computing technology to reach its full impact, it must be well-understood, well-documented, and supported by mature, well-maintained tools. The key ideas should be clearly distilled, minimizing the onboarding time needing to bring new practitioners up to date. Mature libraries should automate common tasks, and exemplar code should make it easy for practitioners to modify, apply, and extend common applications to suit their needs. Take dynamic web applications as an example. Despite a large number of companies, like Amazon, developing successful database-driven web applications in the 1990s, the potential of this technology to aid creative entrepreneurs has been realized to a far greater degree in the past ten years, owing in part to the development of powerful, well-documented frameworks.

Testing the potential of deep learning presents unique challenges because any single application brings together various disciplines. Applying deep learning requires simultaneously understanding (i) the motivations for casting a problem in a particular way; (ii) the mathematics of a given modeling approach; (iii) the optimization algorithms for fitting the models to data; and (iv) and the engineering required to train models efficiently, navigating the pitfalls of numerical computing and getting the most out of available hardware. Teaching both the critical thinking skills required to formulate problems, the mathematics to solve them, and the software tools to implement those solutions all in one place presents formidable challenges. Our goal in this book is to present a unified resource to bring would-be practitioners up to speed.

We started this book project in July 2017 when we needed to explain MXNet's (then new) Gluon interface to our users. At the time, there were no resources that simultaneously (i) were up to date; (ii) covered the full breadth of modern machine learning with substantial technical depth; and (iii) interleaved exposition of the quality one expects from an engaging textbook with the clean runnable code that one expects to find in hands-on tutorials. We found plenty of code examples for how to use a given deep learning framework (e.g., how to do basic numerical computing with matrices in TensorFlow) or for implementing particular techniques (e.g., code snippets for LeNet, AlexNet, ResNets, etc) scattered across various blog posts and GitHub repositories. However, these examples typically focused on *how* to implement a given approach, but left out the discussion of *why* certain algorithmic decisions are made. While some interactive resources have popped up sporadically to address a particular topic, e.g., the engaging blog posts published on the website Distill[1], or personal blogs, they only covered selected topics in deep learning, and often lacked associated code. On the other hand, while several textbooks have emerged, most notably (Goodfellow et al., 2016), which offers a comprehensive survey of the concepts behind deep learning, these resources do not marry the descriptions to realizations of the concepts in code, sometimes leaving readers clueless as to how to implement them. Moreover, too many resources are hidden behind the paywalls of commercial course providers.

We set out to create a resource that could (1) be freely available for everyone; (2) offer sufficient technical depth to provide a starting point on the path to actually becoming an applied machine learning scientist; (3) include runnable code, showing readers *how* to solve problems in practice; (4) that allowed for rapid updates, both by us and also by the community at large; and (5) be complemented by a forum[2] for interactive discussion of technical details and to answer questions.

These goals were often in conflict. Equations, theorems, and citations are best managed and laid out in LaTeX. Code is best described in Python. And webpages are native in HTML and JavaScript. Furthermore, we want the content to be accessible both as executable code, as a physical book, as a downloadable PDF, and on the internet as a website. At present there exist no tools and no workflow perfectly suited to these demands, so we had to assemble our own. We describe our approach in detail in Section 18.6. We settled on Github to share the source and to allow for edits, Jupyter notebooks for mixing code, equations and text, Sphinx as a rendering engine to generate multiple outputs, and Discourse for the forum. While our system is not yet perfect, these choices provide a good compromise among the competing concerns. We believe that this might be the first book published using such an integrated workflow.

[1] http://distill.pub

[2] http://discuss.mxnet.io

Learning by Doing

Many textbooks teach a series of topics, each in exhaustive detail. For example, Chris Bishop's excellent textbook (Bishop, 2006), teaches each topic so thoroughly, that getting to the chapter on linear regression requires a non-trivial amount of work. While experts love this book precisely for its thoroughness, for beginners, this property limits its usefulness as an introductory text.

In this book, we will teach most concepts *just in time*. In other words, you will learn concepts at the very moment that they are needed to accomplish some practical end. While we take some time at the outset to teach fundamental preliminaries, like linear algebra and probability, we want you to taste the satisfaction of training your first model before worrying about more esoteric probability distributions.

Aside from a few preliminary notebooks that provide a crash course in the basic mathematical background, each subsequent chapter introduces both a reasonable number of new concepts and provides single self-contained working examples—using real datasets. This presents an organizational challenge. Some models might logically be grouped together in a single notebook. And some ideas might be best taught by executing several models in succession. On the other hand, there is a big advantage to adhering to a policy of *1 working example, 1 notebook*: This makes it as easy as possible for you to start your own research projects by leveraging our code. Just copy a notebook and start modifying it.

We will interleave the runnable code with background material as needed. In general, we will often err on the side of making tools available before explaining them fully (and we will follow up by explaining the background later). For instance, we might use *stochastic gradient descent* before fully explaining why it is useful or why it works. This helps to give practitioners the necessary ammunition to solve problems quickly, at the expense of requiring the reader to trust us with some curatorial decisions.

Throughout, we will be working with the MXNet library, which has the rare property of being flexible enough for research while being fast enough for production. This book will teach deep learning concepts from scratch. Sometimes, we want to delve into fine details about the models that would typically be hidden from the user by Gluon's advanced abstractions. This comes up especially in the basic tutorials, where we want you to understand everything that happens in a given layer or optimizer. In these cases, we will often present two versions of the example: one where we implement everything from scratch, relying only on the NumPy interface and automatic differentiation, and another, more practical example, where we write succinct code using Gluon. Once we have taught you how some component works, we can just use the Gluon version in subsequent tutorials.

Content and Structure

The book can be roughly divided into three parts, which are presented by different colors in Fig. 1:

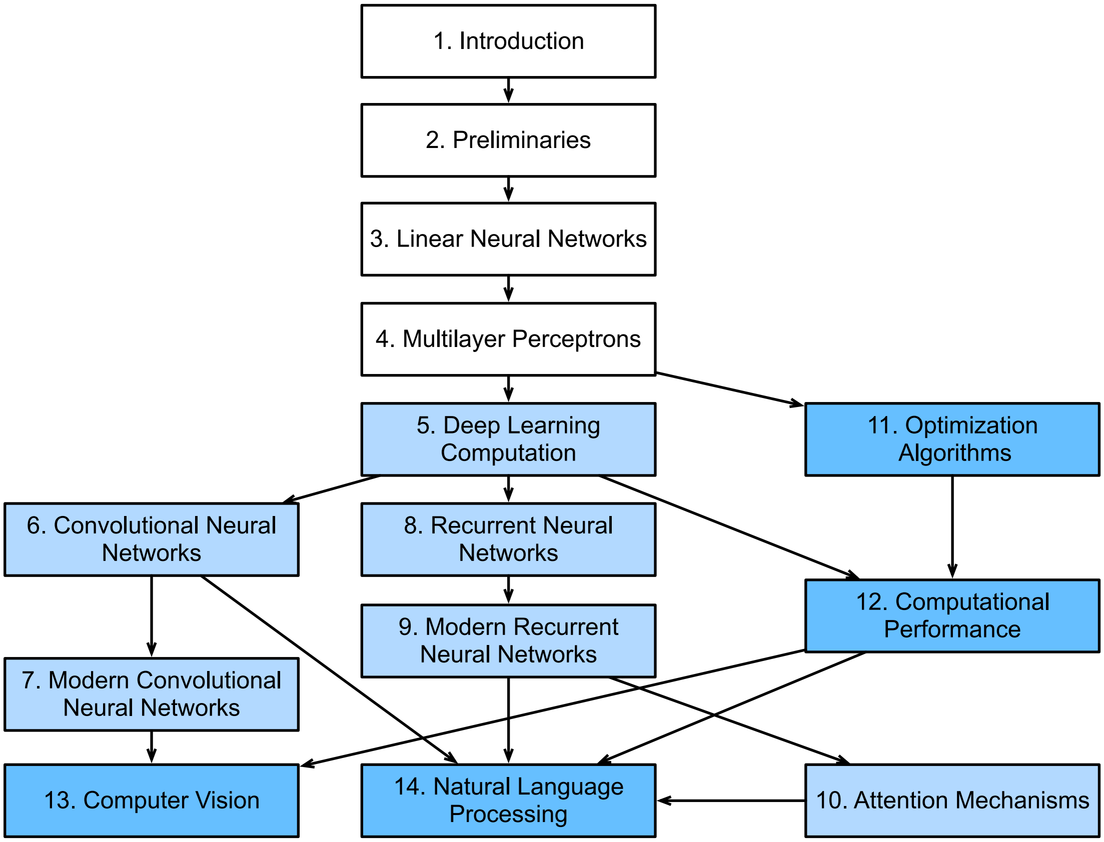

1. Introduction
2. Preliminaries
3. Linear Neural Networks
4. Multilayer Perceptrons
5. Deep Learning Computation
11. Optimization Algorithms
6. Convolutional Neural Networks
8. Recurrent Neural Networks
12. Computational Performance
9. Modern Recurrent Neural Networks
7. Modern Convolutional Neural Networks
13. Computer Vision
14. Natural Language Processing
10. Attention Mechanisms

Fig. 1: Book structure

- The first part covers basics and preliminaries. Chapter 1 offers an introduction to deep learning. Then, in Chapter 2, we quickly bring you up to speed on the prerequisites required for hands-on deep learning, such as how to store and manipulate data, and how to apply various numerical operations based on basic concepts from linear algebra, calculus, and probability. Chapter 3 and Chapter 4 cover the most basic concepts and techniques of deep learning, such as linear regression, multilayer perceptrons and regularization.
- The next five chapters focus on modern deep learning techniques. Chapter 5 describes the various key components of deep learning calculations and lays the groundwork for us to subsequently implement more complex models. Next, in Chapter 6 and Chapter 7, we introduce convolutional neural networks (CNNs), powerful tools that form the backbone of most modern computer vision systems. Subsequently, in Chapter 8 and Chapter 9, we introduce recurrent neural networks (RNNs), models that exploit temporal or sequential structure in data, and are commonly used for natural language processing and time series prediction. In Chapter 10, we introduce a new class of models that employ a technique called attention mechanisms and they have recently begun to displace RNNs in natural language processing. These sections will get you up to speed on the basic tools behind most modern applications of deep learning.
- Part three discusses scalability, efficiency, and applications. First, in Chapter 11, we discuss several common optimization algorithms used to train deep learning models. The next

chapter, Chapter 12 examines several key factors that influence the computational performance of your deep learning code. In Chapter 13 and Chapter 14, we illustrate major applications of deep learning in computer vision and natural language processing, respectively.

Code

Most sections of this book feature executable code because of our belief in the importance of an interactive learning experience in deep learning. At present, certain intuitions can only be developed through trial and error, tweaking the code in small ways and observing the results. Ideally, an elegant mathematical theory might tell us precisely how to tweak our code to achieve a desired result. Unfortunately, at present, such elegant theories elude us. Despite our best attempts, formal explanations for various techniques are still lacking, both because the mathematics to characterize these models can be so difficult and also because serious inquiry on these topics has only just recently kicked into high gear. We are hopeful that as the theory of deep learning progresses, future editions of this book will be able to provide insights in places the present edition cannot.

Most of the code in this book is based on Apache MXNet. MXNet is an open-source framework for deep learning and the preferred choice of AWS (Amazon Web Services), as well as many colleges and companies. All of the code in this book has passed tests under the newest MXNet version. However, due to the rapid development of deep learning, some code *in the print edition* may not work properly in future versions of MXNet. However, we plan to keep the online version remain up-to-date. In case you encounter any such problems, please consult *Installation* (page 9) to update your code and runtime environment.

At times, to avoid unnecessary repetition, we encapsulate the frequently-imported and referred-to functions, classes, etc. in this book in the `d2l` package. For any block block such as a function, a class, or multiple imports to be saved in the package, we will mark it with `# Saved in the d2l package for later use`. The `d2l` package is light-weight and only requires the following packages and modules as dependencies:

```python
# Saved in the d2l package for later use
import collections
from collections import defaultdict
from IPython import display
import math
from matplotlib import pyplot as plt
from mxnet import autograd, context, gluon, image, init, np, npx
from mxnet.gluon import nn, rnn
import os
import pandas as pd
import random
import re
import sys
import tarfile
import time
import zipfile
```

We offer a detailed overview of these functions and classes in Section 18.7.

Target Audience

This book is for students (undergraduate or graduate), engineers, and researchers, who seek a solid grasp of the practical techniques of deep learning. Because we explain every concept from scratch, no previous background in deep learning or machine learning is required. Fully explaining the methods of deep learning requires some mathematics and programming, but we will only assume that you come in with some basics, including (the very basics of) linear algebra, calculus, probability, and Python programming. Moreover, in the Appendix, we provide a refresher on most of the mathematics covered in this book. Most of the time, we will prioritize intuition and ideas over mathematical rigor. There are many terrific books which can lead the interested reader further. For instance, Linear Analysis by Bela Bollobas (Bollobas, 1999) covers linear algebra and functional analysis in great depth. All of Statistics (Wasserman, 2013) is a terrific guide to statistics. And if you have not used Python before, you may want to peruse this Python tutorial[3].

Forum

Associated with this book, we have launched a discussion forum, located at discuss.mxnet.io[4]. When you have questions on any section of the book, you can find the associated discussion page by scanning the QR code at the end of the section to participate in its discussions. The authors of this book and broader MXNet developer community frequently participate in forum discussions.

Acknowledgments

We are indebted to the hundreds of contributors for both the English and the Chinese drafts. They helped improve the content and offered valuable feedback. Specifically, we thank every contributor of this English draft for making it better for everyone. Their GitHub IDs or names are (in no particular order): alxnorden, avinashingit, bowen0701, brettkoonce, Chaitanya Prakash Bapat, cryptonaut, Davide Fiocco, edgarroman, gkutiel, John Mitro, Liang Pu, Rahul Agarwal, Mohamed Ali Jamaoui, Michael (Stu) Stewart, Mike Müller, NRauschmayr, Prakhar Srivastav, sad-, sfermigier, Sheng Zha, sundeepteki, topecongiro, tpdi, vermicelli, Vishaal Kapoor, vishwesh5, YaYaB, Yuhong Chen, Evgeniy Smirnov, lgov, Simon Corston-Oliver, IgorDzreyev, Ha Nguyen, pmuens, alukovenko, senorcinco, vfdev-5, dsweet, Mohammad Mahdi Rahimi, Abhishek Gupta, uwsd, DomKM, Lisa Oakley, Bowen Li, Aarush Ahuja, Prasanth Buddareddygari, brianhendee, mani2106, mtn, lkevinzc, caojilin, Lakshya, Fiete Lüer, Surbhi Vijayvargeeya, Muhyun Kim, dennismalmgren, adursun, Anirudh Dagar, liqingnz, Pedro Larroy, lgov, ati-ozgur, Jun Wu, Matthias Blume, Lin Yuan, geogunow, Josh Gardner, Maximilian Böther, Rakib Islam, Leonard Lausen, Abhinav Upadhyay, rongruosong, Steve Sedlmeyer, ruslo, Rafael Schlatter, liusy182, Giannis Pappas, ruslo, ati-ozgur, qbaza, dchoi77, Adam Gerson, Phuc Le, Mark Atwood, christabella, vn09, Haibin Lin, jjangga0214, RichyChen.

We thank Amazon Web Services, especially Swami Sivasubramanian, Raju Gulabani, Charlie Bell, and Andrew Jassy for their generous support in writing this book. Without the available time, resources, discussions with colleagues, and continuous encouragement this book would not have happened.

[3] http://learnpython.org/

[4] https://discuss.mxnet.io/

Summary

- Deep learning has revolutionized pattern recognition, introducing technology that now powers a wide range of technologies, including computer vision, natural language processing, automatic speech recognition.
- To successfully apply deep learning, you must understand how to cast a problem, the mathematics of modeling, the algorithms for fitting your models to data, and the engineering techniques to implement it all.
- This book presents a comprehensive resource, including prose, figures, mathematics, and code, all in one place.
- To answer questions related to this book, visit our forum at https://discuss.mxnet.io/.
- Apache MXNet is a powerful library for coding up deep learning models and running them in parallel across GPU cores.
- Gluon is a high level library that makes it easy to code up deep learning models using Apache MXNet.
- Conda is a Python package manager that ensures that all software dependencies are met.
- All notebooks are available for download on GitHub.
- If you plan to run this code on GPUs, do not forget to install the necessary drivers and update your configuration.

Exercises

1. Register an account on the discussion forum of this book discuss.mxnet.io[5].
2. Install Python on your computer.
3. Follow the links at the bottom of the section to the forum, where you will be able to seek out help and discuss the book and find answers to your questions by engaging the authors and broader community.
4. Create an account on the forum and introduce yourself.

[5] https://discuss.mxnet.io/

Installation

In order to get you up and running for hands-on learning experience, we need to set you up with an environment for running Python, Jupyter notebooks, the relevant libraries, and the code needed to run the book itself.

Installing Miniconda

The simplest way to get going will be to install Miniconda[7]. The Python 3.x version is recommended. You can skip the following steps if conda has already been installed. Download the corresponding Miniconda sh file from the website and then execute the installation from the command line using `sh <FILENAME> -b`. For macOS users:

```
# The file name is subject to changes
sh Miniconda3-latest-MacOSX-x86_64.sh -b
```

For Linux users:

```
# The file name is subject to changes
sh Miniconda3-latest-Linux-x86_64.sh -b
```

Next, initialize the shell so we can run `conda` directly.

```
~/miniconda3/bin/conda init
```

Now close and re-open your current shell. You should be able to create a new environment as following:

```
conda create --name d2l -y
```

[7] https://conda.io/en/latest/miniconda.html

Downloading the D2L Notebooks

Next, we need to download the code of this book. You can use the link[8] to download and unzip the code. Alternatively, if you have unzip (otherwise run sudo apt install unzip) available:

```
mkdir d2l-en && cd d2l-en
curl https://d2l.ai/d2l-en-0.7.1.zip -o d2l-en.zip
unzip d2l-en.zip && rm d2l-en.zip
```

Now we will want to activate the d2l environment and install pip. Enter y for the queries that follow this command.

```
conda activate d2l
conda install python=3.7 pip -y
```

Installing MXNet and the d2l Package

Before installing MXNet, please first check whether or not you have proper GPUs on your machine (the GPUs that power the display on a standard laptop do not count for our purposes). If you are installing on a GPU server, proceed to *GPU Support* (page 11) for instructions to install a GPU-supported MXNet.

Otherwise, you can install the CPU version. That will be more than enough horsepower to get you through the first few chapters but you will want to access GPUs before running larger models.

```
# For Windows users
pip install mxnet==1.6.0b20190926

# For Linux and macOS users
pip install mxnet==1.6.0b20191122
```

We also install the d2l package that encapsulates frequently used functions and classes in this book.

```
pip install d2l==0.11.1
```

Once they are installed, we now open the Jupyter notebook by running:

```
jupyter notebook
```

At this point, you can open http://localhost:8888 (it usually opens automatically) in your Web browser. Then we can run the code for each section of the book. Please always execute conda activate d2l to activate the runtime environment before running the code of the book or updating MXNet or the d2l package. To exit the environment, run conda deactivate.

[8] https://d2l.ai/d2l-en-0.7.1.zip

Upgrading to a New Version

Both this book and MXNet are keeping improving. Please check a new version from time to time.

1. The URL https://d2l.ai/d2l-en.zip always points to the latest contents.
2. Please upgrade the `d2l` package by `pip install d2l --upgrade`.
3. For the CPU version, MXNet can be upgraded by `pip install -U --pre mxnet`.

GPU Support

By default, MXNet is installed without GPU support to ensure that it will run on any computer (including most laptops). Part of this book requires or recommends running with GPU. If your computer has NVIDIA graphics cards and has installed CUDA[9], then you should install a GPU-enabled MXNet. If you have installed the CPU-only version, you may need to remove it first by running:

```
pip uninstall mxnet
```

Then we need to find the CUDA version you installed. You may check it through `nvcc --version` or `cat /usr/local/cuda/version.txt`. Assume that you have installed CUDA 10.1, then you can install MXNet with the following command:

```
# For Windows users
pip install mxnet-cu101==1.6.0b20190926

# For Linux and macOS users
pip install mxnet-cu101==1.6.0b20191122
```

Like the CPU version, the GPU-enabled MXNet can be upgraded by `pip install -U --pre mxnet-cu101`. You may change the last digits according to your CUDA version, e.g., `cu100` for CUDA 10.0 and `cu90` for CUDA 9.0. You can find all available MXNet versions via `pip search mxnet`.

Exercises

1. Download the code for the book and install the runtime environment.

[9] https://developer.nvidia.com/cuda-downloads

Notation

The notation used throughout this book is summarized below.

Numbers

- x: A scalar
- $\mathbf{x}$: A vector
- $\mathbf{X}$: A matrix
- X: A tensor
- $\mathbf{I}$: An identity matrix
- x_i, $[\mathbf{x}]_i$: The i^th element of vector $\mathbf{x}$
- x_{ij}, $[\mathbf{X}]_{ij}$: The element of matrix $\mathbf{X}$ at row i and column j

Set Theory

- $\mathcal{X}$: A set
- $\mathbb{Z}$: The set of integers
- $\mathbb{R}$: The set of real numbers
- $\mathbb{R}^n$: The set of n-dimensional vectors of real numbers
- $\mathbb{R}^{a\times b}$: The set of matrices of real numbers with a rows and b columns
- $\mathcal{A}\cup\mathcal{B}$: Union of sets $\mathcal{A}$ and $\mathcal{B}$
- $\mathcal{A}\cap\mathcal{B}$: Intersection of sets $\mathcal{A}$ and $\mathcal{B}$
- $\mathcal{A}\setminus\mathcal{B}$: Subtraction of set $\mathcal{B}$ from set $\mathcal{A}$

Functions and Operators

- $f(\cdot)$: A function
- $\log(\cdot)$: The natural logarithm
- $\exp(\cdot)$: The exponential function
- $\mathbf{1}_{\mathcal{X}}$: The indicator function
- $(\cdot)^\top$: Transpose of a vector or a matrix
- $\mathbf{X}^{-1}$: Inverse of matrix $\mathbf{X}$
- $\odot$: Hadamard (elementwise) product
- $|\mathcal{X}|$: Cardinality of set $\mathcal{X}$
- $\|\cdot\|_p$: ℓ_p norm
- $\|\cdot\|$: ℓ_2 norm
- $\langle \mathbf{x}, \mathbf{y} \rangle$: Dot product of vectors $\mathbf{x}$ and $\mathbf{y}$
- $\sum$: Series addition
- $\prod$: Series multiplication

Calculus

- $\frac{dy}{dx}$: Derivative of y with respect to x
- $\frac{\partial y}{\partial x}$: Partial derivative of y with respect to x
- $\nabla_{\mathbf{x}} y$: Gradient of y with respect to $\mathbf{x}$
- $\int_a^b f(x)\;dx$: Definite integral of f from a to b with respect to x
- $\int f(x)\;dx$: Indefinite integral of f with respect to x

Probability and Information Theory

- $P(\cdot)$: Probability distribution
- $z \sim P$: Random variable z has probability distribution P
- $P(X \mid Y)$: Conditional probability of $X \mid Y$
- $p(x)$: probability density function
- $E_x[f(x)]$: Expectation of f with respect to x
- $X \perp Y$: Random variables X and Y are independent
- $X \perp Y \mid Z$: Random variables X and Y are conditionally independent given random variable Z
- $\mathrm{Var}(X)$: Variance of random variable X

- σ_X: Standard deviation of random variable X
- $\mathrm{Cov}(X, Y)$: Covariance of random variables X and Y
- $\rho(X, Y)$: Correlation of random variables X and Y
- $H(X)$: Entropy of random variable X
- $D_{\mathrm{KL}}(P \| Q)$: KL-divergence of distributions P and Q

Complexity

- $\mathcal{O}$: Big O notation

1 | Introduction

Until recently, nearly every computer program that we interact with daily were coded by software developers from first principles. Say that we wanted to write an application to manage an e-commerce platform. After huddling around a whiteboard for a few hours to ponder the problem, we would come up with the broad strokes of a working solution that might probably look something like this: (i) users interact with the application through an interface running in a web browser or mobile application; (ii) our application interacts with a commercial-grade database engine to keep track of each user's state and maintain records of historical transactions; and (iii) at the heart of our application, the *business logic* (you might say, the *brains*) of our application spells out in methodical detail the appropriate action that our program should take in every conceivable circumstance.

To build the *brains* of our application, we'd have to step through every possible corner case that we anticipate encountering, devising appropriate rules. Each time a customer clicks to add an item to their shopping cart, we add an entry to the shopping cart database table, associating that user's ID with the requested product's ID. While few developers ever get it completely right the first time (it might take some test runs to work out the kinks), for the most part, we could write such a program from first principles and confidently launch it *before ever seeing a real customer*. Our ability to design automated systems from first principles that drive functioning products and systems, often in novel situations, is a remarkable cognitive feat. And when you are able to devise solutions that work 100% of the time, *you should not be using machine learning*.

Fortunately for the growing community of ML scientists, many tasks that we would like to automate do not bend so easily to human ingenuity. Imagine huddling around the whiteboard with the smartest minds you know, but this time you are tackling one of the following problems:

- Write a program that predicts tomorrow's weather given geographic information, satellite images, and a trailing window of past weather.
- Write a program that takes in a question, expressed in free-form text, and answers it correctly.
- Write a program that given an image can identify all the people it contains, drawing outlines around each.
- Write a program that presents users with products that they are likely to enjoy but unlikely, in the natural course of browsing, to encounter.

In each of these cases, even elite programmers are incapable of coding up solutions from scratch. The reasons for this can vary. Sometimes the program that we are looking for follows a pattern that changes over time, and we need our programs to adapt. In other cases, the relationship (say between pixels, and abstract categories) may be too complicated, requiring thousands or millions of computations that are beyond our conscious understanding (even if our eyes manage the task effortlessly). Machine learning (ML) is the study of powerful techniques that can *learn* from *expe-*

rience. As ML algorithm accumulates more experience, typically in the form of observational data or interactions with an environment, their performance improves. Contrast this with our deterministic e-commerce platform, which performs according to the same business logic, no matter how much experience accrues, until the developers themselves *learn* and decide that it is time to update the software. In this book, we will teach you the fundamentals of machine learning, and focus in particular on deep learning, a powerful set of techniques driving innovations in areas as diverse as computer vision, natural language processing, healthcare, and genomics.

1.1 A Motivating Example

Before we could begin writing, the authors of this book, like much of the work force, had to become caffeinated. We hopped in the car and started driving. Using an iPhone, Alex called out "Hey Siri", awakening the phone's voice recognition system. Then Mu commanded "directions to Blue Bottle coffee shop". The phone quickly displayed the transcription of his command. It also recognized that we were asking for directions and launched the Maps application to fulfill our request. Once launched, the Maps app identified a number of routes. Next to each route, the phone displayed a predicted transit time. While we fabricated this story for pedagogical convenience, it demonstrates that in the span of just a few seconds, our everyday interactions with a smart phone can engage several machine learning models.

Imagine just writing a program to respond to a *wake word* like "Alexa", "Okay, Google" or "Siri". Try coding it up in a room by yourself with nothing but a computer and a code editor, as illustrated in Fig. 1.1.1. How would you write such a program from first principles? Think about it... the problem is hard. Every second, the microphone will collect roughly 44,000 samples. Each sample is a measurement of the amplitude of the sound wave. What rule could map reliably from a snippet of raw audio to confident predictions `{yes, no}` on whether the snippet contains the wake word? If you are stuck, do not worry. We do not know how to write such a program from scratch either. That is why we use ML.

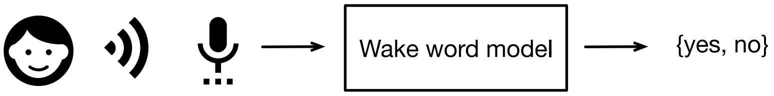


Fig. 1.1.1: Identify an awake word.

Here's the trick. Often, even when we do not know how to tell a computer explicitly how to map from inputs to outputs, we are nonetheless capable of performing the cognitive feat ourselves. In other words, even if you do not know *how to program a computer* to recognize the word "Alexa", you yourself *are able* to recognize the word "Alexa". Armed with this ability, we can collect a huge *dataset* containing examples of audio and label those that *do* and that *do not* contain the wake word. In the ML approach, we do not attempt to design a system *explicitly* to recognize wake words. Instead, we define a flexible program whose behavior is determined by a number of *parameters*. Then we use the dataset to determine the best possible set of parameters, those that improve the performance of our program with respect to some measure of performance on the task of interest.

You can think of the parameters as knobs that we can turn, manipulating the behavior of the program. Fixing the parameters, we call the program a *model*. The set of all distinct programs (input-output mappings) that we can produce just by manipulating the parameters is called a *family* of models. And the *meta-program* that uses our dataset to choose the parameters is called a *learning algorithm*.

Before we can go ahead and engage the learning algorithm, we have to define the problem precisely, pinning down the exact nature of the inputs and outputs, and choosing an appropriate model family. In this case, our model receives a snippet of audio as *input*, and it generates a selection among `{yes, no}` as *output*. If all goes according to plan the model's guesses will typically be correct as to whether (or not) the snippet contains the wake word.

If we choose the right family of models, then there should exist one setting of the knobs such that the model fires `yes` every time it hears the word "Alexa". Because the exact choice of the wake word is arbitrary, we will probably need a model family sufficiently rich that, via another setting of the knobs, it could fire `yes` only upon hearing the word "Apricot". We expect that the same model family should be suitable for *"Alexa" recognition* and *"Apricot" recognition* because they seem, intuitively, to be similar tasks. However, we might need a different family of models entirely if we want to deal with fundamentally different inputs or outputs, say if we wanted to map from images to captions, or from English sentences to Chinese sentences.

As you might guess, if we just set all of the knobs randomly, it is not likely that our model will recognize "Alexa", "Apricot", or any other English word. In deep learning, the *learning* is the process by which we discover the right setting of the knobs coercing the desired behavior from our model.

As shown in Fig. 1.1.2, the training process usually looks like this:

1. Start off with a randomly initialized model that cannot do anything useful.
2. Grab some of your labeled data (e.g., audio snippets and corresponding `{yes, no}` labels)
3. Tweak the knobs so the model sucks less with respect to those examples
4. Repeat until the model is awesome.

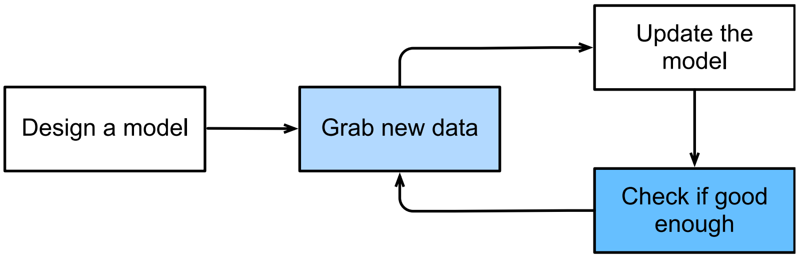


Fig. 1.1.2: A typical training process.

To summarize, rather than code up a wake word recognizer, we code up a program that can *learn* to recognize wake words, *if we present it with a large labeled dataset*. You can think of this act of determining a program's behavior by presenting it with a dataset as *programming with data*. We can "program" a cat detector by providing our machine learning system with many examples of cats and dogs, such as the images below:

cat	cat	dog	dog

This way the detector will eventually learn to emit a very large positive number if it is a cat, a very large negative number if it is a dog, and something closer to zero if it is not sure, and this barely scratches the surface of what ML can do.

Deep learning is just one among many popular methods for solving machine learning problems. Thus far, we have only talked about machine learning broadly and not deep learning. To see why deep learning is important, we should pause for a moment to highlight a couple crucial points.

First, the problems that we have discussed thus far—learning from raw audio signal, the raw pixel values of images, or mapping between sentences of arbitrary lengths and their counterparts in foreign languages—are problems where deep learning excels and where traditional ML methods faltered. Deep models are *deep* in precisely the sense that they learn many *layers* of computation. It turns out that these many-layered (or hierarchical) models are capable of addressing low-level perceptual data in a way that previous tools could not. In bygone days, the crucial part of applying ML to these problems consisted of coming up with manually-engineered ways of transforming the data into some form amenable to *shallow* models. One key advantage of deep learning is that it replaces not only the *shallow* models at the end of traditional learning pipelines, but also the labor-intensive process of feature engineering. Second, by replacing much of the *domain-specific preprocessing*, deep learning has eliminated many of the boundaries that previously separated computer vision, speech recognition, natural language processing, medical informatics, and other application areas, offering a unified set of tools for tackling diverse problems.

1.2 The Key Components: Data, Models, and Algorithms

In our *wake-word* example, we described a dataset consisting of audio snippets and binary labels gave a hand-wavy sense of how we might *train* a model to approximate a mapping from snippets to classifications. This sort of problem, where we try to predict a designated unknown *label* given known *inputs*, given a dataset consisting of examples, for which the labels are known is called *supervised learning*, and it is just one among many *kinds* of machine learning problems. In the next section, we will take a deep dive into the different ML problems. First, we'd like to shed more light on some core components that will follow us around, no matter what kind of ML problem we take on:

1. The *data* that we can learn from.
2. A *model* of how to transform the data.
3. A *loss* function that quantifies the *badness* of our model.
4. An *algorithm* to adjust the model's parameters to minimize the loss.

1.2.1 Data

It might go without saying that you cannot do data science without data. We could lose hundreds of pages pondering what precisely constitutes data, but for now we will err on the practical side and focus on the key properties to be concerned with. Generally we are concerned with a collection of *examples* (also called *data points*, *samples*, or *instances*). In order to work with data usefully, we typically need to come up with a suitable numerical representation. Each *example* typically consists of a collection of numerical attributes called *features*. In the supervised learning problems above, a special feature is designated as the prediction *target*, (sometimes called the *label* or *dependent variable*). The given features from which the model must make its predictions can then simply be called the *features*, (or often, the *inputs*, *covariates*, or *independent variables*).

If we were working with image data, each individual photograph might constitute an *example*, each represented by an ordered list of numerical values corresponding to the brightness of each pixel. A 200×200 color photograph would consist of $200 \times 200 \times 3 = 120000$ numerical values, corresponding to the brightness of the red, green, and blue channels for each spatial location. In a more traditional task, we might try to predict whether or not a patient will survive, given a standard set of features such as age, vital signs, diagnoses, etc.

When every example is characterized by the same number of numerical values, we say that the data consists of *fixed-length* vectors and we describe the (constant) length of the vectors as the *dimensionality* of the data. As you might imagine, fixed length can be a convenient property. If we wanted to train a model to recognize cancer in microscopy images, fixed-length inputs means we have one less thing to worry about.

However, not all data can easily be represented as fixed length vectors. While we might expect microscope images to come from standard equipment, we cannot expect images mined from the Internet to all show up with the same resolution or shape. For images, we might consider cropping them all to a standard size, but that strategy only gets us so far. We risk losing information in the cropped out portions. Moreover, text data resists fixed-length representations even more stubbornly. Consider the customer reviews left on e-commerce sites like Amazon, IMDB, or TripAdvisor. Some are short: "it stinks!". Others ramble for pages. One major advantage of deep learning over traditional methods is the comparative grace with which modern models can handle *varying-length* data.

Generally, the more data we have, the easier our job becomes. When we have more data, we can train more powerful models, and rely less heavily on pre-conceived assumptions. The regime change from (comparatively small) to big data is a major contributor to the success of modern deep learning. To drive the point home, many of the most exciting models in deep learning either do not work without large datasets. Some others work in the low-data regime, but no better than traditional approaches.

Finally it is not enough to have lots of data and to process it cleverly. We need the *right* data. If the data is full of mistakes, or if the chosen features are not predictive of the target quantity of interest, learning is going to fail. The situation is captured well by the cliché: *garbage in, garbage out*. Moreover, poor predictive performance is not the only potential consequence. In sensitive applications of machine learning, like predictive policing, resumé screening, and risk models used for lending, we must be especially alert to the consequences of garbage data. One common failure mode occurs in datasets where some groups of people are unrepresented in the training data. Imagine applying a skin cancer recognition system in the wild that had never seen black skin before. Failure can also occur when the data does not merely under-represent some groups, but reflects societal prejudices. For example if past hiring decisions are used to train a predictive model that will be used to screen resumes, then machine learning models could inadvertently capture and automate historical injustices. Note that this can all happen without the data scientist actively conspiring, or even being aware.

1.2.2 Models

Most machine learning involves *transforming* the data in some sense. We might want to build a system that ingests photos and predicts *smiley-ness*. Alternatively, we might want to ingest a set of sensor readings and predict how *normal* vs *anomalous* the readings are. By *model*, we denote the computational machinery for ingesting data of one type, and spitting out predictions of a possibly different type. In particular, we are interested in statistical models that can be estimated from data. While simple models are perfectly capable of addressing appropriately simple problems the problems that we focus on in this book stretch the limits of classical methods. Deep learning is differentiated from classical approaches principally by the set of powerful models that it focuses on. These models consist of many successive transformations of the data that are chained together top to bottom, thus the name *deep learning*. On our way to discussing deep neural networks, we will discuss some more traditional methods.

1.2.3 Objective functions

Earlier, we introduced machine learning as "learning from experience". By *learning* here, we mean *improving* at some task over time. But who is to say what constitutes an improvement? You might imagine that we could propose to update our model, and some people might disagree on whether the proposed update constituted an improvement or a decline.

In order to develop a formal mathematical system of learning machines, we need to have formal measures of how good (or bad) our models are. In machine learning, and optimization more generally, we call these objective functions. By convention, we usually define objective functions so that *lower* is *better*. This is merely a convention. You can take any function f for which higher is better, and turn it into a new function f' that is qualitatively identical but for which lower is better by setting $f' = -f$. Because lower is better, these functions are sometimes called *loss functions* or *cost functions*.

When trying to predict numerical values, the most common objective function is squared error $(y-\hat{y})^2$. For classification, the most common objective is to minimize error rate, i.e., the fraction of instances on which our predictions disagree with the ground truth. Some objectives (like squared error) are easy to optimize. Others (like error rate) are difficult to optimize directly, owing to non-differentiability or other complications. In these cases, it is common to optimize a *surrogate objective*.

Typically, the loss function is defined with respect to the model's parameters and depends upon the dataset. The best values of our model's parameters are learned by minimizing the loss incurred on a *training set* consisting of some number of *examples* collected for training. However, doing well on the training data does not guarantee that we will do well on (unseen) test data. So we will typically want to split the available data into two partitions: the training data (for fitting model parameters) and the test data (which is held out for evaluation), reporting the following two quantities:

- **Training Error:** The error on that data on which the model was trained. You could think of this as being like a student's scores on practice exams used to prepare for some real exam. Even if the results are encouraging, that does not guarantee success on the final exam.
- **Test Error:** This is the error incurred on an unseen test set. This can deviate significantly from the training error. When a model performs well on the training data but fails to generalize to unseen data, we say that it is *overfitting*. In real-life terms, this is like flunking the real exam despite doing well on practice exams.

1.2.4 Optimization algorithms

Once we have got some data source and representation, a model, and a well-defined objective function, we need an algorithm capable of searching for the best possible parameters for minimizing the loss function. The most popular optimization algorithms for neural networks follow an approach called gradient descent. In short, at each step, they check to see, for each parameter, which way the training set loss would move if you perturbed that parameter just a small amount. They then update the parameter in the direction that reduces the loss.

1.3 Kinds of Machine Learning

In the following sections, we discuss a few *kinds* of machine learning problems in greater detail. We begin with a list of *objectives*, i.e., a list of things that we would like machine learning to do. Note that the objectives are complemented with a set of techniques of *how* to accomplish them, including types of data, models, training techniques, etc. The list below is just a sampling of the problems ML can tackle to motivate the reader and provide us with some common language for when we talk about more problems throughout the book.

1.3.1 Supervised learning

Supervised learning addresses the task of predicting *targets* given *inputs*. The targets, which we often call *labels*, are generally denoted by *y*. The input data, also called the *features* or covariates, are typically denoted **x**. Each (input, target) pair is called an *examples* or an *instances*. Some times, when the context is clear, we may use the term examples, to refer to a collection of inputs, even when the corresponding targets are unknown. We denote any particular instance with a subscript, typically i, for instance $(\mathbf{x}_i, y_i)$. A dataset is a collection of n instances $\{\mathbf{x}_i, y_i\}_{i=1}^n$. Our goal is to produce a model f_θ that maps any input $\mathbf{x}_i$ to a prediction $f_\theta(\mathbf{x}_i)$.

To ground this description in a concrete example, if we were working in healthcare, then we might want to predict whether or not a patient would have a heart attack. This observation, *heart attack* or *no heart attack*, would be our label y. The input data $\mathbf{x}$ might be vital signs such as heart rate, diastolic and systolic blood pressure, etc.

The supervision comes into play because for choosing the parameters θ, we (the supervisors) provide the model with a dataset consisting of *labeled examples* $(\mathbf{x}_i, y_i)$, where each example $\mathbf{x}_i$ is matched with the correct label.

In probabilistic terms, we typically are interested in estimating the conditional probability $P(y|x)$. While it is just one among several paradigms within machine learning, supervised learning accounts for the majority of successful applications of machine learning in industry. Partly, that is because many important tasks can be described crisply as estimating the probability of something unknown given a particular set of available data:

- Predict cancer vs not cancer, given a CT image.
- Predict the correct translation in French, given a sentence in English.
- Predict the price of a stock next month based on this month's financial reporting data.

Even with the simple description "predict targets from inputs" supervised learning can take a great many forms and require a great many modeling decisions, depending on (among other considerations) the type, size, and the number of inputs and outputs. For example, we use different models to process sequences (like strings of text or time series data) and for processing fixed-length vector representations. We will visit many of these problems in depth throughout the first 9 parts of this book.

Informally, the learning process looks something like this: Grab a big collection of examples for which the covariates are known and select from them a random subset, acquiring the ground truth labels for each. Sometimes these labels might be available data that has already been collected (e.g., did a patient die within the following year?) and other times we might need to employ human annotators to label the data, (e.g., assigning images to categories).

Together, these inputs and corresponding labels comprise the training set. We feed the training dataset into a supervised learning algorithm, a function that takes as input a dataset and outputs another function, *the learned model*. Finally, we can feed previously unseen inputs to the learned model, using its outputs as predictions of the corresponding label. The full process in drawn in Fig. 1.3.1.

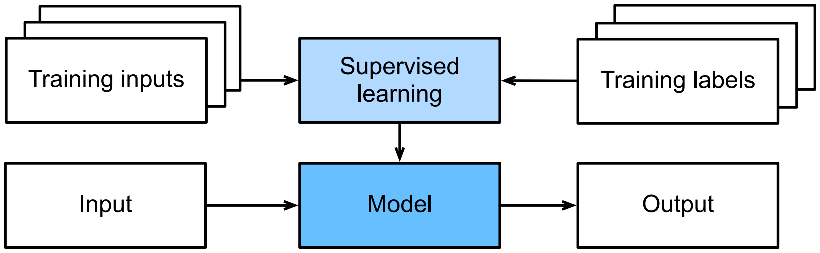


Fig. 1.3.1: Supervised learning.

Regression

Perhaps the simplest supervised learning task to wrap your head around is *regression*. Consider, for example a set of data harvested from a database of home sales. We might construct a table, where each row corresponds to a different house, and each column corresponds to some relevant attribute, such as the square footage of a house, the number of bedrooms, the number of bathrooms, and the number of minutes (walking) to the center of town. In this dataset each *example* would be a specific house, and the corresponding *feature vector* would be one row in the table.

If you live in New York or San Francisco, and you are not the CEO of Amazon, Google, Microsoft, or Facebook, the (sq. footage, no. of bedrooms, no. of bathrooms, walking distance) feature vector for your home might look something like: $[100, 0, .5, 60]$. However, if you live in Pittsburgh, it might look more like $[3000, 4, 3, 10]$. Feature vectors like this are essential for most classic machine learning algorithms. We will continue to denote the feature vector correspond to any example i as $\mathbf{x}_i$ and we can compactly refer to the full table containing all of the feature vectors as X.

What makes a problem a *regression* is actually the outputs. Say that you are in the market for a new home. You might want to estimate the fair market value of a house, given some features like these. The target value, the price of sale, is a *real number*. If you remember the formal definition of the reals you might be scratching your head now. Homes probably never sell for fractions of a cent, let alone prices expressed as irrational numbers. In cases like this, when the target is actually discrete, but where the rounding takes place on a sufficiently fine scale, we will abuse language just a bit cn continue to describe our outputs and targets as real-valued numbers.

We denote any individual target y_i (corresponding to example $\mathbf{x_i}$) and the set of all targets $\mathbf{y}$ (corresponding to all examples X). When our targets take on arbitrary values in some range, we call this a regression problem. Our goal is to produce a model whose predictions closely approximate the actual target values. We denote the predicted target for any instance $\hat{y}_i$. Do not worry if the notation is bogging you down. We will unpack it more thoroughly in the subsequent chapters.

Lots of practical problems are well-described regression problems. Predicting the rating that a user will assign to a movie can be thought of as a regression problem and if you designed a great algorithm to accomplish this feat in 2009, you might have won the 1-million-dollar Netflix prize[12]. Predicting the length of stay for patients in the hospital is also a regression problem. A good rule of thumb is that any *How much?* or *How many?* problem should suggest regression.

- "How many hours will this surgery take?": *regression*
- "How many dogs are in this photo?": *regression*.

However, if you can easily pose your problem as "Is this a _ ?", then it is likely, classification, a different kind of supervised problem that we will cover next. Even if you have never worked with machine learning before, you have probably worked through a regression problem informally. Imagine, for example, that you had your drains repaired and that your contractor spent $x_1 = 3$ hours removing gunk from your sewage pipes. Then she sent you a bill of $y_1 = \$350$. Now imagine that your friend hired the same contractor for $x_2 = 2$ hours and that she received a bill of $y_2 = \$250$. If someone then asked you how much to expect on their upcoming gunk-removal invoice you might make some reasonable assumptions, such as more hours worked costs more dollars. You might also assume that there is some base charge and that the contractor then charges per hour. If these assumptions held true, then given these two data points, you could already identify the contractor's pricing structure: $100 per hour plus $50 to show up at your house. If you followed that much then you already understand the high-level idea behind linear regression (and you just implicitly designed a linear model with a bias term).

In this case, we could produce the parameters that exactly matched the contractor's prices. Sometimes that is not possible, e.g., if some of the variance owes to some factors besides your two features. In these cases, we will try to learn models that minimize the distance between our predictions and the observed values. In most of our chapters, we will focus on one of two very common losses, the L1 loss[13] where

$$l(y, y') = \sum_i |y_i - y'_i| \tag{1.3.1}$$

and the least mean squares loss, or L2 loss[14], where

$$l(y, y') = \sum_i (y_i - y'_i)^2. \tag{1.3.2}$$

As we will see later, the L_2 loss corresponds to the assumption that our data was corrupted by Gaussian noise, whereas the L_1 loss corresponds to an assumption of noise from a Laplace distribution.

[12] https://en.wikipedia.org/wiki/Netflix_Prize
[13] http://mxnet.incubator.apache.org/api/python/gluon/loss.html#mxnet.gluon.loss.L1Loss
[14] http://mxnet.incubator.apache.org/api/python/gluon/loss.html#mxnet.gluon.loss.L2Loss

Classification

While regression models are great for addressing *how many?* questions, lots of problems do not bend comfortably to this template. For example, a bank wants to add check scanning to their mobile app. This would involve the customer snapping a photo of a check with their smart phone's camera and the machine learning model would need to be able to automatically understand text seen in the image. It would also need to understand hand-written text to be even more robust. This kind of system is referred to as optical character recognition (OCR), and the kind of problem it addresses is called *classification*. It is treated with a different set of algorithms than those used for regression (although many techniques will carry over).

In classification, we want our model to look at a feature vector, e.g., the pixel values in an image, and then predict which category (formally called *classes*), among some (discrete) set of options, an example belongs. For hand-written digits, we might have 10 classes, corresponding to the digits 0 through 9. The simplest form of classification is when there are only two classes, a problem which we call binary classification. For example, our dataset X could consist of images of animals and our *labels* Y might be the classes $\{\text{cat}, \text{dog}\}$. While in regression, we sought a *regressor* to output a real value $\hat{y}$, in classification, we seek a *classifier*, whose output $\hat{y}$ is the predicted class assignment.

For reasons that we will get into as the book gets more technical, it can be hard to optimize a model that can only output a hard categorical assignment, e.g., either *cat* or *dog*. In these cases, it is usually much easier to instead express our model in the language of probabilities. Given an example x, our model assigns a probability $\hat{y}_k$ to each label k. Because these are probabilities, they need to be positive numbers and add up to 1 and thus we only need $K-1$ numbers to assign probabilities of K categories. This is easy to see for binary classification. If there is a 0.6 (60%) probability that an unfair coin comes up heads, then there is a 0.4 (40%) probability that it comes up tails. Returning to our animal classification example, a classifier might see an image and output the probability that the image is a cat $P(y = \text{cat} \mid x) = 0.9$. We can interpret this number by saying that the classifier is 90% sure that the image depicts a cat. The magnitude of the probability for the predicted class conveys one notion of uncertainty. It is not the only notion of uncertainty and we will discuss others in more advanced chapters.

When we have more than two possible classes, we call the problem *multiclass classification*. Common examples include hand-written character recognition `[0, 1, 2, 3 ... 9, a, b, c, ...]`. While we attacked regression problems by trying to minimize the L1 or L2 loss functions, the common loss function for classification problems is called cross-entropy. In MXNet Gluon, the corresponding loss function can be found here[15].

Note that the most likely class is not necessarily the one that you are going to use for your decision. Assume that you find this beautiful mushroom in your backyard as shown in Fig. 1.3.2.

[15] https://mxnet.incubator.apache.org/api/python/gluon/loss.html#mxnet.gluon.loss.SoftmaxCrossEntropyLoss

Fig. 1.3.2: Death cap—do not eat!

Now, assume that you built a classifier and trained it to predict if a mushroom is poisonous based on a photograph. Say our poison-detection classifier outputs $P(y = \text{deathcap}|\text{image}) = 0.2$. In other words, the classifier is 80% sure that our mushroom *is not* a death cap. Still, you'd have to be a fool to eat it. That is because the certain benefit of a delicious dinner is not worth a 20% risk of dying from it. In other words, the effect of the *uncertain risk* outweighs the benefit by far. We can look at this more formally. Basically, we need to compute the expected risk that we incur, i.e., we need to multiply the probability of the outcome with the benefit (or harm) associated with it:

$$L(\text{action}|x) = E_{y \sim p(y|x)}[\text{loss}(\text{action}, y)]. \tag{1.3.3}$$

Hence, the loss L incurred by eating the mushroom is $L(a = \text{eat}|x) = 0.2 * \infty + 0.8 * 0 = \infty$, whereas the cost of discarding it is $L(a = \text{discard}|x) = 0.2 * 0 + 0.8 * 1 = 0.8$.

Our caution was justified: as any mycologist would tell us, the above mushroom actually *is* a death cap. Classification can get much more complicated than just binary, multiclass, or even multilabel classification. For instance, there are some variants of classification for addressing hierarchies. Hierarchies assume that there exist some relationships among the many classes. So not all errors are equal—if we must err, we would prefer to misclassify to a related class rather than to a distant class. Usually, this is referred to as *hierarchical classification*. One early example is due to Linnaeus[16], who organized the animals in a hierarchy.

In the case of animal classification, it might not be so bad to mistake a poodle for a schnauzer, but our model would pay a huge penalty if it confused a poodle for a dinosaur. Which hierarchy is relevant might depend on how you plan to use the model. For example, rattle snakes and garter snakes might be close on the phylogenetic tree, but mistaking a rattler for a garter could be deadly.

[16] https://en.wikipedia.org/wiki/Carl_Linnaeus

Tagging

Some classification problems do not fit neatly into the binary or multiclass classification setups. For example, we could train a normal binary classifier to distinguish cats from dogs. Given the current state of computer vision, we can do this easily, with off-the-shelf tools. Nonetheless, no matter how accurate our model gets, we might find ourselves in trouble when the classifier encounters an image of the Town Musicians of Bremen.

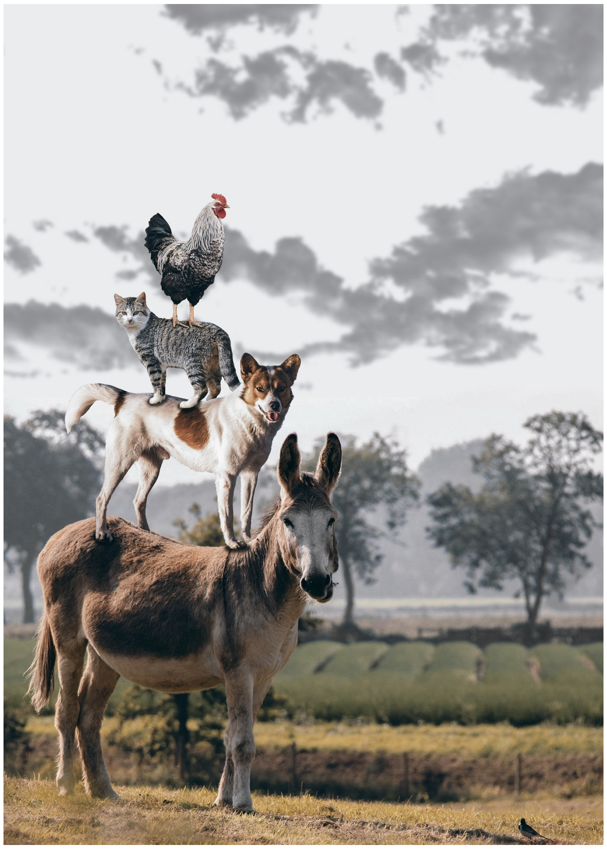


Fig. 1.3.3: A cat, a roster, a dog and a donkey

As you can see, there is a cat in the picture, and a rooster, a dog, a donkey and a bird, with some trees in the background. Depending on what we want to do with our model ultimately, treating this as a binary classification problem might not make a lot of sense. Instead, we might want to give the model the option of saying the image depicts a cat *and* a dog *and* a donkey *and* a rooster *and* a bird.

The problem of learning to predict classes that are *not mutually exclusive* is called multi-label classification. Auto-tagging problems are typically best described as multi-label classification problems. Think of the tags people might apply to posts on a tech blog, e.g., “machine learning”, “technology”, “gadgets”, “programming languages”, “linux”, “cloud computing”, “AWS”. A typical article might have 5-10 tags applied because these concepts are correlated. Posts about “cloud computing” are likely to mention “AWS” and posts about “machine learning” could also deal with “programming languages”.

We also have to deal with this kind of problem when dealing with the biomedical literature, where correctly tagging articles is important because it allows researchers to do exhaustive reviews of the literature. At the National Library of Medicine, a number of professional annotators go over each article that gets indexed in PubMed to associate it with the relevant terms from MeSH, a collection of roughly 28k tags. This is a time-consuming process and the annotators typically have a one year lag between archiving and tagging. Machine learning can be used here to provide provisional tags until each article can have a proper manual review. Indeed, for several years, the BioASQ organization has hosted a competition[17] to do precisely this.

Search and ranking

Sometimes we do not just want to assign each example to a bucket or to a real value. In the field of information retrieval, we want to impose a ranking on a set of items. Take web search for example, the goal is less to determine whether a particular page is relevant for a query, but rather, which one of the plethora of search results is *most relevant* for a particular user. We really care about the ordering of the relevant search results and our learning algorithm needs to produce ordered subsets of elements from a larger set. In other words, if we are asked to produce the first 5 letters from the alphabet, there is a difference between returning `A B C D E` and `C A B E D`. Even if the result set is the same, the ordering within the set matters.

One possible solution to this problem is to first assign to every element in the set a corresponding relevance score and then to retrieve the top-rated elements. PageRank[18], the original secret sauce behind the Google search engine was an early example of such a scoring system but it was peculiar in that it did not depend on the actual query. Here they relied on a simple relevance filter to identify the set of relevant items and then on PageRank to order those results that contained the query term. Nowadays, search engines use machine learning and behavioral models to obtain query-dependent relevance scores. There are entire academic conferences devoted to this subject.

Recommender systems

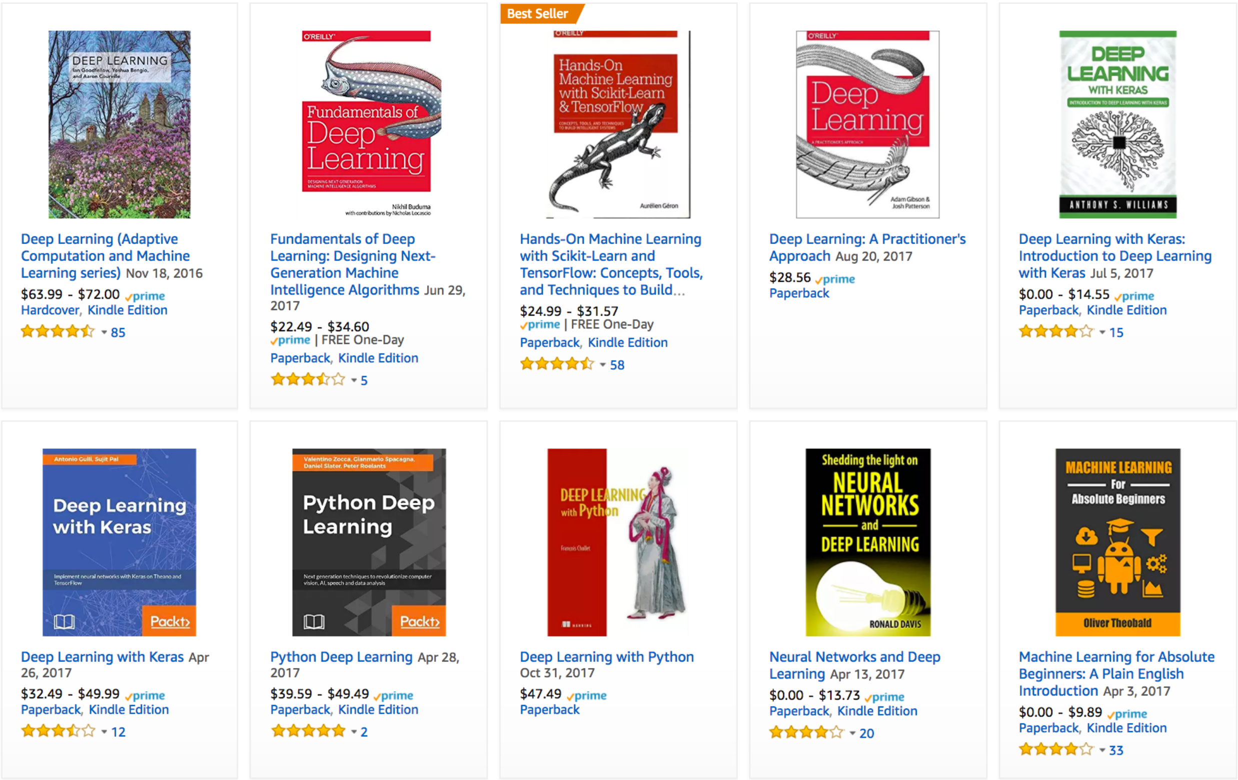
Recommender systems are another problem setting that is related to search and ranking. The problems are similar insofar as the goal is to display a set of relevant items to the user. The main difference is the emphasis on *personalization* to specific users in the context of recommender systems. For instance, for movie recommendations, the results page for a SciFi fan and the results page for a connoisseur of Peter Sellers comedies might differ significantly. Similar problems pop up in other recommendation settings, e.g., for retail products, music, or news recommendation.

In some cases, customers provide explicit feedback communicating how much they liked a particular product (e.g., the product ratings and reviews on Amazon, IMDB, GoodReads, etc.). In some other cases, they provide implicit feedback, e.g., by skipping titles on a playlist, which might indicate dissatisfaction but might just indicate that the song was inappropriate in context. In the simplest formulations, these systems are trained to estimate some score y_{ij}, such as an estimated rating or the probability of purchase, given a user u_i and product p_j.

Given such a model, then for any given user, we could retrieve the set of objects with the largest scores y_{ij}, which are could then be recommended to the customer. Production systems are considerably more advanced and take detailed user activity and item characteristics into account when computing such scores. Fig. 1.3.4 is an example of deep learning books recommended by Amazon based on personalization algorithms tuned to capture the author's preferences.

[17] http://bioasq.org/

[18] https://en.wikipedia.org/wiki/PageRank



Best Seller

Deep Learning (Adaptive Computation and Machine Learning series) Nov 18, 2016
$63.99 - $72.00 prime
Hardcover, Kindle Edition
85

Fundamentals of Deep Learning: Designing Next-Generation Machine Intelligence Algorithms Jun 29, 2017
$22.49 - $34.60
prime | FREE One-Day
Paperback, Kindle Edition
5

Hands-On Machine Learning with Scikit-Learn and TensorFlow: Concepts, Tools, and Techniques to Build…
$24.99 - $31.57
prime | FREE One-Day
Paperback, Kindle Edition
58

Deep Learning: A Practitioner's Approach Aug 20, 2017
$28.56 prime
Paperback

Deep Learning with Keras: Introduction to Deep Learning with Keras Jul 5, 2017
$0.00 - $14.55 prime
Paperback, Kindle Edition
15

Deep Learning with Keras Apr 26, 2017
$32.49 - $49.99 prime
Paperback, Kindle Edition
12

Python Deep Learning Apr 28, 2017
$39.59 - $49.49 prime
Paperback, Kindle Edition
2

Deep Learning with Python Oct 31, 2017
$47.49 prime
Paperback

Neural Networks and Deep Learning Apr 13, 2017
$0.00 - $13.73 prime
Paperback, Kindle Edition
20

Machine Learning for Absolute Beginners: A Plain English Introduction Apr 3, 2017
$0.00 - $9.89 prime
Paperback, Kindle Edition
33

Fig. 1.3.4: Deep learning books recommended by Amazon.

Despite their tremendous economic value, recommendation systems naively built on top of predictive models suffer some serious conceptual flaws. To start, we only observe *censored feedback*. Users preferentially rate movies that they feel strongly about: you might notice that items receive many 5 and 1 star ratings but that there are conspicuously few 3-star ratings. Moreover, current purchase habits are often a result of the recommendation algorithm currently in place, but learning algorithms do not always take this detail into account. Thus it is possible for feedback loops to form where a recommender system preferentially pushes an item that is then taken to be better (due to greater purchases) and in turn is recommended even more frequently. Many of these problems about how to deal with censoring, incentives, and feedback loops, are important open research questions.

Sequence Learning

So far, we have looked at problems where we have some fixed number of inputs and produce a fixed number of outputs. Before we considered predicting home prices from a fixed set of features: square footage, number of bedrooms, number of bathrooms, walking time to downtown. We also discussed mapping from an image (of fixed dimension) to the predicted probabilities that it belongs to each of a fixed number of classes, or taking a user ID and a product ID, and predicting a star rating. In these cases, once we feed our fixed-length input into the model to generate an output, the model immediately forgets what it just saw.

This might be fine if our inputs truly all have the same dimensions and if successive inputs truly have nothing to do with each other. But how would we deal with video snippets? In this case, each snippet might consist of a different number of frames. And our guess of what is going on in each frame might be much stronger if we take into account the previous or succeeding frames.

Same goes for language. One popular deep learning problem is machine translation: the task of ingesting sentences in some source language and predicting their translation in another language.

These problems also occur in medicine. We might want a model to monitor patients in the intensive care unit and to fire off alerts if their risk of death in the next 24 hours exceeds some threshold. We definitely would not want this model to throw away everything it knows about the patient history each hour and just make its predictions based on the most recent measurements.

These problems are among the most exciting applications of machine learning and they are instances of *sequence learning*. They require a model to either ingest sequences of inputs or to emit sequences of outputs (or both!). These latter problems are sometimes referred to as `seq2seq` problems. Language translation is a `seq2seq` problem. Transcribing text from spoken speech is also a `seq2seq` problem. While it is impossible to consider all types of sequence transformations, a number of special cases are worth mentioning:

Tagging and Parsing. This involves annotating a text sequence with attributes. In other words, the number of inputs and outputs is essentially the same. For instance, we might want to know where the verbs and subjects are. Alternatively, we might want to know which words are the named entities. In general, the goal is to decompose and annotate text based on structural and grammatical assumptions to get some annotation. This sounds more complex than it actually is. Below is a very simple example of annotating a sentence with tags indicating which words refer to named entities.

```
Tom has dinner in Washington with Sally.
Ent  -    -    -     Ent      -    Ent
```

Automatic Speech Recognition. With speech recognition, the input sequence x is an audio recording of a speaker (shown in Fig. 1.3.5), and the output y is the textual transcript of what the speaker said. The challenge is that there are many more audio frames (sound is typically sampled at 8kHz or 16kHz) than text, i.e., there is no 1:1 correspondence between audio and text, since thousands of samples correspond to a single spoken word. These are `seq2seq` problems where the output is much shorter than the input.

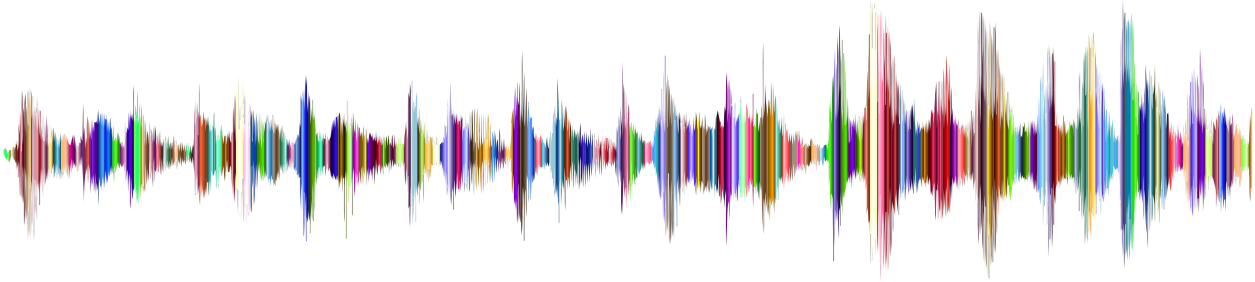


Fig. 1.3.5: `-D-e-e-p- L-ea-r-ni-ng-`

Text to Speech. Text-to-Speech (TTS) is the inverse of speech recognition. In other words, the input x is text and the output y is an audio file. In this case, the output is *much longer* than the input. While it is easy for *humans* to recognize a bad audio file, this is not quite so trivial for computers.

Machine Translation. Unlike the case of speech recognition, where corresponding inputs and outputs occur in the same order (after alignment), in machine translation, order inversion can be vital. In other words, while we are still converting one sequence into another, neither the number of inputs and outputs nor the order of corresponding data points are assumed to be the same. Consider the following illustrative example of the peculiar tendency of Germans to place the verbs at the end of sentences.

```
German:           Haben Sie sich schon dieses grossartige Lehrwerk angeschaut?
English:          Did you already check out this excellent tutorial?
Wrong alignment:  Did you yourself already this excellent tutorial looked-at?
```

Many related problems pop up in other learning tasks. For instance, determining the order in which a user reads a Webpage is a two-dimensional layout analysis problem. Dialogue problems exhibit all kinds of additional complications, where determining what to say next requires taking into account real-world knowledge and the prior state of the conversation across long temporal distances. This is an active area of research.

1.3.2 Unsupervised learning

All the examples so far were related to *Supervised Learning*, i.e., situations where we feed the model a giant dataset containing both the features and corresponding target values. You could think of the supervised learner as having an extremely specialized job and an extremely anal boss. The boss stands over your shoulder and tells you exactly what to do in every situation until you learn to map from situations to actions. Working for such a boss sounds pretty lame. On the other hand, it is easy to please this boss. You just recognize the pattern as quickly as possible and imitate their actions.

In a completely opposite way, it could be frustrating to work for a boss who has no idea what they want you to do. However, if you plan to be a data scientist, you'd better get used to it. The boss might just hand you a giant dump of data and tell you to *do some data science with it!* This sounds vague because it is. We call this class of problems *unsupervised learning*, and the type and number of questions we could ask is limited only by our creativity. We will address a number of unsupervised learning techniques in later chapters. To whet your appetite for now, we describe a few of the questions you might ask:

- Can we find a small number of prototypes that accurately summarize the data? Given a set of photos, can we group them into landscape photos, pictures of dogs, babies, cats, mountain peaks, etc.? Likewise, given a collection of users' browsing activity, can we group them into users with similar behavior? This problem is typically known as *clustering*.
- Can we find a small number of parameters that accurately capture the relevant properties of the data? The trajectories of a ball are quite well described by velocity, diameter, and mass of the ball. Tailors have developed a small number of parameters that describe human body shape fairly accurately for the purpose of fitting clothes. These problems are referred to as *subspace estimation* problems. If the dependence is linear, it is called *principal component analysis*.
- Is there a representation of (arbitrarily structured) objects in Euclidean space (i.e., the space of vectors in $\mathbb{R}^n$) such that symbolic properties can be well matched? This is called *representation learning* and it is used to describe entities and their relations, such as Rome $-$ Italy $+$ France $=$ Paris.
- Is there a description of the root causes of much of the data that we observe? For instance, if we have demographic data about house prices, pollution, crime, location, education, salaries, etc., can we discover how they are related simply based on empirical data? The fields concerned with *causality* and *probabilistic graphical models* address this problem.

- Another important and exciting recent development in unsupervised learning is the advent of *generative adversarial networks*. These give us a procedural way to synthesize data, even complicated structured data like images and audio. The underlying statistical mechanisms are tests to check whether real and fake data are the same. We will devote a few notebooks to them.

1.3.3 Interacting with an Environment

So far, we have not discussed where data actually comes from, or what actually *happens* when a machine learning model generates an output. That is because supervised learning and unsupervised learning do not address these issues in a very sophisticated way. In either case, we grab a big pile of data up front, then set our pattern recognition machines in motion without ever interacting with the environment again. Because all of the learning takes place after the algorithm is disconnected from the environment, this is sometimes called *offline learning*. For supervised learning, the process looks like Fig. 1.3.6.

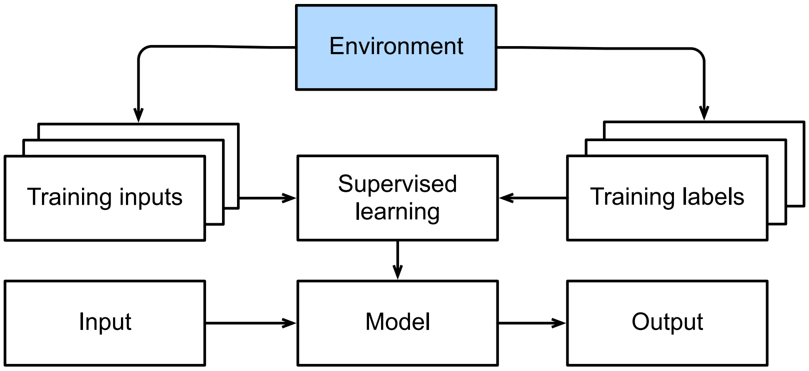


Fig. 1.3.6: Collect data for supervised learning from an environment.

This simplicity of offline learning has its charms. The upside is we can worry about pattern recognition in isolation, without any distraction from these other problems. But the downside is that the problem formulation is quite limiting. If you are more ambitious, or if you grew up reading Asimov's Robot Series, then you might imagine artificially intelligent bots capable not only of making predictions, but of taking actions in the world. We want to think about intelligent *agents*, not just predictive *models*. That means we need to think about choosing *actions*, not just making *predictions*. Moreover, unlike predictions, actions actually impact the environment. If we want to train an intelligent agent, we must account for the way its actions might impact the future observations of the agent.

Considering the interaction with an environment opens a whole set of new modeling questions. Does the environment:

- Remember what we did previously?
- Want to help us, e.g., a user reading text into a speech recognizer?
- Want to beat us, i.e., an adversarial setting like spam filtering (against spammers) or playing a game (vs an opponent)?
- Not care (as in many cases)?
- Have shifting dynamics (does future data always resemble the past or do the patterns change over time, either naturally or in response to our automated tools)?

This last question raises the problem of *distribution shift*, (when training and test data are different). It is a problem that most of us have experienced when taking exams written by a lecturer, while the homeworks were composed by her TAs. We will briefly describe reinforcement learning and adversarial learning, two settings that explicitly consider interaction with an environment.

1.3.4 Reinforcement learning

If you are interested in using machine learning to develop an agent that interacts with an environment and takes actions, then you are probably going to wind up focusing on *reinforcement learning* (RL). This might include applications to robotics, to dialogue systems, and even to developing AI for video games. *Deep reinforcement learning* (DRL), which applies deep neural networks to RL problems, has surged in popularity. The breakthrough deep Q-network that beat humans at Atari games using only the visual input[19], and the AlphaGo program that dethroned the world champion at the board game Go[20] are two prominent examples.

Reinforcement learning gives a very general statement of a problem, in which an agent interacts with an environment over a series of *timesteps*. At each timestep t, the agent receives some observation o_t from the environment and must choose an action a_t that is subsequently transmitted back to the environment via some mechanism (sometimes called an actuator). Finally, the agent receives a reward r_t from the environment. The agent then receives a subsequent observation, and chooses a subsequent action, and so on. The behavior of an RL agent is governed by a *policy*. In short, a *policy* is just a function that maps from observations (of the environment) to actions. The goal of reinforcement learning is to produce a good policy.

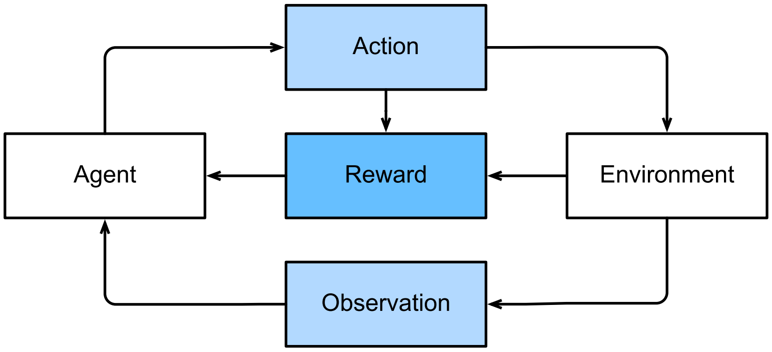


Fig. 1.3.7: The interaction between reinforcement learning and an environment.

It is hard to overstate the generality of the RL framework. For example, we can cast any supervised learning problem as an RL problem. Say we had a classification problem. We could create an RL agent with one *action* corresponding to each class. We could then create an environment which gave a reward that was exactly equal to the loss function from the original supervised problem.

That being said, RL can also address many problems that supervised learning cannot. For example, in supervised learning we always expect that the training input comes associated with the correct label. But in RL, we do not assume that for each observation, the environment tells us the optimal action. In general, we just get some reward. Moreover, the environment may not even tell us which actions led to the reward.

Consider for example the game of chess. The only real reward signal comes at the end of the game when we either win, which we might assign a reward of 1, or when we lose, which we could

[19] https://www.wired.com/2015/02/google-ai-plays-atari-like-pros/
[20] https://www.wired.com/2017/05/googles-alphago-trounces-humans-also-gives-boost/

assign a reward of -1. So reinforcement learners must deal with the *credit assignment problem*: determining which actions to credit or blame for an outcome. The same goes for an employee who gets a promotion on October 11. That promotion likely reflects a large number of well-chosen actions over the previous year. Getting more promotions in the future requires figuring out what actions along the way led to the promotion.

Reinforcement learners may also have to deal with the problem of partial observability. That is, the current observation might not tell you everything about your current state. Say a cleaning robot found itself trapped in one of many identical closets in a house. Inferring the precise location (and thus state) of the robot might require considering its previous observations before entering the closet.

Finally, at any given point, reinforcement learners might know of one good policy, but there might be many other better policies that the agent has never tried. The reinforcement learner must constantly choose whether to *exploit* the best currently-known strategy as a policy, or to *explore* the space of strategies, potentially giving up some short-run reward in exchange for knowledge.

MDPs, bandits, and friends

The general reinforcement learning problem is a very general setting. Actions affect subsequent observations. Rewards are only observed corresponding to the chosen actions. The environment may be either fully or partially observed. Accounting for all this complexity at once may ask too much of researchers. Moreover, not every practical problem exhibits all this complexity. As a result, researchers have studied a number of *special cases* of reinforcement learning problems.

When the environment is fully observed, we call the RL problem a *Markov Decision Process* (MDP). When the state does not depend on the previous actions, we call the problem a *contextual bandit problem*. When there is no state, just a set of available actions with initially unknown rewards, this problem is the classic *multi-armed bandit problem*.

1.4 Roots

Although many deep learning methods are recent inventions, humans have held the desire to analyze data and to predict future outcomes for centuries. In fact, much of natural science has its roots in this. For instance, the Bernoulli distribution is named after Jacob Bernoulli (1655-1705)[21], and the Gaussian distribution was discovered by Carl Friedrich Gauss (1777-1855)[22]. He invented for instance the least mean squares algorithm, which is still used today for countless problems from insurance calculations to medical diagnostics. These tools gave rise to an experimental approach in the natural sciences—for instance, Ohm's law relating current and voltage in a resistor is perfectly described by a linear model.

Even in the middle ages, mathematicians had a keen intuition of estimates. For instance, the geometry book of Jacob Köbel (1460-1533)[23] illustrates averaging the length of 16 adult men's feet to obtain the average foot length.

[21] https://en.wikipedia.org/wiki/Jacob_Bernoulli

[22] https://en.wikipedia.org/wiki/Carl_Friedrich_Gauss

[23] https://www.maa.org/press/periodicals/convergence/mathematical-treasures-jacob-kobels-geometry

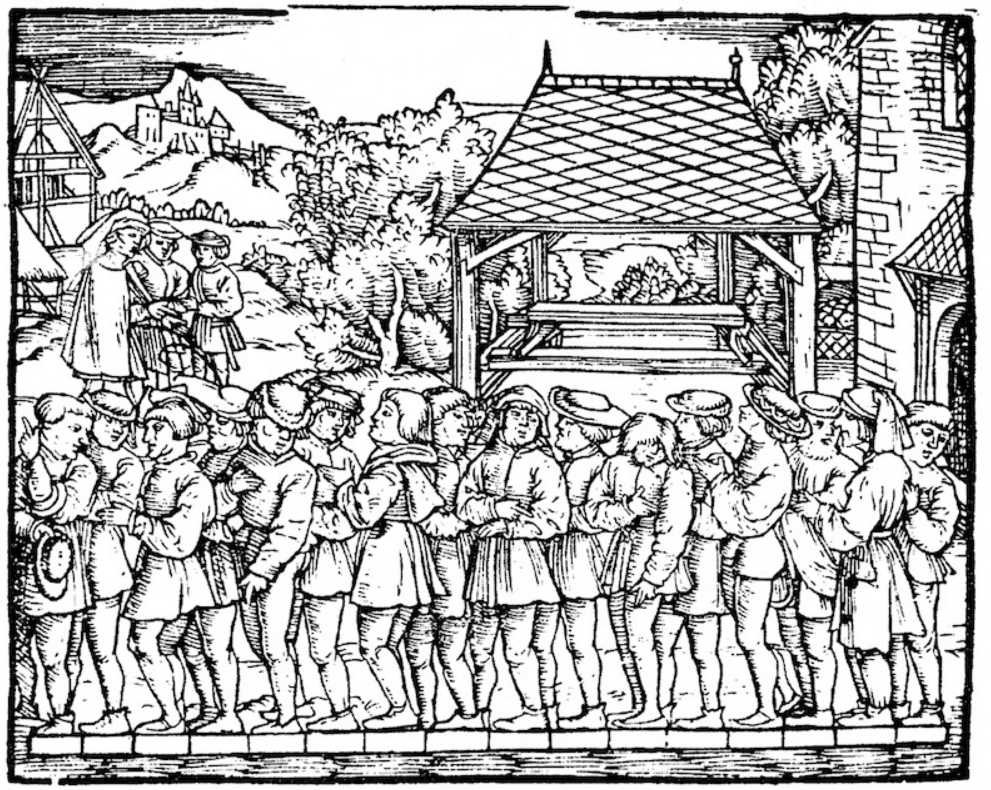


Fig. 1.4.1: Estimating the length of a foot

Fig. 1.4.1 illustrates how this estimator works. The 16 adult men were asked to line up in a row, when leaving church. Their aggregate length was then divided by 16 to obtain an estimate for what now amounts to 1 foot. This "algorithm" was later improved to deal with misshapen feet—the 2 men with the shortest and longest feet respectively were sent away, averaging only over the remainder. This is one of the earliest examples of the trimmed mean estimate.

Statistics really took off with the collection and availability of data. One of its titans, Ronald Fisher (1890-1962)[24], contributed significantly to its theory and also its applications in genetics. Many of his algorithms (such as Linear Discriminant Analysis) and formula (such as the Fisher Information Matrix) are still in frequent use today (even the Iris dataset that he released in 1936 is still used sometimes to illustrate machine learning algorithms). Fisher was also a proponent of eugenics, which should remind us that the morally dubious use data science has as long and enduring a history as its productive use in industry and the natural sciences.

A second influence for machine learning came from Information Theory (Claude Shannon, 1916-2001)[25] and the Theory of computation via Alan Turing (1912-1954)[26]. Turing posed the question "can machines think?" in his famous paper Computing machinery and intelligence[27] (Mind, October 1950). In what he described as the Turing test, a machine can be considered intelligent if it is difficult for a human evaluator to distinguish between the replies from a machine and a human based on textual interactions.

Another influence can be found in neuroscience and psychology. After all, humans clearly exhibit intelligent behavior. It is thus only reasonable to ask whether one could explain and possibly re-

[24] https://en.wikipedia.org/wiki/Ronald_Fisher

[25] https://en.wikipedia.org/wiki/Claude_Shannon

[26] https://en.wikipedia.org/wiki/Alan_Turing

[27] https://en.wikipedia.org/wiki/Computing_Machinery_and_Intelligence

verse engineer this capacity. One of the oldest algorithms inspired in this fashion was formulated by Donald Hebb (1904-1985)[28]. In his groundbreaking book The Organization of Behavior (Hebb & Hebb, 1949), he posited that neurons learn by positive reinforcement. This became known as the Hebbian learning rule. It is the prototype of Rosenblatt's perceptron learning algorithm and it laid the foundations of many stochastic gradient descent algorithms that underpin deep learning today: reinforce desirable behavior and diminish undesirable behavior to obtain good settings of the parameters in a neural network.

Biological inspiration is what gave *neural networks* their name. For over a century (dating back to the models of Alexander Bain, 1873 and James Sherrington, 1890), researchers have tried to assemble computational circuits that resemble networks of interacting neurons. Over time, the interpretation of biology has become less literal but the name stuck. At its heart, lie a few key principles that can be found in most networks today:

- The alternation of linear and nonlinear processing units, often referred to as *layers*.
- The use of the chain rule (also known as *backpropagation*) for adjusting parameters in the entire network at once.

After initial rapid progress, research in neural networks languished from around 1995 until 2005. This was due to a number of reasons. Training a network is computationally very expensive. While RAM was plentiful at the end of the past century, computational power was scarce. Second, datasets were relatively small. In fact, Fisher's Iris dataset from 1932 was a popular tool for testing the efficacy of algorithms. MNIST with its 60,000 handwritten digits was considered huge.

Given the scarcity of data and computation, strong statistical tools such as Kernel Methods, Decision Trees and Graphical Models proved empirically superior. Unlike neural networks, they did not require weeks to train and provided predictable results with strong theoretical guarantees.

1.5 The Road to Deep Learning

Much of this changed with the ready availability of large amounts of data, due to the World Wide Web, the advent of companies serving hundreds of millions of users online, a dissemination of cheap, high quality sensors, cheap data storage (Kryder's law), and cheap computation (Moore's law), in particular in the form of GPUs, originally engineered for computer gaming. Suddenly algorithms and models that seemed computationally infeasible became relevant (and vice versa). This is best illustrated in Table 1.5.1.

Table 1.5.1: Dataset versus computer memory and computational power

Decade	Dataset	Memory	Floating Point Calculations per Second
1970	100 (Iris)	1 KB	100 KF (Intel 8080)
1980	1 K (House prices in Boston)	100 KB	1 MF (Intel 80186)
1990	10 K (optical character recognition)	10 MB	10 MF (Intel 80486)
2000	10 M (web pages)	100 MB	1 GF (Intel Core)
2010	10 G (advertising)	1 GB	1 TF (Nvidia C2050)
2020	1 T (social network)	100 GB	1 PF (Nvidia DGX-2)

It is evident that RAM has not kept pace with the growth in data. At the same time, the increase

[28] https://en.wikipedia.org/wiki/Donald_O._Hebb

in computational power has outpaced that of the data available. This means that statistical models needed to become more memory efficient (this is typically achieved by adding nonlinearities) while simultaneously being able to spend more time on optimizing these parameters, due to an increased compute budget. Consequently the sweet spot in machine learning and statistics moved from (generalized) linear models and kernel methods to deep networks. This is also one of the reasons why many of the mainstays of deep learning, such as multilayer perceptrons (McCulloch & Pitts, 1943), convolutional neural networks (LeCun et al., 1998), Long Short-Term Memory (Hochreiter & Schmidhuber, 1997), and Q-Learning (Watkins & Dayan, 1992), were essentially "rediscovered" in the past decade, after laying comparatively dormant for considerable time.

The recent progress in statistical models, applications, and algorithms, has sometimes been likened to the Cambrian Explosion: a moment of rapid progress in the evolution of species. Indeed, the state of the art is not just a mere consequence of available resources, applied to decades old algorithms. Note that the list below barely scratches the surface of the ideas that have helped researchers achieve tremendous progress over the past decade.

- Novel methods for capacity control, such as Dropout (Srivastava et al., 2014) have helped to mitigate the danger of overfitting. This was achieved by applying noise injection (Bishop, 1995) throughout the network, replacing weights by random variables for training purposes.
- Attention mechanisms solved a second problem that had plagued statistics for over a century: how to increase the memory and complexity of a system without increasing the number of learnable parameters. (Bahdanau et al., 2014) found an elegant solution by using what can only be viewed as a learnable pointer structure. Rather than having to remember an entire sentence, e.g., for machine translation in a fixed-dimensional representation, all that needed to be stored was a pointer to the intermediate state of the translation process. This allowed for significantly increased accuracy for long sentences, since the model no longer needed to remember the entire sentence before commencing the generation of a new sentence.
- Multi-stage designs, e.g., via the Memory Networks (MemNets) (Sukhbaatar et al., 2015) and the Neural Programmer-Interpreter (Reed & DeFreitas, 2015) allowed statistical modelers to describe iterative approaches to reasoning. These tools allow for an internal state of the deep network to be modified repeatedly, thus carrying out subsequent steps in a chain of reasoning, similar to how a processor can modify memory for a computation.
- Another key development was the invention of GANS (Goodfellow et al., 2014). Traditionally, statistical methods for density estimation and generative models focused on finding proper probability distributions and (often approximate) algorithms for sampling from them. As a result, these algorithms were largely limited by the lack of flexibility inherent in the statistical models. The crucial innovation in GANs was to replace the sampler by an arbitrary algorithm with differentiable parameters. These are then adjusted in such a way that the discriminator (effectively a two-sample test) cannot distinguish fake from real data. Through the ability to use arbitrary algorithms to generate data, it opened up density estimation to a wide variety of techniques. Examples of galloping Zebras (Zhu et al., 2017) and of fake celebrity faces (Karras et al., 2017) are both testimony to this progress.
- In many cases, a single GPU is insufficient to process the large amounts of data available for training. Over the past decade the ability to build parallel distributed training algorithms has improved significantly. One of the key challenges in designing scalable algorithms is that the workhorse of deep learning optimization, stochastic gradient descent, relies on relatively small minibatches of data to be processed. At the same time, small batches limit the efficiency of GPUs. Hence, training on 1024 GPUs with a minibatch size of, say 32 images

per batch amounts to an aggregate minibatch of 32k images. Recent work, first by Li (Li, 2017), and subsequently by (You et al., 2017) and (Jia et al., 2018) pushed the size up to 64k observations, reducing training time for ResNet50 on ImageNet to less than 7 minutes. For comparison—initially training times were measured in the order of days.

- The ability to parallelize computation has also contributed quite crucially to progress in reinforcement learning, at least whenever simulation is an option. This has led to significant progress in computers achieving superhuman performance in Go, Atari games, Starcraft, and in physics simulations (e.g., using MuJoCo). See e.g., (Silver et al., 2016) for a description of how to achieve this in AlphaGo. In a nutshell, reinforcement learning works best if plenty of (state, action, reward)triples are available, i.e., whenever it is possible to try out lots of things to learn how they relate to each other. Simulation provides such an avenue.
- Deep Learning frameworks have played a crucial role in disseminating ideas. The first generation of frameworks allowing for easy modeling encompassed Caffe[29], Torch[30], and Theano[31]. Many seminal papers were written using these tools. By now, they have been superseded by TensorFlow[32], often used via its high level API Keras[33], CNTK[34], Caffe 2[35], and Apache MxNet[36]. The third generation of tools, namely imperative tools for deep learning, was arguably spearheaded by Chainer[37], which used a syntax similar to Python NumPy to describe models. This idea was adopted by PyTorch[38] and the Gluon API[39] of MXNet. It is the latter group that this course uses to teach deep learning.

The division of labor between systems researchers building better tools and statistical modelers building better networks has greatly simplified things. For instance, training a linear logistic regression model used to be a nontrivial homework problem, worthy to give to new machine learning PhD students at Carnegie Mellon University in 2014. By now, this task can be accomplished with less than 10 lines of code, putting it firmly into the grasp of programmers.

1.6 Success Stories

Artificial Intelligence has a long history of delivering results that would be difficult to accomplish otherwise. For instance, mail is sorted using optical character recognition. These systems have been deployed since the 90s (this is, after all, the source of the famous MNIST and USPS sets of handwritten digits). The same applies to reading checks for bank deposits and scoring creditworthiness of applicants. Financial transactions are checked for fraud automatically. This forms the backbone of many e-commerce payment systems, such as PayPal, Stripe, AliPay, WeChat, Apple, Visa, MasterCard. Computer programs for chess have been competitive for decades. Machine learning feeds search, recommendation, personalization and ranking on the Internet. In other words, artificial intelligence and machine learning are pervasive, albeit often hidden from sight.

It is only recently that AI has been in the limelight, mostly due to solutions to problems that were considered intractable previously.

[29] https://github.com/BVLC/caffe
[30] https://github.com/torch
[31] https://github.com/Theano/Theano
[32] https://github.com/tensorflow/tensorflow
[33] https://github.com/keras-team/keras
[34] https://github.com/Microsoft/CNTK
[35] https://github.com/caffe2/caffe2
[36] https://github.com/apache/incubator-mxnet
[37] https://github.com/chainer/chainer
[38] https://github.com/pytorch/pytorch
[39] https://github.com/apache/incubator-mxnet

- Intelligent assistants, such as Apple's Siri, Amazon's Alexa, or Google's assistant are able to answer spoken questions with a reasonable degree of accuracy. This includes menial tasks such as turning on light switches (a boon to the disabled) up to making barber's appointments and offering phone support dialog. This is likely the most noticeable sign that AI is affecting our lives.
- A key ingredient in digital assistants is the ability to recognize speech accurately. Gradually the accuracy of such systems has increased to the point where they reach human parity (Xiong et al., 2018) for certain applications.
- Object recognition likewise has come a long way. Estimating the object in a picture was a fairly challenging task in 2010. On the ImageNet benchmark (Lin et al., 2010) achieved a top-5 error rate of 28%. By 2017, (Hu et al., 2018) reduced this error rate to 2.25%. Similarly stunning results have been achieved for identifying birds, or diagnosing skin cancer.
- Games used to be a bastion of human intelligence. Starting from TDGammon [23], a program for playing Backgammon using temporal difference (TD) reinforcement learning, algorithmic and computational progress has led to algorithms for a wide range of applications. Unlike Backgammon, chess has a much more complex state space and set of actions. DeepBlue beat Gary Kasparov, Campbell et al. (Campbell et al., 2002), using massive parallelism, special purpose hardware and efficient search through the game tree. Go is more difficult still, due to its huge state space. AlphaGo reached human parity in 2015, (Silver et al., 2016) using Deep Learning combined with Monte Carlo tree sampling. The challenge in Poker was that the state space is large and it is not fully observed (we do not know the opponents' cards). Libratus exceeded human performance in Poker using efficiently structured strategies (Brown & Sandholm, 2017). This illustrates the impressive progress in games and the fact that advanced algorithms played a crucial part in them.
- Another indication of progress in AI is the advent of self-driving cars and trucks. While full autonomy is not quite within reach yet, excellent progress has been made in this direction, with companies such as Tesla, NVIDIA, and Waymo shipping products that enable at least partial autonomy. What makes full autonomy so challenging is that proper driving requires the ability to perceive, to reason and to incorporate rules into a system. At present, deep learning is used primarily in the computer vision aspect of these problems. The rest is heavily tuned by engineers.

Again, the above list barely scratches the surface of where machine learning has impacted practical applications. For instance, robotics, logistics, computational biology, particle physics, and astronomy owe some of their most impressive recent advances at least in parts to machine learning. ML is thus becoming a ubiquitous tool for engineers and scientists.

Frequently, the question of the AI apocalypse, or the AI singularity has been raised in non-technical articles on AI. The fear is that somehow machine learning systems will become sentient and decide independently from their programmers (and masters) about things that directly affect the livelihood of humans. To some extent, AI already affects the livelihood of humans in an immediate way—creditworthiness is assessed automatically, autopilots mostly navigate cars, decisions about whether to grant bail use statistical data as input. More frivolously, we can ask Alexa to switch on the coffee machine.

Fortunately, we are far from a sentient AI system that is ready to manipulate its human creators (or burn their coffee). First, AI systems are engineered, trained and deployed in a specific, goal-oriented manner. While their behavior might give the illusion of general intelligence, it is a combination of rules, heuristics and statistical models that underlie the design. Second, at present tools for *artificial general intelligence* simply do not exist that are able to improve themselves, reason about themselves, and that are able to modify, extend and improve their own architecture while trying to solve general tasks.

A much more pressing concern is how AI is being used in our daily lives. It is likely that many menial tasks fulfilled by truck drivers and shop assistants can and will be automated. Farm robots will likely reduce the cost for organic farming but they will also automate harvesting operations. This phase of the industrial revolution may have profound consequences on large swaths of society (truck drivers and shop assistants are some of the most common jobs in many states). Furthermore, statistical models, when applied without care can lead to racial, gender or age bias and raise reasonable concerns about procedural fairness if automated to drive consequential decisions. It is important to ensure that these algorithms are used with care. With what we know today, this strikes us a much more pressing concern than the potential of malevolent superintelligence to destroy humanity.

Summary

- Machine learning studies how computer systems can leverage *experience* (often data) to improve performance at specific tasks. It combines ideas from statistics, data mining, artificial intelligence, and optimization. Often, it is used as a means of implementing artificially-intelligent solutions.
- As a class of machine learning, representational learning focuses on how to automatically find the appropriate way to represent data. This is often accomplished by a progression of learned transformations.
- Much of the recent progress in deep learning has been triggered by an abundance of data arising from cheap sensors and Internet-scale applications, and by significant progress in computation, mostly through GPUs.
- Whole system optimization is a key component in obtaining good performance. The availability of efficient deep learning frameworks has made design and implementation of this significantly easier.

Exercises

1. Which parts of code that you are currently writing could be "learned", i.e., improved by learning and automatically determining design choices that are made in your code? Does your code include heuristic design choices?
2. Which problems that you encounter have many examples for how to solve them, yet no specific way to automate them? These may be prime candidates for using deep learning.
3. Viewing the development of artificial intelligence as a new industrial revolution, what is the relationship between algorithms and data? Is it similar to steam engines and coal (what is the fundamental difference)?
4. Where else can you apply the end-to-end training approach? Physics? Engineering? Econometrics?

2 | Preliminaries

To get started with deep learning, we will need to develop a few basic skills. All machine learning is concerned with extracting information from data. So we will begin by learning the practical skills for storing, manipulating, and preprocessing data.

Moreover, machine learning typically requires working with large datasets, which we can think of as tables, where the rows correspond to examples and the columns correspond to attributes. Linear algebra gives us a powerful set of techniques for working with tabular data. We will not go too far into the weeds but rather focus on the basic of matrix operations and their implementation.

Additionally, deep learning is all about optimization. We have a model with some parameters and we want to find those that fit our data *the best*. Determining which way to move each parameter at each step of an algorithm requires a little bit of calculus, which will be briefly introduced. Fortunately, the `autograd` package automatically computes differentiation for us, and we will cover it next.

Next, machine learning is concerned with making predictions: what is the likely value of some unknown attribute, given the information that we observe? To reason rigorously under uncertainty we will need to invoke the language of probability.

In the end, the official documentation provides plenty of descriptions and examples that are beyond this book. To conclude the chapter, we will show you how to look up documentation for the needed information.

This book has kept the mathematical content to the minimum necessary to get a proper understanding of deep learning. However, it does not mean that this book is mathematics free. Thus, this chapter provides a rapid introduction to basic and frequently-used mathematics to allow anyone to understand at least *most* of the mathematical content of the book. If you wish to understand *all* of the mathematical content, further reviewing Chapter 17 should be sufficient.

2.1 Data Manipulation

In order to get anything done, we need some way to store and manipulate data. Generally, there are two important things we need to do with data: (i) acquire them; and (ii) process them once they are inside the computer. There is no point in acquiring data absent some way to store it, so let's get our hands dirty first by playing with synthetic data. To start, we introduce the n-dimensional array (`ndarray`), MXNet's primary tool for storing and transforming data. In MXNet, `ndarray` is a class and we call any instance "an `ndarray`".

If you have worked with NumPy, the most widely-used scientific computing package in Python, then will find this section familiar. That's by design. We designed MXNet's `ndarray` to be an extension to NumPy's `ndarray` with a few killer features. First, MXNet's `ndarray` supports asynchronous computation on CPU, GPU, and distributed cloud architectures, whereas NumPy only supports CPU computation. Second, MXNet's ndarray supports automatic differentiation. These properties make MXNet's ndarray suitable for deep learning. Throughout the book, when we say ndarray, we are referring to MXNet's ndarray unless otherwise stated.

2.1.1 Getting Started

In this section, we aim to get you up and running, equipping you with the the basic math and numerical computing tools that you will build on as you progress through the book. Do not worry if you struggle to grok some of the mathematical concepts or library functions. The following sections will revisit this material in the context practical examples and it will sink. On the other hand, if you already have some background and want to go deeper into the mathematical content, just skip this section.

To start, we import the np (numpy) and npx (numpy_extension) modules from MXNet. Here, the np module includes functions supported by NumPy, while the npx module contains a set of extensions developed to empower deep learning within a NumPy-like environment. When using ndarray, we almost always invoke the set_np function: this is for compatibility of ndarray processing by other components of MXNet.

```
from mxnet import np, npx
npx.set_np()
```

An ndarray represents a (possibly multi-dimensional) array of numerical values. With one axis, an ndarray corresponds (in math) to a *vector*. With two axes, an ndarray corresponds to a *matrix*. Arrays with more than two axes do not have special mathematical names—we simply call them *tensors*.

To start, we can use arange to create a row vector x containing the first 12 integers starting with 0, though they are created as floats by default. Each of the values in an ndarray is called an *element* of the ndarray. For instance, there are 12 elements in the ndarray x. Unless otherwise specified, a new ndarray will be stored in main memory and designated for CPU-based computation.

```
x = np.arange(12)
x
```

```
array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10., 11.])
```

We can access an ndarray's *shape* (the length along each axis) by inspecting its shape property.

```
x.shape
```

```
(12,)
```

If we just want to know the total number of elements in an ndarray, i.e., the product of all of the shape elements, we can inspect its size property. Because we are dealing with a vector here, the single element of its shape is identical to its size.

```
x.size
```

```
12
```

To change the shape of an `ndarray` without altering either the number of elements or their values, we can invoke the `reshape` function. For example, we can transform our `ndarray`, `x`, from a row vector with shape $(12,)$ to a matrix with shape $(3, 4)$. This new `ndarray` contains the exact same values, but views them as a matrix organized as 3 rows and 4 columns. To reiterate, although the shape has changed, the elements in `x` have not. Note that the `size` is unaltered by reshaping.

```
x = x.reshape(3, 4)
x
```

```
array([[ 0.,  1.,  2.,  3.],
       [ 4.,  5.,  6.,  7.],
       [ 8.,  9., 10., 11.]])
```

Reshaping by manually specifying every dimension is unnecessary. If our target shape is a matrix with shape (height, width), then after we know the width, the height is given implicitly. Why should we have to perform the division ourselves? In the example above, to get a matrix with 3 rows, we specified both that it should have 3 rows and 4 columns. Fortunately, `ndarray` can automatically work out one dimension given the rest. We invoke this capability by placing `-1` for the dimension that we would like `ndarray` to automatically infer. In our case, instead of calling `x.reshape(3, 4)`, we could have equivalently called `x.reshape(-1, 4)` or `x.reshape(3, -1)`.

The `empty` method grabs a chunk of memory and hands us back a matrix without bothering to change the value of any of its entries. This is remarkably efficient but we must be careful because the entries might take arbitrary values, including very big ones!

```
np.empty((3, 4))
```

```
array([[4.5769305e-11, 4.5798638e-41, 3.9092172e-04, 3.0637990e-41],
       [0.0000000e+00, 0.0000000e+00, 0.0000000e+00, 0.0000000e+00],
       [0.0000000e+00, 0.0000000e+00, 0.0000000e+00, 0.0000000e+00]])
```

Typically, we will want our matrices initialized either with zeros, ones, some other constants, or numbers randomly sampled from a specific distribution. We can create an `ndarray` representing a tensor with all elements set to 0 and a shape of $(2, 3, 4)$ as follows:

```
np.zeros((2, 3, 4))
```

```
array([[[0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.]],

       [[0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.]]])
```

Similarly, we can create tensors with each element set to 1 as follows:

```
np.ones((2, 3, 4))
```

```
array([[[1., 1., 1., 1.],
        [1., 1., 1., 1.],
        [1., 1., 1., 1.]],

       [[1., 1., 1., 1.],
        [1., 1., 1., 1.],
        [1., 1., 1., 1.]]])
```

Often, we want to randomly sample the values for each element in an `ndarray` from some probability distribution. For example, when we construct arrays to serve as parameters in a neural network, we will typically inititialize their values randomly. The following snippet creates an `ndarray` with shape $(3, 4)$. Each of its elements is randomly sampled
from a standard Gaussian (normal) distribution with a mean of 0 and a standard deviation of 1.

```
np.random.normal(0, 1, size=(3, 4))
```

```
array([[ 2.2122064 ,  1.1630787 ,  0.7740038 ,  0.4838046 ],
       [ 1.0434405 ,  0.29956347,  1.1839255 ,  0.15302546],
       [ 1.8917114 , -1.1688148 , -1.2347414 ,  1.5580711 ]])
```

We can also specify the exact values for each element in the desired `ndarray` by supplying a Python list (or list of lists) containing the numerical values. Here, the outermost list corresponds to axis 0, and the inner list to axis 1.

```
np.array([[2, 1, 4, 3], [1, 2, 3, 4], [4, 3, 2, 1]])
```

```
array([[2., 1., 4., 3.],
       [1., 2., 3., 4.],
       [4., 3., 2., 1.]])
```

2.1.2 Operations

This book is not about software engineering. Our interests are not limited to simply reading and writing data from/to arrays. We want to perform mathematical operations on those arrays. Some of the simplest and most useful operations are the *elementwise* operations. These apply a standard scalar operation to each element of an array. For functions that take two arrays as inputs, elementwise operations apply some standard binary operator on each pair of corresponding elements from the two arrays. We can create an elementwise function from any function that maps from a scalar to a scalar.

In mathematical notation, we would denote such a *unary* scalar operator (taking one input) by the signature $f : \mathbb{R} \rightarrow \mathbb{R}$. This just mean that the function is mapping from any real number ($\mathbb{R}$) onto another. Likewise, we denote a *binary* scalar operator (taking two real inputs, and yielding one output) by the signature $f : \mathbb{R}, \mathbb{R} \rightarrow \mathbb{R}$. Given any two vectors $\mathbf{u}$ and $\mathbf{v}$ *of the same shape*, and a binary operator f, we can produce a vector $\mathbf{c} = F(\mathbf{u}, \mathbf{v})$ by setting $c_i \leftarrow f(u_i, v_i)$ for all i, where c_i, u_i, and v_i are the i^{th} elements of vectors $\mathbf{c}, \mathbf{u}$, and $\mathbf{v}$. Here, we produced the vector-valued $F : \mathbb{R}^d, \mathbb{R}^d \rightarrow \mathbb{R}^d$ by *lifting* the scalar function to an elementwise vector operation.

In MXNet, the common standard arithmetic operators (+, -, *, /, and **) have all been *lifted* to elementwise operations for any identically-shaped tensors of arbitrary shape. We can call elementwise operations on any two tensors of the same shape. In the following example, we use commas to formulate a 5-element tuple, where each element is the result of an elementwise operation.

```
x = np.array([1, 2, 4, 8])
y = np.array([2, 2, 2, 2])
x + y, x - y, x * y, x / y, x ** y  # The ** operator is exponentiation
```

```
(array([ 3.,  4.,  6., 10.]),
 array([-1.,  0.,  2.,  6.]),
 array([ 2.,  4.,  8., 16.]),
 array([0.5, 1. , 2. , 4. ]),
 array([ 1.,  4., 16., 64.]))
```

Many more operations can be applied elementwise, including unary operators like exponentiation.

```
np.exp(x)
```

```
array([2.7182817e+00, 7.3890562e+00, 5.4598148e+01, 2.9809580e+03])
```

In addition to elementwise computations, we can also perform linear algebra operations, including vector dot products and matrix multiplication. We will explain the crucial bits of linear algebra (with no assumed prior knowledge) in Section 2.3.

We can also *concatenate* multiple ndarrays together, stacking them end-to-end to form a larger ndarray. We just need to provide a list of ndarrays and tell the system along which axis to concatenate. The example below shows what happens when we concatenate two matrices along rows (axis 0, the first element of the shape) vs. columns (axis 1, the second element of the shape). We can see that, the first output ndarray's axis-0 length (6) is the sum of the two input ndarrays' axis-0 lengths ($3 + 3$); while the second output ndarray's axis-1 length (8) is the sum of the two input ndarrays' axis-1 lengths ($4 + 4$).

```
x = np.arange(12).reshape(3, 4)
y = np.array([[2, 1, 4, 3], [1, 2, 3, 4], [4, 3, 2, 1]])
np.concatenate([x, y], axis=0), np.concatenate([x, y], axis=1)
```

```
(array([[ 0.,  1.,  2.,  3.],
        [ 4.,  5.,  6.,  7.],
        [ 8.,  9., 10., 11.],
        [ 2.,  1.,  4.,  3.],
        [ 1.,  2.,  3.,  4.],
        [ 4.,  3.,  2.,  1.]]),
 array([[ 0.,  1.,  2.,  3.,  2.,  1.,  4.,  3.],
        [ 4.,  5.,  6.,  7.,  1.,  2.,  3.,  4.],
        [ 8.,  9., 10., 11.,  4.,  3.,  2.,  1.]]))
```

Sometimes, we want to construct a binary ndarray via *logical statements*. Take x == y as an example. For each position, if x and y are equal at that position, the corresponding entry in the new ndarray takes a value of 1, meaning that the logical statement x == y is true at that position; otherwise that position takes 0.

```
x == y
```

```
array([[False,  True, False,  True],
       [False, False, False, False],
       [False, False, False, False]])
```

Summing all the elements in the `ndarray` yields an `ndarray` with only one element.

```
x.sum()
```

```
array(66.)
```

For stylistic convenience, we can write `x.sum()`as `np.sum(x)`.

2.1.3 Broadcasting Mechanism

In the above section, we saw how to perform elementwise operations on two `ndarrays` of the same shape. Under certain conditions, even when shapes differ, we can still perform elementwise operations by invoking the *broadcasting mechanism*. These mechanisms work in the following way: First, expand one or both arrays by copying elements appropriately so that after this transformation, the two `ndarrays` have the same shape. Second, carry out the elementwise operations on the resulting arrays.

In most cases, we broadcast along an axis where an array initially only has length 1, such as in the following example:

```
a = np.arange(3).reshape(3, 1)
b = np.arange(2).reshape(1, 2)
a, b
```

```
(array([[0.],
        [1.],
        [2.]]), array([[0., 1.]]))
```

Since `a` and `b` are 3×1 and 1×2 matrices respectively, their shapes do not match up if we want to add them. We *broadcast* the entries of both matrices into a larger 3×2 matrix as follows: for matrix `a` it replicates the columns and for matrix `b` it replicates the rows before adding up both elementwise.

```
a + b
```

```
array([[0., 1.],
       [1., 2.],
       [2., 3.]])
```

2.1.4 Indexing and Slicing

Just as in any other Python array, elements in an ndarray can be accessed by index. As in any Python array, the first element has index 0 and ranges are specified to include the first but *before* the last element. As in standard Python lists, we can access elements according to their relative position to the end of the list by using negative indices.

Thus, [-1] selects the last element and [1:3] selects the second and the third elements as follows:

```
x[-1], x[1:3]
```

```
(array([ 8.,  9., 10., 11.]), array([[ 4.,  5.,  6.,  7.],
        [ 8.,  9., 10., 11.]]))
```

Beyond reading, we can also write elements of a matrix by specifying indices.

```
x[1, 2] = 9
x
```

```
array([[ 0.,  1.,  2.,  3.],
       [ 4.,  5.,  9.,  7.],
       [ 8.,  9., 10., 11.]])
```

If we want to assign multiple elements the same value, we simply index all of them and then assign them the value. For instance, [0:2, :] accesses the first and second rows, where : takes all the elements along axis 1 (column). While we discussed indexing for matrices, this obviously also works for vectors and for tensors of more than 2 dimensions.

```
x[0:2, :] = 12
x
```

```
array([[12., 12., 12., 12.],
       [12., 12., 12., 12.],
       [ 8.,  9., 10., 11.]])
```

2.1.5 Saving Memory

In the previous example, every time we ran an operation, we allocated new memory to host its results. For example, if we write y = x + y, we will dereference the ndarray that y used to point to and instead point y at the newly allocated memory. In the following example, we demonstrate this with Python's id() function, which gives us the exact address of the referenced object in memory. After running y = y + x, we will find that id(y) points to a different location. That is because Python first evaluates y + x, allocating new memory for the result and then makes y point to this new location in memory.

```
before = id(y)
y = y + x
id(y) == before
```

```
False
```

This might be undesirable for two reasons. First, we do not want to run around allocating memory unnecessarily all the time. In machine learning, we might have hundreds of megabytes of parameters and update all of them multiple times per second. Typically, we will want to perform these updates *in place*. Second, we might point at the same parameters from multiple variables. If we do not update in place, this could cause that discarded memory is not released, and make it possible for parts of our code to inadvertently reference stale parameters.

Fortunately, performing in-place operations in MXNet is easy. We can assign the result of an operation to a previously allocated array with slice notation, e.g., `y[:] = <expression>`. To illustrate this concept, we first create a new matrix `z` with the same shape as another `y`, using `zeros_like` to allocate a block of 0 entries.

```
z = np.zeros_like(y)
print('id(z):', id(z))
z[:] = x + y
print('id(z):', id(z))
```

```
id(z): 140368458524304
id(z): 140368458524304
```

If the value of `x` is not reused in subsequent computations, we can also use `x[:] = x + y` or `x += y` to reduce the memory overhead of the operation.

```
before = id(x)
x += y
id(x) == before
```

```
True
```

2.1.6 Conversion to Other Python Objects

Converting an MXNet `ndarray` to a NumPy `ndarray`, or vice versa, is easy. The converted result does not share memory. This minor inconvenience is actually quite important: when you perform operations on the CPU or on GPUs, you do not want MXNet to halt computation, waiting to see whether the NumPy package of Python might want to be doing something else with the same chunk of memory. The `array` and `asnumpy` functions do the trick.

```
a = x.asnumpy()
b = np.array(a)
type(a), type(b)
```

```
(numpy.ndarray, mxnet.numpy.ndarray)
```

To convert a size-1 `ndarray` to a Python scalar, we can invoke the `item` function or Python's built-in functions.

```
a = np.array([3.5])
a, a.item(), float(a), int(a)
```

```
(array([3.5]), 3.5, 3.5, 3)
```

Summary

- MXNet's ndarray is an extension to NumPy's ndarray with a few killer advantages that make it suitable for deep learning.
- MXNet's ndarray provides a variety of functionalities including basic mathematics operations, broadcasting, indexing, slicing, memory saving, and conversion to other Python objects.

Exercises

1. Run the code in this section. Change the conditional statement `x == y` in this section to `x < y` or `x > y`, and then see what kind of ndarray you can get.
2. Replace the two ndarrays that operate by element in the broadcasting mechanism with other shapes, e.g., three dimensional tensors. Is the result the same as expected?

2.2 Data Preprocessing

So far we have introduced a variety of techniques for manipulating data that are already stored in ndarrays. To apply deep learning to solving real-world problems, we often begin with preprocessing raw data, rather than those nicely prepared data in the ndarray format. Among popular data analytic tools in Python, the pandas package is commonly used. Like many other extension packages in the vast ecosystem of Python, pandas can work together with ndarray. So, we will briefly walk through steps for preprocessing raw data with pandas and converting them into the ndarray format. We will cover more data preprocessing techniques in later chapters.

2.2.1 Reading the Dataset

As an example, we begin by creating an artificial dataset that is stored in a csv (comma-separated values) file. Data stored in other formats may be processed in similar ways.

```
# Write the dataset row by row into a csv file
data_file = '../data/house_tiny.csv'
with open(data_file, 'w') as f:
    f.write('NumRooms,Alley,Price\n')  # Column names
```

(continues on next page)

(continued from previous page)

```
    f.write('NA,Pave,127500\n')  # Each row is a data point
    f.write('2,NA,106000\n')
    f.write('4,NA,178100\n')
    f.write('NA,NA,140000\n')
```

To load the raw dataset from the created csv file, we import the pandas package and invoke the read_csv function. This dataset has 4 rows and 3 columns, where each row describes the number of rooms ("NumRooms"), the alley type ("Alley"), and the price ("Price") of a house.

```
# If pandas is not installed, just uncomment the following line:
# !pip install pandas
import pandas as pd

data = pd.read_csv(data_file)
print(data)
```

```
   NumRooms Alley   Price
0       NaN  Pave  127500
1       2.0   NaN  106000
2       4.0   NaN  178100
3       NaN   NaN  140000
```

2.2.2 Handling Missing Data

Note that "NaN" entries are missing values. To handle missing data, typical methods include *imputation* and *deletion*, where imputation replaces missing values with substituted ones, while deletion ignores missing values. Here we will consider imputation.

By integer-location based indexing (iloc), we split data into inputs and outputs, where the former takes the first 2 columns while the later only keeps the last column. For numerical values in inputs that are missing, we replace the "NaN" entries with the mean value of the same column.

```
inputs, outputs = data.iloc[:, 0:2], data.iloc[:, 2]
inputs = inputs.fillna(inputs.mean())
print(inputs)
```

```
   NumRooms Alley
0       3.0  Pave
1       2.0   NaN
2       4.0   NaN
3       3.0   NaN
```

For categorical or discrete values in inputs, we consider "NaN" as a category. Since the "Alley" column only takes 2 types of categorical values "Pave" and "NaN", pandas can automatically convert this column to 2 columns "Alley_Pave" and "Alley_nan". A row whose alley type is "Pave" will set values of "Alley_Pave" and "Alley_nan" to 1 and 0. A row with a missing alley type will set their values to 0 and 1.

```
inputs = pd.get_dummies(inputs, dummy_na=True)
print(inputs)
```

```
   NumRooms  Alley_Pave  Alley_nan
0       3.0           1          0
1       2.0           0          1
2       4.0           0          1
3       3.0           0          1
```

2.2.3 Conversion to the ndarray Format

Now that all the entries in inputs and outputs are numerical, they can be converted to the ndarray format. Once data are in this format, they can be further manipulated with those ndarray functionalities that we have introduced in Section 2.1.

```
from mxnet import np

X, y = np.array(inputs.values), np.array(outputs.values)
X, y
```

```
(array([[3., 1., 0.],
        [2., 0., 1.],
        [4., 0., 1.],
        [3., 0., 1.]], dtype=float64),
 array([127500, 106000, 178100, 140000], dtype=int64))
```

Summary

- Like many other extension packages in the vast ecosystem of Python, pandas can work together with ndarray.
- Imputation and deletion can be used to handle missing data.

Exercises

Create a raw dataset with more rows and columns.

1. Delete the column with the most missing values.
2. Convert the preprocessed dataset to the ndarray format.

2.3 Linear Algebra

Now that you can store and manipulate data, let's briefly review the subset of basic linear algebra that you will need to understand and implement most of models covered in this book. Below, we introduce the basic mathematical objects, arithmetic, and operations in linear algebra, expressing each both through mathematical notation and the corresponding implementation in code.

2.3.1 Scalars

If you never studied linear algebra or machine learning, then your past experience with math probably consisted of thinking about one number at a time. And, if you ever balanced a checkbook or even paid for dinner at a restaurant then you already know how to do basic things like adding and multiplying pairs of numbers. For example, the temperature in Palo Alto is 52 degrees Fahrenheit. Formally, we call values consisting of just one numerical quantity *scalars*. If you wanted to convert this value to Celsius (the metric system's more sensible temperature scale), you would evaluate the expression $c = \frac{5}{9}(f - 32)$, setting f to 52. In this equation, each of the terms—5, 9, and 32—are scalar values. The placeholders c and f are called *variables* and they represented unknown scalar values.

In this book, we adopt the mathematical notation where scalar variables are denoted by ordinary lower-cased letters (e.g., x, y, and z). We denote the space of all (continuous) *real-valued* scalars by $\mathbb{R}$. For expedience, we will punt on rigorous definitions of what precisely *space* is, but just remember for now that the expression $x \in \mathbb{R}$ is a formal way to say that x is a real-valued scalar. The symbol $\in$ can be pronounced "in" and simply denotes membership in a set. Analogously, we could write $x, y \in \{0, 1\}$ to state that x and y are numbers whose value can only be 0 or 1.

In MXNet code, a scalar is represented by an `ndarray` with just one element. In the next snippet, we instantiate two scalars and perform some familiar arithmetic operations with them, namely addition, multiplication, division, and exponentiation.

```
from mxnet import np, npx
npx.set_np()

x = np.array(3.0)
y = np.array(2.0)

x + y, x * y, x / y, x ** y
```

```
(array(5.), array(6.), array(1.5), array(9.))
```

2.3.2 Vectors

You can think of a vector as simply a list of scalar values. We call these values the *elements* (*entries* or *components*) of the vector. When our vectors represent examples from our dataset, their values hold some real-world significance. For example, if we were training a model to predict the risk that a loan defaults, we might associate each applicant with a vector whose components correspond to their income, length of employment, number of previous defaults, and other factors. If we were studying the risk of heart attacks hospital patients potentially face, we might represent each patient by a vector whose components capture their most recent vital signs, cholesterol levels, minutes of exercise per day, etc. In math notation, we will usually denote vectors as bold-faced, lower-cased letters (e.g., $\mathbf{x}$, $\mathbf{y}$, and $\mathbf{z}$).

In MXNet, we work with vectors via 1-dimensional `ndarray`s. In general `ndarray`s can have arbitrary lengths, subject to the memory limits of your machine.

```
x = np.arange(4)
x
```

```
array([0., 1., 2., 3.])
```

We can refer to any element of a vector by using a subscript. For example, we can refer to the i^th element of $\mathbf{x}$ by x_i. Note that the element x_i is a scalar, so we do not bold-face the font when referring to it. Extensive literature considers column vectors to be the default orientation of vectors, so does this book. In math, a vector $\mathbf{x}$ can be written as

$$\mathbf{x} = \begin{bmatrix}x_{1} \\ x_{2} \\ \vdots \\ x_{n}\end{bmatrix}, \tag{2.3.1}$$

where $x_1, \ldots, x_n$ are elements of the vector. In code, we access any element by indexing into the `ndarray`.

```
x[3]
```

```
array(3.)
```

Length, Dimensionality, and Shape

Let's revisit some concepts from Section 2.1. A vector is just an array of numbers. And just as every array has a length, so does every vector. In math notation, if we want to say that a vector $\mathbf{x}$ consists of n real-valued scalars, we can express this as $\mathbf{x} \in \mathbb{R}^n$. The length of a vector is commonly called the *dimension* of the vector.

As with an ordinary Python array, we can access the length of an `ndarray` by calling Python's built-in `len()` function.

```
len(x)
```

```
4
```

When an `ndarray` represents a vector (with precisely one axis), we can also access its length via the `.shape` attribute. The shape is a tuple that lists the length (dimensionality) along each axis of the `ndarray`. For `ndarray`s with just one axis, the shape has just one element.

```
x.shape
```

```
(4,)
```

Note that the word “dimension” tends to get overloaded in these contexts and this tends to confuse people. To clarify, we use the dimensionality of a *vector* or an *axis* to refer to its length, i.e., the number of elements of a vector or an axis. However, we use the dimensionality of an `ndarray` to refer to the number of axes that an `ndarray` has. In this sense, the dimensionality of an `ndarray`’s some axis will be the length of that axis.

2.3.3 Matrices

Just as vectors generalize scalars from order 0 to order 1, matrices generalize vectors from order 1 to order 2. Matrices, which we will typically denote with bold-faced, capital letters (e.g., $\mathbf{X}$, $\mathbf{Y}$, and $\mathbf{Z}$), are represented in code as `ndarrays` with 2 axes.

In math notation, we use $\mathbf{A} \in \mathbb{R}^{m \times n}$ to express that the matrix $\mathbf{A}$ consists of m rows and n columns of real-valued scalars. Visually, we can illustrate any matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$ as a table, where each element a_{ij} belongs to the i^{th} row and j^{th} column:

$$\mathbf{A}=\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \\ \end{bmatrix}. \tag{2.3.2}$$

For any $\mathbf{A} \in \mathbb{R}^{m \times n}$, the shape of $\mathbf{A}$ is (m, n) or $m \times n$. Specifically, when a matrix has the same number of rows and columns, its shape becomes a square; thus, it is called a *square matrix*.

We can create an $m \times n$ matrix in MXNet by specifying a shape with two components m and n when calling any of our favorite functions for instantiating an `ndarray`.

```
A = np.arange(20).reshape(5, 4)
A
```

```
array([[ 0.,  1.,  2.,  3.],
       [ 4.,  5.,  6.,  7.],
       [ 8.,  9., 10., 11.],
       [12., 13., 14., 15.],
       [16., 17., 18., 19.]])
```

We can access the scalar element a_{ij} of a matrix $\mathbf{A}$ in (2.3.2) by specifying the indices for the row (i) and column (j), such as $[\mathbf{A}]_{ij}$. When the scalar elements of a matrix $\mathbf{A}$, such as in (2.3.2), are not given, we may simply use the lower-case letter of the matrix $\mathbf{A}$ with the index subscript, a_{ij}, to refer to $[\mathbf{A}]_{ij}$. To keep notation simple, commas are inserted to separate indices only when necessary, such as $a_{2,3j}$ and $[\mathbf{A}]_{2i-1,3}$.

Sometimes, we want to flip the axes. When we exchange a matrix’s rows and columns, the result is called the *transpose* of the matrix. Formally, we signify a matrix $\mathbf{A}$’s transpose by $\mathbf{A}^\top$ and if $\mathbf{B} = \mathbf{A}^\top$, then $b_{ij} = a_{ji}$ for any i and j. Thus, the transpose of $\mathbf{A}$ in (2.3.2) is a $n \times m$ matrix:

$$\mathbf{A}^\top = \begin{bmatrix} a_{11} & a_{21} & \dots & a_{m1} \\ a_{12} & a_{22} & \dots & a_{m2} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1n} & a_{2n} & \dots & a_{mn} \end{bmatrix}. \tag{2.3.3}$$

In code, we access a matrix's transpose via the `T` attribute.

```
A.T
```

```
array([[ 0.,  4.,  8., 12., 16.],
       [ 1.,  5.,  9., 13., 17.],
       [ 2.,  6., 10., 14., 18.],
       [ 3.,  7., 11., 15., 19.]])
```

As a special type of the square matrix, a *symmetric matrix* $\mathbf{A}$ is equal to its transpose: $\mathbf{A} = \mathbf{A}^\top$.

```
B = np.array([[1, 2, 3], [2, 0, 4], [3, 4, 5]])
B
```

```
array([[1., 2., 3.],
       [2., 0., 4.],
       [3., 4., 5.]])
```

```
B == B.T
```

```
array([[ True,  True,  True],
       [ True,  True,  True],
       [ True,  True,  True]])
```

Matrices are useful data structures: they allow us to organize data that have different modalities of variation. For example, rows in our matrix might correspond to different houses (data points), while columns might correspond to different attributes. This should sound familiar if you have ever used spreadsheet software or have read Section 2.2. Thus, although the default orientation of a single vector is a column vector, in a matrix that represents a tabular dataset, it is more conventional to treat each data point as a row vector in the matrix. And, as we will see in later chapters, this convention will enable common deep learning practices. For example, along the outermost axis of an `ndarray`, we can access or enumerate minibatches of data points, or just data points if no minibatch exists.

2.3.4 Tensors

Just as vectors generalize scalars, and matrices generalize vectors, we can build data structures with even more axes. Tensors give us a generic way of describing `ndarray`s with an arbitrary number of axes. Vectors, for example, are first-order tensors, and matrices are second-order tensors. Tensors are denoted with capital letters of a special font face (e.g., X, Y, and Z) and their indexing mechanism (e.g., x_{ijk} and $[\mathsf{X}]_{1,2i-1,3}$) is similar to that of matrices.

Tensors will become more important when we start working with images, which arrive as `ndarray`s with 3 axes corresponding to the height, width, and a *channel* axis for stacking the color channels (red, green, and blue). For now, we will skip over higher order tensors and focus on the basics.

```
X = np.arange(24).reshape(2, 3, 4)
X
```

```
array([[[ 0.,  1.,  2.,  3.],
        [ 4.,  5.,  6.,  7.],
        [ 8.,  9., 10., 11.]],

       [[12., 13., 14., 15.],
        [16., 17., 18., 19.],
        [20., 21., 22., 23.]]])
```

2.3.5 Basic Properties of Tensor Arithmetic

Scalars, vectors, matrices, and tensors of an arbitrary number of axes have some nice properties that often come in handy. For example, you might have noticed from the definition of an elementwise operation that any elementwise unary operation does not change the shape of its operand. Similarly, given any two tensors with the same shape, the result of any binary elementwise operation will be a tensor of that same shape. For example, adding two matrices of the same shape performs elementwise addition over these two matrices.

```
A = np.arange(20).reshape(5, 4)
B = A.copy()  # Assign a copy of A to B by allocating new memory
A, A + B
```

```
(array([[ 0.,  1.,  2.,  3.],
        [ 4.,  5.,  6.,  7.],
        [ 8.,  9., 10., 11.],
        [12., 13., 14., 15.],
        [16., 17., 18., 19.]]), array([[ 0.,  2.,  4.,  6.],
        [ 8., 10., 12., 14.],
        [16., 18., 20., 22.],
        [24., 26., 28., 30.],
        [32., 34., 36., 38.]]))
```

Specifically, elementwise multiplication of two matrices is called their *Hadamard product* (math notation $\odot$). Consider matrix $\mathbf{B} \in \mathbb{R}^{m\times n}$ whose element of row i and column j is b_{ij}. The Hadamard product of matrices $\mathbf{A}$ (defined in (2.3.2)) and $\mathbf{B}$

$$\mathbf{A} \odot \mathbf{B} = \begin{bmatrix} a_{11}b_{11} & a_{12}b_{12} & \dots & a_{1n}b_{1n} \\ a_{21}b_{21} & a_{22}b_{22} & \dots & a_{2n}b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1}b_{m1} & a_{m2}b_{m2} & \dots & a_{mn}b_{mn} \end{bmatrix}. \tag{2.3.4}$$

```
A * B
```

```
array([[  0.,   1.,   4.,   9.],
       [ 16.,  25.,  36.,  49.],
       [ 64.,  81., 100., 121.],
       [144., 169., 196., 225.],
       [256., 289., 324., 361.]])
```

Multiplying or adding a tensor by a scalar also does not change the shape of the tensor, where each element of the operand tensor will be added or multiplied by the scalar.

```
a = 2
X = np.arange(24).reshape(2, 3, 4)
a + X, (a * X).shape
```

```
(array([[[ 2.,  3.,  4.,  5.],
         [ 6.,  7.,  8.,  9.],
         [10., 11., 12., 13.]],

        [[14., 15., 16., 17.],
         [18., 19., 20., 21.],
         [22., 23., 24., 25.]]]), (2, 3, 4))
```

2.3.6 Reduction

One useful operation that we can perform with arbitrary tensors is to calculate the sum of their elements. In mathematical notation, we express sums using the $\sum$ symbol. To express the sum of the elements in a vector $\mathbf{x}$ of length d, we write $\sum_{i=1}^{d} x_i$. In code, we can just call the `sum` function.

```
x = np.arange(4)
x, x.sum()
```

```
(array([0., 1., 2., 3.]), array(6.))
```

We can express sums over the elements of tensors of arbitrary shape. For example, the sum of the elements of an $m \times n$ matrix $\mathbf{A}$ could be written $\sum_{i=1}^{m} \sum_{j=1}^{n} a_{ij}$.

```
A.shape, A.sum()
```

```
((5, 4), array(190.))
```

By default, invoking the `sum` function *reduces* a tensor along all its axes to a scalar. We can also specify the axes along which the tensor is reduced via summation. Take matrices as an example. To reduce the row dimension (axis 0) by summing up elements of all the rows, we specify `axis=0` when invoking `sum`. Since the input matrix reduces along axis 0 to generate the output vector, the dimension of axis 0 of the input is lost in the output shape.

```
A_sum_axis0 = A.sum(axis=0)
A_sum_axis0, A_sum_axis0.shape
```

```
(array([40., 45., 50., 55.]), (4,))
```

Specifying `axis=1` will reduce the column dimension (axis 1) by summing up elements of all the columns. Thus, the dimension of axis 1 of the input is lost in the output shape.

```
A_sum_axis1 = A.sum(axis=1)
A_sum_axis1, A_sum_axis1.shape
```

```
(array([ 6., 22., 38., 54., 70.]), (5,))
```

Reducing a matrix along both rows and columns via summation is equivalent to summing up all the elements of the matrix.

```
A.sum(axis=[0, 1])  # Same as A.sum()
```

```
array(190.)
```

A related quantity is the *mean*, which is also called the *average*. We calculate the mean by dividing the sum by the total number of elements. In code, we could just call `mean` on tensors of arbitrary shape.

```
A.mean(), A.sum() / A.size
```

```
(array(9.5), array(9.5))
```

Like `sum`, `mean` can also reduce a tensor along the specified axes.

```
A.mean(axis=0), A.sum(axis=0) / A.shape[0]
```

```
(array([ 8.,  9., 10., 11.]), array([ 8.,  9., 10., 11.]))
```

Non-Reduction Sum

However, sometimes it can be useful to keep the number of axes unchanged when invoking `sum` or `mean` by setting `keepdims=True`.

```
sum_A = A.sum(axis=1, keepdims=True)
sum_A
```

```
array([[ 6.],
       [22.],
       [38.],
       [54.],
       [70.]])
```

For instance, since `sum_A` still keeps its 2 axes after summing each row, we can divide `A` by `sum_A` with broadcasting.

```
A / sum_A
```

```
array([[0.        , 0.16666667, 0.33333334, 0.5       ],
       [0.18181819, 0.22727273, 0.27272728, 0.3181818 ],
       [0.21052632, 0.23684211, 0.2631579 , 0.28947368],
       [0.22222222, 0.24074075, 0.25925925, 0.2777778 ],
       [0.22857143, 0.24285714, 0.25714287, 0.27142859]])
```

If we want to calculate the cumulative sum of elements of `A` along some axis, say `axis=0` (row by row), we can call the `cumsum` function. This function will not reduce the input tensor along any axis.

```
A.cumsum(axis=0)
```

```
array([[ 0.,  1.,  2.,  3.],
       [ 4.,  6.,  8., 10.],
       [12., 15., 18., 21.],
       [24., 28., 32., 36.],
       [40., 45., 50., 55.]])
```

2.3.7 Dot Products

So far, we have only performed elementwise operations, sums, and averages. And if this was all we could do, linear algebra probably would not deserve its own section. However, one of the most fundamental operations is the dot product. Given two vectors $\mathbf{x}, \mathbf{y} \in \mathbb{R}^d$, their *dot product* $\mathbf{x}^\top \mathbf{y}$ (or $\langle \mathbf{x}, \mathbf{y} \rangle$) is a sum over the products of the elements at the same position: $\mathbf{x}^\top \mathbf{y} = \sum_{i=1}^{d} x_i y_i$.

```
y = np.ones(4)
x, y, np.dot(x, y)
```

```
(array([0., 1., 2., 3.]), array([1., 1., 1., 1.]), array(6.))
```

Note that we can express the dot product of two vectors equivalently by performing an elementwise multiplication and then a sum:

```
np.sum(x * y)
```

```
array(6.)
```

Dot products are useful in a wide range of contexts. For example, given some set of values, denoted by a vector $\mathbf{x} \in \mathbb{R}^d$ and a set of weights denoted by $\mathbf{w} \in \mathbb{R}^d$, the weighted sum of the values in $\mathbf{x}$ according to the weights $\mathbf{w}$ could be expressed as the dot product $\mathbf{x}^\top \mathbf{w}$. When the weights are non-negative and sum to one (i.e., $\left(\sum_{i=1}^{d} w_i = 1\right)$), the dot product expresses a *weighted average*. After normalizing two vectors to have the unit length, the dot products express the cosine of the angle between them. We will formally introduce this notion of *length* later in this section.

2.3.8 Matrix-Vector Products

Now that we know how to calculate dot products, we can begin to understand *matrix-vector products*. Recall the matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$ and the vector $\mathbf{x} \in \mathbb{R}^n$ defined and visualized in (2.3.2) and (2.3.1) respectively. Let's start off by visualizing the matrix $\mathbf{A}$ in terms of its row vectors

$$\mathbf{A} = \begin{bmatrix} \mathbf{a}^\top_1 \\ \mathbf{a}^\top_2 \\ \vdots \\ \mathbf{a}^\top_m \end{bmatrix}, \tag{2.3.5}$$

where each $\mathbf{a}^\top_i \in \mathbb{R}^n$ is a row vector representing the $i^\textrm{th}$ row of the matrix $\mathbf{A}$. The matrix-vector product $\mathbf{A}\mathbf{x}$ is simply a column vector of length m, whose $i^\textrm{th}$ element is the dot product $\mathbf{a}^\top_i \mathbf{x}$:

$$\mathbf{A}\mathbf{x} = \begin{bmatrix} \mathbf{a}^\top_{1} \\ \mathbf{a}^\top_{2} \\ \vdots \\ \mathbf{a}^\top_m \end{bmatrix}\mathbf{x} = \begin{bmatrix} \mathbf{a}^\top_{1} \mathbf{x} \\ \mathbf{a}^\top_{2} \mathbf{x} \\ \vdots\\ \mathbf{a}^\top_{m} \mathbf{x}\end{bmatrix}. \tag{2.3.6}$$

We can think of multiplication by a matrix $\mathbf{A}\in \mathbb{R}^{m \times n}$ as a transformation that projects vectors from $\mathbb{R}^{n}$ to $\mathbb{R}^{m}$. These transformations turn out to be remarkably useful. For example, we can represent rotations as multiplications by a square matrix. As we will see in subsequent chapters, we can also use matrix-vector products to describe the most intensive calculations required when computing each layer in a neural network given the values of the previous layer.

Expressing matrix-vector products in code with `ndarray`s, we use the same `dot` function as for dot products. When we call `np.dot(A, x)` with a matrix `A` and a vector `x`, the matrix-vector product is performed. Note that the column dimension of `A` (its length along axis 1) must be the same as the dimension of `x` (its length).

```
A.shape, x.shape, np.dot(A, x)
```

```
((5, 4), (4,), array([ 14.,  38.,  62.,  86., 110.]))
```

2.3.9 Matrix-Matrix Multiplication

If you have gotten the hang of dot products and matrix-vector products, then *matrix-matrix multiplication* should be straightforward.

Say that we have two matrices $\mathbf{A} \in \mathbb{R}^{n \times k}$ and $\mathbf{B} \in \mathbb{R}^{k \times m}$:

$$\mathbf{A}=\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1k} \\ a_{21} & a_{22} & \cdots & a_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nk} \\ \end{bmatrix},\quad \mathbf{B}=\begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1m} \\ b_{21} & b_{22} & \cdots & b_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ b_{k1} & b_{k2} & \cdots & b_{km} \\ \end{bmatrix}. \tag{2.3.7}$$

Denote by $\mathbf{a}^\top_{i} \in \mathbb{R}^k$ the row vector representing the $i^\textrm{th}$ row of the matrix $\mathbf{A}$, and let $\mathbf{b}_{j} \in \mathbb{R}^k$ be the column vector from the $j^\textrm{th}$ column of the matrix $\mathbf{B}$. To produce the matrix product $\mathbf{C} = \mathbf{A}\mathbf{B}$, it is easiest to think of $\mathbf{A}$ in terms of its row vectors and $\mathbf{B}$ in terms of its column vectors:

$$\mathbf{A}= \begin{bmatrix} \mathbf{a}^\top_{1} \\ \mathbf{a}^\top_{2} \\ \vdots \\ \mathbf{a}^\top_n \\ \end{bmatrix}, \quad \mathbf{B}=\begin{bmatrix} \mathbf{b}_{1} & \mathbf{b}_{2} & \cdots & \mathbf{b}_{m} \\ \end{bmatrix}. \tag{2.3.8}$$

Then the matrix product $\mathbf{C} \in \mathbb{R}^{n \times m}$ is produced as we simply compute each element c_{ij} as the dot product $\mathbf{a}^\top_i \mathbf{b}_j$:

$$\mathbf{C} = \mathbf{AB} = \begin{bmatrix} \mathbf{a}^\top_{1} \\ \mathbf{a}^\top_{2} \\ \vdots \\ \mathbf{a}^\top_n \\ \end{bmatrix} \begin{bmatrix} \mathbf{b}_{1} & \mathbf{b}_{2} & \cdots & \mathbf{b}_{m} \\ \end{bmatrix} = \begin{bmatrix} \mathbf{a}^\top_{1} \mathbf{b}_1 & \mathbf{a}^\top_{1}\mathbf{b}_2& \cdots & \mathbf{a}^\top_{1} \mathbf{b}_m \\ \mathbf{a}^\top_{2}\mathbf{b}_1 & \mathbf{a}^\top_{2} \mathbf{b}_2 & \cdots & \mathbf{a}^\top_{2} \mathbf{b}_m \\ \vdots & \vdots & \ddots &\vdots\\ \mathbf{a}^\top_{n} \mathbf{b}_1 & \mathbf{a}^\top_{n}\mathbf{b}_2& \cdots& \mathbf{a}^\top_{n} \mathbf{b}_m \end{bmatrix}. \tag{2.3.9}$$

We can think of the matrix-matrix multiplication **AB** as simply performing m matrix-vector products and stitching the results together to form an $n \times m$ matrix. Just as with ordinary dot products and matrix-vector products, we can compute matrix-matrix multiplication by using the `dot` function. In the following snippet, we perform matrix multiplication on `A` and `B`. Here, `A` is a matrix with 5 rows and 4 columns, and `B` is a matrix with 4 rows and 3 columns. After multiplication, we obtain a matrix with 5 rows and 3 columns.

```
B = np.ones(shape=(4, 3))
np.dot(A, B)
```

```
array([[ 6.,  6.,  6.],
       [22., 22., 22.],
       [38., 38., 38.],
       [54., 54., 54.],
       [70., 70., 70.]])
```

Matrix-matrix multiplication can be simply called *matrix multiplication*, and should not be confused with the Hadamard product.

2.3.10 Norms

Some of the most useful operators in linear algebra are *norms*. Informally, the norm of a vector tells us how *big* a vector is. The notion of *size* under consideration here concerns not dimensionality but rather the magnitude of the components.

In linear algebra, a vector norm is a function f that maps a vector to a scalar, satisfying a handful of properties. Given any vector **x**, the first property says that if we scale all the elements of a vector by a constant factor α, its norm also scales by the *absolute value* of the same constant factor:

$$f(\alpha \mathbf{x}) = |\alpha| f(\mathbf{x}). \tag{2.3.10}$$

The second property is the familiar triangle inequality:

$$f(\mathbf{x} + \mathbf{y}) \leq f(\mathbf{x}) + f(\mathbf{y}). \tag{2.3.11}$$

The third property simply says that the norm must be non-negative:

$$f(\mathbf{x}) \geq 0. \tag{2.3.12}$$

That makes sense, as in most contexts the smallest *size* for anything is 0. The final property requires that the smallest norm is achieved and only achieved by a vector consisting of all zeros.

$$\forall i, [\mathbf{x}]_i = 0 \Leftrightarrow f(\mathbf{x}) = 0. \tag{2.3.13}$$

You might notice that norms sound a lot like measures of distance. And if you remember Euclidean distances (think Pythagoras' theorem) from grade school, then the concepts of non-negativity and the triangle inequality might ring a bell. In fact, the Euclidean distance is a norm: specifically it is the ℓ_2 norm. Suppose that the elements in the n-dimensional vector **x** are $x_1, \ldots, x_n$. The ℓ_2 *norm* of **x** is the square root of the sum of the squares of the vector elements:

$$\|\mathbf{x}\|_2 = \sqrt{\sum_{i=1}^n x_i^2}, \tag{2.3.14}$$

where the subscript 2 is often omitted in ℓ_2 norms, i.e., $\|\mathbf{x}\|$ is equivalent to $\|\mathbf{x}\|_2$. In code, we can calculate the ℓ_2 norm of a vector by calling `linalg.norm`.

```
u = np.array([3, -4])
np.linalg.norm(u)
```

```
array(5.)
```

In deep learning, we work more often with the squared ℓ_2 norm. You will also frequently encounter the ℓ_1 *norm*, which is expressed as the sum of the absolute values of the vector elements:

$$\|\mathbf{x}\|_1 = \sum_{i=1}^{n} |x_i| . \tag{2.3.15}$$

As compared with the ℓ_2 norm, it is less influenced by outliers. To calculate the ℓ_1 norm, we compose the absolute value function with a sum over the elements.

```
np.abs(u).sum()
```

```
array(7.)
```

Both the ℓ_2 norm and the ℓ_1 norm are special cases of the more general ℓ_p *norm*:

$$\|\mathbf{x}\|_p = \left(\sum_{i=1}^{n} |x_i|^p\right)^{1/p} . \tag{2.3.16}$$

Analogous to ℓ_2 norms of vectors, the *Frobenius norm* of a matrix $\mathbf{X} \in \mathbb{R}^{m \times n}$ is the square root of the sum of the squares of the matrix elements:

$$\|\mathbf{X}\|_F = \sqrt{\sum_{i=1}^{m} \sum_{j=1}^{n} x_{ij}^2}. \tag{2.3.17}$$

The Frobenius norm satisfies all the properties of vector norms. It behaves as if it were an ℓ_2 norm of a matrix-shaped vector. Invoking `linalg.norm` will calculate the Frobenius norm of a matrix.

```
np.linalg.norm(np.ones((4, 9)))
```

```
array(6.)
```

Norms and Objectives

While we do not want to get too far ahead of ourselves, we can plant some intuition already about why these concepts are useful. In deep learning, we are often trying to solve optimization problems: *maximize* the probability assigned to observed data; *minimize* the distance between predictions and the ground-truth observations. Assign vector representations to items (like words, products, or news articles) such that the distance between similar items is minimized, and the distance between dissimilar items is maximized. Oftentimes, the objectives, perhaps the most important components of deep learning algorithms (besides the data), are expressed as norms.

2.3.11 More on Linear Algebra

In just this section, we have taught you all the linear algebra that you will need to understand a remarkable chunk of modern deep learning. There is a lot more to linear algebra and a lot of that mathematics is useful for machine learning. For example, matrices can be decomposed into factors, and these decompositions can reveal low-dimensional structure in real-world datasets. There are entire subfields of machine learning that focus on using matrix decompositions and their generalizations to high-order tensors to discover structure in datasets and solve prediction problems. But this book focuses on deep learning. And we believe you will be much more inclined to learn more mathematics once you have gotten your hands dirty deploying useful machine learning models on real datasets. So while we reserve the right to introduce more mathematics much later on, we will wrap up this section here.

If you are eager to learn more about linear algebra, you may refer to either Section 17.1 or other excellent resources (Strang, 1993; Kolter, 2008; Petersen et al., 2008).

Summary

- Scalars, vectors, matrices, and tensors are basic mathematical objects in linear algebra.
- Vectors generalize scalars, and matrices generalize vectors.
- In the `ndarray` representation, scalars, vectors, matrices, and tensors have 0, 1, 2, and an arbitrary number of axes, respectively.
- A tensor can be reduced along the specified axes by `sum` and `mean`.
- Elementwise multiplication of two matrices is called their Hadamard product. It is different from matrix multiplication.
- In deep learning, we often work with norms such as the ℓ_1 norm, the ℓ_2 norm, and the Frobenius norm.
- We can perform a variety of operations over scalars, vectors, matrices, and tensors with `ndarray` functions.

Exercises

1. Prove that the transpose of a matrix $\mathbf{A}$'s transpose is $\mathbf{A}$: $(\mathbf{A}^\top)^\top = \mathbf{A}$.
2. Given two matrices $\mathbf{A}$ and $\mathbf{B}$, show that the sum of transposes is equal to the transpose of a sum: $\mathbf{A}^\top + \mathbf{B}^\top = (\mathbf{A} + \mathbf{B})^\top$.
3. Given any square matrix $\mathbf{A}$, is $\mathbf{A} + \mathbf{A}^\top$ always symmetric? Why?
4. We defined the tensor `X` of shape $(2, 3, 4)$ in this section. What is the output of `len(X)`?
5. For a tensor `X` of arbitrary shape, does `len(X)` always correspond to the length of a certain axis of `X`? What is that axis?
6. Run `A / A.sum(axis=1)` and see what happens. Can you analyze the reason?
7. When traveling between two points in Manhattan, what is the distance that you need to cover in terms of the coordinates, i.e., in terms of avenues and streets? Can you travel diagonally?

8. Consider a tensor with shape $(2, 3, 4)$. What are the shapes of the summation outputs along axis 0, 1, and 2?

9. Feed a tensor with 3 or more axes to the `linalg.norm` function and observe its output. What does this function compute for `ndarrays` of arbitrary shape?

2.4 Calculus

Finding the area of a polygon had remained mysterious until at least $2,500$ years ago, when ancient Greeks divided a polygon into triangles and summed their areas. To find the area of curved shapes, such as a circle, ancient Greeks inscribed polygons in such shapes. As shown in Fig. 2.4.1, an inscribed polygon with more sides of equal length better approximates the circle. This process is also known as the *method of exhaustion*.

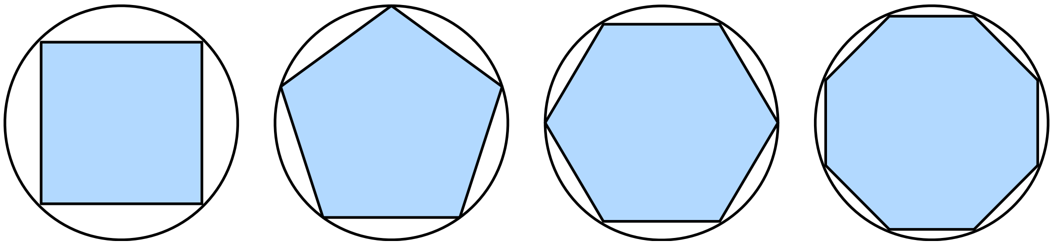


Fig. 2.4.1: Find the area of a circle with the method of exhaustion.

In fact, the method of exhaustion is where *integral calculus* (will be described in Section 17.5) originates from. More than $2,000$ years later, the other branch of calculus, *differential calculus*, was invented. Among the most critical applications of differential calculus, optimization problems consider how to do something *the best*. As discussed in Section 2.3.10, such problems are ubiquitous in deep learning.

In deep learning, we *train* models, updating them successively so that they get better and better as they see more and more data. Usually, getting better means minimizing a *loss function*, a score that answers the question "how *bad* is our model?" This question is more subtle than it appears. Ultimately, what we really care about is producing a model that performs well on data that we have never seen before. But we can only fit the model to data that we can actually see. Thus we can decompose the task of fitting models into two key concerns: i) *optimization*: the process of fitting our models to observed data; ii) *generalization*: the mathematical principles and practitioners' wisdom that guide as to how to produce models whose validity extends beyond the exact set of data points used to train them.

To help you understand optimization problems and methods in later chapters, here we give a very brief primer on differential calculus that is commonly used in deep learning.

2.4.1 Derivatives and Differentiation

We begin by addressing the calculation of derivatives, a crucial step in nearly all deep learning optimization algorithms. In deep learning, we typically choose loss functions that are differentiable with respect to our model's parameters. Put simply, this means that for each parameter, we can determine how rapidly the loss would increase or decrease, were we to *increase* or *decrease* that parameter by an infinitesimally small amount.

Suppose that we have a function $f : \mathbb{R} \rightarrow \mathbb{R}$, whose input and output are both scalars. The *derivative* of f is defined as

$$f'(x) = \lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h}, \tag{2.4.1}$$

if this limit exists. If $f'(a)$ exists, f is said to be *differentiable* at a. If f is differentiable at every number of an interval, then this function is differentiable on this interval. We can interpret the derivative $f'(x)$ in (2.4.1) as the *instantaneous* rate of change of $f(x)$ with respect to x. The so-called instantaneous rate of change is based on the variation h in x, which approaches 0.

To illustrate derivatives, let's experiment with an example. Define $u = f(x) = 3x^2 - 4x$.

```python
%matplotlib inline
import d2l
from IPython import display
from mxnet import np, npx
npx.set_np()

def f(x):
    return 3 * x ** 2 - 4 * x
```

By setting $x = 1$ and letting h approach 0, the numerical result of $\frac{f(x+h)-f(x)}{h}$ in (2.4.1) approaches 2. Though this experiment is not a mathematical proof, we will see later that the derivative u' is 2 when $x = 1$.

```python
def numerical_lim(f, x, h):
    return (f(x + h) - f(x)) / h

h = 0.1
for i in range(5):
    print('h=%.5f, numerical limit=%.5f' % (h, numerical_lim(f, 1, h)))
    h *= 0.1
```

```
h=0.10000, numerical limit=2.30000
h=0.01000, numerical limit=2.03000
h=0.00100, numerical limit=2.00300
h=0.00010, numerical limit=2.00030
h=0.00001, numerical limit=2.00003
```

Let's familiarize ourselves with a few equivalent notations for derivatives. Given $y = f(x)$, where x and y are the independent variable and the dependent variable of the function f, respectively. The following expressions are equivalent:

$$f'(x) = y' = \frac{dy}{dx} = \frac{df}{dx} = \frac{d}{dx} f(x) = Df(x) = D_x f(x), \tag{2.4.2}$$

where symbols $\frac{d}{dx}$ and D are *differentiation operators* that indicate operation of *differentiation*. We can use the following rules to differentiate common functions:

- $DC = 0$ (C is a constant),
- $Dx^n = nx^{n-1}$ (the *power rule*, n is any real number),
- $De^x = e^x$,
- $D\ln(x) = 1/x$.

To differentiate a function that is formed from a few simpler functions such as the above common functions, the following rules can be handy for us. Suppose that functions f and g are both differentiable and C is a constant, we have the *constant multiple rule*

$$\frac{d}{dx}[Cf(x)] = C\frac{d}{dx}f(x), \tag{2.4.3}$$

the *sum rule*

$$\frac{d}{dx}[f(x) + g(x)] = \frac{d}{dx}f(x) + \frac{d}{dx}g(x), \tag{2.4.4}$$

the *product rule*

$$\frac{d}{dx}[f(x)g(x)] = f(x)\frac{d}{dx}[g(x)] + g(x)\frac{d}{dx}[f(x)], \tag{2.4.5}$$

and the *quotient rule*

$$\frac{d}{dx}\left[\frac{f(x)}{g(x)}\right] = \frac{g(x)\frac{d}{dx}[f(x)] - f(x)\frac{d}{dx}[g(x)]}{[g(x)]^2}. \tag{2.4.6}$$

Now we can apply a few of the above rules to find $u' = f'(x) = 3\frac{d}{dx}x^2 - 4\frac{d}{dx}x = 6x - 4$. Thus, by setting $x = 1$, we have $u' = 2$: this is supported by our earlier experiment in this section where the numerical result approaches 2. This derivative is also the slope of the tangent line to the curve $u = f(x)$ when $x = 1$.

To visualize such an interpretation of derivatives, we will use `matplotlib`, a popular plotting library in Python. To configure properties of the figures produced by `matplotlib`, we need to define a few functions. In the following, the `use_svg_display` function specifies the `matplotlib` package to output the svg figures for sharper images. The comment `# Saved in the d2l package for later` use is a special mark where the following function, class, or import statements are also saved in the `d2l` package so that we can directly invoke `d2l.use_svg_display()` later.

```python
# Saved in the d2l package for later use
def use_svg_display():
    """Use the svg format to display a plot in Jupyter."""
    display.set_matplotlib_formats('svg')
```

We define the `set_figsize` function to specify the figure sizes. Note that here we directly use `d2l.plt` since the import statement `from matplotlib import pyplot as plt` has been marked for being saved in the `d2l` package in the preface.

```python
# Saved in the d2l package for later use
def set_figsize(figsize=(3.5, 2.5)):
    """Set the figure size for matplotlib."""
    use_svg_display()
    d2l.plt.rcParams['figure.figsize'] = figsize
```

The following `set_axes` function sets properties of axes of figures produced by `matplotlib`.

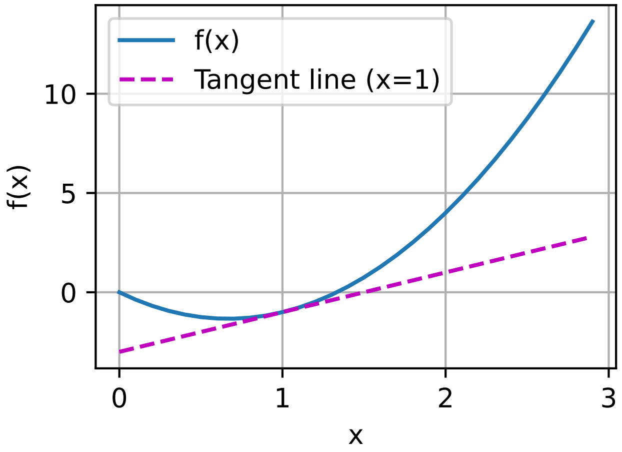
```
# Saved in the d2l package for later use
def set_axes(axes, xlabel, ylabel, xlim, ylim, xscale, yscale, legend):
    """Set the axes for matplotlib."""
    axes.set_xlabel(xlabel)
    axes.set_ylabel(ylabel)
    axes.set_xscale(xscale)
    axes.set_yscale(yscale)
    axes.set_xlim(xlim)
    axes.set_ylim(ylim)
    if legend:
        axes.legend(legend)
    axes.grid()
```

With these 3 functions for figure configurations, we define the `plot` function to plot multiple curves succinctly since we will need to visualize many curves throughout the book.

```
# Saved in the d2l package for later use
def plot(X, Y=None, xlabel=None, ylabel=None, legend=[], xlim=None,
         ylim=None, xscale='linear', yscale='linear',
         fmts=['-', 'm--', 'g-.', 'r:'], figsize=(3.5, 2.5), axes=None):
    """Plot data points."""
    d2l.set_figsize(figsize)
    axes = axes if axes else d2l.plt.gca()

    # Return True if X (ndarray or list) has 1 axis
    def has_one_axis(X):
        return (hasattr(X, "ndim") and X.ndim == 1 or isinstance(X, list)
                and not hasattr(X[0], "__len__"))

    if has_one_axis(X):
        X = [X]
    if Y is None:
        X, Y = [[]] * len(X), X
    elif has_one_axis(Y):
        Y = [Y]
    if len(X) != len(Y):
        X = X * len(Y)
    axes.cla()
    for x, y, fmt in zip(X, Y, fmts):
        if len(x):
            axes.plot(x, y, fmt)
        else:
            axes.plot(y, fmt)
    set_axes(axes, xlabel, ylabel, xlim, ylim, xscale, yscale, legend)
```

Now we can plot the function $u = f(x)$ and its tangent line $y = 2x - 3$ at $x = 1$, where the coefficient 2 is the slope of the tangent line.

```
x = np.arange(0, 3, 0.1)
plot(x, [f(x), 2 * x - 3], 'x', 'f(x)', legend=['f(x)', 'Tangent line (x=1)'])
```



2.4.2 Partial Derivatives

So far we have dealt with the differentiation of functions of just one variable. In deep learning, functions often depend on *many* variables. Thus, we need to extend the ideas of differentiation to these *multivariate* functions.

Let $y = f(x_1, x_2, \ldots, x_n)$ be a function with n variables. The *partial derivative* of y with respect to its i^{th} parameter x_i is

$$\frac{\partial y}{\partial x_i} = \lim_{h \rightarrow 0} \frac{f(x_1, \ldots, x_{i-1}, x_i + h, x_{i+1}, \ldots, x_n) - f(x_1, \ldots, x_i, \ldots, x_n)}{h}. \tag{2.4.7}$$

To calculate $\frac{\partial y}{\partial x_i}$, we can simply treat $x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n$ as constants and calculate the derivative of y with respect to x_i. For notation of partial derivatives, the following are equivalent:

$$\frac{\partial y}{\partial x_i} = \frac{\partial f}{\partial x_i} = f_{x_i} = f_i = D_i f = D_{x_i} f. \tag{2.4.8}$$

2.4.3 Gradients

We can concatenate partial derivatives of a multivariate function with respect to all its variables to obtain the *gradient* vector of the function. Suppose that the input of function $f : \mathbb{R}^n \rightarrow \mathbb{R}$ is an n-dimensional vector $\mathbf{x} = [x_1, x_2, \ldots, x_n]^\top$ and the output is a scalar. The gradient of the function $f(\mathbf{x})$ with respect to $\mathbf{x}$ is a vector of n partial derivatives:

$$\nabla_{\mathbf{x}} f(\mathbf{x}) = \left[\frac{\partial f(\mathbf{x})}{\partial x_1}, \frac{\partial f(\mathbf{x})}{\partial x_2}, \ldots, \frac{\partial f(\mathbf{x})}{\partial x_n}\right]^\top, \tag{2.4.9}$$

where $\nabla_{\mathbf{x}} f(\mathbf{x})$ is often replaced by $\nabla f(\mathbf{x})$ when there is no ambiguity.

Let $\mathbf{x}$ be an n-dimensional vector, the following rules are often used when differentiating multivariate functions:

- For all $\mathbf{A} \in \mathbb{R}^{m \times n}$, $\nabla_{\mathbf{x}} \mathbf{A} \mathbf{x} = \mathbf{A}^\top$,
- For all $\mathbf{A} \in \mathbb{R}^{n \times m}$, $\nabla_{\mathbf{x}} \mathbf{x}^\top \mathbf{A} = \mathbf{A}$,
- For all $\mathbf{A} \in \mathbb{R}^{n \times n}$, $\nabla_{\mathbf{x}} \mathbf{x}^\top \mathbf{A} \mathbf{x} = (\mathbf{A} + \mathbf{A}^\top)\mathbf{x}$,
- $\nabla_{\mathbf{x}} \|\mathbf{x}\|^2 = \nabla_{\mathbf{x}} \mathbf{x}^\top \mathbf{x} = 2\mathbf{x}$.

Similarly, for any matrix $\mathbf{X}$, we have $\nabla_{\mathbf{X}} \|\mathbf{X}\|_F^2 = 2\mathbf{X}$. As we will see later, gradients are useful for designing optimization algorithms in deep learning.

2.4.4 Chain Rule

However, such gradients can be hard to find. This is because multivariate functions in deep learning are often *composite*, so we may not apply any of the aforementioned rules to differentiate these functions. Fortunately, the *chain rule* enables us to differentiate composite functions.

Let's first consider functions of a single variable. Suppose that functions $y = f(u)$ and $u = g(x)$ are both differentiable, then the chain rule states that

$$\frac{dy}{dx} = \frac{dy}{du}\frac{du}{dx}. \tag{2.4.10}$$

Now let's turn our attention to a more general scenario where functions have an arbitrary number of variables. Suppose that the differentiable function y has variables $u_1, u_2, \ldots, u_m$, where each differentiable function u_i has variables $x_1, x_2, \ldots, x_n$. Note that y is a function of $x_1, x_2, \ldots, x_n$. Then the chain rule gives

$$\frac{dy}{dx_i} = \frac{dy}{du_1}\frac{du_1}{dx_i} + \frac{dy}{du_2}\frac{du_2}{dx_i} + \cdots + \frac{dy}{du_m}\frac{du_m}{dx_i} \tag{2.4.11}$$

for any $i = 1, 2, \ldots, n$.

Summary

- Differential calculus and integral calculus are two branches of calculus, where the former can be applied to the ubiquitous optimization problems in deep learning.
- A derivative can be interpreted as the instantaneous rate of change of a function with respect to its variable. It is also the slope of the tangent line to the curve of the function.
- A gradient is a vector whose components are the partial derivatives of a multivariate function with respect to all its variables.
- The chain rule enables us to differentiate composite functions.

Exercises

1. Plot the function $y = f(x) = x^3 - \frac{1}{x}$ and its tangent line when $x = 1$.
2. Find the gradient of the function $f(\mathbf{x}) = 3x_1^2 + 5e^{x_2}$.
3. What is the gradient of the function $f(\mathbf{x}) = \|\mathbf{x}\|_2$?
4. Can you write out the chain rule for the case where $u = f(x, y, z)$ and $x = x(a, b)$, $y = y(a, b)$, and $z = z(a, b)$?

2.5 Automatic Differentiation

As we have explained in Section 2.4, differentiation is a crucial step in nearly all deep learning optimization algorithms. While the calculations for taking these derivatives are straightforward, requiring only some basic calculus, for complex models, working out the updates by hand can be a pain (and often error-prone).

The `autograd` package expedites this work by automatically calculating derivatives, i.e., *automatic differentiation*. And while many other libraries require that we compile a symbolic graph to take automatic derivatives, `autograd` allows us to take derivatives while writing ordinary imperative code. Every time we pass data through our model, `autograd` builds a graph on the fly, tracking which data combined through which operations to produce the output. This graph enables `autograd` to subsequently backpropagate gradients on command. Here, *backpropagate* simply means to trace through the *computational graph*, filling in the partial derivatives with respect to each parameter.

```
from mxnet import autograd, np, npx
npx.set_np()
```

2.5.1 A Simple Example

As a toy example, say that we are interested in differentiating the function $y = 2\mathbf{x}^\top\mathbf{x}$ with respect to the column vector $\mathbf{x}$. To start, let's create the variable `x` and assign it an initial value.

```
x = np.arange(4)
x
```

```
array([0., 1., 2., 3.])
```

Note that before we even calculate the gradient of y with respect to $\mathbf{x}$, we will need a place to store it. It is important that we do not allocate new memory every time we take a derivative with respect to a parameter because we will often update the same parameters thousands or millions of times and could quickly run out of memory.

Note also that a gradient of a scalar-valued function with respect to a vector $\mathbf{x}$ is itself vector-valued and has the same shape as $\mathbf{x}$. Thus it is intuitive that in code, we will access a gradient taken with respect to `x` as an attribute of the `ndarray` `x` itself. We allocate memory for an `ndarray`'s gradient by invoking its `attach_grad` method.

```
x.attach_grad()
```

After we calculate a gradient taken with respect to `x`, we will be able to access it via the `grad` attribute. As a safe default, `x.grad` is initialized as an array containing all zeros. That is sensible because our most common use case for taking gradient in deep learning is to subsequently update parameters by adding (or subtracting) the gradient to maximize (or minimize) the differentiated function. By initializing the gradient to an array of zeros, we ensure that any update accidentally executed before a gradient has actually been calculated will not alter the parameters' value.

```
x.grad
```

```
array([0., 0., 0., 0.])
```

Now let's calculate y. Because we wish to subsequently calculate gradients, we want MXNet to generate a computational graph on the fly. We could imagine that MXNet would be turning on a recording device to capture the exact path by which each variable is generated.

Note that building the computational graph requires a nontrivial amount of computation. So MXNet will only build the graph when explicitly told to do so. We can invoke this behavior by placing our code inside an `autograd.record` scope.

```
with autograd.record():
    y = 2 * np.dot(x, x)
y
```

```
array(28.)
```

Since `x` is an `ndarray` of length 4, `np.dot` will perform an inner product of `x` and `x`, yielding the scalar output that we assign to `y`. Next, we can automatically calculate the gradient of `y` with respect to each component of `x` by calling `y`'s `backward` function.

```
y.backward()
```

If we recheck the value of `x.grad`, we will find its contents overwritten by the newly calculated gradient.

```
x.grad
```

```
array([ 0.,  4.,  8., 12.])
```

The gradient of the function $y = 2\mathbf{x}^\top\mathbf{x}$ with respect to $\mathbf{x}$ should be $4\mathbf{x}$. Let's quickly verify that our desired gradient was calculated correctly. If the two `ndarray`s are indeed the same, then the equality between them holds at every position.

```
x.grad == 4 * x
```

```
array([ True,  True,  True,  True])
```

If we subsequently compute the gradient of another variable whose value was calculated as a function of `x`, the contents of `x.grad` will be overwritten.

```
with autograd.record():
    y = x.sum()
y.backward()
x.grad
```

```
array([1., 1., 1., 1.])
```

2.5.2 Backward for Non-Scalar Variables

Technically, when y is not a scalar, the most natural interpretation of the gradient of y (a vector of length m) with respect to x (a vector of length n) is the *Jacobian* (an $m \times n$ matrix). For higher-order and higher-dimensional y and x, the Jacobian could be a gnarly high-order tensor.

However, while these more exotic objects do show up in advanced machine learning (including in deep learning), more often when we are calling backward on a vector, we are trying to calculate the derivatives of the loss functions for each constituent of a *batch* of training examples. Here, our intent is not to calculate the Jacobian but rather the sum of the partial derivatives computed individually for each example in the batch.

Thus when we invoke backward on a vector-valued variable y, which is a function of x, MXNet assumes that we want the sum of the gradients. In short, MXNet will create a new scalar variable by summing the elements in y, and compute the gradient of that scalar variable with respect to x.

```
with autograd.record():
    y = x * x  # y is a vector
y.backward()

u = x.copy()
u.attach_grad()
with autograd.record():
    v = (u * u).sum()  # v is a scalar
v.backward()

x.grad == u.grad
```

```
array([ True,  True,  True,  True])
```

2.5.3 Detaching Computation

Sometimes, we wish to move some calculations outside of the recorded computational graph. For example, say that y was calculated as a function of x, and that subsequently z was calculated as a function of both y and x. Now, imagine that we wanted to calculate the gradient of z with respect to x, but wanted for some reason to treat y as a constant, and only take into account the role that x played after y was calculated.

Here, we can call u = y.detach() to return a new variable u that has the same value as y but discards any information about how y was computed in the computational graph. In other words, the gradient will not flow backwards through u to x. This will provide the same functionality as if we had calculated u as a function of x outside of the autograd.record scope, yielding a u that will be treated as a constant in any backward call. Thus, the following backward function computes the partial derivative of z = u * x with respect to x while treating u as a constant, instead of the partial derivative of z = x * x * x with respect to x.

```
with autograd.record():
    y = x * x
    u = y.detach()
    z = u * x
z.backward()
x.grad == u
```

```
array([ True,  True,  True,  True])
```

Since the computation of y was recorded, we can subsequently call y.backward() to get the derivative of y = x * x with respect to x, which is 2 * x.

```
y.backward()
x.grad == 2 * x
```

```
array([ True,  True,  True,  True])
```

Note that attaching gradients to a variable x implicitly calls x = x.detach(). If x is computed based on other variables, this part of computation will not be used in the backward function.

```
y = np.ones(4) * 2
y.attach_grad()
with autograd.record():
    u = x * y
    u.attach_grad()  # Implicitly run u = u.detach()
    z = 5 * u - x
z.backward()
x.grad, u.grad, y.grad
```

```
(array([-1., -1., -1., -1.]), array([5., 5., 5., 5.]), array([0., 0., 0., 0.]))
```

2.5.4 Computing the Gradient of Python Control Flow

One benefit of using automatic differentiation is that even if building the computational graph of a function required passing through a maze of Python control flow (e.g., conditionals, loops, and arbitrary function calls), we can still calculate the gradient of the resulting variable. In the following snippet, note that the number of iterations of the while loop and the evaluation of the if statement both depend on the value of the input a.

```
def f(a):
    b = a * 2
    while np.linalg.norm(b) < 1000:
        b = b * 2
    if b.sum() > 0:
        c = b
    else:
        c = 100 * b
    return c
```

Again to compute gradients, we just need to record the calculation and then call the backward function.

```
a = np.random.normal()
a.attach_grad()
with autograd.record():
    d = f(a)
d.backward()
```

We can now analyze the f function defined above. Note that it is piecewise linear in its input a. In other words, for any a there exists some constant scalar k such that f(a) = k * a, where the value of k depends on the input a. Consequently d / a allows us to verify that the gradient is correct.

```
a.grad == d / a
```

```
array(True)
```

2.5.5 Training Mode and Prediction Mode

As we have seen, after we call autograd.record, MXNet logs the operations in the following block. There is one more subtle detail to be aware of. Additionally, autograd.record will change the running mode from *prediction mode* to *training mode*. We can verify this behavior by calling the is_training function.

```
print(autograd.is_training())
with autograd.record():
    print(autograd.is_training())
```

```
False
True
```

When we get to complicated deep learning models, we will encounter some algorithms where the model behaves differently during training and when we subsequently use it to make predictions. We will cover these differences in detail in later chapters.

Summary

- MXNet provides the autograd package to automate the calculation of derivatives. To use it, we first attach gradients to those variables with respect to which we desire partial derivatives. We then record the computation of our target value, execute its backward function, and access the resulting gradient via our variable's grad attribute.
- We can detach gradients to control the part of the computation that will be used in the backward function.
- The running modes of MXNet include training mode and prediction mode. We can determine the running mode by calling the is_training function.

Exercises

1. Why is the second derivative much more expensive to compute than the first derivative?
2. After running y.backward(), immediately run it again and see what happens.
3. In the control flow example where we calculate the derivative of d with respect to a, what would happen if we changed the variable a to a random vector or matrix. At this point, the result of the calculation f(a) is no longer a scalar. What happens to the result? How do we analyze this?

4. Redesign an example of finding the gradient of the control flow. Run and analyze the result.
5. Let $f(x) = \sin(x)$. Plot $f(x)$ and $\frac{df(x)}{dx}$, where the latter is computed without exploiting that $f'(x) = \cos(x)$.
6. In a second-price auction (such as in eBay or in computational advertising), the winning bidder pays the second-highest price. Compute the gradient of the final price with respect to the winning bidder's bid using `autograd`. What does the result tell you about the mechanism? If you are curious to learn more about second-price auctions, check out the paper by Edelman et al. (Edelman et al., 2007).

2.6 Probability

In some form or another, machine learning is all about making predictions. We might want to predict the *probability* of a patient suffering a heart attack in the next year, given their clinical history. In anomaly detection, we might want to assess how *likely* a set of readings from an airplane's jet engine would be, were it operating normally. In reinforcement learning, we want an agent to act intelligently in an environment. This means we need to think about the probability of getting a high reward under each of the available action. And when we build recommender systems we also need to think about probability. For example, say *hypothetically* that we worked for a large online bookseller. We might want to estimate the probability that a particular user would buy a particular book. For this we need to use the language of probability. Entire courses, majors, theses, careers, and even departments, are devoted to probability. So naturally, our goal in this section is not to teach the whole subject. Instead we hope to get you off the ground, to teach you just enough that you can start building your first deep learning models, and to give you enough of a flavor for the subject that you can begin to explore it on your own if you wish.

We have already invoked probabilities in previous sections without articulating what precisely they are or giving a concrete example. Let's get more serious now by considering the first case: distinguishing cats and dogs based on photographs. This might sound simple but it is actually a formidable challenge. To start with, the difficulty of the problem may depend on the resolution of the image.

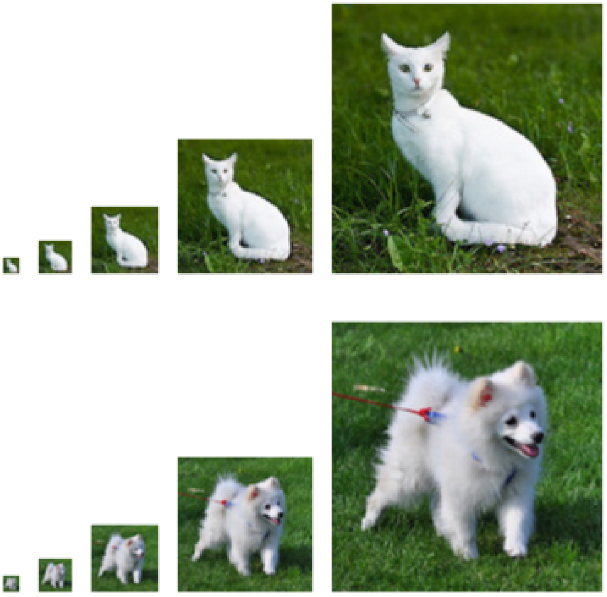


Fig. 2.6.1: Images of varying resolutions (10×10, 20×20, 40×40, 80×80, and 160×160 pixels).

As shown in Fig. 2.6.1, while it is easy for humans to recognize cats and dogs at the resolution of 160×160 pixels, it becomes challenging at 40×40 pixels and next to impossible at 10×10 pixels. In other words, our ability to tell cats and dogs apart at a large distance (and thus low resolution) might approach uninformed guessing. Probability gives us a formal way of reasoning about our level of certainty. If we are completely sure that the image depicts a cat, we say that the *probability* that the corresponding label y is "cat", denoted $P(y = \text{“cat”})$ equals 1. If we had no evidence to suggest that $y = \text{“cat”}$ or that $y = $ "dog", then we might say that the two possibilities were equally *likely* expressing this as $P(y = \text{“cat”}) = P(y = \text{“dog”}) = 0.5$. If we were reasonably confident, but not sure that the image depicted a cat, we might assign a probability $0.5 < P(y = \text{“cat”}) < 1$.

Now consider the second case: given some weather monitoring data, we want to predict the probability that it will rain in Taipei tomorrow. If it is summertime, the rain might come with probability 0.5.

In both cases, we have some value of interest. And in both cases we are uncertain about the outcome. But there is a key difference between the two cases. In this first case, the image is in fact either a dog or a cat, and we just do not know which. In the second case, the outcome may actually be a random event, if you believe in such things (and most physicists do). So probability is a flexible language for reasoning about our level of certainty, and it can be applied effectively in a broad set of contexts.

2.6.1 Basic Probability Theory

Say that we cast a die and want to know what the chance is of seeing a 1 rather than another digit. If the die is fair, all the 6 outcomes $\{1, \ldots, 6\}$ are equally likely to occur, and thus we would see a 1 in one out of six cases. Formally we state that 1 occurs with probability $\frac{1}{6}$.

For a real die that we receive from a factory, we might not know those proportions and we would need to check whether it is tainted. The only way to investigate the die is by casting it many times and recording the outcomes. For each cast of the die, we will observe a value in $\{1, \ldots, 6\}$. Given these outcomes, we want to investigate the probability of observing each outcome.

One natural approach for each value is to take the individual count for that value and to divide it by the total number of tosses. This gives us an *estimate* of the probability of a given *event*. The *law of large numbers* tell us that as the number of tosses grows this estimate will draw closer and closer to the true underlying probability. Before going into the details of what is going here, let's try it out.

To start, let's import the necessary packages.

```
%matplotlib inline
import d2l
from mxnet import np, npx
import random
npx.set_np()
```

Next, we will want to be able to cast the die. In statistics we call this process of drawing examples from probability distributions *sampling*. The distribution that assigns probabilities to a number of discrete choices is called the *multinomial distribution*. We will give a more formal definition of *distribution* later, but at a high level, think of it as just an assignment of probabilities to events. In MXNet, we can sample from the multinomial distribution via the aptly named `np.random.multinomial` function. The function can be called in many ways, but we will focus on the simplest. To draw a single sample, we simply pass in a vector of probabilities. The output of the `np.random.multinomial` function is another vector of the same length: its value at index i is the number of times the sampling outcome corresponds to i.

```
fair_probs = [1.0 / 6] * 6
np.random.multinomial(1, fair_probs)
```

```
array([0, 0, 0, 1, 0, 0], dtype=int64)
```

If you run the sampler a bunch of times, you will find that you get out random values each time. As with estimating the fairness of a die, we often want to generate many samples from the same distribution. It would be unbearably slow to do this with a Python `for` loop, so `random.multinomial` supports drawing multiple samples at once, returning an array of independent samples in any shape we might desire.

```
np.random.multinomial(10, fair_probs)
```

```
array([1, 1, 5, 1, 1, 1], dtype=int64)
```

We can also conduct, say 3, groups of experiments, where each group draws 10 samples, all at once.

```
counts = np.random.multinomial(10, fair_probs, size=3)
counts
```

```
array([[1, 2, 1, 2, 4, 0],
       [3, 2, 2, 1, 0, 2],
       [1, 2, 1, 3, 1, 2]], dtype=int64)
```

Now that we know how to sample rolls of a die, we can simulate 1000 rolls. We can then go through and count, after each of the 1000 rolls, how many times each number was rolled. Specifically, we calculate the relative frequency as the estimate of the true probability.

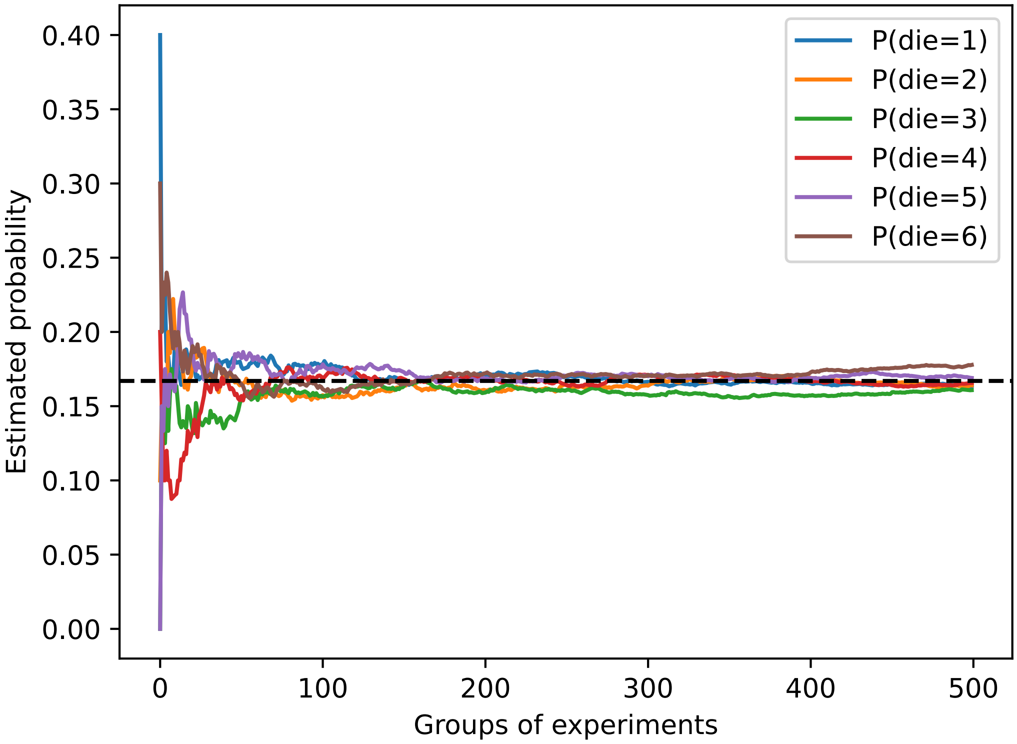
```
# Store the results as 32-bit floats for division
counts = np.random.multinomial(1000, fair_probs).astype(np.float32)
counts / 1000  # Reletive frequency as the estimate
```

```
array([0.164, 0.153, 0.181, 0.163, 0.163, 0.176])
```

Because we generated the data from a fair die, we know that each outcome has true probability $\frac{1}{6}$, roughly 0.167, so the above output estimates look good.

We can also visualize how these probabilities converge over time towards the true probability. Let's conduct 500 groups of experiments where each group draws 10 samples.

```
counts = np.random.multinomial(10, fair_probs, size=500)
cum_counts = counts.astype(np.float32).cumsum(axis=0)
estimates = cum_counts / cum_counts.sum(axis=1, keepdims=True)

d2l.set_figsize((6, 4.5))
for i in range(6):
    d2l.plt.plot(estimates[:, i].asnumpy(),
                 label=("P(die=" + str(i + 1) + ")"))
d2l.plt.axhline(y=0.167, color='black', linestyle='dashed')
d2l.plt.gca().set_xlabel('Groups of experiments')
d2l.plt.gca().set_ylabel('Estimated probability')
d2l.plt.legend();
```



Each solid curve corresponds to one of the six values of the die and gives our estimated probability that the die turns up that value as assessed after each group of experiments. The dashed black line gives the true underlying probability. As we get more data by conducting more experiments, the 6 solid curves converge towards the true probability.

Axioms of Probability Theory

When dealing with the rolls of a die, we call the set $\mathcal{S} = \{1, 2, 3, 4, 5, 6\}$ the *sample space* or *outcome space*, where each element is an *outcome*. An *event* is a set of outcomes from a given sample space. For instance, "seeing a 5" ($\{5\}$) and "seeing an odd number" ($\{1, 3, 5\}$) are both valid events of rolling a die. Note that if the outcome of a random experiment is in event $\mathcal{A}$, then event $\mathcal{A}$ has occurred. That is to say, if 3 dots faced up after rolling a die, since $3 \in \{1, 3, 5\}$, we can say that the event "seeing an odd number" has occurred.

Formally, *probability* can be thought of a function that maps a set to a real value. The probability of an event $\mathcal{A}$ in the given sample space $\mathcal{S}$, denoted as $P(\mathcal{A})$, satisfies the following properties:

- For any event $\mathcal{A}$, its probability is never negative, i.e., $P(\mathcal{A}) \geq 0$;
- Probability of the entire sample space is 1, i.e., $P(\mathcal{S}) = 1$;
- For any countable sequence of events $\mathcal{A}_1, \mathcal{A}_2, \ldots$ that are *mutually exclusive* ($\mathcal{A}_i \cap \mathcal{A}_j = \emptyset$ for all $i \neq j$), the probability that any happens is equal to the sum of their individual probabilities, i.e., $P(\bigcup_{i=1}^{\infty} \mathcal{A}_i) = \sum_{i=1}^{\infty} P(\mathcal{A}_i)$.

These are also the axioms of probability theory, proposed by Kolmogorov in 1933. Thanks to this axiom system, we can avoid any philosophical dispute on randomness; instead, we can reason rigorously with a mathematical language. For instance, by letting event $\mathcal{A}_1$ be the entire sample space and $\mathcal{A}_i = \emptyset$ for all $i > 1$, we can prove that $P(\emptyset) = 0$, i.e., the probability of an impossible event is 0.

Random Variables

In our random experiment of casting a die, we introduced the notion of a *random variable*. A random variable can be pretty much any quantity and is not deterministic. It could take one value among a set of possibilities in a random experiment. Consider a random variable X whose value is in the sample space $\mathcal{S} = \{1, 2, 3, 4, 5, 6\}$ of rolling a die. We can denote the event "seeing a 5" as $\{X = 5\}$ or $X = 5$, and its probability as $P(\{X = 5\})$ or $P(X = 5)$. By $P(X = a)$, we make a distinction between the random variable X and the values (e.g., a) that X can take. However, such pedantry results in a cumbersome notation. For a compact notation, on one hand, we can just denote $P(X)$ as the *distribution* over the random variable X: the distribution tells us the probability that X takes any value. On the other hand, we can simply write $P(a)$ to denote the probability that a random variable takes the value a. Since an event in probability theory is a set of outcomes from the sample space, we can specify a range of values for a random variable to take. For example, $P(1 \leq X \leq 3)$ denotes the probability of the event $\{1 \leq X \leq 3\}$, which means $\{X = 1, 2, \text{or}, 3\}$. Equivalently, $P(1 \leq X \leq 3)$ represents the probability that the random variable X can take a value from $\{1, 2, 3\}$.

Note that there is a subtle difference between *discrete* random variables, like the sides of a die, and *continuous* ones, like the weight and the height of a person. There is little point in asking whether two people have exactly the same height. If we take precise enough measurements you will find that no two people on the planet have the exact same height. In fact, if we take a fine enough measurement, you will not have the same height when you wake up and when you go to sleep. So there is no purpose in asking about the probability that someone is

$1.801392782910287192101967405274862O2$ meters tall. Given the world population of humans the probability is virtually 0. It makes more sense in this case to ask whether someone's height falls into a given interval, say between 1.79 and 1.81 meters. In these cases we quantify the likelihood that we see a value as a *density*. The height of exactly 1.80 meters has no probability, but nonzero density. In the interval between any two different heights we have nonzero probability. In the rest of this section, we consider probability in discrete space. For probability over continuous random variables, you may refer to Section 17.6.

2.6.2 Dealing with Multiple Random Variables

Very often, we will want to consider more than one random variable at a time. For instance, we may want to model the relationship between diseases and symptoms. Given a disease and a symptom, say "flu" and "cough", either may or may not occur in a patient with some probability. While we hope that the probability of both would be close to zero, we may want to estimate these probabilities and their relationships to each other so that we may apply our inferences to effect better medical care.

As a more complicated example, images contain millions of pixels, thus millions of random variables. And in many cases images will come with a label, identifying objects in the image. We can also think of the label as a random variable. We can even think of all the metadata as random variables such as location, time, aperture, focal length, ISO, focus distance, and camera type. All of these are random variables that occur jointly. When we deal with multiple random variables, there are several quantities of interest.

Joint Probability

The first is called the *joint probability* $P(A = a, B = b)$. Given any values a and b, the joint probability lets us answer, what is the probability that $A = a$ and $B = b$ simultaneously? Note that for any values a and b, $P(A = a, B = b) \leq P(A = a)$. This has to be the case, since for $A = a$ and $B = b$ to happen, $A = a$ has to happen *and* $B = b$ also has to happen (and vice versa). Thus, $A = a$ and $B = b$ cannot be more likely than $A = a$ or $B = b$ individually.

Conditional Probability

This brings us to an interesting ratio: $0 \leq \frac{P(A=a,B=b)}{P(A=a)} \leq 1$. We call this ratio a *conditional probability* and denote it by $P(B = b \mid A = a)$: it is the probability of $B = b$, provided that $A = a$ has occurred.

Bayes' theorem

Using the definition of conditional probabilities, we can derive one of the most useful and celebrated equations in statistics: *Bayes' theorem*. It goes as follows. By construction, we have the *multiplication rule* that $P(A, B) = P(B \mid A)P(A)$. By symmetry, this also holds for $P(A, B) = P(A \mid B)P(B)$. Assume that $P(B) > 0$. Solving for one of the conditional variables we get

$$P(A \mid B) = \frac{P(B \mid A)P(A)}{P(B)}. \tag{2.6.1}$$

Note that here we use the more compact notation where $P(A, B)$ is a *joint distribution* and $P(A \mid B)$ is a *conditional distribution*. Such distributions can be evaluated for particular values $A = a, B = b$.

Marginalization

Bayes' theorem is very useful if we want to infer one thing from the other, say cause and effect, but we only know the properties in the reverse direction, as we will see later in this section. One important operation that we need, to make this work, is *marginalization*. It is the operation of determining $P(B)$ from $P(A, B)$. We can see that the probability of B amounts to accounting for all possible choices of A and aggregating the joint probabilities over all of them:

$$P(B) = \sum_{A} P(A, B), \tag{2.6.2}$$

which is also known as the *sum rule*. The probability or distribution as a result of marginalization is called a *marginal probability* or a *marginal distribution*.

Independence

Another useful property to check for is *dependence* vs. *independence*. Two random variables A and B are independent means that the occurrence of one event of A does not reveal any information about the occurrence of an event of B. In this case $P(B \mid A) = P(B)$. Statisticians typically express this as $A \perp B$. From Bayes' theorem, it follows immediately that also $P(A \mid B) = P(A)$. In all the other cases we call A and B dependent. For instance, two successive rolls of a die are independent. In contrast, the position of a light switch and the brightness in the room are not (they are not perfectly deterministic, though, since we could always have a broken light bulb, power failure, or a broken switch).

Since $P(A \mid B) = \frac{P(A,B)}{P(B)} = P(A)$ is equivalent to $P(A, B) = P(A)P(B)$, two random variables are independent if and only if their joint distribution is the product of their individual distributions. Likewise, two random variables A and B are *conditionally independent* given another random variable C if and only if $P(A, B \mid C) = P(A \mid C)P(B \mid C)$. This is expressed as $A \perp B \mid C$.

Application

Let's put our skills to the test. Assume that a doctor administers an AIDS test to a patient. This test is fairly accurate and it fails only with 1% probability if the patient is healthy but reporting him as diseased. Moreover, it never fails to detect HIV if the patient actually has it. We use D_1 to indicate the diagnosis (1 if positive and 0 if negative) and H to denote the HIV status (1 if positive and 0 if negative). Table 2.6.1 lists such conditional probability.

Table 2.6.1: Conditional probability of $P(D_1 \mid H)$.

Conditional probability	$H = 1$	$H = 0$
$P(D_1 = 1 \mid H)$	1	0.01
$P(D_1 = 0 \mid H)$	0	0.99

Note that the column sums are all 1 (but the row sums are not), since the conditional probability needs to sum up to 1, just like the probability. Let's work out the probability of the patient having AIDS if the test comes back positive, i.e., $P(H = 1 \mid D_1 = 1)$. Obviously this is going to depend on how common the disease is, since it affects the number of false alarms. Assume that the population is quite healthy, e.g., $P(H = 1) = 0.0015$. To apply Bayes' Theorem, we need to apply marginalization and the multiplication rule to determine

$$\begin{aligned} & P(D_1 = 1) \\ =& P(D_1 = 1, H = 0) + P(D_1 = 1, H = 1) \\ =& P(D_1 = 1 \mid H = 0) P(H = 0) + P(D_1 = 1 \mid H = 1) P(H = 1) \\ =& 0.011485. \end{aligned} \tag{2.6.3}$$

Thus, we get

$$\begin{aligned} & P(H = 1 \mid D_1 = 1) \\ =& \frac{P(D_1 = 1 \mid H = 1) P(H = 1)}{P(D_1 = 1)}. \\ =& 0.1306 \end{aligned} \tag{2.6.4}$$

In other words, there is only a 13.06% chance that the patient actually has AIDS, despite using a very accurate test. As we can see, probability can be quite counterintuitive.

What should a patient do upon receiving such terrifying news? Likely, the patient would ask the physician to administer another test to get clarity. The second test has different characteristics and it is not as good as the first one, as shown in Table 2.6.2.

Table 2.6.2: Conditional probability of $P(D_2 \mid H)$.

Conditional probability	$H = 1$	$H = 0$
$P(D_2 = 1 \mid H)$	0.98	0.03
$P(D_2 = 0 \mid H)$	0.02	0.97

Unfortunately, the second test comes back positive, too. Let's work out the requisite probabilities to invoke Bayes' Theorem by assuming the conditional independence:

$$\begin{aligned} & P(D_1 = 1, D_2 = 1 \mid H = 0) \\ =& P(D_1 = 1 \mid H = 0) P(D_2 = 1 \mid H = 0) \\ =& 0.0003, \end{aligned} \tag{2.6.5}$$

$$\begin{aligned} & P(D_1 = 1, D_2 = 1 \mid H = 1) \\ =& P(D_1 = 1 \mid H = 1) P(D_2 = 1 \mid H = 1) \\ =& 0.98. \end{aligned} \tag{2.6.6}$$

Now we can apply marginalization and the multiplication rule:

$$\begin{aligned} & P(D_1 = 1, D_2 = 1) \\ =& P(D_1 = 1, D_2 = 1, H = 0) + P(D_1 = 1, D_2 = 1, H = 1) \\ =& P(D_1 = 1, D_2 = 1 \mid H = 0) P(H = 0) + P(D_1 = 1, D_2 = 1 \mid H = 1) P(H = 1) \\ =& 0.00176955. \end{aligned} \tag{2.6.7}$$

In the end, the probability of the patient having AIDS given both positive tests is

$$\begin{aligned} & P(H = 1 \mid D_1 = 1, D_2 = 1) \\ =& \frac{P(D_1 = 1, D_2 = 1 \mid H = 1) P(H = 1)}{P(D_1 = 1, D_2 = 1)} \\ =& 0.8307. \end{aligned} \tag{2.6.8}$$

That is, the second test allowed us to gain much higher confidence that not all is well. Despite the second test being considerably less accurate than the first one, it still significantly improved our estimate.

2.6.3 Expectation and Variance

To summarize key characteristics of probability distributions, we need some measures. The *expectation* (or average) of the random variable X is denoted as

$$E[X] = \sum_x x P(X = x). \tag{2.6.9}$$

When the input of a function $f(x)$ is a random variable drawn from the distribution P with different values x, the expectation of $f(x)$ is computed as

$$E_{x \sim P}[f(x)] = \sum_x f(x) P(x). \tag{2.6.10}$$

In many cases we want to measure by how much the random variable X deviates from its expectation. This can be quantified by the variance

$$\mathrm{Var}[X] = E\left[(X - E[X])^2\right] = E[X^2] - E[X]^2. \tag{2.6.11}$$

Its square root is called the *standard deviation*. The variance of a function of a random variable measures by how much the function deviates from the expectation of the function, as different values x of the random variable are sampled from its distribution:

$$\mathrm{Var}[f(x)] = E\left[\left(f(x) - E[f(x)]\right)^2\right]. \tag{2.6.12}$$

Summary

- We can use MXNet to sample from probability distributions.
- We can analyze multiple random variables using joint distribution, conditional distribution, Bayes' theorem, marginalization, and independence assumptions.
- Expectation and variance offer useful measures to summarize key characteristics of probability distributions.

Exercises

1. We conducted $m = 500$ groups of experiments where each group draws $n = 10$ samples. Vary m and n. Observe and analyze the experimental results.
2. Given two events with probability $P(\mathcal{A})$ and $P(\mathcal{B})$, compute upper and lower bounds on $P(\mathcal{A} \cup \mathcal{B})$ and $P(\mathcal{A} \cap \mathcal{B})$. (Hint: display the situation using a Venn Diagram[46].)
3. Assume that we have a sequence of random variables, say A, B, and C, where B only depends on A, and C only depends on B, can you simplify the joint probability $P(A, B, C)$? (Hint: this is a Markov Chain[47].)
4. In Section 2.6.2, the first test is more accurate. Why not just run the first test a second time?

[46] https://en.wikipedia.org/wiki/Venn_diagram

[47] https://en.wikipedia.org/wiki/Markov_chain

2.7 Documentation

Due to constraints on the length of this book, we cannot possibly introduce every single MXNet function and class (and you probably would not want us to). The API documentation and additional tutorials and examples provide plenty of documentation beyond the book. In this section we provide you with some guidance to exploring the MXNet API.

2.7.1 Finding All the Functions and Classes in a Module

In order to know which functions and classes can be called in a module, we invoke the `dir` function. For instance, we can query all properties in the `np.random` module as follows:

```python
from mxnet import np
print(dir(np.random))
```

```
['__all__', '__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__', '_
↪_package__', '__spec__', '_mx_nd_np', 'absolute_import', 'choice', 'multinomial', 'normal',
↪ 'rand', 'randint', 'shuffle', 'uniform']
```

Generally, we can ignore functions that start and end with `__` (special objects in Python) or functions that start with a single `_`(usually internal functions). Based on the remaining function or attribute names, we might hazard a guess that this module offers various methods for generating random numbers, including sampling from the uniform distribution (`uniform`), normal distribution (`normal`), and multinomial distribution (`multinomial`).

2.7.2 Finding the Usage of Specific Functions and Classes

For more specific instructions on how to use a given function or class, we can invoke the `help` function. As an example, let's explore the usage instructions for `ndarray`'s `ones_like` function.

```python
help(np.ones_like)
```

```
Help on function ones_like in module mxnet.numpy:

ones_like(a)
    Return an array of ones with the same shape and type as a given array.

    Parameters
    ----------
    a : ndarray
        The shape and data-type of a define these same attributes of
        the returned array.

    Returns
    -------
    out : ndarray
        Array of ones with the same shape and type as a.

    Examples
    --------
    >>> x = np.arange(6)
    >>> x = x.reshape((2, 3))
    >>> x
    array([[0., 1., 2.],
           [3., 4., 5.]])
    >>> np.ones_like(x)
    array([[1., 1., 1.],
           [1., 1., 1.]])

    >>> y = np.arange(3, dtype=float)
    >>> y
    array([0., 1., 2.], dtype=float64)
    >>>
    >>> np.ones_like(y)
    array([1., 1., 1.], dtype=float64)
```

From the documentation, we can see that the `ones_like` function creates a new array with the same shape as the supplied `ndarray` and sets all the elements to `1`. Whenever possible, you should run a quick test to confirm your interpretation:

```
x = np.array([[0, 0, 0], [2, 2, 2]])
np.ones_like(x)
```

```
array([[1., 1., 1.],
       [1., 1., 1.]])
```

In the Jupyter notebook, we can use `?` to display the document in another window. For example, `np.random.uniform?` will create content that is almost identical to `help(np.random.uniform)`, displaying it in a new browser window. In addition, if we use two question marks, such as `np.random.uniform??`, the code implementing the function will also be displayed.

2.7.3 API Documentation

For further details on the API details check the MXNet website at http://mxnet.apache.org/. You can find the details under the appropriate headings (also for programming languages other than Python).

Summary

- The official documentation provides plenty of descriptions and examples that are beyond this book.
- We can look up documentation for the usage of MXNet API by calling the `dir` and `help` functions, or checking the MXNet website.

Exercises

1. Look up `ones_like` and `autograd` on the MXNet website.
2. What are all the possible outputs after running `np.random.choice(4, 2)`?
3. Can you rewrite `np.random.choice(4, 2)` by using the `np.random.randint` function?

3 | Linear Neural Networks

Before we get into the details of deep neural networks, we need to cover the basics of neural network training. In this chapter, we will cover the entire training process, including defining simple neural network architectures, handling data, specifying a loss function, and training the model. In order to make things easier to grasp, we begin with the simplest concepts. Fortunately, classic statistical learning techniques such as linear and logistic regression can be cast as *shallow* neural networks. Starting from these classic algorithms, we will introduce you to the basics, providing the basis for more complex techniques such as softmax regression (introduced at the end of this chapter) and multilayer perceptrons (introduced in the next chapter).

3.1 Linear Regression

Regression refers to a set of methods for modeling the relationship between data points $\mathbf{x}$ and corresponding real-valued targets y. In the natural sciences and social sciences, the purpose of regression is most often to *characterize* the relationship between the inputs and outputs. Machine learning, on the other hand, is most often concerned with *prediction*.

Regression problems pop up whenever we want to predict a numerical value. Common examples include predicting prices (of homes, stocks, etc.), predicting length of stay (for patients in the hospital), demand forecasting (for retail sales), among countless others. Not every prediction problem is a classic *regression* problem. In subsequent sections, we will introduce classification problems, where the goal is to predict membership among a set of categories.

3.1.1 Basic Elements of Linear Regression

Linear regression may be both the simplest and most popular among the standard tools to regression. Dating back to the dawn of the 19th century, linear regression flows from a few simple assumptions. First, we assume that the relationship between the *features* $\mathbf{x}$ and targets y is linear, i.e., that y can be expressed as a weighted sum of the inputs $\mathbf{x}$, give or take some noise on the observations. Second, we assume that any noise is well-behaved (following a Gaussian distribution). To motivate the approach, let's start with a running example. Suppose that we wish to estimate the prices of houses (in dollars) based on their area (in square feet) and age (in years).

To actually fit a model for predicting house prices, we would need to get our hands on a dataset consisting of sales for which we know the sale price, area and age for each home. In the terminology of machine learning, the dataset is called a *training data* or *training set*, and each row (here the data corresponding to one sale) is called an *instance* or *example*. The thing we are trying to predict (here, the price) is called a *target* or *label*. The variables (here *age* and *area*) upon which the predictions are based are called *features* or *covariates*.

Typically, we will use n to denote the number of examples in our dataset. We index the samples by i, denoting each input data point as $x^{(i)} = [x_1^{(i)}, x_2^{(i)}]$ and the corresponding label as $y^{(i)}$.

Linear Model

The linearity assumption just says that the target (price) can be expressed as a weighted sum of the features (area and age):

$$\mathrm{price} = w_{\mathrm{area}} \cdot \mathrm{area} + w_{\mathrm{age}} \cdot \mathrm{age} + b. \tag{3.1.1}$$

Here, w_{area} and w_{age} are called *weights*, and b is called a *bias* (also called an *offset* or *intercept*). The weights determine the influence of each feature on our prediction and the bias just says what value the predicted price should take when all of the features take value 0. Even if we will never see any homes with zero area, or that are precisely zero years old, we still need the intercept or else we will limit the expressivity of our linear model.

Given a dataset, our goal is to choose the weights w and bias b such that on average, the predictions made according our model best fit the true prices observed in the data.

In disciplines where it is common to focus on datasets with just a few features, explicitly expressing models long-form like this is common. In ML, we usually work with high-dimensional datasets, so it is more convenient to employ linear algebra notation. When our inputs consist of d features, we express our prediction $\hat{y}$ as

$$\hat{y} = w_1 \cdot x_1 + ... + w_d \cdot x_d + b. \tag{3.1.2}$$

Collecting all features into a vector $\mathbf{x}$ and all weights into a vector $\mathbf{w}$, we can express our model compactly using a dot product:

$$\hat{y} = \mathbf{w}^T \mathbf{x} + b. \tag{3.1.3}$$

Here, the vector $\mathbf{x}$ corresponds to a single data point. We will often find it convenient to refer to our entire dataset via the *design matrix* X. Here, X contains one row for every example and one column for every feature.

For a collection of data points $\mathbf{X}$, the predictions $\hat{\mathbf{y}}$ can be expressed via the matrix-vector product:

$$\hat{\mathbf{y}} = \mathbf{X}\mathbf{w} + b. \tag{3.1.4}$$

Given a training dataset X and corresponding (known) targets $\mathbf{y}$, the goal of linear regression is to find the *weight* vector w and bias term b that given some a new data point $\mathbf{x}_i$, sampled from the same distribution as the training data will (in expectation) predict the target y_i with the lowest error.

Even if we believe that the best model for predicting y given $\mathbf{x}$ is linear, we would not expect to find real-world data where y_i exactly equals $\mathbf{w}^T\mathbf{x} + b$ for all points $(\mathbf{x}, y)$. For example, whatever instruments we use to observe the features X and labels $\mathbf{y}$ might suffer small amount of measurement error. Thus, even when we are confident that the underlying relationship is linear, we will incorporate a noise term to account for such errors.

Before we can go about searching for the best parameters w and b, we will need two more things: (i) a quality measure for some given model; and (ii) a procedure for updating the model to improve its quality.

Loss Function

Before we start thinking about how *to fit* our model, we need to determine a measure of *fitness*. The *loss function* quantifies the distance between the *real* and *predicted* value of the target. The loss will usually be a non-negative number where smaller values are better and perfect predictions incur a loss of 0. The most popular loss function in regression problems is the sum of squared errors. When our prediction for some example i is $\hat{y}^{(i)}$ and the corresponding true label is $y^{(i)}$, the squared error is given by:

$$l^{(i)}(\mathbf{w}, b) = \frac{1}{2} \left(\hat{y}^{(i)} - y^{(i)}\right)^2. \tag{3.1.5}$$

The constant $1/2$ makes no real difference but will prove notationally convenient, cancelling out when we take the derivative of the loss. Since the training dataset is given to us, and thus out of our control, the empirical error is only a function of the model parameters. To make things more concrete, consider the example below where we plot a regression problem for a one-dimensional case as shown in Fig. 3.1.1.

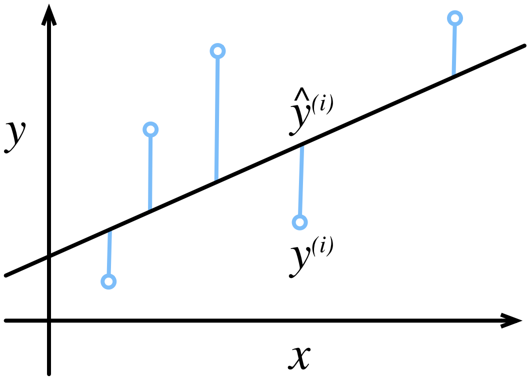


Fig. 3.1.1: Fit data with a linear model.

Note that large differences between estimates $\hat{y}^{(i)}$ and observations $y^{(i)}$ lead to even larger contributions to the loss, due to the quadratic dependence. To measure the quality of a model on the entire dataset, we simply average (or equivalently, sum) the losses on the training set.

$$L(\mathbf{w}, b) = \frac{1}{n}\sum_{i=1}^n l^{(i)}(\mathbf{w}, b) = \frac{1}{n} \sum_{i=1}^n \frac{1}{2}\left(\mathbf{w}^\top \mathbf{x}^{(i)} + b - y^{(i)}\right)^2. \tag{3.1.6}$$

When training the model, we want to find parameters $(\mathbf{w}^*, b^*)$ that minimize the total loss across all training samples:

$$\mathbf{w}^*, b^* = \operatorname*{argmin}_{\mathbf{w}, b}\ L(\mathbf{w}, b). \tag{3.1.7}$$

Analytic Solution

Linear regression happens to be an unusually simple optimization problem. Unlike most other models that we will encounter in this book, linear regression can be solved analytically by applying a simple formula, yielding a global optimum. To start, we can subsume the bias b into the parameter $\mathbf{w}$ by appending a column to the design matrix consisting of all $1s$. Then our prediction problem is to minimize $||\mathbf{y} - X\mathbf{w}||$. Because this expression has a quadratic form, it is convex, and so long as the problem is not degenerate (our features are linearly independent), it is strictly convex.

Thus there is just one critical point on the loss surface and it corresponds to the global minimum. Taking the derivative of the loss with respect to $\mathbf{w}$ and setting it equal to 0 yields the analytic solution:

$$\mathbf{w}^* = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T y. \tag{3.1.8}$$

While simple problems like linear regression may admit analytic solutions, you should not get used to such good fortune. Although analytic solutions allow for nice mathematical analysis, the requirement of an analytic solution is so restrictive that it would exclude all of deep learning.

Gradient descent

Even in cases where we cannot solve the models analytically, and even when the loss surfaces are high-dimensional and nonconvex, it turns out that we can still train models effectively in practice. Moreover, for many tasks, these difficult-to-optimize models turn out to be so much better that figuring out how to train them ends up being well worth the trouble.

The key technique for optimizing nearly any deep learning model, and which we will call upon throughout this book, consists of iteratively reducing the error by updating the parameters in the direction that incrementally lowers the loss function. This algorithm is called *gradient descent*. On convex loss surfaces, it will eventually converge to a global minimum, and while the same cannot be said for nonconvex surfaces, it will at least lead towards a (hopefully good) local minimum.

The most naive application of gradient descent consists of taking the derivative of the true loss, which is an average of the losses computed on every single example in the dataset. In practice, this can be extremely slow. We must pass over the entire dataset before making a single update. Thus, we will often settle for sampling a random minibatch of examples every time we need to computer the update, a variant called *stochastic gradient descent*.

In each iteration, we first randomly sample a minibatch $\mathcal{B}$ consisting of a fixed number of training data examples. We then compute the derivative (gradient) of the average loss on the mini batch with regard to the model parameters. Finally, we multiply the gradient by a predetermined step size $\eta > 0$ and subtract the resulting term from the current parameter values.

We can express the update mathematically as follows (∂ denotes the partial derivative) :

$$(\mathbf{w}, b) \leftarrow (\mathbf{w}, b) - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \partial_{(\mathbf{w}, b)} l^{(i)}(\mathbf{w}, b). \tag{3.1.9}$$

To summarize, steps of the algorithm are the following: (i) we initialize the values of the model parameters, typically at random; (ii) we iteratively sample random batches from the the data (many times), updating the parameters in the direction of the negative gradient.

For quadratic losses and linear functions, we can write this out explicitly as follows: Note that $\mathbf{w}$ and $\mathbf{x}$ are vectors. Here, the more elegant vector notation makes the math much more readable than expressing things in terms of coefficients, say $w_1, w_2, \ldots, w_d$.

$$\begin{aligned}
\mathbf{w} &\leftarrow \mathbf{w} - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \partial_{\mathbf{w}} l^{(i)}(\mathbf{w}, b) &&= w - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \mathbf{x}^{(i)} \left(\mathbf{w}^\top \mathbf{x}^{(i)} + b - y^{(i)}\right),\\
b &\leftarrow b - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \partial_b l^{(i)}(\mathbf{w}, b) &&= b - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \left(\mathbf{w}^\top \mathbf{x}^{(i)} + b - y^{(i)}\right).
\end{aligned} \tag{3.1.10}$$

In the above equation, $|\mathcal{B}|$ represents the number of examples in each minibatch (the *batch size*) and η denotes the *learning rate*. We emphasize that the values of the batch size and learning rate are manually pre-specified and not typically learned through model training. These parameters that are tunable but not updated in the training loop are called *hyper-parameters*. *Hyperparameter tuning* is the process by which these are chosen, and typically requires that we adjust the hyperparameters based on the results of the inner (training) loop as assessed on a separate *validation* split of the data.

After training for some predetermined number of iterations (or until some other stopping criteria is met), we record the estimated model parameters, denoted $\hat{\mathbf{w}}, \hat{b}$ (in general the "hat" symbol denotes estimates). Note that even if our function is truly linear and noiseless, these parameters will not be the exact minimizers of the loss because, although the algorithm converges slowly towards a local minimum it cannot achieve it exactly in a finite number of steps.

Linear regression happens to be a convex learning problem, and thus there is only one (global) minimum. However, for more complicated models, like deep networks, the loss surfaces contain many minima. Fortunately, for reasons that are not yet fully understood, deep learning practitioners seldom struggle to find parameters that minimize the loss *on training data*. The more formidable task is to find parameters that will achieve low loss on data that we have not seen before, a challenge called *generalization*. We return to these topics throughout the book.

Making Predictions with the Learned Model

Given the learned linear regression model $\hat{\mathbf{w}}^\top x + \hat{b}$, we can now estimate the price of a new house (not contained in the training data) given its area x_1 and age (year) x_2. Estimating targets given features is commonly called *prediction* and *inference*.

We will try to stick with *prediction* because calling this step *inference*, despite emerging as standard jargon in deep learning, is somewhat of a misnomer. In statistics, *inference* more often denotes estimating parameters based on a dataset. This misuse of terminology is a common source of confusion when deep learning practitioners talk to statisticians.

Vectorization for Speed

When training our models, we typically want to process whole minibatches of examples simultaneously. Doing this efficiently requires that we vectorize the calculations and leverage fast linear algebra libraries rather than writing costly for-loops in Python.

To illustrate why this matters so much, we can consider two methods for adding vectors. To start we instantiate two 100000-dimensional vectors containing all ones. In one method we will loop over the vectors with a Python `for` loop. In the other method we will rely on a single call to `np`.

```
%matplotlib inline
import d2l
import math
from mxnet import np
import time

n = 10000
a = np.ones(n)
b = np.ones(n)
```

Since we will benchmark the running time frequently in this book, let's define a timer (hereafter accessed via the d2l package to track the running time.

```
# Saved in the d2l package for later use
class Timer(object):
    """Record multiple running times."""
    def __init__(self):
        self.times = []
        self.start()

    def start(self):
        # Start the timer
        self.start_time = time.time()

    def stop(self):
        # Stop the timer and record the time in a list
        self.times.append(time.time() - self.start_time)
        return self.times[-1]

    def avg(self):
        # Return the average time
        return sum(self.times)/len(self.times)

    def sum(self):
        # Return the sum of time
        return sum(self.times)

    def cumsum(self):
        # Return the accumuated times
        return np.array(self.times).cumsum().tolist()
```

Now we can benchmark the workloads. First, we add them, one coordinate at a time, using a for loop.

```
timer = Timer()
c = np.zeros(n)
for i in range(n):
    c[i] = a[i] + b[i]
'%.5f sec' % timer.stop()
```

```
'3.51575 sec'
```

Alternatively, we rely on np to compute the elementwise sum:

```
timer.start()
d = a + b
'%.5f sec' % timer.stop()
```

```
'0.00023 sec'
```

You probably noticed that the second method is dramatically faster than the first. Vectorizing code often yields order-of-magnitude speedups. Moreover, we push more of the math to the library and need not write as many calculations ourselves, reducing the potential for errors.

3.1.2 The Normal Distribution and Squared Loss

While you can already get your hands dirty using only the information above, in the following section we can more formally motivate the square loss objective via assumptions about the distribution of noise.

Recall from the above that the squared loss $l(y, \hat{y}) = \frac{1}{2}(y - \hat{y})^2$ has many convenient properties. These include a simple derivative $\partial_{\hat{y}} l(y, \hat{y}) = (\hat{y} - y)$.

As we mentioned earlier, linear regression was invented by Gauss in 1795, who also discovered the normal distribution (also called the *Gaussian*). It turns out that the connection between the normal distribution and linear regression runs deeper than common parentage. To refresh your memory, the probability density of a normal distribution with mean μ and variance σ^2 is given as follows:

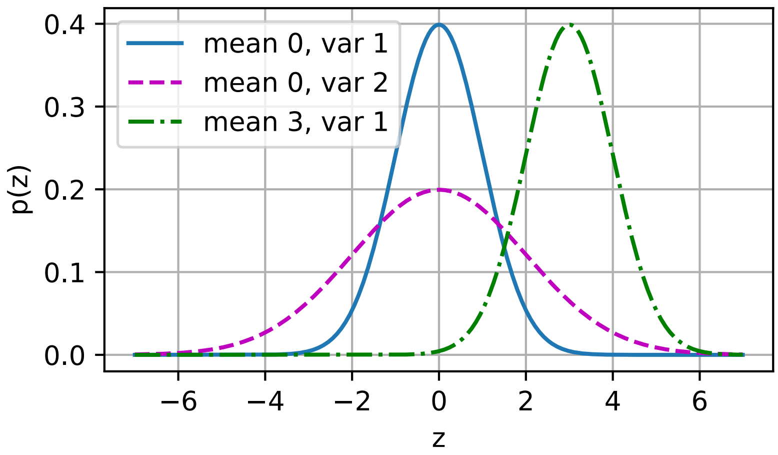
$$p(z) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2\sigma^2}(z-\mu)^2\right). \tag{3.1.11}$$

Below we define a Python function to compute the normal distribution.

```
x = np.arange(-7, 7, 0.01)

def normal(z, mu, sigma):
    p = 1 / math.sqrt(2 * math.pi * sigma**2)
    return p * np.exp(- 0.5 / sigma**2 * (z - mu)**2)
```

We can now visualize the normal distributions.

```
# Mean and variance pairs
parameters = [(0, 1), (0, 2), (3, 1)]
d2l.plot(x, [normal(x, mu, sigma) for mu, sigma in parameters], xlabel='z',
         ylabel='p(z)', figsize=(4.5, 2.5),
         legend=['mean %d, var %d' % (mu, sigma) for mu, sigma in parameters])
```



As you can see, changing the mean corresponds to a shift along the *x axis*, and increasing the variance spreads the distribution out, lowering its peak.

One way to motivate linear regression with the mean squared error loss function is to formally assume that observations arise from noisy observations, where the noise is normally distributed as follows

$$y = \mathbf{w}^\top \mathbf{x} + b + \epsilon \text{ where } \epsilon \sim \mathcal{N}(0, \sigma^2). \tag{3.1.12}$$

Thus, we can now write out the *likelihood* of seeing a particular y for a given $\mathbf{x}$ via

$$p(y|\mathbf{x}) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2\sigma^2}(y - \mathbf{w}^\top \mathbf{x} - b)^2\right). \tag{3.1.13}$$

Now, according to the *maximum likelihood principle*, the best values of b and $\mathbf{w}$ are those that maximize the *likelihood* of the entire dataset:

$$P(Y \mid X) = \prod_{i=1}^{n} p(y^{(i)}|\mathbf{x}^{(i)}). \tag{3.1.14}$$

Estimators chosen according to the *maximum likelihood principle* are called *Maximum Likelihood Estimators* (MLE). While, maximizing the product of many exponential functions, might look difficult, we can simplify things significantly, without changing the objective, by maximizing the log of the likelihood instead. For historical reasons, optimizations are more often expressed as minimization rather than maximization. So, without changing anything we can minimize the *Negative Log-Likelihood (NLL)* $-\log p(\mathbf{y}|\mathbf{X})$. Working out the math gives us:

$$-\log p(\mathbf{y}|\mathbf{X}) = \sum_{i=1}^{n} \frac{1}{2} \log(2\pi\sigma^2) + \frac{1}{2\sigma^2}\left(y^{(i)} - \mathbf{w}^\top \mathbf{x}^{(i)} - b\right)^2. \tag{3.1.15}$$

Now we just need one more assumption: that σ is some fixed constant. Thus we can ignore the first term because it does not depend on $\mathbf{w}$ or b. Now the second term is identical to the squared error objective introduced earlier, but for the multiplicative constant $\frac{1}{\sigma^2}$. Fortunately, the solution does not depend on σ. It follows that minimizing squared error is equivalent to maximum likelihood estimation of a linear model under the assumption of additive Gaussian noise.

3.1.3 From Linear Regression to Deep Networks

So far we only talked about linear functions. While neural networks cover a much richer family of models, we can begin thinking of the linear model as a neural network by expressing it the language of neural networks. To begin, let's start by rewriting things in a 'layer' notation.

Neural Network Diagram

Deep learning practitioners like to draw diagrams to visualize what is happening in their models. In Fig. 3.1.2, we depict our linear model as a neural network. Note that these diagrams indicate the connectivity pattern (here, each input is connected to the output) but not the values taken by the weights or biases.

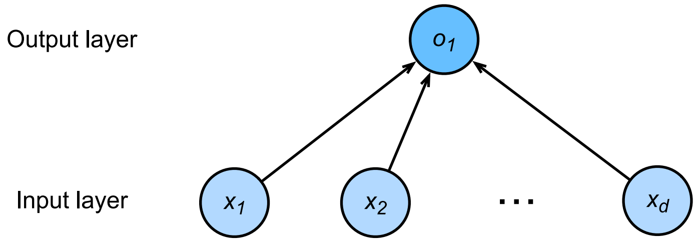


Fig. 3.1.2: Linear regression is a single-layer neural network.

Because there is just a single computed neuron (node) in the graph (the input values are not computed but given), we can think of linear models as neural networks consisting of just a single artificial neuron. Since for this model, every input is connected to every output (in this case there is only one output!), we can regard this transformation as a *fully-connected layer*, also commonly called a *dense layer*. We will talk a lot more about networks composed of such layers in the next chapter on multilayer perceptrons.

Biology

Although linear regression (invented in 1795) predates computational neuroscience, so it might seem anachronistic to describe linear regression as a neural network. To see why linear models were a natural place to begin when the cyberneticists/neurophysiologists Warren McCulloch and Walter Pitts looked when they began to develop models of artificial neurons, consider the cartoonish picture of a biological neuron in Fig. 3.1.3, consisting of *dendrites* (input terminals), the *nucleus* (CPU), the *axon* (output wire), and the *axon terminals* (output terminals), enabling connections to other neurons via *synapses*.

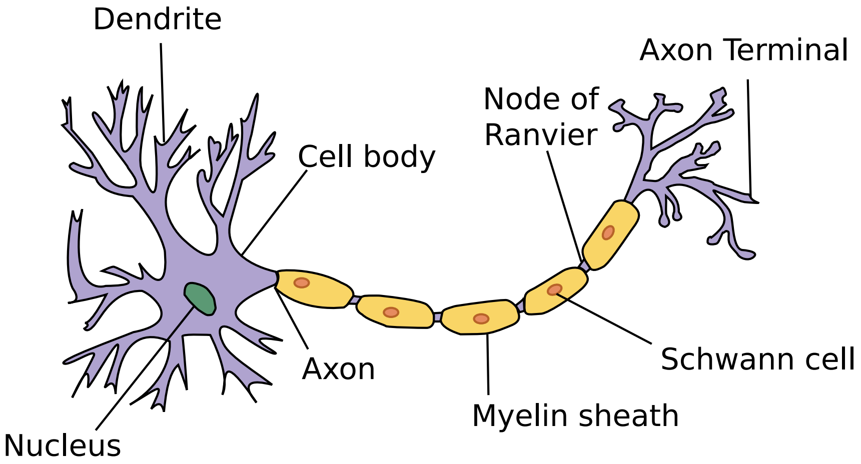


Fig. 3.1.3: The real neuron

Information x_i arriving from other neurons (or environmental sensors such as the retina) is received in the dendrites. In particular, that information is weighted by *synaptic weights* w_i determining the effect of the inputs (e.g., activation or inhibition via the product $x_i w_i$). The weighted inputs arriving from multiple sources are aggregated in the nucleus as a weighted sum $y = \sum_i x_i w_i + b$, and this information is then sent for further processing in the axon y, typically after some nonlinear processing via $\sigma(y)$. From there it either reaches its destination (e.g., a muscle) or is fed into another neuron via its dendrites.

Certainly, the high-level idea that many such units could be cobbled together with the right connectivity and right learning algorithm, to produce far more interesting and complex behavior than any one neuron along could express owes to our study of real biological neural systems.

At the same time, most research in deep learning today draws little direct inspiration in neuroscience. We invoke Stuart Russell and Peter Norvig who, in their classic AI text book *Artificial Intelligence: A Modern Approach* (Russell & Norvig, 2016), pointed out that although airplanes might have been *inspired* by birds, orninthology has not been the primary driver of aeronautics innovation for some centuries. Likewise, inspiration in deep learning these days comes in equal or greater measure from mathematics, statistics, and computer science.

Summary

- Key ingredients in a machine learning model are training data, a loss function, an optimization algorithm, and quite obviously, the model itself.
- Vectorizing makes everything better (mostly math) and faster (mostly code).
- Minimizing an objective function and performing maximum likelihood can mean the same thing.
- Linear models are neural networks, too.

Exercises

1. Assume that we have some data $x_1, \ldots, x_n \in \mathbb{R}$. Our goal is to find a constant b such that $\sum_i (x_i - b)^2$ is minimized.
    - Find a closed-form solution for the optimal value of b.
    - How does this problem and its solution relate to the normal distribution?
2. Derive the closed-form solution to the optimization problem for linear regression with squared error. To keep things simple, you can omit the bias b from the problem (we can do this in principled fashion by adding one column to X consisting of all ones).
    - Write out the optimization problem in matrix and vector notation (treat all the data as a single matrix, all the target values as a single vector).
    - Compute the gradient of the loss with respect to w.
    - Find the closed form solution by setting the gradient equal to zero and solving the matrix equation.
    - When might this be better than using stochastic gradient descent? When might this method break?
3. Assume that the noise model governing the additive noise ϵ is the exponential distribution. That is, $p(\epsilon) = \frac{1}{2} \exp(-|\epsilon|)$.
    - Write out the negative log-likelihood of the data under the model $-\log P(Y \mid X)$.
    - Can you find a closed form solution?
    - Suggest a stochastic gradient descent algorithm to solve this problem. What could possibly go wrong (hint - what happens near the stationary point as we keep on updating the parameters). Can you fix this?

3.2 Linear Regression Implementation from Scratch

Now that you understand the key ideas behind linear regression, we can begin to work through a hands-on implementation in code. In this section, we will implement the entire method from scratch, including the data pipeline, the model, the loss function, and the gradient descent optimizer. While modern deep learning frameworks can automate nearly all of this work, implementing things from scratch is the only to make sure that you really know what you are doing. Moreover, when it comes time to customize models, defining our own layers, loss functions, etc., understanding how things work under the hood will prove handy. In this section, we will rely only on `ndarray` and `autograd`. Afterwards, we will introduce a more compact implementation, taking advantage of Gluon's bells and whistles. To start off, we import the few required packages.

```
%matplotlib inline
import d2l
```

(continues on next page)

(continued from previous page)

```
from mxnet import autograd, np, npx
import random
npx.set_np()
```

3.2.1 Generating the Dataset

To keep things simple, we will construct an artificial dataset according to a linear model with additive noise. Out task will be to recover this model's parameters using the finite set of examples contained in our dataset. We will keep the data low-dimensional so we can visualize it easily. In the following code snippet, we generated a dataset containing 1000 examples, each consisting of 2 features sampled from a standard normal distribution. Thus our synthetic dataset will be an object $\mathbf{X} \in \mathbb{R}^{1000 \times 2}$.

The true parameters generating our data will be $\mathbf{w} = [2, -3.4]^\top$ and $b = 4.2$ and our synthetic labels will be assigned according to the following linear model with noise term ϵ:

$$\mathbf{y} = \mathbf{X}\mathbf{w} + b + \epsilon. \tag{3.2.1}$$

You could think of ϵ as capturing potential measurement errors on the features and labels. We will assume that the standard assumptions hold and thus that ϵ obeys a normal distribution with mean of 0. To make our problem easy, we will set its standard deviation to 0.01. The following code generates our synthetic dataset:

```
# Saved in the d2l package for later use
def synthetic_data(w, b, num_examples):
    """generate y = X w + b + noise"""
    X = np.random.normal(0, 1, (num_examples, len(w)))
    y = np.dot(X, w) + b
    y += np.random.normal(0, 0.01, y.shape)
    return X, y

true_w = np.array([2, -3.4])
true_b = 4.2
features, labels = synthetic_data(true_w, true_b, 1000)
```

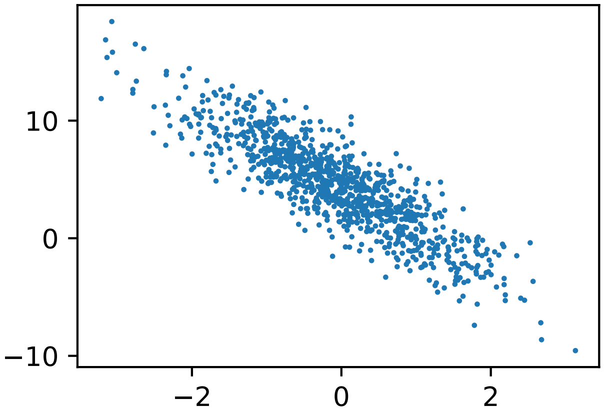
Note that each row in `features` consists of a 2-dimensional data point and that each row in `labels` consists of a 1-dimensional target value (a scalar).

```
print('features:', features[0],'\nlabel:', labels[0])
```

```
features: [2.2122064 1.1630787]
label: 4.662078
```

By generating a scatter plot using the second `features[:, 1]` and `labels`, we can clearly observe the linear correlation between the two.

```
d2l.set_figsize((3.5, 2.5))
d2l.plt.scatter(features[:, 1].asnumpy(), labels.asnumpy(), 1);
```



3.2.2 Reading the Dataset

Recall that training models consists of making multiple passes over the dataset, grabbing one minibatch of examples at a time, and using them to update our model. Since this process is so fundamental to training machine learning algorithms, its worth defining a utility function to shuffle the data and access it in minibatches.

In the following code, we define a `data_iter` function to demonstrate one possible implementation of this functionality. The function takes a batch size, a design matrix, and a vector of labels, yielding minibatches of size `batch_size`. Each minibatch consists of an tuple of features and labels.

```
def data_iter(batch_size, features, labels):
    num_examples = len(features)
    indices = list(range(num_examples))
    # The examples are read at random, in no particular order
    random.shuffle(indices)
    for i in range(0, num_examples, batch_size):
        batch_indices = np.array(
            indices[i: min(i + batch_size, num_examples)])
        yield features[batch_indices], labels[batch_indices]
```

In general, note that we want to use reasonably sized minibatches to take advantage of the GPU hardware, which excels at parallelizing operations. Because each example can be fed through our models in parallel and the gradient of the loss function for each example can also be taken in parallel, GPUs allow us to process hundreds of examples in scarcely more time than it might take to process just a single example.

To build some intuition, let's read and print the first small batch of data examples. The shape of the features in each minibatch tells us both the minibatch size and the number of input features. Likewise, our minibatch of labels will have a shape given by `batch_size`.

```
batch_size = 10

for X, y in data_iter(batch_size, features, labels):
    print(X, '\n', y)
    break
```

```
[[ 0.75600237  0.16707687]
 [-0.41116217 -0.5758122 ]
 [ 0.07801306 -0.98644793]
 [-0.8773337  -1.2138577 ]
 [-0.16882208  0.32604918]
 [-0.51751894 -0.5553699 ]
 [-0.06807706 -0.13130364]
 [-0.07048063  0.26596692]
 [-1.1500514   0.11681478]
 [-0.6699576  -0.06915967]]
 [5.1450267 5.3464904 7.7167864 6.5756965 2.7615652 5.0592566 4.5233345
 3.1558564 1.512002  3.0932438]
```

As we run the iterator, we obtain distinct minibatches successively until all the data has been exhausted (try this). While the iterator implemented above is good for didactic purposes, it is inefficient in ways that might get us in trouble on real problems. For example, it requires that we load all data in memory and that we perform lots of random memory access. The built-in iterators implemented in Apache MXNet are considerably efficient and they can deal both with data stored on file and data fed via a data stream.

3.2.3 Initializing Model Parameters

Before we can begin optimizing our model's parameters by gradient descent, we need to have some parameters in the first place. In the following code, we initialize weights by sampling random numbers from a normal distribution with mean 0 and a standard deviation of 0.01, setting the bias b to 0.

```
w = np.random.normal(0, 0.01, (2, 1))
b = np.zeros(1)
```

Now that we have initialized our parameters, our next task is to update them until they fit our data sufficiently well. Each update requires taking the gradient (a multi-dimensional derivative) of our loss function with respect to the parameters. Given this gradient, we can update each parameter in the direction that reduces the loss.

Since nobody wants to compute gradients explicitly (this is tedious and error prone), we use automatic differentiation to compute the gradient. See Section 2.5 for more details. Recall from the autograd chapter that in order for `autograd` to know that it should store a gradient for our parameters, we need to invoke the `attach_grad` function, allocating memory to store the gradients that we plan to take.

```
w.attach_grad()
b.attach_grad()
```

3.2.4 Defining the Model

Next, we must define our model, relating its inputs and parameters to its outputs. Recall that to calculate the output of the linear model, we simply take the matrix-vector dot product of the examples $\mathbf{X}$ and the models weights w, and add the offset b to each example. Note that below `np.dot(X, w)` is a vector and `b` is a scalar. Recall that when we add a vector and a scalar, the scalar is added to each component of the vector.

```
# Saved in the d2l package for later use
def linreg(X, w, b):
    return np.dot(X, w) + b
```

3.2.5 Defining the Loss Function

Since updating our model requires taking the gradient of our loss function, we ought to define the loss function first. Here we will use the squared loss function as described in the previous section. In the implementation, we need to transform the true value `y` into the predicted value's shape `y_hat`. The result returned by the following function will also be the same as the `y_hat` shape.

```
# Saved in the d2l package for later use
def squared_loss(y_hat, y):
    return (y_hat - y.reshape(y_hat.shape)) ** 2 / 2
```

3.2.6 Defining the Optimization Algorithm

As we discussed in the previous section, linear regression has a closed-form solution. However, this is not a book about linear regression, it is a book about deep learning. Since none of the other models that this book introduces can be solved analytically, we will take this opportunity to introduce your first working example of stochastic gradient descent (SGD).

At each step, using one batch randomly drawn from our dataset, we will estimate the gradient of the loss with respect to our parameters. Next, we will update our parameters (a small amount) in the direction that reduces the loss. Recall from Section 2.5 that after we call `backward` each parameter (`param`) will have its gradient stored in `param.grad`. The following code applies the SGD update, given a set of parameters, a learning rate, and a batch size. The size of the update step is determined by the learning rate `lr`. Because our loss is calculated as a sum over the batch of examples, we normalize our step size by the batch size (`batch_size`), so that the magnitude of a typical step size does not depend heavily on our choice of the batch size.

```
# Saved in the d2l package for later use
def sgd(params, lr, batch_size):
    for param in params:
        param[:] = param - lr * param.grad / batch_size
```

3.2.7 Training

Now that we have all of the parts in place, we are ready to implement the main training loop. It is crucial that you understand this code because you will see nearly identical training loops over and over again throughout your career in deep learning.

In each iteration, we will grab minibatches of models, first passing them through our model to obtain a set of predictions. After calculating the loss, we call the `backward` function to initiate the backwards pass through the network, storing the gradients with respect to each parameter in its corresponding `.grad` attribute. Finally, we will call the optimization algorithm `sgd` to update the model parameters. Since we previously set the batch size `batch_size` to 10, the loss shape `l` for each minibatch is $(10, 1)$.

In summary, we will execute the following loop:

- Initialize parameters $(\mathbf{w}, b)$
- Repeat until done
  - Compute gradient $\mathbf{g} \leftarrow \partial_{(\mathbf{w},b)} \frac{1}{\mathcal{B}} \sum_{i \in \mathcal{B}} l(\mathbf{x}^i, y^i, \mathbf{w}, b)$
  - Update parameters $(\mathbf{w}, b) \leftarrow (\mathbf{w}, b) - \eta \mathbf{g}$

In the code below, `l` is a vector of the losses for each example in the minibatch. Because `l` is not a scalar variable, running `l.backward()` adds together the elements in `l` to obtain the new variable and then calculates the gradient.

In each epoch (a pass through the data), we will iterate through the entire dataset (using the `data_iter` function) once passing through every examples in the training dataset (assuming the number of examples is divisible by the batch size). The number of epochs `num_epochs` and the learning rate `lr` are both hyper-parameters, which we set here to 3 and 0.03, respectively. Unfortunately, setting hyper-parameters is tricky and requires some adjustment by trial and error. We elide these details for now but revise them later in Chapter 11.

```
lr = 0.03  # Learning rate
num_epochs = 3  # Number of iterations
net = linreg  # Our fancy linear model
loss = squared_loss  # 0.5 (y-y')^2

for epoch in range(num_epochs):
    # Assuming the number of examples can be divided by the batch size, all
    # the examples in the training dataset are used once in one epoch
    # iteration. The features and tags of minibatch examples are given by X
    # and y respectively
    for X, y in data_iter(batch_size, features, labels):
        with autograd.record():
            l = loss(net(X, w, b), y)  # Minibatch loss in X and y
        l.backward()  # Compute gradient on l with respect to [w, b]
        sgd([w, b], lr, batch_size)  # Update parameters using their gradient
    train_l = loss(net(features, w, b), labels)
    print('epoch %d, loss %f' % (epoch + 1, train_l.mean().asnumpy()))
```

```
epoch 1, loss 0.025000
epoch 2, loss 0.000089
epoch 3, loss 0.000051
```

In this case, because we synthesized the data ourselves, we know precisely what the true parameters are. Thus, we can evaluate our success in training by comparing the true parameters with those that we learned through our training loop. Indeed they turn out to be very close to each other.

```
print('Error in estimating w', true_w - w.reshape(true_w.shape))
print('Error in estimating b', true_b - b)
```

```
Error in estimating w [0.00050855 0.00011015]
Error in estimating b [0.00079584]
```

Note that we should not take it for granted that we are able to recover the parameters accurately. This only happens for a special category problems: strongly convex optimization problems with "enough" data to ensure that the noisy samples allow us to recover the underlying dependency. In most cases this is *not* the case. In fact, the parameters of a deep network are rarely the same (or even close) between two different runs, unless all conditions are identical, including the order in which the data is traversed. However, in machine learning, we are typically less concerned with recovering true underlying parameters, and more concerned with parameters that lead to accurate prediction. Fortunately, even on difficult optimization problems, stochastic gradient descent can often find remarkably good solutions, owing partly to the fact that, for deep networks, there exist many configurations of the parameters that lead to accurate prediction.

Summary

We saw how a deep network can be implemented and optimized from scratch, using just `ndarray` and `autograd`, without any need for defining layers, fancy optimizers, etc. This only scratches the surface of what is possible. In the following sections, we will describe additional models based on the concepts that we have just introduced and learn how to implement them more concisely.

Exercises

1. What would happen if we were to initialize the weights $\mathbf{w} = 0$. Would the algorithm still work?
2. Assume that you are Georg Simon Ohm[51] trying to come up with a model between voltage and current. Can you use `autograd` to learn the parameters of your model.
3. Can you use Planck's Law[52] to determine the temperature of an object using spectral energy density?
4. What are the problems you might encounter if you wanted to extend `autograd` to second derivatives? How would you fix them?
5. Why is the `reshape` function needed in the `squared_loss` function?
6. Experiment using different learning rates to find out how fast the loss function value drops.
7. If the number of examples cannot be divided by the batch size, what happens to the `data_iter` function's behavior?

[51] https://en.wikipedia.org/wiki/Georg_Ohm
[52] https://en.wikipedia.org/wiki/Planck%27s_law

3.3 Concise Implementation of Linear Regression

Broad and intense interest in deep learning for the past several years has inspired both companies, academics, and hobbyists to develop a variety of mature open source frameworks for automating the repetitive work of implementing gradient-based learning algorithms. In the previous section, we relied only on (i) `ndarray` for data storage and linear algebra; and (ii) `autograd` for calculating derivatives. In practice, because data iterators, loss functions, optimizers, and neural network layers (and some whole architectures) are so common, modern libraries implement these components for us as well.

In this section, we will show you how to implement the linear regression model from Section 3.2 concisely by using Gluon.

3.3.1 Generating the Dataset

To start, we will generate the same dataset as in the previous section.

```
import d2l
from mxnet import autograd, gluon, np, npx
npx.set_np()

true_w = np.array([2, -3.4])
true_b = 4.2
features, labels = d2l.synthetic_data(true_w, true_b, 1000)
```

3.3.2 Reading the Dataset

Rather than rolling our own iterator, we can call upon Gluon's `data` module to read data. The first step will be to instantiate an `ArrayDataset`. This object's constructor takes one or more `ndarrays` as arguments. Here, we pass in `features` and `labels` as arguments. Next, we will use the `ArrayDataset` to instantiate a `DataLoader`, which also requires that we specify a `batch_size` and specify a Boolean value `shuffle` indicating whether or not we want the `DataLoader` to shuffle the data on each epoch (pass through the dataset).

```
# Saved in the d2l package for later use
def load_array(data_arrays, batch_size, is_train=True):
    """Construct a Gluon data loader"""
    dataset = gluon.data.ArrayDataset(*data_arrays)
    return gluon.data.DataLoader(dataset, batch_size, shuffle=is_train)

batch_size = 10
data_iter = load_array((features, labels), batch_size)
```

Now we can use `data_iter` in much the same way as we called the `data_iter` function in the previous section. To verify that it is working, we can read and print the first minibatch of instances.

```
for X, y in data_iter:
    print(X, '\n', y)
    break
```

```
[[ 1.9118724   0.99435186]
 [-0.69279176 -0.11865307]
 [-0.83655155 -0.58954424]
 [-0.9345107   1.0642178 ]
 [-0.34534106 -0.7556803 ]
 [ 0.30967948 -1.7559303 ]
 [-0.6308071   0.53733146]
 [ 0.46879596 -0.30394194]
 [ 0.75600237  0.16707687]
 [-0.16452314 -0.21437684]]
 [ 4.6377006  3.2253058  4.530804  -1.2874792  6.07641   10.791387
  1.0948168  6.1717677  5.1450267  4.620073 ]
```

3.3.3 Defining the Model

When we implemented linear regression from scratch (in :numref``sec_linear_scratch``), we defined our model parameters explicitly and coded up the calculations to produce output using basic linear algebra operations. You *should* know how to do this. But once your models get more complex, and once you have to do this nearly every day, you will be glad for the assistance. The situation is similar to coding up your own blog from scratch. Doing it once or twice is rewarding and instructive, but you would be a lousy web developer if every time you needed a blog you spent a month reinventing the wheel.

For standard operations, we can use Gluon's predefined layers, which allow us to focus especially on the layers used to construct the model rather than having to focus on the implementation. To define a linear model, we first import the `nn` module, which defines a large number of neural network layers (note that "nn" is an abbreviation for neural networks). We will first define a model variable `net`, which will refer to an instance of the `Sequential` class. In Gluon, `Sequential` defines a container for several layers that will be chained together. Given input data, a `Sequential` passes it through the first layer, in turn passing the output as the second layer's input and so forth. In the following example, our model consists of only one layer, so we do not really need `Sequential`. But since nearly all of our future models will involve multiple layers, we will use it anyway just to familiarize you with the most standard workflow.

```
from mxnet.gluon import nn
net = nn.Sequential()
```

Recall the architecture of a single-layer network as shown in Fig. 3.3.1. The layer is said to be *fully-connected* because each of its inputs are connected to each of its outputs by means of a matrix-vector multiplication. In Gluon, the fully-connected layer is defined in the `Dense` class. Since we only want to generate a single scalar output, we set that number to 1.

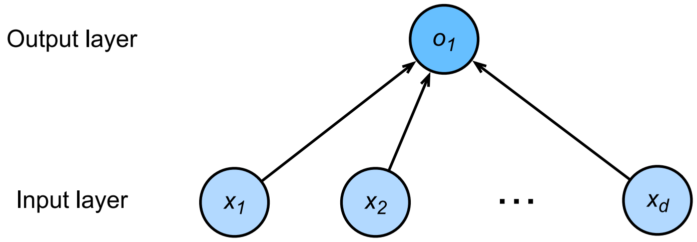


Fig. 3.3.1: Linear regression is a single-layer neural network.

```
net.add(nn.Dense(1))
```

It is worth noting that, for convenience, Gluon does not require us to specify the input shape for each layer. So here, we do not need to tell Gluon how many inputs go into this linear layer. When we first try to pass data through our model, e.g., when we execute `net(X)` later, Gluon will automatically infer the number of inputs to each layer. We will describe how this works in more detail in the chapter "Deep Learning Computation".

3.3.4 Initializing Model Parameters

Before using `net`, we need to initialize the model parameters, such as the weights and biases in the linear regression model. We will import the `initializer` module from MXNet. This module provides various methods for model parameter initialization. Gluon makes `init` available as a shortcut (abbreviation) to access the `initializer` package. By calling `init.Normal(sigma=0.01)`, we specify that each *weight* parameter should be randomly sampled from a normal distribution with mean 0 and standard deviation 0.01. The *bias* parameter will be initialized to zero by default. Both the weight vector and bias will have attached gradients.

```
from mxnet import init
net.initialize(init.Normal(sigma=0.01))
```

The code above may look straightforward but you should note that something strange is happening here. We are initializing parameters for a network even though Gluon does not yet know how many dimensions the input will have! It might be 2 as in our example or it might be 2000. Gluon lets us get away with this because behind the scenes, the initialization is actually *deferred*. The real initialization will take place only when we for the first time attempt to pass data through the network. Just be careful to remember that since the parameters have not been initialized yet, we cannot access or manipulate them.

3.3.5 Defining the Loss Function

In Gluon, the `loss` module defines various loss functions. We will the imported module `loss` with the pseudonym `gloss`, to avoid confusing it for the variable holding our chosen loss function. In this example, we will use the Gluon implementation of squared loss (`L2Loss`).

```
from mxnet.gluon import loss as gloss
loss = gloss.L2Loss()  # The squared loss is also known as the L2 norm loss
```

3.3.6 Defining the Optimization Algorithm

Minibatch SGD and related variants are standard tools for optimizing neural networks and thus Gluon supports SGD alongside a number of variations on this algorithm through its `Trainer` class. When we instantiate the `Trainer`, we will specify the parameters to optimize over (obtainable from our net via `net.collect_params()`), the optimization algorithm we wish to use (`sgd`), and a dictionary of hyper-parameters required by our optimization algorithm. SGD just requires that we set the value `learning_rate`, (here we set it to 0.03).

```
from mxnet import gluon
trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': 0.03})
```

3.3.7 Training

You might have noticed that expressing our model through Gluon requires comparatively few lines of code. We did not have to individually allocate parameters, define our loss function, or implement stochastic gradient descent. Once we start working with much more complex models, Gluon's advantages will grow considerably. However, once we have all the basic pieces in place, the training loop itself is strikingly similar to what we did when implementing everything from scratch.

To refresh your memory: for some number of epochs, we will make a complete pass over the dataset (train_data), iteratively grabbing one minibatch of inputs and the corresponding ground-truth labels. For each minibatch, we go through the following ritual:

- Generate predictions by calling `net(X)` and calculate the loss `l` (the forward pass).
- Calculate gradients by calling `l.backward()` (the backward pass).
- Update the model parameters by invoking our SGD optimizer (note that `trainer` already knows which parameters to optimize over, so we just need to pass in the minibatch size.

For good measure, we compute the loss after each epoch and print it to monitor progress.

```
num_epochs = 3
for epoch in range(1, num_epochs + 1):
    for X, y in data_iter:
        with autograd.record():
            l = loss(net(X), y)
        l.backward()
        trainer.step(batch_size)
    l = loss(net(features), labels)
    print('epoch %d, loss: %f' % (epoch, l.mean().asnumpy()))
```

```
epoch 1, loss: 0.025167
epoch 2, loss: 0.000090
epoch 3, loss: 0.000051
```

Below, we compare the model parameters learned by training on finite data and the actual parameters that generated our dataset. To access parameters with Gluon, we first access the layer that we need from `net` and then access that layer's weight (`weight`) and bias (`bias`). To access each parameter's values as an `ndarray`, we invoke its `data` method. As in our from-scratch implementation, note that our estimated parameters are close to their ground truth counterparts.

```
w = net[0].weight.data()
print('Error in estimating w', true_w.reshape(w.shape) - w)
b = net[0].bias.data()
print('Error in estimating b', true_b - b)
```

```
Error in estimating w [[0.00019813 0.0003016 ]]
Error in estimating b [0.00057745]
```

Summary

- Using Gluon, we can implement models much more succinctly.
- In Gluon, the `data` module provides tools for data processing, the `nn` module defines a large number of neural network layers, and the `loss` module defines many common loss functions.
- MXNet's module `initializer` provides various methods for model parameter initialization.
- Dimensionality and storage are automatically inferred (but be careful not to attempt to access parameters before they have been initialized).

Exercises

1. If we replace `l = loss(output, y)` with `l = loss(output, y).mean()`, we need to change `trainer.step(batch_size)` to `trainer.step(1)` for the code to behave identically. Why?
2. Review the MXNet documentation to see what loss functions and initialization methods are provided in the modules `gluon.loss` and `init`. Replace the loss by Huber's loss.
3. How do you access the gradient of `dense.weight`?

3.4 Softmax Regression

In Section 3.1, we introduced linear regression, working through implementations from scratch in Section 3.2 and again using Gluon in Section 3.3 to do the heavy lifting.

Regression is the hammer we reach for when we want to answer *how much?* or *how many?* questions. If you want to predict the number of dollars (the *price*) at which a house will be sold, or the number of wins a baseball team might have, or the number of days that a patient will remain hospitalized before being discharged, then you are probably looking for a regression model.

In practice, we are more often interested in classification: asking not *how much?* but *which one?*

- Does this email belong in the spam folder or the inbox*?
- Is this customer more likely *to sign up* or *not to sign up* for a subscription service?*

- Does this image depict a donkey, a dog, a cat, or a rooster?
- Which movie is Aston most likely to watch next?

Colloquially, machine learning practitioners overload the word *classification* to describe two subtly different problems: (i) those where we are interested only in *hard* assignments of examples to categories; and (ii) those where we wish to make *soft assignments*, i.e., to assess the *probability* that each category applies. The distinction tends to get blurred, in part, because often, even when we only care about hard assignments, we still use models that make soft assignments.

3.4.1 Classification Problems

To get our feet wet, let's start off with a simple image classification problem. Here, each input consists of a 2×2 grayscale image. We can represent each pixel value with a single scalar, giving us four features x_1, x_2, x_3, x_4. Further, let's assume that each image belongs to one among the categories "cat", "chicken" and "dog".

Next, we have to choose how to represent the labels. We have two obvious choices. Perhaps the most natural impulse would be to choose $y \in \{1, 2, 3\}$, where the integers represent {dog, cat, chicken} respectively. This is a great way of *storing* such information on a computer. If the categories had some natural ordering among them, say if we were trying to predict {baby, toddler, adolescent, young adult, adult, geriatric}, then it might even make sense to cast this problem as regression and keep the labels in this format.

But general classification problems do not come with natural orderings among the classes. Fortunately, statisticians long ago invented a simple way to represent categorical data: the *one hot encoding*. A one-hot encoding is a vector with as many components as we have categories. The component corresponding to particular instance's category is set to 1 and all other components are set to 0.

$$y \in \{(1, 0, 0), (0, 1, 0), (0, 0, 1)\}. \tag{3.4.1}$$

In our case, y would be a three-dimensional vector, with $(1, 0, 0)$ corresponding to "cat", $(0, 1, 0)$ to "chicken" and $(0, 0, 1)$ to "dog".

Network Architecture

In order to estimate the conditional probabilities associated with each classes, we need a model with multiple outputs, one per class. To address classification with linear models, we will need as many linear functions as we have outputs. Each output will correspond to its own linear function. In our case, since we have 4 features and 3 possible output categories, we will need 12 scalars to represent the weights, (w with subscripts) and 3 scalars to represent the biases (b with subscripts). We compute these three *logits*, o_1, o_2, and o_3, for each input:

$$\begin{aligned}
o_1 &= x_1 w_{11} + x_2 w_{12} + x_3 w_{13} + x_4 w_{14} + b_1,\\
o_2 &= x_1 w_{21} + x_2 w_{22} + x_3 w_{23} + x_4 w_{24} + b_2,\\
o_3 &= x_1 w_{31} + x_2 w_{32} + x_3 w_{33} + x_4 w_{34} + b_3.
\end{aligned} \tag{3.4.2}$$

We can depict this calculation with the neural network diagram shown in Fig. 3.4.1. Just as in linear regression, softmax regression is also a single-layer neural network. And since the calculation of each output, o_1, o_2, and o_3, depends on all inputs, x_1, x_2, x_3, and x_4, the output layer of softmax regression can also be described as fully-connected layer.

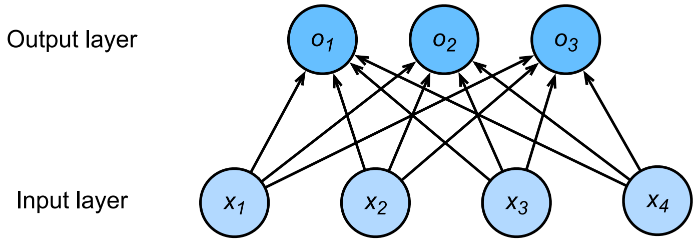


Fig. 3.4.1: Softmax regression is a single-layer neural network.

To express the model more compactly, we can use linear algebra notation. In vector form, we arrive at $\mathbf{o} = \mathbf{W}\mathbf{x} + \mathbf{b}$, a form better suited both for mathematics, and for writing code. Note that we have gathered all of our weights into a 3×4 matrix and that for a given example $\mathbf{x}$, our outputs are given by a matrix-vector product of our weights by our inputs plus our biases $\mathbf{b}$.

Softmax Operation

The main approach that we are going to take here is to interpret the outputs of our model as probabilities. We will optimize our parameters to produce probabilities that maximize the likelihood of the observed data. Then, to generate predictions, we will set a threshold, for example, choosing the *argmax* of the predicted probabilities.

Put formally, we would like outputs $\hat{y}_k$ that we can interpret as the probability that a given item belongs to class k. Then we can choose the class with the largest output value as our prediction $\operatorname{argmax}_k y_k$. For example, if $\hat{y}_1$, $\hat{y}_2$, and $\hat{y}_3$ are 0.1, $.8$, and 0.1, respectively, then we predict category 2, which (in our example) represents "chicken".

You might be tempted to suggest that we interpret the logits o directly as our outputs of interest. However, there are some problems with directly interpreting the output of the linear layer as a probability. Nothing constrains these numbers to sum to 1. Moreover, depending on the inputs, they can take negative values. These violate basic axioms of probability presented in Section 2.6

To interpret our outputs as probabilities, we must guarantee that (even on new data), they will be nonnegative and sum up to 1. Moreover, we need a training objective that encourages the model to estimate faithfully *probabilities*. Of all instances when a classifier outputs $.5$, we hope that half of those examples will *actually* belong to the predicted class. This is a property called *calibration*.

The *softmax function*, invented in 1959 by the social scientist R Duncan Luce in the context of *choice models* does precisely this. To transform our logits such that they become nonnegative and sum to 1, while requiring that the model remains differentiable, we first exponentiate each logit (ensuring non-negativity) and then divide by their sum (ensuring that they sum to 1).

$$\hat{\mathbf{y}} = \mathrm{softmax}(\mathbf{o}) \quad \text{where} \quad \hat{y}_i = \frac{\exp(o_i)}{\sum_j \exp(o_j)}. \tag{3.4.3}$$

It is easy to see $\hat{y}_1 + \hat{y}_2 + \hat{y}_3 = 1$ with $0 \leq \hat{y}_i \leq 1$ for all i. Thus, $\hat{y}$ is a proper probability distribution and the values of $\hat{\mathbf{y}}$ can be interpreted accordingly. Note that the softmax operation does not change the ordering among the logits, and thus we can still pick out the most likely class by:

$$\hat{\imath}(\mathbf{o}) = \operatorname*{argmax}_i o_i = \operatorname*{argmax}_i \hat{y}_i. \tag{3.4.4}$$

The logits $\mathbf{o}$ then are simply the pre-softmax values that determining the probabilities assigned to each category. Summarizing it all in vector notation we get $\mathbf{o}^{(i)} = \mathbf{W}\mathbf{x}^{(i)} + \mathbf{b}$, where $\hat{\mathbf{y}}^{(i)} = \mathrm{softmax}(\mathbf{o}^{(i)})$.

Vectorization for Minibatches

To improve computational efficiency and take advantage of GPUs, we typically carry out vector calculations for minibatches of data. Assume that we are given a minibatch $\mathbf{X}$ of examples with dimensionality d and batch size n. Moreover, assume that we have q categories (outputs). Then the minibatch features $\mathbf{X}$ are in $\mathbb{R}^{n\times d}$, weights $\mathbf{W} \in \mathbb{R}^{d\times q}$, and the bias satisfies $\mathbf{b} \in \mathbb{R}^q$.

$$\begin{aligned} \mathbf{O} &= \mathbf{X}\mathbf{W} + \mathbf{b}, \\ \hat{\mathbf{Y}} &= \mathrm{softmax}(\mathbf{O}). \end{aligned} \tag{3.4.5}$$

This accelerates the dominant operation into a matrix-matrix product $\mathbf{W}\mathbf{X}$ vs the matrix-vector products we would be executing if we processed one example at a time. The softmax itself can be computed by exponentiating all entries in $\mathbf{O}$ and then normalizing them by the sum.

3.4.2 Loss Function

Next, we need a *loss function* to measure the quality of our predicted probabilities. We will rely on *likelihood maximization*, the very same concept that we encountered when providing a probabilistic justification for the least squares objective in linear regression (Section 3.1).

Log-Likelihood

The softmax function gives us a vector $\hat{\mathbf{y}}$, which we can interpret as estimated conditional probabilities of each class given the input x, e.g., $\hat{y}_1$ = $\hat{P}(y = \text{cat} \mid \mathbf{x})$. We can compare the estimates with reality by checking how probable the *actual* classes are according to our model, given the features.

$$P(Y \mid X) = \prod_{i=1}^{n} P(y^{(i)} \mid x^{(i)}) \text{ and thus } -\log P(Y \mid X) = \sum_{i=1}^{n} -\log P(y^{(i)} \mid x^{(i)}). \tag{3.4.6}$$

Maximizing $P(Y \mid X)$ (and thus equivalently minimizing $-\log P(Y \mid X)$) corresponds to predicting the label well. This yields the loss function (we dropped the superscript (i) to avoid notation clutter):

$$l = -\log P(y \mid x) = -\sum_j y_j \log \hat{y}_j. \tag{3.4.7}$$

For reasons explained later on, this loss function is commonly called the *cross-entropy* loss. Here, we used that by construction $\hat{y}$ is a discrete probability distribution and that the vector $\mathbf{y}$ is a one-hot vector. Hence the the sum over all coordinates j vanishes for all but one term. Since all $\hat{y}_j$ are probabilities, their logarithm is never larger than 0. Consequently, the loss function cannot be minimized any further if we correctly predict y with *certainty*, i.e., if $P(y \mid x) = 1$ for the correct label. Note that this is often not possible. For example, there might be label noise in the dataset (some examples may be mislabeled). It may also not be possible when the input features are not sufficiently informative to classify every example perfectly.

Softmax and Derivatives

Since the softmax and the corresponding loss are so common, it is worth while understanding a bit better how it is computed. Plugging o into the definition of the loss l and using the definition of the softmax we obtain:

$$l = -\sum_j y_j \log \hat{y}_j = \sum_j y_j \log \sum_k \exp(o_k) - \sum_j y_j o_j = \log \sum_k \exp(o_k) - \sum_j y_j o_j. \tag{3.4.8}$$

To understand a bit better what is going on, consider the derivative with respect to o. We get

$$\partial_{o_j} l = \frac{\exp(o_j)}{\sum_k \exp(o_k)} - y_j = \mathrm{softmax}(\mathbf{o})_j - y_j = P(y = j \mid x) - y_j. \tag{3.4.9}$$

In other words, the gradient is the difference between the probability assigned to the true class by our model, as expressed by the probability $P(y \mid x)$, and what actually happened, as expressed by y. In this sense, it is very similar to what we saw in regression, where the gradient was the difference between the observation y and estimate $\hat{y}$. This is not coincidence. In any exponential family[55] model, the gradients of the log-likelihood are given by precisely this term. This fact makes computing gradients easy in practice.

Cross-Entropy Loss

Now consider the case where we observe not just a single outcome but an entire distribution over outcomes. We can use the same representation as before for y. The only difference is that rather than a vector containing only binary entries, say $(0, 0, 1)$, we now have a generic probability vector, say $(0.1, 0.2, 0.7)$. The math that we used previously to define the loss l still works out fine, just that the interpretation is slightly more general. It is the expected value of the loss for a distribution over labels.

$$l(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_j y_j \log \hat{y}_j. \tag{3.4.10}$$

This loss is called the cross-entropy loss and it is one of the most commonly used losses for multi-class classification. We can demystify the name by introducing the basics of information theory.

3.4.3 Information Theory Basics

Information theory deals with the problem of encoding, decoding, transmitting and manipulating information (also known as data) in as concise form as possible.

Entropy

The central idea in information theory is to quantify the information content in data. This quantity places a hard limit on our ability to compress the data. In information theory, this quantity is called the entropy[56] of a distribution p, and it is captured by the following equation:

$$H[p] = \sum_j -p(j) \log p(j). \tag{3.4.11}$$

[55] https://en.wikipedia.org/wiki/Exponential_family

[56] https://en.wikipedia.org/wiki/Entropy

One of the fundamental theorems of information theory states that in order to encode data drawn randomly from the distribution p, we need at least $H[p]$ "nats" to encode it. If you wonder what a "nat" is, it is the equivalent of bit but when using a code with base e rather than one with base 2. One nat is $\frac{1}{\log(2)} \approx 1.44$ bit. $H[p]/2$ is often also called the binary entropy.

Surprisal

You might be wondering what compression has to do with prediction. Imagine that we have a stream of data that we want to compress. If it is always easy for us to predict the next token, then this data is easy to compress! Take the extreme example where every token in the stream always takes the same value. That is a very boring data stream! And not only is it boring, but it is easy to predict. Because they are always the same, we do not have to transmit any information to communicate the contents of the stream. Easy to predict, easy to compress.

However if we cannot perfectly predict every event, then we might some times be surprised. Our surprise is greater when we assigned an event lower probability. For reasons that we will elaborate in the appendix, Claude Shannon settled on $\log(1/p(j)) = -\log p(j)$ to quantify one's *surprisal* at observing an event j having assigned it a (subjective) probability $p(j)$. The entropy is then the *expected surprisal* when one assigned the correct probabilities (that truly match the data-generating process). The entropy of the data is then the least surprised that one can ever be (in expectation).

Cross-Entropy Revisited

So if entropy is level of surprise experienced by someone who knows the true probability, then you might be wondering, *what is cross-entropy?* The cross-entropy *from :math:`p` to :math:`q`*, denoted H(p, q), is the expected surprisal of an observer with subjective probabilities q upon seeing data that was actually generated according to probabilities p. The lowest possible cross-entropy is achieved when $p = q$. In this case, the cross-entropy from p to q is $H(p, p) = H(p)$. Relating this back to our classification objective, even if we get the best possible predictions, if the best possible possible, then we will never be perfect. Our loss is lower-bounded by the entropy given by the actual conditional distributions $P(\mathbf{y} \mid \mathbf{x})$.

Kullback Leibler Divergence

Perhaps the most common way to measure the distance between two distributions is to calculate the *Kullback Leibler divergence* $D(p\|q)$. This is simply the difference between the cross-entropy and the entropy, i.e., the additional cross-entropy incurred over the irreducible minimum value it could take:

$$D(p\|q) = H(p, q) - H[p] = \sum_j p(j) \log \frac{p(j)}{q(j)}. \tag{3.4.12}$$

Note that in classification, we do not know the true p, so we cannot compute the entropy directly. However, because the entropy is out of our control, minimizing $D(p\|q)$ with respect to q is equivalent to minimizing the cross-entropy loss.

In short, we can think of the cross-entropy classification objective in two ways: (i) as maximizing the likelihood of the observed data; and (ii) as minimizing our surprise (and thus the number of bits) required to communicate the labels.

3.4.4 Model Prediction and Evaluation

After training the softmax regression model, given any example features, we can predict the probability of each output category. Normally, we use the category with the highest predicted probability as the output category. The prediction is correct if it is consistent with the actual category (label). In the next part of the experiment, we will use accuracy to evaluate the model's performance. This is equal to the ratio between the number of correct predictions a nd the total number of predictions.

Summary

- We introduced the softmax operation which takes a vector maps it into probabilities.
- Softmax regression applies to classification problems. It uses the probability distribution of the output category in the softmax operation.
- cross-entropy is a good measure of the difference between two probability distributions. It measures the number of bits needed to encode the data given our model.

Exercises

1. Show that the Kullback-Leibler divergence $D(p\|q)$ is nonnegative for all distributions p and q. Hint: use Jensen's inequality, i.e., use the fact that $-\log x$ is a convex function.
2. Show that $\log \sum_j \exp(o_j)$ is a convex function in o.
3. We can explore the connection between exponential families and the softmax in some more depth
    - Compute the second derivative of the cross-entropy loss $l(y, \hat{y})$ for the softmax.
    - Compute the variance of the distribution given by $\mathrm{softmax}(o)$ and show that it matches the second derivative computed above.
4. Assume that we three classes which occur with equal probability, i.e., the probability vector is $(\frac{1}{3}, \frac{1}{3}, \frac{1}{3})$.
    - What is the problem if we try to design a binary code for it? Can we match the entropy lower bound on the number of bits?
    - Can you design a better code. Hint: what happens if we try to encode two independent observations? What if we encode n observations jointly?
5. Softmax is a misnomer for the mapping introduced above (but everyone in deep learning uses it). The real softmax is defined as $\mathrm{RealSoftMax}(a, b) = \log(\exp(a) + \exp(b))$.
    - Prove that $\mathrm{RealSoftMax}(a, b) > \max(a, b)$.
    - Prove that this holds for $\lambda^{-1}\mathrm{RealSoftMax}(\lambda a, \lambda b)$, provided that $\lambda > 0$.
    - Show that for $\lambda \to \infty$ we have $\lambda^{-1}\mathrm{RealSoftMax}(\lambda a, \lambda b) \to \max(a, b)$.
    - What does the soft-min look like?
    - Extend this to more than two numbers.

3.5 The Image Classification Dataset (Fashion-MNIST)

In Section 17.8, we trained a naive Bayes classifier, using the MNIST dataset introduced in 1998 (LeCun et al., 1998). While MNIST had a good run as a benchmark dataset, even simple models by today's standards achieve classification accuracy over 95%. making it unsuitable for distinguishing between stronger models and weaker ones. Today, MNIST serves as more of sanity checks than as a benchmark. To up the ante just a bit, we will focus our discussion in the coming sections on the qualitatively similar, but comparatively complex Fashion-MNIST dataset (Xiao et al., 2017), which was released in 2017.

```
%matplotlib inline
import d2l
from mxnet import gluon
import sys

d2l.use_svg_display()
```

3.5.1 Getting the Dataset

Just as with MNIST, Gluon makes it easy to download and load the FashionMNIST dataset into memory via the FashionMNIST class contained in gluon.data.vision. We briefly work through the mechanics of loading and exploring the dataset below. Please refer to Section 17.8 for more details on loading data.

```
mnist_train = gluon.data.vision.FashionMNIST(train=True)
mnist_test = gluon.data.vision.FashionMNIST(train=False)
```

FashionMNIST consists of images from 10 categories, each represented by 6k images in the training set and by 1k in the test set. Consequently the training set and the test set contain 60k and 10k images, respectively.

```
len(mnist_train), len(mnist_test)
```

```
(60000, 10000)
```

The images in Fashion-MNIST are associated with the following categories: t-shirt, trousers, pullover, dress, coat, sandal, shirt, sneaker, bag and ankle boot. The following function converts between numeric label indices and their names in text.

```
# Saved in the d2l package for later use
def get_fashion_mnist_labels(labels):
    text_labels = ['t-shirt', 'trouser', 'pullover', 'dress', 'coat',
```

(continues on next page)

(continued from previous page)

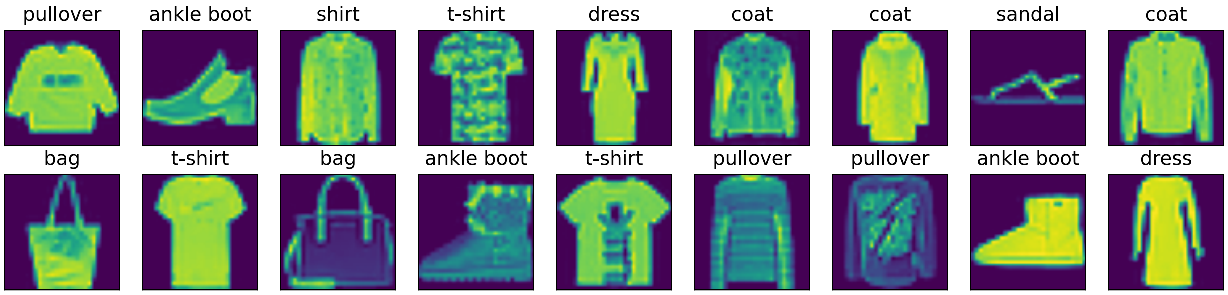
```
                   'sandal', 'shirt', 'sneaker', 'bag', 'ankle boot']
    return [text_labels[int(i)] for i in labels]
```

We can now create a function to visualize these examples.

```
# Saved in the d2l package for later use
def show_images(imgs, num_rows, num_cols, titles=None, scale=1.5):
    """Plot a list of images."""
    figsize = (num_cols * scale, num_rows * scale)
    _, axes = d2l.plt.subplots(num_rows, num_cols, figsize=figsize)
    axes = axes.flatten()
    for i, (ax, img) in enumerate(zip(axes, imgs)):
        ax.imshow(img.asnumpy())
        ax.axes.get_xaxis().set_visible(False)
        ax.axes.get_yaxis().set_visible(False)
        if titles:
            ax.set_title(titles[i])
    return axes
```

Here are the images and their corresponding labels (in text) for the first few examples in the training dataset.

```
X, y = mnist_train[:18]
show_images(X.squeeze(axis=-1), 2, 9, titles=get_fashion_mnist_labels(y));
```



3.5.2 Reading a Minibatch

To make our life easier when reading from the training and test sets, we use a DataLoader rather than creating one from scratch, as we did in Section 3.2. Recall that at each iteration, a DataLoader reads a minibatch of data with size batch_size each time.

During training, reading data can be a significant performance bottleneck, especially when our model is simple or when our computer is fast. A handy feature of Gluon's DataLoader is the ability to use multiple processes to speed up data reading. For instance, we can set aside 4 processes to read the data (via num_workers). Because this feature is not currently supported on Windows the following code checks the platform to make sure that we do not saddle our Windows-using friends with error messages later on.

```
# Saved in the d2l package for later use
def get_dataloader_workers(num_workers=4):
```

(continues on next page)

(continued from previous page)

```
    # 0 means no additional process is used to speed up the reading of data.
    if sys.platform.startswith('win'):
        return 0
    else:
        return num_workers
```

Below, we convert the image data from uint8 to 32-bit floating point numbers using the ToTensor class. Additionally, the transformer will divide all numbers by 255 so that all pixels have values between 0 and 1. The ToTensor class also moves the image channel from the last dimension to the first dimension to facilitate the convolutional neural network calculations introduced later. Through the transform_first function of the dataset, we apply the transformation of ToTensor to the first element of each instance (image and label).

```
batch_size = 256
transformer = gluon.data.vision.transforms.ToTensor()
train_iter = gluon.data.DataLoader(mnist_train.transform_first(transformer),
                                   batch_size, shuffle=True,
                                   num_workers=get_dataloader_workers())
```

Let's look at the time it takes to read the training data.

```
timer = d2l.Timer()
for X, y in train_iter:
    continue
'%.2f sec' % timer.stop()
```

```
'1.65 sec'
```

3.5.3 Putting All Things Together

Now we define the load_data_fashion_mnist function that obtains and reads the Fashion-MNIST dataset. It returns the data iterators for both the training set and validation set. In addition, it accepts an optional argument to resize images to another shape.

```
# Saved in the d2l package for later use
def load_data_fashion_mnist(batch_size, resize=None):
    """Download the Fashion-MNIST dataset and then load into memory."""
    dataset = gluon.data.vision
    trans = [dataset.transforms.Resize(resize)] if resize else []
    trans.append(dataset.transforms.ToTensor())
    trans = dataset.transforms.Compose(trans)
    mnist_train = dataset.FashionMNIST(train=True).transform_first(trans)
    mnist_test = dataset.FashionMNIST(train=False).transform_first(trans)
    return (gluon.data.DataLoader(mnist_train, batch_size, shuffle=True,
                                  num_workers=get_dataloader_workers()),
            gluon.data.DataLoader(mnist_test, batch_size, shuffle=False,
                                  num_workers=get_dataloader_workers()))
```

Below, we verify that image resizing works.

```
train_iter, test_iter = load_data_fashion_mnist(32, (64, 64))
for X, y in train_iter:
    print(X.shape)
    break
```

```
(32, 1, 64, 64)
```

We are now ready to work with the FashionMNIST dataset in the sections that follow.

Summary

- Fashion-MNIST is an apparel classification dataset consisting of images representing 10 categories.
- We will use this dataset in subsequent sections and chapters to evaluate various classification algorithms.
- We store the shape of each image with height h width w pixels as $h \times w$ or `(h, w)`.
- Data iterators are a key component for efficient performance. Rely on well-implemented iterators that exploit multi-threading to avoid slowing down your training loop.

Exercises

1. Does reducing the `batch_size` (for instance, to 1) affect read performance?
2. For non-Windows users, try modifying `num_workers` to see how it affects read performance. Plot the performance against the number of works employed.
3. Use the MXNet documentation to see which other datasets are available in `mxnet.gluon.data.vision`.
4. Use the MXNet documentation to see which other transformations are available in `mxnet.gluon.data.vision.transforms`.

3.6 Implementation of Softmax Regression from Scratch

Just as we implemented linear regression from scratch, we believe that multiclass logistic (softmax) regression is similarly fundamental and you ought to know the gory details of how to implement it yourself. As with linear regression, after doing things by hand we will breeze through an implementation in Gluon for comparison. To begin, let's import the familiar packages.

```
import d2l
from mxnet import autograd, np, npx, gluon
from IPython import display
npx.set_np()
```

We will work with the Fashion-MNIST dataset, just introduced in Section 3.5, setting up an iterator with batch size 256.

```
batch_size = 256
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)
```

3.6.1 Initializing Model Parameters

As in our linear regression example, each example here will be represented by a fixed-length vector. Each example in the raw data is a 28×28 image. In this section, we will flatten each image, treating them as 784 1D vectors. In the future, we will talk about more sophisticated strategies for exploiting the spatial structure in images, but for now we treat each pixel location as just another feature.

Recall that in softmax regression, we have as many outputs as there are categories. Because our dataset has 10 categories, our network will have an output dimension of 10. Consequently, our weights will constitute a 784×10 matrix and the biases will constitute a 1×10 vector. As with linear regression, we will initialize our weights W with Gaussian noise and our biases to take the initial value 0.

```
num_inputs = 784
num_outputs = 10

W = np.random.normal(0, 0.01, (num_inputs, num_outputs))
b = np.zeros(num_outputs)
```

Recall that we need to *attach gradients* to the model parameters. More literally, we are allocating memory for future gradients to be stored and notifiying MXNet that we will want to calculate gradients with respect to these parameters in the future.

```
W.attach_grad()
b.attach_grad()
```

3.6.2 The Softmax

Before implementing the softmax regression model, let's briefly review how operators such as `sum` work along specific dimensions in an `ndarray`. Given a matrix `X` we can sum over all elements (default) or only over elements in the same axis, *i.e.*, the column (`axis=0`) or the same row (`axis=1`). Note that if `X` is an array with shape `(2, 3)` and we sum over the columns (`X.sum(axis=0)`), the result will be a (1D) vector with shape `(3,)`. If we want to keep the number of axes in the original array (resulting in a 2D array with shape `(1, 3)`), rather than collapsing out the dimension that we summed over we can specify `keepdims=True` when invoking `sum`.

```
X = np.array([[1, 2, 3], [4, 5, 6]])
print(X.sum(axis=0, keepdims=True), '\n', X.sum(axis=1, keepdims=True))
```

```
[[5. 7. 9.]]
 [[ 6.]
 [15.]]
```

We are now ready to implement the softmax function. Recall that softmax consists of two steps: First, we exponentiate each term (using `exp`). Then, we sum over each row (we have one row per example in the batch) to get the normalization constants for each example. Finally, we divide each row by its normalization constant, ensuring that the result sums to 1. Before looking at the code, let's recall what this looks expressed as an equation:

$$\mathrm{softmax}(\mathbf{X})_{ij} = \frac{\exp(X_{ij})}{\sum_k \exp(X_{ik})}. \tag{3.6.1}$$

The denominator, or normalization constant, is also sometimes called the partition function (and its logarithm is called the log-partition function). The origins of that name are in statistical physics[59] where a related equation models the distribution over an ensemble of particles).

```
def softmax(X):
    X_exp = np.exp(X)
    partition = X_exp.sum(axis=1, keepdims=True)
    return X_exp / partition  # The broadcast mechanism is applied here
```

As you can see, for any random input, we turn each element into a non-negative number. Moreover, each row sums up to 1, as is required for a probability. Note that while this looks correct mathematically, we were a bit sloppy in our implementation because failed to take precautions against numerical overflow or underflow due to large (or very small) elements of the matrix, as we did in Section 17.8.

```
X = np.random.normal(size=(2, 5))
X_prob = softmax(X)
X_prob, X_prob.sum(axis=1)
```

```
(array([[0.22376052, 0.06659239, 0.06583703, 0.29964197, 0.3441681 ],
        [0.63209665, 0.03179282, 0.194987  , 0.09209415, 0.04902935]]),
 array([1.        , 0.99999994]))
```

3.6.3 The Model

Now that we have defined the softmax operation, we can implement the softmax regression model. The below code defines the forward pass through the network. Note that we flatten each original image in the batch into a vector with length `num_inputs` with the `reshape` function before passing the data through our model.

```
def net(X):
    return softmax(np.dot(X.reshape(-1, num_inputs), W) + b)
```

[59] https://en.wikipedia.org/wiki/Partition_function_(statistical_mechanics)

3.6.4 The Loss Function

Next, we need to implement the cross-entropy loss function, introduced in Section 3.4. This may be the most common loss function in all of deep learning because, at the moment, classification problems far outnumber regression problems.

Recall that cross-entropy takes the negative log likelihood of the predicted probability assigned to the true label $-\log P(y \mid x)$. Rather than iterating over the predictions with a Python for loop (which tends to be inefficient), we can use the pick function which allows us to easily select the appropriate terms from the matrix of softmax entries. Below, we illustrate the pick function on a toy example, with 3 categories and 2 examples.

```
y_hat = np.array([[0.1, 0.3, 0.6], [0.3, 0.2, 0.5]])
y_hat[[0, 1], [0, 2]]
```

```
array([0.1, 0.5])
```

Now we can implement the cross-entropy loss function efficiently with just one line of code.

```
def cross_entropy(y_hat, y):
    return - np.log(y_hat[range(len(y_hat)), y])
```

3.6.5 Classification Accuracy

Given the predicted probability distribution y_hat, we typically choose the class with highest predicted probability whenever we must output a *hard* prediction. Indeed, many applications require that we make a choice. Gmail must categorize an email into Primary, Social, Updates, or Forums. It might estimate probabilities internally, but at the end of the day it has to choose one among the categories.

When predictions are consistent with the actual category y, they are correct. The classification accuracy is the fraction of all predictions that are correct. Although it can be difficult optimize accuracy directly (it is not differentiable), it is often the performance metric that we care most about, and we will nearly always report it when training classifiers.

To compute accuracy we do the following: First, we execute y_hat.argmax(axis=1) to gather the predicted classes (given by the indices for the largest entires each row). The result has the same shape as the variable y. Now we just need to check how frequently the two match. Since the equality operator == is datatype-sensitive (e.g., an int and a float32 are never equal), we also need to convert both to the same type (we pick float32). The result is an ndarray containing entries of 0 (false) and 1 (true). Taking the mean yields the desired result.

```
# Saved in the d2l package for later use
def accuracy(y_hat, y):
    if y_hat.shape[1] > 1:
        return float((y_hat.argmax(axis=1) == y.astype('float32')).sum())
    else:
        return float((y_hat.astype('int32') == y.astype('int32')).sum())
```

We will continue to use the variables y_hat and y defined in the pick function, as the predicted probability distribution and label, respectively. We can see that the first example's prediction category is 2 (the largest element of the row is 0.6 with an index of 2), which is inconsistent with the actual label, 0. The second example's prediction category is 2 (the largest element of the row is 0.5 with an index of 2), which is consistent with the actual label, 2. Therefore, the classification accuracy rate for these two examples is 0.5.

```
y = np.array([0, 2])
accuracy(y_hat, y) / len(y)
```

```
0.5
```

Similarly, we can evaluate the accuracy for model `net` on the dataset (accessed via `data_iter`).

```
# Saved in the d2l package for later use
def evaluate_accuracy(net, data_iter):
    metric = Accumulator(2)  # num_corrected_examples, num_examples
    for X, y in data_iter:
        metric.add(accuracy(net(X), y), y.size)
    return metric[0] / metric[1]
```

Here `Accumulator` is a utility class to accumulated sum over multiple numbers.

```
# Saved in the d2l package for later use
class Accumulator(object):
    """Sum a list of numbers over time"""

    def __init__(self, n):
        self.data = [0.0] * n

    def add(self, *args):
        self.data = [a+b for a, b in zip(self.data, args)]

    def reset(self):
        self.data = [0] * len(self.data)

    def __getitem__(self, i):
        return self.data[i]
```

Because we initialized the `net` model with random weights, the accuracy of this model should be close to random guessing, i.e., 0.1 for 10 classes.

```
evaluate_accuracy(net, test_iter)
```

```
0.0811
```

3.6.6 Model Training

The training loop for softmax regression should look strikingly familiar if you read through our implementation of linear regression in Section 3.2. Here we refactor the implementation to make it reusable. First, we define a function to train for one data epoch. Note that `updater` is general function to update the model parameters, which accepts the batch size as an argument. It can be either a wrapper of `d2l.sgd` or a Gluon trainer.

```python
# Saved in the d2l package for later use
def train_epoch_ch3(net, train_iter, loss, updater):
    metric = Accumulator(3)  # train_loss_sum, train_acc_sum, num_examples
    if isinstance(updater, gluon.Trainer):
        updater = updater.step
    for X, y in train_iter:
        # Compute gradients and update parameters
        with autograd.record():
            y_hat = net(X)
            l = loss(y_hat, y)
        l.backward()
        updater(X.shape[0])
        metric.add(float(l.sum()), accuracy(y_hat, y), y.size)
    # Return training loss and training accuracy
    return metric[0]/metric[2], metric[1]/metric[2]
```

Before showing the implementation of the training function, we define a utility class that draw data in animation. Again, it aims to simplify the codes in later chapters.

```python
# Saved in the d2l package for later use
class Animator(object):
    def __init__(self, xlabel=None, ylabel=None, legend=[], xlim=None,
                 ylim=None, xscale='linear', yscale='linear', fmts=None,
                 nrows=1, ncols=1, figsize=(3.5, 2.5)):
        """Incrementally plot multiple lines."""
        d2l.use_svg_display()
        self.fig, self.axes = d2l.plt.subplots(nrows, ncols, figsize=figsize)
        if nrows * ncols == 1:
            self.axes = [self.axes, ]
        # Use a lambda to capture arguments
        self.config_axes = lambda: d2l.set_axes(
            self.axes[0], xlabel, ylabel, xlim, ylim, xscale, yscale, legend)
        self.X, self.Y, self.fmts = None, None, fmts

    def add(self, x, y):
        """Add multiple data points into the figure."""
        if not hasattr(y, "__len__"):
            y = [y]
        n = len(y)
        if not hasattr(x, "__len__"):
            x = [x] * n
        if not self.X:
            self.X = [[] for _ in range(n)]
        if not self.Y:
            self.Y = [[] for _ in range(n)]
        if not self.fmts:
            self.fmts = ['-'] * n
```

(continues on next page)

(continued from previous page)

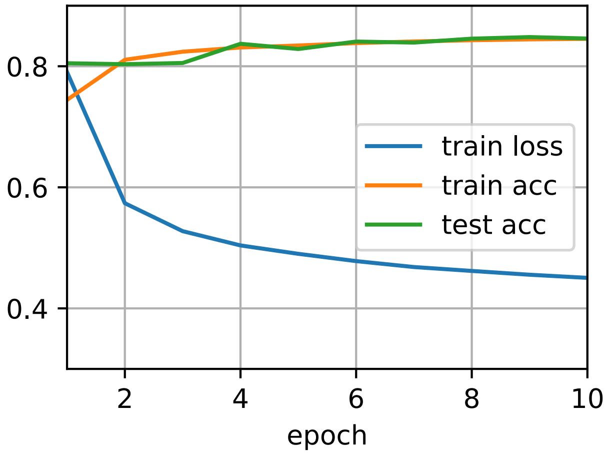
```
        for i, (a, b) in enumerate(zip(x, y)):
            if a is not None and b is not None:
                self.X[i].append(a)
                self.Y[i].append(b)
        self.axes[0].cla()
        for x, y, fmt in zip(self.X, self.Y, self.fmts):
            self.axes[0].plot(x, y, fmt)
        self.config_axes()
        display.display(self.fig)
        display.clear_output(wait=True)
```

The training function then runs multiple epochs and visualize the training progress.

```
# Saved in the d2l package for later use
def train_ch3(net, train_iter, test_iter, loss, num_epochs, updater):
    animator = Animator(xlabel='epoch', xlim=[1, num_epochs],
                        ylim=[0.3, 0.9],
                        legend=['train loss', 'train acc', 'test acc'])
    for epoch in range(num_epochs):
        train_metrics = train_epoch_ch3(net, train_iter, loss, updater)
        test_acc = evaluate_accuracy(net, test_iter)
        animator.add(epoch+1, train_metrics+(test_acc,))
```

Again, we use the minibatch stochastic gradient descent to optimize the loss function of the model. Note that the number of epochs (num_epochs), and learning rate (lr) are both adjustable hyperparameters. By changing their values, we may be able to increase the classification accuracy of the model. In practice we will want to split our data three ways into training, validation, and test data, using the validation data to choose the best values of our hyperparameters.

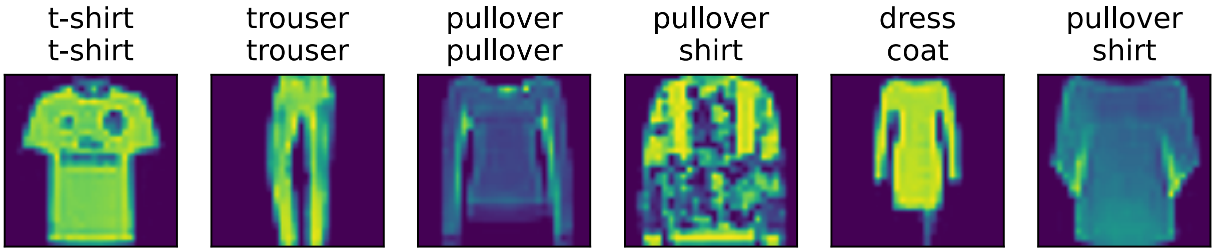
```
num_epochs, lr = 10, 0.1

def updater(batch_size):
    return d2l.sgd([W, b], lr, batch_size)

train_ch3(net, train_iter, test_iter, cross_entropy, num_epochs, updater)
```



3.6.7 Prediction

Now that training is complete, our model is ready to classify some images. Given a series of images, we will compare their actual labels (first line of text output) and the model predictions (second line of text output).

```
# Saved in the d2l package for later use
def predict_ch3(net, test_iter, n=6):
    for X, y in test_iter:
        break
    trues = d2l.get_fashion_mnist_labels(y)
    preds = d2l.get_fashion_mnist_labels(net(X).argmax(axis=1))
    titles = [true+'\n' + pred for true, pred in zip(trues, preds)]
    d2l.show_images(X[0:n].reshape(n, 28, 28), 1, n, titles=titles[0:n])

predict_ch3(net, test_iter)
```



Summary

With softmax regression, we can train models for multi-category classification. The training loop is very similar to that in linear regression: retrieve and read data, define models and loss functions, then train models using optimization algorithms. As you will soon find out, most common deep learning models have similar training procedures.

Exercises

1. In this section, we directly implemented the softmax function based on the mathematical definition of the softmax operation. What problems might this cause (hint: try to calculate the size of $\exp(50)$)?
2. The function `cross_entropy` in this section is implemented according to the definition of the cross-entropy loss function. What could be the problem with this implementation (hint: consider the domain of the logarithm)?
3. What solutions you can think of to fix the two problems above?
4. Is it always a good idea to return the most likely label. E.g. would you do this for medical diagnosis?
5. Assume that we want to use softmax regression to predict the next word based on some features. What are some problems that might arise from a large vocabulary?

3.7 Concise Implementation of Softmax Regression

Just as Gluon made it much easier to implement linear regression in Section 3.3, we will find it similarly (or possibly more) convenient for implementing classification models. Again, we begin with our import ritual.

```
import d2l
from mxnet import gluon, init, npx
from mxnet.gluon import nn
npx.set_np()
```

Let's stick with the Fashion-MNIST dataset and keep the batch size at 256 as in the last section.

```
batch_size = 256
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)
```

3.7.1 Initializing Model Parameters

As mentioned in Section 3.4, the output layer of softmax regression is a fully-connected (`Dense`) layer. Therefore, to implement our model, we just need to add one `Dense` layer with 10 outputs to our `Sequential`. Again, here, the `Sequential` is not really necessary, but we might as well form the habit since it will be ubiquitous when implementing deep models. Again, we initialize the weights at random with zero mean and standard deviation 0.01.

```
net = nn.Sequential()
net.add(nn.Dense(10))
net.initialize(init.Normal(sigma=0.01))
```

3.7.2 The Softmax

In the previous example, we calculated our model's output and then ran this output through the cross-entropy loss. Mathematically, that is a perfectly reasonable thing to do. However, from a computational perspective, exponentiation can be a source of numerical stability issues (as discussed in Section 17.8). Recall that the softmax function calculates $\hat{y}_j = \frac{e^{z_j}}{\sum_{i=1}^{n} e^{z_i}}$, where $\hat{y}_j$ is the j^{th} element of `yhat` and z_j is the j^{th} element of the input `y_linear` variable, as computed by the softmax.

If some of the z_i are very large (i.e., very positive), then e^{z_i} might be larger than the largest number we can have for certain types of `float` (i.e., overflow). This would make the denominator (and/or numerator) `inf` and we wind up encountering either 0, `inf`, or `nan` for $\hat{y}_j$. In these situations we do not get a well-defined return value for `cross_entropy`. One trick to get around this is to first subtract $\max(z_i)$ from all z_i before proceeding with the `softmax` calculation. You can verify that this shifting of each z_i by constant factor does not change the return value of `softmax`.

After the subtraction and normalization step, it might be that possible that some z_j have large negative values and thus that the corresponding e^{z_j} will take values close to zero. These might be rounded to zero due to finite precision (i.e underflow), making $\hat{y}_j$ zero and giving us `-inf` for $\log(\hat{y}_j)$. A few steps down the road in backpropagation, we might find ourselves faced with a screenful of the dreaded not-a-number (`nan`) results.

Fortunately, we are saved by the fact that even though we are computing exponential functions, we ultimately intend to take their log (when calculating the cross-entropy loss). By combining these two operators (`softmax` and `cross_entropy`) together, we can escape the numerical stability issues that might otherwise plague us during backpropagation. As shown in the equation below, we avoided calculating e^{z_j} and can instead z_j directly due to the canceling in $\log(\exp(\cdot))$.

$$\begin{aligned}
\log\left(\hat{y}_j\right) &= \log\left(\frac{e^{z_j}}{\sum_{i=1}^{n} e^{z_i}}\right) \\
&= \log\left(e^{z_j}\right) - \log\left(\sum_{i=1}^{n} e^{z_i}\right) \\
&= z_j - \log\left(\sum_{i=1}^{n} e^{z_i}\right).
\end{aligned} \tag{3.7.1}$$

We will want to keep the conventional softmax function handy in case we ever want to evaluate the probabilities output by our model. But instead of passing softmax probabilities into our new loss function, we will just pass the logits and compute the softmax and its log all at once inside the softmax_cross_entropy loss function, which does smart things like the log-sum-exp trick (see on Wikipedia[61]).

```
loss = gluon.loss.SoftmaxCrossEntropyLoss()
```

3.7.3 Optimization Algorithm

Here, we use minibatch stochastic gradient descent with a learning rate of 0.1 as the optimization algorithm. Note that this is the same as we applied in the linear regression example and it illustrates the general applicability of the optimizers.

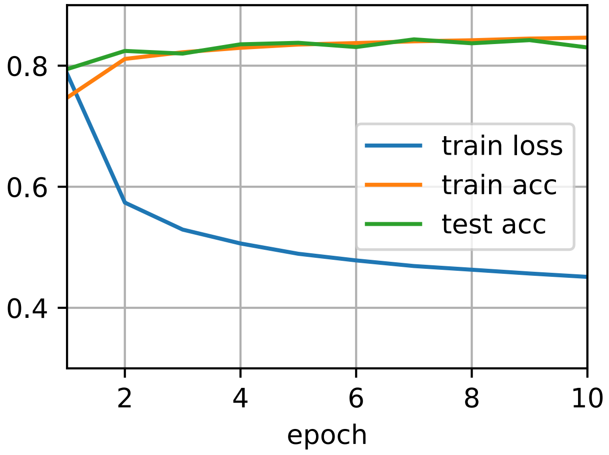
```
trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': 0.1})
```

3.7.4 Training

Next we call the training function defined in the last section to train a model.

```
num_epochs = 10
d2l.train_ch3(net, train_iter, test_iter, loss, num_epochs, trainer)
```

[61] https://en.wikipedia.org/wiki/LogSumExp



As before, this algorithm converges to a solution that achieves an accuracy of 83.7%, albeit this time with fewer lines of code than before. Note that in many cases, Gluon takes additional precautions beyond these most well-known tricks to ensure numerical stability, saving us from even more pitfalls that we would encounter if we tried to code all of our models from scratch in practice.

Exercises

1. Try adjusting the hyper-parameters, such as batch size, epoch, and learning rate, to see what the results are.
2. Why might the test accuracy decrease again after a while? How could we fix this?

4 | Multilayer Perceptrons

In this chapter, we will introduce your first truly *deep* networks. The simplest deep networks are called multilayer perceptrons, and they consist of many layers of neurons each fully connected to those in the layer below (from which they receive input) and those above (which they, in turn, influence). When we train high-capacity models we run the risk of overfitting. Thus, we will need to provide your first rigorous introduction to the notions of overfitting, underfitting, and capacity control. To help you combat these problems, we will introduce regularization techniques such as dropout and weight decay. We will also discuss issues relating to numerical stability and parameter initialization that are key to successfully training deep networks. Throughout, we focus on applying models to real data, aiming to give the reader a firm grasp not just of the concepts but also of the practice of using deep networks. We punt matters relating to the computational performance, scalability and efficiency of our models to subsequent chapters.

4.1 Multilayer Perceptron

In the previous chapters, we showed how you could implement multiclass logistic regression (also called softmax regression) for classifying images of clothing into the 10 possible categories. To get there, we had to learn how to wrangle data, coerce our outputs into a valid probability distribution (via `softmax`), how to apply an appropriate loss function, and how to optimize over our parameters. Now that we've covered these preliminaries, we are free to focus our attention on the more exciting enterprise of designing powerful models using deep neural networks.

4.1.1 Hidden Layers

Let's recall linear regression and softmax regression with an example as illustrated in Fig. 4.1.1. In general, we mapped our inputs directly to our outputs via a single linear transformation:

$$\hat{\mathbf{o}} = \mathrm{softmax}(\mathbf{W}\mathbf{x} + \mathbf{b}). \tag{4.1.1}$$

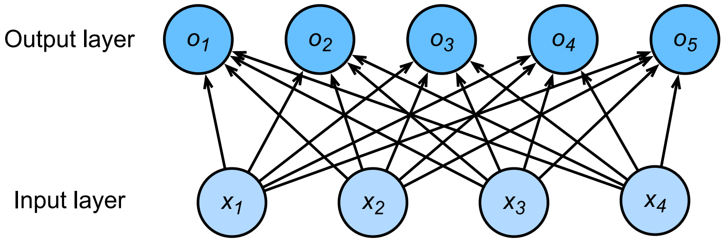


Fig. 4.1.1: Single layer perceptron with 5 output units.

If our labels really were related to our input data by an approximately linear function, then this approach would be perfect. But linearity is a *strong assumption*. Linearity implies that for whatever target value we are trying to predict, increasing the value of each of our inputs should either drive the value of the output up or drive it down, irrespective of the value of the other inputs.

Sometimes this makes sense! Say we are trying to predict whether an individual will or will not repay a loan. We might reasonably imagine that all else being equal, an applicant with a higher income would be more likely to repay than one with a lower income. In these cases, linear models might perform well, and they might even be hard to beat.

But what about classifying images in FashionMNIST? Should increasing the intensity of the pixel at location (13, 17) always increase the likelihood that the image depicts a pocketbook? That seems ridiculous because we all know that you cannot make sense out of an image without accounting for the interactions among pixels.

From One to Many

As another case, consider trying to classify images based on whether they depict *cats* or *dogs* given black-and-white images.

If we use a linear model, we'd basically be saying that for each pixel, increasing its value (making it more white) must always increase the probability that the image depicts a dog or must always increase the probability that the image depicts a cat. We would be making the absurd assumption that the only requirement for differentiating cats vs. dogs is to assess how bright they are. That approach is doomed to fail in a work that contains both black dogs and black cats, and both white dogs and white cats.

Teasing out what is depicted in an image generally requires allowing more complex relationships between our inputs and outputs. Thus we need models capable of discovering patterns that might be characterized by interactions among the many features. We can over come these limitations of linear models and handle a more general class of functions by incorporating one or more hidden layers. The easiest way to do this is to stack many layers of neurons on top of each other. Each layer feeds into the layer above it, until we generate an output. This architecture is commonly called a *multilayer perceptron*, often abbreviated as *MLP*. The neural network diagram for an MLP looks like Fig. 4.1.2.

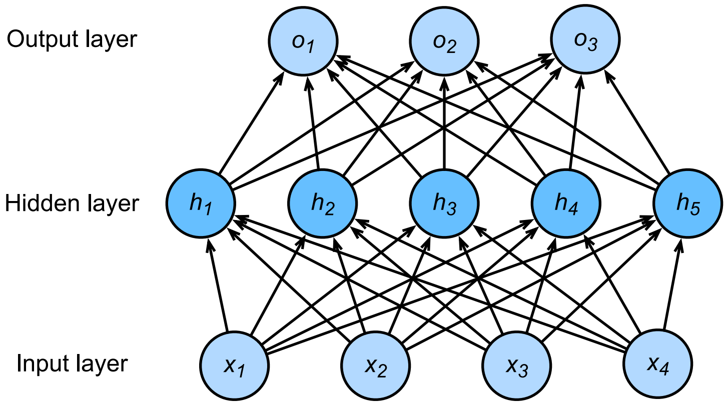


Fig. 4.1.2: Multilayer perceptron with hidden layers. This example contains a hidden layer with 5 hidden units in it.

The multilayer perceptron above has 4 inputs and 3 outputs, and the hidden layer in the middle contains 5 hidden units. Since the input layer does not involve any calculations, building this network would consist of implementing 2 layers of computation. The neurons in the input layer are fully connected to the inputs in the hidden layer. Likewise, the neurons in the hidden layer are fully connected to the neurons in the output layer.

From Linear to Nonlinear

We can write out the calculations that define this one-hidden-layer MLP in mathematical notation as follows:

$$\begin{aligned} \mathbf{h} &= \mathbf{W}_1 \mathbf{x} + \mathbf{b}_1, \\ \mathbf{o} &= \mathbf{W}_2 \mathbf{h} + \mathbf{b}_2, \\ \hat{\mathbf{y}} &= \mathrm{softmax}(\mathbf{o}). \end{aligned} \tag{4.1.2}$$

By adding another layer, we have added two new sets of parameters, but what have we gained in exchange? In the model defined above, we do not achieve anything for our troubles!

That is because our hidden units are just a linear function of the inputs and the outputs (pre-softmax) are just a linear function of the hidden units. A linear function of a linear function is itself a linear function. That means that for any values of the weights, we could just collapse out the hidden layer yielding an equivalent single-layer model using $\mathbf{W} = \mathbf{W}_2 \mathbf{W}_1$ and $\mathbf{b} = \mathbf{W}_2 \mathbf{b}_1 + \mathbf{b}_2$.

$$\mathbf{o} = \mathbf{W}_2 \mathbf{h} + \mathbf{b}_2 = \mathbf{W}_2 (\mathbf{W}_1 \mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2 = (\mathbf{W}_2 \mathbf{W}_1) \mathbf{x} + (\mathbf{W}_2 \mathbf{b}_1 + \mathbf{b}_2) = \mathbf{W} \mathbf{x} + \mathbf{b}. \tag{4.1.3}$$

In order to get a benefit from multilayer architectures, we need another key ingredient—a nonlinearity σ to be applied to each of the hidden units after each layer's linear transformation. The most popular choice for the nonlinearity these days is the rectified linear unit (ReLU) max$(x, 0)$. After incorporating these non-linearities it becomes impossible to merge layers.

$$\begin{aligned} \mathbf{h} &= \sigma(\mathbf{W}_1 \mathbf{x} + \mathbf{b}_1), \\ \mathbf{o} &= \mathbf{W}_2 \mathbf{h} + \mathbf{b}_2, \\ \hat{\mathbf{y}} &= \mathrm{softmax}(\mathbf{o}). \end{aligned} \tag{4.1.4}$$

Clearly, we could continue stacking such hidden layers, e.g., $\mathbf{h}_1 = \sigma(\mathbf{W}_1 \mathbf{x} + \mathbf{b}_1)$ and $\mathbf{h}_2 = \sigma(\mathbf{W}_2 \mathbf{h}_1 + \mathbf{b}_2)$ on top of each other to obtain a true multilayer perceptron.

Multilayer perceptrons can account for complex interactions in the inputs because the hidden neurons depend on the values of each of the inputs. It's easy to design a hidden node that does arbitrary computation, such as, for instance, logical operations on its inputs. Moreover, for certain choices of the activation function it's widely known that multilayer perceptrons are universal approximators. That means that even for a single-hidden-layer neural network, with enough nodes, and the right set of weights, we can model any function at all! *Actually learning that function is the hard part.*

Moreover, just because a single-layer network *can* learn any function does not mean that you should try to solve all of your problems with single-layer networks. It turns out that we can approximate many functions much more compactly if we use deeper (vs wider) neural networks. We'll get more into the math in a subsequent chapter, but for now let's actually build an MLP. In this example, we'll implement a multilayer perceptron with two hidden layers and one output layer.

Vectorization and Minibatch

As before, by the matrix $\mathbf{X}$, we denote a minibatch of inputs. The calculations to produce outputs from an MLP with two hidden layers can thus be expressed:

$$\begin{aligned}
\mathbf{H}_1 &= \sigma(\mathbf{W}_1\mathbf{X} + \mathbf{b}_1), \\
\mathbf{H}_2 &= \sigma(\mathbf{W}_2\mathbf{H}_1 + \mathbf{b}_2), \\
\mathbf{O} &= \mathrm{softmax}(\mathbf{W}_3\mathbf{H}_2 + \mathbf{b}_3).
\end{aligned} \tag{4.1.5}$$

With some abuse of notation, we define the nonlinearity σ to apply to its inputs on a row-wise fashion, i.e., one observation at a time. Note that we are also using the notation for *softmax* in the same way to denote a row-wise operation. Often, as in this section, the activation functions that we apply to hidden layers are not merely row-wise, but component wise. That means that after computing the linear portion of the layer, we can calculate each nodes activation without looking at the values taken by the other hidden units. This is true for most activation functions (the batch normalization operation will be introduced in Section 7.5 is a notable exception to that rule).

```
%matplotlib inline
import d2l
from mxnet import autograd, np, npx
npx.set_np()
```

4.1.2 Activation Functions

Because they are so fundamental to deep learning, before going further, let's take a brief look at some common activation functions.

ReLU Function

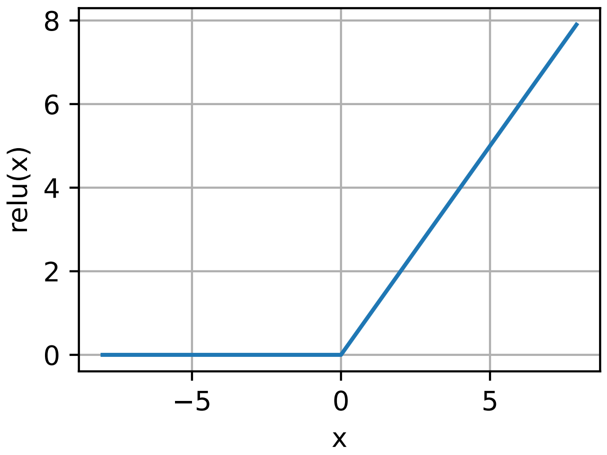
As stated above, the most popular choice, due to its simplicity of implementation and its efficacy in training is the rectified linear unit (ReLU). ReLUs provide a very simple nonlinear transformation. Given the element z, the function is defined as the maximum of that element and 0.

$$\mathrm{ReLU}(z) = \max(z, 0). \tag{4.1.6}$$

It can be understood that the ReLU function retains only positive elements and discards negative elements (setting those nodes to 0). To get a better idea of what it looks like, we can plot it.

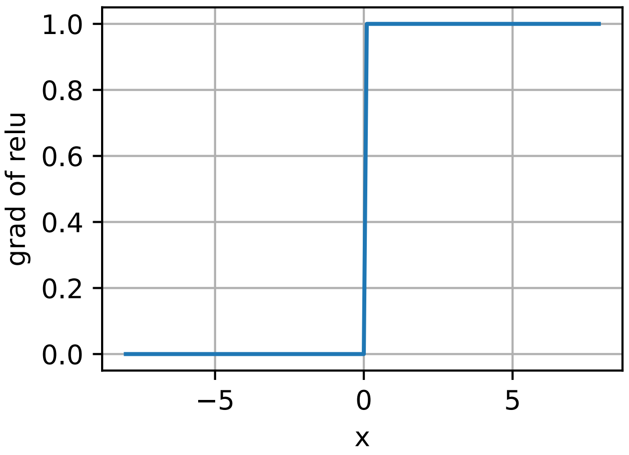
Because it is used so commonly, NDarray supports the `relu` function as a basic native operator. As you can see, the activation function is piece-wise linear.

```
x = np.arange(-8.0, 8.0, 0.1)
x.attach_grad()
with autograd.record():
    y = npx.relu(x)
d2l.set_figsize((4, 2.5))
d2l.plot(x, y, 'x', 'relu(x)')
```



When the input is negative, the derivative of ReLU function is 0 and when the input is positive, the derivative of ReLU function is 1. Note that the ReLU function is not differentiable when the input takes value precisely equal to 0. In these cases, we go with the left-hand-side (LHS) derivative and say that the derivative is 0 when the input is 0. We can get away with this because the input may never actually be zero. There is an old adage that if subtle boundary conditions matter, we are probably doing (*real*) mathematics, not engineering. That conventional wisdom may apply here. See the derivative of the ReLU function plotted below.

```
y.backward()
d2l.plot(x, x.grad, 'x', 'grad of relu')
```



Note that there are many variants to the ReLU function, such as the parameterized ReLU (pReLU) of He et al., 2015[63]. This variation adds a linear term to the ReLU, so some information still gets through, even when the argument is negative.

$$\text{pReLU}(x) = \max(0, x) + \alpha \min(0, x). \tag{4.1.7}$$

The reason for using the ReLU is that its derivatives are particularly well behaved: either they vanish or they just let the argument through. This makes optimization better behaved and it reduces the issue of the vanishing gradient problem (more on this later).

[63] https://arxiv.org/abs/1502.01852

Sigmoid Function

The sigmoid function transforms its inputs which take values in $\mathbb{R}$ to the interval $(0, 1)$. For that reason, the sigmoid is often called a *squashing* function: it squashes any input in the range (-inf, inf) to some value in the range (0, 1).

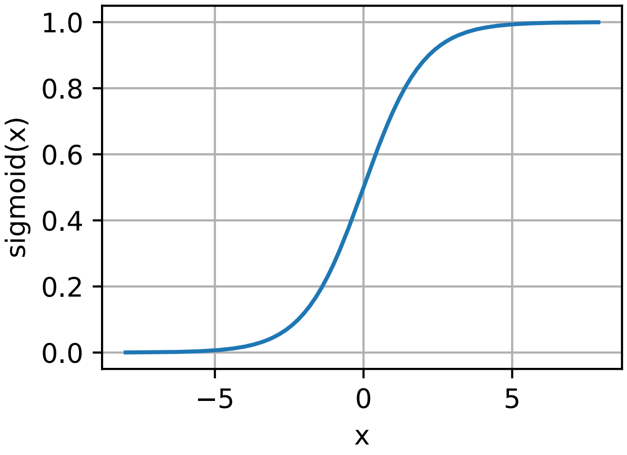
$$\text{sigmoid}(x) = \frac{1}{1 + \exp(-x)}. \tag{4.1.8}$$

In the earliest neural networks, scientists were interested in modeling biological neurons which either *fire* or *do not fire*. Thus the pioneers of this field, going all the way back to McCulloch and Pitts in the 1940s, were focused on thresholding units. A thresholding function takes either value 0 (if the input is below the threshold) or value 1 (if the input exceeds the threshold)

When attention shifted to gradient based learning, the sigmoid function was a natural choice because it is a smooth, differentiable approximation to a thresholding unit. Sigmoids are still common as activation functions on the output units, when we want to interpret the outputs as probabilities for binary classification problems (you can think of the sigmoid as a special case of the softmax) but the sigmoid has mostly been replaced by the simpler and easier to train ReLU for most use in hidden layers. In the "Recurrent Neural Network" chapter, we will describe how sigmoid units can be used to control the flow of information in a neural network thanks to its capacity to transform the value range between 0 and 1.

See the sigmoid function plotted below. When the input is close to 0, the sigmoid function approaches a linear transformation.

```
with autograd.record():
    y = npx.sigmoid(x)
d2l.plot(x, y, 'x', 'sigmoid(x)')
```

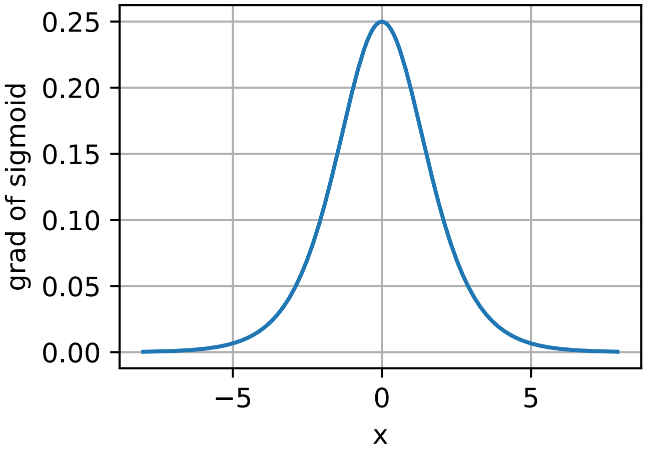


The derivative of sigmoid function is given by the following equation:

$$\frac{d}{dx}\text{sigmoid}(x) = \frac{\exp(-x)}{(1 + \exp(-x))^2} = \text{sigmoid}(x)\left(1 - \text{sigmoid}(x)\right). \tag{4.1.9}$$

The derivative of sigmoid function is plotted below. Note that when the input is 0, the derivative of the sigmoid function reaches a maximum of 0.25. As the input diverges from 0 in either direction, the derivative approaches 0.

```
y.backward()
d2l.plot(x, x.grad, 'x', 'grad of sigmoid')
```



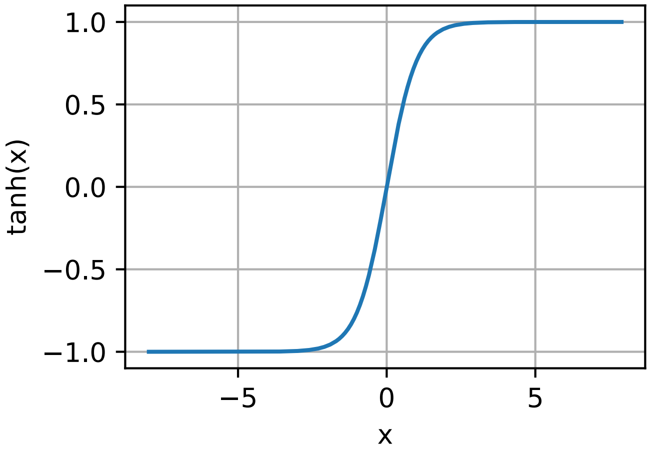
Tanh Function

Like the sigmoid function, the tanh (Hyperbolic Tangent) function also squashes its inputs, transforms them into elements on the interval between -1 and 1:

$$\tanh(x) = \frac{1 - \exp(-2x)}{1 + \exp(-2x)}. \tag{4.1.10}$$

We plot the tanh function blow. Note that as the input nears 0, the tanh function approaches a linear transformation. Although the shape of the function is similar to the sigmoid function, the tanh function exhibits point symmetry about the origin of the coordinate system.

```
with autograd.record():
    y = np.tanh(x)
d2l.plot(x, y, 'x', 'tanh(x)')
```

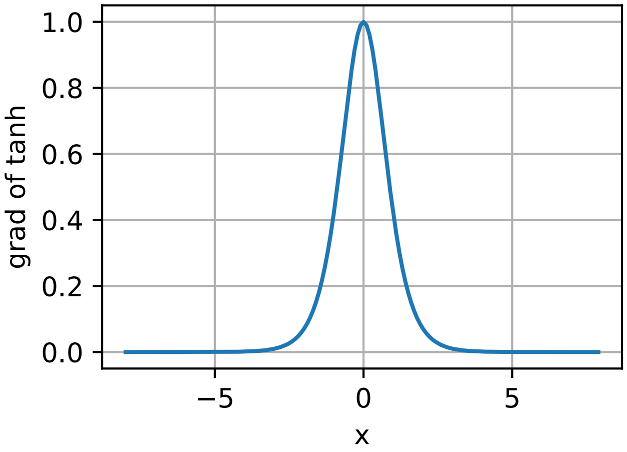


The derivative of the Tanh function is:

$$\frac{d}{dx}\tanh(x) = 1 - \tanh^2(x). \tag{4.1.11}$$

The derivative of tanh function is plotted below. As the input nears 0, the derivative of the tanh function approaches a maximum of 1. And as we saw with the sigmoid function, as the input moves away from 0 in either direction, the derivative of the tanh function approaches 0.

```
y.backward()
d2l.plot(x, x.grad, 'x', 'grad of tanh')
```



In summary, we now know how to incorporate nonlinearities to build expressive multilayer neural network architectures. As a side note, your knowledge now already puts you in command of the state of the art in deep learning, circa 1990. In fact, you have an advantage over anyone working the 1990s, because you can leverage powerful open-source deep learning frameworks to build models rapidly, using only a few lines of code. Previously, getting these nets training required researchers to code up thousands of lines of C and Fortran.

Summary

- The multilayer perceptron adds one or multiple fully-connected hidden layers between the output and input layers and transforms the output of the hidden layer via an activation function.
- Commonly-used activation functions include the ReLU function, the sigmoid function, and the tanh function.

Exercises

1. Compute the derivative of the tanh and the pReLU activation function.
2. Show that a multilayer perceptron using only ReLU (or pReLU) constructs a continuous piecewise linear function.
3. Show that $\tanh(x) + 1 = 2\text{sigmoid}(2x)$.
4. Assume we have a multilayer perceptron *without* nonlinearities between the layers. In particular, assume that we have d input dimensions, d output dimensions and that one of the layers had only $d/2$ dimensions. Show that this network is less expressive (powerful) than a single layer perceptron.

5. Assume that we have a nonlinearity that applies to one minibatch at a time. What kinds of problems do you expect this to cause?

4.2 Implementation of Multilayer Perceptron from Scratch

Now that we know how multilayer perceptrons (MLPs) work in theory, let's implement them. First, we import the required packages.

```
import d2l
from mxnet import gluon, np, npx
npx.set_np()
```

To compare against the results we previously achieved with vanilla softmax regression, we continue to use the Fashion-MNIST image classification dataset.

```
batch_size = 256
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)
```

4.2.1 Initializing Model Parameters

Recall that this dataset contains 10 classes and that each image consists of a $28 \times 28 = 784$ grid of pixel values. Since we will be discarding the spatial structure (for now), we can just think of this as a classification dataset with 784 input features and 10 classes. In particular we will implement our MLP with one hidden layer and 256 hidden units. Note that we can regard both of these choices as *hyperparameters* that could be set based on performance on validation data. Typically, we will choose layer widths as powers of 2 to make everything align nicely in memory.

Again, we will represent our parameters with several ndarrays. Note that we now have one weight matrix and one bias vector *per layer*. As always, we must call attach_grad to allocate memory for the gradients with respect to these parameters.

```
num_inputs, num_outputs, num_hiddens = 784, 10, 256

W1 = np.random.normal(scale=0.01, size=(num_inputs, num_hiddens))
b1 = np.zeros(num_hiddens)
W2 = np.random.normal(scale=0.01, size=(num_hiddens, num_outputs))
b2 = np.zeros(num_outputs)
params = [W1, b1, W2, b2]

for param in params:
    param.attach_grad()
```

4.2.2 Activation Function

To make sure we know how everything works, we will use the maximum function to implement ReLU ourselves, instead of invoking npx.relu directly.

```
def relu(X):
    return np.maximum(X, 0)
```

4.2.3 The model

As in softmax regression, we will reshape each 2D image into a flat vector of length num_inputs. Finally, we can implement our model with just a few lines of code.

```
def net(X):
    X = X.reshape(-1, num_inputs)
    H = relu(np.dot(X, W1) + b1)
    return np.dot(H, W2) + b2
```

4.2.4 The Loss Function

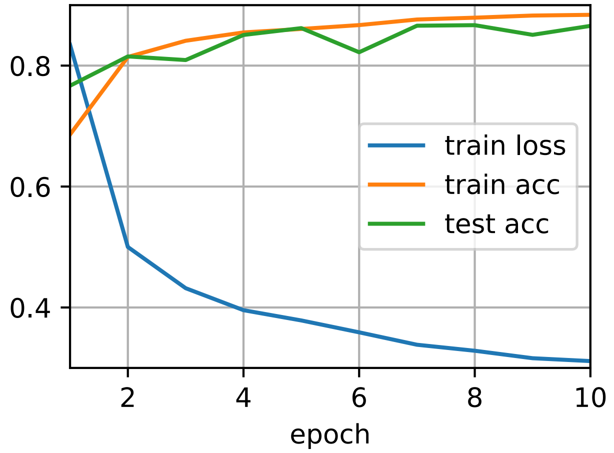
For better numerical stability and because we already know how to implement softmax regression completely from scratch in Section 3.6, we will use Gluon's integrated function for calculating the softmax and cross-entropy loss. Recall that we discussed some of these intricacies in Section 4.1. We encourage the interested reader to examing the source code for mxnet.gluon.loss.SoftmaxCrossEntropyLoss for more details.

```
loss = gluon.loss.SoftmaxCrossEntropyLoss()
```

4.2.5 Training

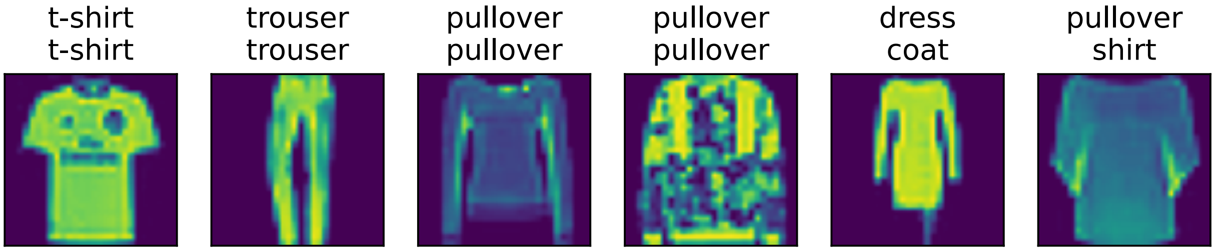
Steps for training the MLP are no different than for softmax regression. In the d2l package, we directly call the train_ch3 function, whose implementation was introduced in Section 3.6. We set the number of epochs to 10 and the learning rate to 0.5.

```
num_epochs, lr = 10, 0.5
d2l.train_ch3(net, train_iter, test_iter, loss, num_epochs,
              lambda batch_size: d2l.sgd(params, lr, batch_size))
```



To see how well we did, let's apply the model to some test data. If you are interested, compare the result to corresponding linear model in Section 3.6.

```
d2l.predict_ch3(net, test_iter)
```



This looks a bit better than our previous result, a good sign that we are on the right path.

Summary

We saw that implementing a simple MLP is easy, even when done manually. That said, with a large number of layers, this can get messy (e.g., naming and keeping track of the model parameters, etc).

Exercises

1. Change the value of the hyperparameter `num_hiddens` in order to see how this hyperparameter influences your results.
2. Try adding a new hidden layer to see how it affects the results.
3. How does changing the learning rate change the result?
4. What is the best result you can get by optimizing over all the parameters (learning rate, iterations, number of hidden layers, number of hidden units per layer)?

4.3 Concise Implementation of Multilayer Perceptron

Now that we learned how multilayer perceptrons (MLPs) work in theory, let's implement them. We begin, as always, by importing modules.

```
import d2l
from mxnet import gluon, init, npx
from mxnet.gluon import nn
npx.set_np()
```

4.3.1 The Model

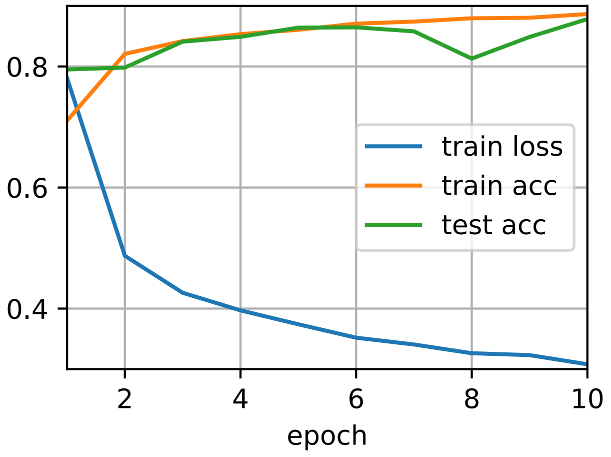
The only difference from our softmax regression implementation is that we add two `Dense` (fully-connected) layers instead of one. The first is our hidden layer, which has *256* hidden units and uses the ReLU activation function.

```
net = nn.Sequential()
net.add(nn.Dense(256, activation='relu'),
        nn.Dense(10))
net.initialize(init.Normal(sigma=0.01))
```

Again, note that as always, Gluon automatically infers the missing input dimensions to each layer.

Training the model follows the exact same steps as in our softmax regression implementation.

```
batch_size, num_epochs = 256, 10
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)
loss = gluon.loss.SoftmaxCrossEntropyLoss()
trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': 0.5})
d2l.train_ch3(net, train_iter, test_iter, loss, num_epochs, trainer)
```



Exercises

1. Try adding a few more hidden layers to see how the result changes.
2. Try out different activation functions. Which ones work best?
3. Try out different initializations of the weights.

4.4 Model Selection, Underfitting and Overfitting

As machine learning scientists, our goal is to discover general patterns. Say, for example, that we wish to learn the pattern that associates genetic markers with the development of dementia in adulthood. It is easy enough to memorize our training set. Each person's genes uniquely identify them, not just among people represented in our dataset, but among all people on earth!

Given the genetic markers representing some person, we do not want our model to simply recognize "oh, that is Bob", and then output the classification, say among {*dementia*, *mild cognitive impairment*, *healthy*}, that corresponds to Bob. Rather, our goal is to discover patterns that capture regularities in the underlying population from which our training set was drawn. If we are successfully in this endeavor, then we could successfully assess risk even for individuals that we have never encountered before. This problem—how to discover patterns that *generalize*—is the fundamental problem of machine learning.

The danger is that when we train models, we access just a small sample of data. The largest public image datasets contain roughly one million images. And more often we have to learn from thousands or tens of thousands. In a large hospital system we might access hundreds of thousands of medical records. With finite samples, we always run the risk that we might discover *apparent* associations that turn out not to hold up when we collect more data.

Let's consider an extreme pathological case. Imagine that you want to learn to predict which people will repay their loans. A lender hires you as a data scientist to investigate, handing over the complete files on 100 applicants, 5 of which defaulted on their loans within 3 years. Realistically, the files might include hundreds of potential features, including income, occupation, credit score, length of employment etc. Moreover, say that they additionally hand over video footage of each applicant's interview with their lending agent.

Now suppose that after featurizing the data into an enormous design matrix, you discover that of the 5 applicants who default, all of them were wearing blue shirts during their interviews, while only 40% of general population wore blue shirts. There is a good chance that if you train a predictive model to predict default, it might rely upon blue-shirt-wearing as an important feature.

Even if in fact defaulters were no more likely to wear blue shirts than people in the general population, there's a $.4^5 = .01$ probability that we would observe all five defaulters wearing blue shirts. With just 5 positive examples of defaults and hundreds or thousands of features, we would probably find a large number of features that appear to be perfectly predictive of our labor just due to random chance. With an unlimited amount of data, we would expect these *spurious* associations to eventually disappear. But we seldom have that luxury.

The phenomena of fitting our training data more closely than we fit the underlying distribution is called overfitting, and the techniques used to combat overfitting are called regularization. In the previous sections, you might have observed this effect while experimenting with the Fashion-MNIST dataset. If you altered the model structure or the hyper-parameters during the experiment, you might have noticed that with enough nodes, layers, and training epochs, the model can eventually reach perfect accuracy on the training set, even as the accuracy on test data deteriorates.

4.4.1 Training Error and Generalization Error

In order to discuss this phenomenon more formally, we need to differentiate between *training error* and *generalization error*. The training error is the error of our model as calculated on the training dataset, while generalization error is the expectation of our model's error were we to apply it to an infinite stream of additional data points drawn from the same underlying data distribution as our original sample.

Problematically, *we can never calculate the generalization error exactly*. That is because the imaginary stream of infinite data is an imaginary object. In practice, we must *estimate* the generalization error by applying our model to an independent test set constituted of a random selection of data points that were withheld from our training set.

The following three thought experiments will help illustrate this situation better. Consider a college student trying to prepare for his final exam. A diligent student will strive to practice well and test her abilities using exams from previous years. Nonetheless, doing well on past exams is no guarantee that she will excel when it matters. For instance, the student might try to prepare by rote learning the answers to the exam questions. This requires the student to memorize many things. She might even remember the answers for past exams perfectly. Another student might prepare by trying to understand the reasons for giving certain answers. In most cases, the latter student will do much better.

Likewise, consider a model that simply uses a lookup table to answer questions. If the set of allowable inputs is discrete and reasonably small, then perhaps after viewing *many* training examples, this approach would perform well. Still this model has no ability to do better than random guessing when faced with examples that it has never seen before. In reality the input spaces are far too large to memorize the answers corresponding to every conceivable input. For example, consider the black and white 28×28 images. If each pixel can take one among 256 gray scale values, then there are 256^{784} possible images. That means that there are far more low-res grayscale thumbnail-sized images than there are atoms in the universe. Even if we could encounter this data, we could never afford to store the lookup table.

Last, consider the problem of trying to classify the outcomes of coin tosses (class 0: heads, class 1: tails) based on some contextual features that might be available. No matter what algorithm we come up with, because the generalization error will always be $\frac{1}{2}$. However, for most algorithms, we should expect our training error to be considerably lower, depending on the luck of the draw, even if we did not have any features! Consider the dataset {0, 1, 1, 1, 0, 1}. Our feature-less would have to fall back on always predicting the *majority class*, which appears from our limited sample to be *1*. In this case, the model that always predicts class 1 will incur an error of $\frac{1}{3}$, considerably better than our generalization error. As we increase the amount of data, the probability that the fraction of heads will deviate significantly from $\frac{1}{2}$ diminishes, and our training error would come to match the generalization error.

Statistical Learning Theory

Since generalization is the fundamental problem in machine learning, you might not be surprised to learn that many mathematicians and theorists have dedicated their lives to developing formal theories to describe this phenomenon. In their eponymous theorem[67], Glivenko and Cantelli derived the rate at which the training error converges to the generalization error. In a series of seminal papers, Vapnik and Chervonenkis[68] extended this theory to more general classes of functions. This work laid the foundations of Statistical Learning Theory[69].

In the *standard supervised learning setting*, which we have addressed up until now and will stick throughout most of this book, we assume that both the training data and the test data are drawn *independently* from *identical* distributions (commonly called the i.i.d. assumption). This means that the process that samples our data has no *memory*. The 2nd example drawn and the 3rd drawn are no more correlated than the 2nd and the 2-millionth sample drawn.

Being a good machine learning scientist requires thinking critically, and already you should be poking holes in this assumption, coming up with common cases where the assumption fails. What if we train a mortality risk predictor on data collected from patients at UCSF, and apply it on patients at Massachusetts General Hospital? These distributions are simply not identical. Moreover, draws might be correlated in time. What if we are classifying the topics of Tweets. The news cycle would create temporal dependencies in the topics being discussed violating any assumptions of independence.

Sometimes we can get away with minor violations of the i.i.d. assumption and our models will continue to work remarkably well. After all, nearly every real-world application involves at least some minor violation of the i.i.d. assumption, and yet we have useful tools for face recognition, speech recognition, language translation, etc.

Other violations are sure to cause trouble. Imagine, for example, if we tried to train a face recognition system by training it exclusively on university students and then want to deploy it as a tool for monitoring geriatrics in a nursing home population. This is unlikely to work well since college students tend to look considerably different from the elderly.

In subsequent chapters and volumes, we will discuss problems arising from violations of the i.i.d. assumption. For now, even taking the i.i.d. assumption for granted, understanding generalization is a formidable problem. Moreover, elucidating the precise theoretical foundations that might explain why deep neural networks generalize as well as they do continues to vexes the greatest minds in learning theory.

When we train our models, we attempt searching for a function that fits the training data as well as possible. If the function is so flexible that it can catch on to spurious patterns just as easily as to the true associations, then it might perform *too well* without producing a model that generalizes well to unseen data. This is precisely what we want to avoid (or at least control). Many of the techniques in deep learning are heuristics and tricks aimed at guarding against overfitting.

[67] https://en.wikipedia.org/wiki/Glivenko%E2%80%93Cantelli_theorem

[68] https://en.wikipedia.org/wiki/Vapnik%E2%80%93Chervonenkis_theory

[69] https://en.wikipedia.org/wiki/Statistical_learning_theory

Model Complexity

When we have simple models and abundant data, we expect the generalization error to resemble the training error. When we work with more complex models and fewer examples, we expect the training error to go down but the generalization gap to grow. What precisely constitutes model complexity is a complex matter. Many factors govern whether a model will generalize well. For example a model with more parameters might be considered more complex. A model whose parameters can take a wider range of values might be more complex. Often with neural networks, we think of a model that takes more training steps as more complex, and one subject to *early stopping* as less complex.

It can be difficult to compare the complexity among members of substantially different model classes (say a decision tree versus a neural network). For now, a simple rule of thumb is quite useful: A model that can readily explain arbitrary facts is what statisticians view as complex, whereas one that has only a limited expressive power but still manages to explain the data well is probably closer to the truth. In philosophy, this is closely related to Popper's criterion of falsifiability[70] of a scientific theory: a theory is good if it fits data and if there are specific tests which can be used to disprove it. This is important since all statistical estimation is post hoc[71], i.e., we estimate after we observe the facts, hence vulnerable to the associated fallacy. For now, we will put the philosophy aside and stick to more tangible issues.

In this section, to give you some intuition, we'll focus on a few factors that tend to influence the generalizability of a model class:

1. The number of tunable parameters. When the number of tunable parameters, sometimes called the *degrees of freedom*, is large, models tend to be more susceptible to overfitting.
2. The values taken by the parameters. When weights can take a wider range of values, models can be more susceptible to over fitting.
3. The number of training examples. It's trivially easy to overfit a dataset containing only one or two examples even if your model is simple. But overfitting a dataset with millions of examples requires an extremely flexible model.

4.4.2 Model Selection

In machine learning, we usually select our final model after evaluating several candidate models. This process is called model selection. Sometimes the models subject to comparison are fundamentally different in nature (say, decision trees vs linear models). At other times, we are comparing members of the same class of models that have been trained with different hyperparameter settings.

With multilayer perceptrons for example, we may wish to compare models with different numbers of hidden layers, different numbers of hidden units, and various choices of the activation functions applied to each hidden layer. In order to determine the best among our candidate models, we will typically employ a validation set.

[70] https://en.wikipedia.org/wiki/Falsifiability

[71] https://en.wikipedia.org/wiki/Post_hoc

Validation Dataset

In principle we should not touch our test set until after we have chosen all our hyper-parameters. Were we to use the test data in the model selection process, there is a risk that we might overfit the test data. Then we would be in serious trouble. If we overfit our training data, there is always the evaluation on test data to keep us honest. But if we overfit the test data, how would we ever know?

Thus, we should never rely on the test data for model selection. And yet we cannot rely solely on the training data for model selection either because we cannot estimate the generalization error on the very data that we use to train the model.

The common practice to address this problem is to split our data three ways, incorporating a *validation set* in addition to the training and test sets.

In practical applications, the picture gets muddier. While ideally we would only touch the test data once, to assess the very best model or to compare a small number of models to each other, real-world test data is seldom discarded after just one use. We can seldom afford a new test set for each round of experiments.

The result is a murky practice where the boundaries between validation and test data are worryingly ambiguous. Unless explicitly stated otherwise, in the experiments in this book we are really working with what should rightly be called training data and validation data, with no true test sets. Therefore, the accuracy reported in each experiment is really the validation accuracy and not a true test set accuracy. The good news is that we do not need too much data in the validation set. The uncertainty in our estimates can be shown to be of the order of $\mathcal{O}(n^{-\frac{1}{2}})$.

K-Fold Cross-Validation

When training data is scarce, we might not even be able to afford to hold out enough data to constitute a proper validation set. One popular solution to this problem is to employ *K-fold cross-validation*. Here, the original training data is split into K non-overlapping subsets. Then model training and validation are executed K times, each time training on $K-1$ subsets and validating on a different subset (the one not used for training in that round). Finally, the training and validation error rates are estimated by averaging over the results from the K experiments.

4.4.3 Underfitting or Overfitting?

When we compare the training and validation errors, we want to be mindful of two common situations: First, we want to watch out for cases when our training error and validation error are both substantial but there is a little gap between them. If the model is unable to reduce the training error, that could mean that our model is too simple (i.e., insufficiently expressive) to capture the pattern that we are trying to model. Moreover, since the *generalization gap* between our training and validation errors is small, we have reason to believe that we could get away with a more complex model. This phenomenon is known as underfitting.

On the other hand, as we discussed above, we want to watch out for the cases when our training error is significantly lower than our validation error, indicating severe overfitting. Note that overfitting is not always a bad thing. With deep learning especially, it is well known that the best predictive models often perform far better on training data than on holdout data. Ultimately, we usually care more about the validation error than about the gap between the training and validation errors.

Whether we overfit or underfit can depend both on the complexity of our model and the size of the available training datasets, two topics that we discuss below.

Model Complexity

To illustrate some classical intuition about overfitting and model complexity, we given an example using polynomials. Given training data consisting of a single feature x and a corresponding real-valued label y, we try to find the polynomial of degree d

$$\hat{y} = \sum_{i=0}^{d} x^i w_i \tag{4.4.1}$$

to estimate the labels y. This is just a linear regression problem where our features are given by the powers of x, the w_i given the model's weights, and the bias is given by w_0 since $x^0 = 1$ for all x. Since this is just a linear regression problem, we can use the squared error as our loss function.

A higher-order polynomial function is more complex than a lower order polynomial function, since the higher-order polynomial has more parameters and the model function's selection range is wider. Fixing the training dataset, higher-order polynomial functions should always achieve lower (at worst, equal) training error relative to lower degree polynomials. In fact, whenever the data points each have a distinct value of x, a polynomial function with degree equal to the number of data points can fit the training set perfectly. We visualize the relationship between polynomial degree and under- vs over-fitting in Fig. 4.4.1.

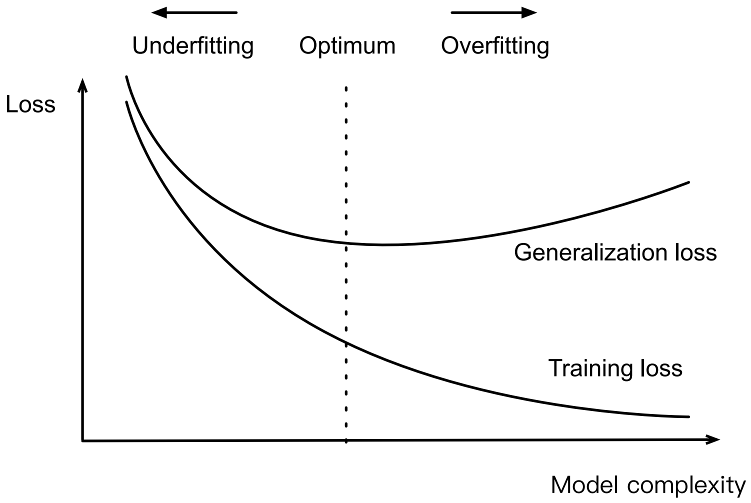


Fig. 4.4.1: Influence of Model Complexity on Underfitting and Overfitting

Dataset Size

The other big consideration to bear in mind is the dataset size. Fixing our model, the fewer samples we have in the training dataset, the more likely (and more severely) we are to encounter overfitting. As we increase the amount of training data, the generalization error typically decreases. Moreover, in general, more data never hurts. For a fixed task and data *distribution*, there is typically a relationship between model complexity and dataset size. Given more data, we might profitably attempt to fit a more complex model. Absent sufficient data, simpler models may be difficult to beat. For many tasks, deep learning only outperforms linear models when many thousands of training examples are available. In part, the current success of deep learning owes to the current abundance of massive datasets due to internet companies, cheap storage, connected devices, and the broad digitization of the economy.

4.4.4 Polynomial Regression

We can now explore these concepts interactively by fitting polynomials to data. To get started we will import our usual packages.

```
import d2l
from mxnet import gluon, np, npx
from mxnet.gluon import nn
npx.set_np()
```

Generating the Dataset

First we need data. Given x, we will use the following cubic polynomial to generate the labels on training and test data:

$$y = 5 + 1.2x - 3.4\frac{x^2}{2!} + 5.6\frac{x^3}{3!} + \epsilon \text{ where } \epsilon \sim \mathcal{N}(0, 0.1). \tag{4.4.2}$$

The noise term ϵ obeys a normal distribution with a mean of 0 and a standard deviation of 0.1. We will synthesize 100 samples each for the training set and test set.

```
maxdegree = 20  # Maximum degree of the polynomial
n_train, n_test = 100, 100  # Training and test dataset sizes
true_w = np.zeros(maxdegree)  # Allocate lots of empty space
true_w[0:4] = np.array([5, 1.2, -3.4, 5.6])

features = np.random.normal(size=(n_train + n_test, 1))
features = np.random.shuffle(features)
poly_features = np.power(features, np.arange(maxdegree).reshape(1, -1))
poly_features = poly_features / (
    npx.gamma(np.arange(maxdegree) + 1).reshape(1, -1))
labels = np.dot(poly_features, true_w)
labels += np.random.normal(scale=0.1, size=labels.shape)
```

For optimization, we typically want to avoid very large values of gradients, losses, etc. This is why the monomials stored in `poly_features` are rescaled from x^i to $\frac{1}{i!}x^i$. It allows us to avoid very large values for large exponents i. Factorials are implemented in Gluon using the Gamma function, where $n! = \Gamma(n + 1)$.

Take a look at the first 2 samples from the generated dataset. The value 1 is technically a feature, namely the constant feature corresponding to the bias.

```
features[:2], poly_features[:2], labels[:2]
```

```
(array([[-0.03716067],
        [-1.1468065 ]]),
 array([[ 1.00000000e+00, -3.71606685e-02,  6.90457586e-04,
```

(continues on next page)

(continued from previous page)

```
         -8.55262169e-06,  7.94552903e-08, -5.90522298e-10,
          3.65736781e-12, -1.94157468e-14,  9.01877669e-17,
         -3.72381952e-19,  1.38379622e-21, -4.67479880e-24,
          1.44765544e-26, -4.13814215e-29,  1.09840096e-31,
         -2.72115417e-34,  6.31999553e-37, -1.38150092e-39,
          2.85164237e-42, -5.60519386e-45],
        [ 1.00000000e+00, -1.14680648e+00,  6.57582462e-01,
         -2.51373291e-01,  7.20691308e-02, -1.65298693e-02,
          3.15942708e-03, -5.17607376e-04,  7.41994299e-05,
         -9.45470947e-06,  1.08427218e-06, -1.13040933e-07,
          1.08030065e-08, -9.52996793e-10,  7.80644993e-11,
         -5.96832479e-12,  4.27782159e-13, -2.88578399e-14,
          1.83857564e-15, -1.10973155e-16]]),
 array([ 5.1432443 , -0.06415092]))
```

Training and Testing Model

Let's first implement a function to evaluate the loss on a given data.

```
# Saved in the d2l package for later use
def evaluate_loss(net, data_iter, loss):
    """Evaluate the loss of a model on the given dataset"""
    metric = d2l.Accumulator(2)  # sum_loss, num_examples
    for X, y in data_iter:
        metric.add(loss(net(X), y).sum(), y.size)
    return metric[0] / metric[1]
```

Now define the training function.

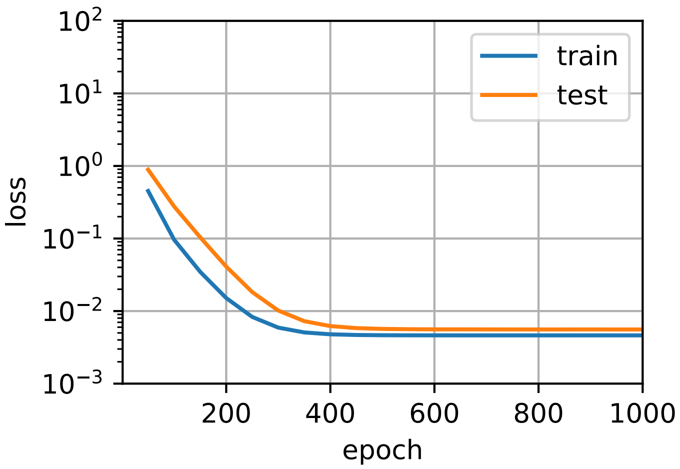
```
def train(train_features, test_features, train_labels, test_labels,
          num_epochs=1000):
    loss = gluon.loss.L2Loss()
    net = nn.Sequential()
    # Switch off the bias since we already catered for it in the polynomial
    # features
    net.add(nn.Dense(1, use_bias=False))
    net.initialize()
    batch_size = min(10, train_labels.shape[0])
    train_iter = d2l.load_array((train_features, train_labels), batch_size)
    test_iter = d2l.load_array((test_features, test_labels), batch_size,
                               is_train=False)
    trainer = gluon.Trainer(net.collect_params(), 'sgd',
                            {'learning_rate': 0.01})
    animator = d2l.Animator(xlabel='epoch', ylabel='loss', yscale='log',
                            xlim=[1, num_epochs], ylim=[1e-3, 1e2],
                            legend=['train', 'test'])
    for epoch in range(1, num_epochs+1):
        d2l.train_epoch_ch3(net, train_iter, loss, trainer)
        if epoch % 50 == 0:
            animator.add(epoch, (evaluate_loss(net, train_iter, loss),
                                 evaluate_loss(net, test_iter, loss)))
    print('weight:', net[0].weight.data().asnumpy())
```

Third-Order Polynomial Function Fitting (Normal)

We will begin by first using a third-order polynomial function with the same order as the data generation function. The results show that this model's training error rate when using the testing dataset is low. The trained model parameters are also close to the true values $w = [5, 1.2, -3.4, 5.6]$.

```
# Pick the first four dimensions, i.e., 1, x, x^2, x^3 from the polynomial
# features
train(poly_features[:n_train, 0:4], poly_features[n_train:, 0:4],
      labels[:n_train], labels[n_train:])
```

```
weight: [[ 5.015801   1.195607  -3.4175324  5.6197805]]
```

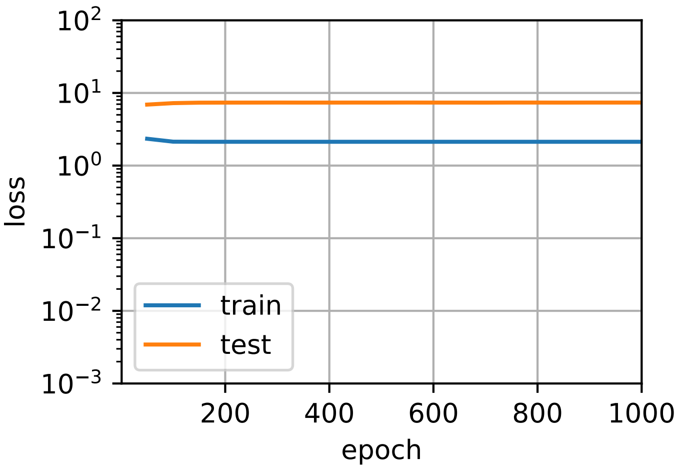


Linear Function Fitting (Underfitting)

Let's take another look at linear function fitting. After the decline in the early epoch, it becomes difficult to further decrease this model's training error rate. After the last epoch iteration has been completed, the training error rate is still high. When used to fit non-linear patterns (like the third-order polynomial function here) linear models are liable to underfit.

```
# Pick the first four dimensions, i.e., 1, x from the polynomial features
train(poly_features[:n_train, 0:3], poly_features[n_train:, 0:3],
      labels[:n_train], labels[n_train:])
```

```
weight: [[ 5.264786   4.0222955 -3.9895918]]
```



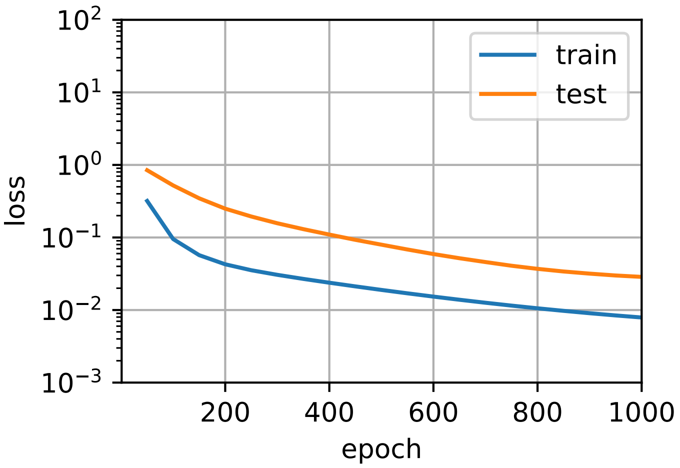
Insufficient Training (Overfitting)

Now let's try to train the model using a polynomial of too high degree. Here, there is insufficient data to learn that the higher-degree coefficients should have values close to zero. As a result, our overly-complex model is far too susceptible to being influenced by noise in the training data. Of course, our training error will now be low (even lower than if we had the right model!) but our test error will be high.

Try out different model complexities (n_degree) and training set sizes (n_subset) to gain some intuition of what is happening.

```
n_subset = 100  # Subset of data to train on
n_degree = 20  # Degree of polynomials
train(poly_features[1:n_subset, 0:n_degree],
      poly_features[n_train:, 0:n_degree], labels[1:n_subset],
      labels[n_train:])
```

```
weight: [[ 4.9480042   1.3321037  -3.209672    5.043265   -0.42166913  1.3474398
   0.07512859  0.19178352 -0.01918835  0.01772064 -0.0509516  -0.02382804
  -0.01497367 -0.04940014  0.06389725 -0.04761839 -0.04380196 -0.05188226
   0.05655775  0.01104914]]
```



In later chapters, we will continue to discuss overfitting problems and methods for dealing with them, such as weight decay and dropout.

Summary

- Since the generalization error rate cannot be estimated based on the training error rate, simply minimizing the training error rate will not necessarily mean a reduction in the generalization error rate. Machine learning models need to be careful to safeguard against overfitting such as to minimize the generalization error.
- A validation set can be used for model selection (provided that it is not used too liberally).
- Underfitting means that the model is not able to reduce the training error rate while overfitting is a result of the model training error rate being much lower than the testing dataset rate.
- We should choose an appropriately complex model and avoid using insufficient training samples.

Exercises

1. Can you solve the polynomial regression problem exactly? Hint: use linear algebra.
2. Model selection for polynomials
   - Plot the training error vs. model complexity (degree of the polynomial). What do you observe?
   - Plot the test error in this case.
   - Generate the same graph as a function of the amount of data?
3. What happens if you drop the normalization of the polynomial features x^i by $1/i!$. Can you fix this in some other way?
4. What degree of polynomial do you need to reduce the training error to 0?
5. Can you ever expect to see 0 generalization error?

4.5 Weight Decay

Now that we have characterized the problem of overfitting and motivated the need for capacity control, we can begin discussing some of the popular techniques used to these ends in practice. Recall that we can always mitigate overfitting by going out and collecting more training data, that can be costly and time consuming, typically making it impossible in the short run. For now, let's assume that we have already obtained as much high-quality data as our resources permit and focus on techniques aimed at limiting the capacity of the function classes under consideration.

In our toy example, we saw that we could control the complexity of a polynomial by adjusting its degree. However, most of machine learning does not consist of polynomial curve fitting. And moreover, even when we focus on polynomial regression, when we deal with high-dimensional data, manipulating model capacity by tweaking the degree d is problematic. To see why, note that for multivariate data we must generalize the concept of polynomials to include *monomials*, which are simply products of powers of variables. For example, $x_1^2 x_2$, and $x_3 x_5^2$ are both monomials of degree 3. The number of such terms with a given degree d blows up as a function of the degree d.

Concretely, for vectors of dimensionality D, the number of monomials of a given degree d is $\binom{D-1+d}{D-1}$. Hence, a small change in degree, even from say 1 to 2 or 2 to 3 would entail a massive blowup in the complexity of our model. Thus, tweaking the degree is too blunt a hammer. Instead, we need a more fine-grained tool for adjusting function complexity.

4.5.1 Squared Norm Regularization

Weight decay (commonly called *L2* regularization), might be the most widely-used technique for regularizing parametric machine learning models. The basic intuition behind weight decay is the notion that among all functions f, the function $f = 0$ is the simplest. Intuitively, we can then measure functions by their proximity to zero. But how precisely should we measure the distance between a function and zero? There is no single right answer. In fact, entire branches of mathematics, e.g., in functional analysis and the theory of Banach spaces are devoted to answering this issue.

For our present purposes, a very simple interpretation will suffice: We will consider a linear function $f(\mathbf{x}) = \mathbf{w}^\top \mathbf{x}$ to be simple if its weight vector is small. We can measure this via $||mathbfw||^2$. One way of keeping the weight vector small is to add its norm as a penalty term to the problem of minimizing the loss. Thus we replace our original objective, *minimize the prediction error on the training labels*, with new objective, *minimize the sum of the prediction error and the penalty term*. Now, if the weight vector becomes too large, our learning algorithm will find more profit in minimizing the norm $||\mathbf{w}||^2$ versus minimizing the training error. That is exactly what we want. To illustrate things in code, let's revive our previous example from Section 3.1 for linear regression. There, our loss was given by

$$l(\mathbf{w}, b) = \frac{1}{n}\sum_{i=1}^n \frac{1}{2}\left(\mathbf{w}^\top \mathbf{x}^{(i)} + b - y^{(i)}\right)^2. \tag{4.5.1}$$

Recall that $\mathbf{x}^{(i)}$ are the observations, $y^{(i)}$ are labels, and $(\mathbf{w}, b)$ are the weight and bias parameters respectively. To arrive at a new loss function that penalizes the size of the weight vector, we need to add $||mathbfw||^2$, but how much should we add? To address this, we need to add a new hyperparameter, that we will call the *regularization constant* and denote by λ:

$$l(\mathbf{w}, b) + \frac{\lambda}{2}\|\mathbf{w}\|^2. \tag{4.5.2}$$

This non-negative parameter $\lambda \geq 0$ governs the amount of regularization. For $\lambda = 0$, we recover our original loss function, whereas for $\lambda > 0$ we ensure that $\mathbf{w}$ cannot grow too large. The astute reader might wonder why we are squaring the norm of the weight vector. We do this for two reasons. First, we do it for computational convenience. By squaring the L2 norm, we remove the square root, leaving the sum of squares of each component of the weight vector. This is convenient because it is easy to compute derivatives of a sum of terms (the sum of derivatives equals the derivative of the sum).

Moreover, you might ask, why the L2 norm in the first place and not the L1 norm, or some other distance function. In fact, several other choices are valid and are popular throughout statistics.

While L2-regularized linear models constitute the classic *ridge regression* algorithm L1-regularized linear regression is a similarly fundamental model in statistics popularly known as *lasso regression*.

One mathematical reason for working with the L2 norm and not some other norm, is that it penalizes large components of the weight vector much more than it penalizes small ones. This encourages our learning algorithm to discover models which distribute their weight across a larger number of features, which might make them more robust in practice since they do not depend precariously on a single feature. The stochastic gradient descent updates for L2-regularized regression are as follows:

$$w \leftarrow \left(1 - \frac{\eta\lambda}{|\mathcal{B}|}\right) \mathbf{w} - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \mathbf{x}^{(i)} \left(\mathbf{w}^\top \mathbf{x}^{(i)} + b - y^{(i)}\right), \tag{4.5.3}$$

As before, we update $\mathbf{w}$ based on the amount by which our estimate differs from the observation. However, we also shrink the size of $\mathbf{w}$ towards 0. That is why the method is sometimes called "weight decay": because the penalty term literally causes our optimization algorithm to *decay* the magnitude of the weight at each step of training. This is more convenient than having to pick the number of parameters as we did for polynomials. In particular, we now have a continuous mechanism for adjusting the complexity of f. Small values of λ correspond to unconstrained $\mathbf{w}$, whereas large values of λ constrain $\mathbf{w}$ considerably. Since we do not want to have large bias terms either, we often add b^2 as a penalty, too.

4.5.2 High-Dimensional Linear Regression

For high-dimensional regression it is difficult to pick the 'right' dimensions to omit. Weight-decay regularization is a much more convenient alternative. We will illustrate this below. First, we will generate some synthetic data as before

$$y = 0.05 + \sum_{i=1}^{d} 0.01 x_i + \epsilon \text{ where } \epsilon \sim \mathcal{N}(0, 0.01). \tag{4.5.4}$$

representing our label as a linear function of our inputs, corrupted by Gaussian noise with zero mean and variance 0.01. To observe the effects of overfitting more easily, we can make our problem high-dimensional, setting the data dimension to $d = 200$ and working with a relatively small number of training examples—here we will set the sample size to 20:

```
%matplotlib inline
import d2l
from mxnet import autograd, gluon, init, np, npx
from mxnet.gluon import nn
npx.set_np()

n_train, n_test, num_inputs, batch_size = 20, 100, 200, 1
true_w, true_b = np.ones((num_inputs, 1)) * 0.01, 0.05
train_data = d2l.synthetic_data(true_w, true_b, n_train)
train_iter = d2l.load_array(train_data, batch_size)
test_data = d2l.synthetic_data(true_w, true_b, n_test)
test_iter = d2l.load_array(test_data, batch_size, is_train=False)
```

4.5.3 Implementation from Scratch

Next, we will show how to implement weight decay from scratch. All we have to do here is to add the squared ℓ_2 penalty as an additional loss term added to the original target function. The squared norm penalty derives its name from the fact that we are adding the second power $\sum_i w_i^2$. The ℓ_2 is just one among an infinite class of norms call p-norms, many of which you might encounter in the future. In general, for some number p, the ℓ_p norm is defined as

$$\|\mathbf{w}\|_p^p := \sum_{i=1}^d |w_i|^p. \tag{4.5.5}$$

Initializing Model Parameters

First, we will define a function to randomly initialize our model parameters and run `attach_grad` on each to allocate memory for the gradients we will calculate.

```
def init_params():
    w = np.random.normal(scale=1, size=(num_inputs, 1))
    b = np.zeros(1)
    w.attach_grad()
    b.attach_grad()
    return [w, b]
```

Defining ℓ_2 Norm Penalty

Perhaps the most convenient way to implement this penalty is to square all terms in place and sum them up. We divide by 2 by convention (when we take the derivative of a quadratic function, the 2 and $1/2$ cancel out, ensuring that the expression for the update looks nice and simple).

```
def l2_penalty(w):
    return (w**2).sum() / 2
```

Defining the Train and Test Functions

The following code defines how to train and test the model separately on the training dataset and the test dataset. Unlike the previous sections, here, the ℓ_2 norm penalty term is added when calculating the final loss function. The linear network and the squared loss have not changed since the previous chapter, so we will just import them via `d2l.linreg` and `d2l.squared_loss` to reduce clutter.

```
def train(lambd):
    w, b = init_params()
    net, loss = lambda X: d2l.linreg(X, w, b), d2l.squared_loss
    num_epochs, lr = 100, 0.003
    animator = d2l.Animator(xlabel='epochs', ylabel='loss', yscale='log',
                            xlim=[1, num_epochs], legend=['train', 'test'])
    for epoch in range(1, num_epochs + 1):
        for X, y in train_iter:
```

(continues on next page)

(continued from previous page)

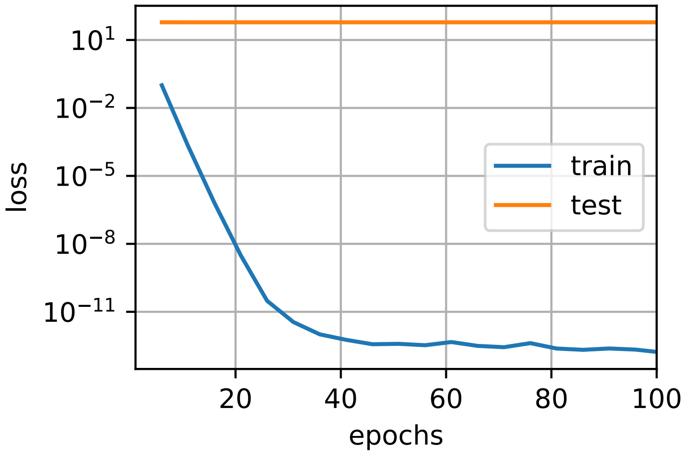
```
            with autograd.record():
                # The L2 norm penalty term has been added
                l = loss(net(X), y) + lambd * l2_penalty(w)
            l.backward()
            d2l.sgd([w, b], lr, batch_size)
        if epoch % 5 == 0:
            animator.add(epoch+1, (d2l.evaluate_loss(net, train_iter, loss),
                                   d2l.evaluate_loss(net, test_iter, loss)))
    print('l1 norm of w:', np.abs(w).sum())
```

Training without Regularization

Next, let's train and test the high-dimensional linear regression model. When `lambd = 0` we do not use weight decay. As a result, while the training error decreases, the test error does not. This is a perfect example of overfitting.

```
train(lambd=0)
```

```
l1 norm of w: 152.89598
```

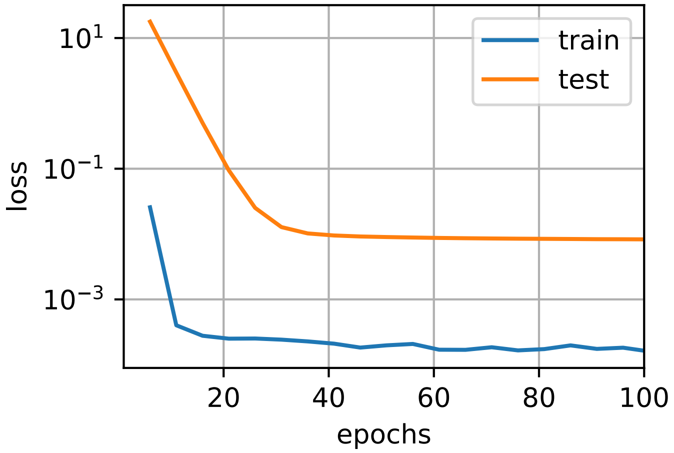


Using Weight Decay

The example below shows that even though the training error increased, the error on the test set decreased. This is precisely the improvement that we expect from using weight decay. While not perfect, overfitting has been mitigated to some extent. In addition, the ℓ_2 norm of the weight $\mathbf{w}$ is smaller than without using weight decay.

```
train(lambd=3)
```

```
l1 norm of w: 0.30536327
```



4.5.4 Concise Implementation

Because weight decay is ubiquitous in neural network optimization, Gluon makes it especially convenient, integrating weight decay into the optimization algorithm itself for easy use in combination with any loss function. Moreover, this integration serves a computational benefit, allowing implementation tricks to add weight decay to the algorithm, without any additional computational overhead. Since the weight decay portion of the update depends only on the current value of each parameter, and the optimizer must to touch each parameter once anyway.

In the following code, we specify the weight decay hyperparameter directly through the `wd` parameter when instantiating our `Trainer`. By default, Gluon decays both weights and biases simultaneously. Note that we can have *different* optimizers for different sets of parameters. For instance, we can have one `Trainer` with weight decay for the weights $\mathbf{w}$ and another without weight decay to take care of the bias b.

```
def train_gluon(wd):
    net = nn.Sequential()
    net.add(nn.Dense(1))
    net.initialize(init.Normal(sigma=1))
    loss = gluon.loss.L2Loss()
    num_epochs, lr = 100, 0.003
    # The weight parameter has been decayed. Weight names generally end with
    # "weight"
    trainer_w = gluon.Trainer(net.collect_params('.*weight'), 'sgd',
                              {'learning_rate': lr, 'wd': wd})
    # The bias parameter has not decayed. Bias names generally end with "bias"
    trainer_b = gluon.Trainer(net.collect_params('.*bias'), 'sgd',
                              {'learning_rate': lr})
    animator = d2l.Animator(xlabel='epochs', ylabel='loss', yscale='log',
                            xlim=[1, num_epochs], legend=['train', 'test'])
    for epoch in range(1, num_epochs+1):
        for X, y in train_iter:
            with autograd.record():
                l = loss(net(X), y)
            l.backward()
            # Call the step function on each of the two Trainer instances to
            # update the weight and bias separately
            trainer_w.step(batch_size)
```

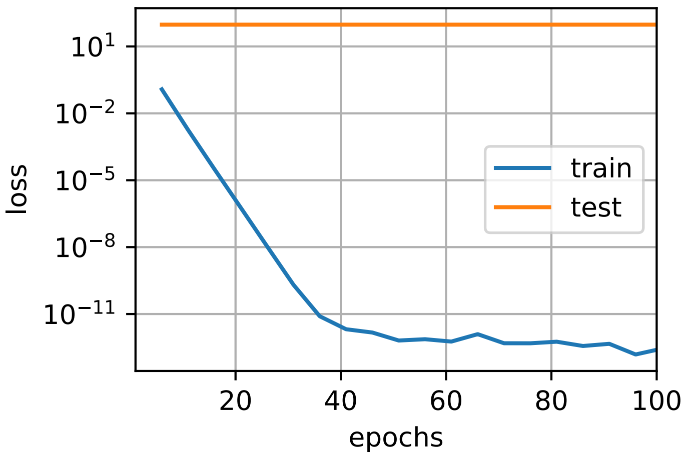
(continues on next page)

```
            trainer_b.step(batch_size)
        if epoch % 5 == 0:
            animator.add(epoch+1, (d2l.evaluate_loss(net, train_iter, loss),
                                   d2l.evaluate_loss(net, test_iter, loss)))
    print('L1 norm of w:', np.abs(net[0].weight.data()).sum())
```

The plots look just the same as when we implemented weight decay from scratch but they run a bit faster and are easier to implement, a benefit that will become more pronounced for large problems.

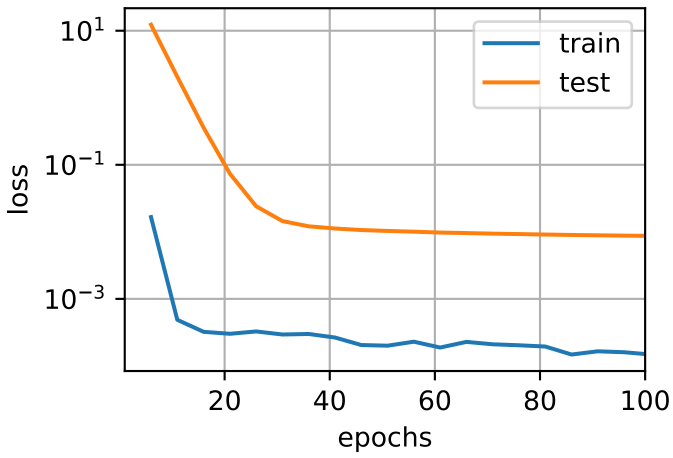
```
train_gluon(0)
```

```
L1 norm of w: 163.57817
```



```
train_gluon(3)
```

```
L1 norm of w: 0.3083761
```



So far, we only touched upon one notion of what constitutes a simple *linear* function. For nonlinear functions, what constitutes *simplicity* can be a far more complex question. For instance, there exist Reproducing Kernel Hilbert Spaces (RKHS)[73] which allow one to use many of the tools introduced for linear functions in a nonlinear context. Unfortunately, RKHS-based algorithms do not always scale well to massive amounts of data. For the purposes of this book, we limit ourselves to simply summing over the weights for different layers, e.g., via $\sum_l \|\mathbf{w}_l\|^2$, which is equivalent to weight decay applied to all layers.

Summary

- Regularization is a common method for dealing with overfitting. It adds a penalty term to the loss function on the training set to reduce the complexity of the learned model.
- One particular choice for keeping the model simple is weight decay using an ℓ_2 penalty. This leads to weight decay in the update steps of the learning algorithm.
- Gluon provides automatic weight decay functionality in the optimizer by setting the hyperparameter `wd`.
- You can have different optimizers within the same training loop, e.g., for different sets of parameters.

Exercises

1. Experiment with the value of λ in the estimation problem in this page. Plot training and test accuracy as a function of λ. What do you observe?
2. Use a validation set to find the optimal value of λ. Is it really the optimal value? Does this matter?
3. What would the update equations look like if instead of $\|\mathbf{w}\|^2$ we used $\sum_i |w_i|$ as our penalty of choice (this is called ℓ_1 regularization).
4. We know that $\|\mathbf{w}\|^2 = \mathbf{w}^\top \mathbf{w}$. Can you find a similar equation for matrices (mathematicians call this the Frobenius norm[74])?
5. Review the relationship between training error and generalization error. In addition to weight decay, increased training, and the use of a model of suitable complexity, what other ways can you think of to deal with overfitting?
6. In Bayesian statistics we use the product of prior and likelihood to arrive at a posterior via $P(w \mid x) \propto P(x \mid w)P(w)$. How can you identify $P(w)$ with regularization?

[73] https://en.wikipedia.org/wiki/Reproducing_kernel_Hilbert_space

[74] https://en.wikipedia.org/wiki/Matrix_norm#Frobenius_norm

4.6 Dropout

Just now, we introduced the classical approach of regularizing statistical models by penalizing the ℓ_2 norm of the weights. In probabilistic terms, we could justify this technique by arguing that we have assumed a prior belief that weights take values from a Gaussian distribution with mean 0. More intuitively, we might argue that we encouraged the model to spread out its weights among many features and rather than depending too much on a small number of potentially spurious associations.

4.6.1 Overfitting Revisited

Given many more features than examples, linear models can overfit. But when there are many more examples than features, we can generally count on linear models not to overfit. Unfortunately, the reliability with which linear models generalize comes at a cost: Linear models can't take into account interactions among features. For every feature, a linear model must assign either a positive or a negative weight. They lack the flexibility to account for context.

In more formal text, you'll see this fundamental tension between generalizability and flexibility discussed as the *bias-variance tradeoff*. Linear models have high bias (they can only represent a small class of functions), but low variance (they give similar results across different random samples of the data).

Deep neural networks take us to the opposite end of the bias-variance spectrum. Neural networks are so flexible because they aren't confined to looking at each feature individually. Instead, they can learn interactions among groups of features. For example, they might infer that "Nigeria" and "Western Union" appearing together in an email indicates spam but that "Nigeria" without "Western Union" does not.

Even when we only have a small number of features, deep neural networks are capable of overfitting. In 2017, a group of researchers presented a now well-known demonstration of the incredible flexibility of neural networks. They presented a neural network with randomly-labeled images (there was no true pattern linking the inputs to the outputs) and found that the neural network, optimized by SGD, could label every image in the training set perfectly.

Consider what this means. If the labels are assigned uniformly at random and there are 10 classes, then no classifier can get better than 10% accuracy on holdout data. Yet even in these situations, when there is no true pattern to be learned, neural networks can perfectly fit the training labels.

4.6.2 Robustness through Perturbations

Let's think briefly about what we expect from a good statistical model. We want it to do well on unseen test data. One way we can accomplish this is by asking what constitutes a "simple" model? Simplicity can come in the form of a small number of dimensions, which is what we did when discussing fitting a model with monomial basis functions. Simplicity can also come in the form of a small norm for the basis functions. This led us to weight decay (ℓ_2 regularization). Yet a third notion of simplicity that we can impose is that the function should be robust under small changes in the input. For instance, when we classify images, we would expect that adding some random noise to the pixels should be mostly harmless.

In 1995, Christopher Bishop formalized a form of this idea when he proved that training with input noise is equivalent to Tikhonov regularization (Bishop, 1995). In other words, he drew a clear mathematical connection between the requirement that a function be smooth (and thus simple), as we discussed in the section on weight decay, with and the requirement that it be resilient to perturbations in the input.

Then in 2014, Srivastava et al. (Srivastava et al., 2014) developed a clever idea for how to apply Bishop's idea to the *internal* layers of the network, too. Namely they proposed to inject noise into each layer of the network before calculating the subsequent layer during training. They realized that when training deep network with many layers, enforcing smoothness just on the input-output mapping misses out on what is happening internally in the network. Their proposed idea is called *dropout*, and it is now a standard technique that is widely used for training neural networks. Throughout training, on each iteration, dropout regularization consists simply of zeroing out some fraction (typically 50%) of the nodes in each layer before calculating the subsequent layer.

The key challenge then is how to inject this noise without introducing undue statistical *bias*. In other words, we want to perturb the inputs to each layer during training in such a way that the expected value of the layer is equal to the value it would have taken had we not introduced any noise at all.

In Bishop's case, when we are adding Gaussian noise to a linear model, this is simple: At each training iteration, just add noise sampled from a distribution with mean zero $\epsilon \sim \mathcal{N}(0, \sigma^2)$ to the input $\mathbf{x}$, yielding a perturbed point $\mathbf{x}' = \mathbf{x} + \epsilon$. In expectation, $E[\mathbf{x}'] = \mathbf{x}$.

In the case of dropout regularization, one can debias each layer by normalizing by the fraction of nodes that were not dropped out. In other words, dropout with drop probability p is applied as follows:

$$h' = \begin{cases} 0 & \text{with probability } p \\ \frac{h}{1-p} & \text{otherwise} \end{cases} \tag{4.6.1}$$

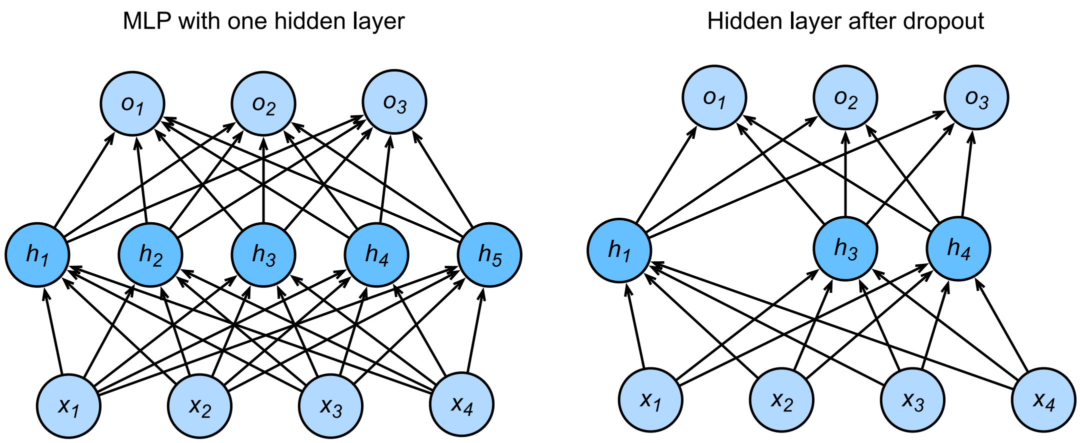
By design, the expectation remains unchanged, i.e., $E[h'] = h$. Intermediate activations h are replaced by a random variable h' with matching expectation. The name "dropout" arises from the notion that some neurons "drop out" of the computation for the purpose of computing the final result. During training, we replace intermediate activations with random variables.

4.6.3 Dropout in Practice

Recall the multilayer perceptron (Section 4.1) with a hidden layer and 5 hidden units. Its architecture is given by

$$\begin{aligned} h &= \sigma(W_1 x + b_1), \\ o &= W_2 h + b_2, \\ \hat{y} &= \text{softmax}(o). \end{aligned} \tag{4.6.2}$$

When we apply dropout to the hidden layer, we are essentially removing each hidden unit with probability p, (i.e., setting their output to 0). We can view the result as a network containing only a subset of the original neurons. In Fig. 4.6.1, h_2 and h_5 are removed. Consequently, the calculation of y no longer depends on h_2 and h_5 and their respective gradient also vanishes when performing backprop. In this way, the calculation of the output layer cannot be overly dependent on any one element of $h_1, \ldots, h_5$. Intuitively, deep learning researchers often explain the intuition thusly: we do not want the network's output to depend too precariously on the exact activation pathway through the network. The original authors of the dropout technique described their intuition as an effort to prevent the *co-adaptation* of feature detectors.


MLP with one hidden layer

Hidden layer after dropout

Fig. 4.6.1: MLP before and after dropout

At test time, we typically do not use dropout. However, we note that there are some exceptions: some researchers use dropout at test time as a heuristic approach for estimating the *confidence* of neural network predictions: if the predictions agree across many different dropout masks, then we might say that the network is more confident. For now we will put off the advanced topic of uncertainty estimation for subsequent chapters and volumes.

4.6.4 Implementation from Scratch

To implement the dropout function for a single layer, we must draw as many samples from a Bernoulli (binary) random variable as our layer has dimensions, where the random variable takes value 1 (keep) with probability $1 - p$ and 0 (drop) with probability p. One easy way to implement this is to first draw samples from the uniform distribution $U[0, 1]$. then we can keep those nodes for which the corresponding sample is greater than p, dropping the rest.

In the following code, we implement a `dropout` function that drops out the elements in the `ndarray` input `X` with probability `drop_prob`, rescaling the remainder as described above (dividing the survivors by `1.0-drop_prob`).

```
import d2l
from mxnet import autograd, gluon, init, np, npx
from mxnet.gluon import nn
npx.set_np()

def dropout(X, drop_prob):
    assert 0 <= drop_prob <= 1
    # In this case, all elements are dropped out
    if drop_prob == 1:
        return np.zeros_like(X)
    mask = np.random.uniform(0, 1, X.shape) > drop_prob
    return mask.astype(np.float32) * X / (1.0-drop_prob)
```

We can test out the `dropout` function on a few examples. In the following lines of code, we pass our input `X` through the dropout operation, with probabilities 0, 0.5, and 1, respectively.

```
X = np.arange(16).reshape(2, 8)
print(dropout(X, 0))
```

(continues on next page)

(continued from previous page)

```
print(dropout(X, 0.5))
print(dropout(X, 1))
```

```
[[ 0.  1.  2.  3.  4.  5.  6.  7.]
 [ 8.  9. 10. 11. 12. 13. 14. 15.]]
[[ 0.  0.  0.  0.  8. 10. 12.  0.]
 [16.  0. 20. 22.  0.  0.  0. 30.]]
[[0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0.]]
```

Defining Model Parameters

Again, we can use the Fashion-MNIST dataset, introduced in Section 3.6. We will define a multilayer perceptron with two hidden layers. The two hidden layers both have 256 outputs.

```
num_inputs, num_outputs, num_hiddens1, num_hiddens2 = 784, 10, 256, 256

W1 = np.random.normal(scale=0.01, size=(num_inputs, num_hiddens1))
b1 = np.zeros(num_hiddens1)
W2 = np.random.normal(scale=0.01, size=(num_hiddens1, num_hiddens2))
b2 = np.zeros(num_hiddens2)
W3 = np.random.normal(scale=0.01, size=(num_hiddens2, num_outputs))
b3 = np.zeros(num_outputs)

params = [W1, b1, W2, b2, W3, b3]
for param in params:
    param.attach_grad()
```

Defining the Model

The model defined below concatenates the fully-connected layer and the activation function ReLU, using dropout for the output of each activation function. We can set the dropout probability of each layer separately. It is generally recommended to set a lower dropout probability closer to the input layer. Below we set it to 0.2 and 0.5 for the first and second hidden layer respectively. By using the `is_training` function described in Section 2.5, we can ensure that dropout is only active during training.

```
drop_prob1, drop_prob2 = 0.2, 0.5

def net(X):
    X = X.reshape(-1, num_inputs)
    H1 = npx.relu(np.dot(X, W1) + b1)
    # Use dropout only when training the model
    if autograd.is_training():
        # Add a dropout layer after the first fully connected layer
        H1 = dropout(H1, drop_prob1)
    H2 = npx.relu(np.dot(H1, W2) + b2)
    if autograd.is_training():
        # Add a dropout layer after the second fully connected layer
```

(continues on next page)

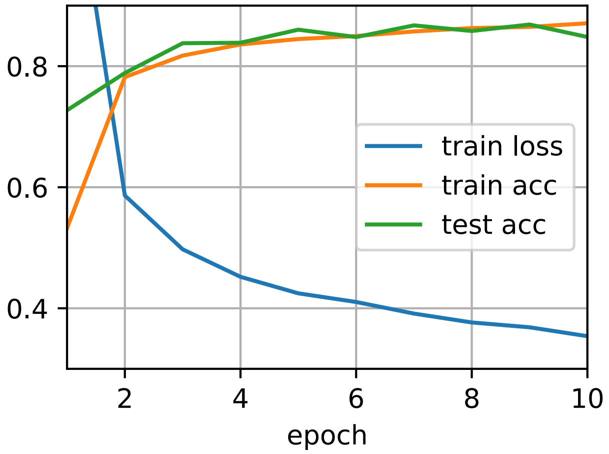
(continued from previous page)

```
        H2 = dropout(H2, drop_prob2)
    return np.dot(H2, W3) + b3
```

Training and Testing

This is similar to the training and testing of multilayer perceptrons described previously.

```
num_epochs, lr, batch_size = 10, 0.5, 256
loss = gluon.loss.SoftmaxCrossEntropyLoss()
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)
d2l.train_ch3(net, train_iter, test_iter, loss, num_epochs,
              lambda batch_size: d2l.sgd(params, lr, batch_size))
```



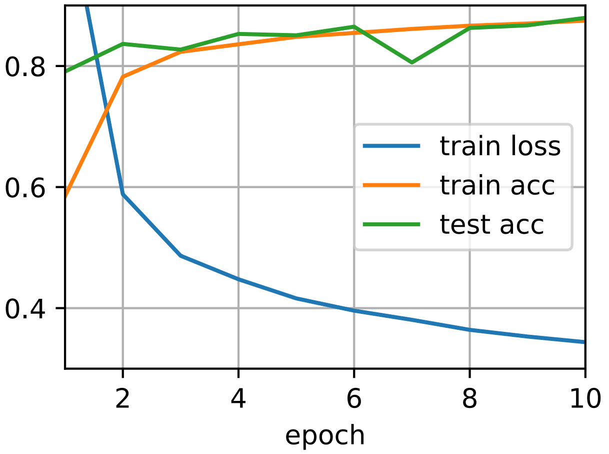
4.6.5 Concise Implementation

Using Gluon, all we need to do is add a Dropout layer (also in the nn package) after each fully-connected layer, passing in the dropout probability as the only argument to its constructor. During training, the Dropout layer will randomly drop out outputs of the previous layer (or equivalently, the inputs to the subsequent layer) according to the specified dropout probability. When MXNet is not in training mode, the Dropout layer simply passes the data through during testing.

```
net = nn.Sequential()
net.add(nn.Dense(256, activation="relu"),
        # Add a dropout layer after the first fully connected layer
        nn.Dropout(drop_prob1),
        nn.Dense(256, activation="relu"),
        # Add a dropout layer after the second fully connected layer
        nn.Dropout(drop_prob2),
        nn.Dense(10))
net.initialize(init.Normal(sigma=0.01))
```

Next, we train and test the model.

```
trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': lr})
d2l.train_ch3(net, train_iter, test_iter, loss, num_epochs, trainer)
```



Summary

- Beyond controlling the number of dimensions and the size of the weight vector, dropout is yet another tool to avoid overfitting. Often all three are used jointly.
- Dropout replaces an activation h with a random variable h' with expected value h and with variance given by the dropout probability p.
- Dropout is only used during training.

Exercises

1. Try out what happens if you change the dropout probabilities for layers 1 and 2. In particular, what happens if you switch the ones for both layers?
2. Increase the number of epochs and compare the results obtained when using dropout with those when not using it.
3. Compute the variance of the the activation random variables after applying dropout.
4. Why should you typically not using dropout?
5. If changes are made to the model to make it more complex, such as adding hidden layer units, will the effect of using dropout to cope with overfitting be more obvious?
6. Using the model in this section as an example, compare the effects of using dropout and weight decay. What if dropout and weight decay are used at the same time?
7. What happens if we apply dropout to the individual weights of the weight matrix rather than the activations?
8. Replace the dropout activation with a random variable that takes on values of $[0, \gamma/2, \gamma]$. Can you design something that works better than the binary dropout function? Why might you want to use it? Why not?

4.7 Forward Propagation, Backward Propagation, and Computational Graphs

In the previous sections, we used minibatch stochastic gradient descent to train our models. When we implemented the algorithm, we only worried about the calculations involved in *forward propagation* through the model. In other words, we implemented the calculations required for the model to generate output corresponding to come given input, but when it came time to calculate the gradients of each of our parameters, we invoked the `backward` function, relying on the `autograd` module to figure out what to do.

The automatic calculation of gradients profoundly simplifies the implementation of deep learning algorithms. Before automatic differentiation, even small changes to complicated models would require recalculating lots of derivatives by hand. Even academic papers would too often have to allocate lots of page real estate to deriving update rules.

While we plan to continue relying on `autograd`, and we have already come a long way without every discussing how these gradients are calculated efficiently under the hood, it is important that you know how updates are actually calculated if you want to go beyond a shallow understanding of deep learning.

In this section, we will peel back the curtain on some of the details of backward propagation (more commonly called *backpropagation* or *backprop*). To convey some insight for both the techniques and how they are implemented, we will rely on both mathematics and computational graphs to describe the mechanics behind neural network computations. To start, we will focus our exposition on a simple multilayer perceptron with a single hidden layer and ℓ_2 norm regularization.

4.7.1 Forward Propagation

Forward propagation refers to the calculation and storage of intermediate variables (including outputs) for the neural network within the models in the order from input layer to output layer. In the following, we work in detail through the example of a deep network with one hidden layer step by step. This is a bit tedious but it will serve us well when discussing what really goes on when we call `backward`.

For the sake of simplicity, let's assume that the input example is $\mathbf{x} \in \mathbb{R}^d$ and there is no bias term. Here the intermediate variable is:

$$\mathbf{z} = \mathbf{W}^{(1)}\mathbf{x}, \tag{4.7.1}$$

where $\mathbf{W}^{(1)} \in \mathbb{R}^{h \times d}$ is the weight parameter of the hidden layer. After entering the intermediate variable $\mathbf{z} \in \mathbb{R}^h$ into the activation function ϕ operated by the basic elements, we will obtain a hidden layer variable with the vector length of h,

$$\mathbf{h} = \phi(\mathbf{z}). \tag{4.7.2}$$

The hidden variable $\mathbf{h}$ is also an intermediate variable. Assuming the parameters of the output layer only possess a weight of $\mathbf{W}^{(2)} \in \mathbb{R}^{q\times h}$, we can obtain an output layer variable with a vector length of q:

$$\mathbf{o} = \mathbf{W}^{(2)}\mathbf{h}. \tag{4.7.3}$$

Assuming the loss function is l and the example label is y, we can then calculate the loss term for a single data example,

$$L = l(\mathbf{o}, y). \tag{4.7.4}$$

According to the definition of ℓ_2 norm regularization, given the hyperparameter λ, the regularization term is

$$s = \frac{\lambda}{2}\left(\|\mathbf{W}^{(1)}\|_F^2 + \|\mathbf{W}^{(2)}\|_F^2\right), \tag{4.7.5}$$

where the Frobenius norm of the matrix is equivalent to the calculation of the L_2 norm after flattening the matrix to a vector. Finally, the model's regularized loss on a given data example is

$$J = L + s. \tag{4.7.6}$$

We refer to J as the objective function of a given data example and refer to it as the *objective function* in the following discussion.

4.7.2 Computational Graph of Forward Propagation

Plotting computational graphs helps us visualize the dependencies of operators and variables within the calculation. Fig. 4.7.1 contains the graph associated with the simple network described above. The lower-left corner signifies the input and the upper right corner the output. Notice that the direction of the arrows (which illustrate data flow) are primarily rightward and upward.

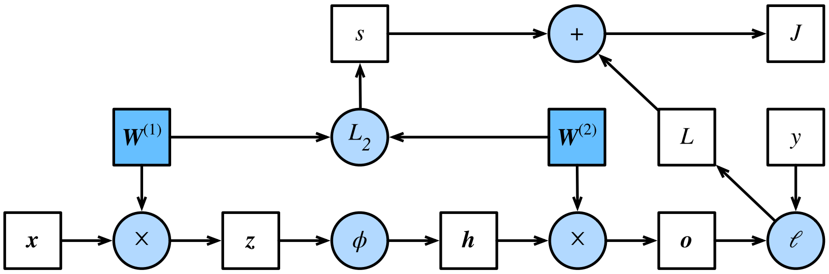


Fig. 4.7.1: Computational Graph

4.7.3 Backpropagation

Backpropagation refers to the method of calculating the gradient of neural network parameters. In general, back propagation calculates and stores the intermediate variables of an objective function related to each layer of the neural network and the gradient of the parameters in the order of the output layer to the input layer according to the 'chain rule' in calculus. Assume that we have functions $\mathsf{Y} = f(\mathsf{X})$ and $\mathsf{Z} = g(\mathsf{Y}) = g \circ f(\mathsf{X})$, in which the input and the output $\mathsf{X}, \mathsf{Y}, \mathsf{Z}$ are tensors of arbitrary shapes. By using the chain rule, we can compute the derivative of Z wrt. X via

$$\frac{\partial \mathsf{Z}}{\partial \mathsf{X}} = \text{prod}\left(\frac{\partial \mathsf{Z}}{\partial \mathsf{Y}}, \frac{\partial \mathsf{Y}}{\partial \mathsf{X}}\right). \tag{4.7.7}$$

Here we use the prod operator to multiply its arguments after the necessary operations, such as transposition and swapping input positions have been carried out. For vectors, this is straightforward: it is simply matrix-matrix multiplication and for higher dimensional tensors we use the appropriate counterpart. The operator prod hides all the notation overhead.

The parameters of the simple network with one hidden layer are $\mathbf{W}^{(1)}$ and $\mathbf{W}^{(2)}$. The objective of backpropagation is to calculate the gradients $\partial J/\partial \mathbf{W}^{(1)}$ and $\partial J/\partial \mathbf{W}^{(2)}$. To accomplish this, we will apply the chain rule and calculate, in turn, the gradient of each intermediate variable and parameter. The order of calculations are reversed relative to those performed in forward propagation, since we need to start with the outcome of the compute graph and work our way towards the parameters. The first step is to calculate the gradients of the objective function $J = L + s$ with respect to the loss term L and the regularization term s.

$$\frac{\partial J}{\partial L} = 1 \text{ and } \frac{\partial J}{\partial s} = 1. \tag{4.7.8}$$

Next, we compute the gradient of the objective function with respect to variable of the output layer **o** according to the chain rule.

$$\frac{\partial J}{\partial \mathbf{o}} = \text{prod}\left(\frac{\partial J}{\partial L}, \frac{\partial L}{\partial \mathbf{o}}\right) = \frac{\partial L}{\partial \mathbf{o}} \in \mathbb{R}^q. \tag{4.7.9}$$

Next, we calculate the gradients of the regularization term with respect to both parameters.

$$\frac{\partial s}{\partial \mathbf{W}^{(1)}} = \lambda \mathbf{W}^{(1)} \text{ and } \frac{\partial s}{\partial \mathbf{W}^{(2)}} = \lambda \mathbf{W}^{(2)}. \tag{4.7.10}$$

Now we are able calculate the gradient $\partial J/\partial \mathbf{W}^{(2)} \in \mathbb{R}^{q \times h}$ of the model parameters closest to the output layer. Using the chain rule yields:

$$\frac{\partial J}{\partial \mathbf{W}^{(2)}} = \text{prod}\left(\frac{\partial J}{\partial \mathbf{o}}, \frac{\partial \mathbf{o}}{\partial \mathbf{W}^{(2)}}\right) + \text{prod}\left(\frac{\partial J}{\partial s}, \frac{\partial s}{\partial \mathbf{W}^{(2)}}\right) = \frac{\partial J}{\partial \mathbf{o}} \mathbf{h}^\top + \lambda \mathbf{W}^{(2)}. \tag{4.7.11}$$

To obtain the gradient with respect to $\mathbf{W}^{(1)}$ we need to continue backpropagation along the output layer to the hidden layer. The gradient with respect to the hidden layer's outputs $\partial J/\partial \mathbf{h} \in \mathbb{R}^h$ is given by

$$\frac{\partial J}{\partial \mathbf{h}} = \text{prod}\left(\frac{\partial J}{\partial \mathbf{o}}, \frac{\partial \mathbf{o}}{\partial \mathbf{h}}\right) = {\mathbf{W}^{(2)}}^\top \frac{\partial J}{\partial \mathbf{o}}. \tag{4.7.12}$$

Since the activation function ϕ applies elementwise, calculating the gradient $\partial J/\partial \mathbf{z} \in \mathbb{R}^h$ of the intermediate variable **z** requires that we use the elementwise multiplication operator, which we denote by $\odot$.

$$\frac{\partial J}{\partial \mathbf{z}} = \text{prod}\left(\frac{\partial J}{\partial \mathbf{h}}, \frac{\partial \mathbf{h}}{\partial \mathbf{z}}\right) = \frac{\partial J}{\partial \mathbf{h}} \odot \phi'\left(\mathbf{z}\right). \tag{4.7.13}$$

Finally, we can obtain the gradient $\partial J/\partial \mathbf{W}^{(1)} \in \mathbb{R}^{h \times d}$ of the model parameters closest to the input layer. According to the chain rule, we get

$$\frac{\partial J}{\partial \mathbf{W}^{(1)}} = \text{prod}\left(\frac{\partial J}{\partial \mathbf{z}}, \frac{\partial \mathbf{z}}{\partial \mathbf{W}^{(1)}}\right) + \text{prod}\left(\frac{\partial J}{\partial s}, \frac{\partial s}{\partial \mathbf{W}^{(1)}}\right) = \frac{\partial J}{\partial \mathbf{z}} \mathbf{x}^\top + \lambda \mathbf{W}^{(1)}. \tag{4.7.14}$$

4.7.4 Training a Model

When training networks, forward and backward propagation depend on each other. In particular, for forward propagation, we traverse the compute graph in the direction of dependencies and compute all the variables on its path. These are then used for backpropagation where the compute order on the graph is reversed. One of the consequences is that we need to retain the intermediate values until backpropagation is complete. This is also one of the reasons why backpropagation requires significantly more memory than plain "inference"—we end up computing tensors as gradients and need to retain all the intermediate variables to invoke the chain rule. Another reason is that we typically train with minibatches containing more than one variable, thus more intermediate activations need to be stored.

Summary

- Forward propagation sequentially calculates and stores intermediate variables within the compute graph defined by the neural network. It proceeds from input to output layer.
- Back propagation sequentially calculates and stores the gradients of intermediate variables and parameters within the neural network in the reversed order.
- When training deep learning models, forward propagation and back propagation are interdependent.
- Training requires significantly more memory and storage.

Exercises

1. Assume that the inputs $\mathbf{x}$ are matrices. What is the dimensionality of the gradients?
2. Add a bias to the hidden layer of the model described in this section.
    - Draw the corresponding compute graph.
    - Derive the forward and backward propagation equations.
3. Compute the memory footprint for training and inference in model described in the current chapter.
4. Assume that you want to compute *second* derivatives. What happens to the compute graph? Is this a good idea?
5. Assume that the compute graph is too large for your GPU.
    - Can you partition it over more than one GPU?
    - What are the advantages and disadvantages over training on a smaller minibatch?

4.8 Numerical Stability and Initialization

In the past few sections, each model that we implemented required initializing our parameters according to some specified distribution. However, until now, we glossed over the details, taking the initialization hyperparameters for granted. You might even have gotten the impression that these choices are not especially important. However, the choice of initialization scheme plays a significant role in neural network learning, and can prove essentially to maintaining numerical stability. Moreover, these choices can be tied up in interesting ways with the choice of the activation function. Which nonlinear activation function we choose, and how we decide to initialize our parameters can play a crucial role in making the optimization algorithm converge rapidly. Failure to be mindful of these issues can lead to either exploding or vanishing gradients. In this section, we delve into these topics with greater detail and discuss some useful heuristics that you may use frequently throughout your career in deep learning.

4.8.1 Vanishing and Exploding Gradients

Consider a deep network with d layers, input $\mathbf{x}$ and output $\mathbf{o}$. Each layer satisfies:

$$\mathbf{h}^{t+1} = f_t(\mathbf{h}^t) \text{ and thus } \mathbf{o} = f_d \circ \ldots, \circ f_1(\mathbf{x}). \tag{4.8.1}$$

If all activations and inputs are vectors, we can write the gradient of $\mathbf{o}$ with respect to any set of parameters $\mathbf{W}_t$ associated with the function f_t at layer t simply as

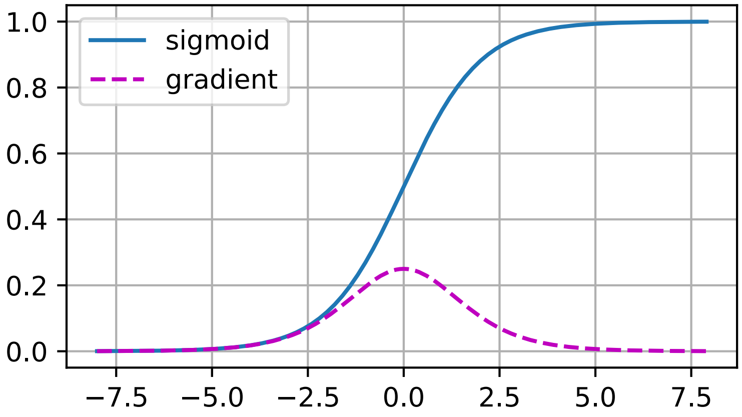
$$\partial_{\mathbf{W}_t} \mathbf{o} = \underbrace{\partial_{\mathbf{h}^{d-1}} \mathbf{h}^d}_{:=\mathbf{M}_d} \cdot \ldots, \cdot \underbrace{\partial_{\mathbf{h}^t} \mathbf{h}^{t+1}}_{:=\mathbf{M}_t} \underbrace{\partial_{\mathbf{W}_t} \mathbf{h}^t}_{:=\mathbf{v}_t}. \tag{4.8.2}$$

In other words, it is the product of $d - t$ matrices $\mathbf{M}_d \cdot \ldots, \cdot \mathbf{M}_t$ and the gradient vector $\mathbf{v}_t$. What happens is similar to the situation when we experienced numerical underflow when multiplying too many probabilities. At the time, we were able to mitigate the problem by switching from into log-space, i.e., by shifting the problem from the mantissa to the exponent of the numerical representation. Unfortunately the problem outlined in the equation above is much more serious: initially the matrices M_t may well have a wide variety of eigenvalues. They might be small, they might be large, and in particular, their product might well be *very large* or *very small*. This is not (only) a problem of numerical representation but it means that the optimization algorithm is bound to fail. It receives gradients that are either excessively large or excessively small. As a result the steps taken are either (i) excessively large (the *exploding* gradient problem), in which case the parameters blow up in magnitude rendering the model useless, or (ii) excessively small, (the *vanishing gradient problem*), in which case the parameters hardly move at all, and thus the learning process makes no progress.

Vanishing Gradients

One major culprit in the vanishing gradient problem is the choices of the activation functions σ that are interleaved with the linear operations in each layer. Historically, the sigmoid function $(1+\exp(-x))$ (introduced in Section 4.1) was a popular choice owing to its similarity to a thresholding function. Since early artificial neural networks were inspired by biological neural networks, the idea of neurons that either fire or do not fire (biological neurons do not partially fire) seemed appealing. Let's take a closer look at the function to see why picking it might be problematic vis-a-vis vanishing gradients.

```
%matplotlib inline
import d2l
from mxnet import autograd, np, npx
npx.set_np()

x = np.arange(-8.0, 8.0, 0.1)
x.attach_grad()
with autograd.record():
    y = npx.sigmoid(x)
y.backward()

d2l.plot(x, [y, x.grad], legend=['sigmoid', 'gradient'], figsize=(4.5, 2.5))
```



As we can see, the gradient of the sigmoid vanishes both when its inputs are large and when they are small. Moreover, when we execute backward propagation, due to the chain rule, this means that unless we are in the Goldilocks zone, where the inputs to most of the sigmoids are in the range of, say $[-4, 4]$, the gradients of the overall product may vanish. When we have many layers, unless we are especially careful, we are likely to find that our gradient is cut off at *some* layer. Before ReLUs ($\max(0, x)$) were proposed as an alternative to squashing functions, this problem used to plague deep network training. As a consequence, ReLUs have become the default choice when designing activation functions in deep networks.

Exploding Gradients

The opposite problem, when gradients explode, can be similarly vexing. To illustrate this a bit better, we draw 100 Gaussian random matrices and multiply them with some initial matrix. For the scale that we picked (the choice of the variance $\sigma^2 = 1$), the matrix product explodes. If this were to happen to us with a deep network, we would have no realistic chance of getting a gradient descent optimizer to converge.

```
M = np.random.normal(size=(4, 4))
print('A single matrix', M)
for i in range(100):
    M = np.dot(M, np.random.normal(size=(4, 4)))

print('After multiplying 100 matrices', M)
```

```
A single matrix [[ 2.2122064   1.1630787   0.7740038   0.4838046 ]
 [ 1.0434405   0.29956347  1.1839255   0.15302546]
 [ 1.8917114  -1.1688148  -1.2347414   1.5580711 ]
 [-1.771029   -0.5459446  -0.45138445 -2.3556297 ]]
After multiplying 100 matrices [[ 3.4459714e+23 -7.8040680e+23  5.9973287e+23  4.5229990e+23]
 [ 2.5275089e+23 -5.7240326e+23  4.3988473e+23  3.3174740e+23]
 [ 1.3731286e+24 -3.1097155e+24  2.3897773e+24  1.8022959e+24]
 [-4.4951040e+23  1.0180033e+24 -7.8232281e+23 -5.9000354e+23]]
```

Symmetry

Another problem in deep network design is the symmetry inherent in their parametrization. Assume that we have a deep network with one hidden layer with two units, say h_1 and h_2. In this case, we could permute the weights $\mathbf{W}_1$ of the first layer and likewise permute the weights of the output layer to obtain the same function. There is nothing special differentiating the first hidden unit vs the second hidden unit. In other words, we have permutation symmetry among the hidden units of each layer.

This is more than just a theoretical nuisance. Imagine what would happen if we initialized all of the parameters of some layer as $\mathbf{W}_l = c$ for some constant c. In this case, the gradients for all dimensions are identical: thus not only would each unit take the same value, but it would receive the same update. Stochastic gradient descent would never break the symmetry on its own and we might never be able to realize the networks expressive power. The hidden layer would behave as if it had only a single unit. As an aside, note that while SGD would not break this symmetry, dropout regularization would!

4.8.2 Parameter Initialization

One way of addressing, or at least mitigating the issues raised above is through careful initialization of the weight vectors. This way we can ensure that (at least initially) the gradients do not vanish and that they maintain a reasonable scale where the network weights do not diverge. Additional care during optimization and suitable regularization ensures that things never get too bad.

Default Initialization

In the previous sections, e.g., in Section 3.3, we used `net.initialize(init.Normal(sigma=0.01))` to initialize the values of our weights. If the initialization method is not specified, such as `net.initialize()`, MXNet will use the default random initialization method: each element of the weight parameter is randomly sampled with a uniform distribution $U[-0.07, 0.07]$ and the bias parameters are all set to 0. Both choices tend to work well in practice for moderate problem sizes.

Xavier Initialization

Let's look at the scale distribution of the activations of the hidden units h_i for some layer. They are given by

$$h_i = \sum_{j=1}^{n_\mathrm{in}} W_{ij} x_j. \tag{4.8.3}$$

The weights W_{ij} are all drawn independently from the same distribution. Furthermore, let's assume that this distribution has zero mean and variance σ^2 (this does not mean that the distribution has to be Gaussian, just that mean and variance need to exist). We do not really have much control over the inputs into the layer x_j but let's proceed with the somewhat unrealistic assumption that they also have zero mean and variance γ^2 and that they are independent of $\mathbf{W}$. In this case, we can compute mean and variance of h_i as follows:

$$\begin{aligned}
E[h_i] & = \sum_{j=1}^{n_\mathrm{in}} E[W_{ij} x_j] = 0, \\
E[h_i^2] & = \sum_{j=1}^{n_\mathrm{in}} E[W^2_{ij} x^2_j] \\
& = \sum_{j=1}^{n_\mathrm{in}} E[W^2_{ij}] E[x^2_j] \\
& = n_\mathrm{in} \sigma^2 \gamma^2.
\end{aligned} \tag{4.8.4}$$

One way to keep the variance fixed is to set $n_\mathrm{in} \sigma^2 = 1$. Now consider backpropagation. There we face a similar problem, albeit with gradients being propagated from the top layers. That is, instead of $\mathbf{W} \mathbf{w}$, we need to deal with $\mathbf{W}^\top \mathbf{g}$, where $\mathbf{g}$ is the incoming gradient from the layer above. Using the same reasoning as for forward propagation, we see that the gradients' variance can blow up unless $n_\mathrm{out} \sigma^2 = 1$. This leaves us in a dilemma: we cannot possibly satisfy both conditions simultaneously. Instead, we simply try to satisfy:

$$\frac{1}{2} (n_\mathrm{in} + n_\mathrm{out}) \sigma^2 = 1 \text{ or equivalently } \sigma = \sqrt{\frac{2}{n_\mathrm{in} + n_\mathrm{out}}}. \tag{4.8.5}$$

This is the reasoning underlying the eponymous Xavier initialization (Glorot & Bengio, 2010). It works well enough in practice. For Gaussian random variables, the Xavier initialization picks a normal distribution with zero mean and variance $\sigma^2 = 2/(n_\mathrm{in} + n_\mathrm{out})$. For uniformly distributed random variables $U[-a, a]$, note that their variance is given by $a^2/3$. Plugging $a^2/3$ into the condition on σ^2 yields that we should initialize uniformly with $U\left[-\sqrt{6/(n_\mathrm{in} + n_\mathrm{out})}, \sqrt{6/(n_\mathrm{in} + n_\mathrm{out})}\right]$.

Beyond

The reasoning above barely scratches the surface of modern approaches to parameter initialization. In fact, MXNet has an entire `mxnet.initializer` module implementing over a dozen different heuristics. Moreover, initialization continues to be a hot area of inquiry within research into the fundamental theory of neural network optimization. Some of these heuristics are especially suited for when parameters are tied (i.e., when parameters of in different parts the network are shared), for super-resolution, sequence models, and related problems. We recommend that the interested reader take a closer look at what is offered as part of this module, and investigate the recent research on parameter initialization. Perhaps you may come across a recent clever idea and contribute its implementation to MXNet, or you may even invent your own scheme!

Summary

- Vanishing and exploding gradients are common issues in very deep networks, unless great care is taking to ensure that gradients and parameters remain well controlled.
- Initialization heuristics are needed to ensure that at least the initial gradients are neither too large nor too small.
- The ReLU addresses one of the vanishing gradient problems, namely that gradients vanish for very large inputs. This can accelerate convergence significantly.
- Random initialization is key to ensure that symmetry is broken before optimization.

Exercises

1. Can you design other cases of symmetry breaking besides the permutation symmetry?
2. Can we initialize all weight parameters in linear regression or in softmax regression to the same value?
3. Look up analytic bounds on the eigenvalues of the product of two matrices. What does this tell you about ensuring that gradients are well conditioned?
4. If we know that some terms diverge, can we fix this after the fact? Look at the paper on LARS for inspiration (You et al., 2017).

4.9 Considering the Environment

So far, we have worked through a number of hands-on implementations fitting machine learning models to a variety of datasets. And yet, until now we skated over the matter of where are data comes from in the first place, and what we plan to ultimately *do* with the outputs from our models. Too often in the practice of machine learning, developers rush ahead with the development of models tossing these fundamental considerations aside.

Many failed machine learning deployments can be traced back to this situation. Sometimes the model does well as evaluated by test accuracy only to fail catastrophically in the real world when the distribution of data suddenly shifts. More insidiously, sometimes the very deployment of a model can be the catalyst which perturbs the data distribution. Say for example that we trained a model to predict loan defaults, finding that the choice of footware was associated with risk of default (Oxfords indicate repayment, sneakers indicate default). We might be inclined to thereafter grant loans to all applicants wearing Oxfords and to deny all applicants wearing sneakers. But our ill-conceived leap from pattern recognition to decision-making and our failure to think critically about the environment might have disastrous consequences. For starters, as soon as we began making decisions based on footware, customers would catch on and change their behavior. Before long, all applicants would be wearing Oxfords, and yet there would be no coinciding improvement in credit-worthiness. Think about this deeply because similar issues abound in the application of machine learning: by introducing our model-based decisions to the environment, we might break the model.

In this section, we describe some common concerns and aim to get you started acquiring the critical thinking that you will need in order to detect these situations early, mitigate the damage, and use machine learning responsibly. Some of the solutions are simple (ask for the "right" data) some are technically difficult (implement a reinforcement learning system), and others require that we enter the realm of philosophy and grapple with difficult questions concerning ethics and informed consent.

4.9.1 Distribution Shift

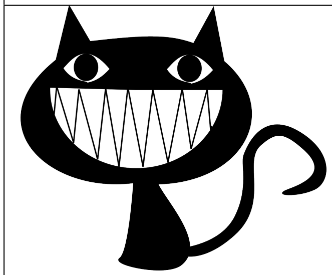
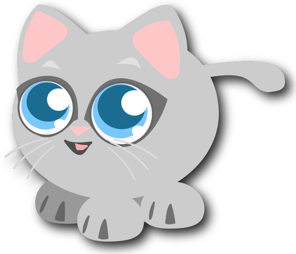
To begin, we return to the observational setting, putting aside for now the impacts of our actions on the environment. In the following sections, we take a deeper look at the various ways that data distributions might shift, and what might be done to salvage model performance. From the outset, we should warn that if the data-generating distribution $p(\mathbf{x}, y)$ can shift in arbitrary ways at any point in time, then learning a robust classifier is impossible. In the most pathological case, if the label definitions themselves can change at a moments notice: if suddenly what we called "cats" are now dogs and what we previously called "dogs" are now in fact cats, without any perceptible change in the distribution of inputs $p(\mathbf{x})$, then there is nothing we could do to detect the change or to correct our classifier at test time. Fortunately, under some restricted assumptions on the ways our data might change in the future, principled algorithms can detect shift and possibly even adapt, achieving higher accuracy than if we naively continued to rely on our original classifier.

Covariate Shift

One of the best-studied forms of distribution shift is *covariate shift*. Here we assume that although the distribution of inputs may change over time, the labeling function, i.e., the conditional distribution $P(y \mid \mathbf{x})$ does not change. While this problem is easy to understand its also easy to overlook it in practice. Consider the challenge of distinguishing cats and dogs. Our training data consists of images of the following kind:

cat	cat	dog	dog

At test time we are asked to classify the following images:

cat	cat	dog	dog
			

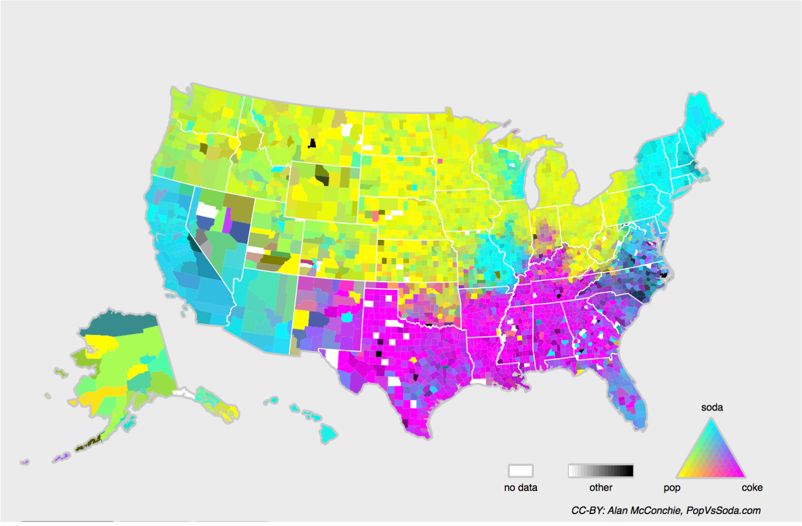
Obviously this is unlikely to work well. The training set consists of photos, while the test set contains only cartoons. The colors are not even realistic. Training on a dataset that looks substantially different from the test set without some plan for how to adapt to the new domain is a bad idea. Unfortunately, this is a very common pitfall. Statisticians call this *covariate shift* because the root of the problem owed to a shift in the distribution of features (i.e., of *covariates*). Mathematically, we could say that $P(\mathbf{x})$ changes but that $P(y \mid \mathbf{x})$ remains unchanged. Although its usefulness is not restricted to this setting, when we believe $\mathbf{x}$ causes y, covariate shift is usually the right assumption to be working with.

Label Shift

The converse problem emerges when we believe that what drives the shift is a change in the marginal distribution over the labels $P(y)$ but that the class-conditional distributions are invariant $P(\mathbf{x} \mid y)$. Label shift is a reasonable assumption to make when we believe that y causes $\mathbf{x}$. For example, commonly we want to predict a diagnosis given its manifestations. In this case we believe that the diagnosis causes the manifestations, i.e., diseases cause symptoms. Sometimes the label shift and covariate shift assumptions can hold simultaneously. For example, when the true labeling function is deterministic and unchanging, then covariate shift will always hold, including if label shift holds too. Interestingly, when we expect both label shift and covariate shift hold, it is often advantageous to work with the methods that flow from the label shift assumption. That is because these methods tend to involve manipulating objects that look like the label, which (in deep learning) tends to be comparatively easy compared to working with the objects that look like the input, which tends (in deep learning) to be a high-dimensional object.

Concept Shift

One more related problem arises in *concept shift*, the situation in which the very label definitions change. This sounds weird—after all, a *cat* is a *cat*. Indeed the definition of a cat might not change, but can we say the same about soft drinks? It turns out that if we navigate around the United States, shifting the source of our data by geography, we will find considerable concept shift regarding the definition of even this simple term as shown in Fig. 4.9.1.



CC-BY: Alan McConchie, PopVsSoda.com

Fig. 4.9.1: Concept shift on soft drink names in the United States.

If we were to build a machine translation system, the distribution $P(y \mid x)$ might be different depending on our location. This problem can be tricky to spot. A saving grace is that often the $P(y \mid x)$ only shifts gradually.

Examples

Before we go into further detail and discuss remedies, we can discuss a number of situations where covariate and concept shift may not be so obvious.

Medical Diagnostics

Imagine that you want to design an algorithm to detect cancer. You collect data from healthy and sick people and you train your algorithm. It works fine, giving you high accuracy and you conclude that you're ready for a successful career in medical diagnostics. Not so fast…

Many things could go wrong. In particular, the distributions that you work with for training and those that you encounter in the wild might differ considerably. This happened to an unfortunate startup, that Alex had the opportunity to consult for many years ago. They were developing a blood test for a disease that affects mainly older men and they'd managed to obtain a fair amount of blood samples from patients. It is considerably more difficult, though, to obtain blood samples from healthy men (mainly for ethical reasons). To compensate for that, they asked a large number of students on campus to donate blood and they performed their test. Then they asked me whether I could help them build a classifier to detect the disease. I told them that it would be very easy to distinguish between both datasets with near-perfect accuracy. After all, the test subjects differed in age, hormone levels, physical activity, diet, alcohol consumption, and many more factors unrelated to the disease. This was unlikely to be the case with real patients: Their sampling procedure made it likely that an extreme case of covariate shift would arise between the *source* and *target* distributions, and at that, one that could not be corrected by conventional means. In other words, training and test data were so different that nothing useful could be done and they had wasted significant amounts of money.

Self Driving Cars

Say a company wanted to build a machine learning system for self-driving cars. One of the key components is a roadside detector. Since real annotated data is expensive to get, they had the (smart and questionable) idea to use synthetic data from a game rendering engine as additional training data. This worked really well on "test data" drawn from the rendering engine. Alas, inside a real car it was a disaster. As it turned out, the roadside had been rendered with a very simplistic texture. More importantly, *all* the roadside had been rendered with the *same* texture and the roadside detector learned about this "feature" very quickly.

A similar thing happened to the US Army when they first tried to detect tanks in the forest. They took aerial photographs of the forest without tanks, then drove the tanks into the forest and took another set of pictures. The so-trained classifier worked "perfectly". Unfortunately, all it had learned was to distinguish trees with shadows from trees without shadows—the first set of pictures was taken in the early morning, the second one at noon.

Nonstationary distributions

A much more subtle situation arises when the distribution changes slowly and the model is not updated adequately. Here are some typical cases:

- We train a computational advertising model and then fail to update it frequently (e.g., we forget to incorporate that an obscure new device called an iPad was just launched).
- We build a spam filter. It works well at detecting all spam that we have seen so far. But then the spammers wisen up and craft new messages that look unlike anything we have seen before.
- We build a product recommendation system. It works throughout the winter… but then it keeps on recommending Santa hats long after Christmas.

More Anecdotes

- We build a face detector. It works well on all benchmarks. Unfortunately it fails on test data—the offending examples are close-ups where the face fills the entire image (no such data was in the training set).
- We build a web search engine for the USA market and want to deploy it in the UK.
- We train an image classifier by compiling a large dataset where each among a large set of classes is equally represented in the dataset, say 1000 categories, represented by 1000 images each. Then we deploy the system in the real world, where the actual label distribution of photographs is decidedly non-uniform.

In short, there are many cases where training and test distributions $p(\mathbf{x}, y)$ are different. In some cases, we get lucky and the models work despite covariate, label, or concept shift. In other cases, we can do better by employing principled strategies to cope with the shift. The remainder of this section grows considerably more technical. The impatient reader could continue on to the next section as this material is not prerequisite to subsequent concepts.

Covariate Shift Correction

Assume that we want to estimate some dependency $P(y \mid \mathbf{x})$ for which we have labeled data $(\mathbf{x}_i, y_i)$. Unfortunately, the observations x_i are drawn from some *target* distribution $q(\mathbf{x})$ rather than the *source* distribution $p(\mathbf{x})$. To make progress, we need to reflect about what exactly is happening during training: we iterate over training data and associated labels $\{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$ and update the weight vectors of the model after every minibatch. We sometimes additionally apply some penalty to the parameters, using weight decay, dropout, or some other related technique. This means that we largely minimize the loss on the training.

$$\underset{w}{\text{minimize}} \frac{1}{n} \sum_{i=1}^{n} l(x_i, y_i, f(x_i)) + \text{some penalty}(w). \tag{4.9.1}$$

Statisticians call the first term an *empirical average*, i.e., an average computed over the data drawn from $P(x)P(y \mid x)$. If the data is drawn from the "wrong" distribution q, we can correct for that by using the following simple identity:

$$\int p(\mathbf{x}) f(\mathbf{x}) dx = \int q(\mathbf{x}) f(\mathbf{x}) \frac{p(\mathbf{x})}{q(\mathbf{x})} dx. \tag{4.9.2}$$

In other words, we need to re-weight each instance by the ratio of probabilities that it would have been drawn from the correct distribution $\beta(\mathbf{x}) := p(\mathbf{x})/q(\mathbf{x})$. Alas, we do not know that ratio, so before we can do anything useful we need to estimate it. Many methods are available, including some fancy operator-theoretic approaches that attempt to recalibrate the expectation operator directly using a minimum-norm or a maximum entropy principle. Note that for any such approach, we need samples drawn from both distributions—the "true" p, e.g., by access to training data, and the one used for generating the training set q (the latter is trivially available). Note however, that we only need samples $\mathbf{x} \sim q(\mathbf{x})$; we do not to access labels $y \sim q(y)$.

In this case, there exists a very effective approach that will give almost as good results: logistic regression. This is all that is needed to compute estimate probability ratios. We learn a classifier to distinguish between data drawn from $p(\mathbf{x})$ and data drawn from $q(x)$. If it is impossible to distinguish between the two distributions then it means that the associated instances are equally likely to come from either one of the two distributions. On the other hand, any instances that can be well discriminated should be significantly overweighted or underweighted accordingly. For simplicity's sake assume that we have an equal number of instances from both distributions, denoted by $\mathbf{x}_i \sim p(\mathbf{x})$ and $\mathbf{x}_i' \sim q(\mathbf{x})$ respectively. Now denote by z_i labels which are 1 for data drawn from p and -1 for data drawn from q. Then the probability in a mixed dataset is given by

$$P(z = 1 \mid \mathbf{x}) = \frac{p(\mathbf{x})}{p(\mathbf{x}) + q(\mathbf{x})} \text{ and hence } \frac{P(z = 1 \mid \mathbf{x})}{P(z = -1 \mid \mathbf{x})} = \frac{p(\mathbf{x})}{q(\mathbf{x})}. \tag{4.9.3}$$

Hence, if we use a logistic regression approach where $P(z = 1 \mid \mathbf{x}) = \frac{1}{1+\exp(-f(\mathbf{x}))}$ it follows that

$$\beta(\mathbf{x}) = \frac{1/(1 + \exp(-f(\mathbf{x})))}{\exp(-f(\mathbf{x})/(1 + \exp(-f(\mathbf{x})))} = \exp(f(\mathbf{x})). \tag{4.9.4}$$

As a result, we need to solve two problems: first one to distinguish between data drawn from both distributions, and then a reweighted minimization problem where we weigh terms by β, e.g., via the head gradients. Here's a prototypical algorithm for that purpose which uses an unlabeled training set X and test set Z:

1. Generate training set with $\{(\mathbf{x}_i, -1)...(\mathbf{z}_j, 1)\}$.
2. Train binary classifier using logistic regression to get function f.
3. Weigh training data using $\beta_i = \exp(f(\mathbf{x}_i))$ or better $\beta_i = \min(\exp(f(\mathbf{x}_i)), c)$.
4. Use weights β_i for training on X with labels Y.

Note that this method relies on a crucial assumption. For this scheme to work, we need that each data point in the target (test time)distribution had nonzero probability of occurring at training time. If we find a point where $q(\mathbf{x}) > 0$ but $p(\mathbf{x}) = 0$, then the corresponding importance weight should be infinity.

Generative Adversarial Networks use a very similar idea to that described above to engineer a *data generator* that outputs data that cannot be distinguished from examples sampled from a reference dataset. In these approaches, we use one network, f to distinguish real versus fake data and a second network g that tries to fool the discriminator f into accepting fake data as real. We will discuss this in much more detail later.

Label Shift Correction

For the discussion of label shift, we will assume for now that we are dealing with a k-way multiclass classification task. When the distribution of labels shifts over time $p(y) \neq q(y)$ but the class-conditional distributions stay the same $p(\mathbf{x}) = q(\mathbf{x})$, our importance weights will correspond to the label likelihood ratios $q(y)/p(y)$. One nice thing about label shift is that if we have a reasonably good model (on the source distribution) then we can get consistent estimates of these weights without ever having to deal with the ambient dimension (in deep learning, the inputs are often high-dimensional perceptual objects like images, while the labels are often easier to work, say vectors whose length corresponds to the number of classes).

To estimate calculate the target label distribution, we first take our reasonably good off the shelf classifier (typically trained on the training data) and compute its confusion matrix using the validation set (also from the training distribution). The confusion matrix C, is simply a $k \times k$ matrix where each column corresponds to the *actual* label and each row corresponds to our model's predicted label. Each cell's value c_{ij} is the fraction of predictions where the true label was j *and* our model predicted y.

Now we cannot calculate the confusion matrix on the target data directly, because we do not get to see the labels for the examples that we see in the wild, unless we invest in a complex real-time annotation pipeline. What we can do, however, is average all of our models predictions at test time together, yielding the mean model output μ_y.

It turns out that under some mild conditions— if our classifier was reasonably accurate in the first place, if the target data contains only classes of images that we have seen before, and if the label shift assumption holds in the first place (far the strongest assumption here), then we can recover the test set label distribution by solving a simple linear system $C \cdot q(y) = \mu_y$. If our classifier is sufficiently accurate to begin with, then the confusion C will be invertible, and we get a solution $q(y) = C^{-1}\mu_y$. Here we abuse notation a bit, using $q(y)$ to denote the vector of label frequencies. Because we observe the labels on the source data, it is easy to estimate the distribution $p(y)$. Then for any training example i with label y, we can take the ratio of our estimates $\hat{q}(y)/\hat{p}(y)$ to calculate the weight w_i, and plug this into the weighted risk minimization algorithm above.

Concept Shift Correction

Concept shift is much harder to fix in a principled manner. For instance, in a situation where suddenly the problem changes from distinguishing cats from dogs to one of distinguishing white from black animals, it will be unreasonable to assume that we can do much better than just collecting new labels and training from scratch. Fortunately, in practice, such extreme shifts are rare. Instead, what usually happens is that the task keeps on changing slowly. To make things more concrete, here are some examples:

- In computational advertising, new products are launched, old products become less popular. This means that the distribution over ads and their popularity changes gradually and any click-through rate predictor needs to change gradually with it.
- Traffic cameras lenses degrade gradually due to environmental wear, affecting image quality progressively.
- News content changes gradually (i.e., most of the news remains unchanged but new stories appear).

In such cases, we can use the same approach that we used for training networks to make them adapt to the change in the data. In other words, we use the existing network weights and simply perform a few update steps with the new data rather than training from scratch.

4.9.2 A Taxonomy of Learning Problems

Armed with knowledge about how to deal with changes in $p(x)$ and in $P(y \mid x)$, we can now consider some other aspects of machine learning problems formulation.

- **Batch Learning.** Here we have access to training data and labels $\{(x_1, y_1), \ldots, (x_n, y_n)\}$, which we use to train a network $f(x, w)$. Later on, we deploy this network to score new data (x, y) drawn from the same distribution. This is the default assumption for any of the problems that we discuss here. For instance, we might train a cat detector based on lots of pictures of cats and dogs. Once we trained it, we ship it as part of a smart catdoor computer vision system that lets only cats in. This is then installed in a customer's home and is never updated again (barring extreme circumstances).
- **Online Learning.** Now imagine that the data (x_i, y_i) arrives one sample at a time. More specifically, assume that we first observe x_i, then we need to come up with an estimate $f(x_i, w)$ and only once we have done this, we observe y_i and with it, we receive a reward (or incur a loss), given our decision. Many real problems fall into this category. E.g. we need to predict tomorrow's stock price, this allows us to trade based on that estimate and at the end of the day we find out whether our estimate allowed us to make a profit. In other words, we have the following cycle where we are continuously improving our model given new observations.

$$\text{model } f_t \longrightarrow \text{data } x_t \longrightarrow \text{estimate } f_t(x_t) \longrightarrow \text{observation } y_t \longrightarrow \text{loss } l(y_t, f_t(x_t)) \longrightarrow \text{model } f_{t+1} \tag{4.9.5}$$

- **Bandits.** They are a *special case* of the problem above. While in most learning problems we have a continuously parametrized function f where we want to learn its parameters (e.g., a deep network), in a bandit problem we only have a finite number of arms that we can pull (i.e., a finite number of actions that we can take). It is not very surprising that for this simpler problem stronger theoretical guarantees in terms of optimality can be obtained. We list it mainly since this problem is often (confusingly) treated as if it were a distinct learning setting.

- **Control (and nonadversarial Reinforcement Learning).** In many cases the environment remembers what we did. Not necessarily in an adversarial manner but it'll just remember and the response will depend on what happened before. E.g. a coffee boiler controller will observe different temperatures depending on whether it was heating the boiler previously. PID (proportional integral derivative) controller algorithms are a popular choice there. Likewise, a user's behavior on a news site will depend on what we showed him previously (e.g., he will read most news only once). Many such algorithms form a model of the environment in which they act such as to make their decisions appear less random (i.e., to reduce variance).
- **Reinforcement Learning.** In the more general case of an environment with memory, we may encounter situations where the environment is trying to *cooperate* with us (cooperative games, in particular for non-zero-sum games), or others where the environment will try to *win*. Chess, Go, Backgammon or StarCraft are some of the cases. Likewise, we might want to build a good controller for autonomous cars. The other cars are likely to respond to the autonomous car's driving style in nontrivial ways, e.g., trying to avoid it, trying to cause an accident, trying to cooperate with it, etc.

One key distinction between the different situations above is that the same strategy that might have worked throughout in the case of a stationary environment, might not work throughout when the environment can adapt. For instance, an arbitrage opportunity discovered by a trader is likely to disappear once he starts exploiting it. The speed and manner at which the environment changes determines to a large extent the type of algorithms that we can bring to bear. For instance, if we *know* that things may only change slowly, we can force any estimate to change only slowly, too. If we know that the environment might change instantaneously, but only very infrequently, we can make allowances for that. These types of knowledge are crucial for the aspiring data scientist to deal with concept shift, i.e., when the problem that he is trying to solve changes over time.

4.9.3 Fairness, Accountability, and Transparency in Machine Learning

Finally, it is important to remember that when you deploy machine learning systems you are not simply minimizing negative log likelihood or maximizing accuracy—you are automating some kind of decision process. Often the automated decision-making systems that we deploy can have consequences for those subject to its decisions. If we are deploying a medical diagnostic system, we need to know for which populations it may work and which it may not. Overlooking foreseeable risks to the welfare of a subpopulation would run afoul of basic ethical principles. Moreover, "accuracy" is seldom the right metric. When translating predictions in to actions we will often want to take into account the potential cost sensitivity of erring in various ways. If one way that you might classify an image could be perceived as a racial sleight, while misclassification to a different category would be harmless, then you might want to adjust your thresholds accordingly, accounting for societal values in designing the decision-making protocol. We also want to be careful about how prediction systems can lead to feedback loops. For example, if prediction systems are applied naively to predictive policing, allocating patrol officers accordingly, a vicious cycle might emerge. Neighborhoods that have more crimes, get more patrols, get more crimes discovered, get more training data, get yet more confident predictions, leading to even more patrols, even more crimes discovered, etc. Additionally, we want to be careful about whether we are addressing the right problem in the first place. Predictive algorithms now play an outsize role in mediating the dissemination of information. Should what news someone is exposed to be determined by which

Facebook pages they have *Liked*? These are just a few among the many profound ethical dilemmas that you might encounter in a career in machine learning.

Summary

- In many cases training and test set do not come from the same distribution. This is called covariate shift.
- Covariate shift can be detected and corrected if the shift is not too severe. Failure to do so leads to nasty surprises at test time.
- In some cases the environment *remembers* what we did and will respond in unexpected ways. We need to account for that when building models.

Exercises

1. What could happen when we change the behavior of a search engine? What might the users do? What about the advertisers?
2. Implement a covariate shift detector. Hint: build a classifier.
3. Implement a covariate shift corrector.
4. What could go wrong if training and test set are very different? What would happen to the sample weights?

4.10 Predicting House Prices on Kaggle

In the previous sections, we introduced the basic tools for building deep networks and performing capacity control via dimensionality-reduction, weight decay and dropout. You are now ready to put all this knowledge into practice by participating in a Kaggle competition. Predicting house prices[80] is a great place to start: the data is reasonably generic and does not have the kind of rigid structure that might require specialized models the way images or audio might. This dataset, collected by Bart de Cock in 2011 (DeCock, 2011), is considerably larger than the famous the Boston housing dataset[81] of Harrison and Rubinfeld (1978). It boasts both more examples and more features, covering house prices in Ames, IA from the period of 2006-2010.

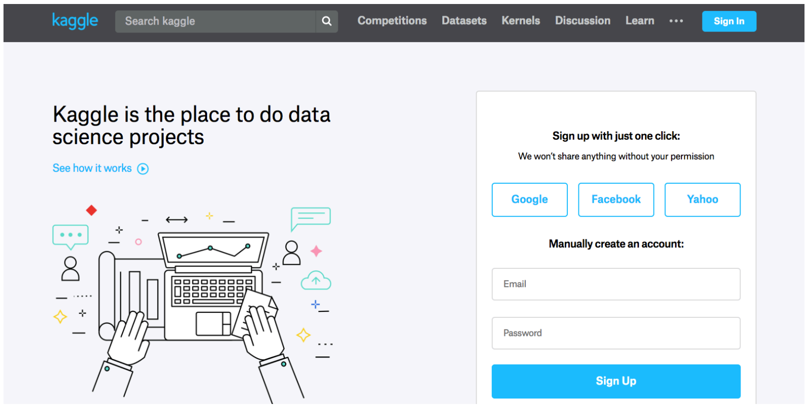
In this section, we will walk you through details of data preprocessing, model design, hyperparameter selection and tuning. We hope that through a hands-on approach, you will be able to observe the effects of capacity control, feature extraction, etc. in practice. This experience is vital to gaining intuition as a data scientist.

[80] https://www.kaggle.com/c/house-prices-advanced-regression-techniques

[81] https://archive.ics.uci.edu/ml/machine-learning-databases/housing/housing.names

4.10.1 Kaggle

Kaggle[82] is a popular platform for machine learning competitions. It combines data, code and users in a way to allow for both collaboration and competition. While leaderboard chasing can sometimes get out of control, there is also a lot to be said for the objectivity in a platform that provides fair and direct quantitative comparisons between your approaches and those devised by your competitors. Moreover, you can checkout the code from (some) other competitors' submissions and pick apart their methods to learn new techniques. If you want to participate in one of the competitions, you need to register for an account as shown in Fig. 4.10.1 (do this now!).



Kaggle is the place to do data science projects

Sign up with just one click:

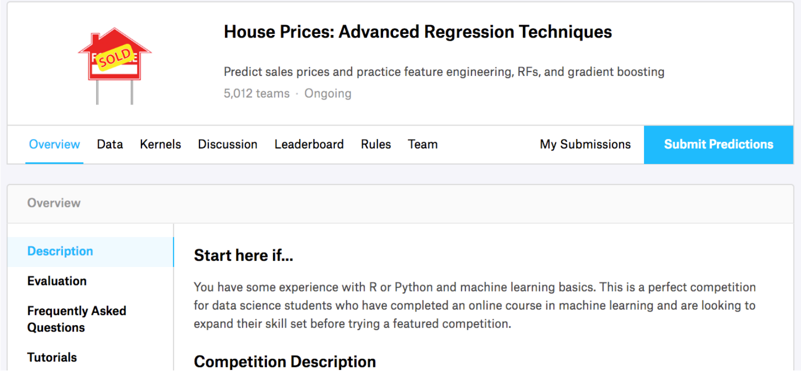
We won't share anything without your permission

Manually create an account:

Fig. 4.10.1: Kaggle website

On the House Prices Prediction page as illustrated in Fig. 4.10.2, you can find the dataset (under the "Data" tab), submit predictions, see your ranking, etc., The URL is right here:

https://www.kaggle.com/c/house-prices-advanced-regression-techniques



House Prices: Advanced Regression Techniques

Predict sales prices and practice feature engineering, RFs, and gradient boosting

5,012 teams · Ongoing

Start here if...

You have some experience with R or Python and machine learning basics. This is a perfect competition for data science students who have completed an online course in machine learning and are looking to expand their skill set before trying a featured competition.

Competition Description

Fig. 4.10.2: House Price Prediction

[82] https://www.kaggle.com

4.10.2 Accessing and Reading the Dataset

Note that the competition data is separated into training and test sets. Each record includes the property value of the house and attributes such as street type, year of construction, roof type, basement condition, etc. The features represent multiple data types. Year of construction, for example, is represented with integers roof type is a discrete categorical feature, other features are represented with floating point numbers. And here is where reality comes in: for some examples, some data is altogether missing with the missing value marked simply as 'na'. The price of each house is included for the training set only (it is a competition after all). You can partition the training set to create a validation set, but you will only find out how you perform on the official test set when you upload your predictions and receive your score. The "Data" tab on the competition tab has links to download the data.

We will read and process the data using `pandas`, an efficient data analysis toolkit[83], so you will want to make sure that you have `pandas` installed before proceeding further. Fortunately, if you are reading in Jupyter, we can install pandas without even leaving the notebook.

```
# If pandas is not installed, please uncomment the following line:
# !pip install pandas

%matplotlib inline
import d2l
from mxnet import autograd, gluon, init, np, npx
from mxnet.gluon import nn
import pandas as pd
npx.set_np()
```

For convenience, we already downloaded the data and stored it in the `../data` directory. To load the two CSV (Comma Separated Values) files containing training and test data respectively we use Pandas.

```
train_data = pd.read_csv('../data/kaggle_house_pred_train.csv')
test_data = pd.read_csv('../data/kaggle_house_pred_test.csv')
```

The training dataset includes $1,460$ examples, 80 features, and 1 label, while the test data contains $1,459$ examples and 80 features.

```
print(train_data.shape)
print(test_data.shape)
```

```
(1460, 81)
(1459, 80)
```

Let's take a look at the first 4 and last 2 features as well as the label (SalePrice) from the first 4 examples:

```
print(train_data.iloc[0:4, [0, 1, 2, 3, -3, -2, -1]])
```

```
   Id  MSSubClass MSZoning  LotFrontage SaleType SaleCondition  SalePrice
0   1          60       RL         65.0       WD        Normal     208500
```

(continues on next page)

[83] http://pandas.pydata.org/pandas-docs/stable/

(continued from previous page)

```
1   2           20        RL          80.0        WD        Normal     181500
2   3           60        RL          68.0        WD        Normal     223500
3   4           70        RL          60.0        WD       Abnorml     140000
```

We can see that in each example, the first feature is the ID. This helps the model identify each training example. While this is convenient, it does not carry any information for prediction purposes. Hence we remove it from the dataset before feeding the data into the network.

```
all_features = pd.concat((train_data.iloc[:, 1:-1], test_data.iloc[:, 1:]))
```

4.10.3 Data Preprocessing

As stated above, we have a wide variety of data types. Before we feed it into a deep network, we need to perform some amount of processing. Let's start with the numerical features. We begin by replacing missing values with the mean. This is a reasonable strategy if features are missing at random. To adjust them to a common scale, we rescale them to zero mean and unit variance. This is accomplished as follows:

$$x \leftarrow \frac{x-\mu}{\sigma}. \tag{4.10.1}$$

To check that this transforms x to data with zero mean and unit variance simply calculate $E[(x-\mu)/\sigma] = (\mu-\mu)/\sigma = 0$. To check the variance we use $E[(x-\mu)^2] = \sigma^2$ and thus the transformed variable has unit variance. The reason for "normalizing" the data is that it brings all features to the same order of magnitude. After all, we do not know *a priori* which features are likely to be relevant.

```
numeric_features = all_features.dtypes[all_features.dtypes != 'object'].index
all_features[numeric_features] = all_features[numeric_features].apply(
    lambda x: (x - x.mean()) / (x.std()))
# After standardizing the data all means vanish, hence we can set missing
# values to 0
all_features[numeric_features] = all_features[numeric_features].fillna(0)
```

Next we deal with discrete values. This includes variables such as 'MSZoning'. We replace them by a one-hot encoding in the same manner as how we transformed multiclass classification data into a vector of 0 and 1. For instance, 'MSZoning' assumes the values 'RL' and 'RM'. They map into vectors $(1,0)$ and $(0,1)$ respectively. Pandas does this automatically for us.

```
# Dummy_na=True refers to a missing value being a legal eigenvalue, and
# creates an indicative feature for it
all_features = pd.get_dummies(all_features, dummy_na=True)
all_features.shape
```

```
(2919, 331)
```

You can see that this conversion increases the number of features from 79 to 331. Finally, via the `values` attribute, we can extract the NumPy format from the Pandas dataframe and convert it into MXNet's native `ndarray` representation for training.

```
n_train = train_data.shape[0]
train_features = np.array(all_features[:n_train].values, dtype=np.float32)
test_features = np.array(all_features[n_train:].values, dtype=np.float32)
train_labels = np.array(train_data.SalePrice.values,
                        dtype=np.float32).reshape(-1, 1)
```

4.10.4 Training

To get started we train a linear model with squared loss. Not surprisingly, our linear model will not lead to a competition winning submission but it provides a sanity check to see whether there is meaningful information in the data. If we cannot do better than random guessing here, then there might be a good chance that we have a data processing bug. And if things work, the linear model will serve as a baseline giving us some intuition about how close the simple model gets to the best reported models, giving us a sense of how much gain we should expect from fancier models.

```
loss = gluon.loss.L2Loss()

def get_net():
    net = nn.Sequential()
    net.add(nn.Dense(1))
    net.initialize()
    return net
```

With house prices, as with stock prices, we care about relative quantities more than absolute quantities. More concretely, we tend to care more about the relative error $\frac{y-\hat{y}}{y}$ than about the absolute error $y - \hat{y}$. For instance, if our prediction is off by USD 100,000 when estimating the price of a house in Rural Ohio, where the value of a typical house is 125,000 USD, then we are probably doing a horrible job. On the other hand, if we err by this amount in Los Altos Hills, California, this might represent a stunningly accurate prediction (their, the median house price exceeds 4 million USD).

One way to address this problem is to measure the discrepancy in the logarithm of the price estimates. In fact, this is also the official error metric used by the competition to measure the quality of submissions. After all, a small value δ of $\log y - \log \hat{y}$ translates into $e^{-\delta} \leq \frac{\hat{y}}{y} \leq e^{\delta}$. This leads to the following loss function:

$$L = \sqrt{\frac{1}{n}\sum_{i=1}^n\left(\log y_i -\log \hat{y}_i\right)^2}. \tag{4.10.2}$$

```
def log_rmse(net, features, labels):
    # To further stabilize the value when the logarithm is taken, set the
    # value less than 1 as 1
    clipped_preds = np.clip(net(features), 1, float('inf'))
    return np.sqrt(2 * loss(np.log(clipped_preds), np.log(labels)).mean())
```

Unlike in previous sections, our training functions here will rely on the Adam optimizer (a slight variant on SGD that we will describe in greater detail later). The main appeal of Adam vs vanilla SGD is that the Adam optimizer, despite doing no better (and sometimes worse) given unlimited resources for hyperparameter optimization, people tend to find that it is significantly less sensitive to the initial learning rate. This will be covered in further detail later on when we discuss the details in Chapter 11.

```
def train(net, train_features, train_labels, test_features, test_labels,
          num_epochs, learning_rate, weight_decay, batch_size):
    train_ls, test_ls = [], []
    train_iter = d2l.load_array((train_features, train_labels), batch_size)
    # The Adam optimization algorithm is used here
    trainer = gluon.Trainer(net.collect_params(), 'adam', {
        'learning_rate': learning_rate, 'wd': weight_decay})
    for epoch in range(num_epochs):
        for X, y in train_iter:
            with autograd.record():
                l = loss(net(X), y)
            l.backward()
            trainer.step(batch_size)
        train_ls.append(log_rmse(net, train_features, train_labels))
        if test_labels is not None:
            test_ls.append(log_rmse(net, test_features, test_labels))
    return train_ls, test_ls
```

4.10.5 k-Fold Cross-Validation

If you are reading in a linear fashion, you might recall that we introduced k-fold cross-validation in the section where we discussed how to deal with model section (Section 4.4). We will put this to good use to select the model design and to adjust the hyperparameters. We first need a function that returns the i^{th} fold of the data in a k-fold cross-validation procedure. It proceeds by slicing out the i^{th} segment as validation data and returning the rest as training data. Note that this is not the most efficient way of handling data and we would definitely do something much smarter if our dataset was considerably larger. But this added complexity might obfuscate our code unnecessarily so we can safely omit here owing to the simplicity of our problem.

```
def get_k_fold_data(k, i, X, y):
    assert k > 1
    fold_size = X.shape[0] // k
    X_train, y_train = None, None
    for j in range(k):
        idx = slice(j * fold_size, (j + 1) * fold_size)
        X_part, y_part = X[idx, :], y[idx]
        if j == i:
            X_valid, y_valid = X_part, y_part
        elif X_train is None:
            X_train, y_train = X_part, y_part
        else:
            X_train = np.concatenate((X_train, X_part), axis=0)
            y_train = np.concatenate((y_train, y_part), axis=0)
    return X_train, y_train, X_valid, y_valid
```

The training and verification error averages are returned when we train k times in the k-fold cross-validation.

```
def k_fold(k, X_train, y_train, num_epochs,
           learning_rate, weight_decay, batch_size):
    train_l_sum, valid_l_sum = 0, 0
    for i in range(k):
```

(continues on next page)

(continued from previous page)

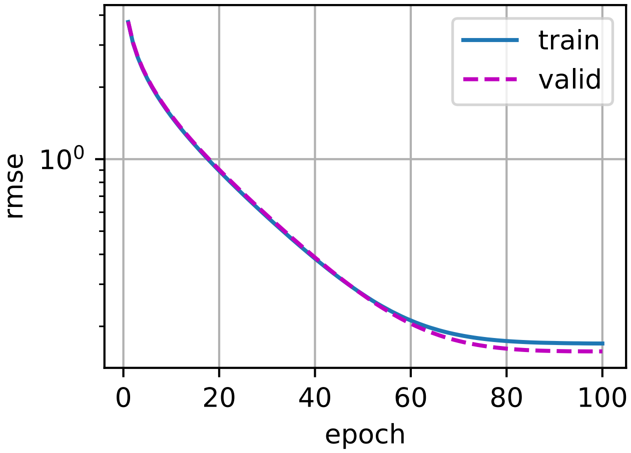
```
        data = get_k_fold_data(k, i, X_train, y_train)
        net = get_net()
        train_ls, valid_ls = train(net, *data, num_epochs, learning_rate,
                                   weight_decay, batch_size)
        train_l_sum += train_ls[-1]
        valid_l_sum += valid_ls[-1]
        if i == 0:
            d2l.plot(list(range(1, num_epochs+1)), [train_ls, valid_ls],
                     xlabel='epoch', ylabel='rmse',
                     legend=['train', 'valid'], yscale='log')
        print('fold %d, train rmse: %f, valid rmse: %f' % (
            i, train_ls[-1], valid_ls[-1]))
    return train_l_sum / k, valid_l_sum / k
```

4.10.6 Model Selection

In this example, we pick an un-tuned set of hyperparameters and leave it up to the reader to improve the model. Finding a good choice can take quite some time, depending on how many things one wants to optimize over. Within reason, the k-fold cross-validation approach is resilient against multiple testing. However, if we were to try out an unreasonably large number of options it might fail since we might just get lucky on the validation split with a particular set of hyperparameters.

```
k, num_epochs, lr, weight_decay, batch_size = 5, 100, 5, 0, 64
train_l, valid_l = k_fold(k, train_features, train_labels, num_epochs, lr,
                          weight_decay, batch_size)
print('%d-fold validation: avg train rmse: %f, avg valid rmse: %f'
      % (k, train_l, valid_l))
```

```
fold 0, train rmse: 0.169580, valid rmse: 0.157184
fold 1, train rmse: 0.162165, valid rmse: 0.190423
fold 2, train rmse: 0.163491, valid rmse: 0.167964
fold 3, train rmse: 0.167756, valid rmse: 0.154808
fold 4, train rmse: 0.163325, valid rmse: 0.183255
5-fold validation: avg train rmse: 0.165264, avg valid rmse: 0.170727
```



You will notice that sometimes the number of training errors for a set of hyper-parameters can be very low, while the number of errors for the K-fold cross-validation may be higher. This is an indicator that we are overfitting. Therefore, when we reduce the amount of training errors, we need to check whether the amount of errors in the k-fold cross-validation have also been reduced accordingly.

4.10.7 Predict and Submit

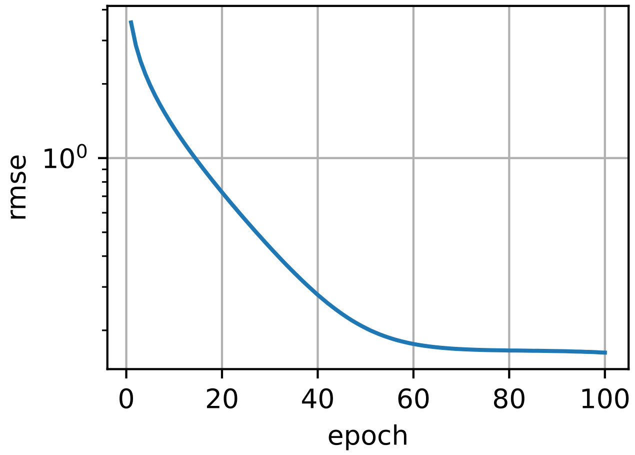
Now that we know what a good choice of hyperparameters should be, we might as well use all the data to train on it (rather than just $1 - 1/k$ of the data that is used in the cross-validation slices). The model that we obtain in this way can then be applied to the test set. Saving the estimates in a CSV file will simplify uploading the results to Kaggle.

```
def train_and_pred(train_features, test_feature, train_labels, test_data,
                   num_epochs, lr, weight_decay, batch_size):
    net = get_net()
    train_ls, _ = train(net, train_features, train_labels, None, None,
                        num_epochs, lr, weight_decay, batch_size)
    d2l.plot(np.arange(1, num_epochs + 1), [train_ls], xlabel='epoch',
             ylabel='rmse', yscale='log')
    print('train rmse %f' % train_ls[-1])
    # Apply the network to the test set
    preds = net(test_features).asnumpy()
    # Reformat it for export to Kaggle
    test_data['SalePrice'] = pd.Series(preds.reshape(1, -1)[0])
    submission = pd.concat([test_data['Id'], test_data['SalePrice']], axis=1)
    submission.to_csv('submission.csv', index=False)
```

Let's invoke our model. One nice sanity check is to see whether the predictions on the test set resemble those of the k-fold cross-validation process. If they do, it is time to upload them to Kaggle. The following code will generate a file called `submission.csv` (CSV is one of the file formats accepted by Kaggle):

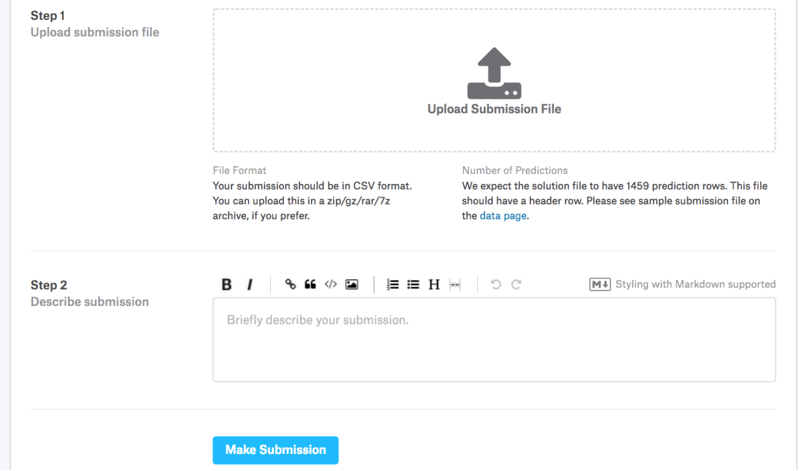
```
train_and_pred(train_features, test_features, train_labels, test_data,
               num_epochs, lr, weight_decay, batch_size)
```

```
train rmse 0.162410
```



Next, as demonstrated in Fig. 4.10.3, we can submit our predictions on Kaggle and see how they compare to the actual house prices (labels) on the test set. The steps are quite simple:

- Log in to the Kaggle website and visit the House Price Prediction Competition page.
- Click the "Submit Predictions" or "Late Submission" button (as of this writing, the button is located on the right).
- Click the "Upload Submission File" button in the dashed box at the bottom of the page and select the prediction file you wish to upload.
- Click the "Make Submission" button at the bottom of the page to view your results.



Step 1
Upload submission file

File Format
Your submission should be in CSV format. You can upload this in a zip/gz/rar/7z archive, if you prefer.

Number of Predictions
We expect the solution file to have 1459 prediction rows. This file should have a header row. Please see sample submission file on the data page.

Step 2
Describe submission

Briefly describe your submission.

Fig. 4.10.3: Submitting data to Kaggle

Summary

- Real data often contains a mix of different data types and needs to be preprocessed.
- Rescaling real-valued data to zero mean and unit variance is a good default. So is replacing missing values with their mean.
- Transforming categorical variables into indicator variables allows us to treat them like vectors.
- We can use k-fold cross validation to select the model and adjust the hyper-parameters.
- Logarithms are useful for relative loss.

Exercises

1. Submit your predictions for this tutorial to Kaggle. How good are your predictions?
2. Can you improve your model by minimizing the log-price directly? What happens if you try to predict the log price rather than the price?
3. Is it always a good idea to replace missing values by their mean? Hint: can you construct a situation where the values are not missing at random?
4. Find a better representation to deal with missing values. Hint: what happens if you add an indicator variable?

5. Improve the score on Kaggle by tuning the hyperparameters through k-fold cross-validation.
6. Improve the score by improving the model (layers, regularization, dropout).
7. What happens if we do not standardize the continuous numerical features like we have done in this section?

5 | Deep Learning Computation

Alongside giant datasets and powerful hardware, great software tools have played an indispensable role in the rapid progress of deep learning. Starting with the pathbreaking Theano library released in 2007, flexible open-source tools have enabled researchers to rapidly prototype models avoiding repetitive work when recycling standard components while still maintaining the ability to make low-level modifications. Over time, deep learning's libraries have evolved to offer increasingly coarse abstractions. Just as semiconductor designers went from specifying transistors to logical circuits to writing code, neural networks researchers have moved from thinking about the behavior of individual artificial neurons to conceiving of networks in terms of whole layers, and now often design architectures with far coarser *blocks* in mind.

So far, we have introduced some basic machine learning concepts, ramping up to fully-functional deep learning models. In the last chapter, we implemented each component of a multilayer perceptron from scratch and even showed how to leverage MXNet's Gluon library to roll out the same models effortlessly. To get you that far that fast, we *called upon* the libraries, but skipped over more advanced details about *how they work*. In this chapter, we will peel back the curtain, digging deeper into the key components of deep learning computation, namely model construction, parameter access and initialization, designing custom layers and blocks, reading and writing models to disk, and leveraging GPUs to achieve dramatic speedups. These insights will move you from *end user* to *power user*, giving you the tools needed to combine the reap the benefits of a mature deep learning library, while retaining the flexibility to implement more complex models, including those you invent yourself! While this chapter does not introduce any new models or datasets, the advanced modeling chapters that follow rely heavily on these techniques.

5.1 Layers and Blocks

When we first started talking about neural networks, we introduced linear models with a single output. Here, the entire model consists of just a single neuron. By itself, a single neuron takes some set of inputs, generates a corresponding (*scalar*) output, and has a set of associated parameters that can be updated to optimize some objective function of interest. Then, once we started thinking about networks with multiple outputs, we leveraged vectorized arithmetic, we showed how we could use linear algebra to efficiently express an entire *layer* of neurons. Layers too expect some inputs, generate corresponding outputs, and are described by a set of tunable parameters.

When we worked through softmax regression, a single *layer* was itself *the model*. However, when we subsequently introduced multilayer perceptrons, we developed models consisting of multiple layers. One interesting property of multilayer neural networks is that the *entire model* and its *constituent layers* share the same basic structure. The model takes the true inputs (as stated in the problem formulation), outputs predictions of the true outputs, and possesses parameters (the combined set of all parameters from all layers) Likewise any individual constituent layer in a multilayer perceptron ingests inputs (supplied by the previous layer) generates outputs (which form the inputs to the subsequent layer), and possesses a set of tunable parameters tht are updated with respect to the ultimate objective (using the signal that flows backwards through the subsequent layer).

While you might think that neurons, layers, and models give us enough abstractions to go about our business, it turns out that we will often want to express our model in terms of a components that are large than an indivudal layer. For example, when designing models, like ResNet-152, which possess hundreds (152, thus the name) of layers, implementing the network one layer at a time can grow tedious. Moreover, this concern is not just hypothetical—such deep networks dominate numerous application areas, especially when training data is abundant. For example the ResNet architecture mentioned above won the 2015 ImageNet and COCO computer vision competitions for both recognition and detection (He et al., 2016a). Deep networks with many layers arranged into components with various repeating patterns are now ubiquitous in other domains including natural language processing and speech.

To facilitate the implementation of networks consisting of components of arbitrary complexity, we introduce a new flexible concept: a neural network *block*. A block could describe a single neuron, a high-dimensional layer, or an arbitrarily-complex component consisting of multiple layers. From a software development, a `Block` is a class. Any subclass of `Block` must define a method called `forward` that transforms its input into output, and must store any necessary parameters. Note that some Blocks do not require any parameters at all! Finally a `Block` must possess a `backward` method, for purposes of calculating gradients. Fortunately, due to some behind-the-scenes magic supplied by the autograd `autograd` package (introduced in Chapter 2) when defining our own `Block` typically requires only that we worry about parameters and the `forward` function.

One benefit of working with the `Block` abstraction is that they can be combined into larger artifacts, often recursively, e.g., as illustrated in Fig. 5.1.1.

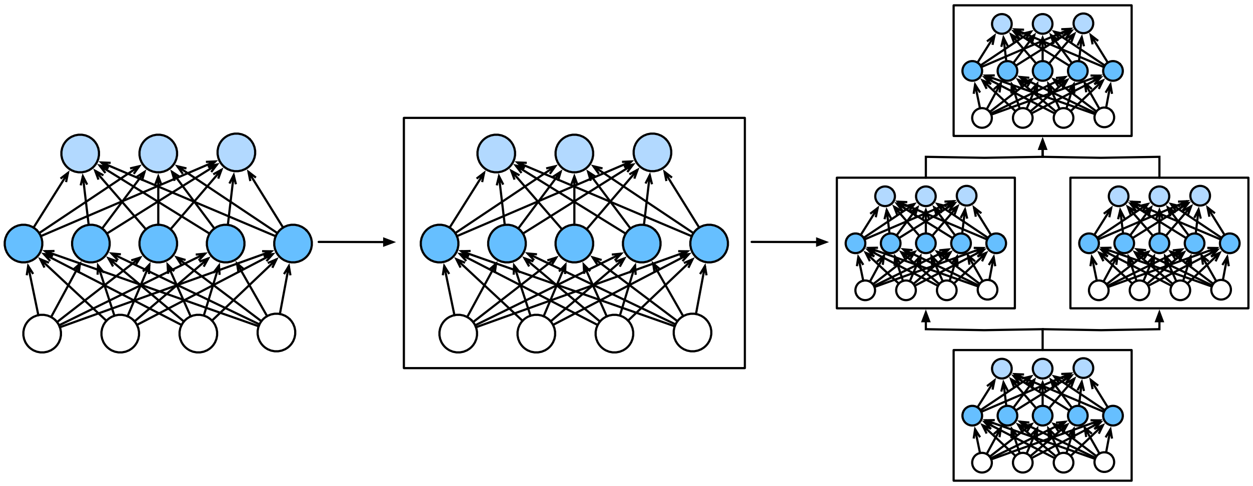


Fig. 5.1.1: Multiple layers are combined into blocks

By defining code to generate Blocks of arbitrary complexity on demand, we can write surprisingly compact code and still implement complex neural networks.

To begin, we revisit the Blocks that played a role in our implementation of the multilayer perceptron (Section 4.3). The following code generates a network with one fully-connected hidden layer containing 256 units followed by a ReLU activation, and then another fully-connected layer consisting of 10 units (with no activation function). Because there are no more layers, this last 10-unit layer is regarded as the *output layer* and its outputs are also the model's output.

```
from mxnet import np, npx
from mxnet.gluon import nn
npx.set_np()

x = np.random.uniform(size=(2, 20))

net = nn.Sequential()
net.add(nn.Dense(256, activation='relu'))
net.add(nn.Dense(10))
net.initialize()
net(x)
```

```
array([[ 0.06240272, -0.03268593,  0.02582653,  0.02254182, -0.03728798,
        -0.04253786,  0.00540613, -0.01364186, -0.09915452, -0.02272738],
       [ 0.02816677, -0.03341204,  0.03565666,  0.02506382, -0.04136416,
        -0.04941845,  0.01738528,  0.01081961, -0.09932579, -0.01176298]])
```

In this example, as in previous chapters, our model consists of an object returned by the `nn.Sequential` constructor. After instantiating a `nn.Sequential` and storing the `net` variable, we repeatedly called its `add` method, appending layers in the order that they should be executed. We suspect that you might have already understood *more or less* what was going on here the first time you saw this code. You may even have understood it well enough to modify the code and design your own networks. However, the details regarding what exactly happens inside `nn.Sequential` have remained mysterious so far.

In short, `nn.Sequential` just defines a special kind of Block. Specifically, an `nn.Sequential` maintains a list of constituent `Blocks`, stored in a particular order. You might think of `nnSequential` as your first meta-Block. The `add` method simply facilitates the addition of each successive `Block` to the list. Note that each our layers are instances of the `Dense` class which is itself a subclass of `Block`. The `forward` function is also remarkably simple: it chains each Block in the list together, passing the output of each as the input to the next.

Note that until now, we have been invoking our models via the construction `net(X)` to obtain their outputs. This is actually just shorthand for `net.forward(X)`, a slick Python trick achieved via the Block class's `__call__` function.

Before we dive in to implementing our own custom `Block`, we briefly summarize the basic functionality that each `Block` must perform the following duties:

1. Ingest input data as arguments to its `forward` function.
2. Generate an output via the value returned by its `forward` function. Note that the output may have a different shape from the input. For example, the first Dense layer in our model above ingests an input of arbitrary dimension but returns an output of dimension 256.
3. Calculate the gradient of its output with respect to its input, which can be accessed via its `backward` method. Typically this happens automatically.
4. Store and provide access to those parameters necessary to execute the `forward` computation.
5. Initialize these parameters as needed.

5.1.1 A Custom Block

Perhaps the easiest way to develop intuition about how nn.Block works is to just dive right in and implement one ourselves. In the following snippet, instead of relying on nn.Sequential, we just code up a Block from scratch that implements a multilayer perceptron with one hidden layer, 256 hidden nodes, and 10 outputs.

Our MLP class below inherits the Block class. While we rely on some predefined methods in the parent class, we need to supply our own __init__ and forward functions to uniquely define the behavior of our model.

```
from mxnet.gluon import nn

class MLP(nn.Block):
    # Declare a layer with model parameters. Here, we declare two fully
    # connected layers
    def __init__(self, **kwargs):
        # Call the constructor of the MLP parent class Block to perform the
        # necessary initialization. In this way, other function parameters can
        # also be specified when constructing an instance, such as the model
        # parameter, params, described in the following sections
        super(MLP, self).__init__(**kwargs)
        self.hidden = nn.Dense(256, activation='relu')  # Hidden layer
        self.output = nn.Dense(10)  # Output layer

    # Define the forward computation of the model, that is, how to return the
    # required model output based on the input x
    def forward(self, x):
        return self.output(self.hidden(x))
```

This code may be easiest to understand by working backwards from forward. Note that the forward method takes as input x. The forward method first evaluates self.hidden(x) to produce the hidden representation, passing this output as the input to the output layer self.output(...).

The constituent layers of each MLP must be instance-level variables. After all, if we instantiated two such models net1 and net2 and trained them on different data, we would expect them to them to represent two different learned models.

The __init__ method is the most natural place to instantiate the layers that we subsequently invoke on each call to the forward method. Note that before getting on with the interesting parts, our customized __init__ method must invoke the parent class's init method: super(MLP, self).__init__(**kwargs) to save us from reimplementing boilerplate code applicable to most Blocks. Then, all that is left is to instantiate our two Dense layers, assigning them to self.hidden and self.output, respectively. Again note that when dealing with standard functionality like this, we do not have to worry about backpropagation, since the backward method is generated for us automatically. The same goes for the initialize method. Let's try this out:

```
net = MLP()
net.initialize()
net(x)
```

```
array([[-0.03989594, -0.1041471 ,  0.06799038,  0.05245074,  0.02526059,
        -0.00640342,  0.04182098, -0.01665319, -0.02067346, -0.07863817],
```

(continues on next page)

(continued from previous page)

```
        [-0.03612847, -0.07210436,  0.09159479,  0.07890771,  0.02494172,
         -0.01028665,  0.01732428, -0.02843242,  0.03772651, -0.06671704]])
```

As we argued earlier, the primary virtue of the Block abstraction is its versatility. We can subclass Block to create layers (such as the Dense class provided by Gluon), entire models (such as the MLP class implemented above), or various components of intermediate complexity, a pattern that we will lean on heavily throughout the next chapters on convolutinoal neural networks.

5.1.2 The Sequential Block

As we described earlier, the Sequential class itself is also just a subclass of Block, designed specifically for daisy-chaining other Blocks together. All we need to do to implement our own MySequential block is to define a few convenience functions: 1. An add method for appending Blocks one by one to a list. 2. A forward method to pass inputs through the chain of Blocks (in the order of addition).

The following MySequential class delivers the same functionality as Gluon's default Sequential class:

```
class MySequential(nn.Block):
    def __init__(self, **kwargs):
        super(MySequential, self).__init__(**kwargs)

    def add(self, block):
        # Here, block is an instance of a Block subclass, and we assume it has
        # a unique name. We save it in the member variable _children of the
        # Block class, and its type is OrderedDict. When the MySequential
        # instance calls the initialize function, the system automatically
        # initializes all members of _children
        self._children[block.name] = block

    def forward(self, x):
        # OrderedDict guarantees that members will be traversed in the order
        # they were added
        for block in self._children.values():
            x = block(x)
        return x
```

At its core is the add method. It adds any block to the ordered dictionary of children. These are then executed in sequence when forward propagation is invoked. Let's see what the MLP looks like now.

```
net = MySequential()
net.add(nn.Dense(256, activation='relu'))
net.add(nn.Dense(10))
net.initialize()
net(x)
```

```
array([[-0.07645682, -0.01130233,  0.04952145, -0.04651389, -0.04131573,
        -0.05884133, -0.0621381 ,  0.01311472, -0.01379425, -0.02514282],
```

(continues on next page)

(continued from previous page)

```
       [-0.05124625,  0.00711231, -0.00155935, -0.07555379, -0.06675334,
        -0.01762914,  0.00589084,  0.01447191, -0.04330775,  0.03317726]])
```

Indeed, it can be observed that the use of the MySequential class is no different from the use of the Sequential class described in Section 4.3.

5.1.3 Blocks with Code

Although the Sequential class can make model construction easier, and you do not need to define the forward method, directly inheriting the Block class can greatly expand the flexibility of model construction. In particular, we will use Python's control flow within the forward method. While we are at it, we need to introduce another concept, that of the *constant* parameter. These are parameters that are not used when invoking backprop. This sounds very abstract but here's what is really going on. Assume that we have some function

$$f(\mathbf{x}, \mathbf{w}) = 3 \cdot \mathbf{w}^\top \mathbf{x}. \tag{5.1.1}$$

In this case 3 is a constant parameter. We could change 3 to something else, say c via

$$f(\mathbf{x}, \mathbf{w}) = c \cdot \mathbf{w}^\top \mathbf{x}. \tag{5.1.2}$$

Nothing has really changed, except that we can adjust the value of c. It is still a constant as far as $\mathbf{w}$ and $\mathbf{x}$ are concerned. However, since Gluon does not know about this beforehand, it is worth while to give it a hand (this makes the code go faster, too, since we are not sending the Gluon engine on a wild goose chase after a parameter that does not change). get_constant is the method that can be used to accomplish this. Let's see what this looks like in practice.

```
class FancyMLP(nn.Block):
    def __init__(self, **kwargs):
        super(FancyMLP, self).__init__(**kwargs)
        # Random weight parameters created with the get_constant are not
        # iterated during training (i.e., constant parameters)
        self.rand_weight = self.params.get_constant(
            'rand_weight', np.random.uniform(size=(20, 20)))
        self.dense = nn.Dense(20, activation='relu')

    def forward(self, x):
        x = self.dense(x)
        # Use the constant parameters created, as well as the relu
        # and dot functions
        x = npx.relu(np.dot(x, self.rand_weight.data()) + 1)
        # Reuse the fully connected layer. This is equivalent to sharing
        # parameters with two fully connected layers
        x = self.dense(x)
        # Here in Control flow, we need to call asscalar to return the scalar
        # for comparison
        while np.abs(x).sum() > 1:
            x /= 2
        if np.abs(x).sum() < 0.8:
            x *= 10
        return x.sum()
```

In this FancyMLP model, we used constant weight Rand_weight (note that it is not a model parameter), performed a matrix multiplication operation (np.dot<), and reused the *same* Dense layer. Note that this is very different from using two dense layers with different sets of parameters. Instead, we used the same network twice. Quite often in deep networks one also says that the parameters are *tied* when one wants to express that multiple parts of a network share the same parameters. Let's see what happens if we construct it and feed data through it.

```
net = FancyMLP()
net.initialize()
net(x)
```

```
array(5.2637568)
```

There is no reason why we couldn't mix and match these ways of build a network. Obviously the example below resembles more a chimera, or less charitably, a Rube Goldberg Machine[85]. That said, it combines examples for building a block from individual blocks, which in turn, may be blocks themselves. Furthermore, we can even combine multiple strategies inside the same forward function. To demonstrate this, here's the network.

```
class NestMLP(nn.Block):
    def __init__(self, **kwargs):
        super(NestMLP, self).__init__(**kwargs)
        self.net = nn.Sequential()
        self.net.add(nn.Dense(64, activation='relu'),
                     nn.Dense(32, activation='relu'))
        self.dense = nn.Dense(16, activation='relu')

    def forward(self, x):
        return self.dense(self.net(x))

chimera = nn.Sequential()
chimera.add(NestMLP(), nn.Dense(20), FancyMLP())

chimera.initialize()
chimera(x)
```

```
array(0.97720534)
```

5.1.4 Compilation

The avid reader is probably starting to worry about the efficiency of this. After all, we have lots of dictionary lookups, code execution, and lots of other Pythonic things going on in what is supposed to be a high performance deep learning library. The problems of Python's Global Interpreter Lock[86] are well known. In the context of deep learning it means that we have a super fast GPU (or multiple of them) which might have to wait until a puny single CPU core running Python gets a chance to tell it what to do next. This is clearly awful and there are many ways around it. The best way to speed up Python is by avoiding it altogether.

Gluon does this by allowing for Hybridization (Section 12.1). In it, the Python interpreter executes

[85] https://en.wikipedia.org/wiki/Rube_Goldberg_machine
[86] https://wiki.python.org/moin/GlobalInterpreterLock

the block the first time it is invoked. The Gluon runtime records what is happening and the next time around it short circuits any calls to Python. This can accelerate things considerably in some cases but care needs to be taken with control flow. We suggest that the interested reader skip forward to the section covering hybridization and compilation after finishing the current chapter.

Summary

- Layers are blocks
- Many layers can be a block
- Many blocks can be a block
- Code can be a block
- Blocks take are of a lot of housekeeping, such as parameter initialization, backprop and related issues.
- Sequential concatenations of layers and blocks are handled by the eponymous `Sequential` block.

Exercises

1. What kind of error message will you get when calling an `__init__` method whose parent class not in the `__init__` function of the parent class?
2. What kinds of problems will occur if you remove the `asscalar` function in the `FancyMLP` class?
3. What kinds of problems will occur if you change `self.net` defined by the Sequential instance in the `NestMLP` class to `self.net = [nn.Dense(64, activation='relu'), nn. Dense(32, activation='relu')]`?
4. Implement a block that takes two blocks as an argument, say `net1` and `net2` and returns the concatenated output of both networks in the forward pass (this is also called a parallel block).
5. Assume that you want to concatenate multiple instances of the same network. Implement a factory function that generates multiple instances of the same block and build a larger network from it.

5.2 Parameter Management

The ultimate goal of training deep networks is to find good parameter values for a given architecture. When everything is standard, the `nn.Sequential` class is a perfectly good tool for it. However, very few models are entirely standard and most scientists want to build things that are novel. This section shows how to manipulate parameters. In particular we will cover the following aspects:

- Accessing parameters for debugging, diagnostics, to visualize them or to save them is the first step to understanding how to work with custom models.

- Second, we want to set them in specific ways, e.g., for initialization purposes. We discuss the structure of parameter initializers.
- Last, we show how this knowledge can be put to good use by building networks that share some parameters.

As always, we start from our trusty Multilayer Perceptron with a hidden layer. This will serve as our choice for demonstrating the various features.

```
from mxnet import init, np, npx
from mxnet.gluon import nn
npx.set_np()

net = nn.Sequential()
net.add(nn.Dense(256, activation='relu'))
net.add(nn.Dense(10))
net.initialize()  # Use the default initialization method

x = np.random.uniform(size=(2, 20))
net(x)  # Forward computation
```

```
array([[ 0.06240272, -0.03268593,  0.02582653,  0.02254182, -0.03728798,
        -0.04253786,  0.00540613, -0.01364186, -0.09915452, -0.02272738],
       [ 0.02816677, -0.03341204,  0.03565666,  0.02506382, -0.04136416,
        -0.04941845,  0.01738528,  0.01081961, -0.09932579, -0.01176298]])
```

5.2.1 Parameter Access

In the case of a Sequential class we can access the parameters with ease, simply by indexing each of the layers in the network. The params variable then contains the required data. Let's try this out in practice by inspecting the parameters of the first layer.

```
print(net[0].params)
print(net[1].params)
```

```
dense0_ (
  Parameter dense0_weight (shape=(256, 20), dtype=float32)
  Parameter dense0_bias (shape=(256,), dtype=float32)
)
dense1_ (
  Parameter dense1_weight (shape=(10, 256), dtype=float32)
  Parameter dense1_bias (shape=(10,), dtype=float32)
)
```

The output tells us a number of things. First, the layer consists of two sets of parameters: dense0_weight and dense0_bias, as we would expect. They are both single precision and they have the necessary shapes that we would expect from the first layer, given that the input dimension is 20 and the output dimension 256. In particular the names of the parameters are very useful since they allow us to identify parameters *uniquely* even in a network of hundreds of layers and with nontrivial structure. The second layer is structured accordingly.

Targeted Parameters

In order to do something useful with the parameters we need to access them, though. There are several ways to do this, ranging from simple to general. Let's look at some of them.

```
print(net[1].bias)
print(net[1].bias.data())
```

```
Parameter dense1_bias (shape=(10,), dtype=float32)
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
```

The first returns the bias of the second layer. Since this is an object containing data, gradients, and additional information, we need to request the data explicitly. Note that the bias is all 0 since we initialized the bias to contain all zeros. Note that we can also access the parameters by name, such as `dense0_weight`. This is possible since each layer comes with its own parameter dictionary that can be accessed directly. Both methods are entirely equivalent but the first method leads to much more readable code.

```
print(net[0].params['dense0_weight'])
print(net[0].params['dense0_weight'].data())
```

```
Parameter dense0_weight (shape=(256, 20), dtype=float32)
[[ 0.06700657 -0.00369488  0.0418822  ... -0.05517294 -0.01194733
  -0.00369594]
 [-0.03296221 -0.04391347  0.03839272 ...  0.05636378  0.02545484
  -0.007007  ]
 [-0.0196689   0.01582889 -0.00881553 ...  0.01509629 -0.01908049
  -0.02449339]
 ...
 [-0.02055008 -0.02618652  0.06762936 ... -0.02315108 -0.06794678
  -0.04618235]
 [ 0.02802853  0.06672969  0.05018687 ... -0.02206502 -0.01315478
  -0.03791244]
 [-0.00638592  0.00914261  0.06667828 ... -0.00800052  0.03406764
  -0.03954004]]
```

Note that the weights are nonzero. This is by design since they were randomly initialized when we constructed the network. `data` is not the only function that we can invoke. For instance, we can compute the gradient with respect to the parameters. It has the same shape as the weight. However, since we did not invoke backpropagation yet, the values are all 0.

```
net[0].weight.grad()
```

```
array([[0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       ...,
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.]])
```

All Parameters at Once

Accessing parameters as described above can be a bit tedious, in particular if we have more complex blocks, or blocks of blocks (or even blocks of blocks of blocks), since we need to walk through the entire tree in reverse order to how the blocks were constructed. To avoid this, blocks come with a method collect_params which grabs all parameters of a network in one dictionary such that we can traverse it with ease. It does so by iterating over all constituents of a block and calls collect_params on subblocks as needed. To see the difference consider the following:

```
# parameters only for the first layer
print(net[0].collect_params())
# parameters of the entire network
print(net.collect_params())
```

```
dense0_ (
  Parameter dense0_weight (shape=(256, 20), dtype=float32)
  Parameter dense0_bias (shape=(256,), dtype=float32)
)
sequential0_ (
  Parameter dense0_weight (shape=(256, 20), dtype=float32)
  Parameter dense0_bias (shape=(256,), dtype=float32)
  Parameter dense1_weight (shape=(10, 256), dtype=float32)
  Parameter dense1_bias (shape=(10,), dtype=float32)
)
```

This provides us with a third way of accessing the parameters of the network. If we wanted to get the value of the bias term of the second layer we could simply use this:

```
net.collect_params()['dense1_bias'].data()
```

```
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

Throughout the book we will see how various blocks name their subblocks (Sequential simply numbers them). This makes it very convenient to use regular expressions to filter out the required parameters.

```
print(net.collect_params('.*weight'))
print(net.collect_params('dense0.*'))
```

```
sequential0_ (
  Parameter dense0_weight (shape=(256, 20), dtype=float32)
  Parameter dense1_weight (shape=(10, 256), dtype=float32)
)
sequential0_ (
  Parameter dense0_weight (shape=(256, 20), dtype=float32)
  Parameter dense0_bias (shape=(256,), dtype=float32)
)
```

Rube Goldberg Striking Again

Let's see how the parameter naming conventions work if we nest multiple blocks inside each other. For that we first define a function that produces blocks (a block factory, so to speak) and then we combine these inside yet larger blocks.

```
def block1():
    net = nn.Sequential()
    net.add(nn.Dense(32, activation='relu'))
    net.add(nn.Dense(16, activation='relu'))
    return net

def block2():
    net = nn.Sequential()
    for i in range(4):
        net.add(block1())
    return net

rgnet = nn.Sequential()
rgnet.add(block2())
rgnet.add(nn.Dense(10))
rgnet.initialize()
rgnet(x)
```

```
array([[-4.1923025e-09,  1.9830502e-09,  8.9444063e-10,  6.2912990e-09,
        -3.3241778e-09,  5.4330038e-09,  1.6013515e-09, -3.7408681e-09,
         8.5468477e-09, -6.4805539e-09],
       [-3.7507064e-09,  1.4866974e-09,  6.8314709e-10,  5.6925784e-09,
        -2.6349172e-09,  4.8626667e-09,  1.4280275e-09, -3.4603027e-09,
         7.4127922e-09, -5.7896132e-09]])
```

Now that we are done designing the network, let's see how it is organized. `collect_params` provides us with this information, both in terms of naming and in terms of logical structure.

```
print(rgnet.collect_params)
print(rgnet.collect_params())
```

```
<bound method Block.collect_params of Sequential(
  (0): Sequential(
    (0): Sequential(
      (0): Dense(20 -> 32, Activation(relu))
      (1): Dense(32 -> 16, Activation(relu))
    )
    (1): Sequential(
      (0): Dense(16 -> 32, Activation(relu))
      (1): Dense(32 -> 16, Activation(relu))
    )
    (2): Sequential(
      (0): Dense(16 -> 32, Activation(relu))
      (1): Dense(32 -> 16, Activation(relu))
    )
    (3): Sequential(
      (0): Dense(16 -> 32, Activation(relu))
      (1): Dense(32 -> 16, Activation(relu))
```

(continues on next page)

(continued from previous page)

```
    )
  )
  (1): Dense(16 -> 10, linear)
)>
sequential1_ (
  Parameter dense2_weight (shape=(32, 20), dtype=float32)
  Parameter dense2_bias (shape=(32,), dtype=float32)
  Parameter dense3_weight (shape=(16, 32), dtype=float32)
  Parameter dense3_bias (shape=(16,), dtype=float32)
  Parameter dense4_weight (shape=(32, 16), dtype=float32)
  Parameter dense4_bias (shape=(32,), dtype=float32)
  Parameter dense5_weight (shape=(16, 32), dtype=float32)
  Parameter dense5_bias (shape=(16,), dtype=float32)
  Parameter dense6_weight (shape=(32, 16), dtype=float32)
  Parameter dense6_bias (shape=(32,), dtype=float32)
  Parameter dense7_weight (shape=(16, 32), dtype=float32)
  Parameter dense7_bias (shape=(16,), dtype=float32)
  Parameter dense8_weight (shape=(32, 16), dtype=float32)
  Parameter dense8_bias (shape=(32,), dtype=float32)
  Parameter dense9_weight (shape=(16, 32), dtype=float32)
  Parameter dense9_bias (shape=(16,), dtype=float32)
  Parameter dense10_weight (shape=(10, 16), dtype=float32)
  Parameter dense10_bias (shape=(10,), dtype=float32)
)
```

Since the layers are hierarchically generated, we can also access them accordingly. For instance, to access the first major block, within it the second subblock and then within it, in turn the bias of the first layer, we perform the following.

```
rgnet[0][1][0].bias.data()
```

```
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

5.2.2 Parameter Initialization

Now that we know how to access the parameters, let's look at how to initialize them properly. We discussed the need for initialization in Section 4.8. By default, MXNet initializes the weight matrices uniformly by drawing from $U[-0.07, 0.07]$ and the bias parameters are all set to 0. However, we often need to use other methods to initialize the weights. MXNet's `init` module provides a variety of preset initialization methods, but if we want something out of the ordinary, we need a bit of extra work.

Built-in Initialization

Let's begin with the built-in initializers. The code below initializes all parameters with Gaussian random variables.

```
# force_reinit ensures that the variables are initialized again, regardless of
# whether they were already initialized previously
net.initialize(init=init.Normal(sigma=0.01), force_reinit=True)
net[0].weight.data()[0]
```

```
array([-9.8788980e-03,  5.3957910e-03, -7.0842835e-03, -7.4317548e-03,
       -1.4880489e-02,  6.4959107e-03, -8.2659349e-03,  1.8743129e-02,
        1.6201857e-02,  1.4534278e-03,  2.2331164e-03,  1.5926110e-02,
       -1.2915777e-02, -8.8592555e-05, -1.7293986e-03, -7.2338698e-03,
        8.7698260e-03, -4.9947016e-03, -9.6906107e-03,  2.0079101e-03])
```

If we wanted to initialize all parameters to 1, we could do this simply by changing the initializer to `Constant(1)`.

```
net.initialize(init=init.Constant(1), force_reinit=True)
net[0].weight.data()[0]
```

```
array([1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
       1., 1., 1.])
```

If we want to initialize only a specific parameter in a different manner, we can simply set the initializer only for the appropriate subblock (or parameter) for that matter. For instance, below we initialize the second layer to a constant value of 42 and we use the `Xavier` initializer for the weights of the first layer.

```
net[1].initialize(init=init.Constant(42), force_reinit=True)
net[0].weight.initialize(init=init.Xavier(), force_reinit=True)
print(net[1].weight.data()[0, 0])
print(net[0].weight.data()[0])
```

```
42.0
[-0.06319056 -0.10960881  0.11757872 -0.07595599 -0.0849717   0.0851637
  0.08330765  0.04028694 -0.0305525   0.02012795 -0.03856885  0.1375024
  0.10155623 -0.05016676 -0.02575382 -0.14205234  0.14225402  0.02719662
 -0.0888046  -0.00962897]
```

Custom Initialization

Sometimes, the initialization methods we need are not provided in the `init` module. At this point, we can implement a subclass of the `Initializer` class so that we can use it like any other initialization method. Usually, we only need to implement the `_init_weight` function and modify the incoming `ndarray` according to the initial result. In the example below, we pick a decidedly bizarre and nontrivial distribution, just to prove the point. We draw the coefficients from the following distribution:

$$w \sim \begin{cases} U[5, 10] & \text{with probability } \frac{1}{4} \\ 0 & \text{with probability } \frac{1}{2} \\ U[-10, -5] & \text{with probability } \frac{1}{4} \end{cases} \tag{5.2.1}$$

```
class MyInit(init.Initializer):
    def _init_weight(self, name, data):
        print('Init', name, data.shape)
        data[:] = np.random.uniform(-10, 10, data.shape)
        data *= np.abs(data) >= 5

net.initialize(MyInit(), force_reinit=True)
net[0].weight.data()[0]
```

```
Init dense0_weight (256, 20)
Init dense1_weight (10, 256)
```

```
array([-5.172625 , -7.0209026,  5.1446533, -9.844563 ,  8.545956 ,
       -0.       ,  0.       , -0.       ,  5.107664 ,  9.658335 ,
        5.8564453,  7.4483128,  0.       ,  0.       , -0.       ,
        7.9034443,  0.       ,  5.4223766,  8.5655575,  5.1224785])
```

If even this functionality is insufficient, we can set parameters directly. Since `data()` returns an `ndarray` we can access it just like any other matrix. A note for advanced users: if you want to adjust parameters within an `autograd` scope you need to use `set_data` to avoid confusing the automatic differentiation mechanics.

```
net[0].weight.data()[:] += 1
net[0].weight.data()[0, 0] = 42
net[0].weight.data()[0]
```

```
array([42.       , -6.0209026,  6.1446533, -8.844563 ,  9.545956 ,
        1.       ,  1.       ,  1.       ,  6.107664 , 10.658335 ,
        6.8564453,  8.448313 ,  1.       ,  1.       ,  1.       ,
        8.903444 ,  1.       ,  6.4223766,  9.5655575,  6.1224785])
```

5.2.3 Tied Parameters

In some cases, we want to share model parameters across multiple layers. For instance when we want to find good word embeddings we may decide to use the same parameters both for encoding and decoding of words. We discussed one such case when we introduced Section 5.1. Let's see how to do this a bit more elegantly. In the following we allocate a dense layer and then use its parameters specifically to set those of another layer.

```
net = nn.Sequential()
# We need to give the shared layer a name such that we can reference its
# parameters
shared = nn.Dense(8, activation='relu')
```

(continues on next page)

(continued from previous page)

```
net.add(nn.Dense(8, activation='relu'),
        shared,
        nn.Dense(8, activation='relu', params=shared.params),
        nn.Dense(10))
net.initialize()

x = np.random.uniform(size=(2, 20))
net(x)

# Check whether the parameters are the same
print(net[1].weight.data()[0] == net[2].weight.data()[0])
net[1].weight.data()[0, 0] = 100
# Make sure that they are actually the same object rather than just having the
# same value
print(net[1].weight.data()[0] == net[2].weight.data()[0])
```

```
[ True  True  True  True  True  True  True  True]
[ True  True  True  True  True  True  True  True]
```

The above example shows that the parameters of the second and third layer are tied. They are identical rather than just being equal. That is, by changing one of the parameters the other one changes, too. What happens to the gradients is quite ingenious. Since the model parameters contain gradients, the gradients of the second hidden layer and the third hidden layer are accumulated in the `shared.params.grad( )` during backpropagation.

Summary

- We have several ways to access, initialize, and tie model parameters.
- We can use custom initialization.
- Gluon has a sophisticated mechanism for accessing parameters in a unique and hierarchical manner.

Exercises

1. Use the FancyMLP defined in Section 5.1 and access the parameters of the various layers.
2. Look at the MXNet documentation[88] and explore different initializers.
3. Try accessing the model parameters after `net.initialize()` and before `net(x)` to observe the shape of the model parameters. What changes? Why?
4. Construct a multilayer perceptron containing a shared parameter layer and train it. During the training process, observe the model parameters and gradients of each layer.
5. Why is sharing parameters a good idea?

[88] http://beta.mxnet.io/api/gluon-related/mxnet.initializer.html

5.3 Deferred Initialization

In the previous examples we played fast and loose with setting up our networks. In particular we did the following things that *shouldn't* work:

- We defined the network architecture with no regard to the input dimensionality.
- We added layers without regard to the output dimension of the previous layer.
- We even "initialized" these parameters without knowing how many parameters were to initialize.

All of those things sound impossible and indeed, they are. After all, there is no way MXNet (or any other framework for that matter) could predict what the input dimensionality of a network would be. Later on, when working with convolutional networks and images this problem will become even more pertinent, since the input dimensionality (i.e., the resolution of an image) will affect the dimensionality of subsequent layers at a long range. Hence, the ability to set parameters without the need to know at the time of writing the code what the dimensionality is can greatly simplify statistical modeling. In what follows, we will discuss how this works using initialization as an example. After all, we cannot initialize variables that we do not know exist.

5.3.1 Instantiating a Network

Let's see what happens when we instantiate a network. We start with our trusty MLP as before.

```
from mxnet import init, np, npx
from mxnet.gluon import nn
npx.set_np()

def getnet():
    net = nn.Sequential()
    net.add(nn.Dense(256, activation='relu'))
    net.add(nn.Dense(10))
    return net

net = getnet()
```

At this point the network does not really know yet what the dimensionalities of the various parameters should be. All one could tell at this point is that each layer needs weights and bias, albeit of unspecified dimensionality. If we try accessing the parameters, that is exactly what happens.

```
print(net.collect_params)
print(net.collect_params())
```

```
<bound method Block.collect_params of Sequential(
  (0): Dense(-1 -> 256, Activation(relu))
  (1): Dense(-1 -> 10, linear)
)>
sequential0_ (
  Parameter dense0_weight (shape=(256, -1), dtype=float32)
  Parameter dense0_bias (shape=(256,), dtype=float32)
  Parameter dense1_weight (shape=(10, -1), dtype=float32)
  Parameter dense1_bias (shape=(10,), dtype=float32)
)
```

In particular, trying to access `net[0].weight.data()` at this point would trigger a runtime error stating that the network needs initializing before it can do anything. Let's see whether anything changes after we initialize the parameters:

```
net.initialize()
net.collect_params()
```

```
sequential0_ (
  Parameter dense0_weight (shape=(256, -1), dtype=float32)
  Parameter dense0_bias (shape=(256,), dtype=float32)
  Parameter dense1_weight (shape=(10, -1), dtype=float32)
  Parameter dense1_bias (shape=(10,), dtype=float32)
)
```

As we can see, nothing really changed. Only once we provide the network with some data do we see a difference. Let's try it out.

```
x = np.random.uniform(size=(2, 20))
net(x)  # Forward computation

net.collect_params()
```

```
sequential0_ (
  Parameter dense0_weight (shape=(256, 20), dtype=float32)
  Parameter dense0_bias (shape=(256,), dtype=float32)
  Parameter dense1_weight (shape=(10, 256), dtype=float32)
  Parameter dense1_bias (shape=(10,), dtype=float32)
)
```

The main difference to before is that as soon as we knew the input dimensionality, $\mathbf{x} \in \mathbb{R}^{20}$ it was possible to define the weight matrix for the first layer, i.e., $\mathbf{W}_1 \in \mathbb{R}^{256 \times 20}$. With that out of the way, we can progress to the second layer, define its dimensionality to be 10×256 and so on through the computational graph and bind all the dimensions as they become available. Once this is known, we can proceed by initializing parameters. This is the solution to the three problems outlined above.

5.3.2 Deferred Initialization in Practice

Now that we know how it works in theory, let's see when the initialization is actually triggered. In order to do so, we mock up an initializer which does nothing but report a debug message stating when it was invoked and with which parameters.

```
class MyInit(init.Initializer):
    def _init_weight(self, name, data):
        print('Init', name, data.shape)
        # The actual initialization logic is omitted here

net = getnet()
net.initialize(init=MyInit())
```

Note that, although `MyInit` will print information about the model parameters when it is called, the above `initialize` function does not print any information after it has been executed. Therefore there is no real initialization parameter when calling the `initialize` function. Next, we define the input and perform a forward calculation.

```
x = np.random.uniform(size=(2, 20))
y = net(x)
```

```
Init dense2_weight (256, 20)
Init dense3_weight (10, 256)
```

At this time, information on the model parameters is printed. When performing a forward calculation based on the input `x`, the system can automatically infer the shape of the weight parameters of all layers based on the shape of the input. Once the system has created these parameters, it calls the `MyInit` instance to initialize them before proceeding to the forward calculation.

Of course, this initialization will only be called when completing the initial forward calculation. After that, we will not re-initialize when we run the forward calculation `net(x)`, so the output of the `MyInit` instance will not be generated again.

```
y = net(x)
```

As mentioned at the beginning of this section, deferred initialization can also cause confusion. Before the first forward calculation, we were unable to directly manipulate the model parameters, for example, we could not use the `data` and `set_data` functions to get and modify the parameters. Therefore, we often force initialization by sending a sample observation through the network.

5.3.3 Forced Initialization

Deferred initialization does not occur if the system knows the shape of all parameters when calling the `initialize` function. This can occur in two cases:

- We have already seen some data and we just want to reset the parameters.
- We specified all input and output dimensions of the network when defining it.

The first case works just fine, as illustrated below.

```
net.initialize(init=MyInit(), force_reinit=True)
```

```
Init dense2_weight (256, 20)
Init dense3_weight (10, 256)
```

The second case requires us to specify the remaining set of parameters when creating the layer. For instance, for dense layers we also need to specify the `in_units` so that initialization can occur immediately once `initialize` is called.

```
net = nn.Sequential()
net.add(nn.Dense(256, in_units=20, activation='relu'))
net.add(nn.Dense(10, in_units=256))

net.initialize(init=MyInit())
```

```
Init dense4_weight (256, 20)
Init dense5_weight (10, 256)
```

Summary

- Deferred initialization is a good thing. It allows Gluon to set many things automatically and it removes a great source of errors from defining novel network architectures.
- We can override this by specifying all implicitly defined variables.
- Initialization can be repeated (or forced) by setting the `force_reinit=True` flag.

Exercises

1. What happens if you specify only parts of the input dimensions. Do you still get immediate initialization?
2. What happens if you specify mismatching dimensions?
3. What would you need to do if you have input of varying dimensionality? Hint - look at parameter tying.

5.4 Custom Layers

One of the reasons for the success of deep learning can be found in the wide range of layers that can be used in a deep network. This allows for a tremendous degree of customization and adaptation. For instance, scientists have invented layers for images, text, pooling, loops, dynamic programming, even for computer programs. Sooner or later you will encounter a layer that does not exist yet in Gluon, or even better, you will eventually invent a new layer that works well for your problem at hand. This is when it is time to build a custom layer. This section shows you how.

5.4.1 Layers without Parameters

Since this is slightly intricate, we start with a custom layer (also known as Block) that does not have any inherent parameters. Our first step is very similar to when we introduced blocks in Section 5.1. The following `CenteredLayer` class constructs a layer that subtracts the mean from the input. We build it by inheriting from the Block class and implementing the `forward` method.

```
from mxnet import gluon, np, npx
from mxnet.gluon import nn
npx.set_np()

class CenteredLayer(nn.Block):
    def __init__(self, **kwargs):
        super(CenteredLayer, self).__init__(**kwargs)

    def forward(self, x):
        return x - x.mean()
```

To see how it works let's feed some data into the layer.

```
layer = CenteredLayer()
layer(np.array([1, 2, 3, 4, 5]))
```

```
array([-2., -1.,  0.,  1.,  2.])
```

We can also use it to construct more complex models.

```
net = nn.Sequential()
net.add(nn.Dense(128), CenteredLayer())
net.initialize()
```

Let's see whether the centering layer did its job. For that we send random data through the network and check whether the mean vanishes. Note that since we are dealing with floating point numbers, we are going to see a very small albeit typically nonzero number.

```
y = net(np.random.uniform(size=(4, 8)))
y.mean()
```

```
array(3.783498e-10)
```

5.4.2 Layers with Parameters

Now that we know how to define layers in principle, let's define layers with parameters. These can be adjusted through training. In order to simplify things for an avid deep learning researcher the `Parameter` class and the `ParameterDict` dictionary provide some basic housekeeping functionality. In particular, they govern access, initialization, sharing, saving and loading model parameters. For instance, this way we do not need to write custom serialization routines for each new custom layer.

For instance, we can use the member variable `params` of the `ParameterDict` type that comes with the Block class. It is a dictionary that maps string type parameter names to model parameters in the `Parameter` type. We can create a `Parameter` instance from `ParameterDict` via the `get` function.

```
params = gluon.ParameterDict()
params.get('param2', shape=(2, 3))
params
```

```
(
  Parameter param2 (shape=(2, 3), dtype=<class 'numpy.float32'>)
)
```

Let's use this to implement our own version of the dense layer. It has two parameters: bias and weight. To make it a bit nonstandard, we bake in the ReLU activation as default. Next, we implement a fully connected layer with both weight and bias parameters. It uses ReLU as an activation function, where `in_units` and `units` are the number of inputs and the number of outputs, respectively.

```
class MyDense(nn.Block):
    # units: the number of outputs in this layer; in_units: the number of
    # inputs in this layer
    def __init__(self, units, in_units, **kwargs):
        super(MyDense, self).__init__(**kwargs)
        self.weight = self.params.get('weight', shape=(in_units, units))
        self.bias = self.params.get('bias', shape=(units,))

    def forward(self, x):
        linear = np.dot(x, self.weight.data()) + self.bias.data()
        return npx.relu(linear)
```

Naming the parameters allows us to access them by name through dictionary lookup later. It is a good idea to give them instructive names. Next, we instantiate the `MyDense` class and access its model parameters.

```
dense = MyDense(units=3, in_units=5)
dense.params
```

```
mydense0_ (
  Parameter mydense0_weight (shape=(5, 3), dtype=<class 'numpy.float32'>)
  Parameter mydense0_bias (shape=(3,), dtype=<class 'numpy.float32'>)
)
```

We can directly carry out forward calculations using custom layers.

```
dense.initialize()
dense(np.random.uniform(size=(2, 5)))
```

```
array([[0.        , 0.01633355, 0.        ],
       [0.        , 0.01581812, 0.        ]])
```

We can also construct models using custom layers. Once we have that we can use it just like the built-in dense layer. The only exception is that in our case size inference is not automatic. Please consult the MXNet documentation[91] for details on how to do this.

```
net = nn.Sequential()
net.add(MyDense(8, in_units=64),
        MyDense(1, in_units=8))
net.initialize()
net(np.random.uniform(size=(2, 64)))
```

```
array([[0.06508517],
       [0.0615553 ]])
```

Summary

- We can design custom layers via the Block class. This is more powerful than defining a block factory, since it can be invoked in many contexts.
- Blocks can have local parameters.

Exercises

1. Design a layer that learns an affine transform of the data, i.e., it removes the mean and learns an additive parameter instead.
2. Design a layer that takes an input and computes a tensor reduction, i.e., it returns $y_k = \sum_{i,j} W_{ijk} x_i x_j$.
3. Design a layer that returns the leading half of the Fourier coefficients of the data. Hint: look up the `fft` function in MXNet.

[91] http://www.mxnet.io

5.5 File I/O

So far we discussed how to process data, how to build, train and test deep learning models. However, at some point we are likely happy with what we obtained and we want to save the results for later use and distribution. Likewise, when running a long training process it is best practice to save intermediate results (checkpointing) to ensure that we do not lose several days worth of computation when tripping over the power cord of our server. At the same time, we might want to load a pre-trained model (e.g., we might have word embeddings for English and use it for our fancy spam classifier). For all of these cases we need to load and store both individual weight vectors and entire models. This section addresses both issues.

5.5.1 Loading and Saving `ndarray`s

In its simplest form, we can directly use the `load` and `save` functions to store and read `ndarray`s separately. This works just as expected.

```
from mxnet import np, npx
from mxnet.gluon import nn
npx.set_np()

x = np.arange(4)
npx.save('x-file', x)
```

Then, we read the data from the stored file back into memory.

```
x2 = npx.load('x-file')
x2
```

```
[array([0., 1., 2., 3.])]
```

We can also store a list of `ndarray`s and read them back into memory.

```
y = np.zeros(4)
npx.save('x-files', [x, y])
x2, y2 = npx.load('x-files')
(x2, y2)
```

```
(array([0., 1., 2., 3.]), array([0., 0., 0., 0.]))
```

We can even write and read a dictionary that maps from a string to an `ndarray`. This is convenient, for instance when we want to read or write all the weights in a model.

```
mydict = {'x': x, 'y': y}
npx.save('mydict', mydict)
mydict2 = npx.load('mydict')
mydict2
```

```
{'x': array([0., 1., 2., 3.]), 'y': array([0., 0., 0., 0.])}
```

5.5.2 Gluon Model Parameters

Saving individual weight vectors (or other `ndarray` tensors) is useful but it gets very tedious if we want to save (and later load) an entire model. After all, we might have hundreds of parameter groups sprinkled throughout. Writing a script that collects all the terms and matches them to an architecture is quite some work. For this reason Gluon provides built-in functionality to load and save entire networks rather than just single weight vectors. An important detail to note is that this saves model *parameters* and not the entire model. I.e. if we have a 3 layer MLP we need to specify the *architecture* separately. The reason for this is that the models themselves can contain arbitrary code, hence they cannot be serialized quite so easily (there is a way to do this for compiled models: please refer to the MXNet documentation[93] for the technical details on it). The result is that in order to reinstate a model we need to generate the architecture in code and then load the parameters from disk. The deferred initialization (Section 5.3) is quite advantageous here since we can simply define a model without the need to put actual values in place. Let's start with our favorite MLP.

```
class MLP(nn.Block):
    def __init__(self, **kwargs):
        super(MLP, self).__init__(**kwargs)
        self.hidden = nn.Dense(256, activation='relu')
        self.output = nn.Dense(10)

    def forward(self, x):
        return self.output(self.hidden(x))

net = MLP()
net.initialize()
x = np.random.uniform(size=(2, 20))
y = net(x)
```

Next, we store the parameters of the model as a file with the name `mlp.params`.

```
net.save_parameters('mlp.params')
```

To check whether we are able to recover the model we instantiate a clone of the original MLP model. Unlike the random initialization of model parameters, here we read the parameters stored in the file directly.

```
clone = MLP()
clone.load_parameters('mlp.params')
```

Since both instances have the same model parameters, the computation result of the same input `x` should be the same. Let's verify this.

```
yclone = clone(x)
yclone == y
```

```
array([[ True,  True,  True,  True,  True,  True,  True,  True,  True,
         True],
       [ True,  True,  True,  True,  True,  True,  True,  True,  True,
         True]])
```

[93] http://www.mxnet.io

Summary

- The save and load functions can be used to perform File I/O for ndarray objects.
- The load_parameters and save_parameters functions allow us to save entire sets of parameters for a network in Gluon.
- Saving the architecture has to be done in code rather than in parameters.

Exercises

1. Even if there is no need to deploy trained models to a different device, what are the practical benefits of storing model parameters?
2. Assume that we want to reuse only parts of a network to be incorporated into a network of a *different* architecture. How would you go about using, say the first two layers from a previous network in a new network.
3. How would you go about saving network architecture and parameters? What restrictions would you impose on the architecture?

5.6 GPUs

In the introduction to this book we discussed the rapid growth of computation over the past two decades. In a nutshell, GPU performance has increased by a factor of 1000 every decade since 2000. This offers great opportunity but it also suggests a significant need to provide such performance.

Decade	Dataset	Memory	Floating Point Calculations per Second
1970	100 (Iris)	1 KB	100 KF (Intel 8080)
1980	1 K (House prices in Boston)	100 KB	1 MF (Intel 80186)
1990	10 K (optical character recognition)	10 MB	10 MF (Intel 80486)
2000	10 M (web pages)	100 MB	1 GF (Intel Core)
2010	10 G (advertising)	1 GB	1 TF (NVIDIA C2050)
2020	1 T (social network)	100 GB	1 PF (NVIDIA DGX-2)

In this section we begin to discuss how to harness this compute performance for your research. First by using single GPUs and at a later point, how to use multiple GPUs and multiple servers (with multiple GPUs). You might have noticed that MXNet ndarray looks almost identical to NumPy. But there are a few crucial differences. One of the key features that differentiates MXNet from NumPy is its support for diverse hardware devices.

In MXNet, every array has a context. In fact, whenever we displayed an ndarray so far, it added a cryptic @cpu(0) notice to the output which remained unexplained so far. As we will discover, this just indicates that the computation is being executed on the CPU. Other contexts might be various GPUs. Things can get even hairier when we deploy jobs across multiple servers. By assigning arrays to contexts intelligently, we can minimize the time spent transferring data between devices. For example, when training neural networks on a server with a GPU, we typically prefer for the model's parameters to live on the GPU.

In short, for complex neural networks and large-scale data, using only CPUs for computation may be inefficient. In this section, we will discuss how to use a single NVIDIA GPU for calculations. First, make sure you have at least one NVIDIA GPU installed. Then, download CUDA[95] and follow the prompts to set the appropriate path. Once these preparations are complete, the `nvidia-smi` command can be used to view the graphics card information.

```
!nvidia-smi
```

```
Sat Dec 14 03:45:01 2019
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 418.67       Driver Version: 418.67       CUDA Version: 10.1     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  Tesla V100-SXM2...  Off  | 00000000:00:1B.0 Off |                    0 |
| N/A   34C    P0    50W / 300W |      0MiB / 16130MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   1  Tesla V100-SXM2...  Off  | 00000000:00:1C.0 Off |                    0 |
| N/A   34C    P0    51W / 300W |      0MiB / 16130MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   2  Tesla V100-SXM2...  Off  | 00000000:00:1D.0 Off |                    0 |
| N/A   36C    P0    54W / 300W |      0MiB / 16130MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   3  Tesla V100-SXM2...  Off  | 00000000:00:1E.0 Off |                    0 |
| N/A   34C    P0    53W / 300W |      0MiB / 16130MiB |      4%      Default |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID   Type   Process name                             Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

Next, we need to confirm that the GPU version of MXNet is installed. If a CPU version of MXNet is already installed, we need to uninstall it first. For example, use the `pip uninstall mxnet` command, then install the corresponding MXNet version according to the CUDA version. Assuming you have CUDA 9.0 installed, you can install the MXNet version that supports CUDA 9.0 by `pip install mxnet-cu90`. To run the programs in this section, you need at least two GPUs.

Note that this might be extravagant for most desktop computers but it is easily available in the cloud, e.g., by using the AWS EC2 multi-GPU instances. Almost all other sections do *not* require multiple GPUs. Instead, this is simply to illustrate how data flows between different devices.

[95] https://developer.nvidia.com/cuda-downloads

5.6.1 Computing Devices

MXNet can specify devices, such as CPUs and GPUs, for storage and calculation. By default, MXNet creates data in the main memory and then uses the CPU to calculate it. In MXNet, the CPU and GPU can be indicated by `cpu()` and `gpu()`. It should be noted that `cpu()` (or any integer in the parentheses) means all physical CPUs and memory. This means that MXNet's calculations will try to use all CPU cores. However, `gpu()` only represents one graphic card and the corresponding graphic memory. If there are multiple GPUs, we use `gpu(i)` to represent the i^{th} GPU (i starts from 0). Also, `gpu(0)` and `gpu()` are equivalent.

```
from mxnet import np, npx
from mxnet.gluon import nn
npx.set_np()

npx.cpu(), npx.gpu(), npx.gpu(1)
```

```
(cpu(0), gpu(0), gpu(1))
```

We can query the number of available GPUs through `num_gpus()`.

```
npx.num_gpus()
```

```
2
```

Now we define two convenient functions that allows us to run codes even if the requested GPUs do not exist.

```
# Saved in the d2l package for later use
def try_gpu(i=0):
    """Return gpu(i) if exists, otherwise return cpu()."""
    return npx.gpu(i) if npx.num_gpus() >= i + 1 else npx.cpu()

# Saved in the d2l package for later use
def try_all_gpus():
    """Return all available GPUs, or [cpu(),] if no GPU exists."""
    ctxes = [npx.gpu(i) for i in range(npx.num_gpus())]
    return ctxes if ctxes else [npx.cpu()]

try_gpu(), try_gpu(3), try_all_gpus()
```

```
(gpu(0), cpu(0), [gpu(0), gpu(1)])
```

5.6.2 ndarray and GPUs

By default, ndarray objects are created on the CPU. Therefore, we will see the @cpu(0) identifier each time we print an ndarray.

```
x = np.array([1, 2, 3])
x
```

```
array([1., 2., 3.])
```

We can use the context property of ndarray to view the device where the ndarray is located. It is important to note that whenever we want to operate on multiple terms they need to be in the same context. For instance, if we sum two variables, we need to make sure that both arguments are on the same device—otherwise MXNet would not know where to store the result or even how to decide where to perform the computation.

```
x.context
```

```
cpu(0)
```

Storage on the GPU

There are several ways to store an ndarray on the GPU. For example, we can specify a storage device with the ctx parameter when creating an ndarray. Next, we create the ndarray variable a on gpu(0). Notice that when printing a, the device information becomes @gpu(0). The ndarray created on a GPU only consumes the memory of this GPU. We can use the nvidia-smi command to view GPU memory usage. In general, we need to make sure we do not create data that exceeds the GPU memory limit.

```
x = np.ones((2, 3), ctx=try_gpu())
x
```

```
array([[1., 1., 1.],
       [1., 1., 1.]], ctx=gpu(0))
```

Assuming you have at least two GPUs, the following code will create a random array on gpu(1).

```
y = np.random.uniform(size=(2, 3), ctx=try_gpu(1))
y
```

```
array([[0.67478997, 0.07540122, 0.9956977 ],
       [0.09488854, 0.415456  , 0.11231736]], ctx=gpu(1))
```

Copying

If we want to compute $\mathbf{x} + \mathbf{y}$ we need to decide where to perform this operation. For instance, as shown in Fig. 5.6.1, we can transfer $\mathbf{x}$ to `gpu(1)` and perform the operation there. *Do not* simply add `x + y` since this will result in an exception. The runtime engine would not know what to do, it cannot find data on the same device and it fails.

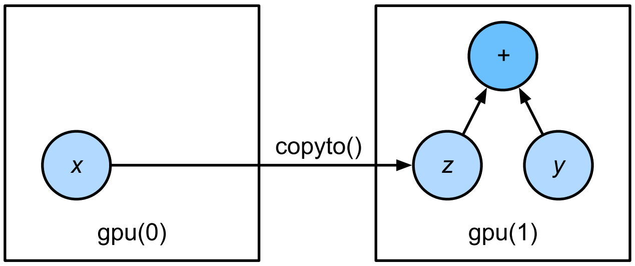


Fig. 5.6.1: Copyto copies arrays to the target device

`copyto` copies the data to another device such that we can add them. Since $\mathbf{y}$ lives on the second GPU we need to move $\mathbf{x}$ there before we can add the two.

```
z = x.copyto(try_gpu(1))
print(x)
print(z)
```

```
[[1. 1. 1.]
 [1. 1. 1.]] @gpu(0)
[[1. 1. 1.]
 [1. 1. 1.]] @gpu(1)
```

Now that the data is on the same GPU (both $\mathbf{z}$ and $\mathbf{y}$ are), we can add them up. In such cases MXNet places the result on the same device as its constituents. In our case that is `@gpu(1)`.

```
y + z
```

```
array([[1.6747899, 1.0754012, 1.9956977],
       [1.0948886, 1.415456 , 1.1123173]], ctx=gpu(1))
```

Imagine that your variable z already lives on your second GPU (gpu(1)). What happens if we call z.copyto(gpu(1))? It will make a copy and allocate new memory, even though that variable already lives on the desired device! There are times where depending on the environment our code is running in, two variables may already live on the same device. So we only want to make a copy if the variables currently lives on different contexts. In these cases, we can call `as_in_context()`. If the variable is already the specified context then this is a no-op. In fact, unless you specifically want to make a copy, `as_in_context()` is the method of choice.

```
z = x.as_in_context(try_gpu(1))
z
```

```
array([[1., 1., 1.],
       [1., 1., 1.]], ctx=gpu(1))
```

It is important to note that, if the context of the source variable and the target variable are consistent, then the as_in_context function causes the target variable and the source variable to share the memory of the source variable.

```
y.as_in_context(try_gpu(1)) is y
```

```
False
```

The copyto function always creates new memory for the target variable.

```
y.copyto(try_gpu(1)) is y
```

```
False
```

Side Notes

People use GPUs to do machine learning because they expect them to be fast. But transferring variables between contexts is slow. So we want you to be 100% certain that you want to do something slow before we let you do it. If MXNet just did the copy automatically without crashing then you might not realize that you had written some slow code.

Also, transferring data between devices (CPU, GPUs, other machines) is something that is *much slower* than computation. It also makes parallelization a lot more difficult, since we have to wait for data to be sent (or rather to be received) before we can proceed with more operations. This is why copy operations should be taken with great care. As a rule of thumb, many small operations are much worse than one big operation. Moreover, several operations at a time are much better than many single operations interspersed in the code (unless you know what you are doing). This is the case since such operations can block if one device has to wait for the other before it can do something else. It is a bit like ordering your coffee in a queue rather than pre-ordering it by phone and finding out that it is ready when you are.

Last, when we print ndarrays or convert ndarrays to the NumPy format, if the data is not in main memory, MXNet will copy it to the main memory first, resulting in additional transmission overhead. Even worse, it is now subject to the dreaded Global Interpreter Lock which makes everything wait for Python to complete.

5.6.3 Gluon and GPUs

Similarly, Gluon's model can specify devices through the ctx parameter during initialization. The following code initializes the model parameters on the GPU (we will see many more examples of how to run models on GPUs in the following, simply since they will become somewhat more compute intensive).

```
net = nn.Sequential()
net.add(nn.Dense(1))
net.initialize(ctx=try_gpu())
```

When the input is an ndarray on the GPU, Gluon will calculate the result on the same GPU.

```
net(x)
```

```
array([[0.04995865],
       [0.04995865]], ctx=gpu(0))
```

Let's confirm that the model parameters are stored on the same GPU.

```
net[0].weight.data()
```

```
array([[0.0068339 , 0.01299825, 0.0301265 ]], ctx=gpu(0))
```

In short, as long as all data and parameters are on the same device, we can learn models efficiently. In the following we will see several such examples.

Summary

- MXNet can specify devices for storage and calculation, such as CPU or GPU. By default, MXNet creates data in the main memory and then uses the CPU to calculate it.
- MXNet requires all input data for calculation to be *on the same device*, be it CPU or the same GPU.
- You can lose significant performance by moving data without care. A typical mistake is as follows: computing the loss for every minibatch on the GPU and reporting it back to the user on the command line (or logging it in a NumPy array) will trigger a global interpreter lock which stalls all GPUs. It is much better to allocate memory for logging inside the GPU and only move larger logs.

Exercises

1. Try a larger computation task, such as the multiplication of large matrices, and see the difference in speed between the CPU and GPU. What about a task with a small amount of calculations?
2. How should we read and write model parameters on the GPU?
3. Measure the time it takes to compute 1000 matrix-matrix multiplications of 100×100 matrices and log the matrix norm tr$MM^\top$ one result at a time vs. keeping a log on the GPU and transferring only the final result.
4. Measure how much time it takes to perform two matrix-matrix multiplications on two GPUs at the same time vs. in sequence on one GPU (hint: you should see almost linear scaling).

6 | Convolutional Neural Networks

In several of our previous examples, we have already come up against image data, which consist of pixels arranged in a 2D grid. Depending on whether we are looking at a black and white or color image, we might have either one or multiple numerical values corresponding to each pixel location. Until now, we have dealt with this rich structure in the least satisfying possible way. We simply threw away this spatial structure by flattening each image into a 1D vector, and fed it into a fully-connected network. These networks are invariant to the order of their inputs. We will get qualitatively identical results out of a multilayer perceptron whether we preserve the original order of our features or if we permute the columns of our design matrix before learning the parameters. Ideally, we would find a way to leverage our prior knowledge that nearby pixels are more related to each other.

In this chapter, we introduce convolutional neural networks (CNNs), a powerful family of neural networks that were designed for precisely this purpose. CNN-based network *architecures* now dominate the field of computer vision to such an extent that hardly anyone these days would develop a commercial application or enter a competition related to image recognition, object detection, or semantic segmentation, without basing their approach on them.

Modern 'convnets', as they are often called owe their design to inspirations from biology, group theory, and a healthy dose of experimental tinkering. In addition to their strong predictive performance, convolutional neural networks tend to be computationally efficient, both because they tend to require fewer parameters than dense architectures and also because convolutions are easy to parallelize across GPU cores. As a result, researchers have sought to apply convnets whenever possible, and increasingly they have emerged as credible competitors even on tasks with 1D sequence structure, such as audio, text, and time series analysis, where recurrent neural networks (introduced in the next chapter) are conventionally used. Some clever adaptations of CNNs have also brought them to bear on graph-structured data and in recommender systems.

First, we will walk through the basic operations that comprise the backbone of all modern convolutional networks. These include the convolutional layers themselves, nitty-gritty details including padding and stride, the pooling layers used to aggregate information across adjacent spatial regions, the use of multiple *channels* (also called *filters*) at each layer, and a careful discussion of the structure of modern architectures. We will conclude the chapter with a full working example of LeNet, the first convolutional network successfully deployed, long before the rise of modern deep learning. In the next chapter we will dive into full implementations of some of the recent popular neural networks whose designs are representative of most of the techniques commonly used to design modern convolutional neural networks.

6.1 From Dense Layers to Convolutions

The models that we have discussed so far are fine options if you are dealing with *tabular* data. By *tabular* we mean that the data consists of rows corresponding to examples and columns corresponding to features. With tabular data, we might anticipate that pattern we seek could require modeling interactions among the features, but do not assume anything a priori about which features are related to each other or in what way.

Sometimes we truly may not have any knowledge to guide the construction of more cleverly-organized architectures. In these cases, a multilayer perceptron is often the best that we can do. However, once we start dealing with high-dimensional perceptual data, these *structure-less* networks can grow unwieldy.

For instance, let's return to our running example of distinguishing cats from dogs. Say that we do a thorough job in data collection, collecting an annotated sets of high-quality 1-megapixel photographs. This means that the input into a network has *1 million dimensions*. Even an aggressive reduction to *1,000 hidden dimensions* would require a *dense* (fully-connected) layer to support 10^9 parameters. Unless we have an extremely large dataset (perhaps billions?), lots of GPUs, a talent for extreme distributed optimization, and an extraordinary amount of patience, learning the parameters of this network may turn out to be impossible.

A careful reader might object to this argument on the basis that 1 megapixel resolution may not be necessary. However, while you could get away with 100,000 pixels, we grossly underestimated the number of hidden nodes that it typically takes to learn good hidden representations of images. Learning a binary classifier with so many parameters might seem to require that we collect an enormous dataset, perhaps comparable to the number of dogs and cats on the planet. And yet both humans and computers are able to distinguish cats from dogs quite well, seemingly contradicting these conclusions. That is because images exhibit rich structure that is typically exploited by humans and machine learning models alike.

6.1.1 Invariances

Imagine that you want to detect an object in an image. It seems reasonable that whatever method we use to recognize objects should not be overly concerned with the precise *location* of the object shouldn't in the image. Ideally we could learn a system that would somehow exploit this knowledge. Pigs usually do not fly and planes usually do not swim. Nonetheless, we could still recognize a flying pig were one to appear. This ideas is taken to an extreme in the children's game 'Where's Waldo', an example is shown in Fig. 6.1.1. The game consists of a number of chaotic scenes bursting with activity and Waldo shows up somewhere in each (typically lurking in some unlikely location). The reader's goal is to locate him. Despite his characteristic outfit, this can be surprisingly difficult, due to the large number of confounders.

Fig. 6.1.1: Image via Walker Books

Back to images, the intuitions we have been discussing could be made more concrete yielding a few key principles for building neural networks for computer vision:

1. Our vision systems should, in some sense, respond similarly to the same object regardless of where it appears in the image (translation invariance).
2. Our visions systems should, in some sense, focus on local regions, without regard for what else is happening in the image at greater distances (locality).

Let's see how this translates into mathematics.

6.1.2 Constraining the MLP

To start off let's consider what an MLP would look like with $h \times w$ images as inputs (represented as matrices in math, and as 2D arrays in code), and hidden representations similarly organized as $h \times w$ matrices / 2D arrays. Let $x[i, j]$ and $h[i, j]$ denote pixel location (i, j) in an image and hidden representation, respectively. Consequently, to have each of the hw hidden nodes receive input from each of the hw inputs, we would switch from using weight matrices (as we did previously in MLPs) to representing our parameters as four-dimensional weight tensors.

We could formally express this dense layer as follows:

$$h[i, j] = u[i, j] + \sum_{k,l} W[i, j, k, l] \cdot x[k, l] = u[i, j] + \sum_{a,b} V[i, j, a, b] \cdot x[i + a, j + b]. \tag{6.1.1}$$

The switch from W to V is entirely cosmetic (for now) since there is a one-to-one correspondence between coefficients in both tensors. We simply re-index the subscripts (k, l) such that $k = i + a$ and $l = j + b$. In other words, we set $V[i, j, a, b] = W[i, j, i + a, j + b]$. The indices a, b run over both positive and negative offsets, covering the entire image. For any given location (i, j) in the hidden layer $h[i, j]$, we compute its value by summing over pixels in x, centered around (i, j) and weighted by $V[i, j, a, b]$.

Now let's invoke the first principle we established above: *translation invariance*. This implies that a shift in the inputs x should simply lead to a shift in the activations h. This is only possible if V

and u do not actually depend on (i, j), i.e., we have $V[i, j, a, b] = V[a, b]$ and u is a constant. As a result we can simplify the definition for h.

$$h[i, j] = u + \sum_{a,b} V[a, b] \cdot x[i + a, j + b]. \tag{6.1.2}$$

This is a convolution! We are effectively weighting pixels $(i + a, j + b)$ in the vicinity of (i, j) with coefficients $V[a, b]$ to obtain the value $h[i, j]$. Note that $V[a, b]$ needs many fewer coefficients than $V[i, j, a, b]$. For a 1 megapixel image it has at most 1 million coefficients. This is 1 million fewer parameters since it no longer depends on the location within the image. We have made significant progress!

Now let's invoke the second principle—*locality*. As motivated above, we believe that we shouldn't have to look very far away from (i, j) in order to glean relevant information to assess what is going on at $h[i, j]$. This means that outside some range $|a|, |b| > \Delta$, we should set $V[a, b] = 0$. Equivalently, we can rewrite $h[i, j]$ as

$$h[i, j] = u + \sum_{a=-\Delta}^{\Delta} \sum_{b=-\Delta}^{\Delta} V[a, b] \cdot x[i + a, j + b]. \tag{6.1.3}$$

This, in a nutshell is the convolutional layer. When the local region (also called a *receptive field*) is small, the difference as compared to a fully-connected network can be dramatic. While previously, we might have required billions of parameters to represent just a single layer in an image-processing network, we now typically need just a few hundred. The price that we pay for this drastic modification is that our features will be translation invariant and that our layer can only take local information into account. All learning depends on imposing inductive bias. When that bias agrees with reality, we get sample-efficient models that generalize well to unseen data. But of course, if those biases do not agree with reality, e.g., if images turned out not to be translation invariant, our models may not generalize well.

6.1.3 Convolutions

Let's briefly review why the above operation is called a *convolution*. In mathematics, the convolution between two functions, say $f, g : \mathbb{R}^d \to R$ is defined as

$$[f \circledast g](x) = \int_{\mathbb{R}^d} f(z)g(x - z)dz. \tag{6.1.4}$$

That is, we measure the overlap between f and g when both functions are shifted by x and "flipped". Whenever we have discrete objects, the integral turns into a sum. For instance, for vectors defined on ℓ_2, i.e., the set of square summable infinite dimensional vectors with index running over $\mathbb{Z}$ we obtain the following definition.

$$[f \circledast g](i) = \sum_{a} f(a)g(i - a). \tag{6.1.5}$$

For two-dimensional arrays, we have a corresponding sum with indices (i, j) for f and $(i-a, j-b)$ for g respectively. This looks similar to definition above, with one major difference. Rather than using $(i+a, j+b)$, we are using the difference instead. Note, though, that this distinction is mostly cosmetic since we can always match the notation by using $\tilde{V}[a, b] = V[-a, -b]$ to obtain $h = x \circledast \tilde{V}$. Also note that the original definition is actually a *cross correlation*. We will come back to this in the following section.

6.1.4 Waldo Revisited

Let's see what this looks like if we want to build an improved Waldo detector. The convolutional layer picks windows of a given size and weighs intensities according to the mask V, as demonstrated in Fig. 6.1.2. We expect that wherever the "waldoness" is highest, we will also find a peak in the hidden layer activations.

Fig. 6.1.2: Find Waldo.

There is just a problem with this approach: so far we blissfully ignored that images consist of 3 channels: red, green and blue. In reality, images are quite two-dimensional objects but rather as a 3rd order tensor, e.g., with shape $1024 \times 1024 \times 3$ pixels. Only two of these axes concern spatial relationships, while the 3rd can be regarded as assigning a multidimensional representation *to each pixel location*.

We thus index **x** as $x[i, j, k]$. The convolutional mask has to adapt accordingly. Instead of $V[a, b]$ we now have $V[a, b, c]$.

Moreover, just as our input consists of a 3rd order tensor it turns out to be a good idea to similarly formulate our hidden representations as 3rd order tensors. In other words, rather than just having a 1D representation corresponding to each spatial location, we want to have a multidimensional hidden representations corresponding to each spatial location. We could think of the hidden representation as comprising a number of 2D grids stacked on top of each other. These are sometimes called *channels* or *feature maps*. Intuitively you might imagine that at lower layers, some channels specialize to recognizing edges, We can take care of this by adding a fourth coordinate to V via $V[a, b, c, d]$. Putting all together we have:

$$h[i, j, k] = \sum_{a=-\Delta}^{\Delta} \sum_{b=-\Delta}^{\Delta} \sum_{c} V[a, b, c, k] \cdot x[i + a, j + b, c]. \tag{6.1.6}$$

This is the definition of a convolutional neural network layer. There are still many operations that we need to address. For instance, we need to figure out how to combine all the activations to a single output (e.g., whether there is a Waldo in the image). We also need to decide how to compute things efficiently, how to combine multiple layers, and whether it is a good idea to have many narrow or a few wide layers. All of this will be addressed in the remainder of the chapter.

Summary

- Translation invariance in images implies that all patches of an image will be treated in the same manner.
- Locality means that only a small neighborhood of pixels will be used for computation.
- Channels on input and output allows for meaningful feature analysis.

Exercises

1. Assume that the size of the convolution mask is $\Delta = 0$. Show that in this case the convolutional mask implements an MLP independently for each set of channels.
2. Why might translation invariance not be a good idea after all? Does it make sense for pigs to fly?
3. What happens at the boundary of an image?
4. Derive an analogous convolutional layer for audio.
5. What goes wrong when you apply the above reasoning to text? Hint: what is the structure of language?
6. Prove that $f \circledast g = g \circledast f$.

6.2 Convolutions for Images

Now that we understand how convolutional layers work in theory, we are ready to see how this works in practice. Since we have motivated convolutional neural networks by their applicability to image data, we will stick with image data in our examples, and begin by revisiting the convolutional layer that we introduced in the previous section. We note that strictly speaking, *convolutional* layers are a slight misnomer, since the operations are typically expressed as cross correlations.

6.2.1 The Cross-Correlation Operator

In a convolutional layer, an input array and a correlation kernel array are combined to produce an output array through a cross-correlation operation. Let's see how this works for two dimensions. In Fig. 6.2.1, the input is a two-dimensional array with a height of 3 and width of 3. We mark the shape of the array as 3×3 or (3, 3). The height and width of the kernel array are both 2. Common names for this array in the deep learning research community include *kernel* and *filter*. The shape of the kernel window (also known as the convolution window) is given precisely by the height and width of the kernel (here it is 2×2).

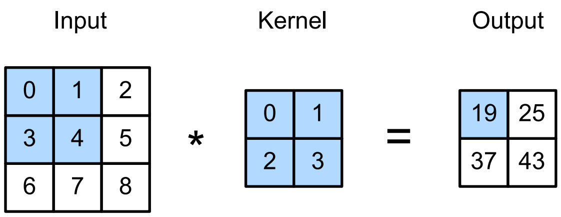


Fig. 6.2.1: Two-dimensional cross-correlation operation. The shaded portions are the first output element and the input and kernel array elements used in its computation: $0 \times 0 + 1 \times 1 + 3 \times 2 + 4 \times 3 = 19$.

In the two-dimensional cross-correlation operation, we begin with the convolution window positioned at the top-left corner of the input array and slide it across the input array, both from left to right and top to bottom. When the convolution window slides to a certain position, the input subarray contained in that window and the kernel array are multiplied (elementwise) and the resulting array is summed up yielding a single scalar value. This result if precisely the value of the output array at the corresponding location. Here, the output array has a height of 2 and width of 2 and the four elements are derived from the two-dimensional cross-correlation operation:

$$
\begin{aligned}
0 \times 0 + 1 \times 1 + 3 \times 2 + 4 \times 3 = 19,\\
1 \times 0 + 2 \times 1 + 4 \times 2 + 5 \times 3 = 25,\\
3 \times 0 + 4 \times 1 + 6 \times 2 + 7 \times 3 = 37,\\
4 \times 0 + 5 \times 1 + 7 \times 2 + 8 \times 3 = 43.
\end{aligned} \tag{6.2.1}
$$

Note that along each axis, the output is slightly *smaller* than the input. Because the kernel has a width greater than one, and we can only computer the cross-correlation for locations where the kernel fits wholly within the image, the output size is given by the input size $H \times W$ minus the size of the convolutional kernel $h \times w$ via $(H - h + 1) \times (W - w + 1)$. This is the case since we need enough space to 'shift' the convolutional kernel across the image (later we will see how to keep the size unchanged by padding the image with zeros around its boundary such that there is enough space to shift the kernel). Next, we implement the above process in the `corr2d` function. It accepts the input array `X` with the kernel array `K` and outputs the array `Y`.

```python
from mxnet import autograd, np, npx
from mxnet.gluon import nn
npx.set_np()

# Saved in the d2l package for later use
def corr2d(X, K):
    """Compute 2D cross-correlation."""
    h, w = K.shape
    Y = np.zeros((X.shape[0] - h + 1, X.shape[1] - w + 1))
    for i in range(Y.shape[0]):
        for j in range(Y.shape[1]):
            Y[i, j] = (X[i: i + h, j: j + w] * K).sum()
    return Y
```

We can construct the input array `X` and the kernel array `K` from the figure above to validate the output of the above implementations of the two-dimensional cross-correlation operation.

```
X = np.array([[0, 1, 2], [3, 4, 5], [6, 7, 8]])
K = np.array([[0, 1], [2, 3]])
corr2d(X, K)
```

```
array([[19., 25.],
       [37., 43.]])
```

6.2.2 Convolutional Layers

A convolutional layer cross-correlates the input and kernels and adds a scalar bias to produce an output. The parameters of the convolutional layer are precisely the values that constitute the kernel and the scalar bias. When training the models based on convolutional layers, we typically initialize the kernels randomly, just as we would with a fully-connected layer.

We are now ready to implement a two-dimensional convolutional layer based on the `corr2d` function defined above. In the `__init__` constructor function, we declare `weight` and `bias` as the two model parameters. The forward computation function `forward` calls the `corr2d` function and adds the bias. As with $h \times w$ cross-correlation we also refer to convolutional layers as $h \times w$ convolutions.

```
class Conv2D(nn.Block):
    def __init__(self, kernel_size, **kwargs):
        super(Conv2D, self).__init__(**kwargs)
        self.weight = self.params.get('weight', shape=kernel_size)
        self.bias = self.params.get('bias', shape=(1,))

    def forward(self, x):
        return corr2d(x, self.weight.data()) + self.bias.data()
```

6.2.3 Object Edge Detection in Images

Let's look at a simple application of a convolutional layer: detecting the edge of an object in an image by finding the location of the pixel change. First, we construct an 'image' of 6×8 pixels. The middle four columns are black (0) and the rest are white (1).

```
X = np.ones((6, 8))
X[:, 2:6] = 0
X
```

```
array([[1., 1., 0., 0., 0., 0., 1., 1.],
       [1., 1., 0., 0., 0., 0., 1., 1.],
       [1., 1., 0., 0., 0., 0., 1., 1.],
       [1., 1., 0., 0., 0., 0., 1., 1.],
       [1., 1., 0., 0., 0., 0., 1., 1.],
       [1., 1., 0., 0., 0., 0., 1., 1.]])
```

Next, we construct a kernel `K` with a height of 1 and width of 2. When we perform the cross-correlation operation with the input, if the horizontally adjacent elements are the same, the output is 0. Otherwise, the output is non-zero.

```
K = np.array([[1, -1]])
```

Enter X and our designed kernel K to perform the cross-correlation operations. As you can see, we will detect 1 for the edge from white to black and -1 for the edge from black to white. The rest of the outputs are 0.

```
Y = corr2d(X, K)
Y
```

```
array([[ 0.,  1.,  0.,  0.,  0., -1.,  0.],
       [ 0.,  1.,  0.,  0.,  0., -1.,  0.],
       [ 0.,  1.,  0.,  0.,  0., -1.,  0.],
       [ 0.,  1.,  0.,  0.,  0., -1.,  0.],
       [ 0.,  1.,  0.,  0.,  0., -1.,  0.],
       [ 0.,  1.,  0.,  0.,  0., -1.,  0.]])
```

Let's apply the kernel to the transposed image. As expected, it vanishes. The kernel K only detects vertical edges.

```
corr2d(X.T, K)
```

```
array([[0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.]])
```

6.2.4 Learning a Kernel

Designing an edge detector by finite differences [1, -1] is neat if we know this is precisely what we are looking for. However, as we look at larger kernels, and consider successive layers of convolutions, it might be impossible to specify precisely what each filter should be doing manually.

Now let's see whether we can learn the kernel that generated Y from X by looking at the (input, output) pairs only. We first construct a convolutional layer and initialize its kernel as a random array. Next, in each iteration, we will use the squared error to compare Y and the output of the convolutional layer, then calculate the gradient to update the weight. For the sake of simplicity, in this convolutional layer, we will ignore the bias.

We previously constructed the Conv2D class. However, since we used single-element assignments, Gluon has some trouble finding the gradient. Instead, we use the built-in Conv2D class provided by Gluon below.

```
# Construct a convolutional layer with 1 output channel
# (channels will be introduced in the following section)
# and a kernel array shape of (1, 2)
conv2d = nn.Conv2D(1, kernel_size=(1, 2))
```

(continues on next page)

(continued from previous page)

```
conv2d.initialize()

# The two-dimensional convolutional layer uses four-dimensional input and
# output in the format of (example channel, height, width), where the batch
# size (number of examples in the batch) and the number of channels are both 1
X = X.reshape(1, 1, 6, 8)
Y = Y.reshape(1, 1, 6, 7)

for i in range(10):
    with autograd.record():
        Y_hat = conv2d(X)
        l = (Y_hat - Y) ** 2
    l.backward()
    # For the sake of simplicity, we ignore the bias here
    conv2d.weight.data()[:] -= 3e-2 * conv2d.weight.grad()
    if (i + 1) % 2 == 0:
        print('batch %d, loss %.3f' % (i + 1, l.sum()))
```

```
batch 2, loss 4.949
batch 4, loss 0.831
batch 6, loss 0.140
batch 8, loss 0.024
batch 10, loss 0.004
```

As you can see, the error has dropped to a small value after 10 iterations. Now we will take a look at the kernel array we learned.

```
conv2d.weight.data().reshape(1, 2)
```

```
array([[ 0.9895    , -0.9873705]])
```

Indeed, the learned kernel array is remarkably close to the kernel array `K` we defined earlier.

6.2.5 Cross-Correlation and Convolution

Recall the observation from the previous section that cross-correlation and convolution are equivalent. In the figure above it is easy to see this correspondence. Simply flip the kernel from the bottom left to the top right. In this case the indexing in the sum is reverted, yet the same result can be obtained. In keeping with standard terminology with deep learning literature, we will continue to refer to the cross-correlation operation as a convolution even though, strictly-speaking, it is slightly different.

Summary

- The core computation of a two-dimensional convolutional layer is a two-dimensional cross-correlation operation. In its simplest form, this performs a cross-correlation operation on the two-dimensional input data and the kernel, and then adds a bias.
- We can design a kernel to detect edges in images.
- We can learn the kernel through data.

Exercises

1. Construct an image `X` with diagonal edges.
    - What happens if you apply the kernel `K` to it?
    - What happens if you transpose `X`?
    - What happens if you transpose `K`?
2. When you try to automatically find the gradient for the `Conv2D` class we created, what kind of error message do you see?
3. How do you represent a cross-correlation operation as a matrix multiplication by changing the input and kernel arrays?
4. Design some kernels manually.
    - What is the form of a kernel for the second derivative?
    - What is the kernel for the Laplace operator?
    - What is the kernel for an integral?
    - What is the minimum size of a kernel to obtain a derivative of degree d?

6.3 Padding and Stride

In the previous example, our input had a height and width of 3 and a convolution kernel with a height and width of 2, yielding an output with a height and a width of 2. In general, assuming the input shape is $n_h \times n_w$ and the convolution kernel window shape is $k_h \times k_w$, then the output shape will be

$$(n_h - k_h + 1) \times (n_w - k_w + 1). \tag{6.3.1}$$

Therefore, the output shape of the convolutional layer is determined by the shape of the input and the shape of the convolution kernel window.

In several cases we might want to incorporate particular techniques—padding and strides, regarding the size of the output:

- In general, since kernels generally have width and height greater than 1, that means that after applying many successive convolutions, we will wind up with an output that is much smaller than our input. If we start with a 240×240 pixel image, 10 layers of 5×5 convolutions reduce the image to 200×200 pixels, slicing off 30% of the image and with it obliterating any interesting information on the boundaries of the original image. *Padding* handles this issue.
- In some cases, we want to reduce the resolution drastically if say we find our
- original input resolution to be unwieldy. *Strides* can help in these instances.

6.3.1 Padding

As described above, one tricky issue when applying convolutional layers is that losing pixels on the perimeter of our image. Since we typically use small kernels, for any given convolution, we might only lose a few pixels, but this can add up as we apply many successive convolutional layers. One straightforward solution to this problem is to add extra pixels of filler around the boundary of our input image, thus increasing the effective size of the image Typically, we set the values of the extra pixels to 0. In Fig. 6.3.1, we pad a 3×5 input, increasing its size to 5×7. The corresponding output then increases to a 4×6 matrix.

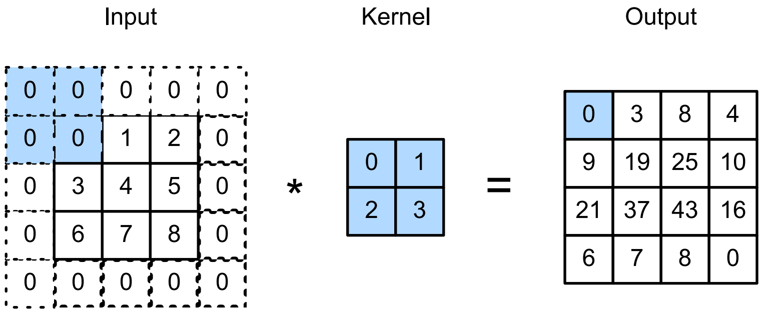


Fig. 6.3.1: Two-dimensional cross-correlation with padding. The shaded portions are the input and kernel array elements used by the first output element: $0 \times 0 + 0 \times 1 + 0 \times 2 + 0 \times 3 = 0$.

In general, if we add a total of p_h rows of padding (roughly half on top and half on bottom) and a total of p_w columns of padding (roughly half on the left and half on the right), the output shape will be

$$(n_h - k_h + p_h + 1) \times (n_w - k_w + p_w + 1). \tag{6.3.2}$$

This means that the height and width of the output will increase by p_h and p_w respectively.

In many cases, we will want to set $p_h = k_h - 1$ and $p_w = k_w - 1$ to give the input and output the same height and width. This will make it easier to predict the output shape of each layer when constructing the network. Assuming that k_h is odd here, we will pad $p_h/2$ rows on both sides of the height. If k_h is even, one possibility is to pad $\lceil p_h/2 \rceil$ rows on the top of the input and $\lfloor p_h/2 \rfloor$ rows on the bottom. We will pad both sides of the width in the same way.

Convolutional neural networks commonly use convolutional kernels with odd height and width values, such as 1, 3, 5, or 7. Choosing odd kernel sizes has the benefit that we can preserve the spatial dimensionality while padding with the same number of rows on top and bottom, and the same number of columns on left and right.

Moreover, this practice of using odd kernels and padding to precisely preserve dimensionality offers a clerical benefit. For any two-dimensional array X, when the kernels size is odd and the number of padding rows and columns on all sides are the same, producing an output with the same height and width as the input, we know that the output Y[i, j] is calculated by cross-correlation of the input and convolution kernel with the window centered on X[i, j].

In the following example, we create a two-dimensional convolutional layer with a height and width of 3 and apply 1 pixel of padding on all sides. Given an input with a height and width of 8, we find that the height and width of the output is also 8.

```
from mxnet import np, npx
from mxnet.gluon import nn
npx.set_np()

# For convenience, we define a function to calculate the convolutional layer.
# This function initializes the convolutional layer weights and performs
# corresponding dimensionality elevations and reductions on the input and
# output
def comp_conv2d(conv2d, X):
    conv2d.initialize()
    # (1, 1) indicates that the batch size and the number of channels
    # (described in later chapters) are both 1
    X = X.reshape((1, 1) + X.shape)
    Y = conv2d(X)
    # Exclude the first two dimensions that do not interest us: batch and
    # channel
    return Y.reshape(Y.shape[2:])

# Note that here 1 row or column is padded on either side, so a total of 2
# rows or columns are added
conv2d = nn.Conv2D(1, kernel_size=3, padding=1)
X = np.random.uniform(size=(8, 8))
comp_conv2d(conv2d, X).shape
```

```
(8, 8)
```

When the height and width of the convolution kernel are different, we can make the output and input have the same height and width by setting different padding numbers for height and width.

```
# Here, we use a convolution kernel with a height of 5 and a width of 3. The
# padding numbers on both sides of the height and width are 2 and 1,
# respectively
conv2d = nn.Conv2D(1, kernel_size=(5, 3), padding=(2, 1))
comp_conv2d(conv2d, X).shape
```

```
(8, 8)
```

6.3.2 Stride

When computing the cross-correlation, we start with the convolution window at the top-left corner of the input array, and then slide it over all locations both down and to the right. In previous examples, we default to sliding one pixel at a time. However, sometimes, either for computational efficiency or because we wish to downsample, we move our window more than one pixel at a time, skipping the intermediate locations.

We refer to the number of rows and columns traversed per slide as the *stride*. So far, we have used strides of 1, both for height and width. Sometimes, we may want to use a larger stride. Fig. 6.3.2 shows a two-dimensional cross-correlation operation with a stride of 3 vertically and 2 horizontally. We can see that when the second element of the first column is output, the convolution window slides down three rows. The convolution window slides two columns to the right when the second element of the first row is output. When the convolution window slides three columns to the right on the input, there is no output because the input element cannot fill the window (unless we add another column of padding).

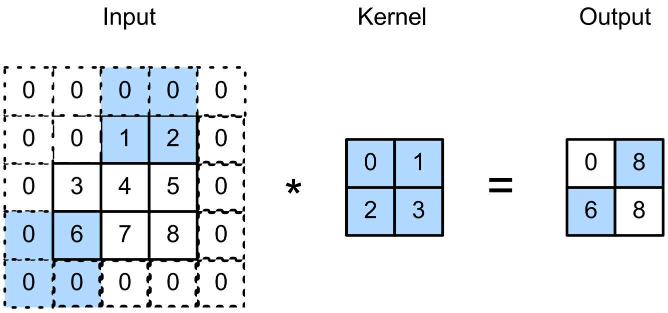


Fig. 6.3.2: Cross-correlation with strides of 3 and 2 for height and width respectively. The shaded portions are the output element and the input and core array elements used in its computation: $0\times0+0\times1+1\times2+2\times3=8$, $0\times0+6\times1+0\times2+0\times3=6$.

In general, when the stride for the height is s_h and the stride for the width is s_w, the output shape is

$$\lfloor(n_h-k_h+p_h+s_h)/s_h\rfloor \times \lfloor(n_w-k_w+p_w+s_w)/s_w\rfloor. \tag{6.3.3}$$

If we set $p_h = k_h - 1$ and $p_w = k_w - 1$, then the output shape will be simplified to $\lfloor(n_h+s_h-1)/s_h\rfloor \times \lfloor(n_w+s_w-1)/s_w\rfloor$. Going a step further, if the input height and width are divisible by the strides on the height and width, then the output shape will be $(n_h/s_h) \times (n_w/s_w)$.

Below, we set the strides on both the height and width to 2, thus halving the input height and width.

```
conv2d = nn.Conv2D(1, kernel_size=3, padding=1, strides=2)
comp_conv2d(conv2d, X).shape
```

```
(4, 4)
```

Next, we will look at a slightly more complicated example.

```
conv2d = nn.Conv2D(1, kernel_size=(3, 5), padding=(0, 1), strides=(3, 4))
comp_conv2d(conv2d, X).shape
```

```
(2, 2)
```

For the sake of brevity, when the padding number on both sides of the input height and width are p_h and p_w respectively, we call the padding (p_h, p_w). Specifically, when $p_h = p_w = p$, the padding is p. When the strides on the height and width are s_h and s_w, respectively, we call the stride (s_h, s_w). Specifically, when $s_h = s_w = s$, the stride is s. By default, the padding is 0 and the stride is 1. In practice we rarely use inhomogeneous strides or padding, i.e., we usually have $p_h = p_w$ and $s_h = s_w$.

Summary

- Padding can increase the height and width of the output. This is often used to give the output the same height and width as the input.
- The stride can reduce the resolution of the output, for example reducing the height and width of the output to only $1/n$ of the height and width of the input (n is an integer greater than 1).
- Padding and stride can be used to adjust the dimensionality of the data effectively.

Exercises

1. For the last example in this section, use the shape calculation formula to calculate the output shape to see if it is consistent with the experimental results.
2. Try other padding and stride combinations on the experiments in this section.
3. For audio signals, what does a stride of 2 correspond to?
4. What are the computational benefits of a stride larger than 1.

6.4 Multiple Input and Output Channels

While we have described the multiple channels that comprise each image (e.g., color images have the standard RGB channels to indicate the amount of red, green and blue), until now, we simplified all of our numerical examples by working with just a single input and a single output channel. This has allowed us to think of our inputs, convolutional kernels, and outputs each as two-dimensional arrays.

When we add channels into the mix, our inputs and hidden representations both become three-dimensional arrays. For example, each RGB input image has shape $3 \times h \times w$. We refer to this axis, with a size of 3, as the channel dimension. In this section, we will take a deeper look at convolution kernels with multiple input and multiple output channels.

6.4.1 Multiple Input Channels

When the input data contains multiple channels, we need to construct a convolution kernel with the same number of input channels as the input data, so that it can perform cross-correlation with the input data. Assuming that the number of channels for the input data is c_i, the number of input channels of the convolution kernel also needs to be c_i. If our convolution kernel's window shape is $k_h \times k_w$, then when $c_i = 1$, we can think of our convolution kernel as just a two-dimensional array of shape $k_h \times k_w$.

However, when $c_i > 1$, we need a kernel that contains an array of shape $k_h \times k_w$ *for each input channel*. Concatenating these c_i arrays together yields a convolution kernel of shape $c_i \times k_h \times k_w$. Since the input and convolution kernel each have c_i channels, we can perform a cross-correlation operation on the two-dimensional array of the input and the two-dimensional kernel array of the convolution kernel for each channel, adding the c_i results together (summing over the channels) to yield a two-dimensional array. This is the result of a two-dimensional cross-correlation between multi-channel input data and a *multi-input channel* convolution kernel.

In Fig. 6.4.1, we demonstrate an example of a two-dimensional cross-correlation with two input channels. The shaded portions are the first output element as well as the input and kernel array elements used in its computation: $(1\times1+2\times2+4\times3+5\times4)+(0\times0+1\times1+3\times2+4\times3) = 56$.

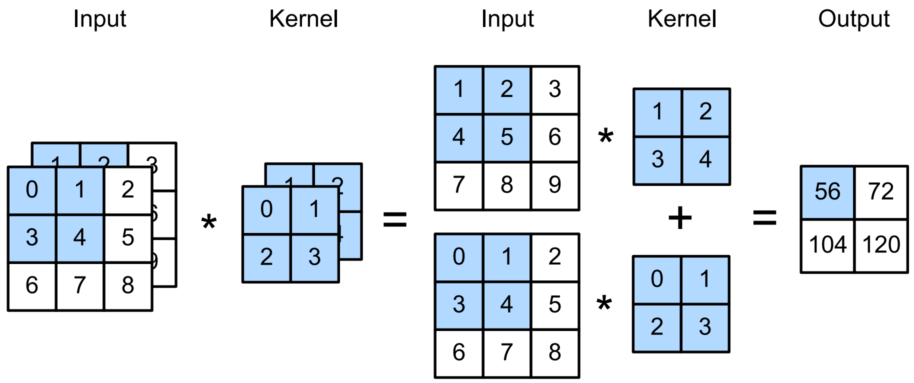


Fig. 6.4.1: Cross-correlation computation with 2 input channels. The shaded portions are the first output element as well as the input and kernel array elements used in its computation: $(1 \times 1 + 2 \times 2 + 4 \times 3 + 5 \times 4) + (0 \times 0 + 1 \times 1 + 3 \times 2 + 4 \times 3) = 56$.

To make sure we really understand what is going on here, we can implement cross-correlation operations with multiple input channels ourselves. Notice that all we are doing is performing one cross-correlation operation per channel and then adding up the results using the `add_n` function.

```
import d2l
from mxnet import np, npx
npx.set_np()

def corr2d_multi_in(X, K):
    # First, traverse along the 0th dimension (channel dimension) of X and K.
    # Then, add them together by using * to turn the result list into a
    # positional argument of the add_n function
    return sum(d2l.corr2d(x, k) for x, k in zip(X, K))
```

We can construct the input array `X` and the kernel array `K` corresponding to the values in the above diagram to validate the output of the cross-correlation operation.

```
X = np.array([[[0, 1, 2], [3, 4, 5], [6, 7, 8]],
              [[1, 2, 3], [4, 5, 6], [7, 8, 9]]])
K = np.array([[[0, 1], [2, 3]], [[1, 2], [3, 4]]])

corr2d_multi_in(X, K)
```

```
array([[ 56.,  72.],
       [104., 120.]])
```

6.4.2 Multiple Output Channels

Regardless of the number of input channels, so far we always ended up with one output channel. However, as we discussed earlier, it turns out to be essential to have multiple channels at each layer. In the most popular neural network architectures, we actually increase the channel dimension as we go higher up in the neural network, typically downsampling to trade off spatial resolution for greater *channel depth*. Intuitively, you could think of each channel as responding to some different set of features. Reality is a bit more complicated than the most naive interpretations of this intuition since representations are not learned independent but are rather optimized to be jointly useful. So it may not be that a single channel learns an edge detector but rather that some direction in channel space corresponds to detecting edges.

Denote by c_i and c_o the number of input and output channels, respectively, and let k_h and k_w be the height and width of the kernel. To get an output with multiple channels, we can create a kernel array of shape $c_i \times k_h \times k_w$ for each output channel. We concatenate them on the output channel dimension, so that the shape of the convolution kernel is $c_o \times c_i \times k_h \times k_w$. In cross-correlation operations, the result on each output channel is calculated from the convolution kernel corresponding to that output channel and takes input from all channels in the input array.

We implement a cross-correlation function to calculate the output of multiple channels as shown below.

```
def corr2d_multi_in_out(X, K):
    # Traverse along the 0th dimension of K, and each time, perform
    # cross-correlation operations with input X. All of the results are merged
    # together using the stack function
    return np.stack([corr2d_multi_in(X, k) for k in K])
```

We construct a convolution kernel with 3 output channels by concatenating the kernel array `K` with `K+1` (plus one for each element in `K`) and `K+2`.

```
K = np.stack((K, K + 1, K + 2))
K.shape
```

```
(3, 2, 2, 2)
```

Below, we perform cross-correlation operations on the input array `X` with the kernel array `K`. Now the output contains 3 channels. The result of the first channel is consistent with the result of the previous input array `X` and the multi-input channel, single-output channel kernel.

```
corr2d_multi_in_out(X, K)
```

```
array([[[ 56.,  72.],
        [104., 120.]],

       [[ 76., 100.],
        [148., 172.]],

       [[ 96., 128.],
        [192., 224.]]])
```

6.4.3 1×1 Convolutional Layer

At first, a 1×1 convolution, i.e., $k_h = k_w = 1$, does not seem to make much sense. After all, a convolution correlates adjacent pixels. A 1×1 convolution obviously does not. Nonetheless, they are popular operations that are sometimes included in the designs of complex deep networks. Let's see in some detail what it actually does.

Because the minimum window is used, the 1×1 convolution loses the ability of larger convolutional layers to recognize patterns consisting of interactions among adjacent elements in the height and width dimensions. The only computation of the 1×1 convolution occurs on the channel dimension.

Fig. 6.4.2 shows the cross-correlation computation using the 1×1 convolution kernel with 3 input channels and 2 output channels. Note that the inputs and outputs have the same height and width. Each element in the output is derived from a linear combination of elements *at the same position* in the input image. You could think of the 1×1 convolutional layer as constituting a fully-connected layer applied at every single pixel location to transform the c_i corresponding input values into c_o output values. Because this is still a convolutional layer, the weights are tied across pixel location Thus the 1×1 convolutional layer requires $c_o \times c_i$ weights (plus the bias terms).

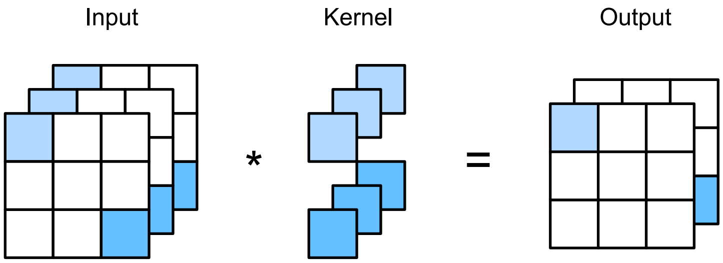


Fig. 6.4.2: The cross-correlation computation uses the 1×1 convolution kernel with 3 input channels and 2 output channels. The inputs and outputs have the same height and width.

Let's check whether this works in practice: we implement the 1×1 convolution using a fully-connected layer. The only thing is that we need to make some adjustments to the data shape before and after the matrix multiplication.

```
def corr2d_multi_in_out_1x1(X, K):
    c_i, h, w = X.shape
    c_o = K.shape[0]
```

(continues on next page)

(continued from previous page)

```
    X = X.reshape(c_i, h * w)
    K = K.reshape(c_o, c_i)
    Y = np.dot(K, X)  # Matrix multiplication in the fully connected layer
    return Y.reshape(c_o, h, w)
```

When performing 1×1 convolution, the above function is equivalent to the previously implemented cross-correlation function `corr2d_multi_in_out`. Let's check this with some reference data.

```
X = np.random.uniform(size=(3, 3, 3))
K = np.random.uniform(size=(2, 3, 1, 1))

Y1 = corr2d_multi_in_out_1x1(X, K)
Y2 = corr2d_multi_in_out(X, K)

np.abs(Y1 - Y2).sum() < 1e-6
```

```
array(True)
```

Summary

- Multiple channels can be used to extend the model parameters of the convolutional layer.
- The 1×1 convolutional layer is equivalent to the fully-connected layer, when applied on a per pixel basis.
- The 1×1 convolutional layer is typically used to adjust the number of channels between network layers and to control model complexity.

Exercises

1. Assume that we have two convolutional kernels of size k_1 and k_2 respectively (with no nonlinearity in between).
    - Prove that the result of the operation can be expressed by a single convolution.
    - What is the dimensionality of the equivalent single convolution?
    - Is the converse true?
2. Assume an input shape of $c_i \times h \times w$ and a convolution kernel with the shape $c_o \times c_i \times k_h \times k_w$, padding of (p_h, p_w), and stride of (s_h, s_w).
    - What is the computational cost (multiplications and additions) for the forward computation?
    - What is the memory footprint?
    - What is the memory footprint for the backward computation?
    - What is the computational cost for the backward computation?
3. By what factor does the number of calculations increase if we double the number of input channels c_i and the number of output channels c_o? What happens if we double the padding?

4. If the height and width of the convolution kernel is $k_h = k_w = 1$, what is the complexity of the forward computation?
5. Are the variables `Y1` and `Y2` in the last example of this section exactly the same? Why?
6. How would you implement convolutions using matrix multiplication when the convolution window is not 1×1?

6.5 Pooling

Often, as we process images, we want to gradually reduce the spatial resolution of our hidden representations, aggregating information so that the higher up we go in the network, the larger the receptive field (in the input) to which each hidden node is sensitive.

Often our ultimate task asks some global question about the image, e.g., *does it contain a cat?* So typically the nodes of our final layer should be sensitive to the entire input. By gradually aggregating information, yielding coarser and coarser maps, we accomplish this goal of ultimately learning a global representation, while keeping all of the advantages of convolutional layers at the intermediate layers of processing.

Moreover, when detecting lower-level features, such as edges (as discussed in Section 6.2), we often want our representations to be somewhat invariant to translation. For instance, if we take the image `X` with a sharp delineation between black and white and shift the whole image by one pixel to the right, i.e., `Z[i, j] = X[i, j+1]`, then the output for the new image `Z` might be vastly different. The edge will have shifted by one pixel and with it all the activations. In reality, objects hardly ever occur exactly at the same place. In fact, even with a tripod and a stationary object, vibration of the camera due to the movement of the shutter might shift everything by a pixel or so (high-end cameras are loaded with special features to address this problem).

This section introduces pooling layers, which serve the dual purposes of mitigating the sensitivity of convolutional layers to location and of spatially downsampling representations.

6.5.1 Maximum Pooling and Average Pooling

Like convolutional layers, pooling operators consist of a fixed-shape window that is slid over all regions in the input according to its stride, computing a single output for each location traversed by the fixed-shape window (sometimes known as the *pooling window*). However, unlike the cross-correlation computation of the inputs and kernels in the convolutional layer, the pooling layer contains no parameters (there is no *filter*). Instead, pooling operators are deterministic, typically calculating either the maximum or the average value of the elements in the pooling window. These operations are called *maximum pooling* (*max pooling* for short) and *average pooling*, respectively.

In both cases, as with the cross-correlation operator, we can think of the pooling window as starting from the top left of the input array and sliding across the input array from left to right and top to bottom. At each location that the pooling window hits, it computes the maximum or average value of the input subarray in the window (depending on whether *max* or *average* pooling is employed).

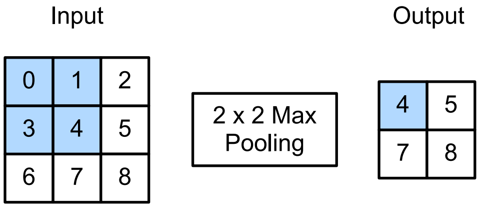


Fig. 6.5.1: Maximum pooling with a pooling window shape of 2×2. The shaded portions represent the first output element and the input element used for its computation: $\max(0, 1, 3, 4) = 4$

The output array in Fig. 6.5.1 above has a height of 2 and a width of 2. The four elements are derived from the maximum value of max:

$$\begin{aligned} \max(0, 1, 3, 4) &= 4, \\ \max(1, 2, 4, 5) &= 5, \\ \max(3, 4, 6, 7) &= 7, \\ \max(4, 5, 7, 8) &= 8. \end{aligned} \tag{6.5.1}$$

A pooling layer with a pooling window shape of $p \times q$ is called a $p \times q$ pooling layer. The pooling operation is called $p \times q$ pooling.

Let's return to the object edge detection example mentioned at the beginning of this section. Now we will use the output of the convolutional layer as the input for 2×2 maximum pooling. Set the convolutional layer input as `X` and the pooling layer output as `Y`. Whether or not the values of `X[i, j]` and `X[i, j+1]` are different, or `X[i, j+1]` and `X[i, j+2]` are different, the pooling layer outputs all include `Y[i, j]=1`. That is to say, using the 2×2 maximum pooling layer, we can still detect if the pattern recognized by the convolutional layer moves no more than one element in height and width.

In the code below, we implement the forward computation of the pooling layer in the `pool2d` function. This function is similar to the `corr2d` function in Section 6.2. However, here we have no kernel, computing the output as either the max or the average of each region in the input..

```
from mxnet import np, npx
from mxnet.gluon import nn
npx.set_np()

def pool2d(X, pool_size, mode='max'):
    p_h, p_w = pool_size
    Y = np.zeros((X.shape[0] - p_h + 1, X.shape[1] - p_w + 1))
    for i in range(Y.shape[0]):
        for j in range(Y.shape[1]):
            if mode == 'max':
                Y[i, j] = np.max(X[i: i + p_h, j: j + p_w])
            elif mode == 'avg':
                Y[i, j] = X[i: i + p_h, j: j + p_w].mean()
    return Y
```

We can construct the input array `X` in the above diagram to validate the output of the two-dimensional maximum pooling layer.

```
X = np.array([[0, 1, 2], [3, 4, 5], [6, 7, 8]])
pool2d(X, (2, 2))
```

```
array([[4., 5.],
       [7., 8.]])
```

At the same time, we experiment with the average pooling layer.

```
pool2d(X, (2, 2), 'avg')
```

```
array([[2., 3.],
       [5., 6.]])
```

6.5.2 Padding and Stride

As with convolutional layers, pooling layers can also change the output shape. And as before, we can alter the operation to achieve a desired output shape by padding the input and adjusting the stride. We can demonstrate the use of padding and strides in pooling layers via the two-dimensional maximum pooling layer MaxPool2D shipped in MXNet Gluon's `nn` module. We first construct an input data of shape `(1, 1, 4, 4)`, where the first two dimensions are batch and channel.

```
X = np.arange(16).reshape(1, 1, 4, 4)
X
```

```
array([[[[ 0.,  1.,  2.,  3.],
         [ 4.,  5.,  6.,  7.],
         [ 8.,  9., 10., 11.],
         [12., 13., 14., 15.]]]])
```

By default, the stride in the `MaxPool2D` class has the same shape as the pooling window. Below, we use a pooling window of shape `(3, 3)`, so we get a stride shape of `(3, 3)` by default.

```
pool2d = nn.MaxPool2D(3)
# Because there are no model parameters in the pooling layer, we do not need
# to call the parameter initialization function
pool2d(X)
```

```
array([[[[10.]]]])
```

The stride and padding can be manually specified.

```
pool2d = nn.MaxPool2D(3, padding=1, strides=2)
pool2d(X)
```

```
array([[[[ 5.,  7.],
         [13., 15.]]]])
```

Of course, we can specify an arbitrary rectangular pooling window and specify the padding and stride for height and width, respectively.

```
pool2d = nn.MaxPool2D((2, 3), padding=(1, 2), strides=(2, 3))
pool2d(X)
```

```
array([[[[ 0.,  3.],
         [ 8., 11.],
         [12., 15.]]]])
```

6.5.3 Multiple Channels

When processing multi-channel input data, the pooling layer pools each input channel separately, rather than adding the inputs of each channel by channel as in a convolutional layer. This means that the number of output channels for the pooling layer is the same as the number of input channels. Below, we will concatenate arrays `X` and `X+1` on the channel dimension to construct an input with 2 channels.

```
X = np.concatenate((X, X + 1), axis=1)
X
```

```
array([[[[ 0.,  1.,  2.,  3.],
         [ 4.,  5.,  6.,  7.],
         [ 8.,  9., 10., 11.],
         [12., 13., 14., 15.]],

        [[ 1.,  2.,  3.,  4.],
         [ 5.,  6.,  7.,  8.],
         [ 9., 10., 11., 12.],
         [13., 14., 15., 16.]]]])
```

As we can see, the number of output channels is still 2 after pooling.

```
pool2d = nn.MaxPool2D(3, padding=1, strides=2)
pool2d(X)
```

```
array([[[[ 5.,  7.],
         [13., 15.]],

        [[ 6.,  8.],
         [14., 16.]]]])
```

Summary

- Taking the input elements in the pooling window, the maximum pooling operation assigns the maximum value as the output and the average pooling operation assigns the average value as the output.
- One of the major functions of a pooling layer is to alleviate the excessive sensitivity of the convolutional layer to location.
- We can specify the padding and stride for the pooling layer.

- Maximum pooling, combined with a stride larger than 1 can be used to reduce the resolution.
- The pooling layer's number of output channels is the same as the number of input channels.

Exercises

1. Can you implement average pooling as a special case of a convolution layer? If so, do it.
2. Can you implement max pooling as a special case of a convolution layer? If so, do it.
3. What is the computational cost of the pooling layer? Assume that the input to the pooling layer is of size $c \times h \times w$, the pooling window has a shape of $p_h \times p_w$ with a padding of (p_h, p_w) and a stride of (s_h, s_w).
4. Why do you expect maximum pooling and average pooling to work differently?
5. Do we need a separate minimum pooling layer? Can you replace it with another operation?
6. Is there another operation between average and maximum pooling that you could consider (hint: recall the softmax)? Why might it not be so popular?

6.6 Convolutional Neural Networks (LeNet)

We are now ready to put all of the tools together to deploy your first fully-functional convolutional neural network. In our first encounter with image data we applied a multilayer perceptron (Section 4.2) to pictures of clothing in the Fashion-MNIST dataset. Each image in Fashion-MNIST consisted of a two-dimensional 28×28 matrix. To make this data amenable to multilayer perceptrons which anticipate receiving inputs as one-dimensional fixed-length vectors, we first flattened each image, yielding vectors of length 784, before processing them with a series of fully-connected layers.

Now that we have introduced convolutional layers, we can keep the image in its original spatially-organized grid, processing it with a series of successive convolutional layers. Moreover, because we are using convolutional layers, we can enjoy a considerable savings in the number of parameters required.

In this section, we will introduce one of the first published convolutional neural networks whose benefit was first demonstrated by Yann Lecun, then a researcher at AT&T Bell Labs, for the purpose of recognizing handwritten digits in images—LeNet5[102]. In the 90s, their experiments with LeNet gave the first compelling evidence that it was possible to train convolutional neural networks by backpropagation. Their model achieved outstanding results at the time (only matched by Support Vector Machines at the time) and was adopted to recognize digits for processing deposits in ATM machines. Some ATMs still run the code that Yann and his colleague Leon Bottou wrote in the 1990s!

[102] http://yann.lecun.com/exdb/lenet/

6.6.1 LeNet

In a rough sense, we can think LeNet as consisting of two parts: (i) a block of convolutional layers; and (ii) a block of fully-connected layers. Before getting into the weeds, let's briefly review the model in Fig. 6.6.1.

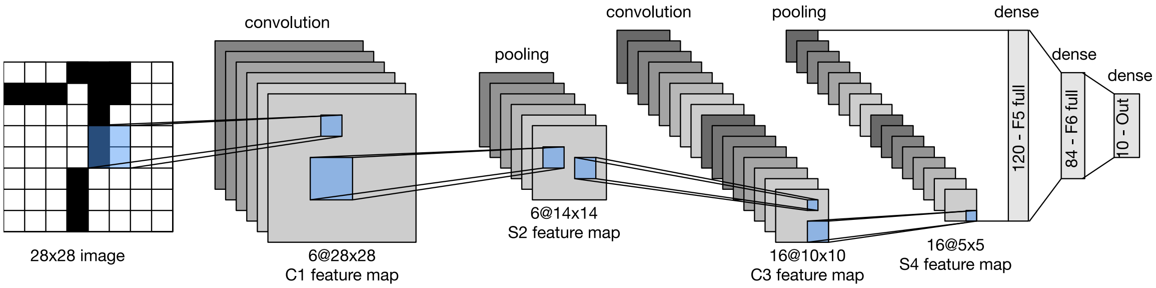


Fig. 6.6.1: Data flow in LeNet 5. The input is a handwritten digit, the output a probabilitiy over 10 possible outcomes.

The basic units in the convolutional block are a convolutional layer and a subsequent average pooling layer (note that max-pooling works better, but it had not been invented in the 90s yet). The convolutional layer is used to recognize the spatial patterns in the image, such as lines and the parts of objects, and the subsequent average pooling layer is used to reduce the dimensionality. The convolutional layer block is composed of repeated stacks of these two basic units. Each convolutional layer uses a 5×5 kernel and processes each output with a sigmoid activation function (again, note that ReLUs are now known to work more reliably, but had not been invented yet). The first convolutional layer has 6 output channels, and second convolutional layer increases channel depth further to 16.

However, coinciding with this increase in the number of channels, the height and width are shrunk considerably. Therefore, increasing the number of output channels makes the parameter sizes of the two convolutional layers similar. The two average pooling layers are of size 2×2 and take stride 2 (note that this means they are non-overlapping). In other words, the pooling layer downsamples the representation to be precisely *one quarter* the pre-pooling size.

The convolutional block emits an output with size given by (batch size, channel, height, width). Before we can pass the convolutional block's output to the fully-connected block, we must flatten each example in the minibatch. In other words, we take this 4D input and transform it into the 2D input expected by fully-connected layers: as a reminder, the first dimension indexes the examples in the minibatch and the second gives the flat vector representation of each example. LeNet's fully-connected layer block has three fully-connected layers, with 120, 84, and 10 outputs, respectively. Because we are still performing classification, the 10 dimensional output layer corresponds to the number of possible output classes.

While getting to the point where you truly understand what is going on inside LeNet may have taken a bit of work, you can see below that implementing it in a modern deep learning library is remarkably simple. Again, we will rely on the Sequential class.

```
import d2l
from mxnet import autograd, gluon, init, np, npx
from mxnet.gluon import nn
npx.set_np()

net = nn.Sequential()
net.add(nn.Conv2D(channels=6, kernel_size=5, padding=2, activation='sigmoid'),
        nn.AvgPool2D(pool_size=2, strides=2),
        nn.Conv2D(channels=16, kernel_size=5, activation='sigmoid'),
        nn.AvgPool2D(pool_size=2, strides=2),
        # Dense will transform the input of the shape (batch size, channel,
        # height, width) into the input of the shape (batch size,
        # channel * height * width) automatically by default
        nn.Dense(120, activation='sigmoid'),
        nn.Dense(84, activation='sigmoid'),
        nn.Dense(10))
```

As compared to the original network, we took the liberty of replacing the Gaussian activation in the last layer by a regular dense layer, which tends to be significantly more convenient to train. Other than that, this network matches the historical definition of LeNet5.

Next, let's take a look of an example. As shown in Fig. 6.6.2, we feed a single-channel example of size 28×28 into the network and perform a forward computation layer by layer printing the output shape at each layer to make sure we understand what is happening here.

```
X = np.random.uniform(size=(1, 1, 28, 28))
net.initialize()
for layer in net:
    X = layer(X)
    print(layer.name, 'output shape:\t', X.shape)
```

Note that the height and width of the representation at each layer throughout the convolutional block is reduced (compared to the previous layer). The convolutional layer uses a kernel with a height and width of 5, which with only 2 pixels of padding in the first convolutional layer and none in the second convolutional layer leads to reductions in both height and width by 2 and 4 pixels, respectively. Moreover each pooling layer halves the height and width. However, as we go up the stack of layers, the number of channels increases layer-over-layer from 1 in the input to 6 after the first convolutional layer and 16 after the second layer. Then, the fully-connected layer reduces dimensionality layer by layer, until emitting an output that matches the number of image classes.

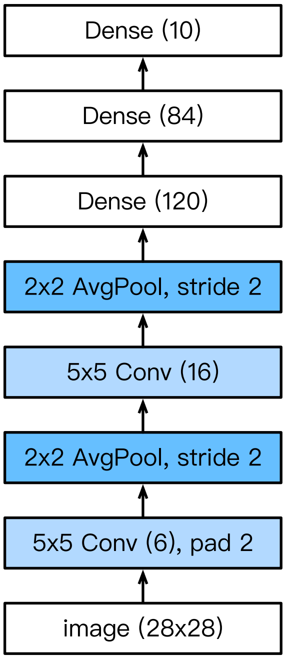


Fig. 6.6.2: Compressed notation for LeNet5

6.6.2 Data Acquisition and Training

Now that we have implemented the model, we might as well run some experiments to see what we can accomplish with the LeNet model. While it might serve nostalgia to train LeNet on the original MNIST dataset, that dataset has become too easy, with MLPs getting over 98% accuracy, so it would be hard to see the benefits of convolutional networks. Thus we will stick with Fashion-MNIST as our dataset because while it has the same shape (28×28 images), this dataset is notably more challenging.

```
batch_size = 256
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size=batch_size)
```

While convolutional networks may have few parameters, they can still be significantly more expensive to compute than a similarly deep multilayer perceptron so if you have access to a GPU, this might be a good time to put it into action to speed up training.

For evaluation, we need to make a slight modification to the `evaluate_accuracy` function that we described in Section 3.6. Since the full dataset lives on the CPU, we need to copy it to the GPU before we can compute our models. This is accomplished via the `as_in_context` function described in Section 5.6.

```
# Saved in the d2l package for later use
def evaluate_accuracy_gpu(net, data_iter, ctx=None):
    if not ctx:  # Query the first device the first parameter is on
        ctx = list(net.collect_params().values())[0].list_ctx()[0]
    metric = d2l.Accumulator(2)  # num_corrected_examples, num_examples
    for X, y in data_iter:
        X, y = X.as_in_context(ctx), y.as_in_context(ctx)
        metric.add(d2l.accuracy(net(X), y), y.size)
    return metric[0]/metric[1]
```

We also need to update our training function to deal with GPUs. Unlike the train_epoch_ch3 defined in Section 3.6, we now need to move each batch of data to our designated context (hopefully, the GPU) prior to making the forward and backward passes.

The training function train_ch5 is also very similar to train_ch3 defined in Section 3.6. Since we will deal with networks with tens of layers now, the function will only support Gluon models. We initialize the model parameters on the device indicated by ctx, this time using the Xavier initializer. The loss function and the training algorithm still use the cross-entropy loss function and minibatch stochastic gradient descent. Since each epoch takes tens of seconds to run, we visualize the training loss in a finer granularity.

```
# Saved in the d2l package for later use
def train_ch5(net, train_iter, test_iter, num_epochs, lr, ctx=d2l.try_gpu()):
    net.initialize(force_reinit=True, ctx=ctx, init=init.Xavier())
    loss = gluon.loss.SoftmaxCrossEntropyLoss()
    trainer = gluon.Trainer(net.collect_params(),
                            'sgd', {'learning_rate': lr})
    animator = d2l.Animator(xlabel='epoch', xlim=[0, num_epochs],
                            legend=['train loss', 'train acc', 'test acc'])
    timer = d2l.Timer()
    for epoch in range(num_epochs):
        metric = d2l.Accumulator(3)  # train_loss, train_acc, num_examples
        for i, (X, y) in enumerate(train_iter):
            timer.start()
            # Here is the only difference compared to train_epoch_ch3
            X, y = X.as_in_context(ctx), y.as_in_context(ctx)
            with autograd.record():
                y_hat = net(X)
                l = loss(y_hat, y)
            l.backward()
            trainer.step(X.shape[0])
            metric.add(l.sum(), d2l.accuracy(y_hat, y), X.shape[0])
            timer.stop()
            train_loss, train_acc = metric[0]/metric[2], metric[1]/metric[2]
            if (i+1) % 50 == 0:
                animator.add(epoch + i/len(train_iter),
                             (train_loss, train_acc, None))
        test_acc = evaluate_accuracy_gpu(net, test_iter)
        animator.add(epoch+1, (None, None, test_acc))
    print('loss %.3f, train acc %.3f, test acc %.3f' % (
        train_loss, train_acc, test_acc))
    print('%.1f exampes/sec on %s' % (metric[2]*num_epochs/timer.sum(), ctx))
```

Now let's train the model.

```
lr, num_epochs = 0.9, 10
train_ch5(net, train_iter, test_iter, num_epochs, lr)
```

Summary

- A convolutional neural network (in short, ConvNet) is a network using convolutional layers.
- In a ConvNet we alternate between convolutions, nonlinearities and often also pooling operations.
- Ultimately the resolution is reduced prior to emitting an output via one (or more) dense layers.
- LeNet was the first successful deployment of such a network.

Exercises

1. Replace the average pooling with max pooling. What happens?
2. Try to construct a more complex network based on LeNet to improve its accuracy.
    - Adjust the convolution window size.
    - Adjust the number of output channels.
    - Adjust the activation function (ReLU?).
    - Adjust the number of convolution layers.
    - Adjust the number of fully connected layers.
    - Adjust the learning rates and other training details (initialization, epochs, etc.)
3. Try out the improved network on the original MNIST dataset.
4. Display the activations of the first and second layer of LeNet for different inputs (e.g., sweaters, coats).

7 Modern Convolutional Neural Networks

Now that we understand the basics of wiring together convolutional neural networks, we will take you through a tour of modern deep learning. In this chapter, each section will correspond to a significant neural network architecture that was at some point (or currently) the base model upon which an enormous amount of research and projects were built. Each of these networks was at briefly a dominant architecture and many were at one point winners or runners-up in the famous ImageNet competition, which has served as a barometer of progress on supervised learning in computer vision since 2010.

These models include AlexNet, the first large-scale network deployed to beat conventional computer vision methods on a large-scale vision challenge; the VGG network, which makes use of a number of repeating blocks of elements; the network in network (NiN) which convolves whole neural networks patch-wise over inputs; the GoogLeNet, which makes use of networks with parallel concatenations (GoogLeNet); residual networks (ResNet) which are currently the most popular go-to architecture today, and densely connected networks (DenseNet), which are expensive to compute but have set some recent benchmarks.

7.1 Deep Convolutional Neural Networks (AlexNet)

Although convolutional neural networks were well known in the computer vision and machine learning communities following the introduction of LeNet, they did not immediately dominate the field. Although LeNet achieved good results on early small datasets, the performance and feasability of training convolutional networks on larger, more realistic datasets had yet to be established In fact, for much of the intervening time between the early 1990s and the watershed results of 2012, neural networks were often surpassed by other machine learning methods, such as support vector machines.

For computer vision, this comparison is perhaps not fair. That is although the inputs to convolutional networks consist of raw or lightly-processed (e.g., by centering) pixel values, practitioners would never feed raw pixels into traditional models. Instead, typical computer vision pipelines consisted of manually engineering feature extraction pipelines. Rather than *learn the features*, the features were *crafted*. Most of the progress came from having more clever ideas for features, and the learning algorithm was often relegated to an afterthought.

Although some neural network accelerators were available in the 1990s, they were not yet sufficiently powerful to make deep multichannel, multilayer convolutional neural networks with a large number of parameters. Moreover, datasets were still relatively small. Added to these obstacles, key tricks for training neural networks including parameter initialization heuristics, clever variants of stochastic gradient descent, non-squashing activation functions, and effective regularization techniques were still missing.

Thus, rather than training *end-to-end* (pixel to classification) systems, classical pipelines looked more like this:

1. Obtain an interesting dataset. In early days, these datasets required expensive sensors (at the time, 1 megapixel images were state of the art).

2. Preprocess the dataset with hand-crafted features based on some knowledge of optics, geometry, other analytic tools, and occasionally on the serendipitous discoveries of lucky graduate students.

3. Feed the data through a standard set of feature extractors such as SIFT[104], the Scale-Invariant Feature Transform, or SURF[105], the Speeded-Up Robust Features, or any number of other hand-tuned pipelines.

4. Dummp the resulting representations into your favorite classifier, likely a linear model or kernel method, to learn a classifier.

If you spoke to machine learning researchers, they believed that machine learning was both important and beautiful. Elegant theories proved the properties of various classifiers. The field of machine learning was thriving, rigorous and eminently useful. However, if you spoke to a computer vision researcher, you'd hear a very different story. The dirty truth of image recognition, they'd tell you, is that features, not learning algorithms, drove progress. Computer vision researchers justifiably believed that a slightly bigger or cleaner dataset or a slightly improved feature-extraction pipeline mattered far more to the final accuracy than any learning algorithm.

7.1.1 Learning Feature Representation

Another way to cast the state of affairs is that the most important part of the pipeline was the representation. And up until 2012 the representation was calculated mechanically. In fact, engineering a new set of feature functions, improving results, and writing up the method was a prominent genre of paper. SIFT[106], SURF[107], HOG[108], Bags of visual words[109] and similar feature extractors ruled the roost.

Another group of researchers, including Yann LeCun, Geoff Hinton, Yoshua Bengio, Andrew Ng, Shun-ichi Amari, and Juergen Schmidhuber, had different plans. They believed that features themselves ought to be learned. Moreover, they believed that to be reasonably complex, the features ought to be hierarchically composed with multiple jointly learned layers, each with learnable parameters. In the case of an image, the lowest layers might come to detect edges, colors, and textures. Indeed, (Krizhevsky et al., 2012) proposed a new variant of a convolutional neural network which achieved excellent performance in the ImageNet challenge.

Interestingly in the lowest layers of the network, the model learned feature extractors that resembled some traditional filters. Fig. 7.1.1 is reproduced from this paper and describes lower-level image descriptors.

[104] https://en.wikipedia.org/wiki/Scale-invariant_feature_transform
[105] https://en.wikipedia.org/wiki/Speeded_up_robust_features
[106] https://en.wikipedia.org/wiki/Scale-invariant_feature_transform
[107] https://en.wikipedia.org/wiki/Speeded_up_robust_features
[108] https://en.wikipedia.org/wiki/Histogram_of_oriented_gradients
[109] https://en.wikipedia.org/wiki/Bag-of-words_model_in_computer_vision

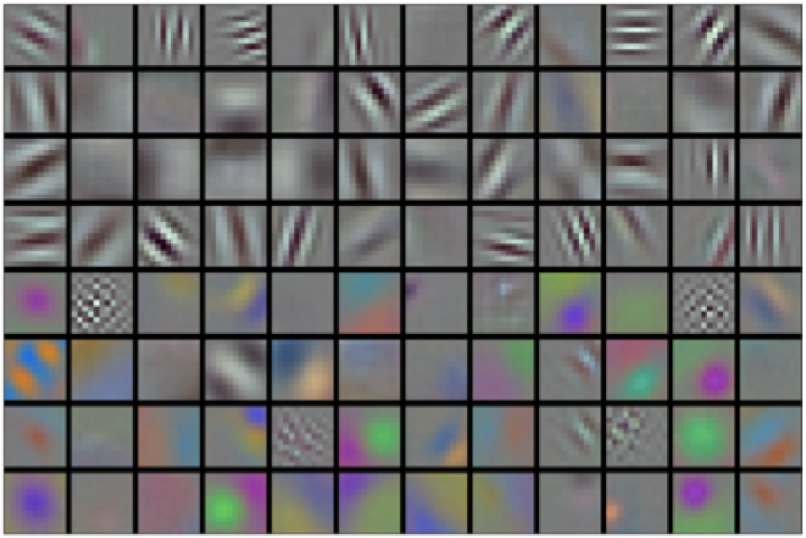


Fig. 7.1.1: Image filters learned by the first layer of AlexNet

Higher layers in the network might build upon these representations to represent larger structures, like eyes, noses, blades of grass, etc. Even higher layers might represent whole objects like people, airplanes, dogs, or frisbees. Ultimately, the final hidden state learns a compact representation of the image that summarizes its contents such that data belonging to different categories be separated easily.

While the ultimate breakthrough for many-layered convolutional networks came in 2012, a core group of researchers had dedicated themselves to this idea, attempting to learn hierarchical representations of visual data for many years. The ultimate breakthrough in 2012 can be attributed to two key factors.

Missing Ingredient - Data

Deep models with many layers require large amounts of data in order to enter the regime where they signficantly outperform traditional methods based on convex optimizations (e.g., linear and kernel methods). However, given the limited storage capacity of computers, the relative expense of sensors, and the comparatively tighter research budgets in the 1990s, most research relied on tiny datasets. Numerous papers addressed the UCI collection of datasets, many of which contained only hundreds or (a few) thousands of images captured in unnatural settings with low resolution.

In 2009, the ImageNet dataset was released, challenging researchers to learn models from 1 million examples, 1,000 each from 1,000 distinct categories of objects. The researchers, led by Fei-Fei Li, who introduced this dataset leveraged Google Image Search to prefilter large candidate sets for each category and employed the Amazon Mechanical Turk crowdsourcing pipeline to confirm for each image whether it belonged to the associated category. This scale was unprecedented. The associated competition, dubbed the ImageNet Challenge pushed computer vision and machine learning research forward, challenging researchers to identify which models performed best at a greater scale than academics had previously considered.

Missing Ingredient - Hardware

Deep learning models are varacious consumers of compute cycles. Training can take hundreds of epochs, and each iteration requires passing data through many layers of computationally-expensive linear algebra operations. This is one of the main reasons why in the 90s and early 2000s, simple algorithms based on the more-efficiently optimized convex objectives were preferred.

Graphical processing units (GPUs) proved to be a game changer in make deep learning feasible. These chips had long been developed for accelerating graphics processing to benefit computer games. In particular, they were optimized for high throughput 4x4 matrix-vector products, which are needed for many computer graphics tasks. Fortunately, this math is strikingly similar to that required to calculate convolutional layers. Around that time, NVIDIA and ATI had begun optimizing GPUs for general compute operations, going as far as to market them as General Purpose GPUs (GPGPU).

To provide some intuition, consider the cores of a modern microprocessor (CPU). Each of the cores is fairly powerful running at a high clock frequency and sporting large caches (up to several MB of L3). Each core is well-suited to executing a wide range of instructions, with branch predictors, a deep pipeline, and other bells and whistles that enable it to run a large variety of programs. This apparent strength, however, is also its Achilles heel: general purpose cores are very expensive to build. They require lots of chip area, a sophisticated support structure (memory interfaces, caching logic between cores, high speed interconnects, etc.), and they are comparatively bad at any single task. Modern laptops have up to 4 cores, and even high end servers rarely exceed 64 cores, simply because it is not cost effective.

By comparison, GPUs consist of 100-1000 small processing elements (the details differ somewhat between NVIDIA, ATI, ARM and other chip vendors), often grouped into larger groups (NVIDIA calls them warps). While each core is relatively weak, sometimes even running at sub-1GHz clock frequency, it is the total number of such cores that makes GPUs orders of magnitude faster than CPUs. For instance, NVIDIA's latest Volta generation offers up to 120 TFlops per chip for specialized instructions (and up to 24 TFlops for more general purpose ones), while floating point performance of CPUs has not exceeded 1 TFlop to date. The reason for why this is possible is actually quite simple: first, power consumption tends to grow *quadratically* with clock frequency. Hence, for the power budget of a CPU core that runs 4x faster (a typical number), you can use 16 GPU cores at 1/4 the speed, which yields 16 x 1/4 = 4x the performance. Furthermore, GPU cores are much simpler (in fact, for a long time they weren't even *able* to execute general purpose code), which makes them more energy efficient. Last, many operations in deep learning require high memory bandwidth. Again, GPUs shine here with buses that are at least 10x as wide as many CPUs.

Back to 2012. A major breakthrough came when Alex Krizhevsky and Ilya Sutskever implemented a deep convolutional neural network that could run on GPU hardware. They realized that the computational bottlenecks in CNNs (convolutions and matrix multiplications) are all operations that could be parallelized in hardware. Using two NIVIDA GTX 580s with 3GB of memory, they implemented fast convolutions. The code cuda-convnet[110] was good enough that for several years it was the industry standard and powered the first couple years of the deep learning boom.

[110] https://code.google.com/archive/p/cuda-convnet/

7.1.2 AlexNet

AlexNet was introduced in 2012, named after Alex Krizhevsky, the first author of the breakthrough ImageNet classification paper (Krizhevsky et al., 2012). AlexNet, which employed an 8-layer convolutional neural network, won the ImageNet Large Scale Visual Recognition Challenge 2012 by a phenomenally large margin. This network proved, for the first time, that the features obtained by learning can transcend manually-design features, breaking the previous paradigm in computer vision. The architectures of AlexNet and LeNet are *very similar*, as Fig. 7.1.2 illustrates. Note that we provide a slightly streamlined version of AlexNet removing some of the design quirks that were needed in 2012 to make the model fit on two small GPUs.

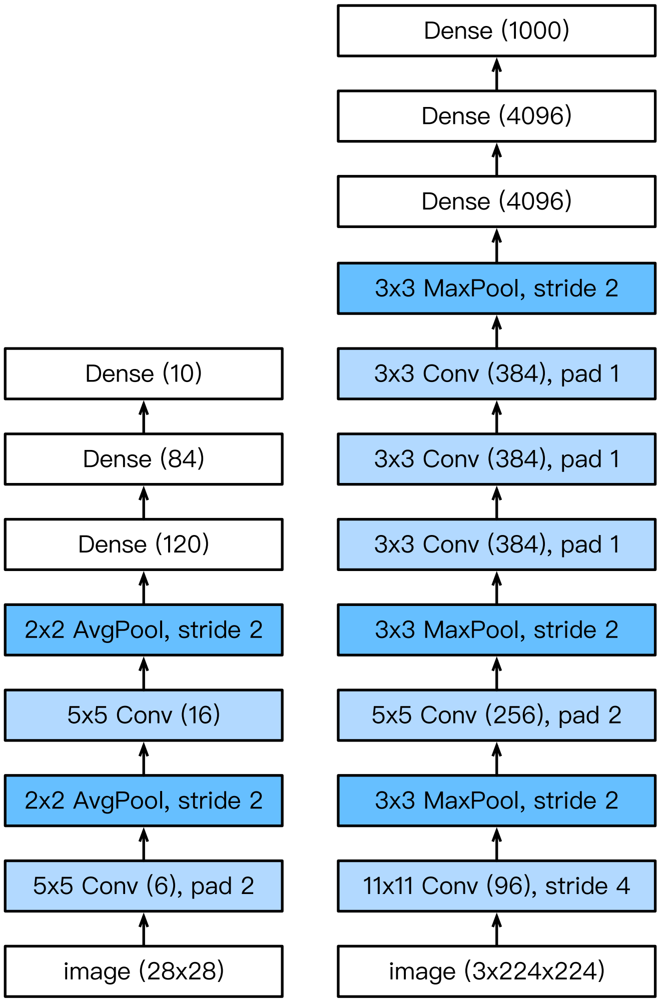


Fig. 7.1.2: LeNet (left) and AlexNet (right)

The design philosophies of AlexNet and LeNet are very similar, but there are also significant differences. First, AlexNet is much deeper than the comparatively small LeNet5. AlexNet consists of eight layers: five convolutional layers, two fully-connected hidden layers, and one fully-connected output layer. Second, AlexNet used the ReLU instead of the sigmoid as its activation function. Let's delve into the details below.

Architecture

In AlexNet's first layer, the convolution window shape is 11×11. Since most images in ImageNet are more than ten times higher and wider than the MNIST images, objects in ImageNet data tend to occupy more pixels. Consequently, a larger convolution window is needed to capture the object. The convolution window shape in the second layer is reduced to 5×5, followed by 3×3. In addition, after the first, second, and fifth convolutional layers, the network adds maximum pooling layers with a window shape of 3×3 and a stride of 2. Moreover, AlexNet has ten times more convolution channels than LeNet.

After the last convolutional layer are two fully-connected layers with 4096 outputs. These two huge fully-connected layers produce model parameters of nearly 1 GB. Due to the limited memory in early GPUs, the original AlexNet used a dual data stream design, so that each of their two GPUs could be responsible for storing and computing only its half of the model. Fortunately, GPU memory is comparatively abundant now, so we rarely need to break up models across GPUs these days (our version of the AlexNet model deviates from the original paper in this aspect).

Activation Functions

Second, AlexNet changed the sigmoid activation function to a simpler ReLU activation function. On the one hand, the computation of the ReLU activation function is simpler. For example, it does not have the exponentiation operation found in the sigmoid activation function. On the other hand, the ReLU activation function makes model training easier when using different parameter initialization methods. This is because, when the output of the sigmoid activation function is very close to 0 or 1, the gradient of these regions is almost 0, so that back propagation cannot continue to update some of the model parameters. In contrast, the gradient of the ReLU activation function in the positive interval is always 1. Therefore, if the model parameters are not properly initialized, the sigmoid function may obtain a gradient of almost 0 in the positive interval, so that the model cannot be effectively trained.

Capacity Control and Preprocessing

AlexNet controls the model complexity of the fully-connected layer by dropout (Section 4.6), while LeNet only uses weight decay. To augment the data even further, the training loop of AlexNet added a great deal of image augmentation, such as flipping, clipping, and color changes. This makes the model more robust and the larger sample size effectively reduces overfitting. We will discuss data augmentation in greater detail in Section 13.1.

```
import d2l
from mxnet import np, npx
from mxnet.gluon import nn
npx.set_np()

net = nn.Sequential()
# Here, we use a larger 11 x 11 window to capture objects. At the same time,
# we use a stride of 4 to greatly reduce the height and width of the output.
# Here, the number of output channels is much larger than that in LeNet
net.add(nn.Conv2D(96, kernel_size=11, strides=4, activation='relu'),
        nn.MaxPool2D(pool_size=3, strides=2),
        # Make the convolution window smaller, set padding to 2 for consistent
```

(continues on next page)

(continued from previous page)

```
        # height and width across the input and output, and increase the
        # number of output channels
        nn.Conv2D(256, kernel_size=5, padding=2, activation='relu'),
        nn.MaxPool2D(pool_size=3, strides=2),
        # Use three successive convolutional layers and a smaller convolution
        # window. Except for the final convolutional layer, the number of
        # output channels is further increased. Pooling layers are not used to
        # reduce the height and width of input after the first two
        # convolutional layers
        nn.Conv2D(384, kernel_size=3, padding=1, activation='relu'),
        nn.Conv2D(384, kernel_size=3, padding=1, activation='relu'),
        nn.Conv2D(256, kernel_size=3, padding=1, activation='relu'),
        nn.MaxPool2D(pool_size=3, strides=2),
        # Here, the number of outputs of the fully connected layer is several
        # times larger than that in LeNet. Use the dropout layer to mitigate
        # overfitting
        nn.Dense(4096, activation="relu"), nn.Dropout(0.5),
        nn.Dense(4096, activation="relu"), nn.Dropout(0.5),
        # Output layer. Since we are using Fashion-MNIST, the number of
        # classes is 10, instead of 1000 as in the paper
        nn.Dense(10))
```

We construct a single-channel data instance with both height and width of 224 to observe the output shape of each layer. It matches our diagram above.

```
X = np.random.uniform(size=(1, 1, 224, 224))
net.initialize()
for layer in net:
    X = layer(X)
    print(layer.name, 'output shape:\t', X.shape)
```

```
conv0 output shape:  (1, 96, 54, 54)
pool0 output shape:  (1, 96, 26, 26)
conv1 output shape:  (1, 256, 26, 26)
pool1 output shape:  (1, 256, 12, 12)
conv2 output shape:  (1, 384, 12, 12)
conv3 output shape:  (1, 384, 12, 12)
conv4 output shape:  (1, 256, 12, 12)
pool2 output shape:  (1, 256, 5, 5)
dense0 output shape:         (1, 4096)
dropout0 output shape:       (1, 4096)
dense1 output shape:         (1, 4096)
dropout1 output shape:       (1, 4096)
dense2 output shape:         (1, 10)
```

7.1.3 Reading the Dataset

Although AlexNet uses ImageNet in the paper, we use Fashion-MNIST here since training an ImageNet model to convergence could take hours or days even on a modern GPU. One of the problems with applying AlexNet directly on Fashion-MNIST is that our images are lower resolution (28×28 pixels) than ImageNet images. To make things work, we upsample them to 244×244 (generally not a smart practice, but we do it here to be faithful to the AlexNet architecture). We perform this resizing with the `resize` argument in `load_data_fashion_mnist`.

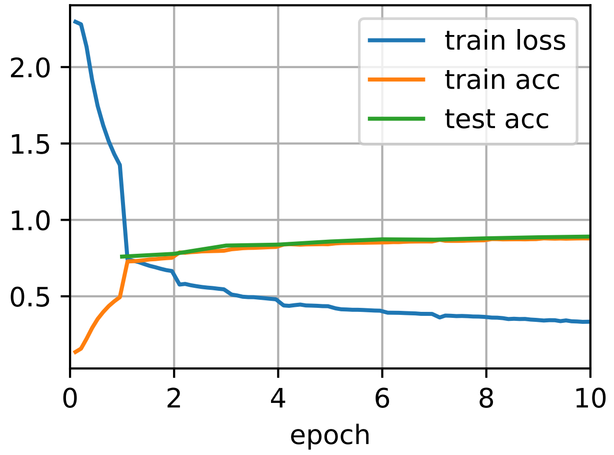
```
batch_size = 128
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size, resize=224)
```

7.1.4 Training

Now, we can start training AlexNet. Compared to LeNet in the previous section, the main change here is the use of a smaller learning rate and much slower training due to the deeper and wider network, the higher image resolution and the more costly convolutions.

```
lr, num_epochs = 0.01, 10
d2l.train_ch5(net, train_iter, test_iter, num_epochs, lr)
```

```
loss 0.333, train acc 0.879, test acc 0.891
4206.3 exampes/sec on gpu(0)
```



Summary

- AlexNet has a similar structure to that of LeNet, but uses more convolutional layers and a larger parameter space to fit the large-scale dataset ImageNet.
- Today AlexNet has been surpassed by much more effective architectures but it is a key step from shallow to deep networks that are used nowadays.
- Although it seems that there are only a few more lines in AlexNet's implementation than in LeNet, it took the academic community many years to embrace this conceptual change and

take advantage of its excellent experimental results. This was also due to the lack of efficient computational tools.

- Dropout, ReLU and preprocessing were the other key steps in achieving excellent performance in computer vision tasks.

Exercises

1. Try increasing the number of epochs. Compared with LeNet, how are the results different? Why?
2. AlexNet may be too complex for the Fashion-MNIST dataset.
    - Try to simplify the model to make the training faster, while ensuring that the accuracy does not drop significantly.
    - Can you design a better model that works directly on 28×28 images.
3. Modify the batch size, and observe the changes in accuracy and GPU memory.
4. Rooflines:
    - What is the dominant part for the memory footprint of AlexNet?
    - What is the dominant part for computation in AlexNet?
    - How about memory bandwidth when computing the results?
5. Apply dropout and ReLU to LeNet5. Does it improve? How about preprocessing?

7.2 Networks Using Blocks (VGG)

While AlexNet proved that deep convolutional neural networks can achieve good results, it did not offer a general template to guide subsequent researchers in designing new networks. In the following sections, we will introduce several heuristic concepts commonly used to design deep networks.

Progress in this field mirrors that in chip design where engineers went from placing transistors to logical elements to logic blocks. Similarly, the design of neural network architectures had grown progressively more abstract, with researchers moving from thinking in terms of individual neurons to whole layers, and now to blocks, repeating patterns of layers.

The idea of using blocks first emerged from the Visual Geometry Group[112] (VGG) at Oxford University. In their eponymously-named VGG network, It is easy to implement these repeated structures in code with any modern deep learning framework by using loops and subroutines.

[112] http://www.robots.ox.ac.uk/~vgg/

7.2.1 VGG Blocks

The basic building block of classic convolutional networks is a sequence of the following layers: (i) a convolutional layer (with padding to maintain the resolution), (ii) a nonlinearity such as a ReLU, One VGG block consists of a sequence of convolutional layers, followed by a max pooling layer for spatial downsampling. In the original VGG paper (Simonyan & Zisserman, 2014), the authors employed convolutions with 3×3 kernels and 2×2 max pooling with stride of 2 (halving the resolution after each block). In the code below, we define a function called `vgg_block` to implement one VGG block. The function takes two arguments corresponding to the number of convolutional layers `num_convs` and the number of output channels `num_channels`.

```python
import d2l
from mxnet import np, npx
from mxnet.gluon import nn
npx.set_np()

def vgg_block(num_convs, num_channels):
    blk = nn.Sequential()
    for _ in range(num_convs):
        blk.add(nn.Conv2D(num_channels, kernel_size=3,
                          padding=1, activation='relu'))
    blk.add(nn.MaxPool2D(pool_size=2, strides=2))
    return blk
```

7.2.2 VGG Network

Like AlexNet and LeNet, the VGG Network can be partitioned into two parts: the first consisting mostly of convolutional and pooling layers and a second consisting of fully-connected layers. The convolutional portion of the net connects several `vgg_block` modules in succession. In Fig. 7.2.1, the variable `conv_arch` consists of a list of tuples (one per block), where each contains two values: the number of convolutional layers and the number of output channels, which are precisely the arguments requires to call the `vgg_block` function. The fully-connected module is identical to that covered in AlexNet.

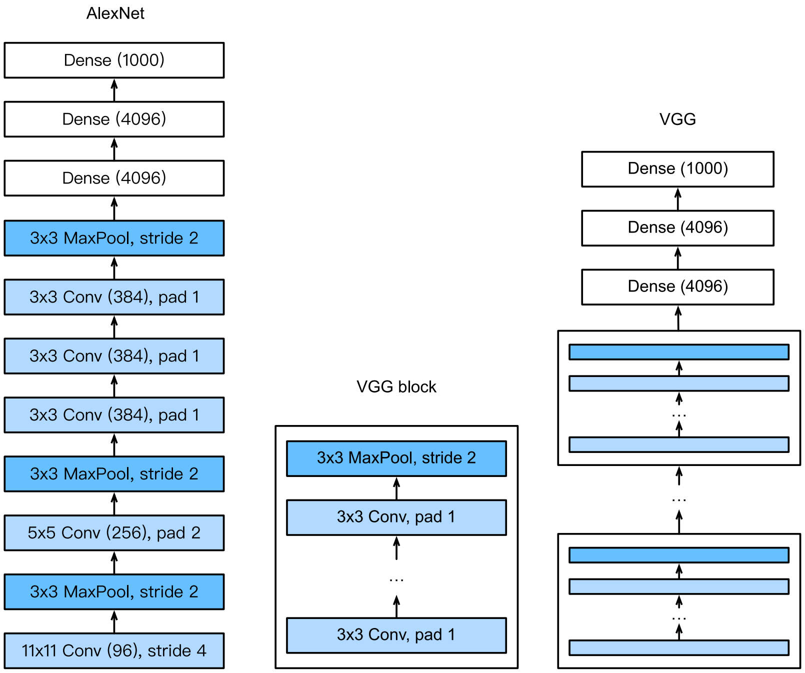


Fig. 7.2.1: Designing a network from building blocks

The original VGG network had 5 convolutional blocks, among which the first two have one convolutional layer each and the latter three contain two convolutional layers each. The first block has 64 output channels and each subsequent block doubles the number of output channels, until that number reaches 512. Since this network uses 8 convolutional layers and 3 fully-connected layers, it is often called VGG-11.

```
conv_arch = ((1, 64), (1, 128), (2, 256), (2, 512), (2, 512))
```

The following code implements VGG-11. This is a simple matter of executing a for loop over conv_arch.

```
def vgg(conv_arch):
    net = nn.Sequential()
    # The convolutional layer part
    for (num_convs, num_channels) in conv_arch:
        net.add(vgg_block(num_convs, num_channels))
    # The fully connected layer part
    net.add(nn.Dense(4096, activation='relu'), nn.Dropout(0.5),
            nn.Dense(4096, activation='relu'), nn.Dropout(0.5),
            nn.Dense(10))
    return net

net = vgg(conv_arch)
```

Next, we will construct a single-channel data example with a height and width of 224 to observe the output shape of each layer.

```
net.initialize()
X = np.random.uniform(size=(1, 1, 224, 224))
for blk in net:
```

(continues on next page)

(continued from previous page)

```
    X = blk(X)
    print(blk.name, 'output shape:\t', X.shape)
```

```
sequential1 output shape:    (1, 64, 112, 112)
sequential2 output shape:    (1, 128, 56, 56)
sequential3 output shape:    (1, 256, 28, 28)
sequential4 output shape:    (1, 512, 14, 14)
sequential5 output shape:    (1, 512, 7, 7)
dense0 output shape:         (1, 4096)
dropout0 output shape:       (1, 4096)
dense1 output shape:         (1, 4096)
dropout1 output shape:       (1, 4096)
dense2 output shape:         (1, 10)
```

As you can see, we halve height and width at each block, finally reaching a height and width of 7 before flattening the representations for processing by the fully-connected layer.

7.2.3 Model Training

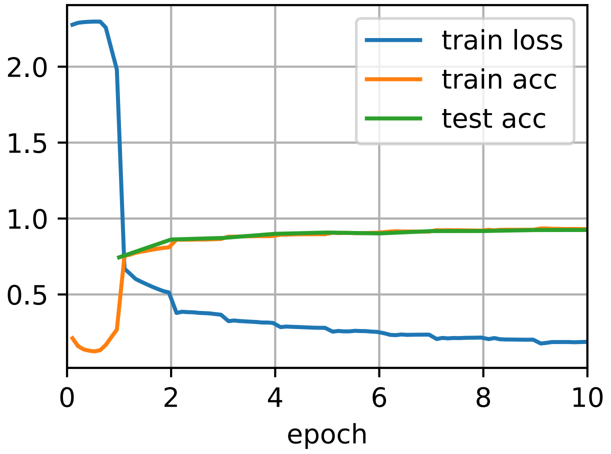
Since VGG-11 is more computationally-heavy than AlexNet we construct a network with a smaller number of channels. This is more than sufficient for training on Fashion-MNIST.

```
ratio = 4
small_conv_arch = [(pair[0], pair[1] // ratio) for pair in conv_arch]
net = vgg(small_conv_arch)
```

Apart from using a slightly larger learning rate, the model training process is similar to that of AlexNet in the last section.

```
lr, num_epochs, batch_size = 0.05, 10, 128,
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size, resize=224)
d2l.train_ch5(net, train_iter, test_iter, num_epochs, lr)
```

```
loss 0.186, train acc 0.931, test acc 0.925
1815.7 exampes/sec on gpu(0)
```



Summary

- VGG-11 constructs a network using reusable convolutional blocks. Different VGG models can be defined by the differences in the number of convolutional layers and output channels in each block.
- The use of blocks leads to very compact representations of the network definition. It allows for efficient design of complex networks.
- In their work Simonyan and Ziserman experimented with various architectures. In particular, they found that several layers of deep and narrow convolutions (i.e., 3×3) were more effective than fewer layers of wider convolutions.

Exercises

1. When printing out the dimensions of the layers we only saw 8 results rather than 11. Where did the remaining 3 layer informations go?
2. Compared with AlexNet, VGG is much slower in terms of computation, and it also needs more GPU memory. Try to analyze the reasons for this.
3. Try to change the height and width of the images in Fashion-MNIST from 224 to 96. What influence does this have on the experiments?
4. Refer to Table 1 in (Simonyan & Zisserman, 2014) to construct other common models, such as VGG-16 or VGG-19.

7.3 Network in Network (NiN)

LeNet, AlexNet, and VGG all share a common design pattern: extract features exploiting *spatial* structure via a sequence of convolutions and pooling layers and then post-process the representations via fully-connected layers. The improvements upon LeNet by AlexNet and VGG mainly lie in how these later networks widen and deepen these two modules. Alternatively, one could imagine using fully-connected layers earlier in the process. However, a careless use of dense layers might give up the spatial structure of the representation entirely, Network in Network (NiN) blocks offer an alternative. They were proposed in (Lin et al., 2013) based on a very simple insight—to use an MLP on the channels for each pixel separately.

7.3.1 NiN Blocks

Recall that the inputs and outputs of convolutional layers consist of four-dimensional arrays with axes corresponding to the batch, channel, height, and width. Also recall that the inputs and outputs of fully-connected layers are typically two-dimensional arrays corresponding to the batch, and features. The idea behind NiN is to apply a fully-connected layer at each pixel location (for each height and width). If we tie the weights across each spatial location, we could think of this as a 1×1 convolutional layer (as described in Section 6.4) or as a fully-connected layer acting independently on each pixel location. Another way to view this is to think of each element in the spatial dimension (height and width) as equivalent to an example and the channel as equivalent to a feature. Fig. 7.3.1 illustrates the main structural differences between NiN and AlexNet, VGG, and other networks.

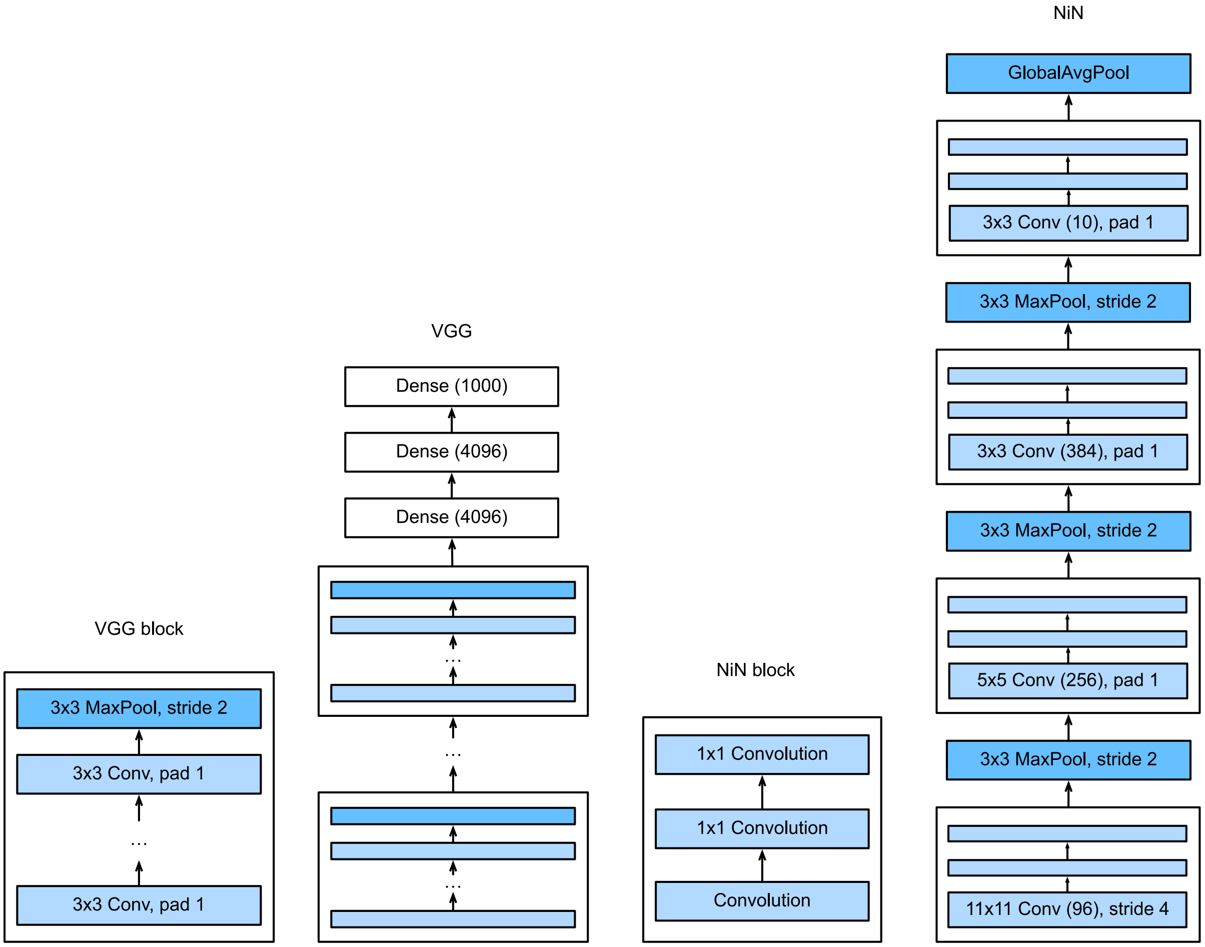


Fig. 7.3.1: The figure on the left shows the network structure of AlexNet and VGG, and the figure on the right shows the network structure of NiN.

The NiN block consists of one convolutional layer followed by two 1×1 convolutional layers that act as per-pixel fully-connected layers with ReLU activations. The convolution width of the first layer is typically set by the user. The subsequent widths are fixed to 1×1.

```
import d2l
from mxnet import np, npx
from mxnet.gluon import nn
```

(continues on next page)

(continued from previous page)

```
npx.set_np()

def nin_block(num_channels, kernel_size, strides, padding):
    blk = nn.Sequential()
    blk.add(nn.Conv2D(num_channels, kernel_size, strides, padding,
                      activation='relu'),
            nn.Conv2D(num_channels, kernel_size=1, activation='relu'),
            nn.Conv2D(num_channels, kernel_size=1, activation='relu'))
    return blk
```

7.3.2 NiN Model

The original NiN network was proposed shortly after AlexNet and clearly draws some inspiration. NiN uses convolutional layers with window shapes of 11×11, 5×5, and 3×3, and the corresponding numbers of output channels are the same as in AlexNet. Each NiN block is followed by a maximum pooling layer with a stride of 2 and a window shape of 3×3.

Once significant difference between NiN and AlexNet is that NiN avoids dense connections altogether. Instead, NiN uses an NiN block with a number of output channels equal to the number of label classes, followed by a *global* average pooling layer, yielding a vector of logits[114]. One advantage of NiN's design is that it significantly reduces the number of required model parameters. However, in practice, this design sometimes requires increased model training time.

```
net = nn.Sequential()
net.add(nin_block(96, kernel_size=11, strides=4, padding=0),
        nn.MaxPool2D(pool_size=3, strides=2),
        nin_block(256, kernel_size=5, strides=1, padding=2),
        nn.MaxPool2D(pool_size=3, strides=2),
        nin_block(384, kernel_size=3, strides=1, padding=1),
        nn.MaxPool2D(pool_size=3, strides=2),
        nn.Dropout(0.5),
        # There are 10 label classes
        nin_block(10, kernel_size=3, strides=1, padding=1),
        # The global average pooling layer automatically sets the window shape
        # to the height and width of the input
        nn.GlobalAvgPool2D(),
        # Transform the four-dimensional output into two-dimensional output
        # with a shape of (batch size, 10)
        nn.Flatten())
```

We create a data example to see the output shape of each block.

```
X = np.random.uniform(size=(1, 1, 224, 224))
net.initialize()
for layer in net:
    X = layer(X)
    print(layer.name, 'output shape:\t', X.shape)
```

[114] https://en.wikipedia.org/wiki/Logit

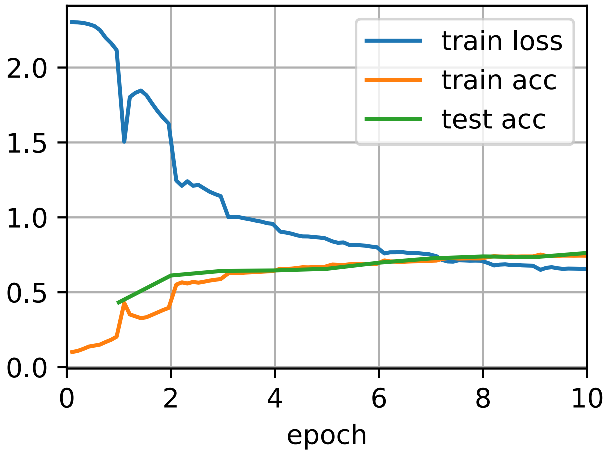
```
sequential1 output shape:    (1, 96, 54, 54)
pool0 output shape:  (1, 96, 26, 26)
sequential2 output shape:    (1, 256, 26, 26)
pool1 output shape:  (1, 256, 12, 12)
sequential3 output shape:    (1, 384, 12, 12)
pool2 output shape:  (1, 384, 5, 5)
dropout0 output shape:       (1, 384, 5, 5)
sequential4 output shape:    (1, 10, 5, 5)
pool3 output shape:  (1, 10, 1, 1)
flatten0 output shape:       (1, 10)
```

7.3.3 Data Acquisition and Training

As before we use Fashion-MNIST to train the model. NiN's training is similar to that for AlexNet and VGG, but it often uses a larger learning rate.

```
lr, num_epochs, batch_size = 0.1, 10, 128
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size, resize=224)
d2l.train_ch5(net, train_iter, test_iter, num_epochs, lr)
```

```
loss 0.657, train acc 0.744, test acc 0.762
3024.6 exampes/sec on gpu(0)
```



Summary

- NiN uses blocks consisting of a convolutional layer and multiple 1×1 convolutional layer. This can be used within the convolutional stack to allow for more per-pixel nonlinearity.
- NiN removes the fully connected layers and replaces them with global average pooling (i.e., summing over all locations) after reducing the number of channels to the desired number of outputs (e.g., 10 for Fashion-MNIST).
- Removing the dense layers reduces overfitting. NiN has dramatically fewer parameters.
- The NiN design influenced many subsequent convolutional neural networks designs.

Exercises

1. Tune the hyper-parameters to improve the classification accuracy.
2. Why are there two 1×1 convolutional layers in the NiN block? Remove one of them, and then observe and analyze the experimental phenomena.
3. Calculate the resource usage for NiN
    - What is the number of parameters?
    - What is the amount of computation?
    - What is the amount of memory needed during training?
    - What is the amount of memory needed during inference?
4. What are possible problems with reducing the $384 \times 5 \times 5$ representation to a $10 \times 5 \times 5$ representation in one step?

7.4 Networks with Parallel Concatenations (GoogLeNet)

In 2014, (Szegedy et al., 2015) won the ImageNet Challenge, proposing a structure that combined the strengths of the NiN and repeated blocks paradigms. One focus of the paper was to address the question of which sized convolutional kernels are best. After all, previous popular networks employed choices as small as 1×1 and as large as 11×11. One insight in this paper was that sometimes it can be advantageous to employ a combination of variously-sized kernels. In this section, we will introduce GoogLeNet, presenting a slightly simplified version of the original model—we omit a few ad hoc features that were added to stabilize training but are unnecessary now with better training algorithms available.

7.4.1 Inception Blocks

The basic convolutional block in GoogLeNet is called an Inception block, likely named due to a quote from the movie Inception ("We Need To Go Deeper"), which launched a viral meme.

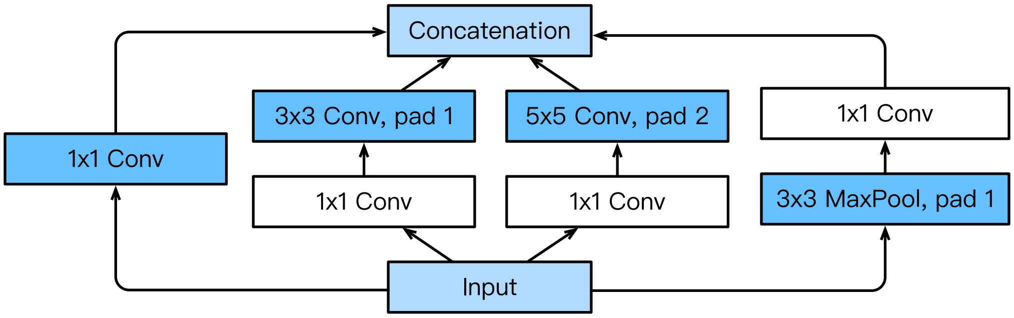


Fig. 7.4.1: Structure of the Inception block.

As depicted in the figure above, the inception block consists of four parallel paths. The first three paths use convolutional layers with window sizes of 1×1, 3×3, and 5×5 to extract information from different spatial sizes. The middle two paths perform a 1×1 convolution on the input to reduce the number of input channels, reducing the model's complexity. The fourth path uses a 3×3 maximum pooling layer, followed by a 1×1 convolutional layer to change the number of channels. The four paths all use appropriate padding to give the input and output the same height and width. Finally, the outputs along each path are concatenated along the channel dimension and comprise the block's output. The commonly-tuned parameters of the Inception block are the number of output channels per layer.

```
import d2l
from mxnet import np, npx
from mxnet.gluon import nn
npx.set_np()

class Inception(nn.Block):
    # c1 - c4 are the number of output channels for each layer in the path
    def __init__(self, c1, c2, c3, c4, **kwargs):
        super(Inception, self).__init__(**kwargs)
        # Path 1 is a single 1 x 1 convolutional layer
        self.p1_1 = nn.Conv2D(c1, kernel_size=1, activation='relu')
        # Path 2 is a 1 x 1 convolutional layer followed by a 3 x 3
        # convolutional layer
        self.p2_1 = nn.Conv2D(c2[0], kernel_size=1, activation='relu')
        self.p2_2 = nn.Conv2D(c2[1], kernel_size=3, padding=1,
                              activation='relu')
        # Path 3 is a 1 x 1 convolutional layer followed by a 5 x 5
        # convolutional layer
        self.p3_1 = nn.Conv2D(c3[0], kernel_size=1, activation='relu')
        self.p3_2 = nn.Conv2D(c3[1], kernel_size=5, padding=2,
                              activation='relu')
        # Path 4 is a 3 x 3 maximum pooling layer followed by a 1 x 1
        # convolutional layer
        self.p4_1 = nn.MaxPool2D(pool_size=3, strides=1, padding=1)
        self.p4_2 = nn.Conv2D(c4, kernel_size=1, activation='relu')

    def forward(self, x):
        p1 = self.p1_1(x)
        p2 = self.p2_2(self.p2_1(x))
        p3 = self.p3_2(self.p3_1(x))
        p4 = self.p4_2(self.p4_1(x))
        # Concatenate the outputs on the channel dimension
        return np.concatenate((p1, p2, p3, p4), axis=1)
```

To gain some intuition for why this network works so well, consider the combination of the filters. They explore the image in varying ranges. This means that details at different extents can be recognized efficiently by different filters. At the same time, we can allocate different amounts of parameters for different ranges (e.g., more for short range but not ignore the long range entirely).

7.4.2 GoogLeNet Model

As shown in Fig. 7.4.2, GoogLeNet uses a stack of a total of 9 inception blocks and global average pooling to generate its estimates. Maximum pooling between inception blocks reduced the dimensionality. The first part is identical to AlexNet and LeNet, the stack of blocks is inherited from VGG and the global average pooling avoids a stack of fully-connected layers at the end. The architecture is depicted below.

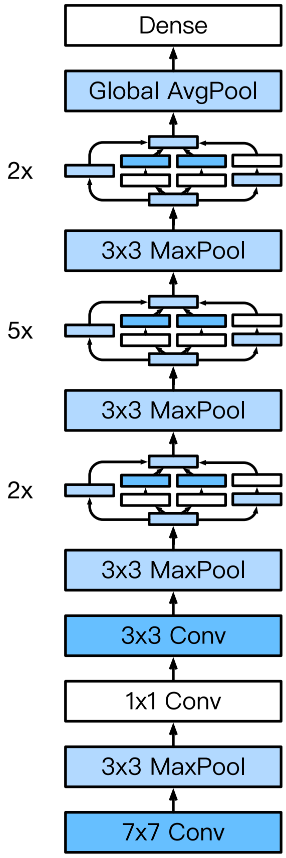


Fig. 7.4.2: Full GoogLeNet Model

We can now implement GoogLeNet piece by piece. The first component uses a 64-channel 7×7 convolutional layer.

```
b1 = nn.Sequential()
b1.add(nn.Conv2D(64, kernel_size=7, strides=2, padding=3, activation='relu'),
       nn.MaxPool2D(pool_size=3, strides=2, padding=1))
```

The second component uses two convolutional layers: first, a 64-channel 1×1 convolutional layer, then a 3×3 convolutional layer that triples the number of channels. This corresponds to the second path in the Inception block.

```
b2 = nn.Sequential()
b2.add(nn.Conv2D(64, kernel_size=1, activation='relu'),
       nn.Conv2D(192, kernel_size=3, padding=1, activation='relu'),
       nn.MaxPool2D(pool_size=3, strides=2, padding=1))
```

The third component connects two complete Inception blocks in series. The number of output channels of the first Inception block is $64 + 128 + 32 + 32 = 256$, and the ratio to the output channels of the four paths is $64 : 128 : 32 : 32 = 2 : 4 : 1 : 1$. The second and third paths first reduce the number of input channels to $96/192 = 1/2$ and $16/192 = 1/12$, respectively, and then connect the second convolutional layer. The number of output channels of the second Inception block is increased to $128 + 192 + 96 + 64 = 480$, and the ratio to the number of output channels per path is $128 : 192 : 96 : 64 = 4 : 6 : 3 : 2$. The second and third paths first reduce the number of input channels to $128/256 = 1/2$ and $32/256 = 1/8$, respectively.

```
b3 = nn.Sequential()
b3.add(Inception(64, (96, 128), (16, 32), 32),
       Inception(128, (128, 192), (32, 96), 64),
       nn.MaxPool2D(pool_size=3, strides=2, padding=1))
```

The fourth block is more complicated. It connects five Inception blocks in series, and they have $192+208+48+64 = 512$, $160+224+64+64 = 512$, $128+256+64+64 = 512$, $112+288+64+64 = 528$, and $256+320+128+128 = 832$ output channels, respectively. The number of channels assigned to these paths is similar to that in the third module: the second path with the 3×3 convolutional layer outputs the largest number of channels, followed by the first path with only the 1×1 convolutional layer, the third path with the 5×5 convolutional layer, and the fourth path with the 3×3 maximum pooling layer. The second and third paths will first reduce the number of channels according the ratio. These ratios are slightly different in different Inception blocks.

```
b4 = nn.Sequential()
b4.add(Inception(192, (96, 208), (16, 48), 64),
       Inception(160, (112, 224), (24, 64), 64),
       Inception(128, (128, 256), (24, 64), 64),
       Inception(112, (144, 288), (32, 64), 64),
       Inception(256, (160, 320), (32, 128), 128),
       nn.MaxPool2D(pool_size=3, strides=2, padding=1))
```

The fifth block has two Inception blocks with $256+320+128+128 = 832$ and $384+384+128+128 = 1024$ output channels. The number of channels assigned to each path is the same as that in the third and fourth modules, but differs in specific values. It should be noted that the fifth block is followed by the output layer. This block uses the global average pooling layer to change the height and width of each channel to 1, just as in NiN. Finally, we turn the output into a two-dimensional array followed by a fully-connected layer whose number of outputs is the number of label classes.

```
b5 = nn.Sequential()
b5.add(Inception(256, (160, 320), (32, 128), 128),
       Inception(384, (192, 384), (48, 128), 128),
       nn.GlobalAvgPool2D())

net = nn.Sequential()
net.add(b1, b2, b3, b4, b5, nn.Dense(10))
```

The GoogLeNet model is computationally complex, so it is not as easy to modify the number of channels as in VGG. To have a reasonable training time on Fashion-MNIST, we reduce the input height and width from 224 to 96. This simplifies the computation. The changes in the shape of the output between the various modules is demonstrated below.

```
X = np.random.uniform(size=(1, 1, 96, 96))
net.initialize()
for layer in net:
    X = layer(X)
    print(layer.name, 'output shape:\t', X.shape)
```

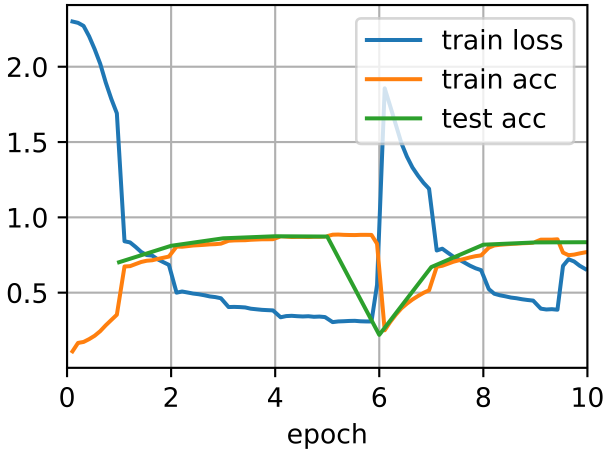
```
sequential0 output shape:	 (1, 64, 24, 24)
sequential1 output shape:	 (1, 192, 12, 12)
sequential2 output shape:	 (1, 480, 6, 6)
sequential3 output shape:	 (1, 832, 3, 3)
sequential4 output shape:	 (1, 1024, 1, 1)
dense0 output shape:	 (1, 10)
```

7.4.3 Data Acquisition and Training

As before, we train our model using the Fashion-MNIST dataset. We transform it to 96×96 pixel resolution before invoking the training procedure.

```
lr, num_epochs, batch_size = 0.1, 10, 128
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size, resize=96)
d2l.train_ch5(net, train_iter, test_iter, num_epochs, lr)
```

```
loss 0.648, train acc 0.770, test acc 0.835
2525.2 exampes/sec on gpu(0)
```



Summary

- The Inception block is equivalent to a subnetwork with four paths. It extracts information in parallel through convolutional layers of different window shapes and maximum pooling layers. 1×1 convolutions reduce channel dimensionality on a per-pixel level. Max-pooling reduces the resolution.
- GoogLeNet connects multiple well-designed Inception blocks with other layers in series. The ratio of the number of channels assigned in the Inception block is obtained through a large number of experiments on the ImageNet dataset.

- GoogLeNet, as well as its succeeding versions, was one of the most efficient models on ImageNet, providing similar test accuracy with lower computational complexity.

Exercises

1. There are several iterations of GoogLeNet. Try to implement and run them. Some of them include the following:
    - Add a batch normalization layer (Ioffe & Szegedy, 2015), as described later in Section 7.5.
    - Make adjustments to the Inception block (Szegedy et al., 2016).
    - Use "label smoothing" for model regularization (Szegedy et al., 2016).
    - Include it in the residual connection (Szegedy et al., 2017), as described later in Section 7.6.
2. What is the minimum image size for GoogLeNet to work?
3. Compare the model parameter sizes of AlexNet, VGG, and NiN with GoogLeNet. How do the latter two network architectures significantly reduce the model parameter size?
4. Why do we need a large range convolution initially?

7.5 Batch Normalization

Training deep neural nets is difficult. And getting them to converge in a reasonable amount of time can be tricky.

In this section, we describe batch normalization (BN) (Ioffe & Szegedy, 2015), a popular and effective technique that consistently accelerates the convergence of deep nets. Together with residual blocks—covered in Section 7.6—BN has made it possible for practitioners to routinely train networks with over 100 layers.

7.5.1 Training Deep Networks

To motivate batch normalization, let's review a few practical challenges that arise when training ML models and neural nets in particular.

1. Choices regarding data preprocessing often make an enormous difference in the final results. Recall our application of multilayer perceptrons to predicting house prices (Section 4.10). Our first step when working with real data was to standardize our input features to each have a mean of *zero* and variance of *one*. Intuitively, this standardization plays nicely with our optimizers because it puts the parameters are a-priori at a similar scale.
2. For a typical MLP or CNN, as we train, the activations in intermediate layers may take values with widely varying magnitudes—both along the layers from the input to the output, across

nodes in the same layer, and over time due to our updates to the model's parameters. The inventors of batch normalization postulated informally that this drift in the distribution of activations could hamper the convergence of the network. Intuitively, we might conjecture that if one layer has activation values that are 100x that of another layer, this might necessitate compensatory adjustments in the learning rates.

3. Deeper networks are complex and easily capable of overfitting. This means that regularization becomes more critical.

Batch normalization is applied to individual layers (optionally, to all of them) and works as follows: In each training iteration, for each layer, we first compute its activations as usual. Then, we normalize the activations of each node by subtracting its mean and dividing by its standard deviation estimating both quantities based on the statistics of the current the current minibatch. It is precisely due to this *normalization* based on *batch* statistics that *batch normalization* derives its name.

Note that if we tried to apply BN with minibatches of size 1, we would not be able to learn anything. That is because after subtracting the means, each hidden node would take value 0! As you might guess, since we are devoting a whole section to BN, with large enough minibatches, the approach proves effective and stable. One takeaway here is that when applying BN, the choice of minibatch size may be even more significant than without BN.

Formally, BN transforms the activations at a given layer $\mathbf{x}$ according to the following expression:

$$\mathrm{BN}(\mathbf{x}) = \gamma \odot \frac{\mathbf{x} - \hat{\mu}}{\hat\sigma} + \beta \tag{7.5.1}$$

Here, $\hat{\mu}$ is the minibatch sample mean and $\hat{\sigma}$ is the minibatch sample variance. After applying BN, the resulting minibatch of activations has zero mean and unit variance. Because the choice of unit variance (vs some other magic number) is an arbitrary choice, we commonly include coordinate-wise scaling coefficients γ and offsets β. Consequently, the activation magnitudes for intermediate layers cannot diverge during training because BN actively centers and rescales them back to a given mean and size (via μ and σ). One piece of practitioner's intuition/wisdom is that BN seems to allows for more aggressive learning rates.

Formally, denoting a particular minibatch by $\mathcal{B}$, we calculate $\hat{\mu}_\mathcal{B}$ and $\hat\sigma_\mathcal{B}$ as follows:

$$\hat{\mu}_\mathcal{B} \leftarrow \frac{1}{|\mathcal{B}|} \sum_{\mathbf{x} \in \mathcal{B}} \mathbf{x} \text{ and } \hat{\sigma}_\mathcal{B}^2 \leftarrow \frac{1}{|\mathcal{B}|} \sum_{\mathbf{x} \in \mathcal{B}} (\mathbf{x} - \mu_{\mathcal{B}})^2 + \epsilon \tag{7.5.2}$$

Note that we add a small constant $\epsilon > 0$ to the variance estimate to ensure that we never attempt division by zero, even in cases where the empirical variance estimate might vanish. The estimates $\hat{\mu}_\mathcal{B}$ and $\hat{\sigma}_\mathcal{B}$ counteract the scaling issue by using noisy estimates of mean and variance. You might think that this noisiness should be a problem. As it turns out, this is actually beneficial.

This turns out to be a recurring theme in deep learning. For reasons that are not yet well-characterized theoretically, various sources of noise in optimization often lead to faster training and less overfitting. While traditional machine learning theorists might buckle at this characterization, this variation appears to act as a form of regularization. In some preliminary research, (Teye et al., 2018) and (Luo et al., 2018) relate the properties of BN to Bayesian Priors and penalties respectively. In particular, this sheds some light on the puzzle of why BN works best for moderate minibatches sizes in the 50–100 range.

Fixing a trained model, you might (rightly) think that we would prefer to use the entire dataset to estimate the mean and variance. Once training is complete, why would we want the same image to be classified differently, depending on the batch in which it happens to reside? During training, such exact calculation is infeasible because the activations for all data points change every time we update our model. However, once the model is trained, we can calculate the means and variances of each layer's activations based on the entire dataset. Indeed this is standard practice for models employing batch normalization and thus MXNet's BN layers function differently in *training mode* (normalizing by minibatch statistics) and in *prediction mode* (normalizing by dataset statistics).

We are now ready to take a look at how batch normalization works in practice.

7.5.2 Batch Normalization Layers

Batch normalization implementations for fully-connected layers and convolutional layers are slightly different. We discuss both cases below. Recall that one key differences between BN and other layers is that because BN operates on a full minibatch at a time, we cannot just ignore the batch dimension as we did before when introducing other layers.

Fully-Connected Layers

When applying BN to fully-connected layers, we usually insert BN after the affine transformation and before the nonlinear activation function. Denoting the input to the layer by $\mathbf{x}$, the linear transform (with weights θ) by $f_\theta(\cdot)$, the activation function by $\phi(\cdot)$, and the BN operation with parameters β and γ by $\mathrm{BN}_{\beta,\gamma}$, we can express the computation of a BN-enabled, fully-connected layer $\mathbf{h}$ as follows:

$$\mathbf{h} = \phi(\mathrm{BN}_{\beta,\gamma}(f_\theta(\mathbf{x}))) \tag{7.5.3}$$

Recall that mean and variance are computed on the *same* minibatch $\mathcal{B}$ on which the transformation is applied. Also recall that the scaling coefficient γ and the offset β are parameters that need to be learned jointly with the more familiar parameters θ.

Convolutional Layers

Similarly, with convolutional layers, we typically apply BN after the convolution and before the nonlinear activation function. When the convolution has multiple output channels, we need to carry out batch normalization for *each* of the outputs of these channels, and each channel has its own scale and shift parameters, both of which are scalars. Assume that our minibatches contain m each and that for each channel, the output of the convolution has height p and width q. For convolutional layers, we carry out each batch normalization over the $m \cdot p \cdot q$ elements per output channel simultaneously. Thus we collect the values over all spatial locations when computing the mean and variance and consequently (within a given channel) apply the same $\hat{\mu}$ and $\hat{\sigma}$ to normalize the values at each spatial location.

Batch Normalization During Prediction

As we mentioned earlier, BN typically behaves differently in training mode and prediction mode. First, the noise in μ and σ arising from estimating each on minibatches are no longer desirable once we have trained the model. Second, we might not have the luxury of computing per-batch normalization statistics, e.g., we might need to apply our model to make one prediction at a time.

Typically, after training, we use the entire dataset to compute stable estimates of the activation statistics and then fix them at prediction time. Consequently, BN behaves differently during training and at test time. Recall that dropout also exhibits this characteristic.

7.5.3 Implementation from Scratch

Below, we implement a batch normalization layer with `ndarray`s from scratch:

```
import d2l
from mxnet import autograd, np, npx, init
from mxnet.gluon import nn
npx.set_np()

def batch_norm(X, gamma, beta, moving_mean, moving_var, eps, momentum):
    # Use autograd to determine whether the current mode is training mode or
    # prediction mode
    if not autograd.is_training():
        # If it is the prediction mode, directly use the mean and variance
        # obtained from the incoming moving average
        X_hat = (X - moving_mean) / np.sqrt(moving_var + eps)
    else:
        assert len(X.shape) in (2, 4)
        if len(X.shape) == 2:
            # When using a fully connected layer, calculate the mean and
            # variance on the feature dimension
            mean = X.mean(axis=0)
            var = ((X - mean) ** 2).mean(axis=0)
        else:
            # When using a two-dimensional convolutional layer, calculate the
            # mean and variance on the channel dimension (axis=1). Here we
            # need to maintain the shape of X, so that the broadcast operation
            # can be carried out later
            mean = X.mean(axis=(0, 2, 3), keepdims=True)
            var = ((X - mean) ** 2).mean(axis=(0, 2, 3), keepdims=True)
        # In training mode, the current mean and variance are used for the
        # standardization
        X_hat = (X - mean) / np.sqrt(var + eps)
        # Update the mean and variance of the moving average
        moving_mean = momentum * moving_mean + (1.0 - momentum) * mean
        moving_var = momentum * moving_var + (1.0 - momentum) * var
    Y = gamma * X_hat + beta  # Scale and shift
    return Y, moving_mean, moving_var
```

We can now create a proper `BatchNorm` layer. Our layer will maintain poper parameters corresponding for scale `gamma` and shift `beta`, both of which will be updated in the course of training. Additionally, our layer will maintain a moving average of the means and variances for subsequent use during model prediction. The `num_features` parameter required by the `BatchNorm` instance is the number of outputs for a fully-connected layer and the number of output channels for a convolutional layer. The num_dims parameter also required by this instance is 2 for a fully-connected layer and 4 for a convolutional layer.

Putting aside the algorithmic details, note the design pattern underlying our implementation of the layer. Typically, we define the math in a separate function, say batch_norm. We then integrate this functionality into a custom layer, whose code mostly addresses bookkeeping matters, such as moving data to the right device context, allocating and initializing any required variables, keeping track of running averages (here for mean and variance), etc. This pattern enables a clean separation of math from boilerplate code. Also note that for the sake of convenience we did not worry about automatically inferring the input shape here, thus our need to specify the number of features throughout. Do not worry, the Gluon BatchNorm layer will care of this for us.

```
class BatchNorm(nn.Block):
    def __init__(self, num_features, num_dims, **kwargs):
        super(BatchNorm, self).__init__(**kwargs)
        if num_dims == 2:
            shape = (1, num_features)
        else:
            shape = (1, num_features, 1, 1)
        # The scale parameter and the shift parameter involved in gradient
        # finding and iteration are initialized to 0 and 1 respectively
        self.gamma = self.params.get('gamma', shape=shape, init=init.One())
        self.beta = self.params.get('beta', shape=shape, init=init.Zero())
        # All the variables not involved in gradient finding and iteration are
        # initialized to 0 on the CPU
        self.moving_mean = np.zeros(shape)
        self.moving_var = np.zeros(shape)

    def forward(self, X):
        # If X is not on the CPU, copy moving_mean and moving_var to the
        # device where X is located
        if self.moving_mean.context != X.context:
            self.moving_mean = self.moving_mean.copyto(X.context)
            self.moving_var = self.moving_var.copyto(X.context)
        # Save the updated moving_mean and moving_var
        Y, self.moving_mean, self.moving_var = batch_norm(
            X, self.gamma.data(), self.beta.data(), self.moving_mean,
            self.moving_var, eps=1e-5, momentum=0.9)
        return Y
```

7.5.4 Using a Batch Normalization LeNet

To see how to apply BatchNorm in context, below we apply it to a traditional LeNet model (Section 6.6). Recall that BN is typically applied after the convolutional layers and fully-connected layers but before the corresponding activation functions.

```
net = nn.Sequential()
net.add(nn.Conv2D(6, kernel_size=5),
        BatchNorm(6, num_dims=4),
        nn.Activation('sigmoid'),
        nn.MaxPool2D(pool_size=2, strides=2),
        nn.Conv2D(16, kernel_size=5),
```

(continues on next page)

(continued from previous page)

```
        BatchNorm(16, num_dims=4),
        nn.Activation('sigmoid'),
        nn.MaxPool2D(pool_size=2, strides=2),
        nn.Dense(120),
        BatchNorm(120, num_dims=2),
        nn.Activation('sigmoid'),
        nn.Dense(84),
        BatchNorm(84, num_dims=2),
        nn.Activation('sigmoid'),
        nn.Dense(10))
```

As before, we will train our network on the Fashion-MNIST dataset. This code is virtually identical to that when we first trained LeNet (Section 6.6). The main difference is the considerably larger learning rate.

```
lr, num_epochs, batch_size = 1.0, 10, 256
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)
d2l.train_ch5(net, train_iter, test_iter, num_epochs, lr)
```

Let's have a look at the scale parameter `gamma` and the shift parameter `beta` learned from the first batch normalization layer.

```
net[1].gamma.data().reshape(-1,), net[1].beta.data().reshape(-1,)
```

7.5.5 Concise Implementation

Compared with the `BatchNorm` class, which we just defined ourselves, the `BatchNorm` class defined by the `nn` model in Gluon is easier to use. In Gluon, we do not have to worry about `num_features` or `num_dims`. Instead, these parameter values will be inferred automatically via delayed initialization. Otherwise, the code looks virtually identical to the application our implementation above.

```
net = nn.Sequential()
net.add(nn.Conv2D(6, kernel_size=5),
        nn.BatchNorm(),
        nn.Activation('sigmoid'),
        nn.MaxPool2D(pool_size=2, strides=2),
        nn.Conv2D(16, kernel_size=5),
        nn.BatchNorm(),
        nn.Activation('sigmoid'),
        nn.MaxPool2D(pool_size=2, strides=2),
        nn.Dense(120),
        nn.BatchNorm(),
        nn.Activation('sigmoid'),
        nn.Dense(84),
        nn.BatchNorm(),
        nn.Activation('sigmoid'),
        nn.Dense(10))
```

Below, we use the same hyper-parameters to train out model. Note that as usual, the Gluon variant runs much faster because its code has been compiled to C++/CUDA while our custom implementation must be interpreted by Python.

```
d2l.train_ch5(net, train_iter, test_iter, num_epochs, lr)
```

7.5.6 Controversy

Intuitively, batch normalization is thought to make the optimization landscape smoother. However, we must be careful to distinguish between speculative intuitions and true explanations for the phenomena that we observe when training deep models. Recall that we do not even know why simpler deep neural networks (MLPs and conventional CNNs) generalize well in the first place. Even with dropout and L2 regularization, they remain so flexible that their ability to generalize to unseen data cannot be explained via conventional learning-theoretic generalization guarantees.

In the original paper proposing batch normalization, the authors, in addition to introducing a powerful and useful tool, offered an explanation for why it works: by reducing *internal covariate shift*. Presumably by *internal covariate shift* the authors meant something like the intuition expressed above—the notion that the distribution of activations changes over the course of training. However there were two problems with this explanation: (1) This drift is very different from *covariate shift*, rendering the name a misnomer. (2) The explanation offers an under-specified intuition but leaves the question of *why precisely this technique works* an open question wanting for a rigorous explanation. Throughout this book, we aim to convey the intuitions that practitioners use to guide their development of deep neural networks. However, we believe that it is important to separate these guiding intuitions from established scientific fact. Eventually, when you master this material and start writing your own research papers you will want to be clear to delineate between technical claims and hunches.

Following the success of batch normalization, its explanation in terms of *internal covariate shift* has repeatedly surfaced in debates in the technical literature and broader discourse about how to present machine learning research. In a memorable speech given while accepting a Test of Time Award at the 2017 NeurIPS conference, Ali Rahimi used *internal covariate shift* as a focal point in an argument likening the modern practice of deep learning to alchemy. Subsequently, the example was revisited in detail in a position paper outlining troubling trends in machine learning (Lipton & Steinhardt, 2018).
In the technical literature other authors ((Santurkar et al., 2018)) have proposed alternative explanations for the success of BN, some claiming that BN's success comes despite exhibiting behavior that is in some ways opposite to those claimed in the original paper.

We note that the *internal covariate shift* is no more worthy of criticism than any of thousands of similarly vague claims made every year in the technical ML literature. Likely, its resonance as a focal point of these debates owes to its broad recognizability to the target audience. Batch normalization has proven an indispensable method, applied in nearly all deployed image classifiers, earning the paper that introduced the technique tens of thousands of citations.

Summary

- During model training, batch normalization continuously adjusts the intermediate output of the neural network by utilizing the mean and standard deviation of the minibatch, so that the values of the intermediate output in each layer throughout the neural network are more stable.
- The batch normalization methods for fully connected layers and convolutional layers are slightly different.
- Like a dropout layer, batch normalization layers have different computation results in training mode and prediction mode.
- Batch Normalization has many beneficial side effects, primarily that of regularization. On the other hand, the original motivation of reducing covariate shift seems not to be a valid explanation.

Exercises

1. Can we remove the fully connected affine transformation before the batch normalization or the bias parameter in convolution computation?
    - Find an equivalent transformation that applies prior to the fully connected layer.
    - Is this reformulation effective. Why (not)?
2. Compare the learning rates for LeNet with and without batch normalization.
    - Plot the decrease in training and test error.
    - What about the region of convergence? How large can you make the learning rate?
3. Do we need Batch Normalization in every layer? Experiment with it?
4. Can you replace Dropout by Batch Normalization? How does the behavior change?
5. Fix the coefficients `beta` and `gamma` (add the parameter `grad_req='null'` at the time of construction to avoid calculating the gradient), and observe and analyze the results.
6. Review the Gluon documentation for `BatchNorm` to see the other applications for Batch Normalization.
7. Research ideas: think of other normalization transforms that you can apply? Can you apply the probability integral transform? How about a full rank covariance estimate?

7.6 Residual Networks (ResNet)

As we design increasingly deeper networks it becomes imperative to understand how adding layers can increase the complexity and expressiveness of the network. Even more important is the ability to design networks where adding layers makes networks strictly more expressive rather than just different. To make some progress we need a bit of theory.

7.6.1 Function Classes

Consider $\mathcal{F}$, the class of functions that a specific network architecture (together with learning rates and other hyperparameter settings) can reach. That is, for all $f \in \mathcal{F}$ there exists some set of parameters W that can be obtained through training on a suitable dataset. Let's assume that f^* is the function that we really would like to find. If it is in $\mathcal{F}$, we are in good shape but typically we will not be quite so lucky. Instead, we will try to find some $f^*_\mathcal{F}$ which is our best bet within $\mathcal{F}$. For instance, we might try finding it by solving the following optimization problem:

$$f^*_\mathcal{F} := \mathop{\mathrm{argmin}}_f L(X, Y, f) \text{ subject to } f \in \mathcal{F}. \tag{7.6.1}$$

It is only reasonable to assume that if we design a different and more powerful architecture $\mathcal{F}'$ we should arrive at a better outcome. In other words, we would expect that $f^*_{\mathcal{F}'}$ is "better" than $f^*_\mathcal{F}$. However, if $\mathcal{F} \not\subseteq \mathcal{F}'$ there is no guarantee that this should even happen. In fact, $f^*_{\mathcal{F}'}$ might well be worse. This is a situation that we often encounter in practice—adding layers does not only make the network more expressive, it also changes it in sometimes not quite so predictable ways. Fig. 7.6.1illustrates this in slightly abstract terms.

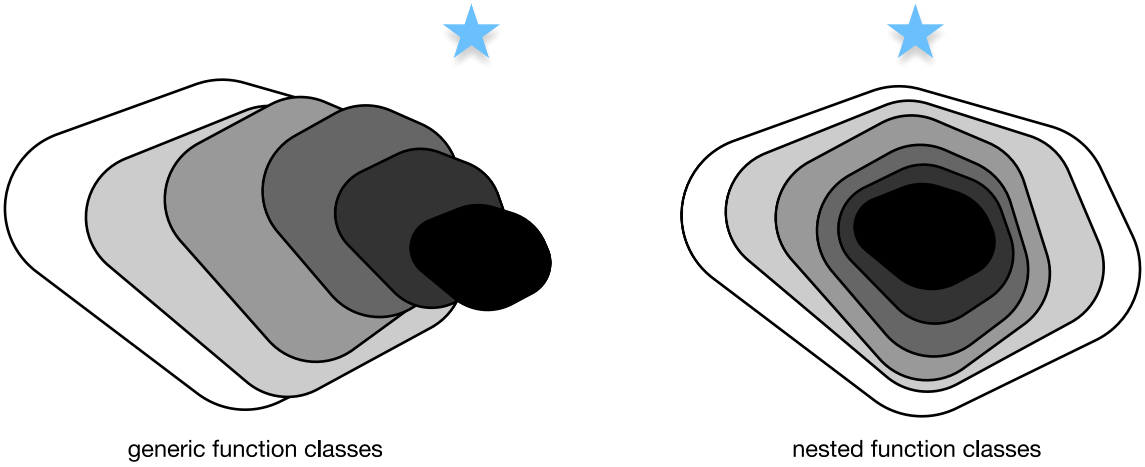


Fig. 7.6.1: Left: non-nested function classes. The distance may in fact increase as the complexity increases. Right: with nested function classes this does not happen.

Only if larger function classes contain the smaller ones are we guaranteed that increasing them strictly increases the expressive power of the network. This is the question that He et al, 2016 considered when working on very deep computer vision models. At the heart of ResNet is the idea that every additional layer should contain the identity function as one of its elements. This means that if we can train the newly-added layer into an identity mapping $f(\mathbf{x}) = \mathbf{x}$, the new model will be as effective as the original model. As the new model may get a better solution to fit the training dataset, the added layer might make it easier to reduce training errors. Even better, the identity function rather than the null $f(\mathbf{x}) = 0$ should be the the simplest function within a layer.

These considerations are rather profound but they led to a surprisingly simple solution, a residual block. With it, (He et al., 2016a) won the ImageNet Visual Recognition Challenge in 2015. The design had a profound influence on how to build deep neural networks.

7.6.2 Residual Blocks

Let's focus on a local neural network, as depicted below. Denote the input by $\mathbf{x}$. We assume that the ideal mapping we want to obtain by learning is $f(\mathbf{x})$, to be used as the input to the activation function. The portion within the dotted-line box in the left image must directly fit the mapping $f(\mathbf{x})$. This can be tricky if we do not need that particular layer and we would much rather retain the input $\mathbf{x}$. The portion within the dotted-line box in the right image now only needs to parametrize the *deviation* from the identity, since we return $\mathbf{x} + f(\mathbf{x})$. In practice, the residual mapping is often easier to optimize. We only need to set $f(\mathbf{x}) = 0$. The right image in Fig. 7.6.2 illustrates the basic Residual Block of ResNet. Similar architectures were later proposed for sequence models which we will study later.

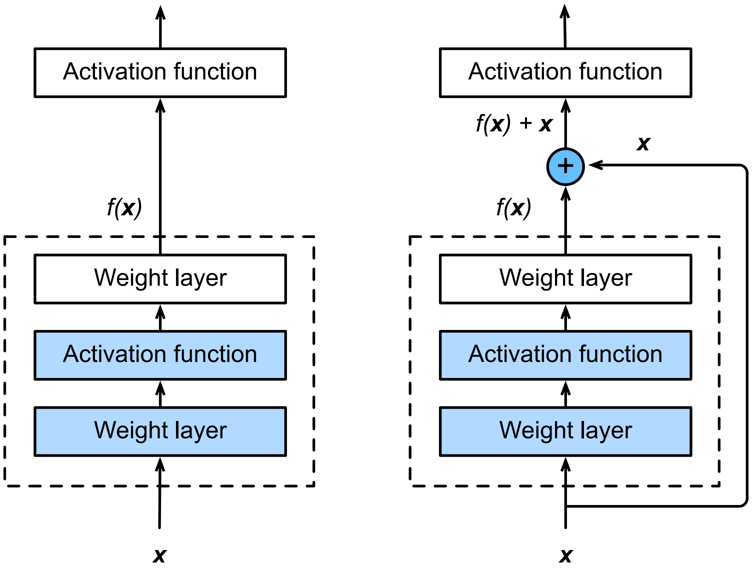


Fig. 7.6.2: The difference between a regular block (left) and a residual block (right). In the latter case, we can short-circuit the convolutions.

ResNet follows VGG's full 3×3 convolutional layer design. The residual block has two 3×3 convolutional layers with the same number of output channels. Each convolutional layer is followed by a batch normalization layer and a ReLU activation function. Then, we skip these two convolution operations and add the input directly before the final ReLU activation function. This kind of design requires that the output of the two convolutional layers be of the same shape as the input, so that they can be added together. If we want to change the number of channels or the the stride, we need to introduce an additional 1×1 convolutional layer to transform the input into the desired shape for the addition operation. Let's have a look at the code below.

```
import d2l
from mxnet import np, npx
from mxnet.gluon import nn
npx.set_np()

# Saved in the d2l package for later use
class Residual(nn.Block):
    def __init__(self, num_channels, use_1x1conv=False, strides=1, **kwargs):
        super(Residual, self).__init__(**kwargs)
        self.conv1 = nn.Conv2D(num_channels, kernel_size=3, padding=1,
                               strides=strides)
        self.conv2 = nn.Conv2D(num_channels, kernel_size=3, padding=1)
        if use_1x1conv:
            self.conv3 = nn.Conv2D(num_channels, kernel_size=1,
                                   strides=strides)
        else:
            self.conv3 = None
        self.bn1 = nn.BatchNorm()
        self.bn2 = nn.BatchNorm()

    def forward(self, X):
        Y = npx.relu(self.bn1(self.conv1(X)))
        Y = self.bn2(self.conv2(Y))
        if self.conv3:
            X = self.conv3(X)
        return npx.relu(Y + X)
```

This code generates two types of networks: one where we add the input to the output before applying the ReLU nonlinearity, and whenever `use_1x1conv=True`, one where we adjust channels and resolution by means of a 1×1 convolution before adding. Fig. 7.6.3 illustrates this:

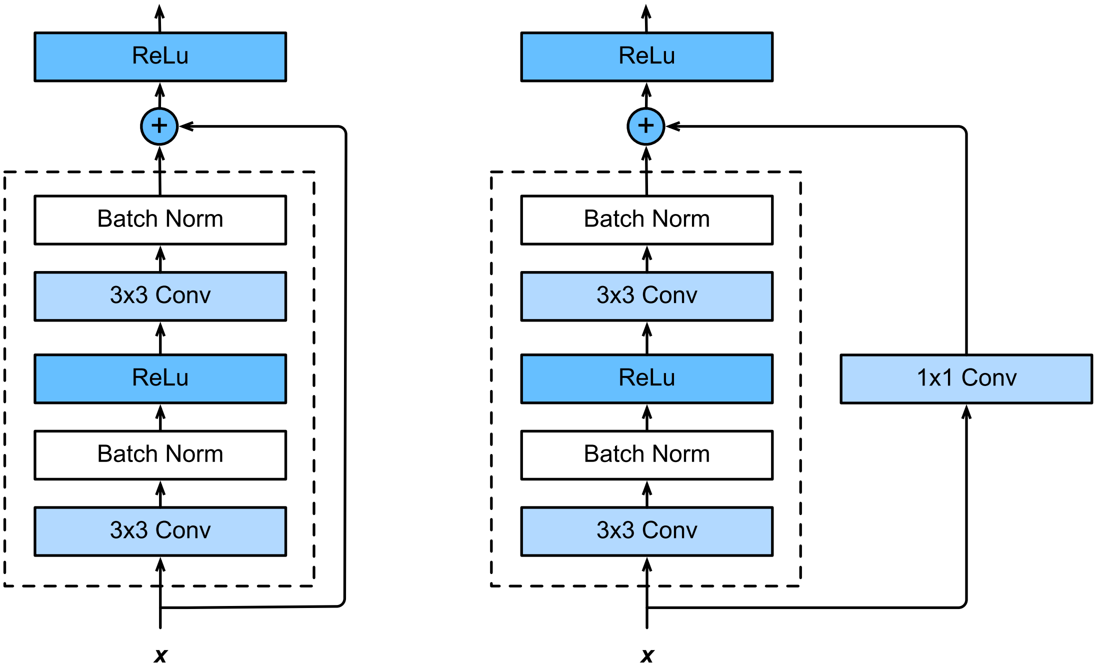


Fig. 7.6.3: Left: regular ResNet block; Right: ResNet block with 1x1 convolution

Now let's look at a situation where the input and output are of the same shape.

```
blk = Residual(3)
blk.initialize()
X = np.random.uniform(size=(4, 3, 6, 6))
blk(X).shape
```

```
(4, 3, 6, 6)
```

We also have the option to halve the output height and width while increasing the number of output channels.

```
blk = Residual(6, use_1x1conv=True, strides=2)
blk.initialize()
blk(X).shape
```

```
(4, 6, 3, 3)
```

7.6.3 ResNet Model

The first two layers of ResNet are the same as those of the GoogLeNet we described before: the 7×7 convolutional layer with 64 output channels and a stride of 2 is followed by the 3×3 maximum pooling layer with a stride of 2. The difference is the batch normalization layer added after each convolutional layer in ResNet.

```
net = nn.Sequential()
net.add(nn.Conv2D(64, kernel_size=7, strides=2, padding=3),
        nn.BatchNorm(), nn.Activation('relu'),
        nn.MaxPool2D(pool_size=3, strides=2, padding=1))
```

GoogLeNet uses four blocks made up of Inception blocks. However, ResNet uses four modules made up of residual blocks, each of which uses several residual blocks with the same number of output channels. The number of channels in the first module is the same as the number of input channels. Since a maximum pooling layer with a stride of 2 has already been used, it is not necessary to reduce the height and width. In the first residual block for each of the subsequent modules, the number of channels is doubled compared with that of the previous module, and the height and width are halved.

Now, we implement this module. Note that special processing has been performed on the first module.

```
def resnet_block(num_channels, num_residuals, first_block=False):
    blk = nn.Sequential()
    for i in range(num_residuals):
        if i == 0 and not first_block:
            blk.add(Residual(num_channels, use_1x1conv=True, strides=2))
        else:
            blk.add(Residual(num_channels))
    return blk
```

Then, we add all the residual blocks to ResNet. Here, two residual blocks are used for each module.

```
net.add(resnet_block(64, 2, first_block=True),
        resnet_block(128, 2),
        resnet_block(256, 2),
        resnet_block(512, 2))
```

Finally, just like GoogLeNet, we add a global average pooling layer, followed by the fully connected layer output.

```
net.add(nn.GlobalAvgPool2D(), nn.Dense(10))
```

There are 4 convolutional layers in each module (excluding the 1×1 convolutional layer). Together with the first convolutional layer and the final fully connected layer, there are 18 layers in total. Therefore, this model is commonly known as ResNet-18. By configuring different numbers of channels and residual blocks in the module, we can create different ResNet models, such as the deeper 152-layer ResNet-152. Although the main architecture of ResNet is similar to that of GoogLeNet, ResNet's structure is simpler and easier to modify. All these factors have resulted in the rapid and widespread use of ResNet. Fig. 7.6.4 is a diagram of the full ResNet-18.

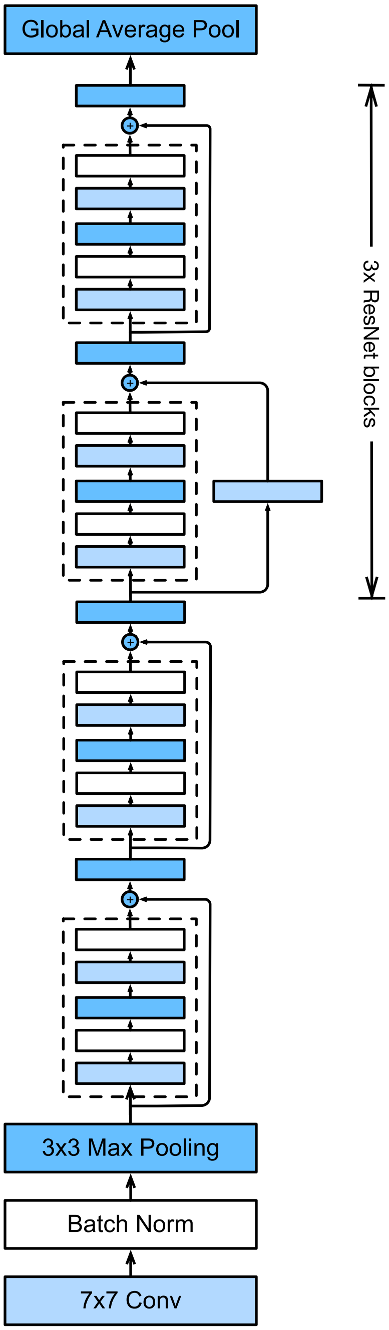


Fig. 7.6.4: ResNet 18

Before training ResNet, let's observe how the input shape changes between different modules in ResNet. As in all previous architectures, the resolution decreases while the number of channels increases up until the point where a global average pooling layer aggregates all features.

```
X = np.random.uniform(size=(1, 1, 224, 224))
net.initialize()
for layer in net:
    X = layer(X)
    print(layer.name, 'output shape:\t', X.shape)
```

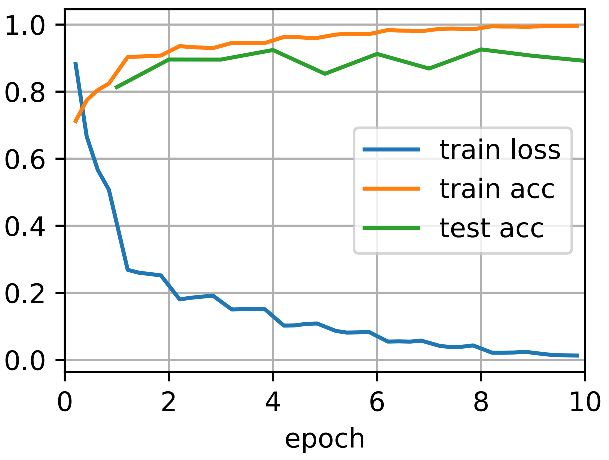
```
conv5 output shape:  (1, 64, 112, 112)
batchnorm4 output shape:     (1, 64, 112, 112)
relu0 output shape:  (1, 64, 112, 112)
pool0 output shape:  (1, 64, 56, 56)
sequential1 output shape:    (1, 64, 56, 56)
sequential2 output shape:    (1, 128, 28, 28)
sequential3 output shape:    (1, 256, 14, 14)
sequential4 output shape:    (1, 512, 7, 7)
pool1 output shape:  (1, 512, 1, 1)
dense0 output shape:         (1, 10)
```

7.6.4 Data Acquisition and Training

We train ResNet on the Fashion-MNIST dataset, just like before. The only thing that has changed is the learning rate that decreased again, due to the more complex architecture.

```
lr, num_epochs, batch_size = 0.05, 10, 256
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size, resize=96)
d2l.train_ch5(net, train_iter, test_iter, num_epochs, lr)
```

```
loss 0.013, train acc 0.997, test acc 0.892
4931.7 exampes/sec on gpu(0)
```



Summary

- Residual blocks allow for a parametrization relative to the identity function $f(\mathbf{x}) = \mathbf{x}$.
- Adding residual blocks increases the function complexity in a well-defined manner.
- We can train an effective deep neural network by having residual blocks pass through cross-layer data channels.
- ResNet had a major influence on the design of subsequent deep neural networks, both for convolutional and sequential nature.

Exercises

1. Refer to Table 1 in the (He et al., 2016a) to implement different variants.
2. For deeper networks, ResNet introduces a "bottleneck" architecture to reduce model complexity. Try to implement it.
3. In subsequent versions of ResNet, the author changed the "convolution, batch normalization, and activation" architecture to the "batch normalization, activation, and convolution" architecture. Make this improvement yourself. See Figure 1 in (He et al., 2016b) for details.
4. Prove that if $\mathbf{x}$ is generated by a ReLU, the ResNet block does indeed include the identity function.
5. Why cannot we just increase the complexity of functions without bound, even if the function classes are nested?

7.7 Densely Connected Networks (DenseNet)

ResNet significantly changed the view of how to parametrize the functions in deep networks. DenseNet is to some extent the logical extension of this. To understand how to arrive at it, let's take a small detour to theory. Recall the Taylor expansion for functions. For scalars it can be written as

$$f(x) = f(0) + f'(x)x + \frac{1}{2}f''(x)x^2 + \frac{1}{6}f'''(x)x^3 + o(x^3). \tag{7.7.1}$$

7.7.1 Function Decomposition

The key point is that it decomposes the function into increasingly higher order terms. In a similar vein, ResNet decomposes functions into

$$f(\mathbf{x}) = \mathbf{x} + g(\mathbf{x}). \tag{7.7.2}$$

That is, ResNet decomposes f into a simple linear term and a more complex nonlinear one. What if we want to go beyond two terms? A solution was proposed by (Huang et al., 2017) in the form of DenseNet, an architecture that reported record performance on the ImageNet dataset.

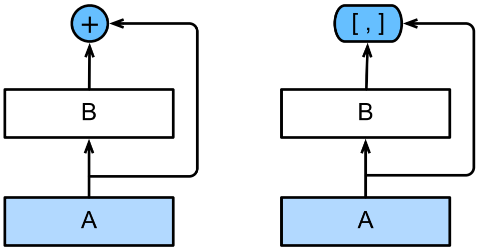


Fig. 7.7.1: The main difference between ResNet (left) and DenseNet (right) in cross-layer connections: use of addition and use of concatenation.

As shown in Fig. 7.7.1, the key difference between ResNet and DenseNet is that in the latter case outputs are *concatenated* rather than added. As a result we perform a mapping from $\mathbf{x}$ to its values after applying an increasingly complex sequence of functions.

$$\mathbf{x} \to [\mathbf{x}, f_1(\mathbf{x}), f_2(\mathbf{x}, f_1(\mathbf{x})), f_3(\mathbf{x}, f_1(\mathbf{x}), f_2(\mathbf{x}, f_1(\mathbf{x}))), \ldots] . \tag{7.7.3}$$

In the end, all these functions are combined in an MLP to reduce the number of features again. In terms of implementation this is quite simple—rather than adding terms, we concatenate them. The name DenseNet arises from the fact that the dependency graph between variables becomes quite dense. The last layer of such a chain is densely connected to all previous layers. The main components that compose a DenseNet are dense blocks and transition layers. The former defines how the inputs and outputs are concatenated, while the latter controls the number of channels so that it is not too large. The dense connections are shown in Fig. 7.7.2.

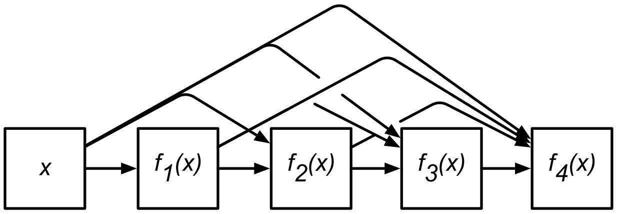


Fig. 7.7.2: Dense connections in DenseNet

7.7.2 Dense Blocks

DenseNet uses the modified "batch normalization, activation, and convolution" architecture of ResNet (see the exercise in Section 7.6). First, we implement this architecture in the `conv_block` function.

```python
import d2l
from mxnet import np, npx
from mxnet.gluon import nn
npx.set_np()

def conv_block(num_channels):
    blk = nn.Sequential()
    blk.add(nn.BatchNorm(),
            nn.Activation('relu'),
            nn.Conv2D(num_channels, kernel_size=3, padding=1))
    return blk
```

A dense block consists of multiple `conv_block` units, each using the same number of output channels. In the forward computation, however, we concatenate the input and output of each block on the channel dimension.

```python
class DenseBlock(nn.Block):
    def __init__(self, num_convs, num_channels, **kwargs):
        super(DenseBlock, self).__init__(**kwargs)
        self.net = nn.Sequential()
        for _ in range(num_convs):
            self.net.add(conv_block(num_channels))
```

(continues on next page)

(continued from previous page)

```
    def forward(self, X):
        for blk in self.net:
            Y = blk(X)
            # Concatenate the input and output of each block on the channel
            # dimension
            X = np.concatenate((X, Y), axis=1)
        return X
```

In the following example, we define a convolution block with two blocks of 10 output channels. When using an input with 3 channels, we will get an output with the $3 + 2 \times 10 = 23$ channels. The number of convolution block channels controls the increase in the number of output channels relative to the number of input channels. This is also referred to as the growth rate.

```
blk = DenseBlock(2, 10)
blk.initialize()
X = np.random.uniform(size=(4, 3, 8, 8))
Y = blk(X)
Y.shape
```

```
(4, 23, 8, 8)
```

7.7.3 Transition Layers

Since each dense block will increase the number of channels, adding too many of them will lead to an excessively complex model. A transition layer is used to control the complexity of the model. It reduces the number of channels by using the 1×1 convolutional layer and halves the height and width of the average pooling layer with a stride of 2, further reducing the complexity of the model.

```
def transition_block(num_channels):
    blk = nn.Sequential()
    blk.add(nn.BatchNorm(), nn.Activation('relu'),
            nn.Conv2D(num_channels, kernel_size=1),
            nn.AvgPool2D(pool_size=2, strides=2))
    return blk
```

Apply a transition layer with 10 channels to the output of the dense block in the previous example. This reduces the number of output channels to 10, and halves the height and width.

```
blk = transition_block(10)
blk.initialize()
blk(Y).shape
```

```
(4, 10, 4, 4)
```

7.7.4 DenseNet Model

Next, we will construct a DenseNet model. DenseNet first uses the same single convolutional layer and maximum pooling layer as ResNet.

```
net = nn.Sequential()
net.add(nn.Conv2D(64, kernel_size=7, strides=2, padding=3),
        nn.BatchNorm(), nn.Activation('relu'),
        nn.MaxPool2D(pool_size=3, strides=2, padding=1))
```

Then, similar to the four residual blocks that ResNet uses, DenseNet uses four dense blocks. Similar to ResNet, we can set the number of convolutional layers used in each dense block. Here, we set it to 4, consistent with the ResNet-18 in the previous section. Furthermore, we set the number of channels (i.e., growth rate) for the convolutional layers in the dense block to 32, so 128 channels will be added to each dense block.

In ResNet, the height and width are reduced between each module by a residual block with a stride of 2. Here, we use the transition layer to halve the height and width and halve the number of channels.

```
# Num_channels: the current number of channels
num_channels, growth_rate = 64, 32
num_convs_in_dense_blocks = [4, 4, 4, 4]

for i, num_convs in enumerate(num_convs_in_dense_blocks):
    net.add(DenseBlock(num_convs, growth_rate))
    # This is the number of output channels in the previous dense block
    num_channels += num_convs * growth_rate
    # A transition layer that haves the number of channels is added between
    # the dense blocks
    if i != len(num_convs_in_dense_blocks) - 1:
        num_channels //= 2
        net.add(transition_block(num_channels))
```

Similar to ResNet, a global pooling layer and fully connected layer are connected at the end to produce the output.

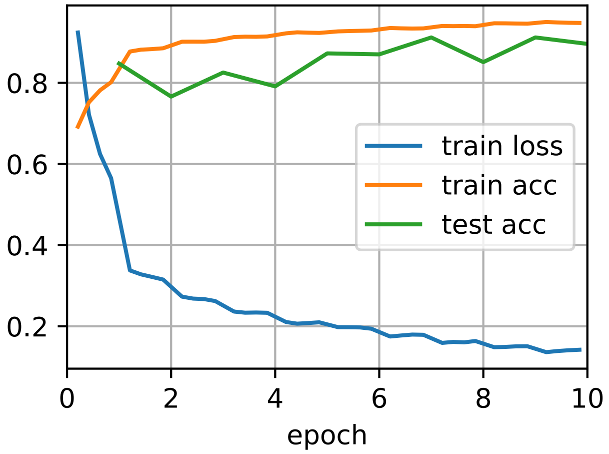
```
net.add(nn.BatchNorm(),
        nn.Activation('relu'),
        nn.GlobalAvgPool2D(),
        nn.Dense(10))
```

7.7.5 Data Acquisition and Training

Since we are using a deeper network here, in this section, we will reduce the input height and width from 224 to 96 to simplify the computation.

```
lr, num_epochs, batch_size = 0.1, 10, 256
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size, resize=96)
d2l.train_ch5(net, train_iter, test_iter, num_epochs, lr)
```

```
loss 0.144, train acc 0.947, test acc 0.896
5628.8 exampes/sec on gpu(0)
```



Summary

- In terms of cross-layer connections, unlike ResNet, where inputs and outputs are added together, DenseNet concatenates inputs and outputs on the channel dimension.
- The main units that compose DenseNet are dense blocks and transition layers.
- We need to keep the dimensionality under control when composing the network by adding transition layers that shrink the number of channels again.

Exercises

1. Why do we use average pooling rather than max-pooling in the transition layer?
2. One of the advantages mentioned in the DenseNet paper is that its model parameters are smaller than those of ResNet. Why is this the case?
3. One problem for which DenseNet has been criticized is its high memory consumption.
    - Is this really the case? Try to change the input shape to 224×224 to see the actual (GPU) memory consumption.
    - Can you think of an alternative means of reducing the memory consumption? How would you need to change the framework?
4. Implement the various DenseNet versions presented in Table 1 of (Huang et al., 2017).
5. Why do we not need to concatenate terms if we are just interested in $\mathbf{x}$ and $f(\mathbf{x})$ for ResNet? Why do we need this for more than two layers in DenseNet?
6. Design a DenseNet for fully connected networks and apply it to the Housing Price prediction task.

8 Recurrent Neural Networks

So far we encountered two types of data: generic vectors and images. For the latter we designed specialized layers to take advantage of the regularity properties in them. In other words, if we were to permute the pixels in an image, it would be much more difficult to reason about its content of something that would look much like the background of a test pattern in the times of analog TV.

Most importantly, so far we tacitly assumed that our data is generated i.i.d., i.e., independently and identically distributed, all drawn from some distribution. Unfortunately, this is not true for most data. For instance, the words in this paragraph are written in sequence, and it would be quite difficult to decipher its meaning if they were permuted randomly. Likewise, image frames in a video, the audio signal in a conversation, or the browsing behavior on a website, all follow sequential order. It is thus only reasonable to assume that specialized models for such data will do better at describing it and at solving estimation problems.

Another issue arises from the fact that we might not only receive a sequence as an input but rather might be expected to continue the sequence. For instance, the task could be to continue the series 2, 4, 6, 8, 10, … This is quite common in time series analysis, to predict the stock market, the fever curve of a patient or the acceleration needed for a race car. Again we want to have models that can handle such data.

In short, while convolutional neural networks can efficiently process spatial information, recurrent neural networks are designed to better handle sequential information. These networks introduce state variables to store past information, and then determine the current outputs, together with the current inputs.

Many of the examples for using recurrent networks are based on text data. Hence, we will emphasize language models in this chapter. After a more formal review of sequence data we discuss basic concepts of a language model and use this discussion as the inspiration for the design of recurrent neural networks. Next, we describe the gradient calculation method in recurrent neural networks to explore problems that may be encountered in recurrent neural network training.

8.1 Sequence Models

Imagine that you are watching movies on Netflix. As a good Netflix user, you decide to rate each of the movies religiously. After all, a good movie is a good movie, and you want to watch more of them, right? As it turns out, things are not quite so simple. People's opinions on movies can change quite significantly over time. In fact, psychologists even have names for some of the effects:

- There is anchoring[120], based on someone else's opinion. For instance after the Oscar awards, ratings for the corresponding movie go up, even though it is still the same movie. This effect

[120] https://en.wikipedia.org/wiki/Anchoring

persists for a few months until the award is forgotten. (Wu et al., 2017) showed that the effect lifts rating by over half a point.

- There is the Hedonic adaptation[121], where humans quickly adapt to accept an improved (or a bad) situation as the new normal. For instance, after watching many good movies, the expectations that the next movie is equally good or better are high, hence even an average movie might be considered a bad movie after many great ones.
- There is seasonality. Very few viewers like to watch a Santa Claus movie in August.
- In some cases movies become unpopular due to the misbehaviors of directors or actors in the production.
- Some movies become cult movies, because they were almost comically bad. *Plan 9 from Outer Space* and *Troll 2* achieved a high degree of notoriety for this reason.

In short, ratings are anything but stationary. Using temporal dynamics helped (Koren, 2009) to recommend movies more accurately. But it is not just about movies.

- Many users have highly particular behavior when it comes to the time when they open apps. For instance, social media apps are much more popular after school with students. Stock market trading apps are more commonly used when the markets are open.
- It is much harder to predict tomorrow's stock prices than to fill in the blanks for a stock price we missed yesterday, even though both are just a matter of estimating one number. After all, hindsight is so much easier than foresight. In statistics the former is called *extrapolation* whereas the latter is called *interpolation*.
- Music, speech, text, movies, steps, etc. are all sequential in nature. If we were to permute them they would make little sense. The headline *dog bites man* is much less surprising than *man bites dog*, even though the words are identical.
- Earthquakes are strongly correlated, i.e., after a massive earthquake there are very likely several smaller aftershocks, much more so than without the strong quake. In fact, earthquakes are spatiotemporally correlated, i.e., the aftershocks typically occur within a short time span and in close proximity.
- Humans interact with each other in a sequential nature, as can be seen in Twitter fights, dance patterns and debates.

8.1.1 Statistical Tools

In short, we need statistical tools and new deep neural networks architectures to deal with sequence data. To keep things simple, we use the stock price illustrated in Fig. 8.1.1 as an example.

[121] https://en.wikipedia.org/wiki/Hedonic_treadmill

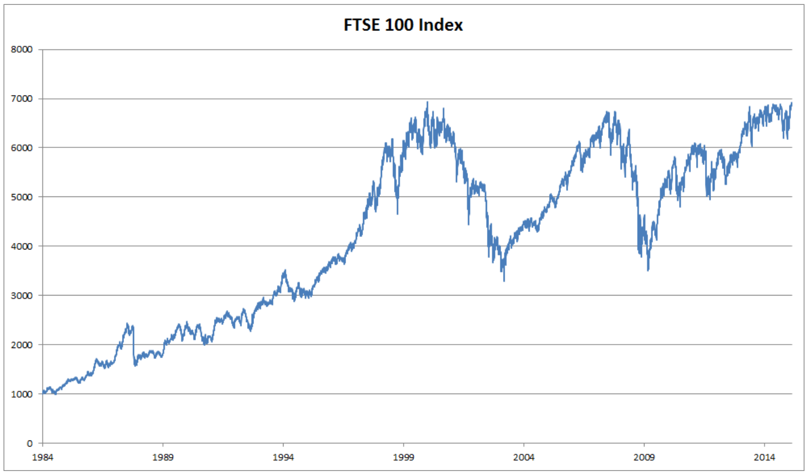


Fig. 8.1.1: FTSE 100 index over 30 years

Let's denote the prices by $x_t \geq 0$, i.e., at time $t \in \mathbb{N}$ we observe some price x_t. For a trader to do well in the stock market on day t he should want to predict x_t via

$$x_t \sim p(x_t \mid x_{t-1}, \ldots, x_1). \tag{8.1.1}$$

Autoregressive Models

In order to achieve this, our trader could use a regressor such as the one we trained in Section 3.3. There is just a major problem: the number of inputs, $x_{t-1}, \ldots, x_1$ varies, depending on t. That is, the number increases with the amount of data that we encounter, and we will need an approximation to make this computationally tractable. Much of what follows in this chapter will revolve around how to estimate $p(x_t \mid x_{t-1}, \ldots, x_1)$ efficiently. In a nutshell it boils down to two strategies:

1. Assume that the potentially rather long sequence $x_{t-1}, \ldots, x_1$ is not really necessary. In this case we might content ourselves with some timespan τ and only use $x_{t-1}, \ldots, x_{t-\tau}$ observations. The immediate benefit is that now the number of arguments is always the same, at least for $t > \tau$. This allows us to train a deep network as indicated above. Such models will be called *autoregressive* models, as they quite literally perform regression on themselves.

2. Another strategy, shown in Fig. 8.1.2, is to try and keep some summary h_t of the past observations, at the same time update h_t in addition to the prediction $\hat{x}_t$. This leads to models that estimate x_t with $\hat{x}_t = p(x_t \mid x_{t-1}, h_t)$ and moreover updates of the form $h_t = g(h_{t-1}, x_{t-1})$. Since h_t is never observed, these models are also called *latent autoregressive models*. LSTMs and GRUs are examples of this.

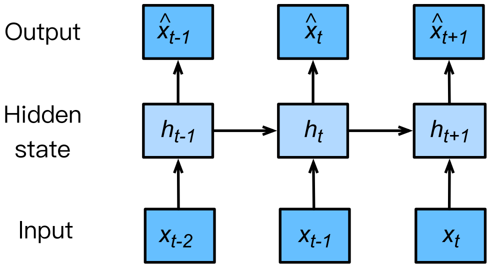


Fig. 8.1.2: A latent autoregressive model.

Both cases raise the obvious question of how to generate training data. One typically uses historical observations to predict the next observation given the ones up to right now. Obviously we do not expect time to stand still. However, a common assumption is that while the specific values of x_t might change, at least the dynamics of the time series itself will not. This is reasonable, since novel dynamics are just that, novel and thus not predictable using data that we have so far. Statisticians call dynamics that do not change *stationary*. Regardless of what we do, we will thus get an estimate of the entire time series via

$$p(x_1, \ldots, x_T) = \prod_{t=1}^T p(x_t \mid x_{t-1}, \ldots, x_1). \tag{8.1.2}$$

Note that the above considerations still hold if we deal with discrete objects, such as words, rather than numbers. The only difference is that in such a situation we need to use a classifier rather than a regressor to estimate $p(x_t \mid x_{t-1}, \ldots, x_1)$.

Markov Model

Recall the approximation that in an autoregressive model we use only $(x_{t-1}, \ldots, x_{t-\tau})$ instead of $(x_{t-1}, \ldots, x_1)$ to estimate x_t. Whenever this approximation is accurate we say that the sequence satisfies a *Markov condition*. In particular, if $\tau = 1$, we have a *first order* Markov model and $p(x)$ is given by

$$p(x_1, \ldots, x_T) = \prod_{t=1}^T p(x_t \mid x_{t-1}). \tag{8.1.3}$$

Such models are particularly nice whenever x_t assumes only a discrete value, since in this case dynamic programming can be used to compute values along the chain exactly. For instance, we can compute $p(x_{t+1} \mid x_{t-1})$ efficiently using the fact that we only need to take into account a very short history of past observations:

$$p(x_{t+1} \mid x_{t-1}) = \sum_{x_t} p(x_{t+1} \mid x_t) p(x_t \mid x_{t-1}). \tag{8.1.4}$$

Going into details of dynamic programming is beyond the scope of this section, but we will introduce it in Section 9.4. Control and reinforcement learning algorithms use such tools extensively.

Causality

In principle, there is nothing wrong with unfolding $p(x_1, \ldots, x_T)$ in reverse order. After all, by conditioning we can always write it via

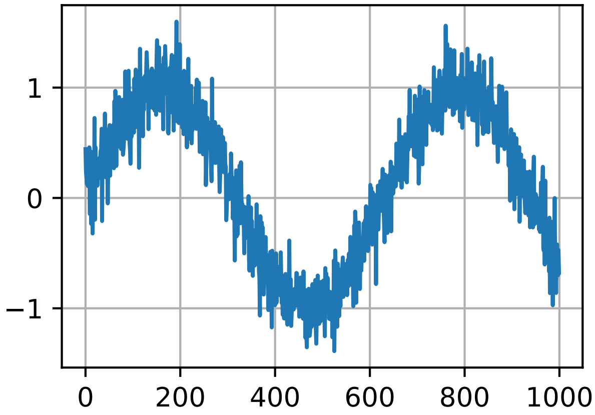
$$p(x_1, \ldots, x_T) = \prod_{t=T}^1 p(x_t \mid x_{t+1}, \ldots, x_T). \tag{8.1.5}$$

In fact, if we have a Markov model, we can obtain a reverse conditional probability distribution, too. In many cases, however, there exists a natural direction for the data, namely going forward in time. It is clear that future events cannot influence the past. Hence, if we change x_t, we may be able to influence what happens for x_{t+1} going forward but not the converse. That is, if we change x_t, the distribution over past events will not change. Consequently, it ought to be easier to explain $p(x_{t+1} \mid x_t)$ rather than $p(x_t \mid x_{t+1})$. For instance, (Hoyer et al., 2009) show that in some cases we can find $x_{t+1} = f(x_t) + \epsilon$ for some additive noise, whereas the converse is not true. This is great news, since it is typically the forward direction that we are interested in estimating. For more on this topic see e.g., the book by (Peters et al., 2017). We are barely scratching the surface of it.

8.1.2 A Toy Example

After so much theory, let's try this out in practice. Let's begin by generating some data. To keep things simple we generate our time series by using a sine function with some additive noise.

```
%matplotlib inline
import d2l
from mxnet import autograd, np, npx, gluon, init
from mxnet.gluon import nn
npx.set_np()

T = 1000  # Generate a total of 1000 points
time = np.arange(0, T)
x = np.sin(0.01 * time) + 0.2 * np.random.normal(size=T)
d2l.plot(time, [x])
```



Next we need to turn this time series into features and labels that the network can train on. Based on the embedding dimension τ we map the data into pairs $y_t = x_t$ and $\mathbf{z}_t = (x_{t-1}, \ldots, x_{t-\tau})$. The astute reader might have noticed that this gives us τ fewer data points, since we do not have sufficient history for the first τ of them. A simple fix, in particular if the time series is long is to discard those few terms. Alternatively we could pad the time series with zeros. The code below is essentially identical to the training code in previous sections. We kept the architecture fairly simple. A few layers of a fully connected network, ReLU activation and ℓ_2 loss. Since much of the modeling is identical to the previous sections when we built regression estimators in Gluon, we will not delve into much detail.

```
tau = 4
features = np.zeros((T-tau, tau))
for i in range(tau):
    features[:, i] = x[i: T-tau+i]
labels = x[tau:]

batch_size, n_train = 16, 600
train_iter = d2l.load_array((features[:n_train], labels[:n_train]),
                            batch_size, is_train=True)
test_iter = d2l.load_array((features[:n_train], labels[:n_train]),
                           batch_size, is_train=False)

# Vanilla MLP architecture
```

(continues on next page)

(continued from previous page)

```
def get_net():
    net = gluon.nn.Sequential()
    net.add(nn.Dense(10, activation='relu'),
            nn.Dense(1))
    net.initialize(init.Xavier())
    return net

# Least mean squares loss
loss = gluon.loss.L2Loss()
```

Now we are ready to train.

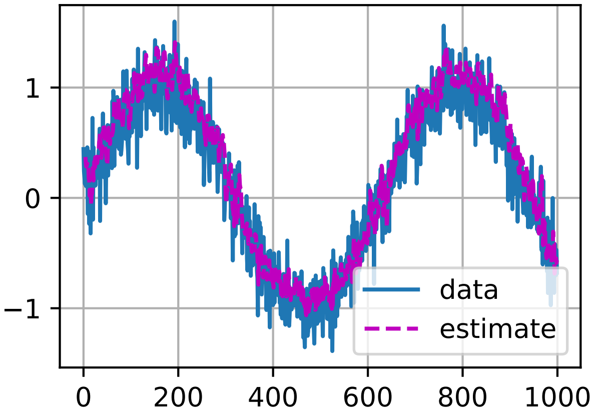
```
def train_net(net, train_iter, loss, epochs, lr):
    trainer = gluon.Trainer(net.collect_params(), 'adam',
                            {'learning_rate': lr})
    for epoch in range(1, epochs + 1):
        for X, y in train_iter:
            with autograd.record():
                l = loss(net(X), y)
            l.backward()
            trainer.step(batch_size)
        print('epoch %d, loss: %f' % (
            epoch, d2l.evaluate_loss(net, train_iter, loss)))

net = get_net()
train_net(net, train_iter, loss, 10, 0.01)
```

```
epoch 1, loss: 0.044017
epoch 2, loss: 0.030621
epoch 3, loss: 0.028620
epoch 4, loss: 0.028395
epoch 5, loss: 0.026994
epoch 6, loss: 0.026267
epoch 7, loss: 0.025798
epoch 8, loss: 0.025620
epoch 9, loss: 0.027264
epoch 10, loss: 0.030226
```

8.1.3 Predictions

Since both training and test loss are small, we would expect our model to work well. Let's see what this means in practice. The first thing to check is how well the model is able to predict what happens in the next timestep.

```
estimates = net(features)
d2l.plot([time, time[tau:]], [x, estimates],
         legend=['data', 'estimate'])
```

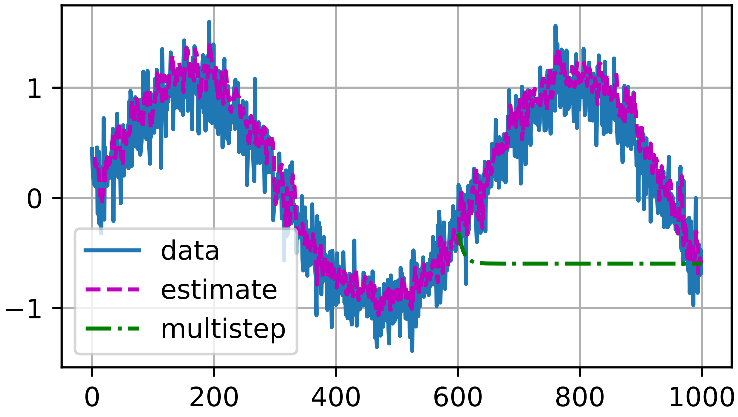


This looks nice, just as we expected it. Even beyond 600 observations the estimates still look rather trustworthy. There is just one little problem to this: if we observe data only until timestep 600, we cannot hope to receive the ground truth for all future predictions. Instead, we need to work our way forward one step at a time:

$$\begin{aligned}
x_{601} &= f(x_{600}, \ldots, x_{597}), \\
x_{602} &= f(x_{601}, \ldots, x_{598}), \\
x_{603} &= f(x_{602}, \ldots, x_{599}).
\end{aligned} \tag{8.1.6}$$

In other words, we will have to use our own predictions to make future predictions. Let's see how well this goes.

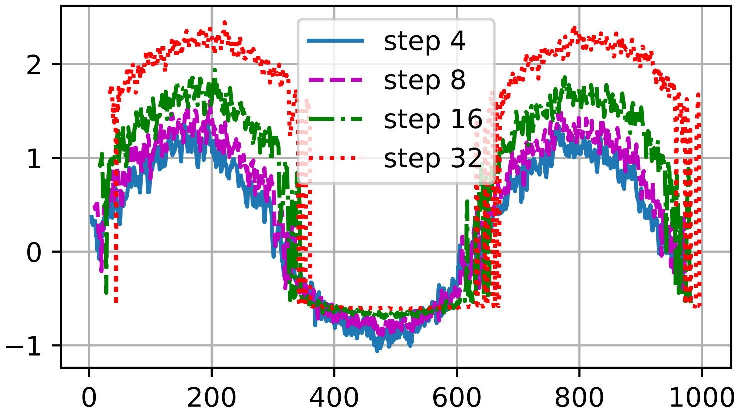
```
predictions = np.zeros(T)
predictions[:n_train] = x[:n_train]
for i in range(n_train, T):
    predictions[i] = net(
        predictions[(i-tau):i].reshape(1, -1)).reshape(1)
d2l.plot([time, time[tau:], time[n_train:]],
         [x, estimates, predictions[n_train:]],
         legend=['data', 'estimate', 'multistep'], figsize=(4.5, 2.5))
```



As the above example shows, this is a spectacular failure. The estimates decay to a constant pretty quickly after a few prediction steps. Why did the algorithm work so poorly? This is ultimately due to the fact that the errors build up. Let's say that after step 1 we have some error $\epsilon_1 = \bar{\epsilon}$. Now the *input* for step 2 is perturbed by ϵ_1, hence we suffer some error in the order of $\epsilon_2 = \bar{\epsilon} + L\epsilon_1$, and so on. The error can diverge rather rapidly from the true observations. This is a common phenomenon. For instance, weather forecasts for the next 24 hours tend to be pretty accurate but beyond that the accuracy declines rapidly. We will discuss methods for improving this throughout this chapter and beyond.

Let's verify this observation by computing the k-step predictions on the entire sequence.

```
k = 33  # Look up to k - tau steps ahead

features = np.zeros((k, T-k))
for i in range(tau):  # Copy the first tau features from x
    features[i] = x[i:T-k+i]

for i in range(tau, k):  # Predict the (i-tau)-th step
    features[i] = net(features[(i-tau):i].T).T

steps = (4, 8, 16, 32)
d2l.plot([time[i:T-k+i] for i in steps], [features[i] for i in steps],
         legend=['step %d' % i for i in steps], figsize=(4.5, 2.5))
```



This clearly illustrates how the quality of the estimates changes as we try to predict further into the future. While the 8-step predictions are still pretty good, anything beyond that is pretty useless.

Summary

- Sequence models require specialized statistical tools for estimation. Two popular choices are autoregressive models and latent-variable autoregressive models.
- As we predict further in time, the errors accumulate and the quality of the estimates degrades, often dramatically.
- There is quite a difference in difficulty between interpolation and extrapolation. Consequently, if you have a time series, always respect the temporal order of the data when training, i.e., never train on future data.
- For causal models (e.g., time going forward), estimating the forward direction is typically a lot easier than the reverse direction.

Exercises

1. Improve the above model.
   - Incorporate more than the past 4 observations? How many do you really need?
   - How many would you need if there was no noise? Hint: you can write sin and cos as a differential equation.
   - Can you incorporate older features while keeping the total number of features constant? Does this improve accuracy? Why?
   - Change the neural network architecture and see what happens.
2. An investor wants to find a good security to buy. She looks at past returns to decide which one is likely to do well. What could possibly go wrong with this strategy?
3. Does causality also apply to text? To which extent?
4. Give an example for when a latent autoregressive model might be needed to capture the dynamic of the data.

8.2 Text Preprocessing

Text is an important example of sequence data. An article can be simply viewed as a sequence of words, or a sequence of characters. Given text data is a major data format besides images we are using in this book, this section will dedicate to explain the common preprocessing steps for text data. Such preprocessing often consists of four steps:

1. Load text as strings into memory.
2. Split strings into tokens, where a token could be a word or a character.
3. Build a vocabulary for these tokens to map them into numerical indices.
4. Map all the tokens in data into indices for ease of feeding into models.

8.2.1 Reading the Dataset

To get started we load text from H. G. Wells' Time Machine[123]. This is a fairly small corpus of just over $30,000$ words, but for the purpose of what we want to illustrate this is just fine. More realistic document collections contain many billions of words. The following function reads the dataset into a list of sentences, each sentence is a string. Here we ignore punctuation and capitalization.

```
import collections
import re
```

(continues on next page)

[123] http://www.gutenberg.org/ebooks/35

(continued from previous page)

```
# Saved in the d2l package for later use
def read_time_machine():
    """Load the time machine book into a list of sentences."""
    with open('../data/timemachine.txt', 'r') as f:
        lines = f.readlines()
    return [re.sub('[^A-Za-z]+', ' ', line.strip().lower())
            for line in lines]

lines = read_time_machine()
'# sentences %d' % len(lines)
```

```
'# sentences 3221'
```

8.2.2 Tokenization

For each sentence, we split it into a list of tokens. A token is a data point the model will train and predict. The following function supports split a sentence into words or characters, and returns a list of split strings.

```
# Saved in the d2l package for later use
def tokenize(lines, token='word'):
    """Split sentences into word or char tokens"""
    if token == 'word':
        return [line.split(' ') for line in lines]
    elif token == 'char':
        return [list(line) for line in lines]
    else:
        print('ERROR: unkown token type '+token)

tokens = tokenize(lines)
tokens[0:2]
```

```
[['the', 'time', 'machine', 'by', 'h', 'g', 'wells', ''], ['']]
```

8.2.3 Vocabulary

The string type of the token is inconvenient to be used by models, which take numerical inputs. Now let's build a dictionary, often called *vocabulary* as well, to map string tokens into numerical indices starting from 0. To do so, we first count the unique tokens in all documents, called *corpus*, and then assign a numerical index to each unique token according to its frequency. Rarely appeared tokens are often removed to reduce the complexity. A token does not exist in corpus or has been removed is mapped into a special unknown ("<unk>") token. We optionally add another three special tokens: "<pad>" a token for padding, "<bos>" to present the beginning for a sentence, and "<eos>" for the ending of a sentence.

```
# Saved in the d2l package for later use
class Vocab(object):
    def __init__(self, tokens, min_freq=0, use_special_tokens=False):
```

(continues on next page)

(continued from previous page)

```
        # Sort according to frequencies
        counter = count_corpus(tokens)
        self.token_freqs = sorted(counter.items(), key=lambda x: x[0])
        self.token_freqs.sort(key=lambda x: x[1], reverse=True)
        if use_special_tokens:
            # For padding, begin of sentence, end of sentence, and unknown
            self.pad, self.bos, self.eos, self.unk = (0, 1, 2, 3)
            uniq_tokens = ['<pad>', '<bos>', '<eos>', '<unk>']
        else:
            self.unk, uniq_tokens = 0, ['<unk>']
        uniq_tokens += [token for token, freq in self.token_freqs
                        if freq >= min_freq and token not in uniq_tokens]
        self.idx_to_token, self.token_to_idx = [], dict()
        for token in uniq_tokens:
            self.idx_to_token.append(token)
            self.token_to_idx[token] = len(self.idx_to_token) - 1

    def __len__(self):
        return len(self.idx_to_token)

    def __getitem__(self, tokens):
        if not isinstance(tokens, (list, tuple)):
            return self.token_to_idx.get(tokens, self.unk)
        return [self.__getitem__(token) for token in tokens]

    def to_tokens(self, indices):
        if not isinstance(indices, (list, tuple)):
            return self.idx_to_token[indices]
        return [self.idx_to_token[index] for index in indices]

# Saved in the d2l package for later use
def count_corpus(sentences):
    # Flatten a list of token lists into a list of tokens
    tokens = [tk for line in sentences for tk in line]
    return collections.Counter(tokens)
```

We construct a vocabulary with the time machine dataset as the corpus, and then print the map between a few tokens and their indices.

```
vocab = Vocab(tokens)
print(list(vocab.token_to_idx.items())[0:10])
```

```
[('<unk>', 0), ('the', 1), ('', 2), ('i', 3), ('and', 4), ('of', 5), ('a', 6), ('to', 7), (
↪'was', 8), ('in', 9)]
```

After that, we can convert each sentence into a list of numerical indices. To illustrate in detail, we print two sentences with their corresponding indices.

```
for i in range(8, 10):
    print('words:', tokens[i])
    print('indices:', vocab[tokens[i]])
```

```
words: ['the', 'time', 'traveller', 'for', 'so', 'it', 'will', 'be', 'convenient', 'to',
↪'speak', 'of', 'him', '']
indices: [1, 20, 72, 17, 38, 12, 120, 43, 706, 7, 660, 5, 112, 2]
words: ['was', 'expounding', 'a', 'recondite', 'matter', 'to', 'us', 'his', 'grey', 'eyes',
↪'shone', 'and']
indices: [8, 1654, 6, 3864, 634, 7, 131, 26, 344, 127, 484, 4]
```

8.2.4 Putting All Things Together

Using the above functions, we package everything into the load_corpus_time_machine function, which returns corpus, a list of token indices, and vocab, the vocabulary of the time machine corpus. The modification we did here is that corpus is a single list, not a list of token lists, since we do not keep the sequence information in the following models. Besides, we use character tokens to simplify the training in later sections.

```
# Saved in the d2l package for later use
def load_corpus_time_machine(max_tokens=-1):
    lines = read_time_machine()
    tokens = tokenize(lines, 'char')
    vocab = Vocab(tokens)
    corpus = [vocab[tk] for line in tokens for tk in line]
    if max_tokens > 0:
        corpus = corpus[:max_tokens]
    return corpus, vocab

corpus, vocab = load_corpus_time_machine()
len(corpus), len(vocab)
```

```
(171489, 28)
```

Summary

- We preprocessed the documents by tokenizing them into words or characters and then mapping into indices.

Exercises

1. Tokenization is a key preprocessing step. It varies for different languages. Try to find another 3 commonly used methods to tokenize sentences.

8.3 Language Models and the Dataset

In Section 8.2, we see how to map text data into tokens, and these tokens can be viewed as a time series of discrete observations. Assuming the tokens in a text of length T are in turn $x_1, x_2, \ldots, x_T$, then, in the discrete time series, $x_t(1 \leq t \leq T)$ can be considered as the output or label of timestep t. Given such a sequence, the goal of a language model is to estimate the probability

$$p(x_1, x_2, \ldots, x_T). \tag{8.3.1}$$

Language models are incredibly useful. For instance, an ideal language model would be able to generate natural text just on its own, simply by drawing one word at a time $w_t \sim p(w_t \mid w_{t-1}, \ldots, w_1)$. Quite unlike the monkey using a typewriter, all text emerging from such a model would pass as natural language, e.g., English text. Furthermore, it would be sufficient for generating a meaningful dialog, simply by conditioning the text on previous dialog fragments. Clearly we are still very far from designing such a system, since it would need to *understand* the text rather than just generate grammatically sensible content.

Nonetheless language models are of great service even in their limited form. For instance, the phrases "to recognize speech" and "to wreck a nice beach" sound very similar. This can cause ambiguity in speech recognition, ambiguity that is easily resolved through a language model which rejects the second translation as outlandish. Likewise, in a document summarization algorithm it is worth while knowing that "dog bites man" is much more frequent than "man bites dog", or that "let's eat grandma" is a rather disturbing statement, whereas "let's eat, grandma" is much more benign.

8.3.1 Estimating a Language Model

The obvious question is how we should model a document, or even a sequence of words. We can take recourse to the analysis we applied to sequence models in the previous section. Let's start by applying basic probability rules:

$$p(w_1, w_2, \ldots, w_T) = p(w_1) \prod_{t=2}^{T} p(w_t \mid w_1, \ldots, w_{t-1}). \tag{8.3.2}$$

For example, the probability of a text sequence containing four tokens consisting of words and punctuation would be given as:

$$p(\text{Statistics}, \text{is}, \text{fun}, .) = p(\text{Statistics})p(\text{is} \mid \text{Statistics})p(\text{fun} \mid \text{Statistics}, \text{is})p(. \mid \text{Statistics}, \text{is}, \text{fun}). \tag{8.3.3}$$

In order to compute the language model, we need to calculate the probability of words and the conditional probability of a word given the previous few words, i.e., language model parameters. Here, we assume that the training dataset is a large text corpus, such as all Wikipedia entries, Project Gutenberg[125], or all text posted online on the web. The probability of words can be calculated from the relative word frequency of a given word in the training dataset.

For example, $p(\text{Statistics})$ can be calculated as the probability of any sentence starting with the word "statistics". A slightly less accurate approach would be to count all occurrences of the word "statistics" and divide it by the total number of words in the corpus. This works fairly well, particularly for frequent words. Moving on, we could attempt to estimate

$$\hat{p}(\text{is} \mid \text{Statistics}) = \frac{n(\text{Statistics is})}{n(\text{Statistics})}. \tag{8.3.4}$$

[125] https://en.wikipedia.org/wiki/Project_Gutenberg

Here $n(w)$ and $n(w, w')$ are the number of occurrences of singletons and pairs of words respectively. Unfortunately, estimating the probability of a word pair is somewhat more difficult, since the occurrences of "Statistics is" are a lot less frequent. In particular, for some unusual word combinations it may be tricky to find enough occurrences to get accurate estimates. Things take a turn for the worse for 3-word combinations and beyond. There will be many plausible 3-word combinations that we likely will not see in our dataset. Unless we provide some solution to give such word combinations nonzero weight, we will not be able to use these as a language model. If the dataset is small or if the words are very rare, we might not find even a single one of them.

A common strategy is to perform some form of Laplace smoothing. We already encountered this in our discussion of naive Bayes in Section 17.8 where the solution was to add a small constant to all counts. This helps with singletons, e.g., via

$$\begin{aligned}
\hat{p}(w) & = \frac{n(w) + \epsilon_1/m}{n + \epsilon_1}, \\
\hat{p}(w' \mid w) & = \frac{n(w, w') + \epsilon_2 \hat{p}(w')}{n(w) + \epsilon_2}, \\
\hat{p}(w'' \mid w', w) & = \frac{n(w, w', w'') + \epsilon_3 \hat{p}(w', w'')}{n(w, w') + \epsilon_3}.
\end{aligned} \tag{8.3.5}$$

Here the coefficients $\epsilon_i > 0$ determine how much we use the estimate for a shorter sequence as a fill-in for longer ones. Moreover, m is the total number of words we encounter. The above is a rather primitive variant of what is Kneser-Ney smoothing and Bayesian nonparametrics can accomplish. See e.g., (Wood et al., 2011) for more detail of how to accomplish this. Unfortunately, models like this get unwieldy rather quickly for the following reasons. First, we need to store all counts. Second, this entirely ignores the meaning of the words. For instance, *"cat"* and *"feline"* should occur in related contexts. It is quite difficult to adjust such models to additional contexts, whereas, deep learning based language models are well suited to take this into account. Last, long word sequences are almost certain to be novel, hence a model that simply counts the frequency of previously seen word sequences is bound to perform poorly there.

8.3.2 Markov Models and n-grams

Before we discuss solutions involving deep learning, we need some more terminology and concepts. Recall our discussion of Markov Models in the previous section. Let's apply this to language modeling. A distribution over sequences satisfies the Markov property of first order if $p(w_{t+1} \mid w_t, \ldots, w_1) = p(w_{t+1} \mid w_t)$. Higher orders correspond to longer dependencies. This leads to a number of approximations that we could apply to model a sequence:

$$\begin{aligned}
p(w_1, w_2, w_3, w_4) & = p(w_1) p(w_2) p(w_3) p(w_4), \\
p(w_1, w_2, w_3, w_4) & = p(w_1) p(w_2 \mid w_1) p(w_3 \mid w_2) p(w_4 \mid w_3), \\
p(w_1, w_2, w_3, w_4) & = p(w_1) p(w_2 \mid w_1) p(w_3 \mid w_1, w_2) p(w_4 \mid w_2, w_3).
\end{aligned} \tag{8.3.6}$$

The probability formulae that involve one, two, and three variables are typically referred to as unigram, bigram, and trigram models respectively. In the following, we will learn how to design better models.

8.3.3 Natural Language Statistics

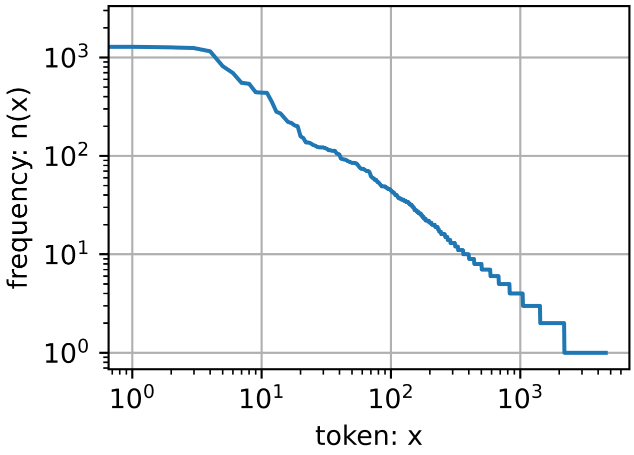
Let's see how this works on real data. We construct a vocabulary based on the time machine data similar to Section 8.2 and print the top 10 most frequent words.

```
import d2l
from mxnet import np, npx
import random
npx.set_np()

tokens = d2l.tokenize(d2l.read_time_machine())
vocab = d2l.Vocab(tokens)
print(vocab.token_freqs[:10])
```

```
[('the', 2261), ('', 1282), ('i', 1267), ('and', 1245), ('of', 1155), ('a', 816), ('to',␣
↪695), ('was', 552), ('in', 541), ('that', 443)]
```

As we can see, the most popular words are actually quite boring to look at. They are often referred to as stop words[126] and thus filtered out. That said, they still carry meaning and we will use them nonetheless. However, one thing that is quite clear is that the word frequency decays rather rapidly. The 10^{th} most frequent word is less than $1/5$ as common as the most popular one. To get a better idea we plot the graph of the word frequency.

```
freqs = [freq for token, freq in vocab.token_freqs]
d2l.plot(freqs, xlabel='token: x', ylabel='frequency: n(x)',
         xscale='log', yscale='log')
```



We are on to something quite fundamental here: the word frequency decays rapidly in a well defined way. After dealing with the first four words as exceptions ('the', 'i', 'and', 'of'), all remaining words follow a straight line on a log-log plot. This means that words satisfy Zipf's law[127] which states that the item frequency is given by

$$n(x) \propto (x + c)^{-\alpha} \text{ and hence } \log n(x) = -\alpha \log(x + c) + \text{const.} \tag{8.3.7}$$

This should already give us pause if we want to model words by count statistics and smoothing. After all, we will significantly overestimate the frequency of the tail, also known as the infrequent

[126] https://en.wikipedia.org/wiki/Stop_words
[127] https://en.wikipedia.org/wiki/Zipf%27s_law

words. But what about the other word combinations (such as bigrams, trigrams, and beyond)? Let's see whether the bigram frequency behaves in the same manner as the unigram frequency.

```
bigram_tokens = [[pair for pair in zip(
    line[:-1], line[1:])] for line in tokens]
bigram_vocab = d2l.Vocab(bigram_tokens)
print(bigram_vocab.token_freqs[:10])
```

```
[(('of', 'the'), 297), (('in', 'the'), 161), (('i', 'had'), 126), (('and', 'the'), 104), (('i
↪', 'was'), 104), (('the', 'time'), 97), (('it', 'was'), 94), (('to', 'the'), 81), (('as',
↪'i'), 75), (('of', 'a'), 69)]
```

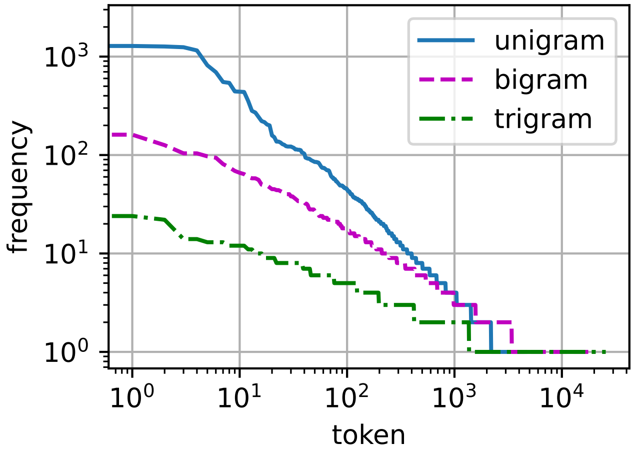
Two things are notable. Out of the 10 most frequent word pairs, 9 are composed of stop words and only one is relevant to the actual book—"the time". Furthermore, let's see whether the trigram frequency behaves in the same manner.

```
trigram_tokens = [[triple for triple in zip(line[:-2], line[1:-1], line[2:])]
                  for line in tokens]
trigram_vocab = d2l.Vocab(trigram_tokens)
print(trigram_vocab.token_freqs[:10])
```

```
[(('the', 'time', 'traveller'), 53), (('the', 'time', 'machine'), 24), (('the', 'medical',
↪'man'), 22), (('it', 'seemed', 'to'), 14), (('it', 'was', 'a'), 14), (('i', 'began', 'to'),
↪ 13), (('i', 'did', 'not'), 13), (('i', 'saw', 'the'), 13), (('here', 'and', 'there'), 12),
↪ (('i', 'could', 'see'), 12)]
```

Last, let's visualize the token frequency among these three gram models: unigrams, bigrams, and trigrams.

```
bigram_freqs = [freq for token, freq in bigram_vocab.token_freqs]
trigram_freqs = [freq for token, freq in trigram_vocab.token_freqs]
d2l.plot([freqs, bigram_freqs, trigram_freqs], xlabel='token',
         ylabel='frequency', xscale='log', yscale='log',
         legend=['unigram', 'bigram', 'trigram'])
```



The graph is quite exciting for a number of reasons. First, beyond unigram words, also sequences of words appear to be following Zipf's law, albeit with a lower exponent, depending on sequence length. Second, the number of distinct n-grams is not that large. This gives us hope that there is quite a lot of structure in language. Third, many n-grams occur very rarely, which makes Laplace smoothing rather unsuitable for language modeling. Instead, we will use deep learning based models.

8.3.4 Training Data Preparation

Before introducing the model, let's assume we will use a neural network to train a language model. Now the question is how to read minibatches of examples and labels at random. Since sequence data is by its very nature sequential, we need to address the issue of processing it. We did so in a rather ad-hoc manner when we introduced in Section 8.1. Let's formalize this a bit.

In Fig. 8.3.1, we visualized several possible ways to obtain 5-grams in a sentence, here a token is a character. Note that we have quite some freedom since we could pick an arbitrary offset.

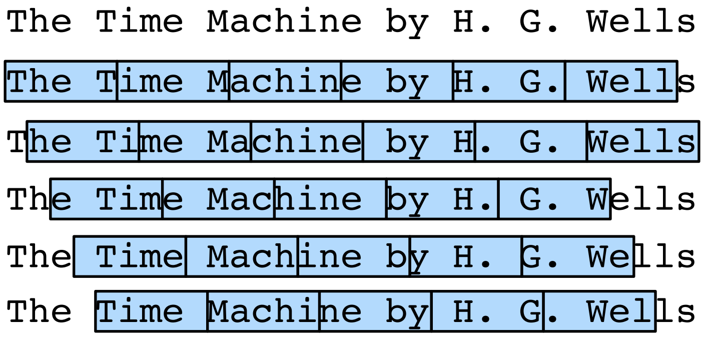

The Time Machine by H. G. Wells
The Time Machine by H. G. Wells
The Time Machine by H. G. Wells
The Time Machine by H. G. Wells
The Time Machine by H. G. Wells
The Time Machine by H. G. Wells

Fig. 8.3.1: Different offsets lead to different subsequences when splitting up text.

In fact, any one of these offsets is fine. Hence, which one should we pick? In fact, all of them are equally good. But if we pick all offsets we end up with rather redundant data due to overlap, particularly if the sequences are long. Picking just a random set of initial positions is no good either since it does not guarantee uniform coverage of the array. For instance, if we pick n elements at random out of a set of n with random replacement, the probability for a particular element not being picked is $(1 - 1/n)^n \to e^{-1}$. This means that we cannot expect uniform coverage this way. Even randomly permuting a set of all offsets does not offer good guarantees. Instead we can use a simple trick to get both *coverage* and *randomness*: use a random offset, after which one uses the terms sequentially. We describe how to accomplish this for both random sampling and sequential partitioning strategies below.

Random Sampling

The following code randomly generates a minibatch from the data each time. Here, the batch size `batch_size` indicates to the number of examples in each minibatch and `num_steps` is the length of the sequence (or timesteps if we have a time series) included in each example. In random sampling, each example is a sequence arbitrarily captured on the original sequence. The positions of two adjacent random minibatches on the original sequence are not necessarily adjacent. The target is to predict the next character based on what we have seen so far, hence the labels are the original sequence, shifted by one character.

```
# Saved in the d2l package for later use
def seq_data_iter_random(corpus, batch_size, num_steps):
    # Offset the iterator over the data for uniform starts
    corpus = corpus[random.randint(0, num_steps):]
    # Subtract 1 extra since we need to account for label
    num_examples = ((len(corpus) - 1) // num_steps)
    example_indices = list(range(0, num_examples * num_steps, num_steps))
    random.shuffle(example_indices)

    def data(pos):
        # This returns a sequence of the length num_steps starting from pos
        return corpus[pos: pos + num_steps]

    # Discard half empty batches
    num_batches = num_examples // batch_size
    for i in range(0, batch_size * num_batches, batch_size):
        # Batch_size indicates the random examples read each time
        batch_indices = example_indices[i:(i+batch_size)]
        X = [data(j) for j in batch_indices]
        Y = [data(j + 1) for j in batch_indices]
        yield np.array(X), np.array(Y)
```

Let's generate an artificial sequence from 0 to 30. We assume that the batch size and numbers of timesteps are 2 and 5 respectively. This means that depending on the offset we can generate between 4 and 5 (x, y) pairs. With a minibatch size of 2, we only get 2 minibatches.

```
my_seq = list(range(30))
for X, Y in seq_data_iter_random(my_seq, batch_size=2, num_steps=6):
    print('X: ', X, '\nY:', Y)
```

```
X:  [[ 2.  3.  4.  5.  6.  7.]
 [20. 21. 22. 23. 24. 25.]]
Y: [[ 3.  4.  5.  6.  7.  8.]
 [21. 22. 23. 24. 25. 26.]]
X:  [[14. 15. 16. 17. 18. 19.]
 [ 8.  9. 10. 11. 12. 13.]]
Y: [[15. 16. 17. 18. 19. 20.]
 [ 9. 10. 11. 12. 13. 14.]]
```

Sequential Partitioning

In addition to random sampling of the original sequence, we can also make the positions of two adjacent random minibatches adjacent in the original sequence.

```
# Saved in the d2l package for later use
def seq_data_iter_consecutive(corpus, batch_size, num_steps):
    # Offset for the iterator over the data for uniform starts
    offset = random.randint(0, num_steps)
    # Slice out data - ignore num_steps and just wrap around
    num_indices = ((len(corpus) - offset - 1) // batch_size) * batch_size
    Xs = np.array(corpus[offset:offset+num_indices])
    Ys = np.array(corpus[offset+1:offset+1+num_indices])
```

(continues on next page)

(continued from previous page)

```
    Xs, Ys = Xs.reshape(batch_size, -1), Ys.reshape(batch_size, -1)
    num_batches = Xs.shape[1] // num_steps
    for i in range(0, num_batches * num_steps, num_steps):
        X = Xs[:, i:(i+num_steps)]
        Y = Ys[:, i:(i+num_steps)]
        yield X, Y
```

Using the same settings, print input X and label Y for each minibatch of examples read by random sampling. The positions of two adjacent minibatches on the original sequence are adjacent.

```
for X, Y in seq_data_iter_consecutive(my_seq, batch_size=2, num_steps=6):
    print('X: ', X, '\nY:', Y)
```

```
X:  [[ 4.  5.  6.  7.  8.  9.]
 [16. 17. 18. 19. 20. 21.]]
Y: [[ 5.  6.  7.  8.  9. 10.]
 [17. 18. 19. 20. 21. 22.]]
X:  [[10. 11. 12. 13. 14. 15.]
 [22. 23. 24. 25. 26. 27.]]
Y: [[11. 12. 13. 14. 15. 16.]
 [23. 24. 25. 26. 27. 28.]]
```

Now we wrap the above two sampling functions to a class so that we can use it as a Gluon data iterator later.

```
# Saved in the d2l package for later use
class SeqDataLoader(object):
    """A iterator to load sequence data"""
    def __init__(self, batch_size, num_steps, use_random_iter, max_tokens):
        if use_random_iter:
            self.data_iter_fn = d2l.seq_data_iter_random
        else:
            self.data_iter_fn = d2l.seq_data_iter_consecutive
        self.corpus, self.vocab = d2l.load_corpus_time_machine(max_tokens)
        self.batch_size, self.num_steps = batch_size, num_steps

    def __iter__(self):
        return self.data_iter_fn(self.corpus, self.batch_size, self.num_steps)
```

Last, we define a function load_data_time_machine that returns both the data iterator and the vocabulary, so we can use it similarly as other functions with load_data prefix.

```
# Saved in the d2l package for later use
def load_data_time_machine(batch_size, num_steps, use_random_iter=False,
                           max_tokens=10000):
    data_iter = SeqDataLoader(
        batch_size, num_steps, use_random_iter, max_tokens)
    return data_iter, data_iter.vocab
```

Summary

- Language models are an important technology for natural language processing.
- n-grams provide a convenient model for dealing with long sequences by truncating the dependence.
- Long sequences suffer from the problem that they occur very rarely or never.
- Zipf's law governs the word distribution for not only unigrams but also the other n-grams.
- There is a lot of structure but not enough frequency to deal with infrequent word combinations efficiently via Laplace smoothing.
- The main choices for sequence partitioning are picking between consecutive and random sequences.
- Given the overall document length, it is usually acceptable to be slightly wasteful with the documents and discard half-empty minibatches.

Exercises

1. Suppose there are $100,000$ words in the training dataset. How much word frequency and multi-word adjacent frequency does a four-gram need to store?
2. Review the smoothed probability estimates. Why are they not accurate? Hint: we are dealing with a contiguous sequence rather than singletons.
3. How would you model a dialogue?
4. Estimate the exponent of Zipf's law for unigrams, bigrams, and trigrams.
5. What other minibatch data sampling methods can you think of?
6. Why is it a good idea to have a random offset?
   - Does it really lead to a perfectly uniform distribution over the sequences on the document?
   - What would you have to do to make things even more uniform?
7. If we want a sequence example to be a complete sentence, what kinds of problems does this introduce in minibatch sampling? Why would we want to do this anyway?

8.4 Recurrent Neural Networks

In Section 8.3 we introduced n-gram models, where the conditional probability of word x_t at position t only depends on the $n-1$ previous words. If we want to check the possible effect of words earlier than $t-(n-1)$ on x_t, we need to increase n. However, the number of model parameters would also increase exponentially with it, as we need to store $|V|^n$ numbers for a vocabulary V. Hence, rather than modeling $p(x_t \mid x_{t-1}, \ldots, x_{t-n+1})$ it is preferable to use a *latent variable model* in which we have

$$p(x_t \mid x_{t-1}, \ldots, x_1) \approx p(x_t \mid x_{t-1}, h_t). \tag{8.4.1}$$

Here h_t is a *latent variable* that stores the sequence information. A latent variable is also called as *hidden variable*, *hidden state* or *hidden state variable*. The hidden state at time t could be computed based on both input x_t and hidden state h_{t-1}, that is

$$h_t = f(x_t, h_{t-1}). \tag{8.4.2}$$

For a sufficiently powerful function f, the latent variable model is not an approximation. After all, h_t could simply store all the data it observed so far. We discussed this in Section 8.1. But it could potentially makes both computation and storage expensive.

Note that we also use h to denote by the number of hidden units of a hidden layer. Hidden layers and hidden states refer to two very different concepts. Hidden layers are, as explained, layers that are hidden from view on the path from input to output. Hidden states are technically speaking *inputs* to whatever we do at a given step. Instead, they can only be computed by looking at data at previous iterations. In this sense they have much in common with latent variable models in statistics, such as clustering or topic models where the clusters affect the output but cannot be directly observed.

Recurrent neural networks are neural networks with hidden states. Before introducing this model, let's first revisit the multi-layer perceptron introduced in Section 4.1.

8.4.1 Recurrent Networks Without Hidden States

Let's take a look at a multilayer perceptron with a single hidden layer. Given a minibatch of the instances $\mathbf{X} \in \mathbb{R}^{n\times d}$ with sample size n and d inputs. Let the hidden layer's activation function be ϕ. Hence, the hidden layer's output $\mathbf{H} \in \mathbb{R}^{n\times h}$ is calculated as

$$\mathbf{H} = \phi(\mathbf{X}\mathbf{W}_{xh} + \mathbf{b}_h). \tag{8.4.3}$$

Here, we have the weight parameter $\mathbf{W}_{xh} \in \mathbb{R}^{d\times h}$, bias parameter $\mathbf{b}_h \in \mathbb{R}^{1\times h}$, and the number of hidden units h, for the hidden layer.

The hidden variable $\mathbf{H}$ is used as the input of the output layer. The output layer is given by

$$\mathbf{O} = \mathbf{H}\mathbf{W}_{hq} + \mathbf{b}_q. \tag{8.4.4}$$

Here, $\mathbf{O} \in \mathbb{R}^{n\times q}$ is the output variable, $\mathbf{W}_{hq} \in \mathbb{R}^{h\times q}$ is the weight parameter, and $\mathbf{b}_q \in \mathbb{R}^{1\times q}$ is the bias parameter of the output layer. If it is a classification problem, we can use softmax($\mathbf{O}$) to compute the probability distribution of the output category.

This is entirely analogous to the regression problem we solved previously in Section 8.1, hence we omit details. Suffice it to say that we can pick (x_t, x_{t-1}) pairs at random and estimate the parameters $\mathbf{W}$ and $\mathbf{b}$ of our network via autograd and stochastic gradient descent.

8.4.2 Recurrent Networks with Hidden States

Matters are entirely different when we have hidden states. Let's look at the structure in some more detail. Remember that we often call iteration t as time t in an optimization algorithm, time in a recurrent neural network refers to steps within an iteration. Assume that we have $\mathbf{X}_t \in \mathbb{R}^{n\times d}$, $t = 1, \ldots, T$, in an iteration. And $\mathbf{H}_t \in \mathbb{R}^{n\times h}$ is the hidden variable of timestep t from the sequence. Unlike the multilayer perceptron, here we save the hidden variable $\mathbf{H}_{t-1}$ from the previous timestep and introduce a new weight parameter $\mathbf{W}_{hh} \in \mathbb{R}^{h\times h}$, to describe how to use the hidden variable of the previous timestep in the current timestep. Specifically, the calculation of the hidden variable of the current timestep is determined by the input of the current timestep together with the hidden variable of the previous timestep:

$$\mathbf{H}_t = \phi(\mathbf{X}_t \mathbf{W}_{xh} + \mathbf{H}_{t-1} \mathbf{W}_{hh} + \mathbf{b}_h). \tag{8.4.5}$$

Compared with (8.4.3), we added one more $\mathbf{H}_{t-1} \mathbf{W}_{hh}$ here. From the relationship between hidden variables $\mathbf{H}_t$ and $\mathbf{H}_{t-1}$ of adjacent timesteps, we know that those variables captured and retained the sequence's historical information up to the current timestep, just like the state or memory of the neural network's current timestep. Therefore, such a hidden variable is called a *hidden state*. Since the hidden state uses the same definition of the previous timestep in the current timestep, the computation of the equation above is recurrent, hence the name recurrent neural network (RNN).

There are many different RNN construction methods. RNNs with a hidden state defined by the equation above are very common. For timestep t, the output of the output layer is similar to the computation in the multilayer perceptron:

$$\mathbf{O}_t = \mathbf{H}_t \mathbf{W}_{hq} + \mathbf{b}_q. \tag{8.4.6}$$

RNN parameters include the weight $\mathbf{W}_{xh} \in \mathbb{R}^{d\times h}, \mathbf{W}_{hh} \in \mathbb{R}^{h\times h}$ of the hidden layer with the bias $\mathbf{b}_h \in \mathbb{R}^{1\times h}$, and the weight $\mathbf{W}_{hq} \in \mathbb{R}^{h\times q}$ of the output layer with the bias $\mathbf{b}_q \in \mathbb{R}^{1\times q}$. It is worth mentioning that RNNs always use these model parameters, even for different timesteps. Therefore, the number of RNN model parameters does not grow as the number of timesteps increases.

Fig. 8.4.1 shows the computational logic of an RNN at three adjacent timesteps. In timestep t, the computation of the hidden state can be treated as an entry of a fully connected layer with the activation function ϕ after concatenating the input $\mathbf{X}_t$ with the hidden state $\mathbf{H}_{t-1}$ of the previous timestep. The output of the fully connected layer is the hidden state of the current timestep $\mathbf{H}_t$. Its model parameter is the concatenation of $\mathbf{W}_{xh}$ and $\mathbf{W}_{hh}$, with a bias of $\mathbf{b}_h$. The hidden state of the current timestep t, $\mathbf{H}_t$, will participate in computing the hidden state $\mathbf{H}_{t+1}$ of the next timestep $t+1$. What is more, $\mathbf{H}_t$ will become the input for $\mathbf{O}_t$, the fully connected output layer of the current timestep.

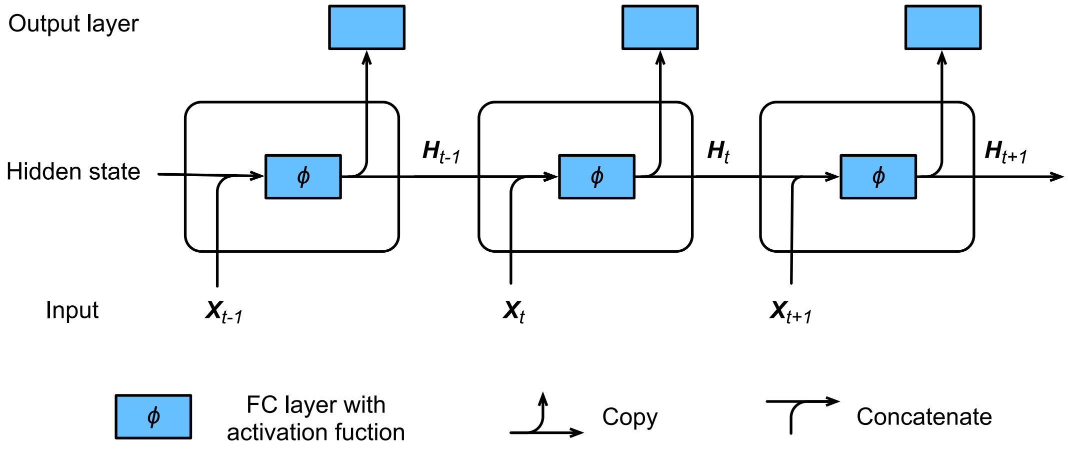


Fig. 8.4.1: An RNN with a hidden state.

8.4.3 Steps in a Language Model

Now we illustrate how RNNs can be used to build a language model. For simplicity of illustration we use words rather than characters as the inputs, since the former are easier to comprehend. Let the minibatch size be 1, and the sequence of the text be the beginning of our dataset, i.e., "the time machine by H. G. Wells". Fig. 8.4.2 illustrates how to estimate the next word based on the present and previous words. During the training process, we run a softmax operation on the output from the output layer for each timestep, and then use the cross-entropy loss function to compute the error between the result and the label. Due to the recurrent computation of the hidden state in the hidden layer, the output of timestep 3, $\mathbf{O}_3$, is determined by the text sequence "the", "time", and "machine" respectively. Since the next word of the sequence in the training data is "by", the loss of timestep 3 will depend on the probability distribution of the next word generated based on the feature sequence "the", "time", "machine" and the label "by" of this timestep.

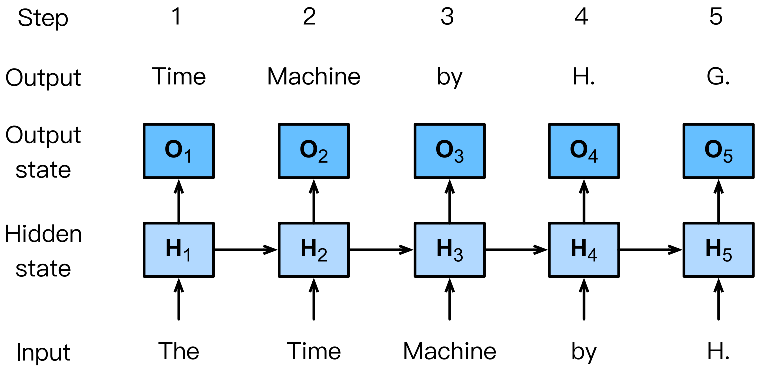


Fig. 8.4.2: Word-level RNN language model. The input and label sequences are `the time machine by H.` and `time machine by H. G.` respectively.

In practice, each word is presented by a d dimensional vector, and we use a batch size $n > 1$. Therefore, the input $\mathbf{X}_t$ at timestep t will be a $n \times d$ matrix, which is identical to what we discussed before.

8.4.4 Perplexity

Last, let's discuss about how to measure the sequence model quality. One way is to check how surprising the text is. A good language model is able to predict with high accuracy tokens that what we will see next. Consider the following continuations of the phrase "It is raining", as proposed by different language models:

1. "It is raining outside"
2. "It is raining banana tree"
3. "It is raining piouw;kcj pwepoiut"

In terms of quality, example 1 is clearly the best. The words are sensible and logically coherent. While it might not quite accurately reflect which word follows semantically ("in San Francisco" and "in winter" would have been perfectly reasonable extensions), the model is able to capture which kind of word follows. Example 2 is considerably worse by producing a nonsensical extension. Nonetheless, at least the model has learned how to spell words and some degree of correlation between words. Last, example 3 indicates a poorly trained model that does not fit data properly.

We might measure the quality of the model by computing $p(w)$, i.e., the likelihood of the sequence. Unfortunately this is a number that is hard to understand and difficult to compare. After all, shorter sequences are much more likely to occur than the longer ones, hence evaluating the model on Tolstoy's magnum opus "War and Peace"[129] will inevitably produce a much smaller likelihood than, say, on Saint-Exupery's novella "The Little Prince"[130]. What is missing is the equivalent of an average.

Information theory comes handy here and we will introduce more in Section 17.10. If we want to compress text, we can ask about estimating the next symbol given the current set of symbols. A lower bound on the number of bits is given by $-\log_2 p(x_t \mid x_{t-1}, \ldots, x_1)$. A good language model should allow us to predict the next word quite accurately. Thus, it should allow us to spend very few bits on compressing the sequence. So we can measure it by the average number of bits that we need to spend.

$$\frac{1}{n}\sum_{t=1}^{n} -\log p(x_t \mid x_{t-1}, \ldots, x_1). \tag{8.4.7}$$

This makes the performance on documents of different lengths comparable. For historical reasons, scientists in natural language processing prefer to use a quantity called *perplexity* rather than bitrate. In a nutshell, it is the exponential of the above:

$$\mathrm{PPL} := \exp\left(-\frac{1}{n}\sum_{t=1}^{n} \log p(x_t \mid x_{t-1}, \ldots, x_1)\right). \tag{8.4.8}$$

It can be best understood as the harmonic mean of the number of real choices that we have when deciding which word to pick next. Note that perplexity naturally generalizes the notion of the cross-entropy loss defined when we introduced the softmax regression (Section 3.4). That is, for a single symbol both definitions are identical bar the fact that one is the exponential of the other. Let's look at a number of cases:

- In the best case scenario, the model always estimates the probability of the next symbol as 1. In this case the perplexity of the model is 1.

[129] https://www.gutenberg.org/files/2600/2600-h/2600-h.htm

[130] https://en.wikipedia.org/wiki/The_Little_Prince

- In the worst case scenario, the model always predicts the probability of the label category as 0. In this situation, the perplexity is infinite.
- At the baseline, the model predicts a uniform distribution over all tokens. In this case, the perplexity equals the size of the dictionary `len(vocab)`. In fact, if we were to store the sequence without any compression, this would be the best we could do to encode it. Hence, this provides a nontrivial upper bound that any model must satisfy.

Summary

- A network that uses recurrent computation is called a recurrent neural network (RNN).
- The hidden state of the RNN can capture historical information of the sequence up to the current timestep.
- The number of RNN model parameters does not grow as the number of timesteps increases.
- We can create language models using a character-level RNN.

Exercises

1. If we use an RNN to predict the next character in a text sequence, how many output dimensions do we need?
2. Can you design a mapping for which an RNN with hidden states is exact? Hint: what about a finite number of words?
3. What happens to the gradient if you backpropagate through a long sequence?
4. What are some of the problems associated with the simple sequence model described above?

8.5 Implementation of Recurrent Neural Networks from Scratch

In this section we implement a language model introduce in Chapter 8 from scratch. It is based on a character-level recurrent neural network trained on H. G. Wells' *The Time Machine*. As before, we start by reading the dataset first, which is introduced in Section 8.3.

```
%matplotlib inline
import d2l
import math
from mxnet import autograd, np, npx, gluon
npx.set_np()

batch_size, num_steps = 32, 35
train_iter, vocab = d2l.load_data_time_machine(batch_size, num_steps)
```

8.5.1 One-hot Encoding

Remember that each token is presented as a numerical index in `train_iter`. Feeding these indices directly to the neural network might make it hard to learn. We often present each token as a more expressive feature vector. The easiest presentation is called *one-hot encoding*.

In a nutshell, we map each index to a different unit vector: assume that the number of different tokens in the vocabulary is N (the `len(vocab)`) and the token indices range from 0 to $N - 1$. If the index of a token is the integer i, then we create a vector $\mathbf{e}_i$ of all 0s with a length of N and set the element at position i to 1. This vector is the one-hot vector of the original token. The one-hot vectors with indices 0 and 2 are shown below.

```
npx.one_hot(np.array([0, 2]), len(vocab))
```

```
array([[1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.]])
```

The shape of the minibatch we sample each time is (batch size, timestep). The `one_hot` function transforms such a minibatch into a 3-D tensor with the last dimension equals to the vocabulary size. We often transpose the input so that we will obtain a (timestep, batch size, vocabulary size) output that fits into a sequence model easier.

```
X = np.arange(batch_size * num_steps).reshape(batch_size, num_steps)
npx.one_hot(X.T, len(vocab)).shape
```

```
(35, 32, 28)
```

8.5.2 Initializing the Model Parameters

Next, we initialize the model parameters for a RNN model. The number of hidden units `num_hiddens` is a tunable parameter.

```
def get_params(vocab_size, num_hiddens, ctx):
    num_inputs = num_outputs = vocab_size

    def normal(shape):
        return np.random.normal(scale=0.01, size=shape, ctx=ctx)
    # Hidden layer parameters
    W_xh = normal((num_inputs, num_hiddens))
    W_hh = normal((num_hiddens, num_hiddens))
    b_h = np.zeros(num_hiddens, ctx=ctx)
    # Output layer parameters
    W_hq = normal((num_hiddens, num_outputs))
    b_q = np.zeros(num_outputs, ctx=ctx)
    # Attach gradients
    params = [W_xh, W_hh, b_h, W_hq, b_q]
    for param in params:
        param.attach_grad()
    return params
```

8.5.3 RNN Model

First, we need an `init_rnn_state` function to return the hidden state at initialization. It returns an `ndarray` filled with 0 and with a shape of (batch size, number of hidden units). Using tuples makes it easier to handle situations where the hidden state contains multiple variables (e.g., when combining multiple layers in an RNN where each layer requires initializing).

```
def init_rnn_state(batch_size, num_hiddens, ctx):
    return (np.zeros(shape=(batch_size, num_hiddens), ctx=ctx), )
```

The following `rnn` function defines how to compute the hidden state and output in a timestep. The activation function here uses the tanh function. As described in Section 4.1, the mean value of the tanh function is 0, when the elements are evenly distributed over the real numbers.

```
def rnn(inputs, state, params):
    # Inputs shape: (num_steps, batch_size, vocab_size)
    W_xh, W_hh, b_h, W_hq, b_q = params
    H, = state
    outputs = []
    for X in inputs:
        H = np.tanh(np.dot(X, W_xh) + np.dot(H, W_hh) + b_h)
        Y = np.dot(H, W_hq) + b_q
        outputs.append(Y)
    return np.concatenate(outputs, axis=0), (H,)
```

Now we have all functions defined, next we create a class to wrap these functions and store parameters.

```
# Saved in the d2l package for later use
class RNNModelScratch(object):
    """A RNN Model based on scratch implementations"""

    def __init__(self, vocab_size, num_hiddens, ctx,
                 get_params, init_state, forward):
        self.vocab_size, self.num_hiddens = vocab_size, num_hiddens
        self.params = get_params(vocab_size, num_hiddens, ctx)
        self.init_state, self.forward_fn = init_state, forward

    def __call__(self, X, state):
        X = npx.one_hot(X.T, self.vocab_size)
        return self.forward_fn(X, state, self.params)

    def begin_state(self, batch_size, ctx):
        return self.init_state(batch_size, self.num_hiddens, ctx)
```

Let's do a sanity check whether inputs and outputs have the correct dimensions, e.g., to ensure that the dimensionality of the hidden state has not changed.

```
vocab_size, num_hiddens, ctx = len(vocab), 512, d2l.try_gpu()
model = RNNModelScratch(len(vocab), num_hiddens, ctx, get_params,
                        init_rnn_state, rnn)
state = model.begin_state(X.shape[0], ctx)
Y, new_state = model(X.as_in_context(ctx), state)
Y.shape, len(new_state), new_state[0].shape
```

```
((1120, 28), 1, (32, 512))
```

We can see that the output shape is (number steps × batch size, vocabulary size), while the hidden state shape remains the same, i.e., (batch size, number of hidden units).

8.5.4 Prediction

We first explain the predicting function so we can regularly check the prediction during training. This function predicts the next `num_predicts` characters based on the `prefix` (a string containing several characters). For the beginning of the sequence, we only update the hidden state. After that we begin generating new characters and emitting them.

```
# Saved in the d2l package for later use
def predict_ch8(prefix, num_predicts, model, vocab, ctx):
    state = model.begin_state(batch_size=1, ctx=ctx)
    outputs = [vocab[prefix[0]]]

    def get_input():
        return np.array([outputs[-1]], ctx=ctx).reshape(1, 1)
    for y in prefix[1:]:  # Warmup state with prefix
        _, state = model(get_input(), state)
        outputs.append(vocab[y])
    for _ in range(num_predicts):  # Predict num_predicts steps
        Y, state = model(get_input(), state)
        outputs.append(int(Y.argmax(axis=1).reshape(1)))
    return ''.join([vocab.idx_to_token[i] for i in outputs])
```

We test the `predict_rnn` function first. Given that we did not train the network it will generate nonsensical predictions. We initialize it with the sequence `traveller` and have it generate 10 additional characters.

```
predict_ch8('time traveller ', 10, model, vocab, ctx)
```

```
'time traveller iiiiiiiii'
```

8.5.5 Gradient Clipping

For a sequence of length T, we compute the gradients over these T timesteps in an iteration, which results in a chain of matrix-products with length $\mathcal{O}(T)$ during backpropagating. As mentioned in Section 4.8, it might result in numerical instability, e.g., the gradients may either explode or vanish, when T is large. Therefore, RNN models often need extra help to stabilize the training.

Recall that when solving an optimization problem, we take update steps for the weights $\mathbf{w}$ in the general direction of the negative gradient $\mathbf{g}_t$ on a minibatch, say $\mathbf{w} - \eta \cdot \mathbf{g}_t$. Let's further assume that the objective is well behaved, i.e., it is Lipschitz continuous with constant L, i.e.,

$$|l(\mathbf{w}) - l(\mathbf{w}')| \leq L\|\mathbf{w} - \mathbf{w}'\|. \tag{8.5.1}$$

In this case we can safely assume that if we update the weight vector by $\eta \cdot \mathbf{g}_t$, we will not observe a change by more than $L\eta\|\mathbf{g}_t\|$. This is both a curse and a blessing. A curse since it limits the speed of making progress, whereas a blessing since it limits the extent to which things can go wrong if we move in the wrong direction.

Sometimes the gradients can be quite large and the optimization algorithm may fail to converge. We could address this by reducing the learning rate η or by some other higher order trick. But what if we only rarely get large gradients? In this case such an approach may appear entirely unwarranted. One alternative is to clip the gradients by projecting them back to a ball of a given radius, say θ via

$$\mathbf{g} \leftarrow \min\left(1, \frac{\theta}{\|\mathbf{g}\|}\right) \mathbf{g}. \tag{8.5.2}$$

By doing so we know that the gradient norm never exceeds θ and that the updated gradient is entirely aligned with the original direction $\mathbf{g}$. It also has the desirable side-effect of limiting the influence any given minibatch (and within it any given sample) can exert on the weight vectors. This bestows a certain degree of robustness to the model. Gradient clipping provides a quick fix to the gradient exploding. While it does not entire solve the problem, it is one of the many techniques to alleviate it.

Below we define a function to clip the gradients of a model that is either a `RNNModelScratch` instance or a Gluon model. Also note that we compute the gradient norm over all parameters.

```python
# Saved in the d2l package for later use
def grad_clipping(model, theta):
    if isinstance(model, gluon.Block):
        params = [p.data() for p in model.collect_params().values()]
    else:
        params = model.params
    norm = math.sqrt(sum((p.grad ** 2).sum() for p in params))
    if norm > theta:
        for param in params:
            param.grad[:] *= theta / norm
```

8.5.6 Training

Let's first define the function to train the model on one data epoch. It differs from the models training of Section 3.6 in three places:

1. Different sampling methods for sequential data (independent sampling and sequential partitioning) will result in differences in the initialization of hidden states.
2. We clip the gradients before updating the model parameters. This ensures that the model does not diverge even when gradients blow up at some point during the training process, and it effectively reduces the step size automatically.
3. We use perplexity to evaluate the model. This ensures that sequences of different length are comparable.

When the consecutive sampling is used, we initialize the hidden state at the beginning of each epoch. Since the i^th example in the next minibatch is adjacent to the current i^th example, so the next minibatch can use the current hidden state directly, we only detach the gradient so that we compute the gradients within a minibatch. When using the random sampling, we need to reinitialize the hidden state for each iteration since each example is sampled with a random position. Same as the `train_epoch_ch3` function in Section 3.6, we use generalized `updater`, which could be either a Gluon trainer or a scratched implementation.

```python
# Saved in the d2l package for later use
def train_epoch_ch8(model, train_iter, loss, updater, ctx, use_random_iter):
    state, timer = None, d2l.Timer()
    metric = d2l.Accumulator(2)  # loss_sum, num_examples
    for X, Y in train_iter:
        if state is None or use_random_iter:
            # Initialize state when either it is the first iteration or
            # using random sampling.
            state = model.begin_state(batch_size=X.shape[0], ctx=ctx)
        else:
            for s in state:
                s.detach()
        y = Y.T.reshape(-1)
        X, y = X.as_in_context(ctx), y.as_in_context(ctx)
        with autograd.record():
            py, state = model(X, state)
            l = loss(py, y).mean()
        l.backward()
        grad_clipping(model, 1)
        updater(batch_size=1)  # Since used mean already
        metric.add(l * y.size, y.size)
    return math.exp(metric[0]/metric[1]), metric[1]/timer.stop()
```

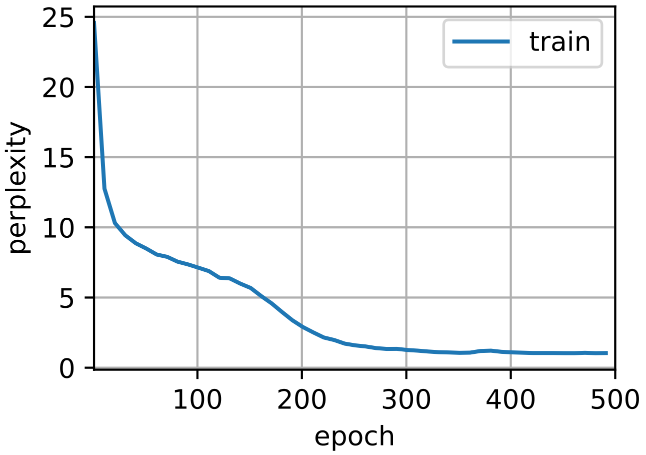
The training function again supports either we implement the model from scratch or using Gluon.

```python
# Saved in the d2l package for later use
def train_ch8(model, train_iter, vocab, lr, num_epochs, ctx,
              use_random_iter=False):
    # Initialize
    loss = gluon.loss.SoftmaxCrossEntropyLoss()
    animator = d2l.Animator(xlabel='epoch', ylabel='perplexity',
                            legend=['train'], xlim=[1, num_epochs])
    if isinstance(model, gluon.Block):
        model.initialize(ctx=ctx, force_reinit=True, init=init.Normal(0.01))
        trainer = gluon.Trainer(model.collect_params(),
                                'sgd', {'learning_rate': lr})

        def updater(batch_size):
            return trainer.step(batch_size)
    else:
        def updater(batch_size):
            return d2l.sgd(model.params, lr, batch_size)

    def predict(prefix):
        return predict_ch8(prefix, 50, model, vocab, ctx)

    # Train and check the progress.
    for epoch in range(num_epochs):
        ppl, speed = train_epoch_ch8(
            model, train_iter, loss, updater, ctx, use_random_iter)
        if epoch % 10 == 0:
            print(predict('time traveller'))
            animator.add(epoch+1, [ppl])
    print('Perplexity %.1f, %d tokens/sec on %s' % (ppl, speed, ctx))
    print(predict('time traveller'))
    print(predict('traveller'))
```

Now we can train a model. Since we only use $10,000$ tokens in the dataset, so here the model needs more epochs to converge.

```
num_epochs, lr = 500, 1
train_ch8(model, train_iter, vocab, lr, num_epochs, ctx)
```

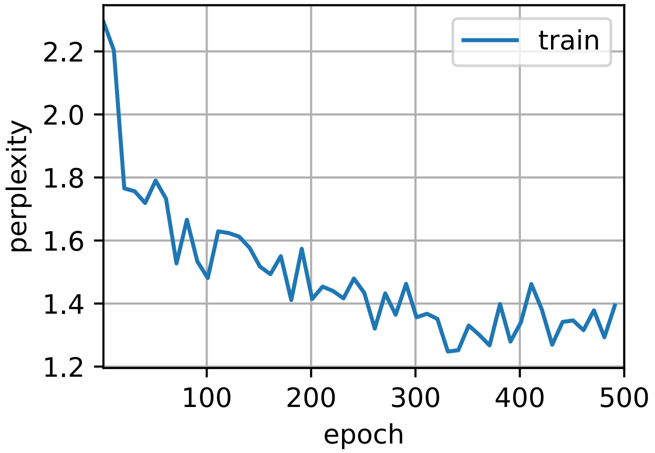
```
Perplexity 1.0, 38730 tokens/sec on gpu(0)
time traveller  it s against reason said filby  what reason said
traveller  it s against reason said filby  what reason said
```



Finally let's check the results to use a random sampling iterator.

```
train_ch8(model, train_iter, vocab, lr, num_epochs, ctx, use_random_iter=True)
```

```
Perplexity 1.3, 38212 tokens/sec on gpu(0)
time traveller after the pauserequired for the proper assimilati
traveller  you can show black is white by argument said fil
```



While implementing the above RNN model from scratch is instructive, it is not convenient. In the next section we will see how to improve significantly on the current model and how to make it faster and easier to implement.

Summary

- Sequence models need state initialization for training.
- Between sequential models you need to ensure to detach the gradients, to ensure that the automatic differentiation does not propagate effects beyond the current sample.
- A simple RNN language model consists of an encoder, an RNN model, and a decoder.
- Gradient clipping prevents gradient explosion (but it cannot fix vanishing gradients).
- Perplexity calibrates model performance across different sequence length. It is the exponentiated average of the cross-entropy loss.
- Sequential partitioning typically leads to better models.

Exercises

1. Show that one-hot encoding is equivalent to picking a different embedding for each object.
2. Adjust the hyperparameters to improve the perplexity.
    - How low can you go? Adjust embeddings, hidden units, learning rate, etc.
    - How well will it work on other books by H. G. Wells, e.g., The War of the Worlds[132].
3. Modify the predict function such as to use sampling rather than picking the most likely next character.
    - What happens?
    - Bias the model towards more likely outputs, e.g., by sampling from $q(w_t \mid w_{t-1}, \ldots, w_1) \propto p^\alpha(w_t \mid w_{t-1}, \ldots, w_1)$ for $\alpha > 1$.
4. Run the code in this section without clipping the gradient. What happens?
5. Change adjacent sampling so that it does not separate hidden states from the computational graph. Does the running time change? How about the accuracy?
6. Replace the activation function used in this section with ReLU and repeat the experiments in this section.
7. Prove that the perplexity is the inverse of the harmonic mean of the conditional word probabilities.

[132] http://www.gutenberg.org/ebooks/36

8.6 Concise Implementation of Recurrent Neural Networks

While Section 8.5 was instructive to see how recurrent neural networks (RNNs) are implemented, this is not convenient or fast. This section will show how to implement the same language model more efficiently using functions provided by Gluon. We begin as before by reading the "Time Machine" corpus.

```
import d2l
from mxnet import np, npx
from mxnet.gluon import nn, rnn
npx.set_np()

batch_size, num_steps = 32, 35
train_iter, vocab = d2l.load_data_time_machine(batch_size, num_steps)
```

8.6.1 Defining the Model

Gluon's `rnn` module provides a recurrent neural network implementation (beyond many other sequence models). We construct the recurrent neural network layer `rnn_layer` with a single hidden layer and 256 hidden units, and initialize the weights.

```
num_hiddens = 256
rnn_layer = rnn.RNN(num_hiddens)
rnn_layer.initialize()
```

Initializing the state is straightforward. We invoke the member function `rnn_layer.begin_state(batch_size)`. This returns an initial state for each element in the minibatch. That is, it returns an object of size (hidden layers, batch size, number of hidden units). The number of hidden layers defaults to be 1. In fact, we have not even discussed yet what it means to have multiple layers—this will happen in Section 9.3. For now, suffice it to say that multiple layers simply amount to the output of one RNN being used as the input for the next RNN.

```
batch_size = 1
state = rnn_layer.begin_state(batch_size=batch_size)
len(state), state[0].shape
```

```
(1, (1, 1, 256))
```

With a state variable and an input, we can compute the output with the updated state.

```
num_steps = 1
X = np.random.uniform(size=(num_steps, batch_size, len(vocab)))
Y, state_new = rnn_layer(X, state)
Y.shape, len(state_new), state_new[0].shape
```

```
((1, 1, 256), 1, (1, 1, 256))
```

Similar to Section 8.5, we define an `RNNModel` block by subclassing the `Block` class for a complete recurrent neural network. Note that `rnn_layer` only contains the hidden recurrent layers, we need to create a separate output layer. While in the previous section, we have the output layer within the rnn block.

```
# Saved in the d2l package for later use
class RNNModel(nn.Block):
    def __init__(self, rnn_layer, vocab_size, **kwargs):
        super(RNNModel, self).__init__(**kwargs)
        self.rnn = rnn_layer
        self.vocab_size = vocab_size
        self.dense = nn.Dense(vocab_size)

    def forward(self, inputs, state):
        X = npx.one_hot(inputs.T, self.vocab_size)
        Y, state = self.rnn(X, state)
        # The fully connected layer will first change the shape of Y to
        # (num_steps * batch_size, num_hiddens). Its output shape is
        # (num_steps * batch_size, vocab_size).
        output = self.dense(Y.reshape(-1, Y.shape[-1]))
        return output, state

    def begin_state(self, *args, **kwargs):
        return self.rnn.begin_state(*args, **kwargs)
```

8.6.2 Training and Predicting

Before training the model, let's make a prediction with the a model that has random weights.

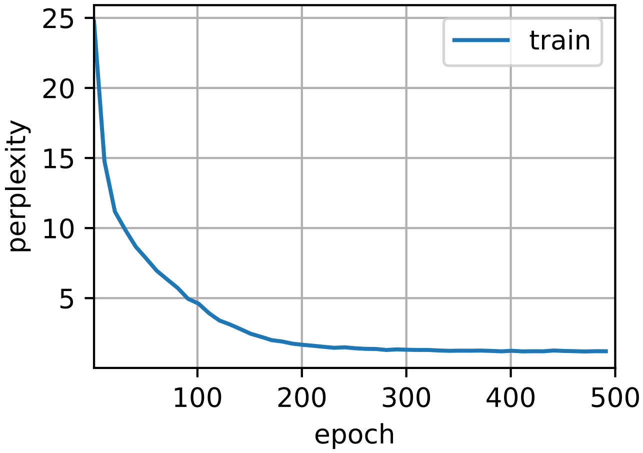
```
ctx = d2l.try_gpu()
model = RNNModel(rnn_layer, len(vocab))
model.initialize(force_reinit=True, ctx=ctx)
d2l.predict_ch8('time traveller', 10, model, vocab, ctx)
```

```
'time travellervmjznnngii'
```

As is quite obvious, this model does not work at all. Next, we call train_ch8 with the same hyperparameters defined in Section 8.5 and train our model with Gluon.

```
num_epochs, lr = 500, 1
d2l.train_ch8(model, train_iter, vocab, lr, num_epochs, ctx)
```

```
Perplexity 1.2, 195791 tokens/sec on gpu(0)
time traveller  you can show black is white by argument said fil
travellerepresent moment aut for the historian thepsycholog
```



Compared with the last section, this model achieves comparable perplexity, albeit within a shorter period of time, due to the code being more optimized.

Summary

- Gluon's `rnn` module provides an implementation at the recurrent neural network layer.
- Gluon's `nn.RNN` instance returns the output and hidden state after forward computation. This forward computation does not involve output layer computation.
- As before, the computational graph needs to be detached from previous steps for reasons of efficiency.

Exercises

1. Compare the implementation with the previous section.
    - Why does Gluon's implementation run faster?
    - If you observe a significant difference beyond speed, try to find the reason.
2. Can you make the model overfit?
    - Increase the number of hidden units.
    - Increase the number of iterations.
    - What happens if you adjust the clipping parameter?
3. Implement the autoregressive model of the introduction to the current chapter using an RNN.
4. What happens if you increase the number of hidden layers in the RNN model? Can you make the model work?
5. How well can you compress the text using this model?
    - How many bits do you need?
    - Why does not everyone use this model for text compression? Hint: what about the compressor itself?

8.7 Backpropagation Through Time

So far we repeatedly alluded to things like *exploding gradients*, *vanishing gradients*, *truncating backprop*, and the need to *detach the computational graph*. For instance, in the previous section we invoked `s.detach()` on the sequence. None of this was really fully explained, in the interest of being able to build a model quickly and to see how it works. In this section we will delve a bit more deeply into the details of backpropagation for sequence models and why (and how) the math works. For a more detailed discussion about randomization and backpropagation also see the paper by (Tallec & Ollivier, 2017).

We encountered some of the effects of gradient explosion when we first implemented recurrent neural networks (Section 8.5). In particular, if you solved the problems in the problem set, you would have seen that gradient clipping is vital to ensure proper convergence. To provide a better understanding of this issue, this section will review how gradients are computed for sequence models. Note that there is nothing conceptually new in how it works. After all, we are still merely applying the chain rule to compute gradients. Nonetheless, it is worth while reviewing backpropagation (Section 4.7) again.

Forward propagation in a recurrent neural network is relatively straightforward. *Backpropagation through time* is actually a specific application of back propagation in recurrent neural networks. It requires us to expand the recurrent neural network one timestep at a time to obtain the dependencies between model variables and parameters. Then, based on the chain rule, we apply backpropagation to compute and store gradients. Since sequences can be rather long, the dependency can be rather lengthy. For instance, for a sequence of 1000 characters, the first symbol could potentially have significant influence on the symbol at position 1000. This is not really computationally feasible (it takes too long and requires too much memory) and it requires over 1000 matrix-vector products before we would arrive at that very elusive gradient. This is a process fraught with computational and statistical uncertainty. In the following we will elucidate what happens and how to address this in practice.

8.7.1 A Simplified Recurrent Network

We start with a simplified model of how an RNN works. This model ignores details about the specifics of the hidden state and how it is updated. These details are immaterial to the analysis and would only serve to clutter the notation, but make it look more intimidating. In this simplified model, we denote h_t as the hidden state, x_t as the input, and o_t as the output at timestep t. In addition, w_h and w_o indicate the weights of hidden states and the output layer, respectively. As a result, the hidden states and outputs at each timesteps can be explained as

$$h_t = f(x_t, h_{t-1}, w_h) \text{ and } o_t = g(h_t, w_o). \tag{8.7.1}$$

Hence, we have a chain of values $\{\ldots, (h_{t-1}, x_{t-1}, o_{t-1}), (h_t, x_t, o_t), \ldots\}$ that depend on each other via recursive computation. The forward pass is fairly straightforward. All we need is to loop through the (x_t, h_t, o_t) triples one step at a time. The discrepancy between outputs o_t and the desired targets y_t is then evaluated by an objective function as

$$L(x, y, w_h, w_o) = \sum_{t=1}^{T} l(y_t, o_t). \tag{8.7.2}$$

For backpropagation, matters are a bit more tricky, especially when we compute the gradients with regard to the parameters w_h of the objective function L. To be specific, by the chain rule,

$$\begin{aligned}\partial_{w_h} L &= \sum_{t=1}^{T} \partial_{w_h} l(y_t, o_t) \\ &= \sum_{t=1}^{T} \partial_{o_t} l(y_t, o_t) \partial_{h_t} g(h_t, w_h) \left[\partial_{w_h} h_t \right].\end{aligned} \tag{8.7.3}$$

The first and the second part of the derivative is easy to compute. The third part $\partial_{w_h} h_t$ is where things get tricky, since we need to compute the effect of the parameters on h_t.

To derive the above gradient, assume that we have three sequences $\{a_t\}, \{b_t\}, \{c_t\}$ satisfying $a_0 = 0, a_1 = b_1$, and $a_t = b_t + c_t a_{t-1}$ for $t = 1, 2, \ldots$. Then for $t \geq 1$, it is easy to show

$$a_t = b_t + \sum_{i=1}^{t-1} \left(\prod_{j=i+1}^{t} c_j \right) b_i. \tag{8.7.4}$$

Now let's apply (8.7.4) with

$$a_t = \partial_{w_h} h_t, \tag{8.7.5}$$

$$b_t = \partial_{w_h} f(x_t, h_{t-1}, w_h), \tag{8.7.6}$$

$$c_t = \partial_{h_{t-1}} f(x_t, h_{t-1}, w_h). \tag{8.7.7}$$

Therefore, $a_t = b_t + c_t a_{t-1}$ becomes the following recursion

$$\partial_{w_h} h_t = \partial_{w_h} f(x_t, h_{t-1}, w) + \partial_h f(x_t, h_{t-1}, w_h) \partial_{w_h} h_{t-1}. \tag{8.7.8}$$

By (8.7.4), the third part will be

$$\partial_{w_h} h_t = \partial_{w_h} f(x_t, h_{t-1}, w_h) + \sum_{i=1}^{t-1} \left(\prod_{j=i+1}^{t} \partial_{h_{j-1}} f(x_j, h_{j-1}, w_h) \right) \partial_{w_h} f(x_i, h_{i-1}, w_h). \tag{8.7.9}$$

While we can use the chain rule to compute $\partial_w h_t$ recursively, this chain can get very long whenever t is large. Let's discuss a number of strategies for dealing with this problem.

- **Compute the full sum.** This is very slow and gradients can blow up, since subtle changes in the initial conditions can potentially affect the outcome a lot. That is, we could see things similar to the butterfly effect where minimal changes in the initial conditions lead to disproportionate changes in the outcome. This is actually quite undesirable in terms of the model that we want to estimate. After all, we are looking for robust estimators that generalize well. Hence this strategy is almost never used in practice.

- **Truncate the sum after τ steps.** This is what we have been discussing so far. This leads to an *approximation* of the true gradient, simply by terminating the sum above at $\partial_w h_{t-\tau}$. The approximation error is thus given by $\partial_h f(x_t, h_{t-1}, w)\partial_w h_{t-1}$ (multiplied by a product of gradients involving $\partial_h f$). In practice this works quite well. It is what is commonly referred to as truncated BPTT (backpropgation through time). One of the consequences of this is that the model focuses primarily on short-term influence rather than long-term consequences. This is actually *desirable*, since it biases the estimate towards simpler and more stable models.
- **Randomized Truncation.** Last we can replace $\partial_{w_h} h_t$ by a random variable which is correct in expectation but which truncates the sequence. This is achieved by using a sequence of ξ_t where $E[\xi_t] = 1$ and $P(\xi_t = 0) = 1 - \pi$ and furthermore $P(\xi_t = \pi^{-1}) = \pi$. We use this to replace the gradient:

$$z_t = \partial_w f(x_t, h_{t-1}, w) + \xi_t \partial_h f(x_t, h_{t-1}, w)\partial_w h_{t-1}. \tag{8.7.10}$$

It follows from the definition of ξ_t that $E[z_t] = \partial_w h_t$. Whenever $\xi_t = 0$ the expansion terminates at that point. This leads to a weighted sum of sequences of varying lengths where long sequences are rare but appropriately overweighted. (Tallec & Ollivier, 2017) proposed this in their paper. Unfortunately, while appealing in theory, the model does not work much better than simple truncation, most likely due to a number of factors. First, the effect of an observation after a number of backpropagation steps into the past is quite sufficient to capture dependencies in practice. Second, the increased variance counteracts the fact that the gradient is more accurate. Third, we actually *want* models that have only a short range of interaction. Hence, BPTT has a slight regularizing effect which can be desirable.

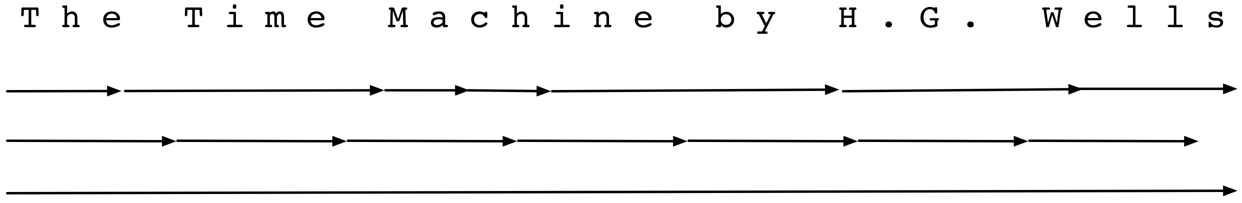


Fig. 8.7.1: From top to bottom: randomized BPTT, regularly truncated BPTT and full BPTT

Fig. 8.7.1 illustrates the three cases when analyzing the first few words of *The Time Machine*: * The first row is the randomized truncation which partitions the text into segments of varying length. * The second row is the regular truncated BPTT which breaks it into sequences of the same length. * The third row is the full BPTT that leads to a computationally infeasible expression.

8.7.2 The Computational Graph

In order to visualize the dependencies between model variables and parameters during computation in a recurrent neural network, we can draw a computational graph for the model, as shown in Fig. 8.7.2. For example, the computation of the hidden states of timestep 3, $\mathbf{h}_3$, depends on the model parameters $\mathbf{W}_{hx}$ and $\mathbf{W}_{hh}$, the hidden state of the last timestep $\mathbf{h}_2$, and the input of the current timestep $\mathbf{x}_3$.

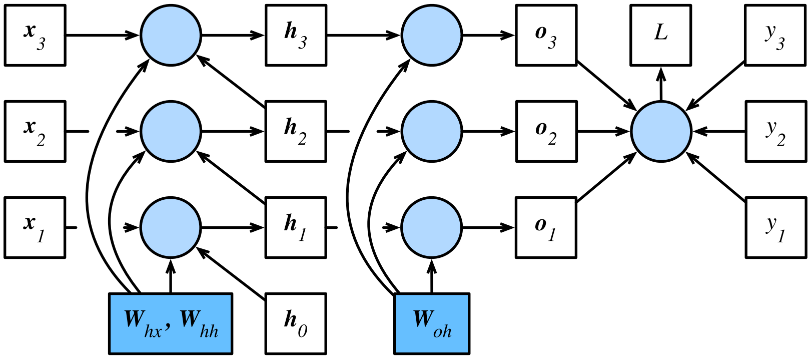


Fig. 8.7.2: Computational dependencies for a recurrent neural network model with three timesteps. Boxes represent variables (not shaded) or parameters (shaded) and circles represent operators.

8.7.3 BPTT in Detail

After discussing the general principle, let's discuss BPTT in detail. By decomposing **W** into different sets of weight matrices ($\mathbf{W}_{hx}, \mathbf{W}_{hh}$ and $\mathbf{W}_{oh}$), we will get a simple linear latent variable model:

$$\mathbf{h}_t = \mathbf{W}_{hx}\mathbf{x}_t + \mathbf{W}_{hh}\mathbf{h}_{t-1} \text{ and } \mathbf{o}_t = \mathbf{W}_{oh}\mathbf{h}_t. \tag{8.7.11}$$

Following the discussion in Section 4.7, we compute the gradients $\frac{\partial L}{\partial \mathbf{W}_{hx}}, \frac{\partial L}{\partial \mathbf{W}_{hh}}, \frac{\partial L}{\partial \mathbf{W}_{oh}}$ for

$$L(\mathbf{x}, \mathbf{y}, \mathbf{W}) = \sum_{t=1}^{T} l(\mathbf{o}_t, y_t), \tag{8.7.12}$$

where $l(\cdot)$ denotes the chosen loss function. Taking the derivatives with respect to W_{oh} is fairly straightforward and we obtain

$$\partial_{\mathbf{W}_{oh}} L = \sum_{t=1}^{T} \text{prod}\left(\partial_{\mathbf{o}_t} l(\mathbf{o}_t, y_t), \mathbf{h}_t\right), \tag{8.7.13}$$

where prod$(\cdot)$ indicates the product of two or more matrices.

The dependency on $\mathbf{W}_{hx}$ and $\mathbf{W}_{hh}$ is a bit more tricky since it involves a chain of derivatives. We begin with

$$\begin{aligned}\partial_{\mathbf{W}_{hh}} L &= \sum_{t=1}^{T} \text{prod}\left(\partial_{\mathbf{o}_t} l(\mathbf{o}_t, y_t), \mathbf{W}_{oh}, \partial_{\mathbf{W}_{hh}} \mathbf{h}_t\right), \\ \partial_{\mathbf{W}_{hx}} L &= \sum_{t=1}^{T} \text{prod}\left(\partial_{\mathbf{o}_t} l(\mathbf{o}_t, y_t), \mathbf{W}_{oh}, \partial_{\mathbf{W}_{hx}} \mathbf{h}_t\right).\end{aligned} \tag{8.7.14}$$

After all, hidden states depend on each other and on past inputs. The key quantity is how past hidden states affect future hidden states.

$$\partial_{\mathbf{h}_t} \mathbf{h}_{t+1} = \mathbf{W}_{hh}^\top \text{ and thus } \partial_{\mathbf{h}_t} \mathbf{h}_T = \left(\mathbf{W}_{hh}^\top\right)^{T-t}. \tag{8.7.15}$$

Chaining terms together yields

$$\begin{aligned}
\partial_{\mathbf{W}_{hh}} \mathbf{h}_t &= \sum_{j=1}^t \left(\mathbf{W}_{hh}^\top\right)^{t-j} \mathbf{h}_j \\
\partial_{\mathbf{W}_{hx}} \mathbf{h}_t &= \sum_{j=1}^t \left(\mathbf{W}_{hh}^\top\right)^{t-j} \mathbf{x}_j.
\end{aligned} \tag{8.7.16}$$

A number of things follow from this potentially very intimidating expression. First, it pays to store intermediate results, i.e., powers of $\mathbf{W}_{hh}$ as we work our way through the terms of the loss function L. Second, this simple linear example already exhibits some key problems of long sequence models: it involves potentially very large powers $\mathbf{W}_{hh}^j$. In it, eigenvalues smaller than 1 vanish for large j and eigenvalues larger than 1 diverge. This is numerically unstable and gives undue importance to potentially irrelevant past detail. One way to address this is to truncate the sum at a computationally convenient size. Later on in Chapter 9 we will see how more sophisticated sequence models such as LSTMs can alleviate this further. In practice, this truncation is effected by *detaching* the gradient after a given number of steps.

Summary

- Backpropagation through time is merely an application of backprop to sequence models with a hidden state.
- Truncation is needed for computational convenience and numerical stability.
- High powers of matrices can lead to divergent and vanishing eigenvalues. This manifests itself in the form of exploding or vanishing gradients.
- For efficient computation, intermediate values are cached.

Exercises

1. Assume that we have a symmetric matrix $\mathbf{M} \in \mathbb{R}^{n \times n}$ with eigenvalues λ_i. Without loss of generality, assume that they are ordered in ascending order $\lambda_i \leq \lambda_{i+1}$. Show that $\mathbf{M}^k$ has eigenvalues λ_i^k.
2. Prove that for a random vector $\mathbf{x} \in \mathbb{R}^n$, with high probability $\mathbf{M}^k \mathbf{x}$ will be very much aligned with the largest eigenvector $\mathbf{v}_n$ of $\mathbf{M}$. Formalize this statement.
3. What does the above result mean for gradients in a recurrent neural network?
4. Besides gradient clipping, can you think of any other methods to cope with gradient explosion in recurrent neural networks?

9 | Modern Recurrent Neural Networks

Although we have learned the basics of recurrent neural networks, they are not sufficient for a practitioner to solve today's sequence learning problems. For instance, given the numerical unstability during gradient calculation, gated recurrent neural networks much more common in practice. We will begin by introducing two of such widely-used networks, namely gated recurrent units (GRUs) and long short term memory (LSTM), with illustrations using the same language modeling problem as introduced in Chapter 8.

Furthermore, we will modify recurrent neural networks with a single undirectional hidden layer. We will describe deep architectures, and discuss the bidirectional design with both forward and backward recursion. They are frequently adopted in modern recurrent networks.

In fact, a large portion of sequence learning problems such as automatic speech recognition, text to speech, and machine translation, consider both inputs and outputs to be sequences of arbitrary length. Finally, we will take machine translation as an example, and introduce the encoder-decoder architecture based on recurrent neural networks and modern practices for such sequence to sequence learning problems.

9.1 Gated Recurrent Units (GRU)

In the previous section, we discussed how gradients are calculated in a recurrent neural network. In particular we found that long products of matrices can lead to vanishing or divergent gradients. Let's briefly think about what such gradient anomalies mean in practice:

- We might encounter a situation where an early observation is highly significant for predicting all future observations. Consider the somewhat contrived case where the first observation contains a checksum and the goal is to discern whether the checksum is correct at the end of the sequence. In this case, the influence of the first token is vital. We would like to have some mechanisms for storing vital early information in a *memory cell*. Without such a mechanism, we will have to assign a very large gradient to this observation, since it affects all subsequent observations.
- We might encounter situations where some symbols carry no pertinent observation. For instance, when parsing a web page there might be auxiliary HTML code that is irrelevant for the purpose of assessing the sentiment conveyed on the page. We would like to have some mechanism for *skipping such symbols* in the latent state representation.
- We might encounter situations where there is a logical break between parts of a sequence. For instance, there might be a transition between chapters in a book, or a transition between a bear, and a bull market for securities. In this case it would be nice to have a means of *resetting* our internal state representation.

A number of methods have been proposed to address this. One of the earliest is Long Short Term Memory (LSTM) (Hochreiter & Schmidhuber, 1997) which we will discuss in Section 9.2. Gated Recurrent Unit (GRU) (Cho et al., 2014) is a slightly more streamlined variant that often offers comparable performance and is significantly faster to compute. See also (Chung et al., 2014) for more details. Due to its simplicity, let's start with the GRU.

9.1.1 Gating the Hidden State

The key distinction between regular RNNs and GRUs is that the latter support gating of the hidden state. This means that we have dedicated mechanisms for when a hidden state should be updated and also when it should be reset. These mechanisms are learned and they address the concerns listed above. For instance, if the first symbol is of great importance we will learn not to update the hidden state after the first observation. Likewise, we will learn to skip irrelevant temporary observations. Last, we will learn to reset the latent state whenever needed. We discuss this in detail below.

Reset Gates and Update Gates

The first thing we need to introduce are reset and update gates. We engineer them to be vectors with entries in $(0, 1)$ such that we can perform convex combinations. For instance, a reset variable would allow us to control how much of the previous state we might still want to remember. Likewise, an update variable would allow us to control how much of the new state is just a copy of the old state.

We begin by engineering gates to generate these variables. Fig. 9.1.1 illustrates the inputs for both reset and update gates in a GRU, given the current timestep input $\mathbf{X}_t$ and the hidden state of the previous timestep $\mathbf{H}_{t-1}$. The output is given by a fully connected layer with a sigmoid as its activation function.

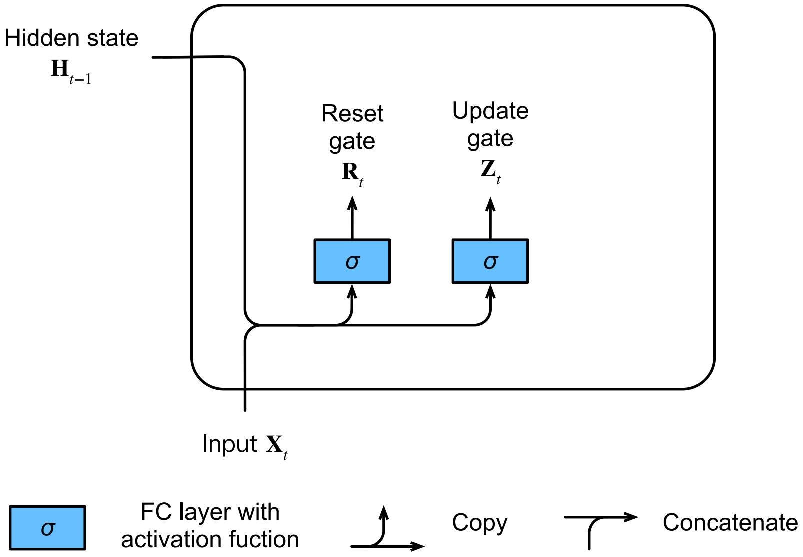

Hidden state $\mathbf{H}_{t-1}$
Reset gate $\mathbf{R}_t$
Update gate $\mathbf{Z}_t$
σ
σ
Input $\mathbf{X}_t$
σ
FC layer with activation fuction
Copy
Concatenate

Fig. 9.1.1: Reset and update gate in a GRU.

For a given timestep t, the minibatch input is $\mathbf{X}_t \in \mathbb{R}^{n \times d}$ (number of examples: n, number of inputs: d) and the hidden state of the last timestep is $\mathbf{H}_{t-1} \in \mathbb{R}^{n \times h}$ (number of hidden states: h).

Then, the reset gate $\mathbf{R}_t \in \mathbb{R}^{n\times h}$ and update gate $\mathbf{Z}_t \in \mathbb{R}^{n\times h}$ are computed as follows:

$$\begin{aligned}\mathbf{R}_t &= \sigma(\mathbf{X}_t\mathbf{W}_{xr} + \mathbf{H}_{t-1}\mathbf{W}_{hr} + \mathbf{b}_r),\\ \mathbf{Z}_t &= \sigma(\mathbf{X}_t\mathbf{W}_{xz} + \mathbf{H}_{t-1}\mathbf{W}_{hz} + \mathbf{b}_z).\end{aligned} \tag{9.1.1}$$

Here, $\mathbf{W}_{xr}, \mathbf{W}_{xz} \in \mathbb{R}^{d\times h}$ and $\mathbf{W}_{hr}, \mathbf{W}_{hz} \in \mathbb{R}^{h\times h}$ are weight parameters and $\mathbf{b}_r, \mathbf{b}_z \in \mathbb{R}^{1\times h}$ are biases. We use a sigmoid function (as introduced in Section 4.1) to transform input values to the interval $(0, 1)$.

Reset Gates in Action

We begin by integrating the reset gate with a regular latent state updating mechanism. In a conventional RNN, we would have an hidden state update of the form

$$\mathbf{H}_t = \tanh(\mathbf{X}_t\mathbf{W}_{xh} + \mathbf{H}_{t-1}\mathbf{W}_{hh} + \mathbf{b}_h). \tag{9.1.2}$$

This is essentially identical to the discussion of the previous section, albeit with a nonlinearity in the form of tanh to ensure that the values of the hidden states remain in the interval $(-1, 1)$. If we want to be able to reduce the influence of the previous states we can multiply $\mathbf{H}_{t-1}$ with $\mathbf{R}_t$ elementwise. Whenever the entries in the reset gate $\mathbf{R}_t$ are close to 1, we recover a conventional RNN. For all entries of the reset gate $\mathbf{R}_t$ that are close to 0, the hidden state is the result of an MLP with $\mathbf{X}_t$ as input. Any pre-existing hidden state is thus reset to defaults. This leads to the following *candidate hidden state* (it is a *candidate* since we still need to incorporate the action of the update gate).

$$\tilde{\mathbf{H}}_t = \tanh(\mathbf{X}_t\mathbf{W}_{xh} + (\mathbf{R}_t \odot \mathbf{H}_{t-1})\,\mathbf{W}_{hh} + \mathbf{b}_h). \tag{9.1.3}$$

Fig. 9.1.2 illustrates the computational flow after applying the reset gate. The symbol $\odot$ indicates pointwise multiplication between tensors.

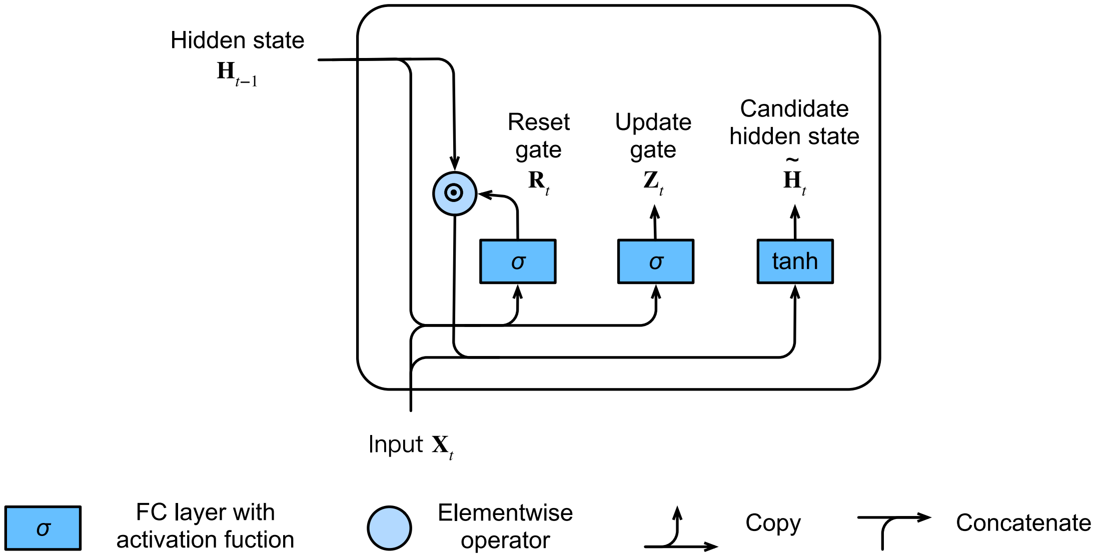


Fig. 9.1.2: Candidate hidden state computation in a GRU. The multiplication is carried out elementwise.

Update Gates in Action

Next we need to incorporate the effect of the update gate $\mathbf{Z}_t$, as shown in Fig. 9.1.3. This determines the extent to which the new state $\mathbf{H}_t$ is just the old state $\mathbf{H}_{t-1}$ and by how much the new candidate state $\tilde{\mathbf{H}}_t$ is used. The gating variable $\mathbf{Z}_t$ can be used for this purpose, simply by taking elementwise convex combinations between both candidates. This leads to the final update equation for the GRU.

$$\mathbf{H}_t = \mathbf{Z}_t \odot \mathbf{H}_{t-1} + (1 - \mathbf{Z}_t) \odot \tilde{\mathbf{H}}_t. \tag{9.1.4}$$

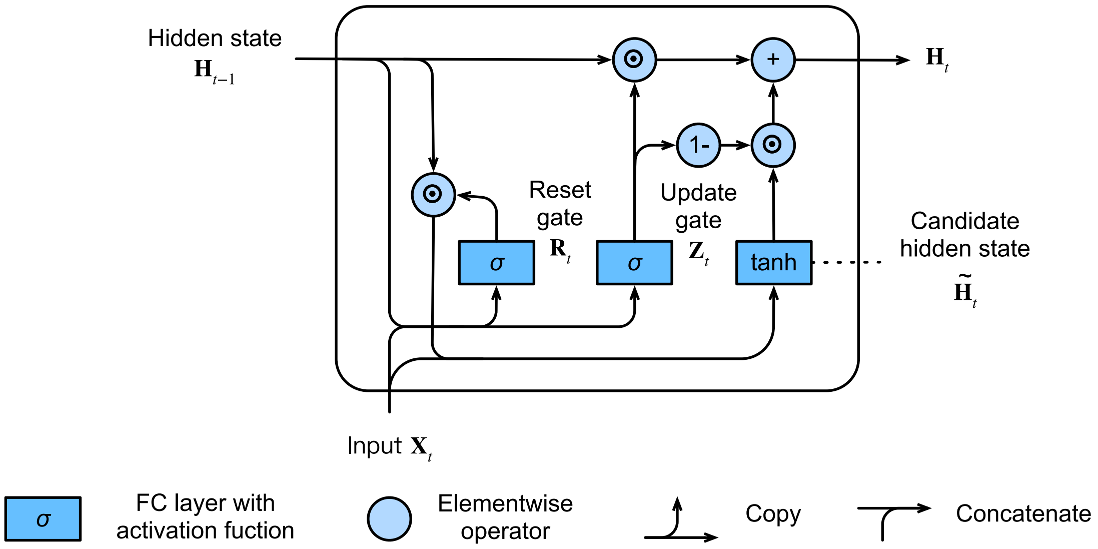


Fig. 9.1.3: Hidden state computation in a GRU. As before, the multiplication is carried out elementwise.

Whenever the update gate $\mathbf{Z}_t$ is close to 1, we simply retain the old state. In this case the information from $\mathbf{X}_t$ is essentially ignored, effectively skipping timestep t in the dependency chain. In contrast, whenever $\mathbf{Z}_t$ is close to 0, the new latent state $\mathbf{H}_t$ approaches the candidate latent state $\tilde{\mathbf{H}}_t$. These designs can help us cope with the vanishing gradient problem in RNNs and better capture dependencies for time series with large timestep distances. In summary, GRUs have the following two distinguishing features:

- Reset gates help capture short-term dependencies in time series.
- Update gates help capture long-term dependencies in time series.

9.1.2 Implementation from Scratch

To gain a better understanding of the model, let's implement a GRU from scratch.

Reading the Dataset

We begin by reading *The Time Machine* corpus that we used in Section 8.5. The code for reading the dataset is given below:

```
import d2l
from mxnet import np, npx
from mxnet.gluon import rnn
npx.set_np()

batch_size, num_steps = 32, 35
train_iter, vocab = d2l.load_data_time_machine(batch_size, num_steps)
```

Initializing Model Parameters

The next step is to initialize the model parameters. We draw the weights from a Gaussian with variance to be 0.01 and set the bias to 0. The hyperparameter `num_hiddens` defines the number of hidden units. We instantiate all weights and biases relating to the update gate, the reset gate, and the candidate hidden state itself. Subsequently, we attach gradients to all the parameters.

```
def get_params(vocab_size, num_hiddens, ctx):
    num_inputs = num_outputs = vocab_size

    def normal(shape):
        return np.random.normal(scale=0.01, size=shape, ctx=ctx)

    def three():
        return (normal((num_inputs, num_hiddens)),
                normal((num_hiddens, num_hiddens)),
                np.zeros(num_hiddens, ctx=ctx))

    W_xz, W_hz, b_z = three()  # Update gate parameter
    W_xr, W_hr, b_r = three()  # Reset gate parameter
    W_xh, W_hh, b_h = three()  # Candidate hidden state parameter
    # Output layer parameters
    W_hq = normal((num_hiddens, num_outputs))
    b_q = np.zeros(num_outputs, ctx=ctx)
    # Attach gradients
    params = [W_xz, W_hz, b_z, W_xr, W_hr, b_r, W_xh, W_hh, b_h, W_hq, b_q]
    for param in params:
        param.attach_grad()
    return params
```

Defining the Model

Now we will define the hidden state initialization function `init_gru_state`. Just like the `init_rnn_state` function defined in Section 8.5, this function returns an `ndarray` with a shape (batch size, number of hidden units) whose values are all zeros.

```
def init_gru_state(batch_size, num_hiddens, ctx):
    return (np.zeros(shape=(batch_size, num_hiddens), ctx=ctx), )
```

Now we are ready to define the GRU model. Its structure is the same as the basic RNN cell, except that the update equations are more complex.

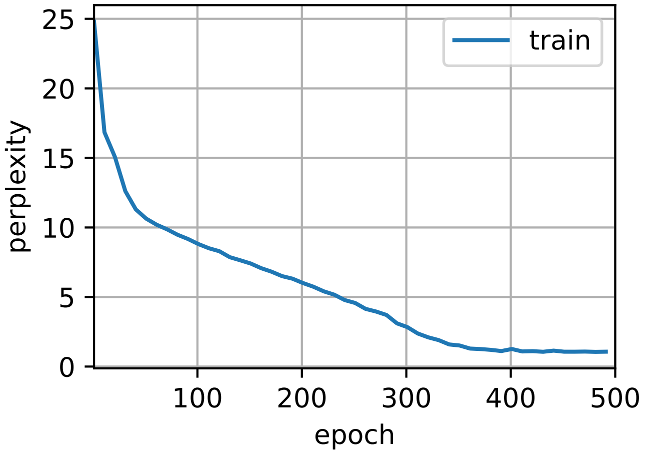
```
def gru(inputs, state, params):
    W_xz, W_hz, b_z, W_xr, W_hr, b_r, W_xh, W_hh, b_h, W_hq, b_q = params
    H, = state
    outputs = []
    for X in inputs:
        Z = npx.sigmoid(np.dot(X, W_xz) + np.dot(H, W_hz) + b_z)
        R = npx.sigmoid(np.dot(X, W_xr) + np.dot(H, W_hr) + b_r)
        H_tilda = np.tanh(np.dot(X, W_xh) + np.dot(R * H, W_hh) + b_h)
        H = Z * H + (1 - Z) * H_tilda
        Y = np.dot(H, W_hq) + b_q
        outputs.append(Y)
    return np.concatenate(outputs, axis=0), (H,)
```

Training and Prediction

Training and prediction work in exactly the same manner as before. After training for one epoch, the perplexity and the output sentence will be like the following.

```
vocab_size, num_hiddens, ctx = len(vocab), 256, d2l.try_gpu()
num_epochs, lr = 500, 1
model = d2l.RNNModelScratch(len(vocab), num_hiddens, ctx, get_params,
                            init_gru_state, gru)
d2l.train_ch8(model, train_iter, vocab, lr, num_epochs, ctx)
```

```
Perplexity 1.1, 14158 tokens/sec on gpu(0)
time traveller  it s against reason said filby  what reason said
traveller  it s against reason said filby  what reason said
```

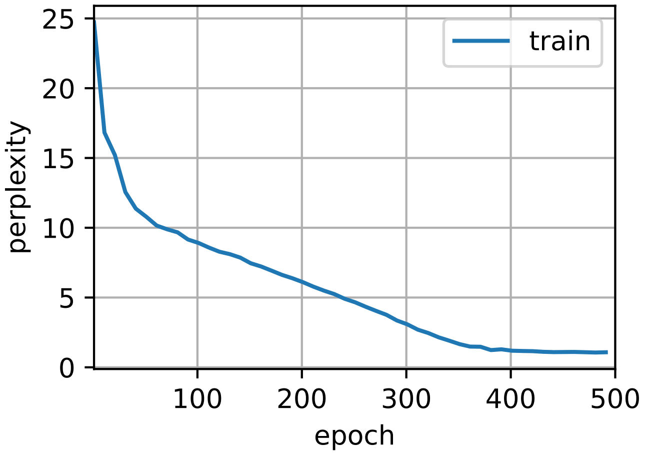


9.1.3 Concise Implementation

In Gluon, we can directly call the `GRU` class in the `rnn` module. This encapsulates all the configuration detail that we made explicit above. The code is significantly faster as it uses compiled operators rather than Python for many details that we spelled out in detail before.

```
gru_layer = rnn.GRU(num_hiddens)
model = d2l.RNNModel(gru_layer, len(vocab))
d2l.train_ch8(model, train_iter, vocab, lr, num_epochs, ctx)
```

```
Perplexity 1.1, 198954 tokens/sec on gpu(0)
time traveller  it s against reason said filby  what reason said
traveller smiled round at us then still smiling faintly and
```



Summary

- Gated recurrent neural networks are better at capturing dependencies for time series with large timestep distances.
- Reset gates help capture short-term dependencies in time series.
- Update gates help capture long-term dependencies in time series.
- GRUs contain basic RNNs as their extreme case whenever the reset gate is switched on. They can ignore sequences as needed.

Exercises

1. Compare runtime, perplexity, and the output strings for `rnn.RNN` and `rnn.GRU` implementations with each other.
2. Assume that we only want to use the input for timestep t' to predict the output at timestep $t > t'$. What are the best values for the reset and update gates for each timestep?
3. Adjust the hyperparameters and observe and analyze the impact on running time, perplexity, and the written lyrics.
4. What happens if you implement only parts of a GRU? That is, implement a recurrent cell that only has a reset gate. Likewise, implement a recurrent cell only with an update gate.

9.2 Long Short Term Memory (LSTM)

The challenge to address long-term information preservation and short-term input skipping in latent variable models has existed for a long time. One of the earliest approaches to address this was the LSTM (Hochreiter & Schmidhuber, 1997). It shares many of the properties of the Gated Recurrent Unit (GRU). Interestingly, LSTM's design is slightly more complex than GRU but predates GRU by almost two decades.

Arguably it is inspired by logic gates of a computer. To control a memory cell we need a number of gates. One gate is needed to read out the entries from the cell (as opposed to reading any other cell). We will refer to this as the *output* gate. A second gate is needed to decide when to read data into the cell. We refer to this as the *input* gate. Last, we need a mechanism to reset the contents of the cell, governed by a *forget* gate. The motivation for such a design is the same as before, namely to be able to decide when to remember and when to ignore inputs in the latent state via a dedicated mechanism. Let's see how this works in practice.

9.2.1 Gated Memory Cells

Three gates are introduced in LSTMs: the input gate, the forget gate, and the output gate. In addition to that we will introduce the memory cell that has the same shape as the hidden state. Strictly speaking this is just a fancy version of a hidden state, engineered to record additional information.

Input Gates, Forget Gates, and Output Gates

Just like with GRUs, the data feeding into the LSTM gates is the input at the current timestep $\mathbf{X}_t$ and the hidden state of the previous timestep $\mathbf{H}_{t-1}$. These inputs are processed by a fully connected layer and a sigmoid activation function to compute the values of input, forget and output gates. As a result, the three gates all output values in the range of $[0, 1]$. Fig. 9.2.1 illustrates the data flow for the input, forget, and output gates.

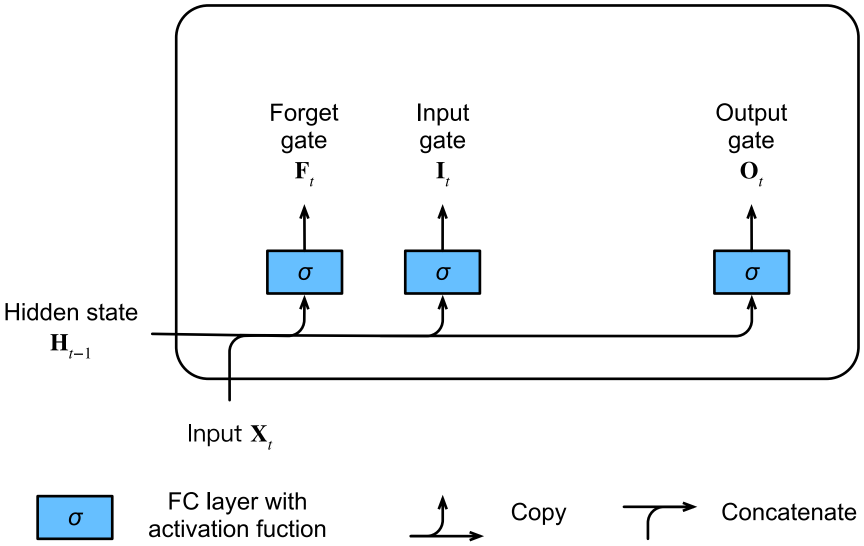


Fig. 9.2.1: Calculation of input, forget, and output gates in an LSTM.

We assume that there are h hidden units, the minibatch is of size n, and number of inputs is d. Thus, the input is $\mathbf{X}_t \in \mathbb{R}^{n\times d}$ and the hidden state of the last timestep is $\mathbf{H}_{t-1} \in \mathbb{R}^{n\times h}$. Correspondingly, the gates are defined as follows: the input gate is $\mathbf{I}_t \in \mathbb{R}^{n\times h}$, the forget gate is $\mathbf{F}_t \in \mathbb{R}^{n\times h}$, and the output gate is $\mathbf{O}_t \in \mathbb{R}^{n\times h}$. They are calculated as follows:

$$\begin{aligned}
\mathbf{I}_t &= \sigma(\mathbf{X}_t\mathbf{W}_{xi} + \mathbf{H}_{t-1}\mathbf{W}_{hi} + \mathbf{b}_i),\\
\mathbf{F}_t &= \sigma(\mathbf{X}_t\mathbf{W}_{xf} + \mathbf{H}_{t-1}\mathbf{W}_{hf} + \mathbf{b}_f),\\
\mathbf{O}_t &= \sigma(\mathbf{X}_t\mathbf{W}_{xo} + \mathbf{H}_{t-1}\mathbf{W}_{ho} + \mathbf{b}_o),
\end{aligned} \tag{9.2.1}$$

where $\mathbf{W}_{xi}, \mathbf{W}_{xf}, \mathbf{W}_{xo} \in \mathbb{R}^{d\times h}$ and $\mathbf{W}_{hi}, \mathbf{W}_{hf}, \mathbf{W}_{ho} \in \mathbb{R}^{h\times h}$ are weight parameters and $\mathbf{b}_i, \mathbf{b}_f, \mathbf{b}_o \in \mathbb{R}^{1\times h}$ are bias parameters.

Candidate Memory Cell

Next we design the memory cell. Since we have not specified the action of the various gates yet, we first introduce the *candidate* memory cell $\tilde{\mathbf{C}}_t \in \mathbb{R}^{n\times h}$. Its computation is similar to the three gates described above, but using a tanh function with a value range for $[-1, 1]$ as the activation function. This leads to the following equation at timestep t.

$$\tilde{\mathbf{C}}_t = \tanh(\mathbf{X}_t\mathbf{W}_{xc} + \mathbf{H}_{t-1}\mathbf{W}_{hc} + \mathbf{b}_c). \tag{9.2.2}$$

Here $\mathbf{W}_{xc} \in \mathbb{R}^{d\times h}$ and $\mathbf{W}_{hc} \in \mathbb{R}^{h\times h}$ are weight parameters and $\mathbf{b}_c \in \mathbb{R}^{1\times h}$ is a bias parameter.

A quick illustration of the candidate memory cell is shown in Fig. 9.2.2.

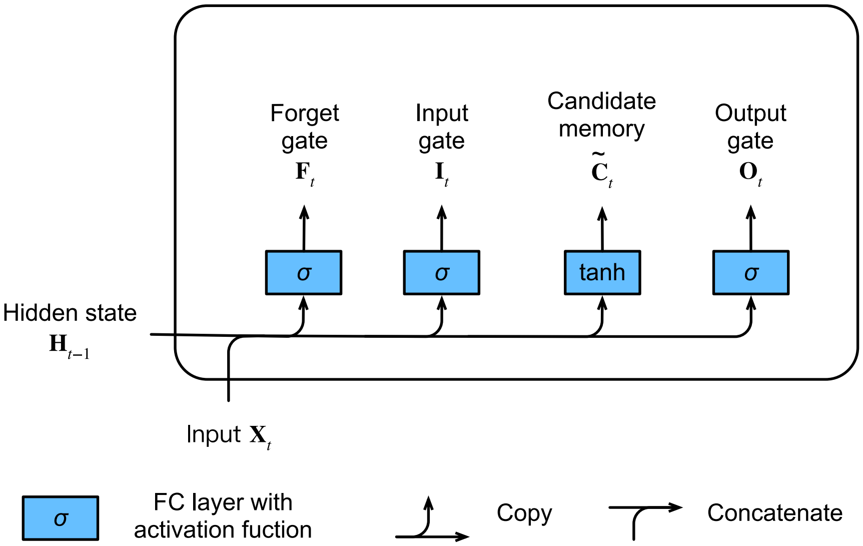

Forget gate $\mathbf{F}_t$
Input gate $\mathbf{I}_t$
Candidate memory $\tilde{\mathbf{C}}_t$
Output gate $\mathbf{O}_t$
σ
σ
tanh
σ
Hidden state $\mathbf{H}_{t-1}$
Input $\mathbf{X}_t$
σ
FC layer with activation fuction
Copy
Concatenate

Fig. 9.2.2: Computation of candidate memory cells in LSTM.

Memory Cell

In GRUs, we had a single mechanism to govern input and forgetting. Here in LSTMs we have two parameters, $\mathbf{I}_t$ which governs how much we take new data into account via $\tilde{\mathbf{C}}_t$ and the forget parameter $\mathbf{F}_t$ which addresses how much of the old memory cell content $\mathbf{C}_{t-1} \in \mathbb{R}^{n\times h}$ we retain. Using the same pointwise multiplication trick as before, we arrive at the following update equation.

$$\mathbf{C}_t = \mathbf{F}_t \odot \mathbf{C}_{t-1} + \mathbf{I}_t \odot \tilde{\mathbf{C}}_t. \tag{9.2.3}$$

If the forget gate is always approximately 1 and the input gate is always approximately 0, the past memory cells $\mathbf{C}_{t-1}$ will be saved over time and passed to the current timestep. This design was introduced to alleviate the vanishing gradient problem and to better capture dependencies for time series with long range dependencies. We thus arrive at the flow diagram in Fig. 9.2.3.

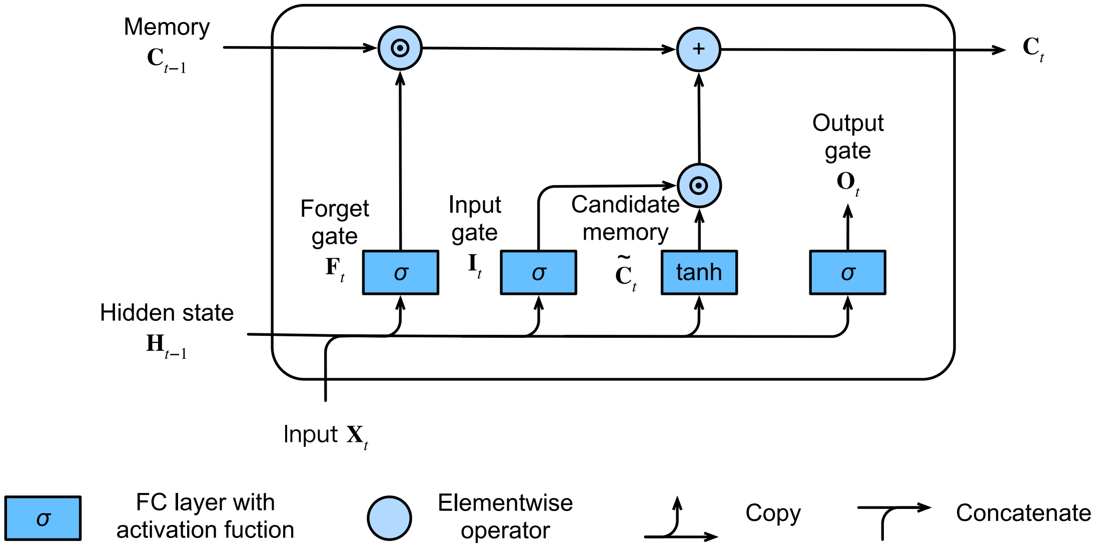


Fig. 9.2.3: Computation of memory cells in an LSTM. Here, the multiplication is carried out elementwise.

Hidden States

Last, we need to define how to compute the hidden state $\mathbf{H}_t \in \mathbb{R}^{n \times h}$. This is where the output gate comes into play. In LSTM it is simply a gated version of the tanh of the memory cell. This ensures that the values of $\mathbf{H}_t$ are always in the interval $(-1, 1)$. Whenever the output gate is 1 we effectively pass all memory information through to the predictor, whereas for output 0 we retain all the information only within the memory cell and perform no further processing. Fig. 9.2.4 has a graphical illustration of the data flow.

$$\mathbf{H}_t = \mathbf{O}_t \odot \tanh(\mathbf{C}_t). \tag{9.2.4}$$

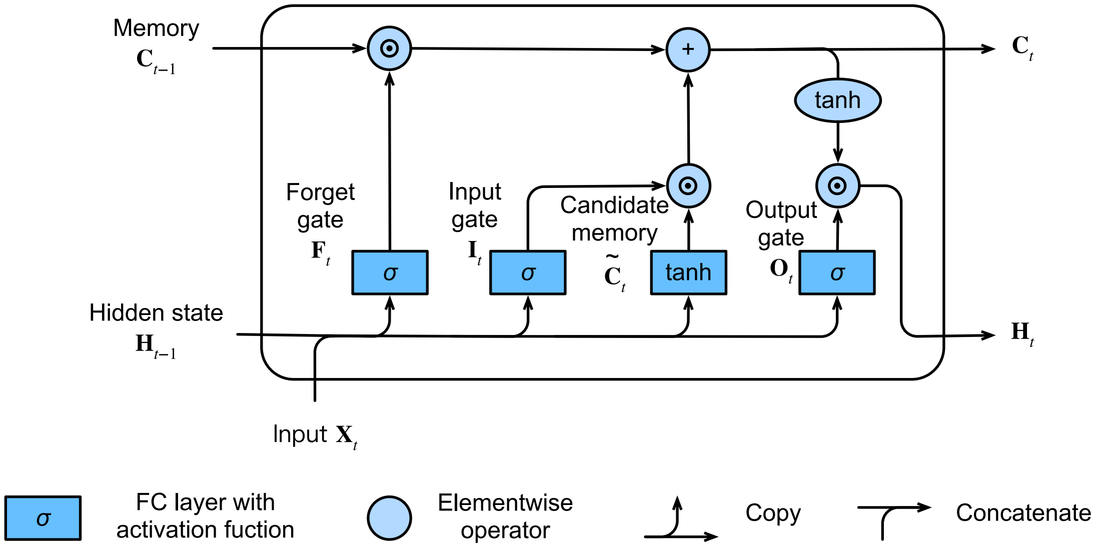


Fig. 9.2.4: Computation of the hidden state. Multiplication is elementwise.

9.2.2 Implementation from Scratch

Now let's implement an LSTM from scratch. As same as the experiments in the previous sections, we first load data of *The Time Machine*.

```
import d2l
from mxnet import np, npx
from mxnet.gluon import rnn
npx.set_np()

batch_size, num_steps = 32, 35
train_iter, vocab = d2l.load_data_time_machine(batch_size, num_steps)
```

Initializing Model Parameters

Next we need to define and initialize the model parameters. As previously, the hyperparameter num_hiddens defines the number of hidden units. We initialize weights following a Gaussian distribution with 0.01 standard deviation, and we set the biases to 0.

```
def get_lstm_params(vocab_size, num_hiddens, ctx):
    num_inputs = num_outputs = vocab_size

    def normal(shape):
        return np.random.normal(scale=0.01, size=shape, ctx=ctx)

    def three():
        return (normal((num_inputs, num_hiddens)),
                normal((num_hiddens, num_hiddens)),
                np.zeros(num_hiddens, ctx=ctx))

    W_xi, W_hi, b_i = three()  # Input gate parameters
    W_xf, W_hf, b_f = three()  # Forget gate parameters
    W_xo, W_ho, b_o = three()  # Output gate parameters
    W_xc, W_hc, b_c = three()  # Candidate cell parameters
    # Output layer parameters
    W_hq = normal((num_hiddens, num_outputs))
    b_q = np.zeros(num_outputs, ctx=ctx)
    # Attach gradients
    params = [W_xi, W_hi, b_i, W_xf, W_hf, b_f, W_xo, W_ho, b_o, W_xc, W_hc,
              b_c, W_hq, b_q]
    for param in params:
        param.attach_grad()
    return params
```

Defining the Model

In the initialization function, the hidden state of the LSTM needs to return an additional memory cell with a value of 0 and a shape of (batch size, number of hidden units). Hence we get the following state initialization.

```
def init_lstm_state(batch_size, num_hiddens, ctx):
    return (np.zeros(shape=(batch_size, num_hiddens), ctx=ctx),
            np.zeros(shape=(batch_size, num_hiddens), ctx=ctx))
```

The actual model is defined just like what we discussed before: providing three gates and an auxiliary memory cell. Note that only the hidden state is passed to the output layer. The memory cells $\mathbf{C}_t$ do not participate in the output computation directly.

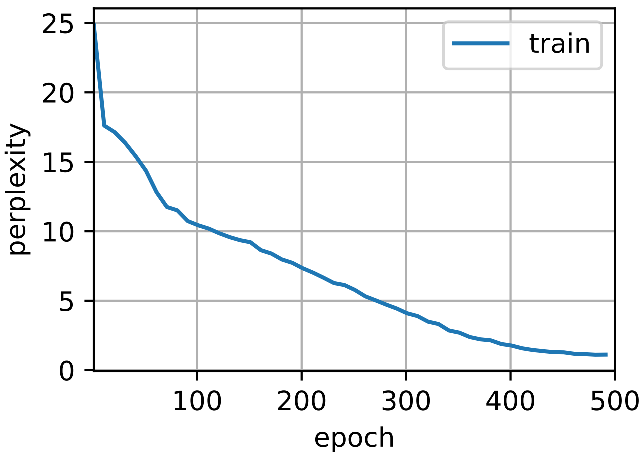
```
def lstm(inputs, state, params):
    [W_xi, W_hi, b_i, W_xf, W_hf, b_f, W_xo, W_ho, b_o, W_xc, W_hc, b_c,
     W_hq, b_q] = params
    (H, C) = state
    outputs = []
    for X in inputs:
        I = npx.sigmoid(np.dot(X, W_xi) + np.dot(H, W_hi) + b_i)
        F = npx.sigmoid(np.dot(X, W_xf) + np.dot(H, W_hf) + b_f)
        O = npx.sigmoid(np.dot(X, W_xo) + np.dot(H, W_ho) + b_o)
        C_tilda = np.tanh(np.dot(X, W_xc) + np.dot(H, W_hc) + b_c)
        C = F * C + I * C_tilda
        H = O * np.tanh(C)
        Y = np.dot(H, W_hq) + b_q
        outputs.append(Y)
    return np.concatenate(outputs, axis=0), (H, C)
```

Training and Prediction

Let's train an LSTM as same as what we did in Section 9.1, by calling the `RNNModelScratch` function as introduced in Section 8.5.

```
vocab_size, num_hiddens, ctx = len(vocab), 256, d2l.try_gpu()
num_epochs, lr = 500, 1
model = d2l.RNNModelScratch(len(vocab), num_hiddens, ctx, get_lstm_params,
                            init_lstm_state, lstm)
d2l.train_ch8(model, train_iter, vocab, lr, num_epochs, ctx)
```

```
Perplexity 1.1, 12735 tokens/sec on gpu(0)
time traveller smiled round at us then still smiling faintly and
traveller and which as regh arefichion simee it raspons the
```

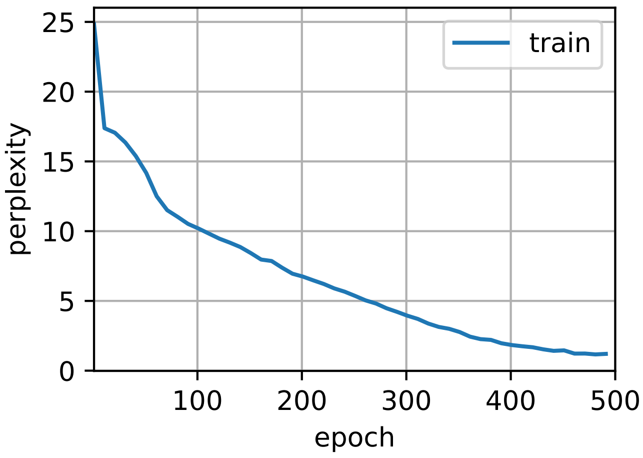


9.2.3 Concise Implementation

In Gluon, we can directly call the `LSTM` class in the `rnn` module. This encapsulates all the configuration details that we made explicit above. The code is significantly faster as it uses compiled operators rather than Python for many details that we spelled out in detail before.

```
lstm_layer = rnn.LSTM(num_hiddens)
model = d2l.RNNModel(lstm_layer, len(vocab))
d2l.train_ch8(model, train_iter, vocab, lr, num_epochs, ctx)
```

```
Perplexity 1.2, 196700 tokens/sec on gpu(0)
time traveller smiled ror time bacnimestler you say four dimensi
traveller fom s accour existence they time drivelyor pe can
```



In many cases, LSTMs perform slightly better than GRUs but they are more costly to train and execute due to the larger latent state size. LSTMs are the prototypical latent variable autoregressive model with nontrivial state control. Many variants thereof have been proposed over the years, e.g., multiple layers, residual connections, different types of regularization. However, training LSTMs and other sequence models (such as GRU) are quite costly due to the long range dependency of the sequence. Later we will encounter alternative models such as Transformers that can be used in some cases.

Summary

- LSTMs have three types of gates: input gates, forget gates, and output gates which control the flow of information.
- The hidden layer output of LSTM includes hidden states and memory cells. Only hidden states are passed into the output layer. Memory cells are entirely internal.
- LSTMs can cope with vanishing and exploding gradients.

Exercises

1. Adjust the hyperparameters. Observe and analyze the impact on runtime, perplexity, and the generated output.
2. How would you need to change the model to generate proper words as opposed to sequences of characters?
3. Compare the computational cost for GRUs, LSTMs, and regular RNNs for a given hidden dimension. Pay special attention to the training and inference cost.
4. Since the candidate memory cells ensure that the value range is between -1 and 1 by using the tanh function, why does the hidden state need to use the tanh function again to ensure that the output value range is between -1 and 1?
5. Implement an LSTM for time series prediction rather than character sequence prediction.

9.3 Deep Recurrent Neural Networks

Up to now, we only discussed recurrent neural networks with a single unidirectional hidden layer. In it the specific functional form of how latent variables and observations interact was rather arbitrary. This is not a big problem as long as we have enough flexibility to model different types of interactions. With a single layer, however, this can be quite challenging. In the case of the perceptron, we fixed this problem by adding more layers. Within RNNs this is a bit more tricky, since we first need to decide how and where to add extra nonlinearity. Our discussion below focuses primarily on LSTMs, but it applies to other sequence models, too.

- We could add extra nonlinearity to the gating mechanisms. That is, instead of using a single perceptron we could use multiple layers. This leaves the *mechanism* of the LSTM unchanged. Instead it makes it more sophisticated. This would make sense if we were led to believe that the LSTM mechanism describes some form of universal truth of how latent variable autoregressive models work.
- We could stack multiple layers of LSTMs on top of each other. This results in a mechanism that is more flexible, due to the combination of several simple layers. In particular, data might be relevant at different levels of the stack. For instance, we might want to keep high-level data about financial market conditions (bear or bull market) available, whereas at a lower level we only record shorter-term temporal dynamics.

Beyond all this abstract discussion it is probably easiest to understand the family of models we are interested in by reviewing Fig. 9.3.1. It describes a deep recurrent neural network with L hidden layers. Each hidden state is continuously passed to both the next timestep of the current layer and the current timestep of the next layer.

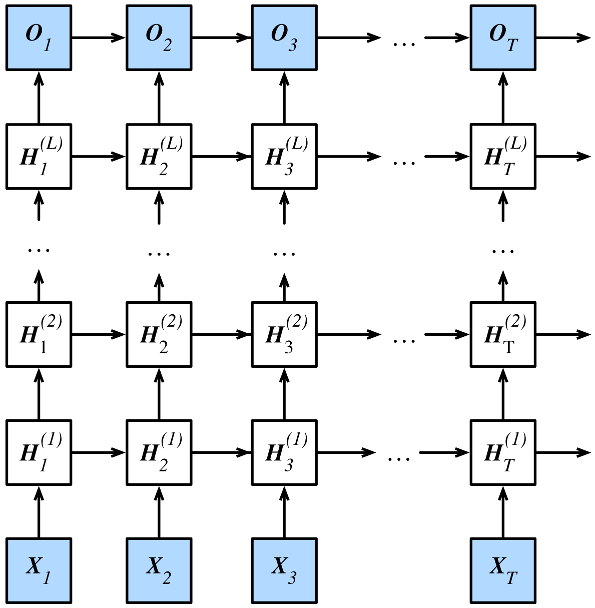


Fig. 9.3.1: Architecture of a deep recurrent neural network.

9.3.1 Functional Dependencies

At timestep t we assume that we have a minibatch $\mathbf{X}_t \in \mathbb{R}^{n\times d}$ (number of examples: n, number of inputs: d). The hidden state of hidden layer ℓ ($\ell = 1, \ldots, T$) is $\mathbf{H}_t^{(\ell)} \in \mathbb{R}^{n\times h}$ (number of hidden units: h), the output layer variable is $\mathbf{O}_t \in \mathbb{R}^{n\times q}$ (number of outputs: q) and a hidden layer activation function f_l for layer l. We compute the hidden state of layer 1 as before, using $\mathbf{X}_t$ as input. For all subsequent layers, the hidden state of the previous layer is used in its place.

$$\begin{aligned} \mathbf{H}_t^{(1)} & = f_1\left(\mathbf{X}_t, \mathbf{H}_{t-1}^{(1)}\right), \\ \mathbf{H}_t^{(l)} & = f_l\left(\mathbf{H}_t^{(l-1)}, \mathbf{H}_{t-1}^{(l)}\right). \end{aligned} \tag{9.3.1}$$

Finally, the output layer is only based on the hidden state of hidden layer L. We use the output function g to address this:

$$\mathbf{O}_t = g\left(\mathbf{H}_t^{(L)}\right). \tag{9.3.2}$$

Just as with multilayer perceptrons, the number of hidden layers L and number of hidden units h are hyper parameters. In particular, we can pick a regular RNN, a GRU, or an LSTM to implement the model.

9.3.2 Concise Implementation

Fortunately many of the logistical details required to implement multiple layers of an RNN are readily available in Gluon. To keep things simple we only illustrate the implementation using such built-in functionality. The code is very similar to the one we used previously for LSTMs. In fact, the only difference is that we specify the number of layers explicitly rather than picking the default of a single layer. Let's begin by importing the appropriate modules and loading data.

```
import d2l
from mxnet import npx
from mxnet.gluon import rnn
npx.set_np()

batch_size, num_steps = 32, 35
train_iter, vocab = d2l.load_data_time_machine(batch_size, num_steps)
```

The architectural decisions (such as choosing parameters) are very similar to those of previous sections. We pick the same number of inputs and outputs as we have distinct tokens, i.e., vocab_size. The number of hidden units is still 256. The only difference is that we now select a nontrivial number of layers `num_layers = 2`.

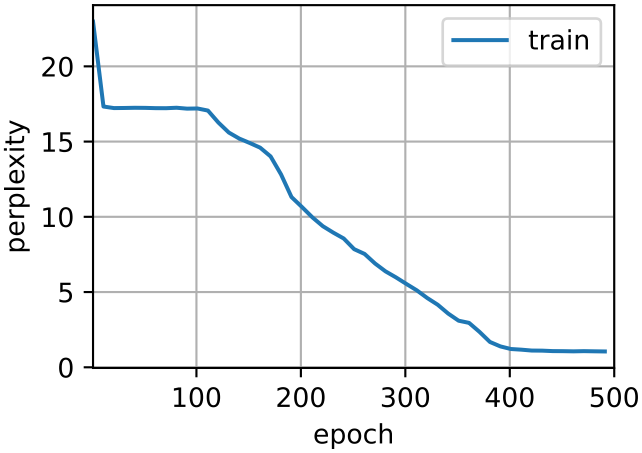
```
vocab_size, num_hiddens, num_layers, ctx = len(vocab), 256, 2, d2l.try_gpu()
lstm_layer = rnn.LSTM(num_hiddens, num_layers)
model = d2l.RNNModel(lstm_layer, len(vocab))
```

9.3.3 Training

The actual invocation logic is identical to before. The only difference is that we now instantiate two layers with LSTMs. This rather more complex architecture and the large number of epochs slow down training considerably.

```
num_epochs, lr = 500, 2
d2l.train_ch8(model, train_iter, vocab, lr, num_epochs, ctx)
```

```
Perplexity 1.0, 145196 tokens/sec on gpu(0)
time traveller for so it will be convenient to speak of him was
traveller smiled round at us then still smiling faintly and
```



Summary

- In deep recurrent neural networks, hidden state information is passed to the next timestep of the current layer and the current timestep of the next layer.
- There exist many different flavors of deep RNNs, such as LSTMs, GRUs, or regular RNNs. Conveniently these models are all available as parts of the `rnn` module in Gluon.
- Initialization of the models requires care. Overall, deep RNNs require considerable amount of work (such as learning rate and clipping) to ensure proper convergence.

Exercises

1. Try to implement a two-layer RNN from scratch using the single layer implementation we discussed in Section 8.5.
2. Replace the LSTM by a GRU and compare the accuracy.
3. Increase the training data to include multiple books. How low can you go on the perplexity scale?
4. Would you want to combine sources of different authors when modeling text? Why is this a good idea? What could go wrong?

9.4 Bidirectional Recurrent Neural Networks

So far we assumed that our goal is to model the next word given what we have seen so far, e.g., in the context of a time series or in the context of a language model. While this is a typical scenario, it is not the only one we might encounter. To illustrate the issue, consider the following three tasks of filling in the blanks in a text:

1. `I am _____`
2. `I am _____ very hungry.`
3. `I am _____ very hungry, I could eat half a pig.`

Depending on the amount of information available, we might fill the blanks with very different words such as "happy", "not", and "very". Clearly the end of the phrase (if available) conveys significant information about which word to pick. A sequence model that is incapable of taking advantage of this will perform poorly on related tasks. For instance, to do well in named entity recognition (e.g., to recognize whether "Green" refers to "Mr. Green" or to the color) longer-range context is equally vital. To get some inspiration for addressing the problem let's take a detour to graphical models.

9.4.1 Dynamic Programming

This section serves to illustrate the dynamic programming problem. The specific technical details do not matter for understanding the deep learning counterpart but they help in motivating why one might use deep learning and why one might pick specific architectures.

If we want to solve the problem using graphical models we could for instance design a latent variable model as follows. We assume that there exists some latent variable h_t which governs the emissions x_t that we observe via $p(x_t \mid h_t)$. Moreover, the transitions $h_t \to h_{t+1}$ are given by some state transition probability $p(h_t + 1 \mid h_t)$. The graphical model then looks as Fig. 9.4.1.

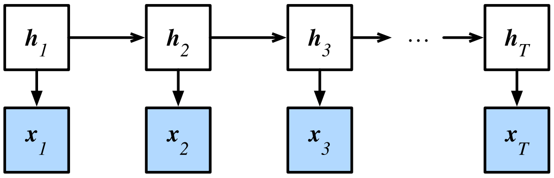


Fig. 9.4.1: Hidden Markov Model.

Thus, for a sequence of T observations we have the following joint probability distribution over observed and hidden states:

$$p(x, h) = p(h_1)p(x_1 \mid h_1) \prod_{t=2}^{T} p(h_t \mid h_{t-1})p(x_t \mid h_t). \tag{9.4.1}$$

Now assume that we observe all x_i with the exception of some x_j and it is our goal to compute $p(x_j \mid x^{-j})$, where $x^{-j} = (x_1, x_2, \ldots, x_{j-1})$. To accomplish this we need to sum over all possible choices of $h = (h_1, \ldots, h_T)$. In case h_i can take on k distinct values, this means that we need to sum over k^T terms—mission impossible! Fortunately there is an elegant solution for this: dynamic programming. To see how it works, consider summing over the first two hidden variable h_1 and

h_2. This yields:

$$\begin{aligned}
p(x) &= \sum_{h_1,\ldots,h_T} p(x_1,\ldots,x_T;h_1,\ldots,h_T) \\
&= \sum_{h_1,\ldots,h_T} p(h_1)p(x_1 \mid h_1)\prod_{t=2}^{T} p(h_t \mid h_{t-1})p(x_t \mid h_t) \\
&= \sum_{h_2,\ldots,h_T} \underbrace{\left[\sum_{h_1} p(h_1)p(x_1 \mid h_1)p(h_2 \mid h_1)\right]}_{=:\pi_2(h_2)} p(x_2 \mid h_2)\prod_{t=3}^{T} p(h_t \mid h_{t-1})p(x_t \mid h_t) \\
&= \sum_{h_3,\ldots,h_T} \underbrace{\left[\sum_{h_2} \pi_2(h_2)p(x_2 \mid h_2)p(h_3 \mid h_2)\right]}_{=:\pi_3(h_3)} p(x_3 \mid h_3)\prod_{t=4}^{T} p(h_t \mid h_{t-1})p(x_t \mid h_t) \\
&= \ldots \\
&= \sum_{h_T} \pi_T(h_T)p(x_T \mid h_T).
\end{aligned} \tag{9.4.2}$$

In general we have the *forward recursion* as

$$\pi_{t+1}(h_{t+1}) = \sum_{h_t} \pi_t(h_t)p(x_t \mid h_t)p(h_{t+1} \mid h_t). \tag{9.4.3}$$

The recursion is initialized as $\pi_1(h_1) = p(h_1)$. In abstract terms this can be written as $\pi_{t+1} = f(\pi_t, x_t)$, where f is some learnable function. This looks very much like the update equation in the hidden variable models we discussed so far in the context of RNNs. Entirely analogously to the forward recursion, we can also start a backward recursion. This yields:

$$\begin{aligned}
p(x) &= \sum_{h_1,\ldots,h_T} p(x_1,\ldots,x_T;h_1,\ldots,h_T) \\
&= \sum_{h_1,\ldots,h_T} \prod_{t=1}^{T-1} p(h_t \mid h_{t-1})p(x_t \mid h_t) \cdot p(h_T \mid h_{T-1})p(x_T \mid h_T) \\
&= \sum_{h_1,\ldots,h_{T-1}} \prod_{t=1}^{T-1} p(h_t \mid h_{t-1})p(x_t \mid h_t) \cdot \underbrace{\left[\sum_{h_T} p(h_T \mid h_{T-1})p(x_T \mid h_T)\right]}_{=:\rho_{T-1}(h_{T-1})} \\
&= \sum_{h_1,\ldots,h_{T-2}} \prod_{t=1}^{T-2} p(h_t \mid h_{t-1})p(x_t \mid h_t) \cdot \underbrace{\left[\sum_{h_{T-1}} p(h_{T-1} \mid h_{T-2})p(x_{T-1} \mid h_{T-1})\rho_{T-1}(h_{T-1})\right]}_{=:\rho_{T-2}(h_{T-2})} \\
&= \ldots \\
&= \sum_{h_1} p(h_1)p(x_1 \mid h_1)\rho_1(h_1).
\end{aligned} \tag{9.4.4}$$

We can thus write the *backward recursion* as

$$\rho_{t-1}(h_{t-1}) = \sum_{h_t} p(h_t \mid h_{t-1})p(x_t \mid h_t)\rho_t(h_t), \tag{9.4.5}$$

with initialization $\rho_T(h_T) = 1$. These two recursions allow us to sum over T variables in $\mathcal{O}(kT)$ (linear) time over all values of $(h_1, \ldots, h_T)$ rather than in exponential time. This is one of the great benefits of the probabilistic inference with graphical models. It is a very special instance of the (Aji & McEliece, 2000) proposed in 2000 by Aji and McEliece. Combining both forward and backward pass, we are able to compute

$$p(x_j \mid x_{-j}) \propto \sum_{h_j} \pi_j(h_j)\rho_j(h_j)p(x_j \mid h_j). \tag{9.4.6}$$

Note that in abstract terms the backward recursion can be written as $\rho_{t-1} = g(\rho_t, x_t)$, where g is a learnable function. Again, this looks very much like an update equation, just running backwards unlike what we have seen so far in RNNs. Indeed, HMMs benefit from knowing future data when it is available. Signal processing scientists distinguish between the two cases of knowing and not knowing future observations as interpolation v.s. extrapolation. See the introductory chapter of the book by (Doucet et al., 2001) on sequential Monte Carlo algorithms for more details.

9.4.2 Bidirectional Model

If we want to have a mechanism in RNNs that offers comparable look-ahead ability as in HMMs, we need to modify the recurrent net design that we have seen so far. Fortunately, this is easy conceptually. Instead of running an RNN only in the forward mode starting from the first symbol, we start another one from the last symbol running from back to front. *Bidirectional recurrent neural networks* add a hidden layer that passes information in a backward direction to more flexibly process such information. Fig. 9.4.2 illustrates the architecture of a bidirectional recurrent neural network with a single hidden layer.

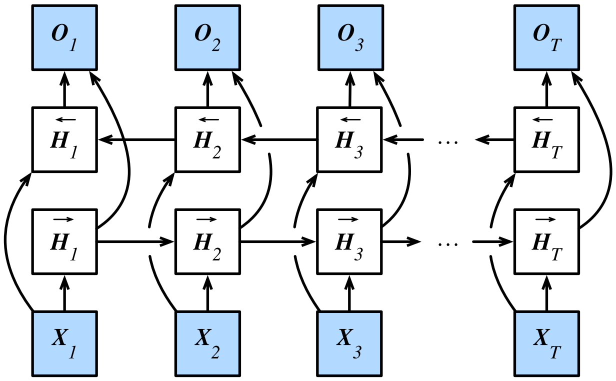


Fig. 9.4.2: Architecture of a bidirectional recurrent neural network.

In fact, this is not too dissimilar to the forward and backward recursion we encountered above. The main distinction is that in the previous case these equations had a specific statistical meaning. Now they are devoid of such easily accessible interpretation and we can just treat them as generic functions. This transition epitomizes many of the principles guiding the design of modern deep networks: first, use the type of functional dependencies of classical statistical models, and then use the models in a generic form.

Definition

Bidirectional RNNs were introduced by (Schuster & Paliwal, 1997). For a detailed discussion of the various architectures see also the paper by (Graves & Schmidhuber, 2005). Let's look at the specifics of such a network.

For a given timestep t, the minibatch input is $\mathbf{X}_t \in \mathbb{R}^{n\times d}$ (number of examples: n, number of inputs: d) and the hidden layer activation function is ϕ. In the bidirectional architecture, we assume that the forward and backward hidden states for this timestep are $\overrightarrow{\mathbf{H}}_t \in \mathbb{R}^{n\times h}$ and $\overleftarrow{\mathbf{H}}_t \in \mathbb{R}^{n\times h}$ respectively. Here h indicates the number of hidden units. We compute the forward and backward hidden state updates as follows:

$$\begin{aligned}\overrightarrow{\mathbf{H}}_t &= \phi(\mathbf{X}_t \mathbf{W}_{xh}^{(f)} + \overrightarrow{\mathbf{H}}_{t-1} \mathbf{W}_{hh}^{(f)} + \mathbf{b}_h^{(f)}),\\ \overleftarrow{\mathbf{H}}_t &= \phi(\mathbf{X}_t \mathbf{W}_{xh}^{(b)} + \overleftarrow{\mathbf{H}}_{t+1} \mathbf{W}_{hh}^{(b)} + \mathbf{b}_h^{(b)}).\end{aligned} \tag{9.4.7}$$

Here, the weight parameters $\mathbf{W}_{xh}^{(f)} \in \mathbb{R}^{d\times h}, \mathbf{W}_{hh}^{(f)} \in \mathbb{R}^{h\times h}, \mathbf{W}_{xh}^{(b)} \in \mathbb{R}^{d\times h}$, and $\mathbf{W}_{hh}^{(b)} \in \mathbb{R}^{h\times h}$, and bias parameters $\mathbf{b}_h^{(f)} \in \mathbb{R}^{1\times h}$ and $\mathbf{b}_h^{(b)} \in \mathbb{R}^{1\times h}$ are all model parameters.

Then we concatenate the forward and backward hidden states $\overrightarrow{\mathbf{H}}_t$ and $\overleftarrow{\mathbf{H}}_t$ to obtain the hidden state $\mathbf{H}_t \in \mathbb{R}^{n\times 2h}$ and feed it to the output layer. In deep bidirectional RNNs, the information is passed on as *input* to the next bidirectional layer. Last, the output layer computes the output $\mathbf{O}_t \in \mathbb{R}^{n\times q}$ (number of outputs: q):

$$\mathbf{O}_t = \mathbf{H}_t \mathbf{W}_{hq} + \mathbf{b}_q. \tag{9.4.8}$$

Here, the weight parameter $\mathbf{W}_{hq} \in \mathbb{R}^{2h\times q}$ and the bias parameter $\mathbf{b}_q \in \mathbb{R}^{1\times q}$ are the model parameters of the output layer. The two directions can have different numbers of hidden units.

Computational Cost and Applications

One of the key features of a bidirectional RNN is that information from both ends of the sequence is used to estimate the output. That is, we use information from both future and past observations to predict the current one (a smoothing scenario). In the case of language models this is not quite what we want. After all, we do not have the luxury of knowing the next to next symbol when predicting the next one. Hence, if we were to use a bidirectional RNN naively we would not get a very good accuracy: during training we have past and future data to estimate the present. During test time we only have past data and thus poor accuracy (we will illustrate this in an experiment below).

To add insult to injury bidirectional RNNs are also exceedingly slow. The main reason for this is that they require both a forward and a backward pass and that the backward pass is dependent on the outcomes of the forward pass. Hence, gradients will have a very long dependency chain.

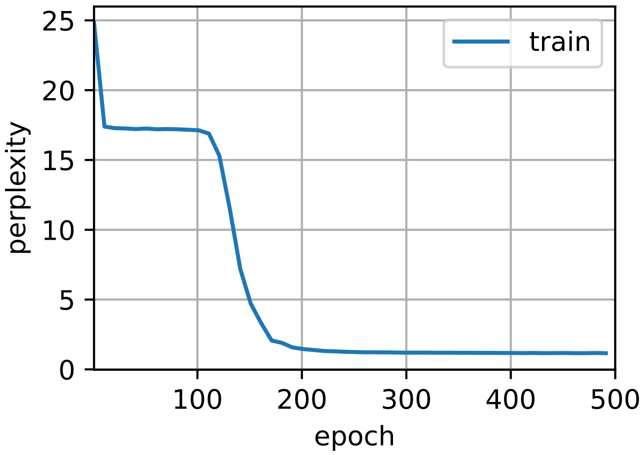
In practice bidirectional layers are used very sparingly and only for a narrow set of applications, such as filling in missing words, annotating tokens (e.g., for named entity recognition), or encoding sequences wholesale as a step in a sequence processing pipeline (e.g., for machine translation). In short, handle with care!

Training a Bidirectional RNN for the Wrong Application

If we were to ignore all advice regarding the fact that bidirectional LSTMs use past and future data and simply apply it to language models, we will get estimates with acceptable perplexity. Nonetheless, the ability of the model to predict future symbols is severely compromised as the example below illustrates. Despite reasonable perplexity, it only generates gibberish even after many iterations. We include the code below as a cautionary example against using them in the wrong context.

```
import d2l
from mxnet import npx
from mxnet.gluon import rnn
npx.set_np()

# Load data
batch_size, num_steps = 32, 35
train_iter, vocab = d2l.load_data_time_machine(batch_size, num_steps)
# Define the model
vocab_size, num_hiddens, num_layers, ctx = len(vocab), 256, 2, d2l.try_gpu()
lstm_layer = rnn.LSTM(num_hiddens, num_layers, bidirectional=True)
model = d2l.RNNModel(lstm_layer, len(vocab))
# Train the model
num_epochs, lr = 500, 1
d2l.train_ch8(model, train_iter, vocab, lr, num_epochs, ctx)
```

```
Perplexity 1.2, 84692 tokens/sec on gpu(0)
time travellererererererererererererererererererererererererer
travellererererererererererererererererererererererererer
```



The output is clearly unsatisfactory for the reasons described above. For a discussion of more effective uses of bidirectional models, please see the sentiment classification in Section 14.9.

Summary

- In bidirectional recurrent neural networks, the hidden state for each timestep is simultaneously determined by the data prior to and after the current timestep.
- Bidirectional RNNs bear a striking resemblance with the forward-backward algorithm in graphical models.
- Bidirectional RNNs are mostly useful for sequence embedding and the estimation of observations given bidirectional context.
- Bidirectional RNNs are very costly to train due to long gradient chains.

Exercises

1. If the different directions use a different number of hidden units, how will the shape of $\mathbf{H}_t$ change?
2. Design a bidirectional recurrent neural network with multiple hidden layers.
3. Implement a sequence classification algorithm using bidirectional RNNs. Hint: use the RNN to embed each word and then aggregate (average) all embedded outputs before sending the output into an MLP for classification. For instance, if we have $(\mathbf{o}_1, \mathbf{o}_2, \mathbf{o}_3)$, we compute $\bar{\mathbf{o}} = \frac{1}{3} \sum_i \mathbf{o}_i$ first and then use the latter for sentiment classification.

9.5 Machine Translation and the Dataset

So far we see how to use recurrent neural networks for language models, in which we predict the next token given all previous tokens in an article. Now let's have a look at a different application, machine translation, whose predict output is no longer a single token, but a list of tokens.

Machine translation (MT) refers to the automatic translation of a segment of text from one language to another. Solving this problem with neural networks is often called neural machine translation (NMT). Compared to language models (Section 8.3), in which the corpus only contains a single language, machine translation dataset has at least two languages, the source language and the target language. In addition, each sentence in the source language is mapped to the according translation in the target language. Therefore, the data preprocessing for machine translation data is different to the one for language models. This section is dedicated to demonstrate how to pre-process such a dataset and then load into a set of minibatches.

```python
import d2l
import zipfile

from mxnet import np, npx, gluon
npx.set_np()
```

9.5.1 Reading and Preprocessing the Dataset

We first download a dataset that contains a set of English sentences with the corresponding French translations. As can be seen that each line contains a English sentence with its French translation, which are separated by a TAB.

```
# Saved in the d2l package for later use
def read_data_nmt():
    fname = gluon.utils.download('http://data.mxnet.io/data/fra-eng.zip')
    with zipfile.ZipFile(fname, 'r') as f:
        return f.read('fra.txt').decode("utf-8")

raw_text = read_data_nmt()
print(raw_text[0:106])
```

```
Downloading fra-eng.zip from http://data.mxnet.io/data/fra-eng.zip...
Go. Va !
Hi. Salut !
Run!        Cours !
Run!        Courez !
Who?        Qui ?
Wow!        Ça alors !
Fire!       Au feu !
Help!       À l'aide !
```

We perform several preprocessing steps on the raw text data, including ignoring cases, replacing UTF-8 non-breaking space with space, and adding space between words and punctuation marks.

```
# Saved in the d2l package for later use
def preprocess_nmt(text):
    text = text.replace('\u202f', ' ').replace('\xa0', ' ')

    def no_space(char, prev_char):
        return (True if char in (',', '!', '.')
                and prev_char != ' ' else False)

    out = [' '+char if i > 0 and no_space(char, text[i-1]) else char
           for i, char in enumerate(text.lower())]
    return ''.join(out)

text = preprocess_nmt(raw_text)
print(text[0:95])
```

```
go .        va !
hi .        salut !
run !       cours !
run !       courez !
who?        qui ?
wow !       ça alors !
fire !      au feu !
```

9.5.2 Tokenization

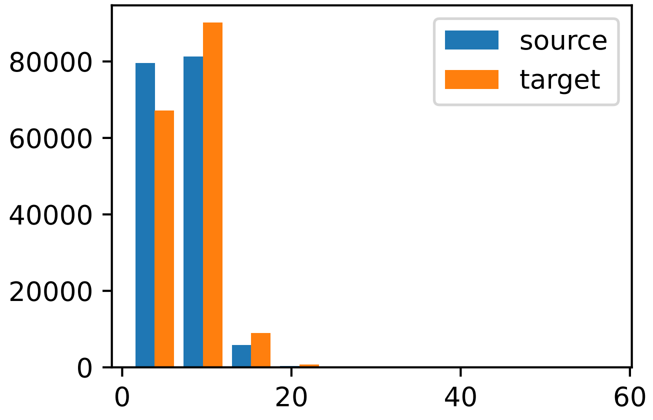
Different to using character tokens in Section 8.3, here a token is either a word or a punctuation mark. The following function tokenizes the text data to return `source` and `target`. Each one is a list of token list, with `source[i]` is the i^{th} sentence in the source language and `target[i]` is the i^{th} sentence in the target language. To make the latter training faster, we sample the first `num_examples` sentences pairs.

```
# Saved in the d2l package for later use
def tokenize_nmt(text, num_examples=None):
    source, target = [], []
    for i, line in enumerate(text.split('\n')):
        if num_examples and i > num_examples:
            break
        parts = line.split('\t')
        if len(parts) == 2:
            source.append(parts[0].split(' '))
            target.append(parts[1].split(' '))
    return source, target

source, target = tokenize_nmt(text)
source[0:3], target[0:3]
```

```
([['go', '.'], ['hi', '.'], ['run', '!']],
 [['va', '!'], ['salut', '!'], ['cours', '!']])
```

We visualize the histogram of the number of tokens per sentence the following figure. As can be seen that a sentence in average contains 5 tokens, and most of them have less than 10 tokens.

```
d2l.set_figsize((3.5, 2.5))
d2l.plt.hist([[len(l) for l in source], [len(l) for l in target]],
             label=['source', 'target'])
d2l.plt.legend(loc='upper right');
```



9.5.3 Vocabulary

Since the tokens in the source language could be different to the ones in the target language, we need to build a vocabulary for each of them. Since we are using words instead of characters as tokens, it makes the vocabulary size significantly large. Here we map every token that appears less than 3 times into the <unk> token Section 8.2. In addition, we need other special tokens such as padding and sentence beginnings.

```
src_vocab = d2l.Vocab(source, min_freq=3, use_special_tokens=True)
len(src_vocab)
```

```
9140
```

9.5.4 Loading the Dataset

In language models, each example is a `num_steps` length sequence from the corpus, which may be a segment of a sentence, or span over multiple sentences. In machine translation, an example should contain a pair of source sentence and target sentence. These sentences might have different lengths, while we need same length examples to form a minibatch.

One way to solve this problem is that if a sentence is longer than `num_steps`, we trim it is length, otherwise pad with a special <pad> token to meet the length. Therefore we could transform any sentence to a fixed length.

```
# Saved in the d2l package for later use
def trim_pad(line, num_steps, padding_token):
    if len(line) > num_steps:
        return line[:num_steps]  # Trim
    return line + [padding_token] * (num_steps - len(line))  # Pad


trim_pad(src_vocab[source[0]], 10, src_vocab.pad)
```

```
[47, 4, 0, 0, 0, 0, 0, 0, 0, 0]
```

Now we can convert a list of sentences into an `(num_example, num_steps)` index array. We also record the length of each sentence without the padding tokens, called *valid length*, which might be used by some models. In addition, we add the special "<bos>" and "<eos>" tokens to the target sentences so that our model will know the signals for starting and ending predicting.

```
# Saved in the d2l package for later use
def build_array(lines, vocab, num_steps, is_source):
    lines = [vocab[l] for l in lines]
    if not is_source:
        lines = [[vocab.bos] + l + [vocab.eos] for l in lines]
    array = np.array([trim_pad(l, num_steps, vocab.pad) for l in lines])
    valid_len = (array != vocab.pad).sum(axis=1)
    return array, valid_len
```

Then we can construct minibatches based on these arrays.

9.5.5 Putting All Things Together

Finally, we define the function load_data_nmt to return the data iterator with the vocabularies for source language and target language.

```
# Saved in the d2l package for later use
def load_data_nmt(batch_size, num_steps, num_examples=1000):
    text = preprocess_nmt(read_data_nmt())
    source, target = tokenize_nmt(text, num_examples)
    src_vocab = d2l.Vocab(source, min_freq=3, use_special_tokens=True)
    tgt_vocab = d2l.Vocab(target, min_freq=3, use_special_tokens=True)
    src_array, src_valid_len = build_array(
        source, src_vocab, num_steps, True)
    tgt_array, tgt_valid_len = build_array(
        target, tgt_vocab, num_steps, False)
    data_arrays = (src_array, src_valid_len, tgt_array, tgt_valid_len)
    data_iter = d2l.load_array(data_arrays, batch_size)
    return src_vocab, tgt_vocab, data_iter
```

Let's read the first batch.

```
src_vocab, tgt_vocab, train_iter = load_data_nmt(batch_size=2, num_steps=8)
for X, X_vlen, Y, Y_vlen, in train_iter:
    print('X =', X.astype('int32'), '\nValid lengths for X =', X_vlen,
          '\nY =', Y.astype('int32'), '\nValid lengths for Y =', Y_vlen)
    break
```

```
X = [[96  9  4  0  0  0  0  0]
 [30 37  4  0  0  0  0  0]]
Valid lengths for X = [3 3]
Y = [[ 1  3  5  2  0  0  0  0]
 [ 1 67 59  3  4  2  0  0]]
Valid lengths for Y = [4 6]
```

Summary

- Machine translation (MT) refers to the automatic translation of a segment of text from one language to another.
- We read, preprocess, and tokenize the datasets from both source language and target language.

Exercises

1. Find a machine translation dataset online and process it.

9.6 Encoder-Decoder Architecture

The *encoder-decoder architecture* is a neural network design pattern. As shown in Fig. 9.6.1, the architecture is partitioned into two parts, the encoder and the decoder. The encoder's role is to encode the inputs into state, which often contains several tensors. Then the state is passed into the decoder to generate the outputs. In machine translation, the encoder transforms a source sentence, e.g., "Hello world.", into state, e.g., a vector, that captures its semantic information. The decoder then uses this state to generate the translated target sentence, e.g., "Bonjour le monde.".

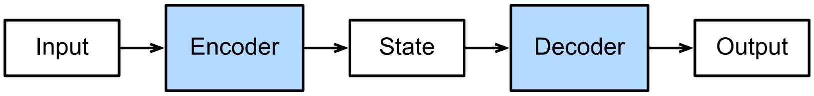


Fig. 9.6.1: The encoder-decoder architecture.

In this section, we will show an interface to implement this encoder-decoder architecture.

9.6.1 Encoder

The encoder is a normal neural network that takes inputs, e.g., a source sentence, to return outputs.

```
from mxnet.gluon import nn

# Saved in the d2l package for later use
class Encoder(nn.Block):
    """The base encoder interface for the encoder-decoder architecture."""
    def __init__(self, **kwargs):
        super(Encoder, self).__init__(**kwargs)

    def forward(self, X):
        raise NotImplementedError
```

9.6.2 Decoder

The decoder has an additional method `init_state` to parse the outputs of the encoder with possible additional information, e.g., the valid lengths of inputs, to return the state it needs. In the forward method, the decoder takes both inputs, e.g., a target sentence and the state. It returns outputs, with potentially modified state if the encoder contains RNN layers.

```
# Saved in the d2l package for later use
class Decoder(nn.Block):
    """The base decoder interface for the encoder-decoder architecture."""
    def __init__(self, **kwargs):
        super(Decoder, self).__init__(**kwargs)

    def init_state(self, enc_outputs, *args):
        raise NotImplementedError

    def forward(self, X, state):
        raise NotImplementedError
```

9.6.3 Model

The encoder-decoder model contains both an encoder and an decoder. We implement its forward method for training. It takes both encoder inputs and decoder inputs, with optional additional arguments. During computation, it first compute encoder outputs to initialize the decoder state, and then returns the decoder outputs.

```
# Saved in the d2l package for later use
class EncoderDecoder(nn.Block):
    """The base class for the encoder-decoder architecture."""
    def __init__(self, encoder, decoder, **kwargs):
        super(EncoderDecoder, self).__init__(**kwargs)
        self.encoder = encoder
        self.decoder = decoder

    def forward(self, enc_X, dec_X, *args):
        enc_outputs = self.encoder(enc_X, *args)
        dec_state = self.decoder.init_state(enc_outputs, *args)
        return self.decoder(dec_X, dec_state)
```

Summary

- An encoder-decoder architecture is a neural network design pattern mainly in natural language processing.
- An encoder is a network (FC, CNN, RNN, etc.) that takes the input, and output a feature map, a vector or a tensor.
- An decoder is a network (usually the same network structure as encoder) that takes the feature vector from the encoder, and gives the best closest match to the actual input or intended output.

Exercises

1. Besides machine translation, can you think of another application scenarios where an encoder-decoder architecture can fit?
2. Can you design a deep encoder-decoder architecture?

9.7 Sequence to Sequence

The sequence to sequence (seq2seq) model is based on the encoder-decoder architecture to generate a sequence output for a sequence input, as demonstrated in Fig. 9.7.1. Both the encoder and the decoder use recurrent neural networks (RNNs) to handle sequence inputs of variable length. The hidden state of the encoder is used directly to initialize the decoder hidden state to pass information from the encoder to the decoder.

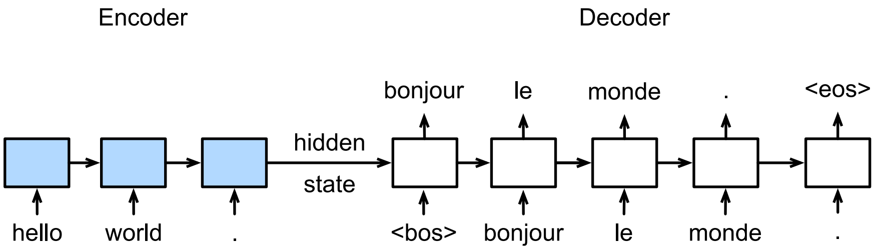


Fig. 9.7.1: The sequence to sequence model architecture.

The layers in the encoder and the decoder are illustrated in Fig. 9.7.2.

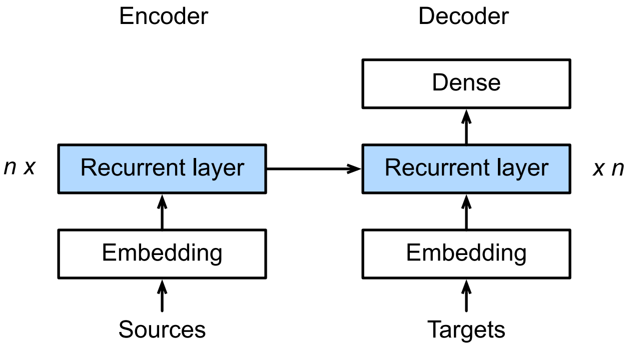


Fig. 9.7.2: Layers in the encoder and the decoder.

In this section we will explain and implement the seq2seq model to train on the machine translation dataset.

```
import d2l
from mxnet import np, npx, init, gluon, autograd
from mxnet.gluon import nn, rnn
npx.set_np()
```

9.7.1 Encoder

Recall that the encoder of seq2seq can transform the inputs of variable length to a fixed-length context vector $\mathbf{c}$ by encoding the sequence information into $\mathbf{c}$. We usually use RNN layers within the encoder. Suppose that we have an input sequence $x_1, \ldots, x_T$, where x_t is the t^{th} word. At timestep t, the RNN will have two vectors as the input: the feature vector $\mathbf{x}_t$ of x_t and the hidden state of the last timestep $\mathbf{h}_{t-1}$. Let's denote the transformation of the RNN's hidden states by a function f:

$$\mathbf{h}_t = f(\mathbf{x}_t, \mathbf{h}_{t-1}). \tag{9.7.1}$$

Next, the encoder captures information of all the hidden states and encodes it into the context vector $\mathbf{c}$ with a function q:

$$\mathbf{c} = q(\mathbf{h}_1, \ldots, \mathbf{h}_T). \tag{9.7.2}$$

For example, if we choose q as $q(\mathbf{h}_1, \ldots, \mathbf{h}_T) = \mathbf{h}_T$, then the context vector will be the final hidden state $\mathbf{h}_T$.

So far what we describe above is a unidirectional RNN, where each timestep's hidden state only depends on the previous timesteps'. We can also use other forms of RNNs such as GRUs, LSTMs and, bidirectional RNNs to encode the sequential input.

Now let's implement the seq2seq's encoder. Here we use the word embedding layer to obtain the feature vector according to the word index of the input language. Those feature vectors will be fed to a multi-layer LSTM. The input for the encoder is a batch of sequences, which is 2-D tensor with shape (batch size, sequence length). The encoder returns both the LSTM outputs, i.e., hidden states of all the timesteps, as well as the hidden state and the memory cell of the final timestep.

```python
# Saved in the d2l package for later use
class Seq2SeqEncoder(d2l.Encoder):
    def __init__(self, vocab_size, embed_size, num_hiddens, num_layers,
                 dropout=0, **kwargs):
        super(Seq2SeqEncoder, self).__init__(**kwargs)
        self.embedding = nn.Embedding(vocab_size, embed_size)
        self.rnn = rnn.LSTM(num_hiddens, num_layers, dropout=dropout)

    def forward(self, X, *args):
        X = self.embedding(X)  # X shape: (batch_size, seq_len, embed_size)
        # RNN needs first axes to be timestep, i.e., seq_len
        X = X.swapaxes(0, 1)
        state = self.rnn.begin_state(batch_size=X.shape[1], ctx=X.context)
        out, state = self.rnn(X, state)
        # out shape: (seq_len, batch_size, num_hiddens)
        # state shape: (num_layers, batch_size, num_hiddens),
        # where "state" contains the hidden state and the memory cell
        return out, state
```

Next, we will create a minibatch sequence input with a batch size of 4 and 7 timesteps. We assume the number of hidden layers of the LSTM unit is 2 and the number of hidden units is 16. The output shape returned by the encoder after performing forward calculation on the input is (number of timesteps, batch size, number of hidden units). The shape of the multi-layer hidden state of the gated recurrent unit in the final timestep is (number of hidden layers, batch size, number of hidden units). For the gated recurrent unit, the `state` list contains only one element, which is the hidden state. If long short-term memory is used, the `state` list will also contain another element, which is the memory cell.

```
encoder = Seq2SeqEncoder(vocab_size=10, embed_size=8, num_hiddens=16,
                         num_layers=2)
encoder.initialize()
X = np.zeros((4, 7))
output, state = encoder(X)
output.shape
```

```
(7, 4, 16)
```

Since an LSTM is used, the `state` list will contain both the hidden state and the memory cell with same shape (number of hidden layers, batch size, number of hidden units). However, if a GRU is used, the `state` list will contain only one element—the hidden state in the final timestep with shape (number of hidden layers, batch size, number of hidden units).

```
len(state), state[0].shape, state[1].shape
```

```
(2, (2, 4, 16), (2, 4, 16))
```

9.7.2 Decoder

As we just introduced, the context vector $\mathbf{c}$ encodes the information from the whole input sequence $x_1, \ldots, x_T$. Suppose that the given outputs in the training set are $y_1, \ldots, y_{T'}$. At each timestep t', the conditional probability of output $y_{t'}$ will depend on the previous output sequence $y_1, \ldots, y_{t'-1}$ and the context vector $\mathbf{c}$, i.e.,

$$P(y_{t'} \mid y_1, \ldots, y_{t'-1}, \mathbf{c}). \tag{9.7.3}$$

Hence, we can use another RNN as the decoder. At timestep t', the decoder will update its hidden state $\mathbf{s}_{t'}$ using three inputs: the feature vector $\mathbf{y}_{t'-1}$ of $y_{t'-1}$, the context vector $\mathbf{c}$, and the hidden state of the last timestep $\mathbf{s}_{t'-1}$. Let's denote the transformation of the RNN's hidden states within the decoder by a function g:

$$\mathbf{s}_{t'} = g(\mathbf{y}_{t'-1}, \mathbf{c}, \mathbf{s}_{t'-1}). \tag{9.7.4}$$

When implementing the decoder, we directly use the hidden state of the encoder in the final timestep as the initial hidden state of the decoder. This requires that the encoder and decoder RNNs have the same numbers of layers and hidden units. The LSTM forward calculation of the decoder is similar to that of the encoder. The only difference is that we add a dense layer after the LSTM layers, where the hidden size is the vocabulary size. The dense layer will predict the confidence score for each word.

```
# Saved in the d2l package for later use
class Seq2SeqDecoder(d2l.Decoder):
    def __init__(self, vocab_size, embed_size, num_hiddens, num_layers,
                 dropout=0, **kwargs):
        super(Seq2SeqDecoder, self).__init__(**kwargs)
        self.embedding = nn.Embedding(vocab_size, embed_size)
        self.rnn = rnn.LSTM(num_hiddens, num_layers, dropout=dropout)
        self.dense = nn.Dense(vocab_size, flatten=False)
```

(continues on next page)

(continued from previous page)

```
    def init_state(self, enc_outputs, *args):
        return enc_outputs[1]

    def forward(self, X, state):
        X = self.embedding(X).swapaxes(0, 1)
        out, state = self.rnn(X, state)
        # Make the batch to be the first dimension to simplify loss
        # computation
        out = self.dense(out).swapaxes(0, 1)
        return out, state
```

We create an decoder with the same hyper-parameters as the encoder. As we can see, the output shape is changed to (batch size, the sequence length, vocabulary size).

```
decoder = Seq2SeqDecoder(vocab_size=10, embed_size=8,
                         num_hiddens=16, num_layers=2)
decoder.initialize()
state = decoder.init_state(encoder(X))
out, state = decoder(X, state)
out.shape, len(state), state[0].shape, state[1].shape
```

```
((4, 7, 10), 2, (2, 4, 16), (2, 4, 16))
```

9.7.3 The Loss Function

For each timestep, the decoder outputs a vocabulary-size confidence score vector to predict words. Similar to language modeling, we can apply softmax to obtain the probabilities and then use cross-entropy loss to calculate the loss. Note that we padded the target sentences to make them have the same length, but we do not need to compute the loss on the padding symbols.

To implement the loss function that filters out some entries, we will use an operator called `SequenceMask`. It can specify to mask the first dimension (`axis=0`) or the second one (`axis=1`). If the second one is chosen, given a valid length vector `len` and 2-dim input `X`, this operator sets `X[i, len[i]:] = 0` for all i's.

```
X = np.array([[1, 2, 3], [4, 5, 6]])
npx.sequence_mask(X, np.array([1, 2]), True, axis=1)
```

```
array([[1., 0., 0.],
       [4., 5., 0.]])
```

Apply to n-dim tensor X, it sets `X[i, len[i]:, :, ..., :] = 0`. In addition, we can specify the filling value such as -1 as shown below.

```
X = np.ones((2, 3, 4))
npx.sequence_mask(X, np.array([1, 2]), True, value=-1, axis=1)
```

```
array([[[ 1.,  1.,  1.,  1.],
        [-1., -1., -1., -1.],
        [-1., -1., -1., -1.]],

       [[ 1.,  1.,  1.,  1.],
        [ 1.,  1.,  1.,  1.],
        [-1., -1., -1., -1.]]])
```

Now we can implement the masked version of the softmax cross-entropy loss. Note that each Gluon loss function allows to specify per-example weights, in default they are 1s. Then we can just use a zero weight for each example we would like to remove. So our customized loss function accepts an additional `valid_length` argument to ignore some failing elements in each sequence.

```
# Saved in the d2l package for later use
class MaskedSoftmaxCELoss(gluon.loss.SoftmaxCELoss):
    # pred shape: (batch_size, seq_len, vocab_size)
    # label shape: (batch_size, seq_len)
    # valid_length shape: (batch_size, )
    def forward(self, pred, label, valid_length):
        # weights shape: (batch_size, seq_len, 1)
        weights = np.expand_dims(np.ones_like(label), axis=-1)
        weights = npx.sequence_mask(weights, valid_length, True, axis=1)
        return super(MaskedSoftmaxCELoss, self).forward(pred, label, weights)
```

For a sanity check, we create identical three sequences, keep 4 elements for the first sequence, 2 elements for the second sequence, and none for the last one. Then the first example loss should be 2 times larger than the second one, and the last loss should be 0.

```
loss = MaskedSoftmaxCELoss()
loss(np.ones((3, 4, 10)), np.ones((3, 4)), np.array([4, 2, 0]))
```

```
array([2.3025851, 1.1512926, 0.       ])
```

9.7.4 Training

During training, if the target sequence has length n, we feed the first $n-1$ tokens into the decoder as inputs, and the last $n-1$ tokens are used as ground truth label.

```
# Saved in the d2l package for later use
def train_s2s_ch9(model, data_iter, lr, num_epochs, ctx):
    model.initialize(init.Xavier(), force_reinit=True, ctx=ctx)
    trainer = gluon.Trainer(model.collect_params(),
                            'adam', {'learning_rate': lr})
    loss = MaskedSoftmaxCELoss()
    animator = d2l.Animator(xlabel='epoch', ylabel='loss',
                            xlim=[1, num_epochs], ylim=[0, 0.25])
    for epoch in range(1, num_epochs + 1):
        timer = d2l.Timer()
        metric = d2l.Accumulator(2)  # loss_sum, num_tokens
        for batch in data_iter:
            X, X_vlen, Y, Y_vlen = [x.as_in_context(ctx) for x in batch]
```

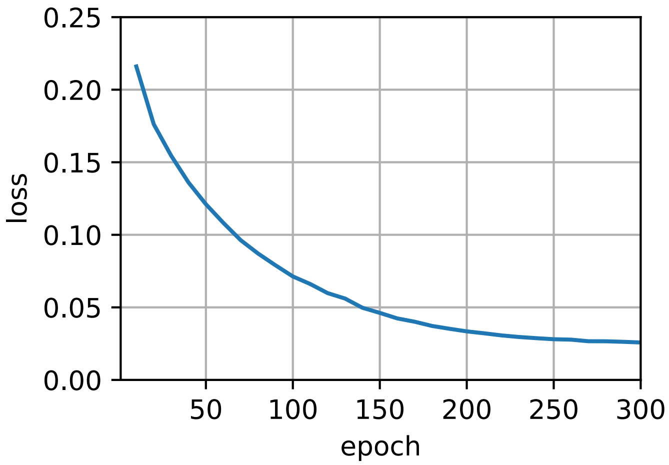
(continues on next page)

```
            Y_input, Y_label, Y_vlen = Y[:, :-1], Y[:, 1:], Y_vlen-1
            with autograd.record():
                Y_hat, _ = model(X, Y_input, X_vlen, Y_vlen)
                l = loss(Y_hat, Y_label, Y_vlen)
            l.backward()
            d2l.grad_clipping(model, 1)
            num_tokens = Y_vlen.sum()
            trainer.step(num_tokens)
            metric.add(l.sum(), num_tokens)
        if epoch % 10 == 0:
            animator.add(epoch, (metric[0]/metric[1],))
    print('loss %.3f, %d tokens/sec on %s ' % (
        metric[0]/metric[1], metric[1]/timer.stop(), ctx))
```

Next, we create a model instance and set hyper-parameters. Then, we can train the model.

```
embed_size, num_hiddens, num_layers, dropout = 32, 32, 2, 0.0
batch_size, num_steps = 64, 10
lr, num_epochs, ctx = 0.005, 300, d2l.try_gpu()

src_vocab, tgt_vocab, train_iter = d2l.load_data_nmt(batch_size, num_steps)
encoder = Seq2SeqEncoder(
    len(src_vocab), embed_size, num_hiddens, num_layers, dropout)
decoder = Seq2SeqDecoder(
    len(tgt_vocab), embed_size, num_hiddens, num_layers, dropout)
model = d2l.EncoderDecoder(encoder, decoder)
train_s2s_ch9(model, train_iter, lr, num_epochs, ctx)
```

```
loss 0.026, 8820 tokens/sec on gpu(0)
```



9.7.5 Predicting

Here we implement the simplest method, greedy search, to generate an output sequence. As illustrated in Fig. 9.7.3, during predicting, we feed the same "<bos>" token to the decoder as training at timestep 0. But the input token for a later timestep is the predicted token from the previous timestep.

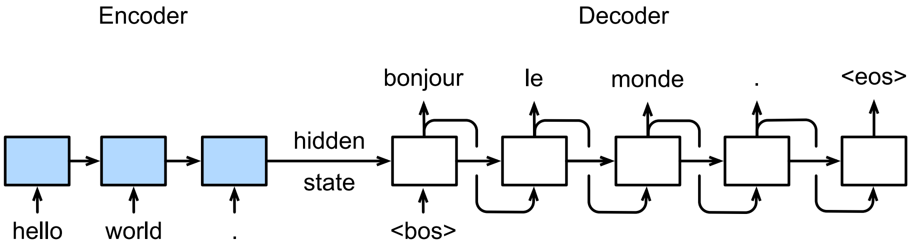


Fig. 9.7.3: Sequence to sequence model predicting with greedy search

```
# Saved in the d2l package for later use
def predict_s2s_ch9(model, src_sentence, src_vocab, tgt_vocab, num_steps,
                    ctx):
    src_tokens = src_vocab[src_sentence.lower().split(' ')]
    enc_valid_length = np.array([len(src_tokens)], ctx=ctx)
    src_tokens = d2l.trim_pad(src_tokens, num_steps, src_vocab.pad)
    enc_X = np.array(src_tokens, ctx=ctx)
    # Add the batch_size dimension
    enc_outputs = model.encoder(np.expand_dims(enc_X, axis=0),
                                enc_valid_length)
    dec_state = model.decoder.init_state(enc_outputs, enc_valid_length)
    dec_X = np.expand_dims(np.array([tgt_vocab.bos], ctx=ctx), axis=0)
    predict_tokens = []
    for _ in range(num_steps):
        Y, dec_state = model.decoder(dec_X, dec_state)
        # The token with highest score is used as the next timestep input
        dec_X = Y.argmax(axis=2)
        py = dec_X.squeeze(axis=0).astype('int32').item()
        if py == tgt_vocab.eos:
            break
        predict_tokens.append(py)
    return ' '.join(tgt_vocab.to_tokens(predict_tokens))
```

Try several examples:

```
for sentence in ['Go .', 'Wow !', "I'm OK .", 'I won !']:
    print(sentence + ' => ' + predict_s2s_ch9(
        model, sentence, src_vocab, tgt_vocab, num_steps, ctx))
```

```
Go . => va !
Wow ! => <unk> !
I'm OK . => ça va .
I won ! => je l'ai emporté !
```

Summary

- The sequence to sequence (seq2seq) model is based on the encoder-decoder architecture to generate a sequence output from a sequence input.
- We use multiple LSTM layers for both the encoder and decoder.

Exercises

1. Can you think of other use cases of seq2seq besides neural machine translation?
2. What if the input sequence in the example of this section is longer?
3. If we do not use the `SequenceMask` in the loss function, what may happen?

9.8 Beam Search

In Section 9.7, we discussed how to train an encoder-decoder with input and output sequences that are both of variable length. In this section, we are going to introduce how to use the encoder-decoder to predict sequences of variable length.

As in Section 9.5, when preparing to train the dataset, we normally attach a special symbol "<eos>" after each sentence to indicate the termination of the sequence. We will continue to use this mathematical symbol in the discussion below. For ease of discussion, we assume that the output of the decoder is a sequence of text. Let the size of output text dictionary $\mathcal{Y}$ (contains special symbol "<eos>") be $|\mathcal{Y}|$, and the maximum length of the output sequence be T'. There are a total $\mathcal{O}(|\mathcal{Y}|^{T'})$ types of possible output sequences. All the subsequences after the special symbol "<eos>" in these output sequences will be discarded. Besides, we still denote the context vector as $\mathbf{c}$, which encodes information of all the hidden states from the input.

9.8.1 Greedy Search

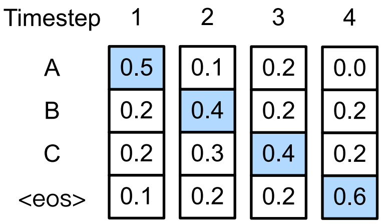
First, we will take a look at a simple solution: greedy search. For any timestep t' of the output sequence, we are going to search for the word with the highest conditional probability from $|\mathcal{Y}|$ numbers of words, with

$$y_{t'} = \operatorname*{argmax}_{y \in \mathcal{Y}} P(y \mid y_1, \ldots, y_{t'-1}, \mathbf{c}) \tag{9.8.1}$$

as the output. Once the "<eos>" symbol is detected, or the output sequence has reached its maximum length T', the output is completed.

As we mentioned in our discussion of the decoder, the conditional probability of generating an output sequence based on the input sequence is $\prod_{t'=1}^{T'} P(y_{t'} \mid y_1, \ldots, y_{t'-1}, \mathbf{c})$. We will take the output sequence with the highest conditional probability as the optimal sequence. The main problem with greedy search is that there is no guarantee that the optimal sequence will be obtained.

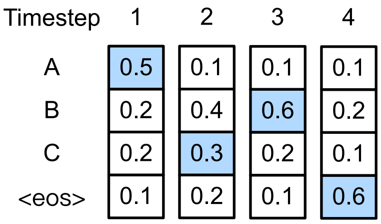
Take a look at the example below. We assume that there are four words "A", "B", "C", and "<eos>" in the output dictionary. The four numbers under each timestep in Fig. 9.8.1 represent the conditional probabilities of generating "A", "B", "C", and "<eos>" at that timestep, respectively. At each timestep, greedy search selects the word with the highest conditional probability. Therefore, the output sequence "A", "B", "C", and "<eos>" will be generated in Fig. 9.8.1. The conditional probability of this output sequence is $0.5 \times 0.4 \times 0.4 \times 0.6 = 0.048$.



Timestep	1	2	3	4
A	**0.5**	0.1	0.2	0.0
B	0.2	**0.4**	0.2	0.2
C	0.2	0.3	**0.4**	0.2
<eos>	0.1	0.2	0.2	**0.6**

Fig. 9.8.1: The four numbers under each timestep represent the conditional probabilities of generating "A", "B", "C", and "<eos>" at that timestep, respectively. At each timestep, greedy search selects the word with the highest conditional probability.

Now, we will look at another example shown in Fig. 9.8.2. Unlike in Fig. 9.8.1, the following figure Fig. 9.8.2 selects the word "C", which has the second highest conditional probability at timestep 2. Since the output subsequences of timesteps 1 and 2, on which timestep 3 is based, are changed from "A" and "B" in Fig. 9.8.1 to "A" and "C" in Fig. 9.8.2, the conditional probability of each word generated at timestep 3 has also changed in Fig. 9.8.2. We choose the word "B", which has the highest conditional probability. Now, the output subsequences of timestep 4 based on the first three timesteps are "A", "C", and "B", which are different from "A", "B", and "C" in Fig. 9.8.1. Therefore, the conditional probability of generating each word in timestep 4 in Fig. 9.8.2 is also different from that in Fig. 9.8.1. We find that the conditional probability of the output sequence "A", "C", "B", and "<eos>" at the current timestep is $0.5 \times 0.3 \times 0.6 \times 0.6 = 0.054$, which is higher than the conditional probability of the output sequence obtained by greedy search. Therefore, the output sequence "A", "B", "C", and "<eos>" obtained by the greedy search is not an optimal sequence.



Timestep	1	2	3	4
A	**0.5**	0.1	0.1	0.1
B	0.2	0.4	**0.6**	0.2
C	0.2	**0.3**	0.2	0.1
<eos>	0.1	0.2	0.1	**0.6**

Fig. 9.8.2: The four numbers under each timestep represent the conditional probabilities of generating "A", "B", "C", and "<eos>" at that timestep. At timestep 2, the word "C", which has the second highest conditional probability, is selected.

9.8.2 Exhaustive Search

If the goal is to obtain the optimal sequence, we may consider using exhaustive search: an exhaustive examination of all possible output sequences, which outputs the sequence with the highest conditional probability.

Although we can use an exhaustive search to obtain the optimal sequence, its computational overhead $\mathcal{O}(|\mathcal{Y}|^{T'})$ is likely to be excessively high. For example, when $|\mathcal{Y}| = 10000$ and $T' = 10$, we will need to evaluate $10000^{10} = 10^{40}$ sequences. This is next to impossible to complete. The computational overhead of greedy search is $\mathcal{O}(|\mathcal{Y}| T')$, which is usually significantly less than the computational overhead of an exhaustive search. For example, when $|\mathcal{Y}| = 10000$ and $T' = 10$, we only need to evaluate $10000 \times 10 = 1 \times 10^5$ sequences.

9.8.3 Beam Search

Beam search is an improved algorithm based on greedy search. It has a hyper-parameter named *beam size*, k. At timestep 1, we select k words with the highest conditional probability to be the first word of the k candidate output sequences. For each subsequent timestep, we are going to select the k output sequences with the highest conditional probability from the total of $k\,|\mathcal{Y}|$ possible output sequences based on the k candidate output sequences from the previous timestep. These will be the candidate output sequence for that timestep. Finally, we will filter out the sequences containing the special symbol "<eos>" from the candidate output sequences of each timestep and discard all the subsequences after it to obtain a set of final candidate output sequences.

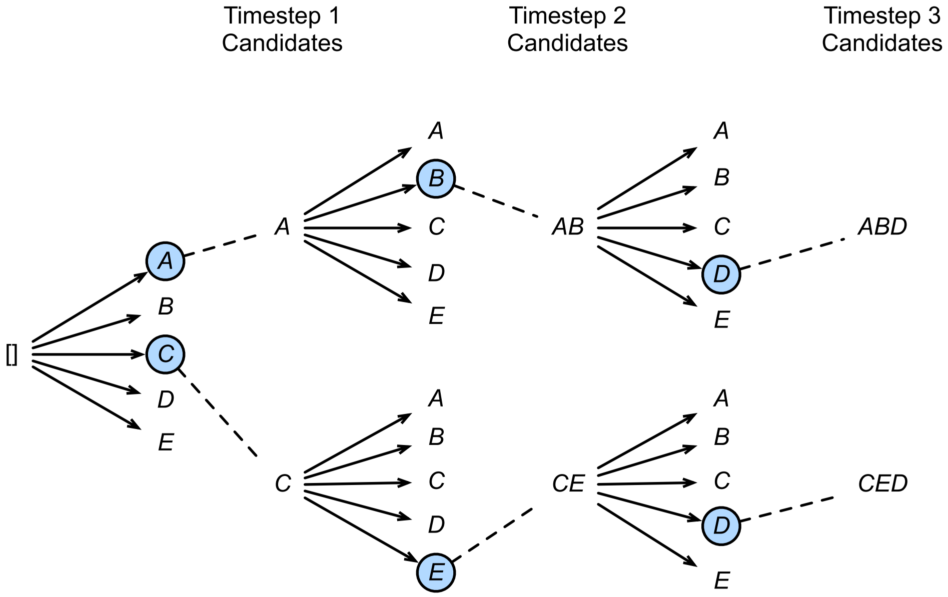


Fig. 9.8.3: The beam search process. The beam size is 2 and the maximum length of the output sequence is 3. The candidate output sequences are A, C, AB, CE, ABD, and CED.

Fig. 9.8.3 demonstrates the process of beam search with an example. Suppose that the vocabulary of the output sequence only contains five elements: $\mathcal{Y} = \{A, B, C, D, E\}$ where one of them is a special symbol "<eos>". Set beam size to 2, the maximum length of the output sequence to 3. At timestep 1 of the output sequence, suppose the words with the highest conditional probability

$P(y_1 \mid \mathbf{c})$ are A and C. At timestep 2, for all $y_2 \in \mathcal{Y}$, we compute

$$P(A, y_2 \mid \mathbf{c}) = P(A \mid \mathbf{c})P(y_2 \mid A, \mathbf{c}) \tag{9.8.2}$$

and

$$P(C, y_2 \mid \mathbf{c}) = P(C \mid \mathbf{c})P(y_2 \mid C, \mathbf{c}), \tag{9.8.3}$$

and pick the largest two among these 10 values, say

$$P(A, B \mid \mathbf{c}) \text{ and } P(C, E \mid \mathbf{c}). \tag{9.8.4}$$

Then at timestep 3, for all $y_3 \in \mathcal{Y}$, we compute

$$P(A, B, y_3 \mid \mathbf{c}) = P(A, B \mid \mathbf{c})P(y_3 \mid A, B, \mathbf{c}) \tag{9.8.5}$$

and

$$P(C, E, y_3 \mid \mathbf{c}) = P(C, E \mid \mathbf{c})P(y_3 \mid C, E, \mathbf{c}), \tag{9.8.6}$$

and pick the largest two among these 10 values, say

$$P(A, B, D \mid \mathbf{c}) \text{ and } P(C, E, D \mid \mathbf{c}). \tag{9.8.7}$$

As a result, we obtain 6 candidates output sequences: (1) A; (2) C; (3) A, B; (4) C, E; (5) A, B, D; and (6) C, E, D. In the end, we will get the set of final candidate output sequences based on these 6 sequences.

In the set of final candidate output sequences, we will take the sequence with the highest score as the output sequence from those below:

$$\frac{1}{L^\alpha} \log P(y_1, \ldots, y_L) = \frac{1}{L^\alpha} \sum_{t'=1}^{L} \log P(y_{t'} \mid y_1, \ldots, y_{t'-1}, \mathbf{c}), \tag{9.8.8}$$

Here, L is the length of the final candidate sequence and the selection for α is generally 0.75. The L^α on the denominator is a penalty on the logarithmic addition scores for the longer sequences above. The computational overhead $\mathcal{O}(k \left|\mathcal{Y}\right| T')$ of the beam search can be obtained through analysis. The result is between the computational overhead of greedy search and exhaustive search. In addition, greedy search can be treated as a beam search with a beam size of 1. Beam search strikes a balance between computational overhead and search quality using a flexible beam size of k.

Summary

- Methods for predicting variable-length sequences include greedy search, exhaustive search, and beam search.
- Beam search strikes a balance between computational overhead and search quality using a flexible beam size.

Exercises

1. Can we treat an exhaustive search as a beam search with a special beam size? Why?
2. We used language models to generate sentences in Section 8.5. Which kind of search does this output use? Can you improve it?

10 | Attention Mechanisms

As a bit of a historical digression, attention research is an enormous field with a long history in cognitive neuroscience. Focalization, concentration of consciousness are of the essence of attention, which enable the human to prioritize the perception in order to deal effectively with others. As a result, we do not process all the information that is available in the sensory input. At any time, we are aware of only a small fraction of the information in the environment. In cognitive neuroscience, there are several types of attention such as selective attention, covert attention, and spatial attention. The theory ignites the spark in recent deep learning is the *feature integration theory* of the selective attention, which was developed by Anne Treisman and Garry Gelade through the paper (Treisman & Gelade, 1980) in 1980. This paper declares that when perceiving a stimulus, features are registered early, automatically, and in parallel, while objects are identified separately and at a later stage in processing. The theory has been one of the most influential psychological models of human visual attention.

However, we will not indulge in too much theory of attention in neuroscience, but rather focus on applying the attention idea in deep learning, where attention can be seen as a generalized pooling method with bias alignment over inputs. In this chapter, we will provide you with some intuition about how to transform the attention idea to the concrete mathematics models, and make them work.

10.1 Attention Mechanisms

In Section 9.7, we encode the source sequence input information in the recurrent unit state and then pass it to the decoder to generate the target sequence. A token in the target sequence may closely relate to one or more tokens in the source sequence, instead of the whole source sequence. For example, when translating "Hello world." to "Bonjour le monde.", "Bonjour" maps to "Hello" and "monde" maps to "world". In the seq2seq model, the decoder may implicitly select the corresponding information from the state passed by the encoder. The attention mechanism, however, makes this selection explicit.

Attention is a generalized pooling method with bias alignment over inputs. The core component in the attention mechanism is the attention layer, or called *attention* for simplicity. An input of the attention layer is called a *query*. For a query, attention returns an output based on the memory—a set of key-value pairs encoded in the attention layer. To be more specific, assume that the memory contains n key-value pairs, $(\mathbf{k}_1, \mathbf{v}_1), \ldots, (\mathbf{k}_n, \mathbf{v}_n)$, with $\mathbf{k}_i \in \mathbb{R}^{d_k}$, $\mathbf{v}_i \in \mathbb{R}^{d_v}$. Given a query $\mathbf{q} \in \mathbb{R}^{d_q}$, the attention layer returns an output $\mathbf{o} \in \mathbb{R}^{d_v}$ with the same shape as the value.

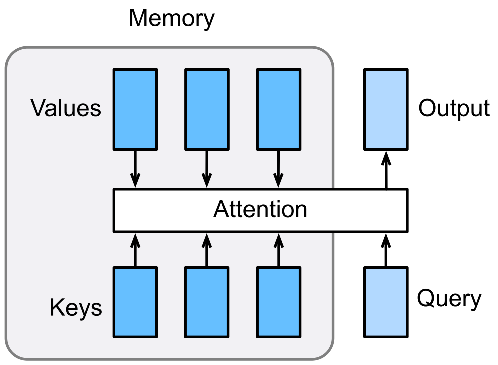


Fig. 10.1.1: The attention layer returns an output based on the input query and its memory.

The full process of attention mechanism is expressed in Fig. 10.1.2. To compute the output of attention, we first use a score function α that measures the similarity between the query and key. Then for each key $(\mathbf{k}_1, \mathbf{v}_1), \ldots, (\mathbf{k}_n, \mathbf{v}_n)$, we compute the scores $a_1, \ldots, a_n$ by

$$a_i = \alpha(\mathbf{q}, \mathbf{k}_i). \tag{10.1.1}$$

Next we use softmax to obtain the attention weights, i.e.,

$$\mathbf{b} = \text{softmax}(\mathbf{a}) \quad \text{, where} \quad b_i = \frac{\exp(a_i)}{\sum_j \exp(a_j)}, \mathbf{b} = [b_1, \ldots, b_n]^T. \tag{10.1.2}$$

Finally, the output is a weighted sum of the values:

$$\mathbf{o} = \sum_{i=1}^{n} b_i \mathbf{v}_i. \tag{10.1.3}$$

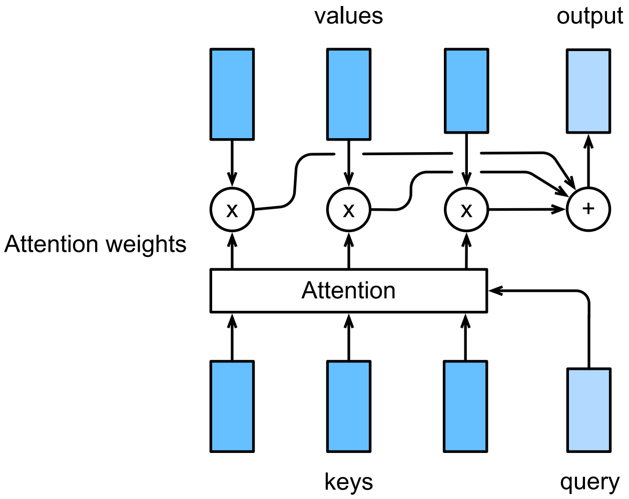


Fig. 10.1.2: The attention output is a weighted sum of the values.

Different choices of the score function lead to different attention layers. Below, we introduce two commonly used attention layers. Before diving into the implementation, we first express two operators to get you up and running: a masked version of the softmax operator `masked_softmax` and a specialized dot operator `batched_dot`.

```
import math
from mxnet import np, npx
from mxnet.gluon import nn
npx.set_np()
```

The masked softmax takes a 3-dimensional input and enables us to filter out some elements by specifying a valid length for the last dimension. (Refer to Section 9.5 for the definition of a valid length.) As a result, any value outside the valid length will be masked as 0. Let's implement the `masked_softmax` function.

```
# Saved in the d2l package for later use
def masked_softmax(X, valid_length):
    # X: 3-D tensor, valid_length: 1-D or 2-D tensor
    if valid_length is None:
        return npx.softmax(X)
    else:
        shape = X.shape
        if valid_length.ndim == 1:
            valid_length = valid_length.repeat(shape[1], axis=0)
        else:
            valid_length = valid_length.reshape(-1)
        # Fill masked elements with a large negative, whose exp is 0
        X = npx.sequence_mask(X.reshape(-1, shape[-1]), valid_length, True,
                              axis=1, value=-1e6)
        return npx.softmax(X).reshape(shape)
```

To illustrate how this function works, we construct two 2×4 matrices as the input. In addition, we specify that the valid length equals to 2 for the first example, and 3 for the second example. Then, as we can see from the following outputs, the values outside valid lengths are masked as zero.

```
masked_softmax(np.random.uniform(size=(2, 2, 4)), np.array([2, 3]))
```

```
array([[[0.488994  , 0.511006  , 0.        , 0.        ],
        [0.43654838, 0.56345165, 0.        , 0.        ]],

       [[0.28817102, 0.3519408 , 0.3598882 , 0.        ],
        [0.29034293, 0.25239873, 0.45725834, 0.        ]]])
```

Moreover, the second operator `batched_dot` takes two inputs X and Y with shapes (b, n, m) and (b, m, k), respectively, and returns an output with shape (b, n, k). To be specific, it computes b dot products for $i = \{1, \ldots, b\}$, i.e.,

$$Z[i, :, :] = X[i, :, :]Y[i, :, :]. \tag{10.1.4}$$

```
npx.batch_dot(np.ones((2, 1, 3)), np.ones((2, 3, 2)))
```

```
array([[[3., 3.]],

       [[3., 3.]]])
```

10.1.1 Dot Product Attention

Equipped with the above two operators: `masked_softmax` and `batched_dot`, let's dive into the details of two widely used attentions layers. The first one is the *dot product attention*: it assumes that the query has the same dimension as the keys, namely $\mathbf{q}, \mathbf{k}_i \in \mathbb{R}^d$ for all i. The dot product attention computes the scores by an dot product between the query and a key, which is then divided by $\sqrt{d}$ to minimize the unrelated influence of the dimension d on the scores. In other words,

$$\alpha(\mathbf{q}, \mathbf{k}) = \langle \mathbf{q}, \mathbf{k} \rangle / \sqrt{d}. \tag{10.1.5}$$

Beyond the single-dimensional queries and keys, we can always generalize them to multi-dimensional queries and keys. Assume that $\mathbf{Q} \in \mathbb{R}^{m \times d}$ contains m queries and $\mathbf{K} \in \mathbb{R}^{n \times d}$ has all the n keys. We can compute all mn scores by

$$\alpha(\mathbf{Q}, \mathbf{K}) = \mathbf{Q}\mathbf{K}^\top / \sqrt{d}. \tag{10.1.6}$$

With (10.1.6), we can implement the dot product attention layer `DotProductAttention` that supports a batch of queries and key-value pairs. In addition, for regularization we also use a dropout layer.

```python
# Saved in the d2l package for later use
class DotProductAttention(nn.Block):
    def __init__(self, dropout, **kwargs):
        super(DotProductAttention, self).__init__(**kwargs)
        self.dropout = nn.Dropout(dropout)

    # query: (batch_size, #queries, d)
    # key: (batch_size, #kv_pairs, d)
    # value: (batch_size, #kv_pairs, dim_v)
    # valid_length: either (batch_size, ) or (batch_size, xx)
    def forward(self, query, key, value, valid_length=None):
        d = query.shape[-1]
        # Set transpose_b=True to swap the last two dimensions of key
        scores = npx.batch_dot(query, key, transpose_b=True) / math.sqrt(d)
        attention_weights = self.dropout(masked_softmax(scores, valid_length))
        return npx.batch_dot(attention_weights, value)
```

Let's test the class `DotProductAttention` in a toy example. First, create two batches, where each batch has one query and 10 key-value pairs.

Via the `valid_length` argument, we specify that we will check the first 2 key-value pairs for the first batch and 6 for the second one. Therefore, even though both batches have the same query and key-value pairs, we obtain different outputs.

```python
atten = DotProductAttention(dropout=0.5)
atten.initialize()
keys = np.ones((2, 10, 2))
values = np.arange(40).reshape(1, 10, 4).repeat(2, axis=0)
atten(np.ones((2, 1, 2)), keys, values, np.array([2, 6]))
```

```
array([[[ 2.        ,  3.        ,  4.        ,  5.        ]],

       [[10.        , 11.        , 12.000001, 13.        ]]])
```

As we can see above, dot product attention simply multiplies the query and key together, and hopes to derive their similarities from there. Whereas, the query and key may not be of the same dimension. To address such an issue, we may resort to the multilayer perceptron attention.

10.1.2 Multilayer Perceptron Attention

In *multilayer perceptron attention*, we project both query and keys into $\mathbb{R}^h$ by learnable weights parameters. Assume that the learnable weights are $\mathbf{W}_k \in \mathbb{R}^{h \times d_k}$, $\mathbf{W}_q \in \mathbb{R}^{h \times d_q}$, and $\mathbf{v} \in \mathbb{R}^h$. Then the score function is defined by

$$\alpha(\mathbf{k}, \mathbf{q}) = \mathbf{v}^\top \text{tanh}(\mathbf{W}_k \mathbf{k} + \mathbf{W}_q \mathbf{q}). \tag{10.1.7}$$

Intuitively, you can imagine $\mathbf{W}_k \mathbf{k} + \mathbf{W}_q \mathbf{q}$ as concatenating the key and value in the feature dimension and feeding them to a single hidden layer perceptron with hidden layer size h and output layer size 1. In this hidden layer, the activation function is tanh and no bias is applied. Now let's implement the multilayer perceptron attention.

```
# Saved in the d2l package for later use
class MLPAttention(nn.Block):
    def __init__(self, units, dropout, **kwargs):
        super(MLPAttention, self).__init__(**kwargs)
        # Use flatten=True to keep query's and key's 3-D shapes
        self.W_k = nn.Dense(units, activation='tanh',
                            use_bias=False, flatten=False)
        self.W_q = nn.Dense(units, activation='tanh',
                            use_bias=False, flatten=False)
        self.v = nn.Dense(1, use_bias=False, flatten=False)
        self.dropout = nn.Dropout(dropout)

    def forward(self, query, key, value, valid_length):
        query, key = self.W_k(query), self.W_q(key)
        # Expand query to (batch_size, #querys, 1, units), and key to
        # (batch_size, 1, #kv_pairs, units). Then plus them with broadcast
        features = np.expand_dims(query, axis=2) + np.expand_dims(key, axis=1)
        scores = np.squeeze(self.v(features), axis=-1)
        attention_weights = self.dropout(masked_softmax(scores, valid_length))
        return npx.batch_dot(attention_weights, value)
```

To test the above `MLPAttention` class, we use the same inputs as in the previous toy example. As we can see below, despite `MLPAttention` containing an additional MLP model, we obtain the same outputs as for `DotProductAttention`.

```
atten = MLPAttention(units=8, dropout=0.1)
atten.initialize()
atten(np.ones((2, 1, 2)), keys, values, np.array([2, 6]))
```

```
array([[[ 2.        ,  3.        ,  4.        ,  5.        ]],

       [[10.        , 11.        , 12.000001, 13.        ]]])
```

Summary

- An attention layer explicitly selects related information.
- An attention layer's memory consists of key-value pairs, so its output is close to the values whose keys are similar to the queries.
- Two commonly used attention models are dot product attention and multilayer perceptron attention.

Exercises

1. What are the advantages and disadvantages for dot product attention and multilayer perceptron attention, respectively?

10.2 Sequence to Sequence with Attention Mechanisms

In this section, we add the attention mechanism to the sequence to sequence (seq2seq) model as introduced in Section 9.7 to explicitly aggregate states with weights. Fig. 10.2.1 shows the model architecture for encoding and decoding at the timestep t. Here, the memory of the attention layer consists of all the information that the encoder has seen—the encoder output at each timestep. During the decoding, the decoder output from the previous timestep $t-1$ is used as the query. The output of the attention model is viewed as the context information, and such context is concatenated with the decoder input D_t. Finally, we feed the concatenation into the decoder.

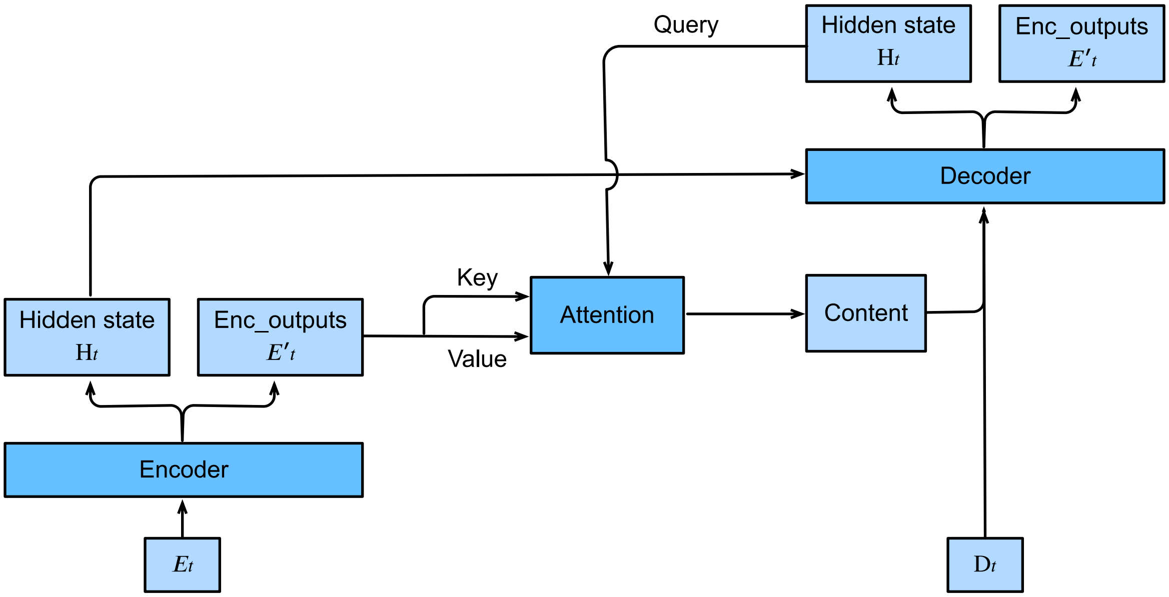


Fig. 10.2.1: The second timestep in decoding for the sequence to sequence model with attention mechanism.

To illustrate the overall architecture of seq2seq with attention model, the layer structure of its encoder and decoder is shown in Fig. 10.2.2.

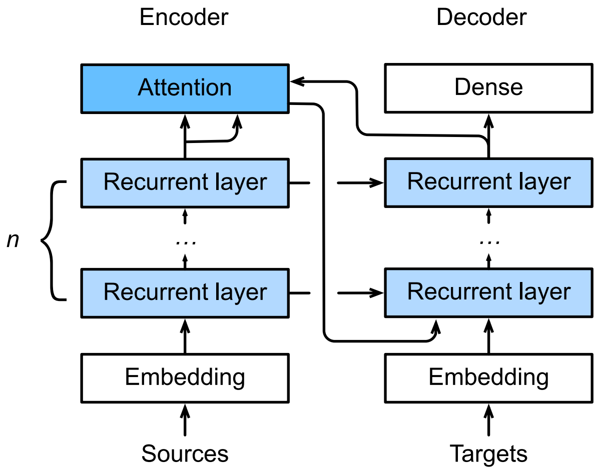


Fig. 10.2.2: The layers in the sequence to sequence model with attention mechanism.

```
import d2l
from mxnet import np, npx
from mxnet.gluon import rnn, nn
npx.set_np()
```

10.2.1 Decoder

Since the encoder of seq2seq with attention mechanisms is the same as Seq2SeqEncoder in Section 9.7, we will just focus on the decoder. We add an MLP attention layer (MLPAttention) which has the same hidden size as the LSTM layer in the decoder. Then we initialize the state of the decoder by passing three items from the encoder:

- **the encoder outputs of all timesteps**: they are used as the attention layer's memory with identical keys and values;
- **the hidden state of the encoder's final timestep**: it is used as the initial decoder's hidden state;
- **the encoder valid length**: so the attention layer will not consider the padding tokens with in the encoder outputs.

At each timestep of the decoding, we use the output of the decoder's last RNN layer as the query for the attention layer. The attention model's output is then concatenated with the input embedding vector to feed into the RNN layer. Although the RNN layer hidden state also contains history information from decoder, the attention output explicitly selects the encoder outputs based on enc_valid_len, so that the attention output suspends other irrelevant information.

Let's implement the Seq2SeqAttentionDecoder, and see how it differs from the decoder in seq2seq from Section 9.7.2.

```
class Seq2SeqAttentionDecoder(d2l.Decoder):
    def __init__(self, vocab_size, embed_size, num_hiddens, num_layers,
                 dropout=0, **kwargs):
        super(Seq2SeqAttentionDecoder, self).__init__(**kwargs)
        self.attention_cell = d2l.MLPAttention(num_hiddens, dropout)
        self.embedding = nn.Embedding(vocab_size, embed_size)
        self.rnn = rnn.LSTM(num_hiddens, num_layers, dropout=dropout)
        self.dense = nn.Dense(vocab_size, flatten=False)

    def init_state(self, enc_outputs, enc_valid_len, *args):
        outputs, hidden_state = enc_outputs
        # Transpose outputs to (batch_size, seq_len, hidden_size)
        return (outputs.swapaxes(0, 1), hidden_state, enc_valid_len)

    def forward(self, X, state):
        enc_outputs, hidden_state, enc_valid_len = state
        X = self.embedding(X).swapaxes(0, 1)
        outputs = []
        for x in X:
            # query shape: (batch_size, 1, hidden_size)
            query = np.expand_dims(hidden_state[0][-1], axis=1)
            # context has same shape as query
            context = self.attention_cell(
                query, enc_outputs, enc_outputs, enc_valid_len)
            # Concatenate on the feature dimension
            x = np.concatenate((context, np.expand_dims(x, axis=1)), axis=-1)
            # Reshape x to (1, batch_size, embed_size+hidden_size)
            out, hidden_state = self.rnn(x.swapaxes(0, 1), hidden_state)
            outputs.append(out)
        outputs = self.dense(np.concatenate(outputs, axis=0))
        return outputs.swapaxes(0, 1), [enc_outputs, hidden_state,
                                        enc_valid_len]
```

Now we can test the seq2seq with attention model. To be consistent with the model without attention in Section 9.7, we use the same hyper-parameters for vocab_size, embed_size, num_hiddens, and num_layers. As a result, we get the same decoder output shape, but the state structure is changed.

```
encoder = d2l.Seq2SeqEncoder(vocab_size=10, embed_size=8,
                             num_hiddens=16, num_layers=2)
encoder.initialize()
decoder = Seq2SeqAttentionDecoder(vocab_size=10, embed_size=8,
                                  num_hiddens=16, num_layers=2)
decoder.initialize()
X = np.zeros((4, 7))
state = decoder.init_state(encoder(X), None)
out, state = decoder(X, state)
out.shape, len(state), state[0].shape, len(state[1]), state[1][0].shape
```

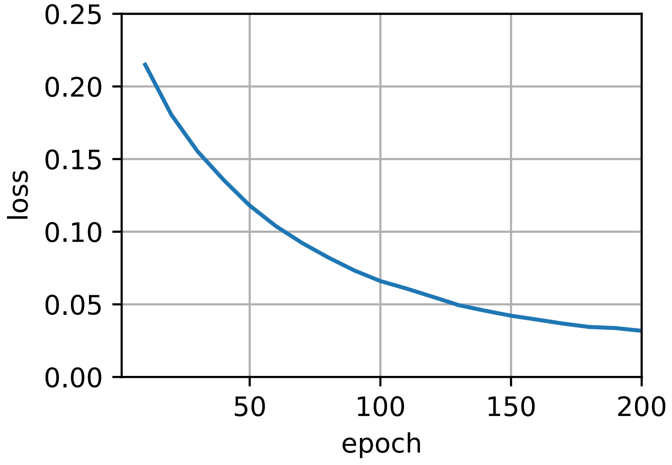
```
((4, 7, 10), 3, (4, 7, 16), 2, (2, 4, 16))
```

10.2.2 Training

Similar to Section 9.7.4, we try a toy model by applying the same training hyperparameters and the same training loss. As we can see from the result, since the sequences in the training dataset are relative short, the additional attention layer does not lead to a significant improvement. Due to the computational overhead of both the encoder's and the decoder's attention layers, this model is much slower than the seq2seq model without attention.

```
embed_size, num_hiddens, num_layers, dropout = 32, 32, 2, 0.0
batch_size, num_steps = 64, 10
lr, num_epochs, ctx = 0.005, 200, d2l.try_gpu()

src_vocab, tgt_vocab, train_iter = d2l.load_data_nmt(batch_size, num_steps)
encoder = d2l.Seq2SeqEncoder(
    len(src_vocab), embed_size, num_hiddens, num_layers, dropout)
decoder = Seq2SeqAttentionDecoder(
    len(tgt_vocab), embed_size, num_hiddens, num_layers, dropout)
model = d2l.EncoderDecoder(encoder, decoder)
d2l.train_s2s_ch9(model, train_iter, lr, num_epochs, ctx)
```

```
loss 0.032, 3714 tokens/sec on gpu(0)
```



Last, we predict several sample examples.

```
for sentence in ['Go .', 'Wow !', "I'm OK .", 'I won !']:
    print(sentence + ' => ' + d2l.predict_s2s_ch9(
        model, sentence, src_vocab, tgt_vocab, num_steps, ctx))
```

```
Go . => va !
Wow ! => <unk> !
I'm OK . => ça va .
I won ! => il est tard !
```

Summary

- The seq2seq model with attention adds an additional attention layer to the model without attention.
- The decoder of the seq2seq with attention model passes three items from the encoder: the encoder outputs of all timesteps, the hidden state of the encoder's final timestep, and the encoder valid length.

Exercises

1. Compare `Seq2SeqAttentionDecoder` and `Seq2seqDecoder` by using the same parameters and checking their losses.
2. Can you think of any use cases where `Seq2SeqAttentionDecoder` will outperform `Seq2seqDecoder`?

10.3 Transformer

In previous chapters, we have covered major neural network architectures such as convolution neural networks (CNNs) and recurrent neural networks (RNNs). Let's recap their pros and cons:

- **CNNs** are easy to parallelize at a layer but cannot capture the variable-length sequential dependency very well.
- **RNNs** are able to capture the long-range, variable-length sequential information, but suffer from inability to parallelize within a sequence.

To combine the advantages from both CNNs and RNNs, (Vaswani et al., 2017) designed a novel architecture using the attention mechanism. This architecture, which is called as *Transformer*, achieves parallelization by capturing recurrence sequence with attention and at the same time encodes each item's position in the sequence. As a result, Transformer leads to a compatible model with significantly shorter training time.

Similar to the seq2seq model in Section 9.7, Transformer is also based on the encoder-decoder architecture. However, Transformer differs to the former by replacing the recurrent layers in seq2seq with *multi-head attention* layers, incorporating the position-wise information through *position encoding*, and applying *layer normalization*. We compare Transformer and seq2seq side-by-side in Fig. 10.3.1.

Overall, these two models are similar to each other: the source sequence embeddings are fed into n repeated blocks. The outputs of the last block are then used as attention memory for the decoder. The target sequence embeddings are similarly fed into n repeated blocks in the decoder, and the final outputs are obtained by applying a dense layer with vocabulary size to the last block's outputs.

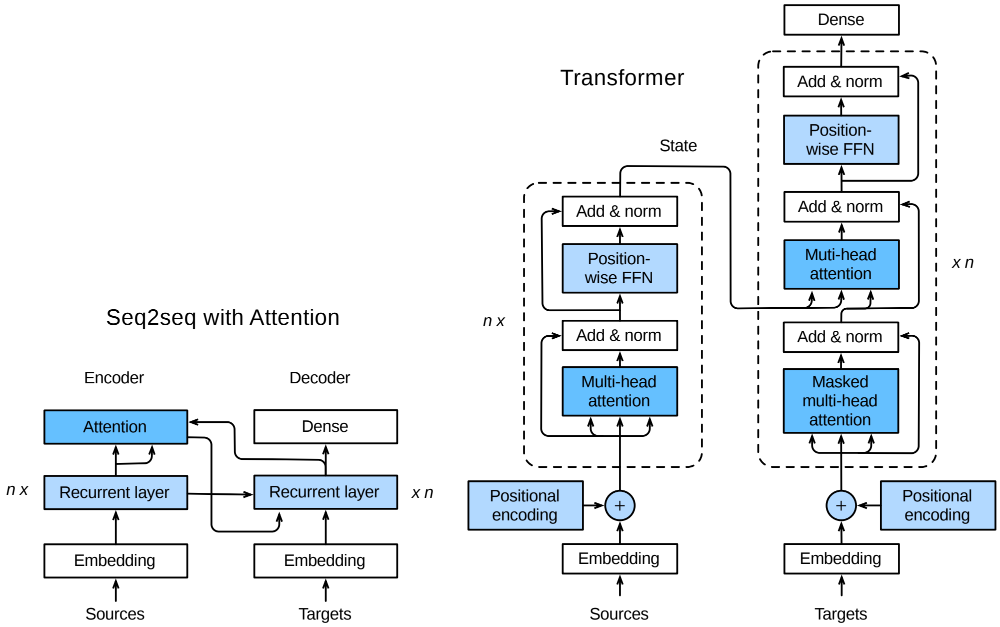


Fig. 10.3.1: The Transformer architecture.

On the flip side, Transformer differs from the seq2seq with attention model in the following:

1. **Transformer block**: a recurrent layer in seq2seq is replaced by a *Transformer block*. This block contains a *multi-head attention* layer and a network with two *position-wise feed-forward*

network layers for the encoder. For the decoder, another multi-head attention layer is used to take the encoder state.

2. **Add and norm**: the inputs and outputs of both the multi-head attention layer or the position-wise feed-forward network, are processed by two "add and norm" layer that contains a residual structure and a *layer normalization* layer.
3. **Position encoding**: since the self-attention layer does not distinguish the item order in a sequence, a positional encoding layer is used to add sequential information into each sequence item.

In the rest of this section, we will equip you with each new component introduced by Transformer, and get you up and running to construct a machine translation model.

```
import d2l
import math
from mxnet import autograd, np, npx
from mxnet.gluon import nn
npx.set_np()
```

10.3.1 Multi-Head Attention

Before the discussion of the *multi-head attention* layer, let's quick express the *self-attention* architecture. The self-attention model is a normal attention model, with its query, its key, and its value being copied exactly the same from each item of the sequential inputs. As we illustrate in Fig. 10.3.2, self-attention outputs a same-length sequential output for each input item. Compared with a recurrent layer, output items of a self-attention layer can be computed in parallel and, therefore, it is easy to obtain a highly-efficient implementation.

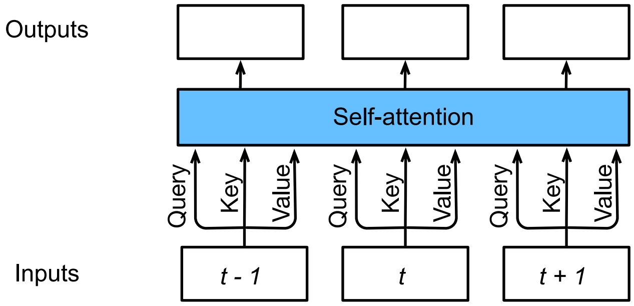


Fig. 10.3.2: Self-attention architecture.

The *multi-head attention* layer consists of h parallel self-attention layers, each one is called a *head*. For each head, before feeding into the attention layer, we project the queries, keys, and values with three dense layers with hidden sizes p_q, p_k, and p_v, respectively. The outputs of these h attention heads are concatenated and then processed by a final dense layer.

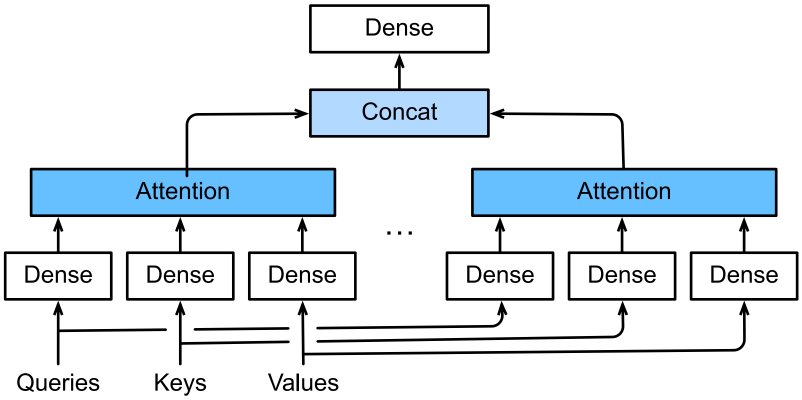


Fig. 10.3.3: Multi-head attention

Assume that the dimension for a query, a key, and a value are d_q, d_k, and d_v, respectively. Then, for each head $i = 1, \ldots, h$, we can train learnable parameters $\mathbf{W}_q^{(i)} \in \mathbb{R}^{p_q\times d_q}$, $\mathbf{W}_k^{(i)} \in \mathbb{R}^{p_k\times d_k}$, and $\mathbf{W}_v^{(i)} \in \mathbb{R}^{p_v\times d_v}$. Therefore, the output for each head is

$$\mathbf{o}^{(i)} = \textrm{attention}(\mathbf{W}_q^{(i)}\mathbf{q}, \mathbf{W}_k^{(i)}\mathbf{k}, \mathbf{W}_v^{(i)}\mathbf{v}), \tag{10.3.1}$$

where attention can be any attention layer, such as the `DotProductAttention` and `MLPAttention` as we introduced in

After that, the output with length p_v from each of the h attention heads are concatenated to be an output of length hp_v, which is then passed the final dense layer with d_o hidden units. The weights of this dense layer can be denoted by $\mathbf{W}_o \in \mathbb{R}^{d_o\times hp_v}$. As a result, the multi-head attention output will be

$$\mathbf{o} = \mathbf{W}_o \begin{bmatrix}\mathbf{o}^{(1)}\\\vdots\\\mathbf{o}^{(h)}\end{bmatrix}. \tag{10.3.2}$$

Now we can implement the multi-head attention. Assume that the multi-head attention contain the number heads `num_heads` $= h$, the hidden size `hidden_size` $= p_q = p_k = p_v$ are the same for the query, key, and value dense layers. In addition, since the multi-head attention keeps the same dimensionality between its input and its output, we have the output feature size $d_o = $ `hidden_size` as well.

```
class MultiHeadAttention(nn.Block):
    def __init__(self, hidden_size, num_heads, dropout, **kwargs):
        super(MultiHeadAttention, self).__init__(**kwargs)
        self.num_heads = num_heads
        self.attention = d2l.DotProductAttention(dropout)
        self.W_q = nn.Dense(hidden_size, use_bias=False, flatten=False)
        self.W_k = nn.Dense(hidden_size, use_bias=False, flatten=False)
        self.W_v = nn.Dense(hidden_size, use_bias=False, flatten=False)
        self.W_o = nn.Dense(hidden_size, use_bias=False, flatten=False)

    def forward(self, query, key, value, valid_length):
        # query, key, and value shape: (batch_size, seq_len, dim),
        # where seq_len is the length of input sequence
        # valid_length shape is either (batch_size, )
```

(continues on next page)

(continued from previous page)

```
        # or (batch_size, seq_len).

        # Project and transpose query, key, and value from
        # (batch_size, seq_len, hidden_size * num_heads) to
        # (batch_size * num_heads, seq_len, hidden_size).
        query = transpose_qkv(self.W_q(query), self.num_heads)
        key = transpose_qkv(self.W_k(key), self.num_heads)
        value = transpose_qkv(self.W_v(value), self.num_heads)

        if valid_length is not None:
            # Copy valid_length by num_heads times
            if valid_length.ndim == 1:
                valid_length = np.tile(valid_length, self.num_heads)
            else:
                valid_length = np.tile(valid_length, (self.num_heads, 1))

        output = self.attention(query, key, value, valid_length)

        # Transpose from (batch_size * num_heads, seq_len, hidden_size) back
        # to (batch_size, seq_len, hidden_size * num_heads)
        output_concat = transpose_output(output, self.num_heads)
        return self.W_o(output_concat)
```

Here are the definitions of the transpose functions `transpose_qkv` and `transpose_output`, who are the inverse of each other.

```
def transpose_qkv(X, num_heads):
    # Original X shape: (batch_size, seq_len, hidden_size * num_heads),
    # -1 means inferring its value, after first reshape, X shape:
    # (batch_size, seq_len, num_heads, hidden_size)
    X = X.reshape(X.shape[0], X.shape[1], num_heads, -1)

    # After transpose, X shape: (batch_size, num_heads, seq_len, hidden_size)
    X = X.transpose(0, 2, 1, 3)

    # Merge the first two dimensions. Use reverse=True to infer shape from
    # right to left.
    # output shape: (batch_size * num_heads, seq_len, hidden_size)
    output = X.reshape(-1, X.shape[2], X.shape[3])
    return output


def transpose_output(X, num_heads):
    # A reversed version of transpose_qkv
    X = X.reshape(-1, num_heads, X.shape[1], X.shape[2])
    X = X.transpose(0, 2, 1, 3)
    return X.reshape(X.shape[0], X.shape[1], -1)
```

Let's test the `MultiHeadAttention` model in the a toy example. Create a multi-head attention with the hidden size $d_o = 100$, the output will share the same batch size and sequence length as the input, but the last dimension will be equal to the `hidden_size` $= 100$.

```
cell = MultiHeadAttention(100, 10, 0.5)
cell.initialize()
```

(continues on next page)

```
X = np.ones((2, 4, 5))
valid_length = np.array([2, 3])
cell(X, X, X, valid_length).shape
```

```
(2, 4, 100)
```

10.3.2 Position-wise Feed-Forward Networks

Another key component in the Transformer block is called *position-wise feed-forward network (FFN)*. It accepts a 3-dimensional input with shape (batch size, sequence length, feature size). The position-wise FFN consists of two dense layers that applies to the last dimension. Since the same two dense layers are used for each position item in the sequence, we referred to it as *position-wise*. Indeed, it is equivalent to applying two 1×1 convolution layers.

Below, the `PositionWiseFFN` shows how to implement a position-wise FFN with two dense layers of hidden size `ffn_hidden_size` and `hidden_size_out`, respectively.

```
class PositionWiseFFN(nn.Block):
    def __init__(self, ffn_hidden_size, hidden_size_out, **kwargs):
        super(PositionWiseFFN, self).__init__(**kwargs)
        self.ffn_1 = nn.Dense(ffn_hidden_size, flatten=False,
                              activation='relu')
        self.ffn_2 = nn.Dense(hidden_size_out, flatten=False)

    def forward(self, X):
        return self.ffn_2(self.ffn_1(X))
```

Similar to the multi-head attention, the position-wise feed-forward network will only change the last dimension size of the input—the feature dimension. In addition, if two items in the input sequence are identical, the according outputs will be identical as well.

```
ffn = PositionWiseFFN(4, 8)
ffn.initialize()
ffn(np.ones((2, 3, 4)))[0]
```

```
array([[-0.00073839,  0.00923239, -0.00016378,  0.00091236, -0.00763499,
         0.00199923,  0.00446541,  0.00189135],
       [-0.00073839,  0.00923239, -0.00016378,  0.00091236, -0.00763499,
         0.00199923,  0.00446541,  0.00189135],
       [-0.00073839,  0.00923239, -0.00016378,  0.00091236, -0.00763499,
         0.00199923,  0.00446541,  0.00189135]])
```

10.3.3 Add and Norm

Besides the above two components in the Transformer block, the "add and norm" within the block also plays a key role to connect the inputs and outputs of other layers smoothly. To explain, we add a layer that contains a residual structure and a *layer normalization* after both the multi-head attention layer and the position-wise FFN network. *Layer normalization* is similar to batch normalization in Section 7.5. One difference is that the mean and variances for the layer normalization are calculated along the last dimension, e.g `X.mean(axis=-1)` instead of the first batch dimension, e.g., `X.mean(axis=0)`. Layer normalization prevents the range of values in the layers from changing too much, which means that faster training and better generalization ability.

MXNet has both `LayerNorm` and `BatchNorm` implemented within the `nn` block. Let's call both of them and see the difference in the example below.

```
layer = nn.LayerNorm()
layer.initialize()
batch = nn.BatchNorm()
batch.initialize()
X = np.array([[1, 2], [2, 3]])
# Compute mean and variance from X in the training mode
with autograd.record():
    print('layer norm:', layer(X), '\nbatch norm:', batch(X))
```

```
layer norm: [[-0.99998  0.99998]
 [-0.99998  0.99998]]
batch norm: [[-0.99998 -0.99998]
 [ 0.99998  0.99998]]
```

Now let's implement the connection block `AddNorm` together. `AddNorm` accepts two inputs X and Y. We can deem X as the original input in the residual network, and Y as the outputs from either the multi-head attention layer or the position-wise FFN network. In addition, we apply dropout on Y for regularization.

```
class AddNorm(nn.Block):
    def __init__(self, dropout, **kwargs):
        super(AddNorm, self).__init__(**kwargs)
        self.dropout = nn.Dropout(dropout)
        self.norm = nn.LayerNorm()

    def forward(self, X, Y):
        return self.norm(self.dropout(Y) + X)
```

Due to the residual connection, X and Y should have the same shape.

```
add_norm = AddNorm(0.5)
add_norm.initialize()
add_norm(np.ones((2, 3, 4)), np.ones((2, 3, 4))).shape
```

```
(2, 3, 4)
```

10.3.4 Positional Encoding

Unlike the recurrent layer, both the multi-head attention layer and the position-wise feed-forward network compute the output of each item in the sequence independently. This feature enables us to parallelize the computation, but it fails to model the sequential information for a given sequence. To better capture the sequential information, the Transformer model uses the *positional encoding* to maintain the positional information of the input sequence.

To explain, assume that $X \in \mathbb{R}^{l \times d}$ is the embedding of an example, where l is the sequence length and d is the embedding size. This positional encoding layer encodes X's position $P \in \mathbb{R}^{l \times d}$ and outputs $P + X$.

The position P is a 2-D matrix, where i refers to the order in the sentence, and j refers to the position along the embedding vector dimension. In this way, each value in the origin sequence is then maintained using the equations below:

$$P_{i,2j} = \sin(i/10000^{2j/d}), \tag{10.3.3}$$

$$P_{i,2j+1} = \cos(i/10000^{2j/d}), \tag{10.3.4}$$

for $i = 0, \ldots, l-1$ and $j = 0, \ldots, \lfloor (d-1)/2 \rfloor$.

Fig. 10.3.4 illustrates the positional encoding.

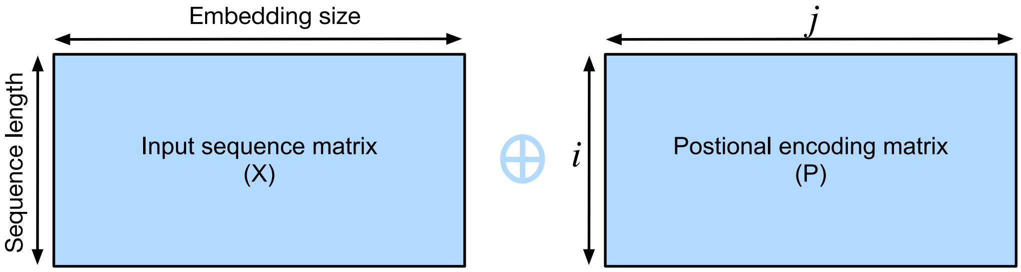
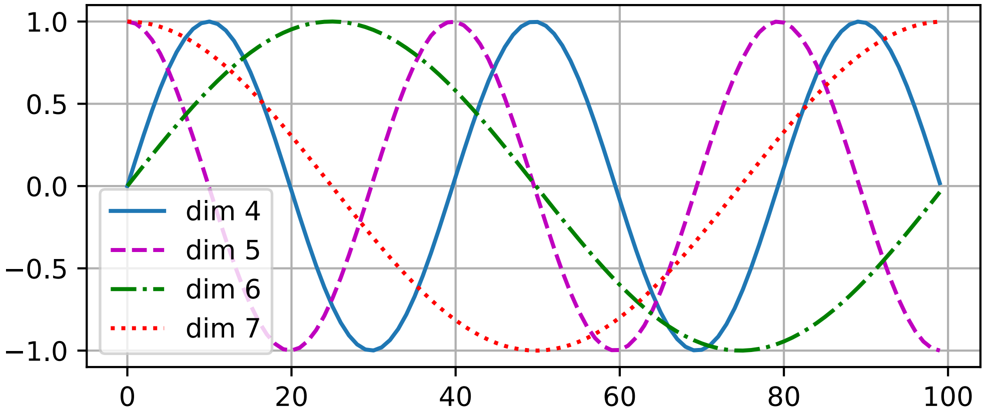


Fig. 10.3.4: Positional encoding.

```
class PositionalEncoding(nn.Block):
    def __init__(self, embedding_size, dropout, max_len=1000):
        super(PositionalEncoding, self).__init__()
        self.dropout = nn.Dropout(dropout)
        # Create a long enough P
        self.P = np.zeros((1, max_len, embedding_size))
        X = np.arange(0, max_len).reshape(-1, 1) / np.power(
            10000, np.arange(0, embedding_size, 2)/embedding_size)
        self.P[:, :, 0::2] = np.sin(X)
        self.P[:, :, 1::2] = np.cos(X)

    def forward(self, X):
        X = X + self.P[:, :X.shape[1], :].as_in_context(X.context)
        return self.dropout(X)
```

Now we test the `PositionalEncoding` class with a toy model for 4 dimensions. As we can see, the 4^{th} dimension has the same frequency as the 5^{th} but with different offset. The 5^{th} and 6^{th} dimensions have a lower frequency.

```
pe = PositionalEncoding(20, 0)
pe.initialize()
Y = pe(np.zeros((1, 100, 20)))
d2l.plot(np.arange(100), Y[0, :, 4:8].T, figsize=(6, 2.5),
         legend=["dim %d" % p for p in [4, 5, 6, 7]])
```



10.3.5 Encoder

Armed with all the essential components of Transformer, let's first build a Transformer encoder block. This encoder contains a multi-head attention layer, a position-wise feed-forward network, and two "add and norm" connection blocks. As shown in the code, for both of the attention model and the positional FFN model in the `EncoderBlock`, their outputs' dimension are equal to the `embedding_size`. This is due to the nature of the residual block, as we need to add these outputs back to the original value during "add and norm".

```
class EncoderBlock(nn.Block):
    def __init__(self, embedding_size, ffn_hidden_size, num_heads,
                 dropout, **kwargs):
        super(EncoderBlock, self).__init__(**kwargs)
        self.attention = MultiHeadAttention(embedding_size, num_heads,
                                            dropout)
        self.addnorm_1 = AddNorm(dropout)
        self.ffn = PositionWiseFFN(ffn_hidden_size, embedding_size)
        self.addnorm_2 = AddNorm(dropout)

    def forward(self, X, valid_length):
        Y = self.addnorm_1(X, self.attention(X, X, X, valid_length))
        return self.addnorm_2(Y, self.ffn(Y))
```

Due to the residual connections, this block will not change the input shape. It means that the `embedding_size` argument should be equal to the input size of the last dimension. In our toy example below, `embedding_size` $= 24$, `ffn_hidden_size` $= 48$, `num_heads` $= 8$, and `dropout` $= 0.5$.

```
X = np.ones((2, 100, 24))
encoder_blk = EncoderBlock(24, 48, 8, 0.5)
encoder_blk.initialize()
encoder_blk(X, valid_length).shape
```

```
(2, 100, 24)
```

Now it comes to the implementation of the entire Transformer encoder. With the Transformer encoder, n blocks of `EncoderBlock` stack up one after another. Because of the residual connection, the embedding layer size d is same as the Transformer block output size. Also note that we multiply the embedding output by $\sqrt{d}$ to prevent its values from being too small.

```
class TransformerEncoder(d2l.Encoder):
    def __init__(self, vocab_size, embedding_size, ffn_hidden_size,
                 num_heads, num_layers, dropout, **kwargs):
        super(TransformerEncoder, self).__init__(**kwargs)
        self.embedding_size = embedding_size
        self.embed = nn.Embedding(vocab_size, embedding_size)
        self.pos_encoding = PositionalEncoding(embedding_size, dropout)
        self.blks = nn.Sequential()
        for i in range(num_layers):
            self.blks.add(
                EncoderBlock(embedding_size, ffn_hidden_size,
                             num_heads, dropout))

    def forward(self, X, valid_length, *args):
        X = self.pos_encoding(self.embed(X) * math.sqrt(self.embedding_size))
        for blk in self.blks:
            X = blk(X, valid_length)
        return X
```

Let's create an encoder with two stacked Transformer encoder blocks, whose hyperparameters are the same as before. Similar to the previous toy example's parameters, we add two more parameters `vocab_size` to be 200 and `num_layers` to be 2 here.

```
encoder = TransformerEncoder(200, 24, 48, 8, 2, 0.5)
encoder.initialize()
encoder(np.ones((2, 100)), valid_length).shape
```

```
(2, 100, 24)
```

10.3.6 Decoder

The Transformer decoder block looks similar to the Transformer encoder block. However, besides the two sub-layers—the multi-head attention layer and the positional encoding network, the decoder Transformer block contains a third sub-layer, which applies multi-head attention on the output of the encoder stack. Similar to the Transformer encoder block, the Transformer decoder block employs "add and norm", i.e., the residual connections and the layer normalization to connect each of the sub-layers.

To be specific, at timestep t, assume that $\mathbf{x}_t$ is the current input, i.e., the query. As illustrated in Fig. 10.3.5, the keys and values of the self-attention layer consist of the current query with all the past queries $\mathbf{x}_1, \ldots, \mathbf{x}_{t-1}$.

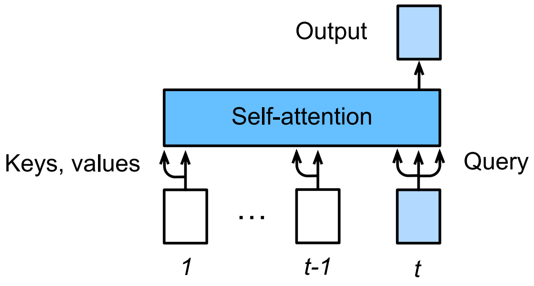


Fig. 10.3.5: Predict at timestep t for a self-attention layer.

During training, the output for the t-query could observe all the previous key-value pairs. It results in an different behavior from prediction. Thus, during prediction we can eliminate the unnecessary information by specifying the valid length to be t for the t^{th} query.

```python
class DecoderBlock(nn.Block):
    # i means it is the i-th block in the decoder
    def __init__(self, embedding_size, ffn_hidden_size, num_heads,
                 dropout, i, **kwargs):
        super(DecoderBlock, self).__init__(**kwargs)
        self.i = i
        self.attention_1 = MultiHeadAttention(embedding_size, num_heads,
                                              dropout)
        self.addnorm_1 = AddNorm(dropout)
        self.attention_2 = MultiHeadAttention(embedding_size, num_heads,
                                              dropout)
        self.addnorm_2 = AddNorm(dropout)
        self.ffn = PositionWiseFFN(ffn_hidden_size, embedding_size)
        self.addnorm_3 = AddNorm(dropout)

    def forward(self, X, state):
        enc_outputs, enc_valid_lengh = state[0], state[1]
        # state[2][i] contains the past queries for this block
        if state[2][self.i] is None:
            key_values = X
        else:
            key_values = np.concatenate((state[2][self.i], X), axis=1)
        state[2][self.i] = key_values
        if autograd.is_training():
            batch_size, seq_len, _ = X.shape
            # Shape: (batch_size, seq_len), the values in the j-th column
            # are j+1
            valid_length = np.tile(np.arange(1, seq_len+1, ctx=X.context),
                                   (batch_size, 1))
        else:
            valid_length = None

        X2 = self.attention_1(X, key_values, key_values, valid_length)
        Y = self.addnorm_1(X, X2)
        Y2 = self.attention_2(Y, enc_outputs, enc_outputs, enc_valid_lengh)
        Z = self.addnorm_2(Y, Y2)
        return self.addnorm_3(Z, self.ffn(Z)), state
```

Similar to the Transformer encoder block, `embedding_size` should be equal to the last dimension size of X.

```
decoder_blk = DecoderBlock(24, 48, 8, 0.5, 0)
decoder_blk.initialize()
X = np.ones((2, 100, 24))
state = [encoder_blk(X, valid_length), valid_length, [None]]
decoder_blk(X, state)[0].shape
```

```
(2, 100, 24)
```

The construction of the entire Transformer decoder is identical to the Transformer encoder, except for the additional dense layer to obtain the output confidence scores.

Let's implement the Transformer decoder `TransformerDecoder`. Besides the regular hyperparameters such as the `vocab_size` and `embedding_size`, the Transformer decoder also needs the encoder Transformer's outputs `enc_outputs` and `env_valid_lengh`.

```
class TransformerDecoder(d2l.Decoder):
    def __init__(self, vocab_size, embedding_size, ffn_hidden_size,
                 num_heads, num_layers, dropout, **kwargs):
        super(TransformerDecoder, self).__init__(**kwargs)
        self.embedding_size = embedding_size
        self.num_layers = num_layers
        self.embed = nn.Embedding(vocab_size, embedding_size)
        self.pos_encoding = PositionalEncoding(embedding_size, dropout)
        self.blks = nn.Sequential()
        for i in range(num_layers):
            self.blks.add(
                DecoderBlock(embedding_size, ffn_hidden_size, num_heads,
                             dropout, i))
        self.dense = nn.Dense(vocab_size, flatten=False)

    def init_state(self, enc_outputs, env_valid_lengh, *args):
        return [enc_outputs, env_valid_lengh, [None]*self.num_layers]

    def forward(self, X, state):
        X = self.pos_encoding(self.embed(X) * math.sqrt(self.embedding_size))
        for blk in self.blks:
            X, state = blk(X, state)
        return self.dense(X), state
```

10.3.7 Training

Finally, we can build a encoder-decoder model with Transformer architecture. Similar to the seq2seq with attention model in Section 10.2, we use the following hyperparameters: two Transformer blocks with both the embedding size and the block output size to be 32. In addition, we use 4 heads, and set the hidden size to be twice larger than the output size.

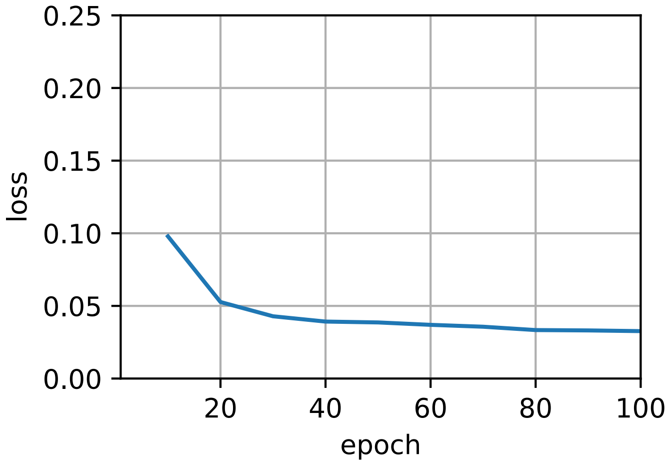
```
embed_size, embedding_size, num_layers, dropout = 32, 32, 2, 0.0
batch_size, num_steps = 64, 10
lr, num_epochs, ctx = 0.005, 100, d2l.try_gpu()
num_hiddens, num_heads = 64, 4

src_vocab, tgt_vocab, train_iter = d2l.load_data_nmt(batch_size, num_steps)
```

(continues on next page)

(continued from previous page)

```
encoder = TransformerEncoder(
    len(src_vocab), embedding_size, num_hiddens, num_heads, num_layers,
    dropout)
decoder = TransformerDecoder(
    len(src_vocab), embedding_size, num_hiddens, num_heads, num_layers,
    dropout)
model = d2l.EncoderDecoder(encoder, decoder)
d2l.train_s2s_ch9(model, train_iter, lr, num_epochs, ctx)
```

```
loss 0.033, 3453 tokens/sec on gpu(0)
```



As we can see from the training time and accuracy, compared with the seq2seq model with attention model, Transformer runs faster per epoch, and converges faster at the beginning.

We can use the trained Transformer to translate some simple sentences.

```
for sentence in ['Go .', 'Wow !', "I'm OK .", 'I won !']:
    print(sentence + ' => ' + d2l.predict_s2s_ch9(
        model, sentence, src_vocab, tgt_vocab, num_steps, ctx))
```

```
Go . => va !
Wow ! => <unk> !
I'm OK . => je vais bien .
I won ! => j'ai gagné !
```

Summary

- The Transformer model is based on the encoder-decoder architecture.
- Multi-head attention layer contains h parallel attention layers.
- Position-wise feed-forward network consists of two dense layers that apply to the last dimension.
- Layer normalization differs from batch normalization by normalizing along the last dimension (the feature dimension) instead of the first (batch size) dimension.
- Positional encoding is the only place that adds positional information to the Transformer model.

Exercises

1. Try a larger size of epochs and compare the loss between seq2seq model and Transformer model.
2. Can you think of any other benefit of positional encoding?
3. Compare layer normalization and batch normalization, when shall we apply which?

11 | Optimization Algorithms

If you read the book in sequence up to this point you already used a number of advanced optimization algorithms to train deep learning models. They were the tools that allowed us to continue updating model parameters and to minimize the value of the loss function, as evaluated on the training set. Indeed, anyone content with treating optimization as a black box device to minimize objective functions in a simple setting might well content oneself with the knowledge that there exists an array of incantations of such a procedure (with names such as "Adam", "NAG", or "SGD").

To do well, however, some deeper knowledge is required. Optimization algorithms are important for deep learning. On one hand, training a complex deep learning model can take hours, days, or even weeks. The performance of the optimization algorithm directly affects the model's training efficiency. On the other hand, understanding the principles of different optimization algorithms and the role of their parameters will enable us to tune the hyperparameters in a targeted manner to improve the performance of deep learning models.

In this chapter, we explore common deep learning optimization algorithms in depth. Almost all optimization problems arising in deep learning are *nonconvex*. Nonetheless, the design and analysis of algorithms in the context of convex problems has proven to be very instructive. It is for that reason that this section includes a primer on convex optimization and the proof for a very simple stochastic gradient descent algorithm on a convex objective function.

11.1 Optimization and Deep Learning

In this section, we will discuss the relationship between optimization and deep learning as well as the challenges of using optimization in deep learning. For a deep learning problem, we will usually define a loss function first. Once we have the loss function, we can use an optimization algorithm in attempt to minimize the loss. In optimization, a loss function is often referred to as the objective function of the optimization problem. By tradition and convention most optimization algorithms are concerned with *minimization*. If we ever need to maximize an objective there is a simple solution: just flip the sign on the objective.

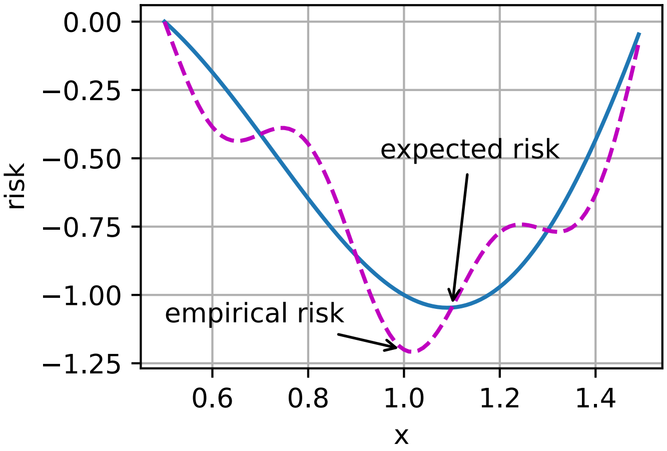
11.1.1 Optimization and Estimation

Although optimization provides a way to minimize the loss function for deep learning, in essence, the goals of optimization and deep learning are fundamentally different. The former is primarily concerned with minimizing an objective whereas the latter is concerned with finding a suitable model, given a finite amount of data. In Section 4.4, we discussed the difference between these two goals in detail. For instance, training error and generalization error generally differ: since the objective function of the optimization algorithm is usually a loss function based on the training dataset, the goal of optimization is to reduce the training error. However, the goal of statistical inference (and thus of deep learning) is to reduce the generalization error. To accomplish the latter we need to pay attention to overfitting in addition to using the optimization algorithm to reduce the training error. We begin by importing a few libraries with a function to annotate in a figure.

```
%matplotlib inline
import d2l
from mpl_toolkits import mplot3d
from mxnet import np, npx
npx.set_np()

# Saved in the d2l package for later use
def annotate(text, xy, xytext):
    d2l.plt.gca().annotate(text, xy=xy, xytext=xytext,
                           arrowprops=dict(arrowstyle='->'))
```

The graph below illustrates the issue in some more detail. Since we have only a finite amount of data the minimum of the training error may be at a different location than the minimum of the expected error (or of the test error).

```
def f(x): return x * np.cos(np.pi * x)
def g(x): return f(x) + 0.2 * np.cos(5 * np.pi * x)

d2l.set_figsize((4.5, 2.5))
x = np.arange(0.5, 1.5, 0.01)
d2l.plot(x, [f(x), g(x)], 'x', 'risk')
annotate('empirical risk', (1.0, -1.2), (0.5, -1.1))
annotate('expected risk', (1.1, -1.05), (0.95, -0.5))
```



11.1.2 Optimization Challenges in Deep Learning

In this chapter, we are going to focus specifically on the performance of the optimization algorithm in minimizing the objective function, rather than a model's generalization error. In Section 3.1 we distinguished between analytical solutions and numerical solutions in optimization problems. In deep learning, most objective functions are complicated and do not have analytical solutions. Instead, we must use numerical optimization algorithms. The optimization algorithms below all fall into this category.

There are many challenges in deep learning optimization. Some of the most vexing ones are local minima, saddle points and vanishing gradients. Let's have a look at a few of them.

Local Minima

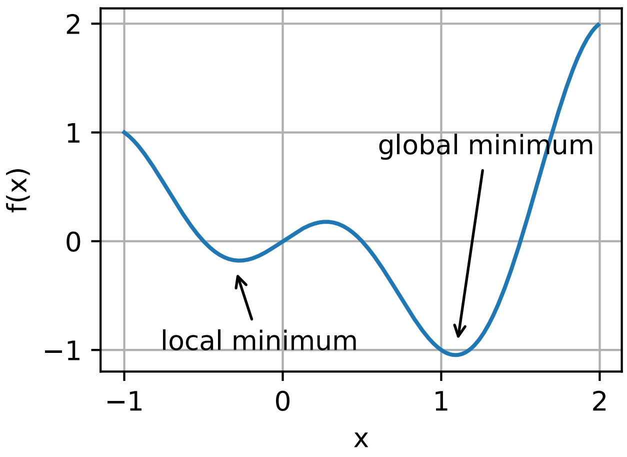
For the objective function $f(x)$, if the value of $f(x)$ at x is smaller than the values of $f(x)$ at any other points in the vicinity of x, then $f(x)$ could be a local minimum. If the value of $f(x)$ at x is the minimum of the objective function over the entire domain, then $f(x)$ is the global minimum.

For example, given the function

$$f(x) = x \cdot \text{cos}(\pi x) \text{ for } -1.0 \leq x \leq 2.0, \tag{11.1.1}$$

we can approximate the local minimum and global minimum of this function.

```
x = np.arange(-1.0, 2.0, 0.01)
d2l.plot(x, [f(x), ], 'x', 'f(x)')
annotate('local minimum', (-0.3, -0.25), (-0.77, -1.0))
annotate('global minimum', (1.1, -0.95), (0.6, 0.8))
```

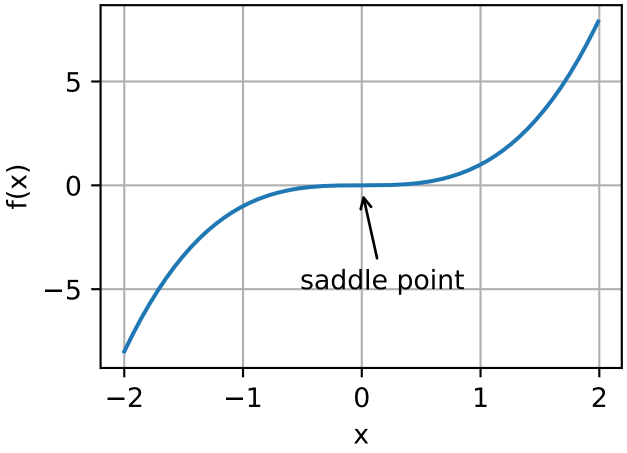


The objective function of deep learning models usually has many local optima. When the numerical solution of an optimization problem is near the local optimum, the numerical solution obtained by the final iteration may only minimize the objective function locally, rather than globally, as the gradient of the objective function's solutions approaches or becomes zero. Only some degree of noise might knock the parameter out of the local minimum. In fact, this is one of the beneficial properties of stochastic gradient descent where the natural variation of gradients over minibatches is able to dislodge the parameters from local minima.

Saddle Points

Besides local minima, saddle points are another reason for gradients to vanish. A saddle point[147] is any location where all gradients of a function vanish but which is neither a global nor a local minimum. Consider the function $f(x) = x^3$. Its first and second derivative vanish for $x = 0$. Optimization might stall at the point, even though it is not a minimum.

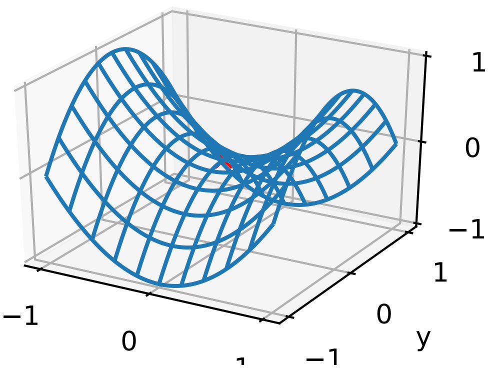
```
x = np.arange(-2.0, 2.0, 0.01)
d2l.plot(x, [x**3], 'x', 'f(x)')
annotate('saddle point', (0, -0.2), (-0.52, -5.0))
```



Saddle points in higher dimensions are even more insidious, as the example below shows. Consider the function $f(x, y) = x^2 - y^2$. It has its saddle point at $(0, 0)$. This is a maximum with respect to y and a minimum with respect to x. Moreover, it *looks* like a saddle, which is where this mathematical property got its name.

```
x, y = np.meshgrid(np.linspace(-1, 1, 101), np.linspace(-1, 1, 101),
                   indexing='ij')

z = x**2 - y**2

ax = d2l.plt.figure().add_subplot(111, projection='3d')
ax.plot_wireframe(x, y, z, **{'rstride': 10, 'cstride': 10})
ax.plot([0], [0], [0], 'rx')
ticks = [-1, 0, 1]
d2l.plt.xticks(ticks)
d2l.plt.yticks(ticks)
ax.set_zticks(ticks)
d2l.plt.xlabel('x')
d2l.plt.ylabel('y');
```

[147] https://en.wikipedia.org/wiki/Saddle_point



We assume that the input of a function is a k-dimensional vector and its output is a scalar, so its Hessian matrix will have k eigenvalues (refer to Section 17.1). The solution of the function could be a local minimum, a local maximum, or a saddle point at a position where the function gradient is zero:

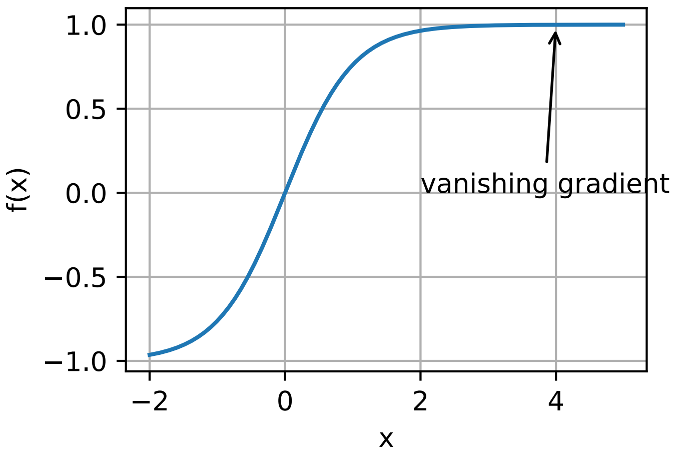
- When the eigenvalues of the function's Hessian matrix at the zero-gradient position are all positive, we have a local minimum for the function.
- When the eigenvalues of the function's Hessian matrix at the zero-gradient position are all negative, we have a local maximum for the function.
- When the eigenvalues of the function's Hessian matrix at the zero-gradient position are negative and positive, we have a saddle point for the function.

For high-dimensional problems the likelihood that at least some of the eigenvalues are negative is quite high. This makes saddle points more likely than local minima. We will discuss some exceptions to this situation in the next section when introducing convexity. In short, convex functions are those where the eigenvalues of the Hessian are never negative. Sadly, though, most deep learning problems do not fall into this category. Nonetheless it is a great tool to study optimization algorithms.

Vanishing Gradients

Probably the most insidious problem to encounter are vanishing gradients. For instance, assume that we want to minimize the function $f(x) = \tanh(x)$ and we happen to get started at $x = 4$. As we can see, the gradient of f is close to nil. More specifically $f'(x) = 1 - \tanh^2(x)$ and thus $f'(4) = 0.0013$. Consequently optimization will get stuck for a long time before we make progress. This turns out to be one of the reasons that training deep learning models was quite tricky prior to the introduction of the ReLU activation function.

```
x = np.arange(-2.0, 5.0, 0.01)
d2l.plot(x, [np.tanh(x)], 'x', 'f(x)')
annotate('vanishing gradient', (4, 1), (2, 0.0))
```



As we saw, optimization for deep learning is full of challenges. Fortunately there exists a robust range of algorithms that perform well and that are easy to use even for beginners. Furthermore, it is not really necessary to find *the* best solution. Local optima or even approximate solutions thereof are still very useful.

Summary

- Minimizing the training error does *not* guarantee that we find the best set of parameters to minimize the expected error.
- The optimization problems may have many local minima.
- The problem may have even more saddle points, as generally the problems are not convex.
- Vanishing gradients can cause optimization to stall. Often a reparametrization of the problem helps. Good initialization of the parameters can be beneficial, too.

Exercises

1. Consider a simple multilayer perceptron with a single hidden layer of, say, d dimensions in the hidden layer and a single output. Show that for any local minimum there are at least $d!$ equivalent solutions that behave identically.
2. Assume that we have a symmetric random matrix $\mathbf{M}$ where the entries $M_{ij} = M_{ji}$ are each drawn from some probability distribution p_{ij}. Furthermore assume that $p_{ij}(x) = p_{ij}(-x)$, i.e., that the distribution is symmetric (see e.g., (Wigner, 1958) for details).
   - Prove that the distribution over eigenvalues is also symmetric. That is, for any eigenvector $\mathbf{v}$ the probability that the associated eigenvalue λ satisfies $P(\lambda > 0) = P(\lambda < 0)$.
   - Why does the above *not* imply $P(\lambda > 0) = 0.5$?
3. What other challenges involved in deep learning optimization can you think of?
4. Assume that you want to balance a (real) ball on a (real) saddle.
   - Why is this hard?
   - Can you exploit this effect also for optimization algorithms?

11.2 Convexity

Convexity plays a vital role in the design of optimization algorithms. This is largely due to the fact that it is much easier to analyze and test algorithms in this context. In other words, if the algorithm performs poorly even in the convex setting we should not hope to see great results otherwise. Furthermore, even though the optimization problems in deep learning are generally nonconvex, they often exhibit some properties of convex ones near local minima. This can lead to exciting new optimization variants such as (Izmailov et al., 2018).

11.2.1 Basics

Let's begin with the basics.

Sets

Sets are the basis of convexity. Simply put, a set X in a vector space is convex if for any $a, b \in X$ the line segment connecting a and b is also in X. In mathematical terms this means that for all $\lambda \in [0, 1]$ we have

$$\lambda \cdot a + (1 - \lambda) \cdot b \in X \text{ whenever } a, b \in X. \tag{11.2.1}$$

This sounds a bit abstract. Consider the picture Fig. 11.2.1. The first set is not convex since there are line segments that are not contained in it. The other two sets suffer no such problem.

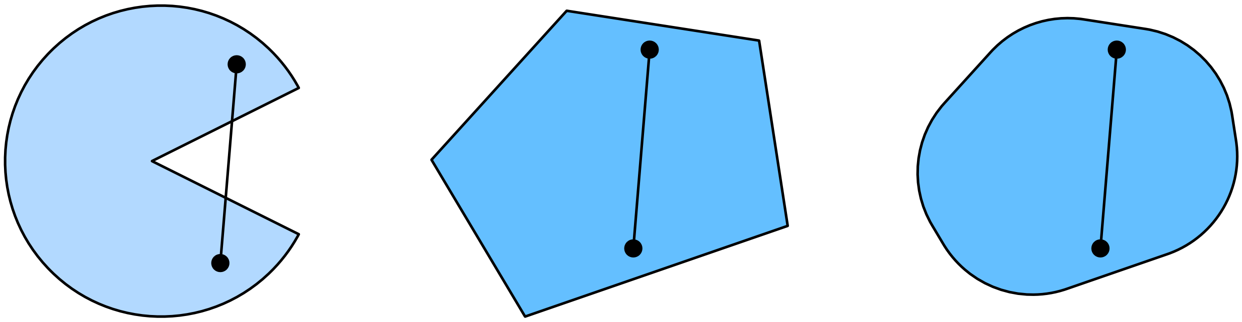


Fig. 11.2.1: Three shapes, the left one is nonconvex, the others are convex

Definitions on their own are not particularly useful unless you can do something with them. In this case we can look at unions and intersections as shown in Fig. 11.2.2. Assume that X and Y are convex sets. Then $X \cap Y$ is also convex. To see this, consider any $a, b \in X \cap Y$. Since X and Y are convex, the line segments connecting a and b are contained in both X and Y. Given that, they also need to be contained in $X \cap Y$, thus proving our first theorem.

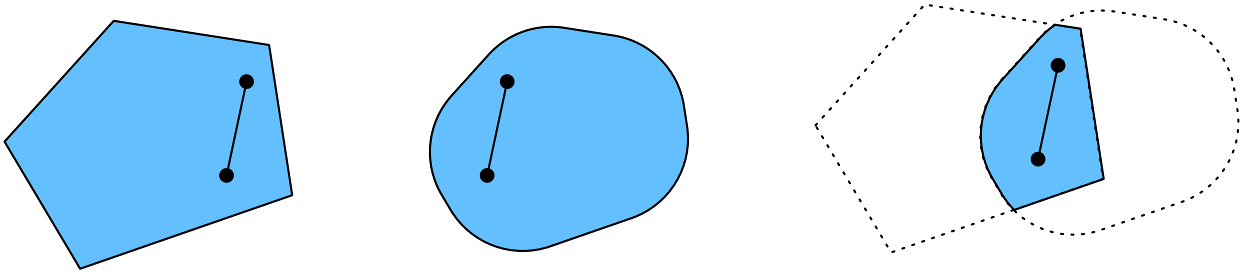


Fig. 11.2.2: The intersection between two convex sets is convex

We can strengthen this result with little effort: given convex sets X_i, their intersection $\cap_i X_i$ is convex. To see that the converse is not true, consider two disjoint sets $X \cap Y = \emptyset$. Now pick $a \in X$ and $b \in Y$. The line segment in Fig. 11.2.3 connecting a and b needs to contain some part that is neither in X nor Y, since we assumed that $X \cap Y = \emptyset$. Hence the line segment is not in $X \cup Y$ either, thus proving that in general unions of convex sets need not be convex.

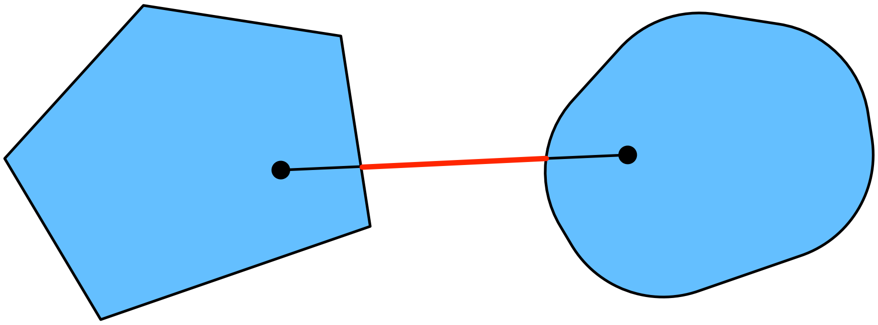


Fig. 11.2.3: The union of two convex sets need not be convex

Typically the problems in deep learning are defined on convex domains. For instance $\mathbb{R}^d$ is a convex set (after all, the line between any two points in $\mathbb{R}^d$ remains in $\mathbb{R}^d$). In some cases we work with variables of bounded length, such as balls of radius r as defined by $\{\mathbf{x} | \mathbf{x} \in \mathbb{R}^d \text{ and } \|\mathbf{x}\|_2 \leq r\}$.

Functions

Now that we have convex sets we can introduce convex functions f. Given a convex set X a function defined on it $f : X \to \mathbb{R}$ is convex if for all $x, x' \in X$ and for all $\lambda \in [0, 1]$ we have

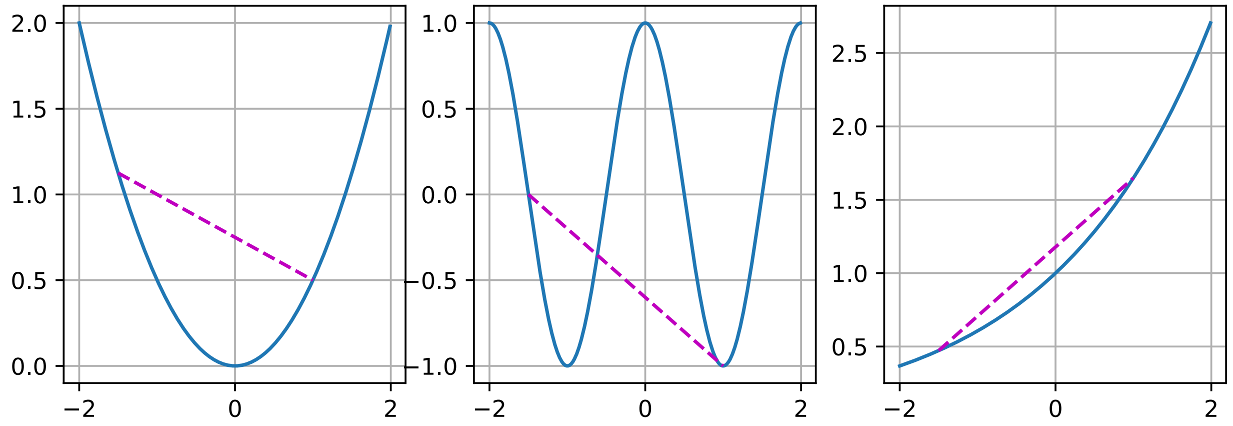
$$\lambda f(x) + (1 - \lambda) f(x') \geq f(\lambda x + (1 - \lambda) x'). \tag{11.2.2}$$

To illustrate this let's plot a few functions and check which ones satisfy the requirement. We need to import a few libraries.

```
%matplotlib inline
import d2l
from mpl_toolkits import mplot3d
from mxnet import np, npx
npx.set_np()
```

Let's define a few functions, both convex and nonconvex.

```
def f(x):
    return 0.5 * x**2  # Convex

def g(x):
    return np.cos(np.pi * x)  # Nonconvex

def h(x):
    return np.exp(0.5 * x)  # Convex

x, segment = np.arange(-2, 2, 0.01), np.array([-1.5, 1])
d2l.use_svg_display()
_, axes = d2l.plt.subplots(1, 3, figsize=(9, 3))

for ax, func in zip(axes, [f, g, h]):
    d2l.plot([x, segment], [func(x), func(segment)], axes=ax)
```



As expected, the cosine function is nonconvex, whereas the parabola and the exponential function are. Note that the requirement that X is necessary for the condition to make sense. Otherwise the outcome of $f(\lambda x + (1 - \lambda)x')$ might not be well defined. Convex functions have a number of desirable properties.

Jensen's Inequality

One of the most useful tools is Jensen's inequality. It amounts to a generalization of the definition of convexity:

$$\sum_i \alpha_i f(x_i) \geq f\left(\sum_i \alpha_i x_i\right) \text{ and } E_x[f(x)] \quad \geq f\left(E_x[x]\right). \tag{11.2.3}$$

In other words, the expectation of a convex function is larger than the convex function of an expectation. To prove the first inequality we repeatedly apply the definition of convexity to one term in the sum at a time. The expectation can be proven by taking the limit over finite segments.

One of the common applications of Jensen's inequality is with regard to the log-likelihood of partially observed random variables. That is, we use

$$E_{y \sim P(y)}[-\log P(x \mid y)] \geq -\log P(x). \tag{11.2.4}$$

This follows since $\int P(y)P(x \mid y)dy = P(x)$. This is used in variational methods. Here y is typically the unobserved random variable, $P(y)$ is the best guess of how it might be distributed and $P(x)$ is the distribution with y integrated out. For instance, in clustering y might be the cluster labels and $P(x \mid y)$ is the generative model when applying cluster labels.

11.2.2 Properties

Convex functions have a few useful properties. We describe them as follows.

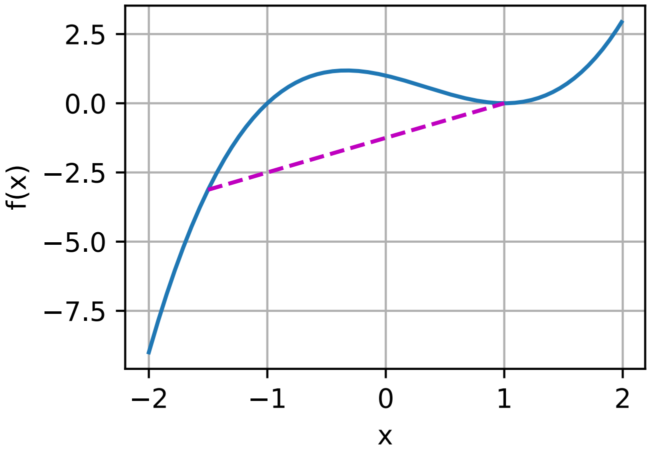
No Local Minima

In particular, convex functions do not have local minima. Let's assume the contrary and prove it wrong. If $x \in X$ is a local minimum there exists some neighborhood of x for which $f(x)$ is the smallest value. Since x is only a local minimum there has to be another $x' \in X$ for which $f(x') < f(x)$. However, by convexity the function values on the entire *line* $\lambda x + (1-\lambda)x'$ have to be less than $f(x')$ since for $\lambda \in [0, 1)$

$$f(x) > \lambda f(x) + (1-\lambda)f(x') \geq f(\lambda x + (1-\lambda)x'). \tag{11.2.5}$$

This contradicts the assumption that $f(x)$ is a local minimum. For instance, the function $f(x) = (x+1)(x-1)^2$ has a local minimum for $x = 1$. However, it is not a global minimum.

```
def f(x):
    return (x-1)**2 * (x+1)

d2l.set_figsize((3.5, 2.5))
d2l.plot([x, segment], [f(x), f(segment)], 'x', 'f(x)')
```



The fact that convex functions have no local minima is very convenient. It means that if we minimize functions we cannot "get stuck". Note, though, that this does not mean that there cannot be more than one global minimum or that there might even exist one. For instance, the function $f(x) = \max(|x|-1, 0)$ attains its minimum value over the interval $[-1, 1]$. Conversely, the function $f(x) = \exp(x)$ does not attain a minimum value on $\mathbb{R}$. For $x \to -\infty$ it asymptotes to 0, however there is no x for which $f(x) = 0$.

Convex Functions and Sets

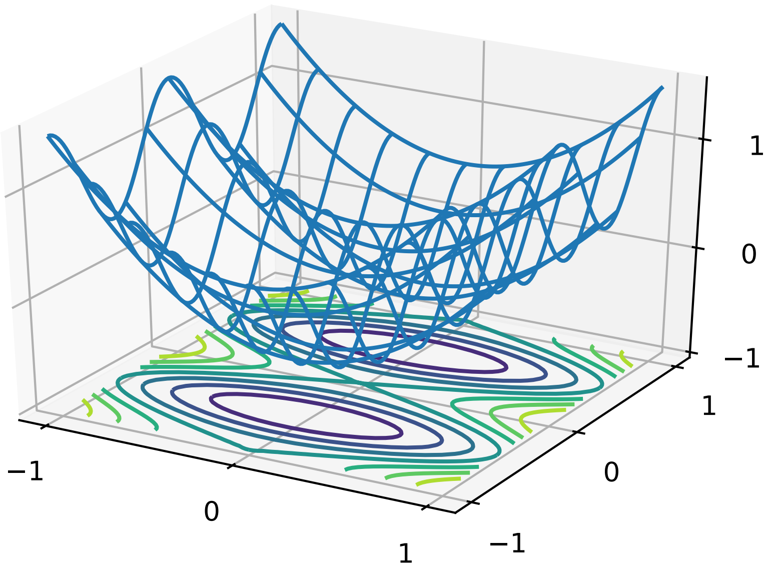
Convex functions define convex sets as *below-sets*. They are defined as

$$S_b := \{x | x \in X \text{ and } f(x) \leq b\}. \tag{11.2.6}$$

Such sets are convex. Let's prove this quickly. Remember that for any $x, x' \in S_b$ we need to show that $\lambda x + (1-\lambda)x' \in S_b$ as long as $\lambda \in [0,1]$. But this follows directly from the definition of convexity since $f(\lambda x + (1-\lambda)x') \leq \lambda f(x) + (1-\lambda)f(x') \leq b$.

Have a look at the function $f(x,y) = 0.5x^2 + \cos(2\pi y)$ below. It is clearly nonconvex. The level sets are correspondingly nonconvex. In fact, they are typically composed of disjoint sets.

```
x, y = np.meshgrid(np.linspace(-1, 1, 101), np.linspace(-1, 1, 101),
                   indexing='ij')

z = x**2 + 0.5 * np.cos(2 * np.pi * y)

# Plot the 3D surface
d2l.set_figsize((6, 4))
ax = d2l.plt.figure().add_subplot(111, projection='3d')
ax.plot_wireframe(x, y, z, **{'rstride': 10, 'cstride': 10})
ax.contour(x, y, z, offset=-1)
ax.set_zlim(-1, 1.5)

# Adjust labels
for func in [d2l.plt.xticks, d2l.plt.yticks, ax.set_zticks]:
    func([-1, 0, 1])
```



Derivatives and Convexity

Whenever the second derivative of a function exists it is very easy to check for convexity. All we need to do is check whether $\partial_x^2 f(x) \succeq 0$, i.e., whether all of its eigenvalues are nonnegative. For instance, the function $f(\mathbf{x}) = \frac{1}{2}\|\mathbf{x}\|_2^2$ is convex since $\partial_{\mathbf{x}}^2 f = \mathbf{1}$, i.e., its derivative is the identity matrix.

The first thing to realize is that we only need to prove this property for one-dimensional functions. After all, in general we can always define some function $g(z) = f(\mathbf{x} + z \cdot \mathbf{v})$. This function has the first and second derivatives $g' = (\partial_{\mathbf{x}} f)^\top \mathbf{v}$ and $g'' = \mathbf{v}^\top (\partial_{\mathbf{x}}^2 f)\mathbf{v}$ respectively. In particular, $g'' \geq 0$ for all $\mathbf{v}$ whenever the Hessian of f is positive semidefinite, i.e., whenever all of its eigenvalues are greater equal than zero. Hence back to the scalar case.

To see that $f''(x) \geq 0$ for convex functions we use the fact that

$$\frac{1}{2} f(x + \epsilon) + \frac{1}{2} f(x - \epsilon) \geq f\left(\frac{x + \epsilon}{2} + \frac{x - \epsilon}{2}\right) = f(x). \tag{11.2.7}$$

Since the second derivative is given by the limit over finite differences it follows that

$$f''(x) = \lim_{\epsilon \to 0} \frac{f(x + \epsilon) + f(x - \epsilon) - 2f(x)}{\epsilon^2} \geq 0. \tag{11.2.8}$$

To see that the converse is true we use the fact that $f'' \geq 0$ implies that f' is a monotonically increasing function. Let $a < x < b$ be three points in $\mathbb{R}$. We use the mean value theorem to express

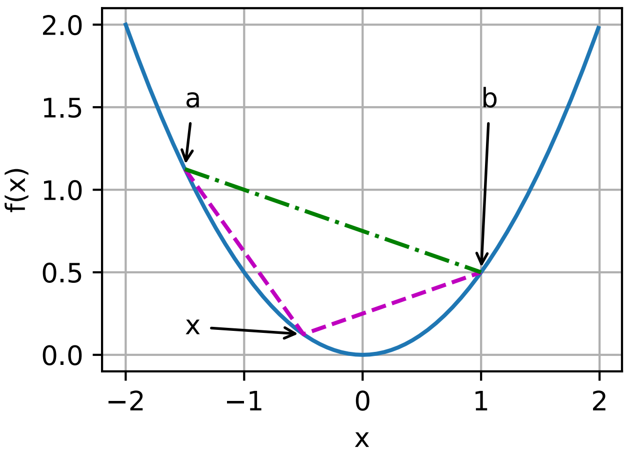
$$\begin{aligned} f(x) - f(a) &= (x - a) f'(\alpha) \text{ for some } \alpha \in [a, x] \text{ and} \\ f(b) - f(x) &= (b - x) f'(\beta) \text{ for some } \beta \in [x, b]. \end{aligned} \tag{11.2.9}$$

By monotonicity $f'(\beta) \geq f'(\alpha)$, hence

$$\begin{aligned} f(b) - f(a) &= f(b) - f(x) + f(x) - f(a) \\ &= (b - x) f'(\beta) + (x - a) f'(\alpha) \\ &\geq (b - a) f'(\alpha). \end{aligned} \tag{11.2.10}$$

By geometry it follows that $f(x)$ is below the line connecting $f(a)$ and $f(b)$, thus proving convexity. We omit a more formal derivation in favor of a graph below.

```
def f(x):
    return 0.5 * x**2

x = np.arange(-2, 2, 0.01)
axb, ab = np.array([-1.5, -0.5, 1]), np.array([-1.5, 1])

d2l.set_figsize((3.5, 2.5))
d2l.plot([x, axb, ab], [f(x) for x in [x, axb, ab]], 'x', 'f(x)')
d2l.annotate('a', (-1.5, f(-1.5)), (-1.5, 1.5))
d2l.annotate('b', (1, f(1)), (1, 1.5))
d2l.annotate('x', (-0.5, f(-0.5)), (-1.5, f(-0.5)))
```



11.2.3 Constraints

One of the nice properties of convex optimization is that it allows us to handle constraints efficiently. That is, it allows us to solve problems of the form:

$$\begin{aligned} &\underset{\mathbf{x}}{\text{minimize}}\ f(\mathbf{x}) \\ &\text{subject to } c_i(\mathbf{x}) \leq 0 \text{ for all } i \in \{1, \ldots, N\}. \end{aligned} \tag{11.2.11}$$

Here f is the objective and the functions c_i are constraint functions. To see what this does consider the case where $c_1(\mathbf{x}) = \|\mathbf{x}\|_2 - 1$. In this case the parameters $\mathbf{x}$ are constrained to the unit ball. If a second constraint is $c_2(\mathbf{x}) = \mathbf{v}^\top \mathbf{x} + b$, then this corresponds to all $\mathbf{x}$ lying on a halfspace. Satisfying both constraints simultaneously amounts to selecting a slice of a ball as the constraint set.

Lagrange Function

In general, solving a constrained optimization problem is difficult. One way of addressing it stems from physics with a rather simple intuition. Imagine a ball inside a box. The ball will roll to the place that is lowest and the forces of gravity will be balanced out with the forces that the sides of the box can impose on the ball. In short, the gradient of the objective function (i.e., gravity) will be offset by the gradient of the constraint function (need to remain inside the box by virtue of the walls "pushing back"). Note that any constraint that is not active (i.e., the ball does not touch the wall) will not be able to exert any force on the ball.

Skipping over the derivation of the Lagrange function L (see e.g., the book by Boyd and Vandenberghe for details (Boyd & Vandenberghe, 2004)) the above reasoning can be expressed via the following saddlepoint optimization problem:

$$L(\mathbf{x}, \alpha) = f(\mathbf{x}) + \sum_i \alpha_i c_i(\mathbf{x}) \text{ where } \alpha_i \geq 0. \tag{11.2.12}$$

Here the variables α_i are the so-called *Lagrange Multipliers* that ensure that a constraint is properly enforced. They are chosen just large enough to ensure that $c_i(\mathbf{x}) \leq 0$ for all i. For instance, for any $\mathbf{x}$ for which $c_i(\mathbf{x}) < 0$ naturally, we'd end up picking $\alpha_i = 0$. Moreover, this is a *saddlepoint* optimization problem where one wants to *maximize* L with respect to α and simultaneously *minimize* it with respect to $\mathbf{x}$. There is a rich body of literature explaining how to arrive at the function $L(\mathbf{x}, \alpha)$. For our purposes it is sufficient to know that the saddlepoint of L is where the original constrained optimization problem is solved optimally.

Penalties

One way of satisfying constrained optimization problems at least approximately is to adapt the Lagrange function L. Rather than satisfying $c_i(\mathbf{x}) \leq 0$ we simply add $\alpha_i c_i(\mathbf{x})$ to the objective function $f(x)$. This ensures that the constraints will not be violated too badly.

In fact, we have been using this trick all along. Consider weight decay in Section 4.5. In it we add $\frac{\lambda}{2}\|\mathbf{w}\|^2$ to the objective function to ensure that $\mathbf{w}$ does not grow too large. Using the constrained optimization point of view we can see that this will ensure that $\|\mathbf{w}\|^2 - r^2 \leq 0$ for some radius r. Adjusting the value of λ allows us to vary the size of $\mathbf{w}$.

In general, adding penalties is a good way of ensuring approximate constraint satisfaction. In practice this turns out to be much more robust than exact satisfaction. Furthermore, for nonconvex problems many of the properties that make the exact approach so appealing in the convex case (e.g., optimality) no longer hold.

Projections

An alternative strategy for satisfying constraints are projections. Again, we encountered them before, e.g., when dealing with gradient clipping in Section 8.5. There we ensured that a gradient has length bounded by c via

$$\mathbf{g} \leftarrow \mathbf{g} \cdot \mathrm{min}(1, c/\|\mathbf{g}\|). \tag{11.2.13}$$

This turns out to be a *projection* of g onto the ball of radius c. More generally, a projection on a (convex) set X is defined as

$$\mathrm{Proj}_X(\mathbf{x}) = \mathop{\mathrm{argmin}}_{\mathbf{x}' \in X} \|\mathbf{x} - \mathbf{x}'\|_2. \tag{11.2.14}$$

It is thus the closest point in X to $\mathbf{x}$. This sounds a bit abstract. Fig. 11.2.4 explains it somewhat more clearly. In it we have two convex sets, a circle and a diamond. Points inside the set (yellow) remain unchanged. Points outside the set (black) are mapped to the closest point inside the set (red). While for ℓ_2 balls this leaves the direction unchanged, this need not be the case in general, as can be seen in the case of the diamond.

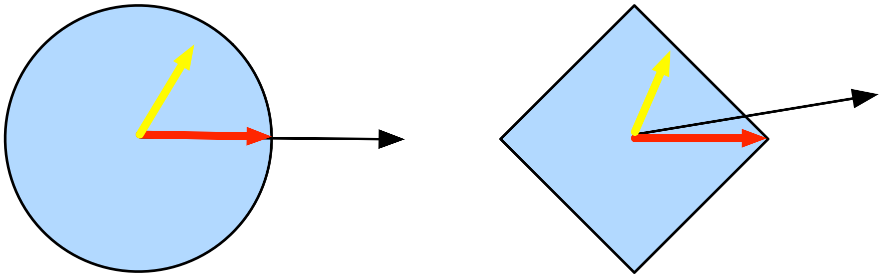


Fig. 11.2.4: Convex Projections

One of the uses for convex projections is to compute sparse weight vectors. In this case we project $\mathbf{w}$ onto an ℓ_1 ball (the latter is a generalized version of the diamond in the picture above).

Summary

In the context of deep learning the main purpose of convex functions is to motivate optimization algorithms and help us understand them in detail. In the following we will see how gradient descent and stochastic gradient descent can be derived accordingly.

- Intersections of convex sets are convex. Unions are not.
- The expectation of a convex function is larger than the convex function of an expectation (Jensen's inequality).
- A twice-differentiable function is convex if and only if its second derivative has only non-negative eigenvalues throughout.
- Convex constraints can be added via the Lagrange function. In practice simply add them with a penalty to the objective function.
- Projections map to points in the (convex) set closest to the original point.

Exercises

1. Assume that we want to verify convexity of a set by drawing all lines between points within the set and checking whether the lines are contained.
   - Prove that it is sufficient to check only the points on the boundary.
   - Prove that it is sufficient to check only the vertices of the set.
2. Denote by $B_p[r] := \{\mathbf{x}|\mathbf{x} \in \mathbb{R}^d \text{ and } \|\mathbf{x}\|_p \leq r\}$ the ball of radius r using the p-norm. Prove that $B_p[r]$ is convex for all $p \geq 1$.
3. Given convex functions f and g show that $\max(f, g)$ is convex, too. Prove that $\min(f, g)$ is not convex.
4. Prove that the normalization of the softmax function is convex. More specifically prove the convexity of $f(x) = \log \sum_i \exp(x_i)$.
5. Prove that linear subspaces are convex sets, i.e., $X = \{\mathbf{x}|\mathbf{W}\mathbf{x} = \mathbf{b}\}$.
6. Prove that in the case of linear subspaces with $\mathbf{b} = 0$ the projection Proj_X can be written as $\mathbf{M}\mathbf{x}$ for some matrix $\mathbf{M}$.
7. Show that for convex twice differentiable functions f we can write $f(x+\epsilon) = f(x) + \epsilon f'(x) + \frac{1}{2}\epsilon^2 f''(x+\xi)$ for some $\xi \in [0, \epsilon]$.
8. Given a vector $\mathbf{w} \in \mathbb{R}^d$ with $\|\mathbf{w}\|_1 > 1$ compute the projection on the ℓ_1 unit ball.
   - As intermediate step write out the penalized objective $\|\mathbf{w} - \mathbf{w}'\|_2^2 + \lambda\|\mathbf{w}'\|_1$ and compute the solution for a given $\lambda > 0$.
   - Can you find the 'right' value of λ without a lot of trial and error?
9. Given a convex set X and two vectors $\mathbf{x}$ and $\mathbf{y}$ prove that projections never increase distances, i.e., $\|\mathbf{x} - \mathbf{y}\| \geq \|\mathrm{Proj}_X(\mathbf{x}) - \mathrm{Proj}_X(\mathbf{y})\|$.

11.3 Gradient Descent

In this section we are going to introduce the basic concepts underlying gradient descent. This is brief by necessity. See e.g., (Boyd & Vandenberghe, 2004) for an in-depth introduction to convex optimization. Although the latter is rarely used directly in deep learning, an understanding of gradient descent is key to understanding stochastic gradient descent algorithms. For instance, the optimization problem might diverge due to an overly large learning rate. This phenomenon can already be seen in gradient descent. Likewise, preconditioning is a common technique in gradient descent and carries over to more advanced algorithms. Let's start with a simple special case.

11.3.1 Gradient Descent in One Dimension

Gradient descent in one dimension is an excellent example to explain why the gradient descent algorithm may reduce the value of the objective function. Consider some continuously differentiable real-valued function $f: \mathbb{R} \rightarrow \mathbb{R}$. Using a Taylor expansion (Section 17.3) we obtain that

$$f(x + \epsilon) = f(x) + \epsilon f'(x) + \mathcal{O}(\epsilon^2). \tag{11.3.1}$$

That is, in first approximation $f(x + \epsilon)$ is given by the function value $f(x)$ and the first derivative $f'(x)$ at x. It is not unreasonable to assume that for small ϵ moving in the direction of the negative gradient will decrease f. To keep things simple we pick a fixed step size $\eta > 0$ and choose $\epsilon = -\eta f'(x)$. Plugging this into the Taylor expansion above we get

$$f(x - \eta f'(x)) = f(x) - \eta f'^2(x) + \mathcal{O}(\eta^2 f'^2(x)). \tag{11.3.2}$$

If the derivative $f'(x) \neq 0$ does not vanish we make progress since $\eta f'^2(x) > 0$. Moreover, we can always choose η small enough for the higher order terms to become irrelevant. Hence we arrive at

$$f(x - \eta f'(x)) \lessapprox f(x). \tag{11.3.3}$$

This means that, if we use

$$x \leftarrow x - \eta f'(x) \tag{11.3.4}$$

to iterate x, the value of function $f(x)$ might decline. Therefore, in gradient descent we first choose an initial value x and a constant $\eta > 0$ and then use them to continuously iterate x until the stop condition is reached, for example, when the magnitude of the gradient $|f'(x)|$ is small enough or the number of iterations has reached a certain value.

For simplicity we choose the objective function $f(x) = x^2$ to illustrate how to implement gradient descent. Although we know that $x = 0$ is the solution to minimize $f(x)$, we still use this simple function to observe how x changes. As always, we begin by importing all required modules.

```
%matplotlib inline
import d2l
from mxnet import np, npx
npx.set_np()

def f(x):
    return x**2  # Objective function

def gradf(x):
    return 2 * x  # Its derivative
```

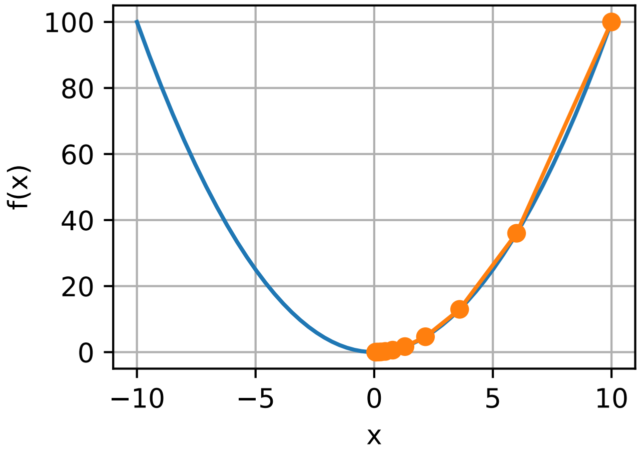
Next, we use $x = 10$ as the initial value and assume $\eta = 0.2$. Using gradient descent to iterate x for 10 times we can see that, eventually, the value of x approaches the optimal solution.

```
def gd(eta):
    x = 10
    results = [x]
    for i in range(10):
        x -= eta * gradf(x)
        results.append(x)
    print('epoch 10, x:', x)
    return results

res = gd(0.2)
```

```
epoch 10, x: 0.06046617599999997
```

The progress of optimizing over x can be plotted as follows.

```
def show_trace(res):
    n = max(abs(min(res)), abs(max(res)))
    f_line = np.arange(-n, n, 0.01)
    d2l.set_figsize((3.5, 2.5))
    d2l.plot([f_line, res], [[f(x) for x in f_line], [f(x) for x in res]],
             'x', 'f(x)', fmts=['-', '-o'])

show_trace(res)
```

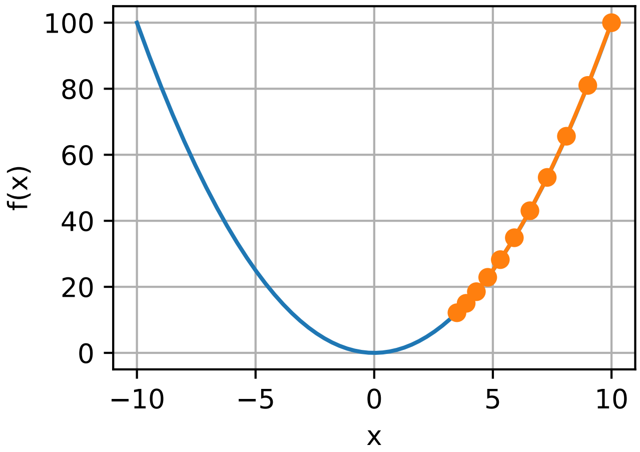


Learning Rate

The learning rate η can be set by the algorithm designer. If we use a learning rate that is too small, it will cause x to update very slowly, requiring more iterations to get a better solution. To show what happens in such a case, consider the progress in the same optimization problem for $\eta = 0.05$. As we can see, even after 10 steps we are still very far from the optimal solution.

```
show_trace(gd(0.05))
```

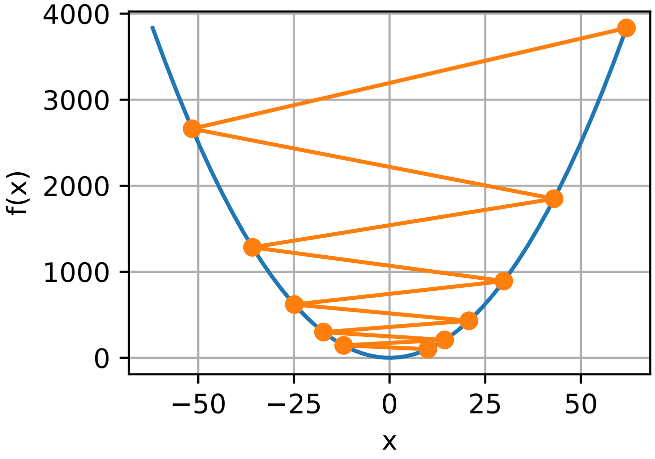
```
epoch 10, x: 3.4867844009999995
```



Conversely, if we use an excessively high learning rate, $|\eta f'(x)|$ might be too large for the first-order Taylor expansion formula. That is, the term $\mathcal{O}(\eta^2 f'^2(x))$ in (11.3.1) might become significant. In this case, we cannot guarantee that the iteration of x will be able to lower the value of $f(x)$. For example, when we set the learning rate to $\eta = 1.1$, x overshoots the optimal solution $x = 0$ and gradually diverges.

```
show_trace(gd(1.1))
```

```
epoch 10, x: 61.917364224000096
```

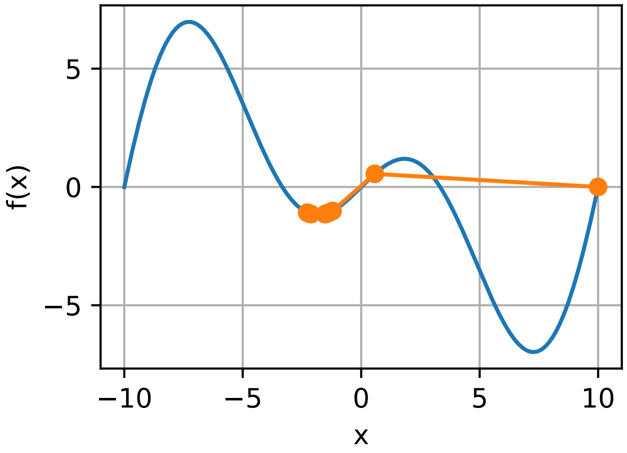


Local Minima

To illustrate what happens for nonconvex functions consider the case of $f(x) = x \cdot \cos cx$. This function has infinitely many local minima. Depending on our choice of learning rate and depending on how well conditioned the problem is, we may end up with one of many solutions. The example below illustrates how an (unrealistically) high learning rate will lead to a poor local minimum.

```
c = 0.15 * np.pi

def f(x):
    return x * np.cos(c * x)

def gradf(x):
    return np.cos(c * x) - c * x * np.sin(c * x)

show_trace(gd(2))
```

```
epoch 10, x: -1.528165927635083
```



11.3.2 Multivariate Gradient Descent

Now that have a better intuition of the univariate case, let's consider the situation where $\mathbf{x} \in \mathbb{R}^d$. That is, the objective function $f : \mathbb{R}^d \to \mathbb{R}$ maps vectors into scalars. Correspondingly its gradient is multivariate, too. It is a vector consisting of d partial derivatives:

$$\nabla f(\mathbf{x}) = \left[\frac{\partial f(\mathbf{x})}{\partial x_1}, \frac{\partial f(\mathbf{x})}{\partial x_2}, \ldots, \frac{\partial f(\mathbf{x})}{\partial x_d}\right]^\top. \tag{11.3.5}$$

Each partial derivative element $\partial f(\mathbf{x})/\partial x_i$ in the gradient indicates the rate of change of f at $\mathbf{x}$ with respect to the input x_i. As before in the univariate case we can use the corresponding Taylor approximation for multivariate functions to get some idea of what we should do. In particular, we have that

$$f(\mathbf{x} + \epsilon) = f(\mathbf{x}) + \epsilon^\top \nabla f(\mathbf{x}) + \mathcal{O}(\|\epsilon\|^2). \tag{11.3.6}$$

In other words, up to second order terms in **epsilon** the direction of steepest descent is given by the negative gradient $-\nabla f(\mathbf{x})$. Choosing a suitable learning rate $\eta > 0$ yields the prototypical gradient descent algorithm:

$$\mathbf{x} \leftarrow \mathbf{x} - \eta \nabla f(\mathbf{x}).$$

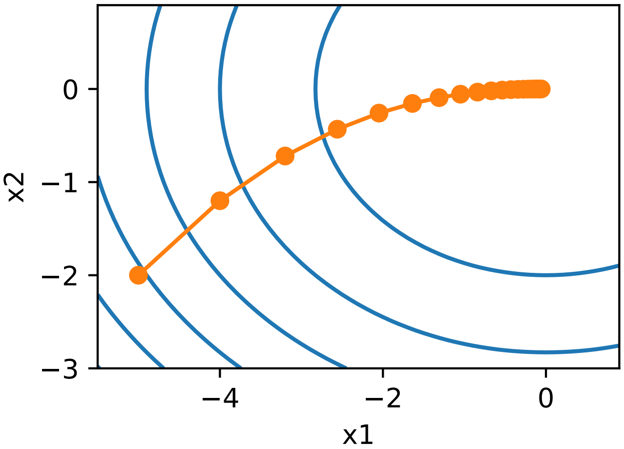
To see how the algorithm behaves in practice let's construct an objective function $f(\mathbf{x}) = x_1^2 + 2x_2^2$ with a two-dimensional vector $\mathbf{x} = [x_1, x_2]^\top$ as input and a scalar as output. The gradient is given by $\nabla f(\mathbf{x}) = [2x_1, 4x_2]^\top$. We will observe the trajectory of $\mathbf{x}$ by gradient descent from the initial position $[-5, -2]$. We need two more helper functions. The first uses an update function and applies it 20 times to the initial value. The second helper visualizes the trajectory of $\mathbf{x}$.

```
# Saved in the d2l package for later use
def train_2d(trainer, steps=20):
    """Optimize a 2-dim objective function with a customized trainer."""
    # s1 and s2 are internal state variables and will
    # be used later in the chapter
    x1, x2, s1, s2 = -5, -2, 0, 0
    results = [(x1, x2)]
    for i in range(steps):
        x1, x2, s1, s2 = trainer(x1, x2, s1, s2)
        results.append((x1, x2))
    print('epoch %d, x1 %f, x2 %f' % (i + 1, x1, x2))
    return results

# Saved in the d2l package for later use
def show_trace_2d(f, results):
    """Show the trace of 2D variables during optimization."""
    d2l.set_figsize((3.5, 2.5))
    d2l.plt.plot(*zip(*results), '-o', color='#ff7f0e')
    x1, x2 = np.meshgrid(np.arange(-5.5, 1.0, 0.1), np.arange(-3.0, 1.0, 0.1))
    d2l.plt.contour(x1, x2, f(x1, x2), colors='#1f77b4')
    d2l.plt.xlabel('x1')
    d2l.plt.ylabel('x2')
```

Next, we observe the trajectory of the optimization variable $\mathbf{x}$ for learning rate $\eta = 0.1$. We can see that after 20 steps the value of $\mathbf{x}$ approaches its minimum at $[0, 0]$. Progress is fairly well-behaved albeit rather slow.

```
def f(x1, x2):
    return x1 ** 2 + 2 * x2 ** 2  # Objective

def gradf(x1, x2):
    return (2 * x1, 4 * x2)  # Gradient

def gd(x1, x2, s1, s2):
    (g1, g2) = gradf(x1, x2)  # Compute gradient
    return (x1 - eta * g1, x2 - eta * g2, 0, 0)  # Update variables

eta = 0.1
show_trace_2d(f, train_2d(gd))
```

```
epoch 20, x1 -0.057646, x2 -0.000073
```



11.3.3 Adaptive Methods

As we could see in Section 11.3.1, getting the learning rate η "just right" is tricky. If we pick it too small, we make no progress. If we pick it too large, the solution oscillates and in the worst case it might even diverge. What if we could determine η automatically or get rid of having to select a step size at all? Second order methods that look not only at the value and gradient of the objective but also at its *curvature* can help in this case. While these methods cannot be applied to deep learning directly due to the computational cost, they provide useful intuition into how to design advanced optimization algorithms that mimic many of the desirable properties of the algorithms outlined below.

Newton's Method

Reviewing the Taylor expansion of f there is no need to stop after the first term. In fact, we can write it as

$$f(\mathbf{x} + \epsilon) = f(\mathbf{x}) + \epsilon^\top \nabla f(\mathbf{x}) + \frac{1}{2}\epsilon^\top \nabla \nabla^\top f(\mathbf{x}) \epsilon + \mathcal{O}(\|\epsilon\|^3). \tag{11.3.7}$$

To avoid cumbersome notation we define $H_f := \nabla \nabla^\top f(\mathbf{x})$ to be the *Hessian* of f. This is a $d \times d$ matrix. For small d and simple problems H_f is easy to compute. For deep networks, on the other hand, H_f may be prohibitively large, due to the cost of storing $\mathcal{O}(d^2)$ entries. Furthermore it may be too expensive to compute via backprop as we would need to apply backprop to the backpropagation call graph. For now let's ignore such considerations and look at what algorithm we'd get.

After all, the minimum of f satisfies $\nabla f(\mathbf{x}) = 0$. Taking derivatives of (11.3.7) with regard to ϵ and ignoring higher order terms we arrive at

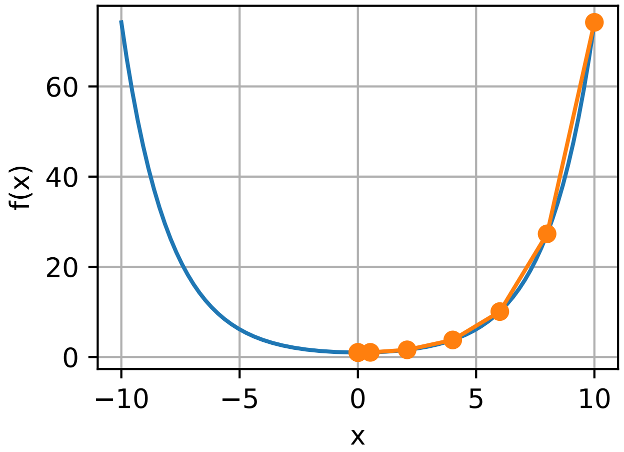
$$\nabla f(\mathbf{x}) + H_f \epsilon = 0 \text{ and hence } \epsilon = -H_f^{-1} \nabla f(\mathbf{x}). \tag{11.3.8}$$

That is, we need to invert the Hessian H_f as part of the optimization problem.

For $f(x) = \frac{1}{2}x^2$ we have $\nabla f(x) = x$ and $H_f = 1$. Hence for any x we obtain $\epsilon = -x$. In other words, a single step is sufficient to converge perfectly without the need for any adjustment! Alas, we got a bit lucky here since the Taylor expansion was exact. Let's see what happens in other problems.

```
c = 0.5

def f(x):
    return np.cosh(c * x)  # Objective

def gradf(x):
    return c * np.sinh(c * x)  # Derivative

def hessf(x):
    return c**2 * np.cosh(c * x)  # Hessian

# Hide learning rate for now
def newton(eta=1):
    x = 10
    results = [x]
    for i in range(10):
        x -= eta * gradf(x) / hessf(x)
        results.append(x)
    print('epoch 10, x:', x)
    return results

show_trace(newton())
```

```
epoch 10, x: 0.0
```



Now let's see what happens when we have a *nonconvex* function, such as $f(x) = x\cos(cx)$. After all, note that in Newton's method we end up dividing by the Hessian. This means that if the second derivative is *negative* we would walk into the direction of *increasing* f. That is a fatal flaw of the algorithm. Let's see what happens in practice.

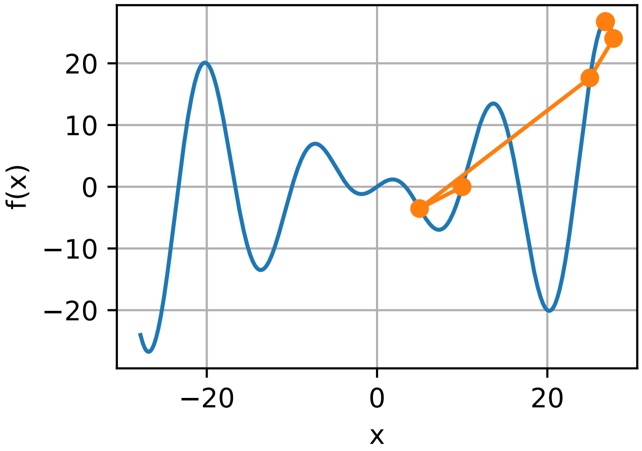
```
c = 0.15 * np.pi

def f(x):
    return x * np.cos(c * x)

def gradf(x):
    return np.cos(c * x) - c * x * np.sin(c * x)
```

(continues on next page)

(continued from previous page)

```
def hessf(x):
    return - 2 * c * np.sin(c * x) - x * c**2 * np.cos(c * x)

show_trace(newton())
```

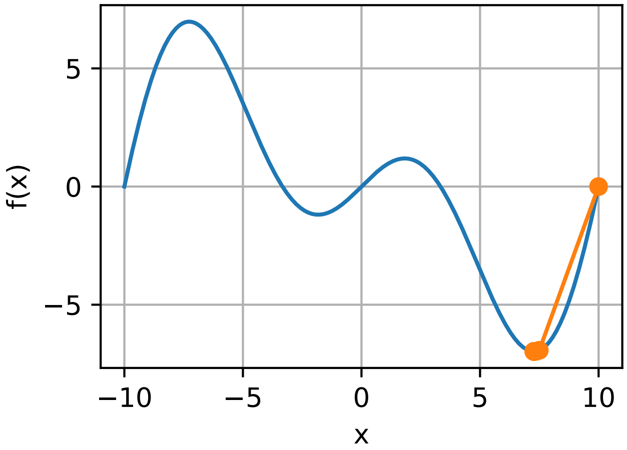
```
epoch 10, x: 26.83413291324767
```



This went spectacularly wrong. How can we fix it? One way would be to "fix" the Hessian by taking its absolute value instead. Another strategy is to bring back the learning rate. This seems to defeat the purpose, but not quite. Having second order information allows us to be cautious whenever the curvature is large and to take longer steps whenever the objective is flat. Let's see how this works with a slightly smaller learning rate, say $\eta = 0.5$. As we can see, we have quite an efficient algorithm.

```
show_trace(newton(0.5))
```

```
epoch 10, x: 7.269860168684531
```



Convergence Analysis

We only analyze the convergence rate for convex and three times differentiable f, where at its minimum x^* the second derivative is nonzero, i.e., where $f''(x^*) > 0$. The multivariate proof is a straightforward extension of the argument below and omitted since it does not help us much in terms of intuition.

Denote by x_k the value of x at the k-th iteration and let $e_k := x_k - x^*$ be the distance from optimality. By Taylor series expansion we have that the condition $f'(x^*) = 0$ can be written as

$$0 = f'(x_k - e_k) = f'(x_k) - e_k f''(x_k) + \frac{1}{2} e_k^2 f'''(\xi_k). \tag{11.3.9}$$

This holds for some $\xi_k \in [x_k - e_k, x_k]$. Recall that we have the update $x_{k+1} = x_k - f'(x_k)/f''(x_k)$. Dividing the above expansion by $f''(x_k)$ yields

$$e_k - f'(x_k)/f''(x_k) = \frac{1}{2} e_k^2 f'''(\xi_k)/f''(x_k). \tag{11.3.10}$$

Plugging in the update equations leads to the following bound $e_{k+1} \leq e_k^2 f'''(\xi_k)/f'(x_k)$. Consequently, whenever we are in a region of bounded $f'''(\xi_k)/f''(x_k) \leq c$, we have a quadratically decreasing error $e_{k+1} \leq c e_k^2$.

As an aside, optimization researchers call this *linear* convergence, whereas a condition such as $e_{k+1} \leq \alpha e_k$ would be called a *constant* rate of convergence. Note that this analysis comes with a number of caveats: We do not really have much of a guarantee when we will reach the region of rapid convergence. Instead, we only know that once we reach it, convergence will be very quick. Second, this requires that f is well-behaved up to higher order derivatives. It comes down to ensuring that f does not have any "surprising" properties in terms of how it might change its values.

Preconditioning

Quite unsurprisingly computing and storing the full Hessian is very expensive. It is thus desirable to find alternatives. One way to improve matters is by avoiding to compute the Hessian in its entirety but only compute the *diagonal* entries. While this is not quite as good as the full Newton method, it is still much better than not using it. Moreover, estimates for the main diagonal elements are what drives some of the innovation in stochastic gradient descent optimization algorithms. This leads to update algorithms of the form

$$\mathbf{x} \leftarrow \mathbf{x} - \eta \mathrm{diag}(H_f)^{-1} \nabla \mathbf{x}. \tag{11.3.11}$$

To see why this might be a good idea consider a situation where one variable denotes height in millimeters and the other one denotes height in kilometers. Assuming that for both the natural scale is in meters we have a terrible mismatch in parameterizations. Using preconditioning removes this. Effectively preconditioning with gradient descent amounts to selecting a different learning rate for each coordinate.

Gradient Descent with Line Search

One of the key problems in gradient descent was that we might overshoot the goal or make insufficient progress. A simple fix for the problem is to use line search in conjunction with gradient descent. That is, we use the direction given by $\nabla f(\mathbf{x})$ and then perform binary search as to which step length η minimizes $f(\mathbf{x} - \eta \nabla f(\mathbf{x}))$.

This algorithm converges rapidly (for an analysis and proof see e.g., (Boyd & Vandenberghe, 2004)). However, for the purpose of deep learning this is not quite so feasible, since each step of the line search would require us to evaluate the objective function on the entire dataset. This is way too costly to accomplish.

Summary

- Learning rates matter. Too large and we diverge, too small and we do not make progress.
- Gradient descent can get stuck in local minima.
- In high dimensions adjusting learning the learning rate is complicated.
- Preconditioning can help with scale adjustment.
- Newton's method is a lot faster *once* it has started working properly in convex problems.
- Beware of using Newton's method without any adjustments for nonconvex problems.

Exercises

1. Experiment with different learning rates and objective functions for gradient descent.
2. Implement line search to minimize a convex function in the interval $[a, b]$.
    - Do you need derivatives for binary search, i.e., to decide whether to pick $[a, (a + b)/2]$ or $[(a + b)/2, b]$.
    - How rapid is the rate of convergence for the algorithm?
    - Implement the algorithm and apply it to minimizing $\log(\exp(x) + \exp(-2 * x - 3))$.
3. Design an objective function defined on $\mathbb{R}^2$ where gradient descent is exceedingly slow. Hint - scale different coordinates differently.
4. Implement the lightweight version of Newton's method using preconditioning:
    - Use diagonal Hessian as preconditioner.
    - Use the absolute values of that rather than the actual (possibly signed) values.
    - Apply this to the problem above.
5. Apply the algorithm above to a number of objective functions (convex or not). What happens if you rotate coordinates by 45 degrees?

11.4 Stochastic Gradient Descent

In this section, we are going to introduce the basic principles of stochastic gradient descent.

```
%matplotlib inline
import d2l
import math
from mxnet import np, npx
npx.set_np()
```

11.4.1 Stochastic Gradient Updates

In deep learning, the objective function is usually the average of the loss functions for each example in the training dataset. We assume that $f_i(\mathbf{x})$ is the loss function of the training data instance with n examples, an index of i, and parameter vector of $\mathbf{x}$, then we have the objective function

$$f(\mathbf{x}) = \frac{1}{n} \sum_{i=1}^{n} f_i(\mathbf{x}). \tag{11.4.1}$$

The gradient of the objective function at $\mathbf{x}$ is computed as

$$\nabla f(\mathbf{x}) = \frac{1}{n} \sum_{i=1}^{n} \nabla f_i(\mathbf{x}). \tag{11.4.2}$$

If gradient descent is used, the computing cost for each independent variable iteration is $\mathcal{O}(n)$, which grows linearly with n. Therefore, when the model training data instance is large, the cost of gradient descent for each iteration will be very high.

Stochastic gradient descent (SGD) reduces computational cost at each iteration. At each iteration of stochastic gradient descent, we uniformly sample an index $i \in \{1, \ldots, n\}$ for data instances at random, and compute the gradient $\nabla f_i(\mathbf{x})$ to update $\mathbf{x}$:

$$\mathbf{x} \leftarrow \mathbf{x} - \eta \nabla f_i(\mathbf{x}). \tag{11.4.3}$$

Here, η is the learning rate. We can see that the computing cost for each iteration drops from $\mathcal{O}(n)$ of the gradient descent to the constant $\mathcal{O}(1)$. We should mention that the stochastic gradient $\nabla f_i(\mathbf{x})$ is the unbiased estimate of gradient $\nabla f(\mathbf{x})$.

$$\mathbb{E}_i \nabla f_i(\mathbf{x}) = \frac{1}{n} \sum_{i=1}^{n} \nabla f_i(\mathbf{x}) = \nabla f(\mathbf{x}). \tag{11.4.4}$$

This means that, on average, the stochastic gradient is a good estimate of the gradient.

Now, we will compare it to gradient descent by adding random noise with a mean of 0 to the gradient to simulate a SGD.

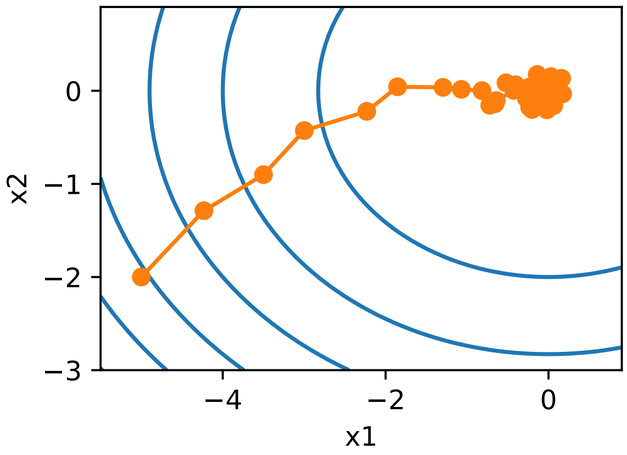
```
def f(x1, x2):
    return x1 ** 2 + 2 * x2 ** 2  # Objective

def gradf(x1, x2):
    return (2 * x1, 4 * x2)  # Gradient
```

(continues on next page)

(continued from previous page)

```
def sgd(x1, x2, s1, s2):  # Simulate noisy gradient
    global lr  # Learning rate scheduler
    (g1, g2) = gradf(x1, x2)  # Compute gradient
    (g1, g2) = (g1 + np.random.normal(0.1), g2 + np.random.normal(0.1))
    eta_t = eta * lr()  # Learning rate at time t
    return (x1 - eta_t * g1, x2 - eta_t * g2, 0, 0)  # Update variables

eta = 0.1
lr = (lambda: 1)  # Constant learning rate
d2l.show_trace_2d(f, d2l.train_2d(sgd, steps=50))
```

```
epoch 50, x1 -0.522513, x2 0.085780
```



As we can see, the trajectory of the variables in the SGD is much more noisy than the one we observed in gradient descent in the previous section. This is due to the stochastic nature of the gradient. That is, even when we arrive near the minimum, we are still subject to the uncertainty injected by the instantaneous gradient via $\eta\nabla f_i(\mathbf{x})$. Even after 50 steps the quality is still not so good. Even worse, it will not improve after additional steps (we encourage the reader to experiment with a larger number of steps to confirm this on his own). This leaves us with the only alternative—change the learning rate η. However, if we pick this too small, we will not make any meaningful progress initially. On the other hand, if we pick it too large, we will not get a good solution, as seen above. The only way to resolve these conflicting goals is to reduce the learning rate *dynamically* as optimization progresses.

This is also the reason for adding a learning rate function `lr` into the `sgd` step function. In the example above any functionality for learning rate scheduling lies dormant as we set the associated `lr` function to be constant, i.e., `lr = (lambda: 1)`.

11.4.2 Dynamic Learning Rate

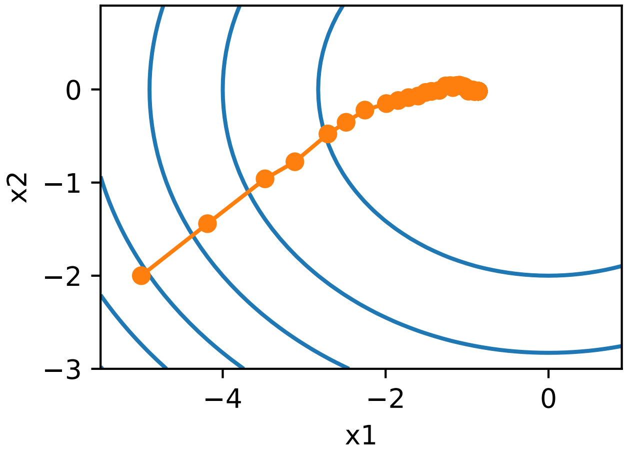
Replacing η with a time-dependent learning rate $\eta(t)$ adds to the complexity of controlling convergence of an optimization algorithm. In particular, need to figure out how rapidly η should decay. If it is too quick, we will stop optimizing prematurely. If we decrease it too slowly, we waste too much time on optimization. There are a few basic strategies that are used in adjusting η over time (we will discuss more advanced strategies in a later chapter):

$$\begin{aligned}
\eta(t) &= \eta_i \text{ if } t_i \leq t \leq t_{i+1} && \text{piecewise constant} \\
\eta(t) &= \eta_0 \cdot e^{-\lambda t} && \text{exponential} \\
\eta(t) &= \eta_0 \cdot (\beta t + 1)^{-\alpha} && \text{polynomial}
\end{aligned} \tag{11.4.5}$$

In the first scenario we decrease the learning rate, e.g., whenever progress in optimization has stalled. This is a common strategy for training deep networks. Alternatively we could decrease it much more aggressively by an exponential decay. Unfortunately this leads to premature stopping before the algorithm has converged. A popular choice is polynomial decay with $\alpha = 0.5$. In the case of convex optimization there are a number of proofs which show that this rate is well behaved. Let's see what this looks like in practice.

```
def exponential():
    global ctr
    ctr += 1
    return math.exp(-0.1 * ctr)

ctr = 1
lr = exponential  # Set up learning rate
d2l.show_trace_2d(f, d2l.train_2d(sgd, steps=1000))
```

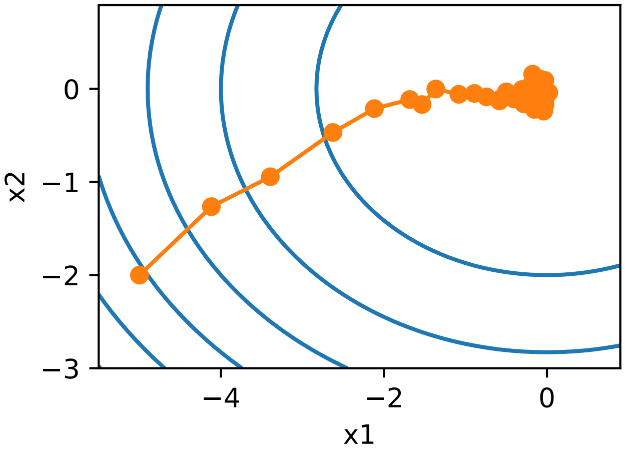
```
epoch 1000, x1 -0.862200, x2 -0.019736
```



As expected, the variance in the parameters is significantly reduced. However, this comes at the expense of failing to converge to the optimal solution $\mathbf{x} = (0, 0)$. Even after 1000 steps are we are still very far away from the optimal solution. Indeed, the algorithm fails to converge at all. On the other hand, if we use a polynomial decay where the learning rate decays with the inverse square root of the number of steps convergence is good.

```
def polynomial():
    global ctr
    ctr += 1
    return (1 + 0.1 * ctr)**(-0.5)

ctr = 1
lr = polynomial  # Set up learning rate
d2l.show_trace_2d(f, d2l.train_2d(sgd, steps=50))
```

```
epoch 50, x1 -0.024847, x2 0.090820
```



There exist many more choices for how to set the learning rate. For instance, we could start with a small rate, then rapidly ramp up and then decrease it again, albeit more slowly. We could even alternate between smaller and larger learning rates. There exists a large variety of such schedules. For now let's focus on learning rate schedules for which a comprehensive theoretical analysis is possible, i.e., on learning rates in a convex setting. For general nonconvex problems it is very difficult to obtain meaningful convergence guarantees, since in general minimizing nonlinear nonconvex problems is NP hard. For a survey see e.g., the excellent lecture notes[151] of Tibshirani 2015.

11.4.3 Convergence Analysis for Convex Objectives

The following is optional and primarily serves to convey more intuition about the problem. We limit ourselves to one of the simplest proofs, as described by (Nesterov & Vial, 2000). Significantly more advanced proof techniques exist, e.g., whenever the objective function is particularly well behaved. (Hazan et al., 2008) show that for strongly convex functions, i.e., for functions that can be bounded from below by $\mathbf{x}^\top \mathbf{Q}\mathbf{x}$, it is possible to minimize them in a small number of steps while decreasing the learning rate like $\eta(t) = \eta_0/(\beta t + 1)$. Unfortunately this case never really occurs in deep learning and we are left with a much more slowly decreasing rate in practice.

Consider the case where

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta_t \partial_{\mathbf{w}} l(\mathbf{x}_t, \mathbf{w}). \tag{11.4.6}$$

[151] https://www.stat.cmu.edu/~ryantibs/convexopt-F15/lectures/26-nonconvex.pdf

In particular, assume that $\mathbf{x}_t$ is drawn from some distribution $P(\mathbf{x})$ and that $l(\mathbf{x}, \mathbf{w})$ is a convex function in $\mathbf{w}$ for all $\mathbf{x}$. Last denote by

$$R(\mathbf{w}) = E_{\mathbf{x}\sim P}[l(\mathbf{x}, \mathbf{w})] \tag{11.4.7}$$

the expected risk and by R^* its minimum with regard to $\mathbf{w}$. Last let $\mathbf{w}^*$ be the minimizer (we assume that it exists within the domain which $\mathbf{w}$ is defined). In this case we can track the distance between the current parameter $\mathbf{w}_t$ and the risk minimizer $\mathbf{w}^*$ and see whether it improves over time:

$$\begin{aligned} \|\mathbf{w}_{t+1} - \mathbf{w}^*\|^2 &= \|\mathbf{w}_t - \eta_t \partial_{\mathbf{w}} l(\mathbf{x}_t, \mathbf{w}) - \mathbf{w}^*\|^2 \\ &= \|\mathbf{w}_t - \mathbf{w}^*\|^2 + \eta_t^2 \|\partial_{\mathbf{w}} l(\mathbf{x}_t, \mathbf{w})\|^2 - 2\eta_t \left\langle \mathbf{w}_t - \mathbf{w}^*, \partial_{\mathbf{w}} l(\mathbf{x}_t, \mathbf{w})\right\rangle. \end{aligned} \tag{11.4.8}$$

The gradient $\partial_{\mathbf{w}} l(\mathbf{x}_t, \mathbf{w})$ can be bounded from above by some Lipschitz constant L, hence we have that

$$\eta_t^2 \|\partial_{\mathbf{w}} l(\mathbf{x}_t, \mathbf{w})\|^2 \leq \eta_t^2 L^2. \tag{11.4.9}$$

We are mostly interested in how the distance between $\mathbf{w}_t$ and $\mathbf{w}^*$ changes *in expectation*. In fact, for any specific sequence of steps the distance might well increase, depending on whichever $\mathbf{x}_t$ we encounter. Hence we need to bound the inner product. By convexity we have that

$$l(\mathbf{x}_t, \mathbf{w}^*) \geq l(\mathbf{x}_t, \mathbf{w}_t) + \left\langle \mathbf{w}^* - \mathbf{w}_t, \partial_{\mathbf{w}} l(\mathbf{x}_t, \mathbf{w}_t)\right\rangle. \tag{11.4.10}$$

Using both inequalities and plugging it into the above we obtain a bound on the distance between parameters at time $t+1$ as follows:

$$\|\mathbf{w}_t - \mathbf{w}^*\|^2 - \|\mathbf{w}_{t+1} - \mathbf{w}^*\|^2 \geq 2\eta_t (l(\mathbf{x}_t, \mathbf{w}_t) - l(\mathbf{x}_t, \mathbf{w}^*)) - \eta_t^2 L^2. \tag{11.4.11}$$

This means that we make progress as long as the expected difference between current loss and the optimal loss outweighs $\eta_t L^2$. Since the former is bound to converge to 0 it follows that the learning rate η_t also needs to vanish.

Next we take expectations over this expression. This yields

$$E_{\mathbf{w}_t}\left[\|\mathbf{w}_t - \mathbf{w}^*\|^2\right] - E_{\mathbf{w}_{t+1}|\mathbf{w}_t}\left[\|\mathbf{w}_{t+1} - \mathbf{w}^*\|^2\right] \geq 2\eta_t [E[R[\mathbf{w}_t]] - R^*] - \eta_t^2 L^2. \tag{11.4.12}$$

The last step involves summing over the inequalities for $t \in \{t, \ldots, T\}$. Since the sum telescopes and by dropping the lower term we obtain

$$\|\mathbf{w}_0 - \mathbf{w}^*\|^2 \geq 2\sum_{t=1}^{T} \eta_t [E[R[\mathbf{w}_t]] - R^*] - L^2 \sum_{t=1}^{T} \eta_t^2. \tag{11.4.13}$$

Note that we exploited that $\mathbf{w}_0$ is given and thus the expectation can be dropped. Last define

$$\bar{\mathbf{w}} := \frac{\sum_{t=1}^{T} \eta_t \mathbf{w}_t}{\sum_{t=1}^{T} \eta_t}. \tag{11.4.14}$$

Then by convexity it follows that

$$\sum_t \eta_t E[R[\mathbf{w}_t]] \geq \sum \eta_t \cdot [E[\bar{\mathbf{w}}]]. \tag{11.4.15}$$

Plugging this into the above inequality yields the bound

$$[E[\bar{\mathbf{w}}]] - R^* \leq \frac{r^2 + L^2 \sum_{t=1}^{T} \eta_t^2}{2\sum_{t=1}^{T} \eta_t}. \tag{11.4.16}$$

Here $r^2 := \|\mathbf{w}_0 - \mathbf{w}^*\|^2$ is a bound on the distance between the initial choice of parameters and the final outcome. In short, the speed of convergence depends on how rapidly the loss function changes via the Lipschitz constant L and how far away from optimality the initial value is r. Note that the bound is in terms of $\bar{\mathbf{w}}$ rather than $\mathbf{w}_T$. This is the case since $\bar{\mathbf{w}}$ is a smoothed version of the optimization path. Now let's analyze some choices for η_t.

- **Known Time Horizon**. Whenever r, L and T are known we can pick $\eta = r/L\sqrt{T}$. This yields as upper bound $rL(1 + 1/T)/2\sqrt{T} < rL/\sqrt{T}$. That is, we converge with rate $\mathcal{O}(1/\sqrt{T})$ to the optimal solution.
- **Unknown Time Horizon**. Whenever we want to have a good solution for *any* time T we can pick $\eta = \mathcal{O}(1/\sqrt{T})$. This costs us an extra logarithmic factor and it leads to an upper bound of the form $\mathcal{O}(\log T/\sqrt{T})$.

Note that for strongly convex losses $l(\mathbf{x}, \mathbf{w}') \geq l(\mathbf{x}, \mathbf{w}) + \langle \mathbf{w}' - \mathbf{w}, \partial_{\mathbf{w}} l(\mathbf{x}, \mathbf{w}) \rangle + \frac{\lambda}{2}\|\mathbf{w} - \mathbf{w}'\|^2$ we can design even more rapidly converging optimization schedules. In fact, an exponential decay in η leads to a bound of the form $\mathcal{O}(\log T/T)$.

11.4.4 Stochastic Gradients and Finite Samples

So far we have played a bit fast and loose when it comes to talking about stochastic gradient descent. We posited that we draw instances x_i, typically with labels y_i from some distribution $p(x, y)$ and that we use this to update the weights w in some manner. In particular, for a finite sample size we simply argued that the discrete distribution $p(x, y) = \frac{1}{n}\sum_{i=1}^n \delta_{x_i}(x)\delta_{y_i}(y)$ allows us to perform SGD over it.

However, this is not really what we did. In the toy examples in the current section we simply added noise to an otherwise non-stochastic gradient, i.e., we pretended to have pairs (x_i, y_i). It turns out that this is justified here (see the exercises for a detailed discussion). More troubling is that in all previous discussions we clearly did not do this. Instead we iterated over all instances exactly once. To see why this is preferable consider the converse, namely that we are sampling n observations from the discrete distribution with replacement. The probability of choosing an element i at random is N^{-1}. Thus to choose it at least once is

$$P(\text{choose } i) = 1 - P(\text{omit } i) = 1 - (1 - N^{-1})^N \approx 1 - e^{-1} \approx 0.63. \tag{11.4.17}$$

A similar reasoning shows that the probability of picking a sample exactly once is given by $\binom{N}{1} N^{-1}(1 - N^{-1})^{N-1} = \frac{N-1}{N}(1 - N^{-1})^N \approx e^{-1} \approx 0.37$. This leads to an increased variance and decreased data efficiency relative to sampling without replacement. Hence, in practice we perform the latter (and this is the default choice throughout this book). Last note that repeated passes through the dataset traverse it in a *different* random order.

Summary

- For convex problems we can prove that for a wide choice of learning rates Stochastic Gradient Descent will converge to the optimal solution.
- For deep learning this is generally not the case. However, the analysis of convex problems gives us useful insight into how to approach optimization, namely to reduce the learning rate progressively, albeit not too quickly.
- Problems occur when the learning rate is too small or too large. In practice a suitable learning rate is often found only after multiple experiments.

- When there are more examples in the training dataset, it costs more to compute each iteration for gradient descent, so SGD is preferred in these cases.
- Optimality guarantees for SGD are in general not available in nonconvex cases since the number of local minima that require checking might well be exponential.

Exercises

1. Experiment with different learning rate schedules for SGD and with different numbers of iterations. In particular, plot the distance from the optimal solution $(0, 0)$ as a function of the number of iterations.
2. Prove that for the function $f(x_1, x_2) = x_1^2 + 2x_2^2$ adding normal noise to the gradient is equivalent to minimizing a loss function $l(\mathbf{x}, \mathbf{w}) = (x_1 - w_1)^2 + 2(x_2 - w_2)^2$ where x is drawn from a normal distribution.
    - Derive mean and variance of the distribution for **x**.
    - Show that this property holds in general for objective functions $f(\mathbf{x}) = \frac{1}{2}(\mathbf{x} - \mu)^\top Q(\mathbf{x} - \mu)$ for $Q \succeq 0$.
3. Compare convergence of SGD when you sample from $\{(x_1, y_1), \ldots, (x_m, y_m)\}$ with replacement and when you sample without replacement.
4. How would you change the SGD solver if some gradient (or rather some coordinate associated with it) was consistently larger than all other gradients?
5. Assume that $f(x) = x^2(1 + \sin x)$. How many local minima does f have? Can you change f in such a way that to minimize it one needs to evaluate all local minima?

11.5 Minibatch Stochastic Gradient Descent

So far we encountered two extremes in the approach to gradient based learning: Section 11.3 uses the full dataset to compute gradients and to update parameters, one pass at a time. Conversely Section 11.4 processes one observation at a time to make progress. Each of them has its own drawbacks. Gradient Descent is not particularly *data efficient* whenever data is very similar. Stochastic Gradient Descent is not particularly *computationally efficient* since CPUs and GPUs cannot exploit the full power of vectorization. This suggests that there might be a happy medium, and in fact, that's what we've been using so far in the examples we discussed.

11.5.1 Vectorization and Caches

At the heart of the decision to use minibatches is computational efficiency. This is most easily understood when considering parallelization to multiple GPUs and multiple servers. In this case we need to send at least one image to each GPU. With 8 GPUs per server and 16 servers we already arrive at a minibatch size of 128.

Things are a bit more subtle when it comes to single GPUs or even CPUs. These devices have multiple types of memory, often multiple type of compute units and different bandwidth constraints between them. For instance, a CPU has a small number of registers and then L1, L2 and in some cases even L3 cache (which is shared between the different processor cores). These caches are of increasing size and latency (and at the same time they're of decreasing bandwidth). Suffice it to say, the processor is capable of performing many more operations than what the main memory interface is able to provide.

- A 2GHz CPU with 16 cores and AVX-512 vectorization can process up to $2 \cdot 10^9 \cdot 16 \cdot 32 = 10^{12}$ bytes per second. The capability of GPUs easily exceeds this number by a factor of 100. On the other hand, a midrange server processor might not have much more than 100 GB/s bandwidth, i.e., less than one tenth of what would be required to keep the processor fed. To make matters worse, not all memory access is created equal: first, memory interfaces are typically 64 bit wide or wider (e.g., on GPUs up to 384 bit), hence reading a single byte incurs the cost of a much wider access.
- There is significant overhead for the first access whereas sequential access is relatively cheap (this is often called a burst read). There are many more things to keep in mind, such as caching when we have multiple sockets, chiplets and other structures. A detailed discussion of this is beyond the scope of this section. See e.g., this Wikipedia article[153] for a more in-depth discussion.

The way to alleviate these constraints is to use a hierarchy of CPU caches which are actually fast enough to supply the processor with data. This is *the* driving force behind batching in deep learning. To keep matters simple, consider matrix-matrix multiplication, say $\mathbf{A} = \mathbf{BC}$. We have a number of options for calculating $\mathbf{A}$. For instance we could try the following:

1. We could compute $\mathbf{A}_{ij} = \mathbf{B}_{i,:}\mathbf{C}_{:,j}^\top$, i.e., we could compute it element-wise by means of dot products.
2. We could compute $\mathbf{A}_{:,j} = \mathbf{B}\mathbf{C}_{:,j}^\top$, i.e., we could compute it one column at a time. Likewise we could compute $\mathbf{A}$ one row $\mathbf{A}_{i,:}$ at a time.
3. We could simply compute $\mathbf{A} = \mathbf{BC}$.
4. We could break $\mathbf{B}$ and $\mathbf{C}$ into smaller block matrices and compute $\mathbf{A}$ one block at a time.

If we follow the first option, we will need to copy one row and one column vector into the CPU each time we want to compute an element $\mathbf{A}_{ij}$. Even worse, due to the fact that matrix elements are aligned sequentially we are thus required to access many disjoint locations for one of the two vectors as we read them from memory. The second option is much more favorable. In it, we are able to keep the column vector $\mathbf{C}_{:,j}$ in the CPU cache while we keep on traversing through B. This halves the memory bandwidth requirement with correspondingly faster access. Of course, option 3 is most desirable. Unfortunately, most matrices might not entirely fit into cache (this is what we're discussing after all). However, option 4 offers a practically useful alternative: we can move blocks of the matrix into cache and multiply them locally. Optimized libraries take care of this for us. Let's have a look at how efficient these operations are in practice.

[153] https://en.wikipedia.org/wiki/Cache_hierarchy

Beyond computational efficiency, the overhead introduced by Python and by the deep learning framework itself is considerable. Recall that each time we execute a command the Python interpreter sends a command to the MXNet engine which needs to insert it into the compute graph and deal with it during scheduling. Such overhead can be quite detrimental. In short, it is highly advisable to use vectorization (and matrices) whenever possible.

```
%matplotlib inline
import d2l
from mxnet import autograd, gluon, init, np, npx
from mxnet.gluon import nn
npx.set_np()

timer = d2l.Timer()
A = np.zeros((1024, 1024))
B = np.random.normal(0, 1, (1024, 1024))
C = np.random.normal(0, 1, (1024, 1024))
```

Element-wise assignment simply iterates over all rows and columns of **B** and **C** respectively to assign the value to **A**.

```
# Compute A = B C one element at a time
timer.start()
for i in range(1024):
    for j in range(1024):
        A[i, j] = np.dot(B[i, :], C[:, j])
A.wait_to_read()
timer.stop()
```

```
931.6748354434967
```

A faster strategy is to perform column-wise assignment.

```
# Compute A = B C one column at a time
timer.start()
for j in range(1024):
    A[:, j] = np.dot(B, C[:, j])
A.wait_to_read()
timer.stop()
```

```
0.6227929592132568
```

Last, the most effective manner is to perform the entire operation in one block. Let's see what the respective speed of the operations is.

```
# Compute A = B C in one go
timer.start()
A = np.dot(B, C)
A.wait_to_read()
timer.stop()

# Multiply and add count as separate operations (fused in practice)
gigaflops = [2/i for i in timer.times]
```

(continues on next page)

(continued from previous page)

```
print("Performance in Gigaflops: element {:.3f}, \
      column {:.3f}, full {:.3f}".format(*gigaflops))
```

```
Performance in Gigaflops: element 0.002,       column 3.211, full 383.707
```

11.5.2 Minibatches

In the past we took it for granted that we would read *minibatches* of data rather than single observations to update parameters. We now give a brief justification for it. Processing single observations requires us to perform many single matrix-vector (or even vector-vector) multiplications, which is quite expensive and which incurs a significant overhead on behalf of the underlying deep learning framework. This applies both to evaluating a network when applied to data (often referred to as inference) and when computing gradients to update parameters. That is, this applies whenever we perform $\mathbf{w} \leftarrow \mathbf{w} - \eta_t \mathbf{g}_t$ where

$$\mathbf{g}_t = \partial_{\mathbf{w}} f(\mathbf{x}_t, \mathbf{w}) \tag{11.5.1}$$

We can increase the *computational* efficiency of this operation by applying it to a minibatch of observations at a time. That is, we replace the gradient $\mathbf{g}_t$ over a single observation by one over a small batch

$$\mathbf{g}_t = \partial_{\mathbf{w}} \frac{1}{|\mathcal{B}_t|} \sum_{i \in \mathcal{B}_t} f(\mathbf{x}_i, \mathbf{w}) \tag{11.5.2}$$

Let's see what this does to the statistical properties of $\mathbf{g}_t$: since both $\mathbf{x}_t$ and also all elements of the minibatch $\mathcal{B}_t$ are drawn uniformly at random from the training set, the expectation of the gradient remains unchanged. The variance, on the other hand, is reduced significantly. Since the minibatch gradient is composed of $b := |\mathcal{B}_t|$ independent gradients which are being averaged, its standard deviation is reduced by a factor of $b^{-\frac{1}{2}}$. This, by itself, is a good thing, since it means that the updates are more reliably aligned with the full gradient.

Naively this would indicate that choosing a large minibatch $\mathcal{B}_t$ would be universally desirable. Alas, after some point, the additional reduction in standard deviation is minimal when compared to the linear increase in computational cost. In practice we pick a minibatch that is large enough to offer good computational efficiency while still fitting into the memory of a GPU. To illustrate the savings let's have a look at some code. In it we perform the same matrix-matrix multiplication, but this time broken up into "minibatches" of 64 columns at a time.

```
timer.start()
for j in range(0, 1024, 64):
    A[:, j:j+64] = np.dot(B, C[:, j:j+64])
timer.stop()
print("Performance in Gigaflops: block {:.3f}".format(2/timer.times[3]))
```

```
Performance in Gigaflops: block 232.842
```

As we can see, the computation on the minibatch is essentially as efficient as on the full matrix. A word of caution is in order. In Section 7.5 we used a type of regularization that was heavily dependent on the amount of variance in a minibatch. As we increase the latter, the variance decreases and with it the benefit of the noise-injection due to batch normalization. See e.g., (Ioffe, 2017) for details on how to rescale and compute the appropriate terms.

11.5.3 Reading the Dataset

Let's have a look at how minibatches are efficiently generated from data. In the following we use a dataset developed by NASA to test the wing noise from different aircraft[154] to compare these optimization algorithms. For convenience we only use the first $1,500$ examples. The data is whitened for preprocessing, i.e., we remove the mean and rescale the variance to 1 per coordinate.

```
# Saved in the d2l package for later use
def get_data_ch11(batch_size=10, n=1500):
    data = np.genfromtxt('../data/airfoil_self_noise.dat',
                         dtype=np.float32, delimiter='\t')
    data = (data - data.mean(axis=0)) / data.std(axis=0)
    data_iter = d2l.load_array(
        (data[:n, :-1], data[:n, -1]), batch_size, is_train=True)
    return data_iter, data.shape[1]-1
```

11.5.4 Implementation from Scratch

Recall the minibatch SGD implementation from Section 3.2. In the following we provide a slightly more general implementation. For convenience it has the same call signature as the other optimization algorithms introduced later in this chapter. Specifically, we add the status input `states` and place the hyperparameter in dictionary `hyperparams`. In addition, we will average the loss of each minibatch example in the training function, so the gradient in the optimization algorithm does not need to be divided by the batch size.

```
def sgd(params, states, hyperparams):
    for p in params:
        p[:] -= hyperparams['lr'] * p.grad
```

Next, we implement a generic training function to facilitate the use of the other optimization algorithms introduced later in this chapter. It initializes a linear regression model and can be used to train the model with minibatch SGD and other algorithms introduced subsequently.

```
# Saved in the d2l package for later use
def train_ch11(trainer_fn, states, hyperparams, data_iter,
               feature_dim, num_epochs=2):
    # Initialization
    w = np.random.normal(scale=0.01, size=(feature_dim, 1))
    b = np.zeros(1)
    w.attach_grad()
    b.attach_grad()
    net, loss = lambda X: d2l.linreg(X, w, b), d2l.squared_loss
    # Train
    animator = d2l.Animator(xlabel='epoch', ylabel='loss',
                            xlim=[0, num_epochs], ylim=[0.22, 0.35])
    n, timer = 0, d2l.Timer()
    for _ in range(num_epochs):
        for X, y in data_iter:
            with autograd.record():
                l = loss(net(X), y).mean()
```

(continues on next page)

[154] https://archive.ics.uci.edu/ml/datasets/Airfoil+Self-Noise

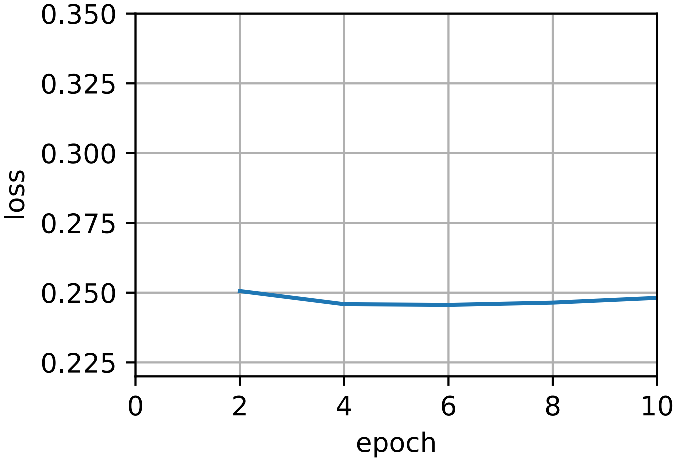
(continued from previous page)

```
            l.backward()
            trainer_fn([w, b], states, hyperparams)
            n += X.shape[0]
            if n % 200 == 0:
                timer.stop()
                animator.add(n/X.shape[0]/len(data_iter),
                             (d2l.evaluate_loss(net, data_iter, loss),))
                timer.start()
    print('loss: %.3f, %.3f sec/epoch' % (animator.Y[0][-1], timer.avg()))
    return timer.cumsum(), animator.Y[0]
```

Let's see how optimization proceeds for batch gradient descent. This can be achieved by setting the minibatch size to 1500 (i.e., to the total number of examples). As a result the model parameters are updated only once per epoch. There is little progress. In fact, after 6 steps progress stalls.

```
def train_sgd(lr, batch_size, num_epochs=2):
    data_iter, feature_dim = get_data_ch11(batch_size)
    return train_ch11(
        sgd, None, {'lr': lr}, data_iter, feature_dim, num_epochs)

gd_res = train_sgd(1, 1500, 10)
```

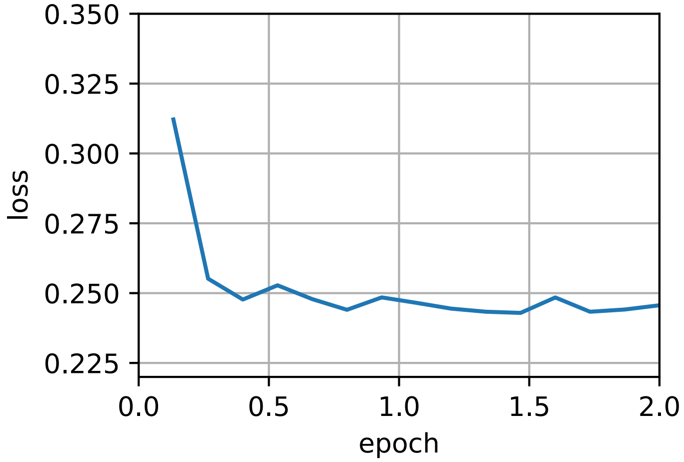
```
loss: 0.248, 0.075 sec/epoch
```



When the batch size equals 1, we use SGD for optimization. For simplicity of implementation we picked a constant (albeit small) learning rate. In SGD, the model parameters are updated whenever an example is processed. In our case this amounts to 1500 updates per epoch. As we can see, the decline in the value of the objective function slows down after one epoch. Although both the procedures processed 1500 examples within one epoch, SGD consumes more time than gradient descent in our experiment. This is because SGD updated the parameters more frequently and since it is less efficient to process single observations one at a time.

```
sgd_res = train_sgd(0.005, 1)
```

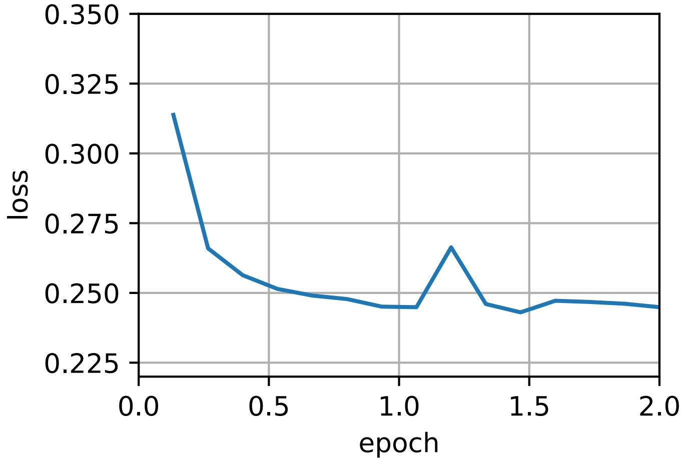
```
loss: 0.246, 0.328 sec/epoch
```



Last, when the batch size equals 100, we use minibatch SGD for optimization. The time required per epoch is longer than the time needed for SGD and the time for batch gradient descent.

```
mini1_res = train_sgd(.4, 100)
```

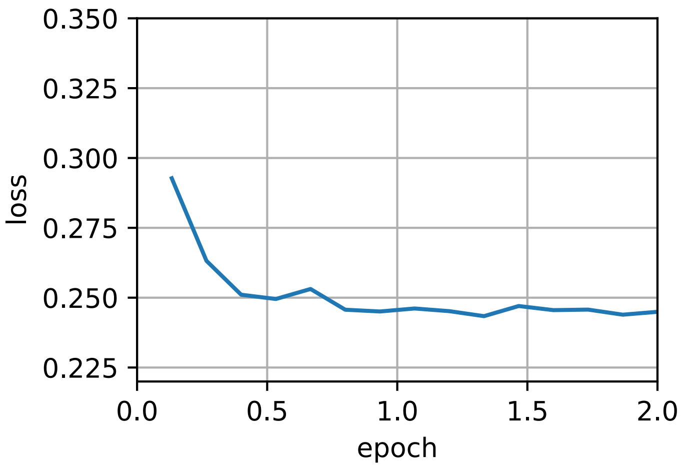
```
loss: 0.245, 0.008 sec/epoch
```



Reducing the batch size to 10, the time for each epoch increases because the workload for each batch is less efficient to execute.

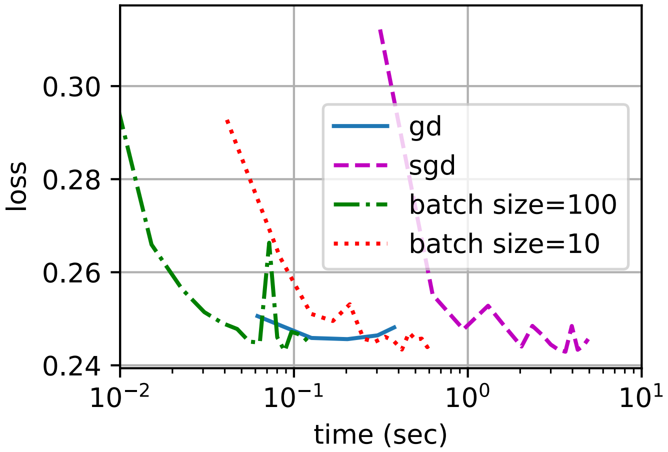
```
mini2_res = train_sgd(.05, 10)
```

```
loss: 0.245, 0.042 sec/epoch
```



Finally, we compare the time versus loss for the preview four experiments. As can be seen, despite SGD converges faster than GD in terms of number of examples processed, it uses more time to reach the same loss than GD because that computing gradient example by example is not efficient. Minibatch SGD is able to trade-off the convergence speed and computation efficiency. A minibatch size 10 is more efficient than SGD; a minibatch size 100 even outperforms GD in terms of runtime.

```
d2l.set_figsize([6, 3])
d2l.plot(*list(map(list, zip(gd_res, sgd_res, mini1_res, mini2_res))),
         'time (sec)', 'loss', xlim=[1e-2, 10],
         legend=['gd', 'sgd', 'batch size=100', 'batch size=10'])
d2l.plt.gca().set_xscale('log')
```



11.5.5 Concise Implementation

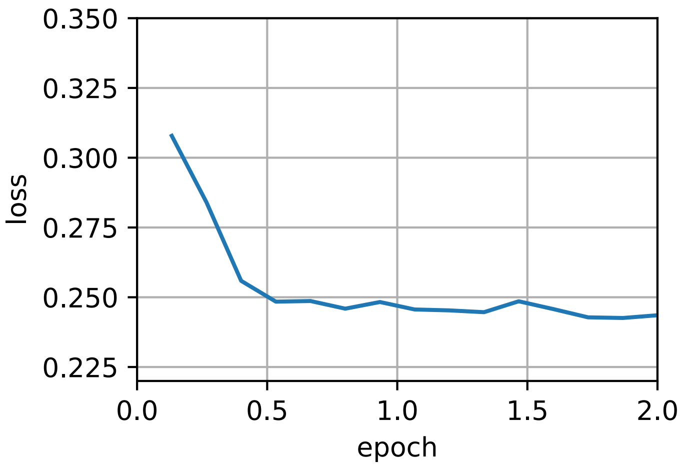
In Gluon, we can use the Trainer class to call optimization algorithms. This is used to implement a generic training function. We will use this throughout the current chapter.

```
# Saved in the d2l package for later use
def train_gluon_ch11(tr_name, hyperparams, data_iter, num_epochs=2):
    # Initialization
    net = nn.Sequential()
    net.add(nn.Dense(1))
    net.initialize(init.Normal(sigma=0.01))
    trainer = gluon.Trainer(net.collect_params(), tr_name, hyperparams)
    loss = gluon.loss.L2Loss()
    animator = d2l.Animator(xlabel='epoch', ylabel='loss',
                            xlim=[0, num_epochs], ylim=[0.22, 0.35])
    n, timer = 0, d2l.Timer()
    for _ in range(num_epochs):
        for X, y in data_iter:
            with autograd.record():
                l = loss(net(X), y)
            l.backward()
            trainer.step(X.shape[0])
            n += X.shape[0]
            if n % 200 == 0:
                timer.stop()
                animator.add(n/X.shape[0]/len(data_iter),
                             (d2l.evaluate_loss(net, data_iter, loss),))
                timer.start()
    print('loss: %.3f, %.3f sec/epoch' % (animator.Y[0][-1], timer.avg()))
```

Using Gluon to repeat the last experiment shows identical behavior.

```
data_iter, _ = get_data_ch11(10)
train_gluon_ch11('sgd', {'learning_rate': 0.05}, data_iter)
```

```
loss: 0.244, 0.035 sec/epoch
```



Summary

- Vectorization makes code more efficient due to reduced overhead arising from the deep learning framework and due to better memory locality and caching on CPUs and GPUs.
- There is a trade-off between statistical efficiency arising from SGD and computational efficiency arising from processing large batches of data at a time.
- Minibatch stochastic gradient descent offers the best of both worlds: computational and statistical efficiency.
- In minibatch SGD we process batches of data obtained by a random permutation of the training data (i.e., each observation is processed only once per epoch, albeit in random order).
- It is advisable to decay the learning rates during training.
- In general, minibatch SGD is faster than SGD and gradient descent for convergence to a smaller risk, when measured in terms of clock time.

Exercises

1. Modify the batch size and learning rate and observe the rate of decline for the value of the objective function and the time consumed in each epoch.
2. Read the MXNet documentation and use the `Trainer` class `set_learning_rate` function to reduce the learning rate of the minibatch SGD to 1/10 of its previous value after each epoch.
3. Compare minibatch SGD with a variant that actually *samples with replacement* from the training set. What happens?
4. An evil genie replicates your dataset without telling you (i.e., each observation occurs twice and your dataset grows to twice its original size, but nobody told you). How does the behavior of SGD, minibatch SGD and that of gradient descent change?

11.6 Momentum

In Section 11.4 we reviewed what happens when performing stochastic gradient descent, i.e., when performing optimization where only a noisy variant of the gradient is available. In particular, we noticed that for noisy gradients we need to be extra cautious when it comes to choosing the learning rate in the face of noise. If we decrease it too rapidly, convergence stalls. If we are too lenient, we fail to converge to a good enough solution since noise keeps on driving us away from optimality.

11.6.1 Basics

In this section, we will explore more effective optimization algorithms, especially for certain types of optimization problems that are common in practice.

Leaky Averages

The previous section saw us discussing minibatch SGD as a means for accelerating computation. It also had the nice side-effect that averaging gradients reduced the amount of variance.

$$\mathbf{g}_t = \partial_{\mathbf{w}} \frac{1}{|\mathcal{B}_t|} \sum_{i \in \mathcal{B}_t} f(\mathbf{x}_i, \mathbf{w}_{t-1}) = \frac{1}{|\mathcal{B}_t|} \sum_{i \in \mathcal{B}_t} \mathbf{g}_{i,t-1}. \tag{11.6.1}$$

Here we used $\mathbf{g}_{ii} = \partial_{\mathbf{w}} f(\mathbf{x}_i, \mathbf{w}_t)$ to keep the notation simple. It would be nice if we could benefit from the effect of variance reduction even beyond averaging gradients on a mini-batch. One option to accomplish this task is to replace the gradient computation by a "leaky average":

$$\mathbf{v}_t = \beta \mathbf{v}_{t-1} + \mathbf{g}_{t,t-1} \tag{11.6.2}$$

for some $\beta \in (0, 1)$. This effectively replaces the instantaneous gradient by one that's been averaged over multiple *past* gradients. $\mathbf{v}$ is called *momentum*. It accumulates past gradients similar to how a heavy ball rolling down the objective function landscape integrates over past forces. To see what is happening in more detail let's expand $\mathbf{v}_t$ recursively into

$$\mathbf{v}_t = \beta^2 \mathbf{v}_{t-2} + \beta \mathbf{g}_{t-1,t-2} + \mathbf{g}_{t,t-1} = \ldots = \sum_{\tau=0}^{t-1} \beta^{\tau} \mathbf{g}_{t-\tau,t-\tau-1}. \tag{11.6.3}$$

Large β amounts to a long-range average, whereas small β amounts to only a slight correction relative to a gradient method. The new gradient replacement no longer points into the direction of steepest descent on a particular instance any longer but rather in the direction of a weighted average of past gradients. This allows us to realize most of the benefits of averaging over a batch without the cost of actually computing the gradients on it. We will revisit this averaging procedure in more detail later.

The above reasoning formed the basis for what is now known as *accelerated* gradient methods, such as gradients with momentum. They enjoy the additional benefit of being much more effective in cases where the optimization problem is ill-conditioned (i.e., where there are some directions where progress is much slower than in others, resembling a narrow canyon). Furthermore, they allow us to average over subsequent gradients to obtain more stable directions of descent. Indeed, the aspect of acceleration even for noise-free convex problems is one of the key reasons why momentum works and why it works so well.

As one would expect, due to its efficacy momentum is a well-studied subject in optimization for deep learning and beyond. See e.g., the beautiful expository article[156] by (Goh, 2017) for an in-depth analysis and interactive animation. It was proposed by (Polyak, 1964). (Nesterov, 2018) has a detailed theoretical discussion in the context of convex optimization. Momentum in deep learning has been known to be beneficial for a long time. See e.g., the discussion by (Sutskever et al., 2013) for details.

[156] https://distill.pub/2017/momentum/

An Ill-conditioned Problem

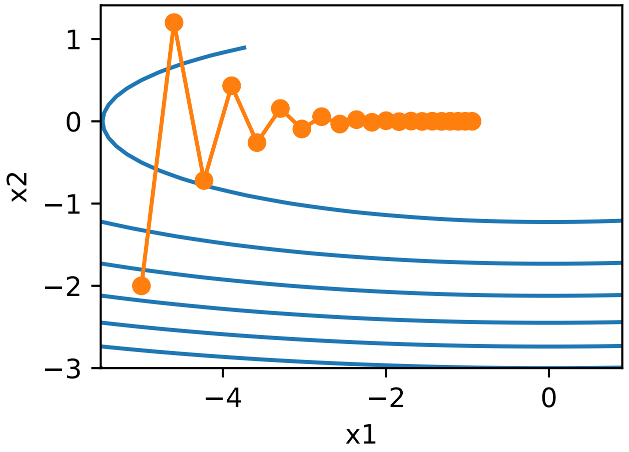
To get a better understanding of the geometric properties of the momentum method we revisit gradient descent, albeit with a significantly less pleasant objective function. Recall that in Section 11.3 we used $f(\mathbf{x}) = x_1^2 + 2x_2^2$, i.e., a moderately distorted ellipsoid objective. We distort this function further by stretching it out in the x_1 direction via

$$f(\mathbf{x}) = 0.1x_1^2 + 2x_2^2. \tag{11.6.4}$$

As before f has its minimum at $(0, 0)$. This function is *very* flat in the direction of x_1. Let's see what happens when we perform gradient descent as before on this new function. We pick a learning rate of 0.4.

```
%matplotlib inline
import d2l
from mxnet import np, npx
npx.set_np()

eta = 0.4
def f_2d(x1, x2):
    return 0.1 * x1 ** 2 + 2 * x2 ** 2
def gd_2d(x1, x2, s1, s2):
    return (x1 - eta * 0.2 * x1, x2 - eta * 4 * x2, 0, 0)

d2l.show_trace_2d(f_2d, d2l.train_2d(gd_2d))
```

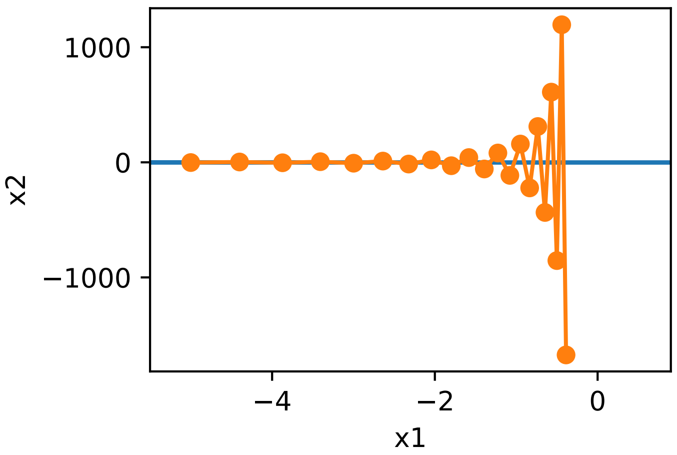
```
epoch 20, x1 -0.943467, x2 -0.000073
```



By construction, the gradient in the x_2 direction is *much* higher and changes much more rapidly than in the horizontal x_1 direction. Thus we are stuck between two undesirable choices: if we pick a small learning rate we ensure that the solution does not diverge in the x_2 direction but we're saddled with slow convergence in the x_1 direction. Conversely, with a large learning rate we progress rapidly in the x_1 direction but diverge in x_2. The example below illustrates what happens even after a slight increase in learning rate from 0.4 to 0.6. Convergence in the x_1 direction improves but the overall solution quality is much worse.

```
eta = 0.6
d2l.show_trace_2d(f_2d, d2l.train_2d(gd_2d))
```

```
epoch 20, x1 -0.387814, x2 -1673.365109
```



The Momentum Method

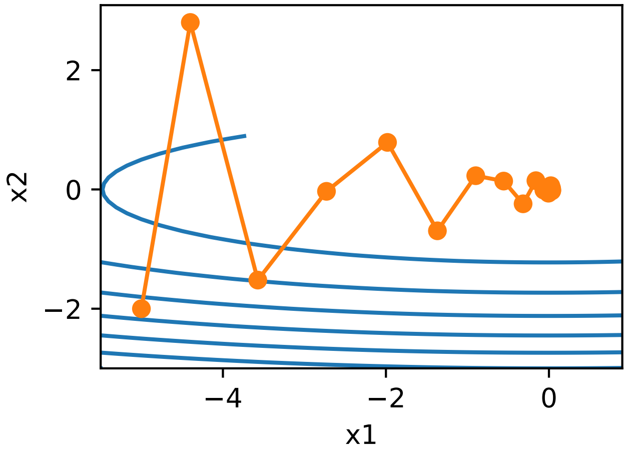
The momentum method allows us to solve the gradient descent problem described above. Looking at the optimization trace above we might intuit that averaging gradients over the past would work well. After all, in the x_1 direction this will aggregate well-aligned gradients, thus increasing the distance we cover with every step. Conversely, in the x_2 direction where gradients oscillate, an aggregate gradient will reduce step size due to oscillations that cancel each other out. Using $\mathbf{v}_t$ instead of the gradient $\mathbf{g}_t$ yields the following update equations:

$$\begin{aligned} \mathbf{v}_t &\leftarrow \beta \mathbf{v}_{t-1} + \mathbf{g}_{t, t-1}, \\ \mathbf{x}_t &\leftarrow \mathbf{x}_{t-1} - \eta_t \mathbf{v}_t. \end{aligned} \tag{11.6.5}$$

Note that for $\beta = 0$ we recover regular gradient descent. Before delving deeper into the mathematical properties let's have a quick look at how the algorithm behaves in practice.

```
def momentum_2d(x1, x2, v1, v2):
    v1 = beta * v1 + 0.2 * x1
    v2 = beta * v2 + 4 * x2
    return x1 - eta * v1, x2 - eta * v2, v1, v2

eta, beta = 0.6, 0.5
d2l.show_trace_2d(f_2d, d2l.train_2d(momentum_2d))
```

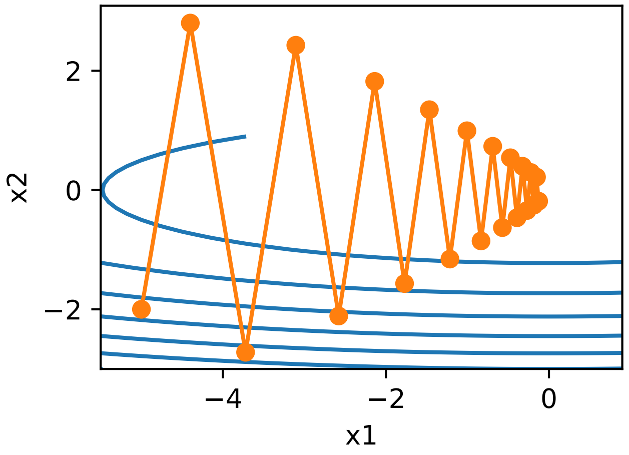
```
epoch 20, x1 0.007188, x2 0.002553
```



As we can see, even with the same learning rate that we used before, momentum still converges well. Let's see what happens when we decrease the momentum parameter. Halving it to $\beta = 0.25$ leads to a trajectory that barely converges at all. Nonetheless, it's a lot better than without momentum (when the solution diverges).

```
eta, beta = 0.6, 0.25
d2l.show_trace_2d(f_2d, d2l.train_2d(momentum_2d))
```

```
epoch 20, x1 -0.126340, x2 -0.186632
```

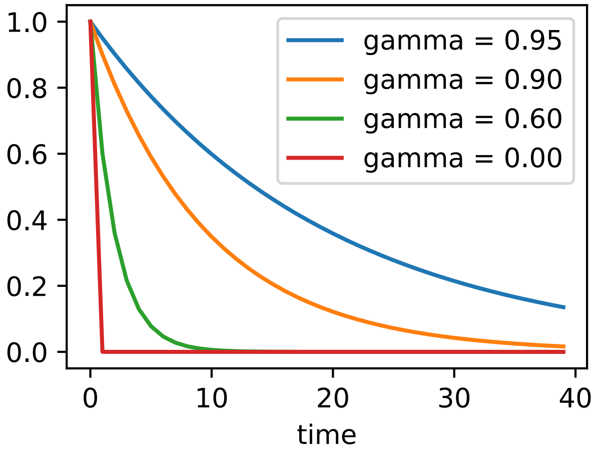


Note that we can combine momentum with SGD and in particular, minibatch-SGD. The only change is that in that case we replace the gradients $\mathbf{g}_{t,t-1}$ with $\mathbf{g}_t$. Last, for convenience we initialize $\mathbf{v}_0 = 0$ at time $t = 0$. Let's look at what leaky averaging actually does to the updates.

Effective Sample Weight

Recall that $\mathbf{v}_t = \sum_{\tau=0}^{t-1} \beta^{\tau} \mathbf{g}_{t-\tau,t-\tau-1}$. In the limit the terms add up to $\sum_{\tau=0}^{\infty} \beta^{\tau} = \frac{1}{1-\beta}$. In other words, rather than taking a step of size η in GD or SGD we take a step of size $\frac{\eta}{1-\beta}$ while at the same time, dealing with a potentially much better behaved descent direction. These are two benefits in one. To illustrate how weighting behaves for different choices of β consider the diagram below.

```
gammas = [0.95, 0.9, 0.6, 0]
d2l.set_figsize((3.5, 2.5))
for gamma in gammas:
    x = np.arange(40).asnumpy()
    d2l.plt.plot(x, gamma ** x, label='gamma = %.2f' % gamma)
d2l.plt.xlabel('time')
d2l.plt.legend();
```



11.6.2 Practical Experiments

Let's see how momentum works in practice, i.e., when used within the context of a proper optimizer. For this we need a somewhat more scalable implementation.

Implementation from Scratch

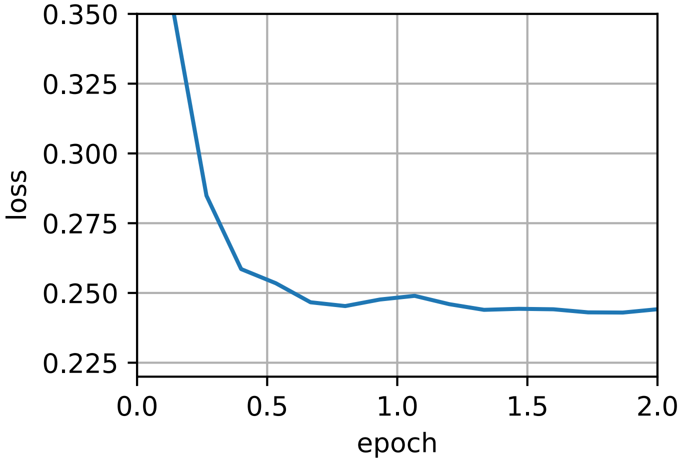
Compared with (minibatch) SGD the momentum method needs to maintain a set of auxiliary variables, i.e., velocity. It has the same shape as the gradients (and variables of the optimization problem). In the implementation below we call these variables `states`.

```
def init_momentum_states(feature_dim):
    v_w = np.zeros((feature_dim, 1))
    v_b = np.zeros(1)
    return (v_w, v_b)

def sgd_momentum(params, states, hyperparams):
    for p, v in zip(params, states):
        v[:] = hyperparams['momentum'] * v + p.grad
        p[:] -= hyperparams['lr'] * v
```

Let's see how this works in practice.

```
def train_momentum(lr, momentum, num_epochs=2):
    d2l.train_ch11(sgd_momentum, init_momentum_states(feature_dim),
                   {'lr': lr, 'momentum': momentum}, data_iter,
                   feature_dim, num_epochs)

data_iter, feature_dim = d2l.get_data_ch11(batch_size=10)
train_momentum(0.02, 0.5)
```

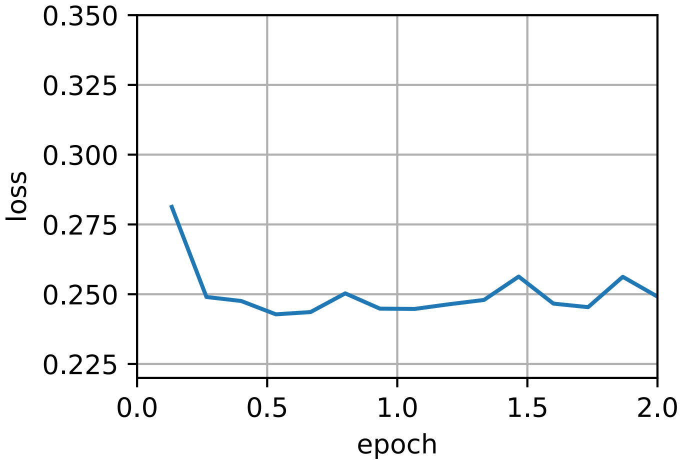
```
loss: 0.244, 0.053 sec/epoch
```



When we increase the momentum hyperparameter `momentum` to 0.9, it amounts to a significantly larger effective sample size of $\frac{1}{1-0.9} = 10$. We reduce the learning rate slightly to 0.01 to keep matters under control.

```
train_momentum(0.01, 0.9)
```

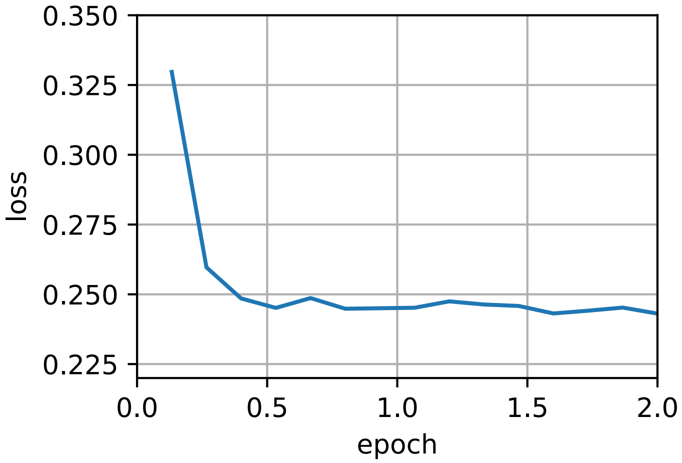
```
loss: 0.249, 0.054 sec/epoch
```



Reducing the learning rate further addresses any issue of non-smooth optimization problems. Setting it to 0.005 yields good convergence properties.

```
train_momentum(0.005, 0.9)
```

```
loss: 0.243, 0.053 sec/epoch
```

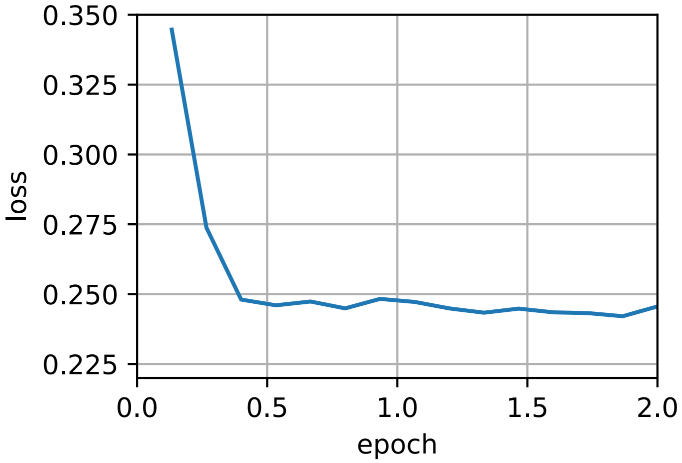


Concise Implementation

There's very little to do in Gluon since the standard sgd solver already had momentum built in. Setting matching parameters yields a very similar trajectory.

```
d2l.train_gluon_ch11('sgd', {'learning_rate': 0.005, 'momentum': 0.9},
                     data_iter)
```

```
loss: 0.246, 0.035 sec/epoch
```



11.6.3 Theoretical Analysis

So far the 2D example of $f(x) = 0.1x_1^2 + 2x_2^2$ seemed rather contrived. We will now see that this is actually quite representative of the types of problem one might encounter, at least in the case of minimizing convex quadratic objective functions.

Quadratic Convex Functions

Consider the function

$$h(\mathbf{x}) = \frac{1}{2}\mathbf{x}^\top \mathbf{Q}\mathbf{x} + \mathbf{x}^\top \mathbf{c} + b. \tag{11.6.6}$$

This is a general quadratic function. For positive semidefinite matrices $\mathbf{Q} \succ 0$, i.e., for matrices with positive eigenvalues this has a minimizer at $\mathbf{x}^* = -\mathbf{Q}^{-1}\mathbf{c}$ with minimum value $b - \frac{1}{2}\mathbf{c}^\top \mathbf{Q}^{-1}\mathbf{c}$. Hence we can rewrite h as

$$h(\mathbf{x}) = \frac{1}{2}(\mathbf{x} - \mathbf{Q}^{-1}\mathbf{c})^\top \mathbf{Q}(\mathbf{x} - \mathbf{Q}^{-1}\mathbf{c}) + b - \frac{1}{2}\mathbf{c}^\top \mathbf{Q}^{-1}\mathbf{c}. \tag{11.6.7}$$

The gradient is given by $\partial_{\mathbf{x}} f(\mathbf{x}) = \mathbf{Q}(\mathbf{x} - \mathbf{Q}^{-1}\mathbf{c})$. That is, it is given by the distance between $\mathbf{x}$ and the minimizer, multiplied by $\mathbf{Q}$. Consequently also the momentum is a linear combination of terms $\mathbf{Q}(\mathbf{x}_t - \mathbf{Q}^{-1}\mathbf{c})$.

Since $\mathbf{Q}$ is positive definite it can be decomposed into its eigensystem via $\mathbf{Q} = \mathbf{O}^\top \boldsymbol{\Lambda}\mathbf{O}$ for an orthogonal (rotation) matrix $\mathbf{O}$ and a diagonal matrix $\boldsymbol{\Lambda}$ of positive eigenvalues. This allows us to perform a change of variables from $\mathbf{x}$ to $\mathbf{z} := \mathbf{O}(\mathbf{x} - \mathbf{Q}^{-1}\mathbf{c})$ to obtain a much simplified expression:

$$h(\mathbf{z}) = \frac{1}{2}\mathbf{z}^\top \boldsymbol{\Lambda}\mathbf{z} + b'. \tag{11.6.8}$$

Here $c' = b - \frac{1}{2}\mathbf{c}^\top \mathbf{Q}^{-1}\mathbf{c}$. Since $\mathbf{O}$ is only an orthogonal matrix this doesn't perturb the gradients in a meaningful way. Expressed in terms of $\mathbf{z}$ gradient descent becomes

$$\mathbf{z}_t = \mathbf{z}_{t-1} - \boldsymbol{\Lambda}\mathbf{z}_{t-1} = (\mathbf{I} - \boldsymbol{\Lambda})\mathbf{z}_{t-1}. \tag{11.6.9}$$

The important fact in this expression is that gradient descent *does not mix* between different eigenspaces. That is, when expressed in terms of the eigensystem of $\mathbf{Q}$ the optimization problem proceeds in a coordinate-wise manner. This also holds for momentum.

$$\begin{aligned}
\mathbf{v}_t & = \beta \mathbf{v}_{t-1} + \boldsymbol{\Lambda}\mathbf{z}_{t-1} \\
\mathbf{z}_t & = \mathbf{z}_{t-1} - \eta \left(\beta \mathbf{v}_{t-1} + \boldsymbol{\Lambda}\mathbf{z}_{t-1}\right) \\
& = (\mathbf{I} - \eta\boldsymbol{\Lambda})\mathbf{z}_{t-1} - \eta\beta\mathbf{v}_{t-1}.
\end{aligned} \tag{11.6.10}$$

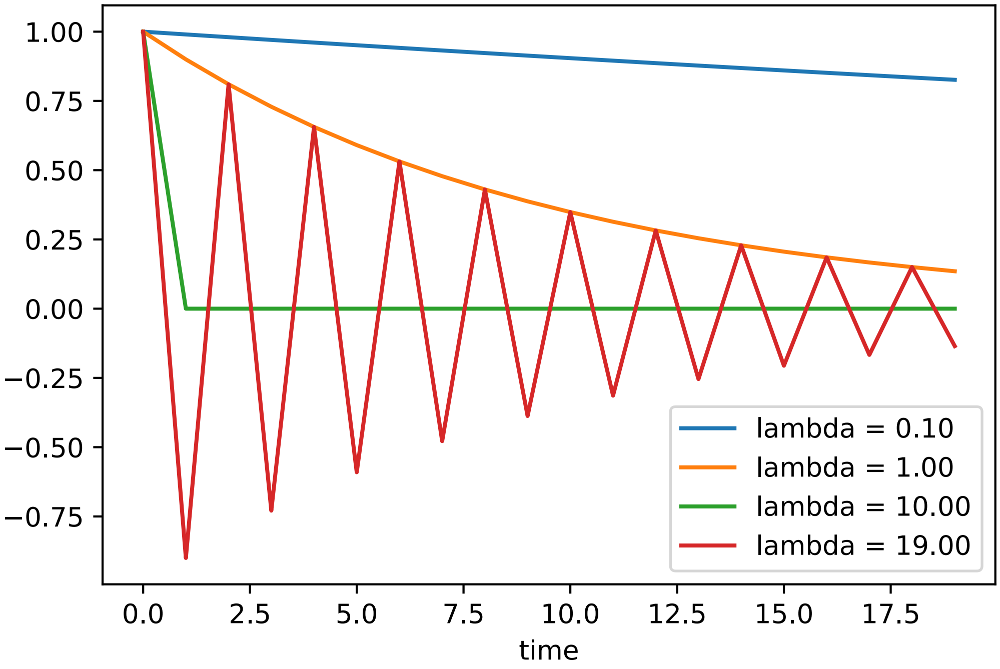
In doing this we just proved the following theorem: Gradient Descent with and without momentum for a convex quadratic function decomposes into coordinate-wise optimization in the direction of the eigenvectors of the quadratic matrix.

Given the above result let's see what happens when we minimize the function $f(x) = \frac{\lambda}{2}x^2$. For gradient descent we have

$$x_{t+1} = x_t - \eta\lambda x_t = (1 - \eta\lambda)x_t. \tag{11.6.11}$$

Whenever $|1 - \eta\lambda| < 1$ this optimization converges at an exponential rate since after t steps we have $x_t = (1 - \eta\lambda)^t x_0$. This shows how the rate of convergence improves initially as we increase the learning rate η until $\eta\lambda = 1$. Beyond that things diverge and for $\eta\lambda > 2$ the optimization problem diverges.

```
lambdas = [0.1, 1, 10, 19]
eta = 0.1
d2l.set_figsize((6, 4))
for lam in lambdas:
    t = np.arange(20).asnumpy()
    d2l.plt.plot(t, (1 - eta * lam) ** t, label='lambda = %.2f' % lam)
d2l.plt.xlabel('time')
d2l.plt.legend();
```



To analyze convergence in the case of momentum we begin by rewriting the update equations in terms of two scalars: one for x and one for the momentum v. This yields:

$$\begin{bmatrix} v_{t+1} \\ x_{t+1} \end{bmatrix} = \begin{bmatrix} \beta & \lambda \\ -\eta\beta & (1 - \eta\lambda) \end{bmatrix} \begin{bmatrix} v_t \\ x_t \end{bmatrix} = \mathbf{R}(\beta, \eta, \lambda) \begin{bmatrix} v_t \\ x_t \end{bmatrix}. \tag{11.6.12}$$

We used $\mathbf{R}$ to denote the 2×2 governing convergence behavior. After t steps the initial choice $[v_0, x_0]$ becomes $\mathbf{R}(\beta, \eta, \lambda)^t[v_0, x_0]$. Hence, it is up to the eigenvalues of $\mathbf{R}$ to detmine the speed of convergence. See the Distill post[157] of (Goh, 2017) for a great animation and (Flammarion & Bach,

[157] https://distill.pub/2017/momentum/

2015) for a detailed analysis. One can show that $0 < \eta\lambda < 2 + 2\beta$ momentum converges. This is a larger range of feasible parameters when compared to $0 < \eta\lambda < 2$ for gradient descent. It also suggests that in general large values of β are desirable. Further details require a fair amount of technical detail and we suggest that the interested reader consult the original publications.

Summary

- Momentum replaces gradients with a leaky average over past gradients. This accelerates convergence significantly.
- It is desirable for both noise-free gradient descent and (noisy) stochastic gradient descent.
- Momentum prevents stalling of the optimization process that is much more likely to occur for stochastic gradient descent.
- The effective number of gradients is given by $\frac{1}{1-\beta}$ due to exponentiated downweighting of past data.
- In the case of convex quadratic problems this can be analyzed explicitly in detail.
- Implementation is quite straightforward but it requires us to store an additional state vector (momentum $\mathbf{v}$).

Exercises

1. Use other combinations of momentum hyperparameters and learning rates and observe and analyze the different experimental results.
2. Try out GD and momentum for a quadratic problem where you have multiple eigenvalues, i.e., $f(x) = \frac{1}{2} \sum_i \lambda_i x_i^2$, e.g., $\lambda_i = 2^{-i}$. Plot how the values of x decrease for the initialization $x_i = 1$.
3. Derive minimum value and minimizer for $h(\mathbf{x}) = \frac{1}{2}\mathbf{x}^\top \mathbf{Q}\mathbf{x} + \mathbf{x}^\top \mathbf{c} + b$.
4. What changes when we perform SGD with momentum? What happens when we use minibatch SGD with momentum? Experiment with the parameters?

11.7 Adagrad

Let us begin by considering learning problems with features that occur infrequently.

11.7.1 Sparse Features and Learning Rates

Imagine that we're training a language model. To get good accuracy we typically want to decrease the learning rate as we keep on training, usually at a rate of $\mathcal{O}(t^{-\frac{1}{2}})$ or slower. Now consider a model training on sparse features, i.e., features that occur only infrequently. This is common for natural language, e.g., it is a lot less likely that we'll see the word *preconditioning* than *learning*. However, it is also common in other areas such as computational advertising and personalized collaborative filtering. After all, there are many things that are of interest only for a small number of people.

Parameters associated with infrequent features only receive meaningful updates whenever these features occur. Given a decreasing learning rate we might end up in a situation where the parameters for common features converge rather quickly to their optimal values, whereas for infrequent features we are still short of observing them sufficiently frequently before their optimal values can be determined. In other words, the learning rate either decreases too slowly for frequent features or too slowly for infrequent ones.

A possible hack to redress this issue would be to count the number of times we see a particular feature and to use this as a clock for adjusting learning rates. That is, rather than choosing a learning rate of the form $\eta = \frac{\eta_0}{\sqrt{t+c}}$ we could use $\eta_i = \frac{\eta_0}{\sqrt{s(i,t)+c}}$. Here $s(i,t)$ counts the number of nonzeros for feature i that we have observed up to time t. This is actually quite easy to implement at no meaningful overhead. However, it fails whenever we don't quite have sparsity but rather just data where the gradients are often very small and only rarely large. After all, it is unclear where one would draw the line between something that qualifies as an observed feature or not.

Adagrad by (Duchi et al., 2011) addresses this by replacing the rather crude counter $s(i,t)$ by an aggregate of the squares of previously observed gradients. In particular, it uses $s(i,t+1) = s(i,t) + (\partial_i f(\mathbf{x}))^2$ as a means to adjust the learning rate. This has two benefits: first, we no longer need to decide just when a gradient is large enough. Second, it scales automatically with the magnitude of the gradients. Coordinates that routinely correspond to large gradients are scaled down significantly, whereas others with small gradients receive a much more gentle treatment. In practice this leads to a very effective optimization procedure for computational advertising and related problems. But this hides some of the additional benefits inherent in Adagrad that are best understood in the context of preconditioning.

11.7.2 Preconditioning

Convex optimization problems are good for analyzing the characteristics of algorithms. After all, for most nonconvex problems it is difficult to derive meaningful theoretical guarantees, but *intuition* and *insight* often carry over. Let's look at the problem of minimizing $f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^\top \mathbf{Q}\mathbf{x} + \mathbf{c}^\top \mathbf{x} + b$.

As we saw in Section 11.6, it is possible to rewrite this problem in terms of its eigendecomposition $\mathbf{Q} = \mathbf{U}^\top \boldsymbol{\Lambda} \mathbf{U}$ to arrive at a much simplified problem where each coordinate can be solved individually:

$$f(\mathbf{x}) = \bar{f}(\bar{\mathbf{x}}) = \frac{1}{2}\bar{\mathbf{x}}^\top \boldsymbol{\Lambda} \bar{\mathbf{x}} + \bar{\mathbf{c}}^\top \bar{\mathbf{x}} + b. \tag{11.7.1}$$

Here we used $\mathbf{x} = \mathbf{U}\mathbf{x}$ and consequently $\mathbf{c} = \mathbf{U}\mathbf{c}$. The modified problem has as its minimizer $\bar{\mathbf{x}} = -\boldsymbol{\Lambda}^{-1}\bar{\mathbf{c}}$ and minimum value $-\frac{1}{2}\bar{\mathbf{c}}^\top \boldsymbol{\Lambda}^{-1}\bar{\mathbf{c}} + b$. This is much easier to compute since $\boldsymbol{\Lambda}$ is a diagonal matrix containing the eigenvalues of $\mathbf{Q}$.

If we perturb $\mathbf{c}$ slightly we would hope to find only slight changes in the minimizer of f. Unfortunately this is not the case. While slight changes in $\mathbf{c}$ lead to equally slight changes in $\bar{\mathbf{c}}$, this is not the case for the minimizer of f (and of $\bar{f}$ respectively). Whenever the eigenvalues $\boldsymbol{\Lambda}_i$ are large we will see only small changes in $\bar{x}_i$ and in the minimum of $\bar{f}$. Conversely, for small $\boldsymbol{\Lambda}_i$ changes in $\bar{x}_i$ can be dramatic. The ratio between the largest and the smallest eigenvalue is called the condition number of an optimization problem.

$$\kappa = \frac{\boldsymbol{\Lambda}_1}{\boldsymbol{\Lambda}_d}. \tag{11.7.2}$$

If the condition number κ is large, it is difficult to solve the optimization problem accurately. We need to ensure that we are careful in getting a large dynamic range of values right. Our analysis leads to an obvious, albeit somewhat naive question: couldn't we simply "fix" the problem by distorting the space such that all eigenvalues are 1. In theory this is quite easy: we only need the eigenvalues and eigenvectors of $\mathbf{Q}$ to rescale the problem from $\mathbf{x}$ to one in $\mathbf{z} := \boldsymbol{\Lambda}^{\frac{1}{2}} \mathbf{U}\mathbf{x}$. In the new coordinate system $\mathbf{x}^\top \mathbf{Q}\mathbf{x}$ could be simplified to $\|\mathbf{z}\|^2$. Alas, this is a rather impractical suggestion. Computing eigenvalues and eigenvectors is in general *much more* expensive than solving the actual problem.

While computing eigenvalues exactly might be expensive, guessing them and computing them even somewhat approximately may already be a lot better than not doing anything at all. In particular, we could use the diagonal entries of $\mathbf{Q}$ and rescale it accordingly. This is *much* cheaper than computing eigenvalues.

$$\tilde{\mathbf{Q}} = \mathrm{diag}^{-\frac{1}{2}}(\mathbf{Q}) \mathbf{Q} \mathrm{diag}^{-\frac{1}{2}}(\mathbf{Q}). \tag{11.7.3}$$

In this case we have $\tilde{\mathbf{Q}}_{ij} = \mathbf{Q}_{ij} / \sqrt{\mathbf{Q}_{ii} \mathbf{Q}_{jj}}$ and specifically $\tilde{\mathbf{Q}}_{ii} = 1$ for all i. In most cases this simplifies the condition number considerably. For instance, the the cases we discussed previously, this would entirely eliminate the problem at hand since the problem is axis aligned.

Unfortunately we face yet another problem: in deep learning we typically don't even have access to the second derivative of the objective function: for $\mathbf{x} \in \mathbb{R}^d$ the second derivative even on a minibatch may require $\mathcal{O}(d^2)$ space and work to compute, thus making it practically infeasible. The ingenious idea of Adagrad is to use a proxy for that elusive diagonal of the Hessian that is both relatively cheap to compute and effective—the magnitude of the gradient itself.

In order to see why this works, let's look at $\bar{f}(\bar{\mathbf{x}})$. We have that

$$\partial_{\bar{\mathbf{x}}} \bar{f}(\bar{\mathbf{x}}) = \boldsymbol{\Lambda} \bar{\mathbf{x}} + \bar{\mathbf{c}} = \boldsymbol{\Lambda} \left(\bar{\mathbf{x}} - \bar{\mathbf{x}}_0\right), \tag{11.7.4}$$

where $\bar{\mathbf{x}}_0$ is the minimizer of $\bar{f}$. Hence the magnitude of the gradient depends both on $\boldsymbol{\Lambda}$ and the distance from optimality. If $\bar{\mathbf{x}} - \bar{\mathbf{x}}_0$ didn't change, this would be all that's needed. After all, in this case the magnitude of the gradient $\partial_{\bar{\mathbf{x}}} \bar{f}(\bar{\mathbf{x}})$ suffices. Since AdaGrad is a stochastic gradient descent algorithm, we will see gradients with nonzero variance even at optimality. As a result we can safely use the variance of the gradients as a cheap proxy for the scale of the Hessian. A thorough analysis is beyond the scope of this section (it would be several pages). We refer the reader to (Duchi et al., 2011) for details.

11.7.3 The Algorithm

Let's formalize the discussion from above. We use the variable $\mathbf{s}_t$ to accumulate past gradient variance as follows.

$$\begin{aligned}
\mathbf{g}_t & = \partial_{\mathbf{w}} l(y_t, f(\mathbf{x}_t, \mathbf{w})), \\
\mathbf{s}_t & = \mathbf{s}_{t-1} + \mathbf{g}_t^2, \\
\mathbf{w}_t & = \mathbf{w}_{t-1} - \frac{\eta}{\sqrt{\mathbf{s}_t + \epsilon}} \cdot \mathbf{g}_t.
\end{aligned} \tag{11.7.5}$$

Here the operation are applied coordinate wise. That is, $\mathbf{v}^2$ has entries v_i^2. Likewise $\frac{1}{\sqrt{v}}$ has entries $\frac{1}{\sqrt{v_i}}$ and $\mathbf{u} \cdot \mathbf{v}$ has entries $u_i v_i$. As before η is the learning rate and ϵ is an additive constant that ensures that we do not divide by 0. Last, we initialize $\mathbf{s}_0 = \mathbf{0}$.

Just like in the case of momentum we need to keep track of an auxiliary variable, in this case to allow for an individual learning rate per coordinate. This doesn't increase the cost of Adagrad significantly relative to SGD, simply since the main cost is typically to compute $l(y_t, f(\mathbf{x}_t, \mathbf{w}))$ and its derivative.

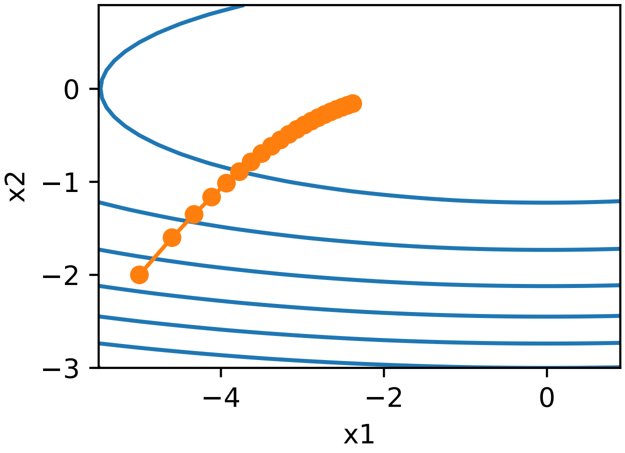
Note that accumulating squared gradients in $\mathbf{s}_t$ means that $\mathbf{s}_t$ grows essentially at linear rate (somewhat slower than linearly in practice, since the gradients initially diminish). This leads to an $\mathcal{O}(t^{-\frac{1}{2}})$ learning rate, albeit adjusted on a per coordinate basis. For convex problems this is perfectly adequate. In deep learning, though, we might want to decrease the learning rate rather more slowly. This led to a number of Adagrad variants that we will discuss in the subsequent chapters. For now let's see how it behaves in a quadratic convex problem. We use the same problem as before:

$$f(\mathbf{x}) = 0.1 x_1^2 + 2 x_2^2. \tag{11.7.6}$$

We are going to implement Adagrad using the same learning rate previously, i.e., $\eta = 0.4$. As we can see, the iterative trajectory of the independent variable is smoother. However, due to the cumulative effect of s_t, the learning rate continuously decays, so the independent variable does not move as much during later stages of iteration.

```
%matplotlib inline
import d2l
import math
from mxnet import np, npx
npx.set_np()

def adagrad_2d(x1, x2, s1, s2):
    eps = 1e-6
    g1, g2 = 0.2 * x1, 4 * x2
    s1 += g1 ** 2
    s2 += g2 ** 2
    x1 -= eta / math.sqrt(s1 + eps) * g1
    x2 -= eta / math.sqrt(s2 + eps) * g2
    return x1, x2, s1, s2

def f_2d(x1, x2):
    return 0.1 * x1 ** 2 + 2 * x2 ** 2

eta = 0.4
d2l.show_trace_2d(f_2d, d2l.train_2d(adagrad_2d))
```

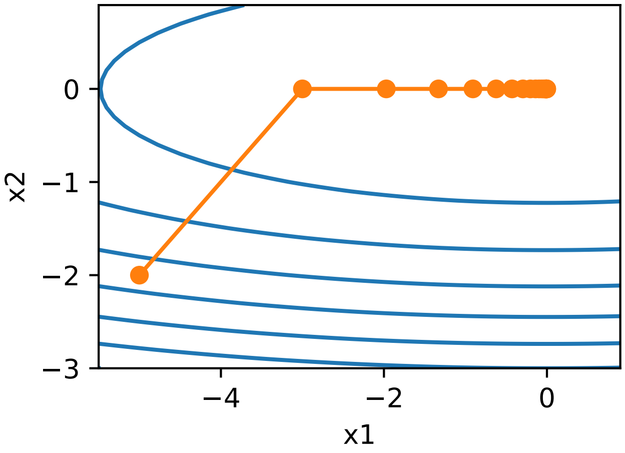
```
epoch 20, x1 -2.382563, x2 -0.158591
```



As we increase the learning rate to 2 we see much better behavior. This already indicates that the decrease in learning rate might be rather aggressive, even in the noise-free case and we need to ensure that parameters converge appropriately.

```
eta = 2
d2l.show_trace_2d(f_2d, d2l.train_2d(adagrad_2d))
```

```
epoch 20, x1 -0.002295, x2 -0.000000
```



11.7.4 Implementation from Scratch

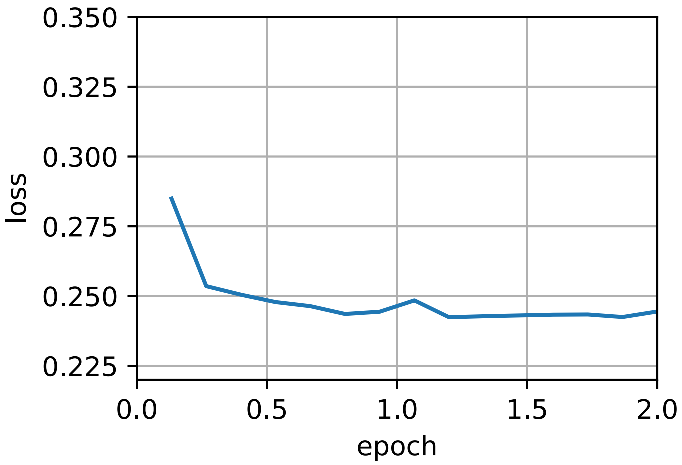
Just like the momentum method, Adagrad needs to maintain a state variable of the same shape as the parameters.

```
def init_adagrad_states(feature_dim):
    s_w = np.zeros((feature_dim, 1))
    s_b = np.zeros(1)
    return (s_w, s_b)

def adagrad(params, states, hyperparams):
    eps = 1e-6
    for p, s in zip(params, states):
        s[:] += np.square(p.grad)
        p[:] -= hyperparams['lr'] * p.grad / np.sqrt(s + eps)
```

Compared to the experiment in Section 11.5 we use a larger learning rate to train the model.

```
data_iter, feature_dim = d2l.get_data_ch11(batch_size=10)
d2l.train_ch11(adagrad, init_adagrad_states(feature_dim),
               {'lr': 0.1}, data_iter, feature_dim);
```

```
loss: 0.244, 0.052 sec/epoch
```

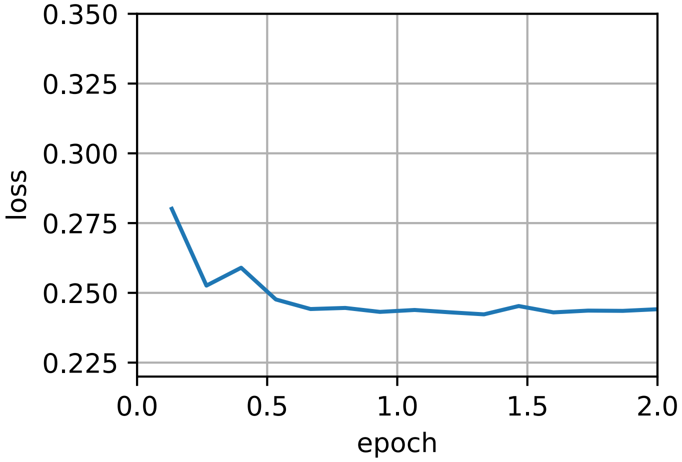


11.7.5 Concise Implementation

Using the `Trainer` instance of the algorithm `adagrad`, we can invoke the Adagrad algorithm in Gluon.

```
d2l.train_gluon_ch11('adagrad', {'learning_rate': 0.1}, data_iter)
```

```
loss: 0.244, 0.069 sec/epoch
```



Summary

- Adagrad decreases the learning rate dynamically on a per-coordinate basis.
- It uses the magnitude of the gradient as a means of adjusting how quickly progress is achieved - coordinates with large gradients are compensated with a smaller learning rate.
- Computing the exact second derivative is typically infeasible in deep learning problems due to memory and computational constraints. The gradient can be a useful proxy.
- If the optimization problem has a rather uneven uneven structure Adagrad can help mitigate the distortion.
- Adagrad is particularly effective for sparse features where the learning rate needs to decrease more slowly for infrequently occurring terms.
- On deep learning problems Adagrad can sometimes be too aggressive in reducing learning rates. We will discuss strategies for mitigating this in the context of Section 11.10.

Exercises

1. Prove that for an orthogonal matrix $\mathbf{U}$ and a vector $\mathbf{c}$ the following holds: $\|\mathbf{c} - \delta\|_2 = \|\mathbf{U}\mathbf{c} - \mathbf{U}\delta\|_2$. Why does this mean that the magnitude of perturbations does not change after an orthogonal change of variables?
2. Try out Adagrad for $f(\mathbf{x}) = 0.1x_1^2 + 2x_2^2$ and also for the objective function was rotated by 45 degrees, i.e., $f(\mathbf{x}) = 0.1(x_1 + x_2)^2 + 2(x_1 - x_2)^2$. Does it behave differently?
3. Prove Gerschgorin's circle theorem[159] which states that eigenvalues λ_i of a matrix $\mathbf{M}$ satisfy $|\lambda_i - \mathbf{M}_{jj}| \leq \sum_{k \neq j} |\mathbf{M}_{jk}|$ for at least one choice of j.
4. What does Gerschgorin's theorem tell us about the eigenvalues of the diagonally preconditioned matrix $\mathrm{diag}^{-\frac{1}{2}}(\mathbf{M})\mathbf{M}\mathrm{diag}^{-\frac{1}{2}}(\mathbf{M})$?
5. Try out Adagrad for a proper deep network, such as Section 6.6 when applied to Fashion MNIST.
6. How would you need to modify Adagrad to achieve a less aggressive decay in learning rate?

[159] https://en.wikipedia.org/wiki/Gershgorin_circle_theorem

11.8 RMSProp

One of the key issues in Section 11.7 is that the learning rate decreases at a predefined schedule of effectively $\mathcal{O}(t^{-\frac{1}{2}})$. While this is generally appropriate for convex problems, it might not be ideal for nonconvex ones, such as those encountered in deep learning. Yet, the coordinate-wise adaptivity of Adagrad is highly desirable as a preconditioner.

(Tieleman & Hinton, 2012) proposed the RMSProp algorithm as a simple fix to decouple rate scheduling from coordinate-adaptive learning rates. The issue is that Adagrad accumulates the squares of the gradient $\mathbf{g}_t$ into a state vector $\mathbf{s}_t = \mathbf{s}_{t-1} + \mathbf{g}_t^2$. As a result $\mathbf{s}_t$ keeps on growing without bound due to the lack of normalization, essentially linarly as the algorithm converges.

One way of fixing this problem would be to use $\mathbf{s}_t / t$. For reasonable distributions of $\mathbf{g}_t$ this will converge. Unfortunately it might take a very long time until the limit behavior starts to matter since the procedure remembers the full trajectory of values. An alternative is to use a leaky average in the same way we used in the momentum method, i.e., $\mathbf{s}_t \leftarrow \gamma \mathbf{s}_{t-1} + (1-\gamma)\mathbf{g}_t^2$ for some parameter $\gamma > 0$. Keeping all other parts unchanged yields RMSProp.

11.8.1 The Algorithm

Let's write out the equations in detail.

$$\begin{aligned}
\mathbf{s}_t & \leftarrow \gamma \mathbf{s}_{t-1} + (1 - \gamma) \mathbf{g}_t^2, \\
\mathbf{x}_t & \leftarrow \mathbf{x}_{t-1} - \frac{\eta}{\sqrt{\mathbf{s}_t + \epsilon}} \odot \mathbf{g}_t.
\end{aligned} \tag{11.8.1}$$

The constant $\epsilon > 0$ is typically set to 10^{-6} to ensure that we don't suffer from division by zero or overly large step sizes. Given this expansion we are now free to control the learning rate η independently of the scaling that is applied on a per-coordinate basis. In terms of leaky averages we can apply the same reasoning as previously applied in the case of the momentum method. Expanding the definition of $\mathbf{s}_t$ yields

$$\begin{aligned}
\mathbf{s}_t & = (1 - \gamma) \mathbf{g}_t^2 + \gamma \mathbf{s}_{t-1} \\
& = (1 - \gamma) \left(\mathbf{g}_t^2 + \gamma \mathbf{g}_{t-1}^2 + \gamma^2 \mathbf{g}_{t-2} + \ldots \right).
\end{aligned} \tag{11.8.2}$$

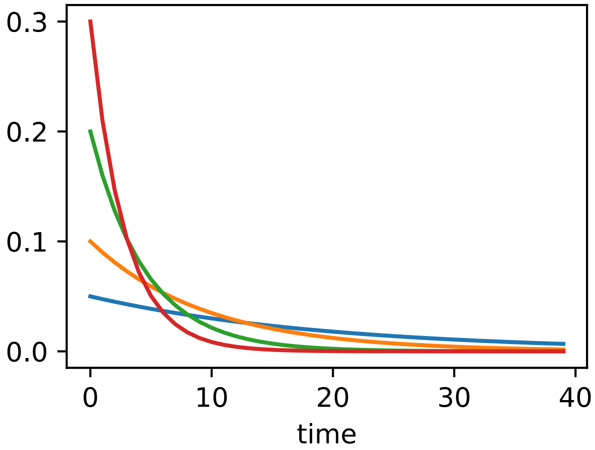
As before in Section 11.6 we use $1 + \gamma + \gamma^2 + \ldots = \frac{1}{1-\gamma}$. Hence the sum of weights is normalized to 1 with a half-life time of an observation of γ^{-1}. Let's visualize the weights for the past 40 timesteps for various choices of γ.

```
%matplotlib inline
import d2l
import math
from mxnet import np, npx
```

(continues on next page)

(continued from previous page)

```
npx.set_np()
d2l.set_figsize((3.5, 2.5))

gammas = [0.95, 0.9, 0.8, 0.7]
for gamma in gammas:
    x = np.arange(40).asnumpy()
    d2l.plt.plot(x, (1-gamma) * gamma ** x, label='gamma = %.2f' % gamma)
d2l.plt.xlabel('time');
```

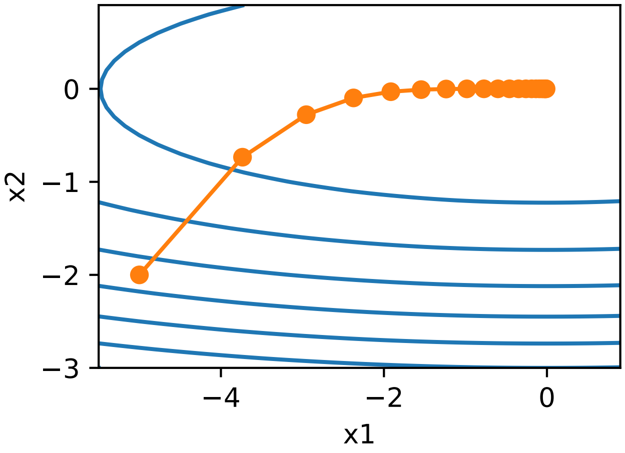


11.8.2 Implementation from Scratch

As before we use the quadratic function $f(\mathbf{x}) = 0.1x_1^2 + 2x_2^2$ to observe the trajectory of RMSProp. Recall that in Section 11.7, when we used Adagrad with a learning rate of 0.4, the variables moved only very slowly in the later stages of the algorithm since the learning rate decreased too quickly. Since η is controlled separately this does not happen with RMSProp.

```
def rmsprop_2d(x1, x2, s1, s2):
    g1, g2, eps = 0.2 * x1, 4 * x2, 1e-6
    s1 = gamma * s1 + (1 - gamma) * g1 ** 2
    s2 = gamma * s2 + (1 - gamma) * g2 ** 2
    x1 -= eta / math.sqrt(s1 + eps) * g1
    x2 -= eta / math.sqrt(s2 + eps) * g2
    return x1, x2, s1, s2

def f_2d(x1, x2):
    return 0.1 * x1 ** 2 + 2 * x2 ** 2

eta, gamma = 0.4, 0.9
d2l.show_trace_2d(f_2d, d2l.train_2d(rmsprop_2d))
```

```
epoch 20, x1 -0.010599, x2 0.000000
```



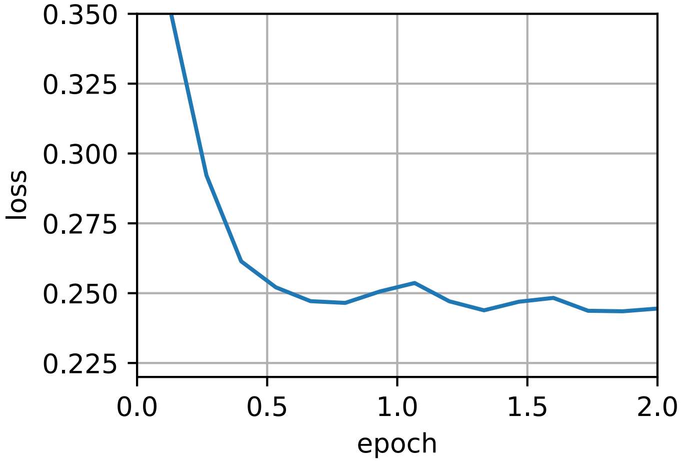
Next, we implement RMSProp to be used in a deep network. This is equally straightforward.

```
def init_rmsprop_states(feature_dim):
    s_w = np.zeros((feature_dim, 1))
    s_b = np.zeros(1)
    return (s_w, s_b)

def rmsprop(params, states, hyperparams):
    gamma, eps = hyperparams['gamma'], 1e-6
    for p, s in zip(params, states):
        s[:] = gamma * s + (1 - gamma) * np.square(p.grad)
        p[:] -= hyperparams['lr'] * p.grad / np.sqrt(s + eps)
```

We set the initial learning rate to 0.01 and the weighting term γ to 0.9. That is, **s** aggregates on average over the past $1/(1-\gamma) = 10$ observations of the square gradient.

```
data_iter, feature_dim = d2l.get_data_ch11(batch_size=10)
d2l.train_ch11(rmsprop, init_rmsprop_states(feature_dim),
               {'lr': 0.01, 'gamma': 0.9}, data_iter, feature_dim);
```

```
loss: 0.244, 0.067 sec/epoch
```

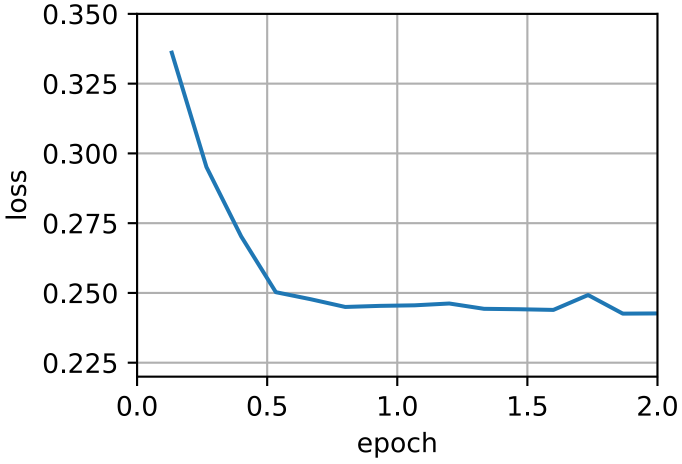


11.8.3 Concise Implementation

Since RMSProp is a rather popular algorithm it is also available in the `Trainer` instance. All we need to do is instantiate it using an algorithm named `rmsprop`, assigning γ to the parameter `gamma1`.

```
d2l.train_gluon_ch11('rmsprop', {'learning_rate': 0.01, 'gamma1': 0.9},
                     data_iter)
```

```
loss: 0.243, 0.036 sec/epoch
```



Summary

- RMSProp is very similar to Adagrad insofar as both use the square of the gradient to scale coefficients.
- RMSProp shares with momentum the leaky averaging. However, RMSProp uses the technique to adjust the coefficient-wise preconditioner.
- The learning rate needs to be scheduled by the experimenter in practice.

- The coefficient γ determines how long the history is when adjusting the per-coordinate scale.

Exercises

1. What happens experimentally if we set $\gamma = 1$? Why?
2. Rotate the optimization problem to minimize $f(\mathbf{x}) = 0.1(x_1 + x_2)^2 + 2(x_1 - x_2)^2$. What happens to the convergence?
3. Try out what happens to RMSProp on a real machine learning problem, such as training on FashionMNIST. Experiment with different choices for adjusting the learning rate.
4. Would you want to adjust γ as optimization progresses? How sensitive is RMSProp to this?

11.9 Adadelta

Adadelta is yet another variant of AdaGrad. The main difference lies in the fact that it decreases the amount by which the learning rate is adaptive to coordinates. Moreover, traditionally it referred to as not having a learning rate since it uses the amount of change itself as calibration for future change. The algorithm was proposed in (Zeiler, 2012). It is fairly straightforward, given the discussion of previous algorithms so far.

11.9.1 The Algorithm

In a nutshell Adadelta uses two state variables, $\mathbf{s}_t$ to store a leaky average of the second moment of the gradient and $\Delta\mathbf{x}_t$ to store a leaky average of the second moment of the change of parameters in the model itself. Note that we use the original notation and naming of the authors for compatibility with other publications and implementations (there's no other real reason why one should use different Greek variables to indicate a parameter serving the same purpose in momentum, Adagrad, RMSProp, and Adadelta). The parameter du jour is ρ. We obtain the following leaky updates:

$$\begin{aligned}
\mathbf{s}_t & = \rho \mathbf{s}_{t-1} + (1 - \rho) \mathbf{g}_t^2, \\
\mathbf{g}_t' & = \sqrt{\frac{\Delta\mathbf{x}_{t-1} + \epsilon}{\mathbf{s}_t + \epsilon}} \odot \mathbf{g}_t, \\
\mathbf{x}_t & = \mathbf{x}_{t-1} - \mathbf{g}_t', \\
\Delta \mathbf{x}_t & = \rho \Delta\mathbf{x}_{t-1} + (1 - \rho) \mathbf{x}_t^2.
\end{aligned} \tag{11.9.1}$$

The difference to before is that we perform updates with the rescaled gradient $\mathbf{g}_t'$ which is computed by taking the ratio between the average squared rate of change and the average second moment of the gradient. The use of $\mathbf{g}_t'$ is purely for notational convenience. In practice we can implement this algorithm without the need to use additional temporary space for $\mathbf{g}_t'$. As before η is a parameter ensuring nontrivial numerical results, i.e., avoiding zero step size or infinite variance. Typically we set this to $\eta = 10^{-5}$.

11.9.2 Implementation

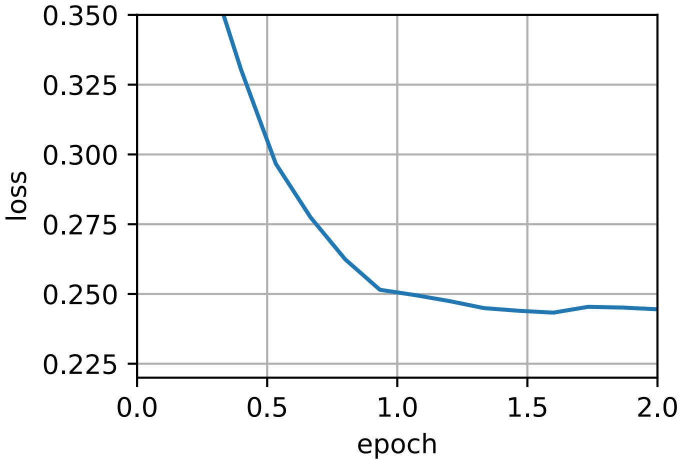
Adadelta needs to maintain two state variables for each variable, $\mathbf{s}_t$ and $\Delta\mathbf{x}_t$. This yields the following implementation.

```
%matplotlib inline
import d2l
from mxnet import np, npx
npx.set_np()

def init_adadelta_states(feature_dim):
    s_w, s_b = np.zeros((feature_dim, 1)), np.zeros(1)
    delta_w, delta_b = np.zeros((feature_dim, 1)), np.zeros(1)
    return ((s_w, delta_w), (s_b, delta_b))

def adadelta(params, states, hyperparams):
    rho, eps = hyperparams['rho'], 1e-5
    for p, (s, delta) in zip(params, states):
        # In-place updates via [:]
        s[:] = rho * s + (1 - rho) * np.square(p.grad)
        g = (np.sqrt(delta + eps) / np.sqrt(s + eps)) * p.grad
        p[:] -= g
        delta[:] = rho * delta + (1 - rho) * g * g
```

Choosing $\rho = 0.9$ amounts to a half-life time of 10 for each parameter update. This tends to work quite well. We get the following behavior.

```
data_iter, feature_dim = d2l.get_data_ch11(batch_size=10)
d2l.train_ch11(adadelta, init_adadelta_states(feature_dim),
               {'rho': 0.9}, data_iter, feature_dim);
```

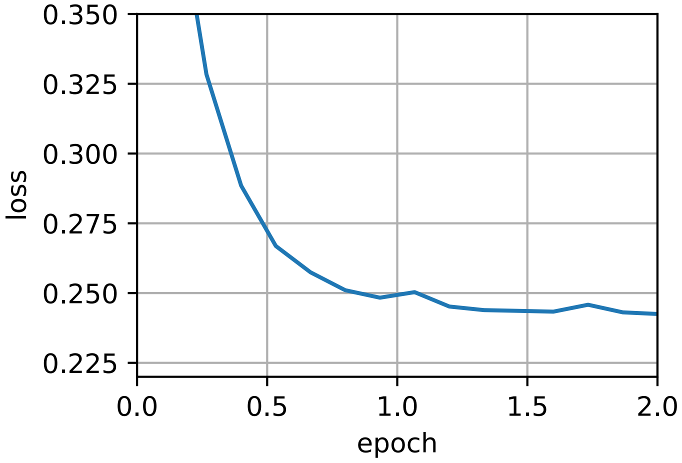
```
loss: 0.245, 0.089 sec/epoch
```



For a concise implementation we simply use the adadelta algorithm from the Trainer class. This yields the following one-liner for a much more compact invocation.

```
d2l.train_gluon_ch11('adadelta', {'rho': 0.9}, data_iter)
```

```
loss: 0.243, 0.089 sec/epoch
```



Summary

- Adadelta has no learning rate parameter. Instead, it uses the rate of change in the parameters itself to adapt the learning rate.
- Adadelta requires two state variables to store the second moments of gradient and the change in parameters.
- Adadelta uses leaky averages to keep a running estimate of the appropriate statistics.

Exercises

1. Adjust the value of ρ. What happens?
2. Show how to implement the algorithm without the use of $\mathbf{g}_t'$. Why might this be a good idea?
3. Is Adadelta really learning rate free? Could you find optimization problems that break Adadelta?
4. Compare Adadelta to Adagrad and RMS prop to discuss their convergence behavior.

11.10 Adam

In the discussions leading up to this section we encountered a number of techniques for efficient optimization. Let's recap them in detail here:

- We saw that Section 11.4 is more effective than Gradient Descent when solving optimization problems, e.g., due to its inherent resilience to redundant data.
- We saw that Section 11.5 affords significant additional efficiency arising from vectorization, using larger sets of observations in one minibatch. This is the key to efficient multi-machine, multi-GPU and overall parallel processing.
- Section 11.6 added a mechanism for aggregating a history of past gradients to accelerate convergence.
- Section 11.7 used per-coordinate scaling to allow for a computationally efficient preconditioner.
- Section 11.8 decoupled per-coordinate scaling from a learning rate adjustment.

Adam (Kingma & Ba, 2014) combines all these techniques into one efficient learning algorithm. As expected, this is an algorithm that has become rather popular as one of the more robust and effective optimization algorithms to use in deep learning. It is not without issues, though. In particular, (Reddi et al., 2019) show that there are situations where Adam can diverge due to poor variance control. In a follow-up work (Zaheer et al., 2018) proposed a hotfix to Adam, called Yogi which addresses these issues. More on this later. For now let's review the Adam algorithm.

11.10.1 The Algorithm

One of the key components of Adam is that it uses exponential weighted moving averages (also known as leaky averaging) to obtain an estimate of both the momentum and also the second moment of the gradient. That is, it uses the state variables

$$\begin{aligned}\mathbf{v}_t &\leftarrow \beta_1 \mathbf{v}_{t-1} + (1 - \beta_1) \mathbf{g}_t, \\ \mathbf{s}_t &\leftarrow \beta_2 \mathbf{s}_{t-1} + (1 - \beta_2) \mathbf{g}_t^2.\end{aligned} \tag{11.10.1}$$

Here β_1 and β_2 are nonnegative weighting parameters. Common choices for them are $\beta_1 = 0.9$ and $\beta_2 = 0.999$. That is, the variance estimate moves *much more slowly* than the momentum term. Note that if we initialize $\mathbf{v}_0 = \mathbf{s}_0 = 0$ we have a significant amount of bias initially towards smaller values. This can be addressed by using the fact that $\sum_{i=0}^t \beta^i = \frac{1 - \beta^t}{1 - \beta}$ to re-normalize terms. Correspondingly the normalized state variables are given by

$$\hat{\mathbf{v}}_t = \frac{\mathbf{v}_t}{1 - \beta_1^t} \text{ and } \hat{\mathbf{s}}_t = \frac{\mathbf{s}_t}{1 - \beta_2^t}. \tag{11.10.2}$$

Armed with the proper estimates we can now write out the update equations. First, we rescale the gradient in a manner very much akin to that of RMSProp to obtain

$$\mathbf{g}_t' = \frac{\eta \hat{\mathbf{v}}_t}{\sqrt{\hat{\mathbf{s}}_t} + \epsilon}. \tag{11.10.3}$$

Unlike RMSProp our update uses the momentum $\hat{\mathbf{v}}_t$ rather than the gradient itself. Moreover, there's a slight cosmetic difference as the rescaling happens using $\frac{1}{\sqrt{\hat{\mathbf{s}}_t} + \epsilon}$ instead of $\frac{1}{\sqrt{\hat{\mathbf{s}}_t + \epsilon}}$. The former works arguably slightly better in practice, hence the deviation from RMSProp. Typically we pick $\epsilon = 10^{-6}$ for a good trade-off between numerical stability and fidelity.

Now we have all the pieces in place to compute updates. This is slightly anticlimactic and we have a simple update of the form

$$\mathbf{x}_t \leftarrow \mathbf{x}_{t-1} - \mathbf{g}'_t. \tag{11.10.4}$$

Reviewing the design of Adam its inspiration is clear. Momentum and scale are clearly visible in the state variables. Their rather peculiar definition forces us to debias terms (this could be fixed by a slightly different initialization and update condition). Second, the combination of both terms is pretty straightforward, given RMSProp. Last, the explicit learning rate η allows us to control the step length to address issues of convergence.

11.10.2 Implementation

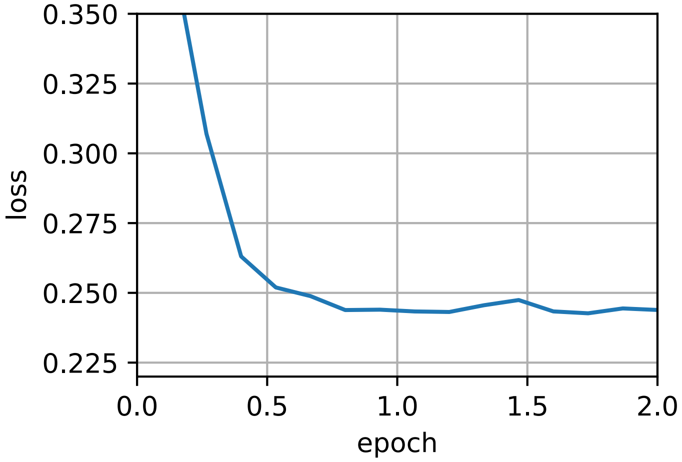
Implementing Adam from scratch isn't very daunting. For convenience we store the timestep counter t in the `hyperparams` dictionary. Beyond that all is straightforward.

```
%matplotlib inline
import d2l
from mxnet import np, npx
npx.set_np()

def init_adam_states(feature_dim):
    v_w, v_b = np.zeros((feature_dim, 1)), np.zeros(1)
    s_w, s_b = np.zeros((feature_dim, 1)), np.zeros(1)
    return ((v_w, s_w), (v_b, s_b))

def adam(params, states, hyperparams):
    beta1, beta2, eps = 0.9, 0.999, 1e-6
    for p, (v, s) in zip(params, states):
        v[:] = beta1 * v + (1 - beta1) * p.grad
        s[:] = beta2 * s + (1 - beta2) * np.square(p.grad)
        v_bias_corr = v / (1 - beta1 ** hyperparams['t'])
        s_bias_corr = s / (1 - beta2 ** hyperparams['t'])
        p[:] -= hyperparams['lr'] * v_bias_corr / (np.sqrt(s_bias_corr) + eps)
    hyperparams['t'] += 1
```

We are ready to use Adam to train the model. We use a learning rate of $\eta = 0.01$.

```
data_iter, feature_dim = d2l.get_data_ch11(batch_size=10)
d2l.train_ch11(adam, init_adam_states(feature_dim),
               {'lr': 0.01, 't': 1}, data_iter, feature_dim);
```

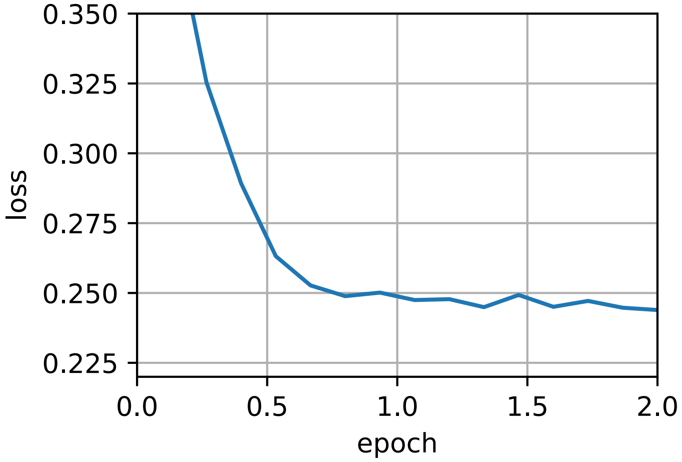
```
loss: 0.244, 0.091 sec/epoch
```



A more concise implementation is straightforward since adam is one of the algorithms provided as part of the Gluon trainer optimization library. Hence we only need to pass configuration parameters for an implementation in Gluon.

```
d2l.train_gluon_ch11('adam', {'learning_rate': 0.01}, data_iter)
```

```
loss: 0.244, 0.039 sec/epoch
```



11.10.3 Yogi

One of the problems of Adam is that it can fail to converge even in convex settings when the second moment estimate in $\mathbf{s}_t$ blows up. As a fix (Zaheer et al., 2018) proposed a refined update (and initialization) for $\mathbf{s}_t$. To understand what's going on, let's rewrite the Adam update as follows:

$$\mathbf{s}_t \leftarrow \mathbf{s}_{t-1} + (1 - \beta_2) \left(\mathbf{g}_t^2 - \mathbf{s}_{t-1}\right). \tag{11.10.5}$$

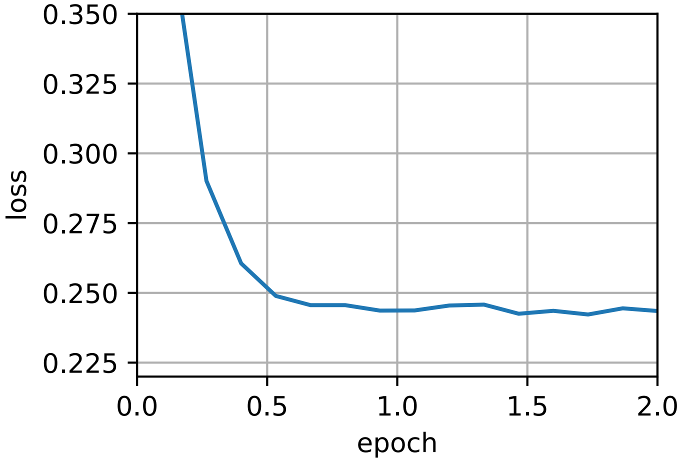
Whenever $\mathbf{g}_t^2$ has high variance or updates are sparse, $\mathbf{s}_t$ might forget past values too quickly. A possible fix for this is to replace $\mathbf{g}_t^2 - \mathbf{s}_{t-1}$ by $\mathbf{g}_t^2 \odot \mathop{\mathrm{sgn}}(\mathbf{g}_t^2 - \mathbf{s}_{t-1})$. Now the magnitude of the update no longer depends on the amount of deviation. This yields the Yogi updates

$$\mathbf{s}_t \leftarrow \mathbf{s}_{t-1} + (1 - \beta_2) \mathbf{g}_t^2 \odot \mathop{\mathrm{sgn}}(\mathbf{g}_t^2 - \mathbf{s}_{t-1}). \tag{11.10.6}$$

The authors furthermore advise to initialize the momentum on a larger initial batch rather than just initial pointwise estimate. We omit the details since they are not material to the discussion and since even without this convergence remains pretty good.

```
def yogi(params, states, hyperparams):
    beta1, beta2, eps = 0.9, 0.999, 1e-3
    for p, (v, s) in zip(params, states):
        v[:] = beta1 * v + (1 - beta1) * p.grad
        s[:] = s + (1 - beta2) * np.sign(
            np.square(p.grad) - s) * np.square(p.grad)
        v_bias_corr = v / (1 - beta1 ** hyperparams['t'])
        s_bias_corr = s / (1 - beta2 ** hyperparams['t'])
        p[:] -= hyperparams['lr'] * v_bias_corr / (np.sqrt(s_bias_corr) + eps)
    hyperparams['t'] += 1

data_iter, feature_dim = d2l.get_data_ch11(batch_size=10)
d2l.train_ch11(yogi, init_adam_states(feature_dim),
               {'lr': 0.01, 't': 1}, data_iter, feature_dim);
```

```
loss: 0.244, 0.092 sec/epoch
```



Summary

- Adam combines features of many optimization algorithms into a fairly robust update rule.
- Created on the basis of RMSProp, Adam also uses EWMA on the minibatch stochastic gradient
- Adam uses bias correction to adjust for a slow startup when estimating momentum and a second moment.
- For gradients with significant variance we may encounter issues with convergence. They can be amended by using larger minibatches or by switching to an improved estimate for $\mathbf{s}_t$. Yogi offers such an alternative.

Exercises

1. Adjust the learning rate and observe and analyze the experimental results.
2. Can you rewrite momentum and second moment updates such that it doesn't require bias correction?
3. Why do you need to reduce the learning rate η as we converge?
4. Try to construct a case for which Adam diverges and Yogi converges?

11.11 Learning Rate Scheduling

So far we primarily focused on optimization *algorithms* for how to update the weight vectors rather than on the *rate* at which they're being updated. Nonetheless, adjusting the learning rate is often just as important as the actual algorithm. There are a number of aspects to consider:

- Most obviously the *magnitude* of the learning rate matters. If it's too large, optimization diverges, if it's too small, it takes too long to train or we end up with a suboptimal result. We saw previously that the condition number of the problem matters (see e.g., Section 11.6 for details). Intuitively it's the ratio of the amount of change in the least sensitive direction vs. the most sensitive one.
- Secondly, the rate of decay is just as important. If the learning rate remains large we may simply end up bouncing around the minimum and thus not reach optimality. Section 11.5 discussed this in some detail and we analyzed performance guarantees in Section 11.4. In short, we want the rate to decay, but probably more slowly than $\mathcal{O}(t^{-\frac{1}{2}})$ which would be a good choice for convex problems.
- Another aspect that is equally important is *initialization*. This pertains both to how the parameters are set initially (review Section 4.8 for details) and also how they evolve initially. This goes under the moniker of *warmup*, i.e., how rapidly we start moving towards the solution initially. Large steps in the beginning might not be beneficial, in particular since the initial set of parameters is random. The initial update directions might be quite meaningless, too.
- Lastly, there are a number of optimization variants that perform cyclical learning rate adjustment. This is beyond the scope of the current chapter. We recommend the reader to review details in (Izmailov et al., 2018), e.g., how to obtain better solutions by averaging over an entire *path* of parameters.

Given the fact that there's a lot of detail needed to manage learning rates, most deep learning frameworks have tools to deal with this automatically. In the current chapter we will review the effects that different schedules have on accuracy and also show how this can be managed efficiently via a *learning rate scheduler*.

11.11.1 Toy Problem

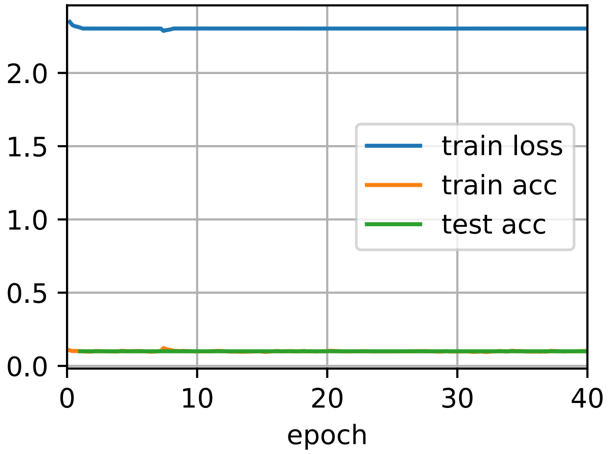
We begin with a toy problem that is cheap enough to compute easily, yet sufficiently nontrivial to illustrate some of the key aspects. For that we pick a slightly modernized version of LeNet (`relu` instead of `sigmoid` activation, MaxPooling rather than AveragePooling), as applied to Fashion MNIST. Moreover, we hybridize the network for performance. Since most of the code is standard we just introduce the basics without further detailed discussion. See Chapter 6 for a refresher as needed.

```
%matplotlib inline
import d2l
from mxnet import autograd, gluon, init, lr_scheduler, np, npx
from mxnet.gluon import nn
npx.set_np()

net = nn.HybridSequential()
net.add(nn.Conv2D(channels=6, kernel_size=5, padding=2, activation='relu'),
        nn.MaxPool2D(pool_size=2, strides=2),
        nn.Conv2D(channels=16, kernel_size=5, activation='relu'),
        nn.MaxPool2D(pool_size=2, strides=2),
        nn.Dense(120, activation='relu'),
        nn.Dense(84, activation='relu'),
        nn.Dense(10))
net.hybridize()
loss = gluon.loss.SoftmaxCrossEntropyLoss()
ctx = d2l.try_gpu()

batch_size = 256
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size=batch_size)

# The code is almost identical to "d2l.train_ch5" that defined in the lenet
# section of chapter convolutional neural networks
def train(net, train_iter, test_iter, num_epochs, loss, trainer, ctx):
    net.initialize(force_reinit=True, ctx=ctx, init=init.Xavier())
    animator = d2l.Animator(xlabel='epoch', xlim=[0, num_epochs],
                            legend=['train loss', 'train acc', 'test acc'])
    for epoch in range(num_epochs):
        metric = d2l.Accumulator(3)  # train_loss, train_acc, num_examples
        for i, (X, y) in enumerate(train_iter):
            X, y = X.as_in_context(ctx), y.as_in_context(ctx)
            with autograd.record():
                y_hat = net(X)
                l = loss(y_hat, y)
            l.backward()
            trainer.step(X.shape[0])
            metric.add(l.sum(), d2l.accuracy(y_hat, y), X.shape[0])
            train_loss, train_acc = metric[0]/metric[2], metric[1]/metric[2]
            if (i+1) % 50 == 0:
                animator.add(epoch + i/len(train_iter),
                             (train_loss, train_acc, None))
        test_acc = d2l.evaluate_accuracy_gpu(net, test_iter)
        animator.add(epoch+1, (None, None, test_acc))
    print('train loss %.3f, train acc %.3f, test acc %.3f' % (
        train_loss, train_acc, test_acc))
```

Let's have a look at what happens if we invoke this algorithm with default settings, such as a learning rate of 0.5 and train for 40 iterations. Note how the training accuracy keeps on increasing while progress in terms of test accuracy stalls beyond a point. The gap between both curves indicates overfitting.

```
lr, num_epochs = 0.5, 40
net.initialize(force_reinit=True, ctx=ctx, init=init.Xavier())
trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': lr})
train(net, train_iter, test_iter, num_epochs, loss, trainer, ctx)
```

```
train loss 2.303, train acc 0.101, test acc 0.100
```



11.11.2 Schedulers

One way of adjusting the learning rate is to set it explicitly at each step. This is conveniently achieved by the set_learning_rate method. We could adjust it downward after every epoch (or even after every minibatch), e.g., in a dynamic manner in response to how optimization is progressing.

```
trainer.set_learning_rate(0.1)
print('Learning rate is now %.2f' % trainer.learning_rate)
```

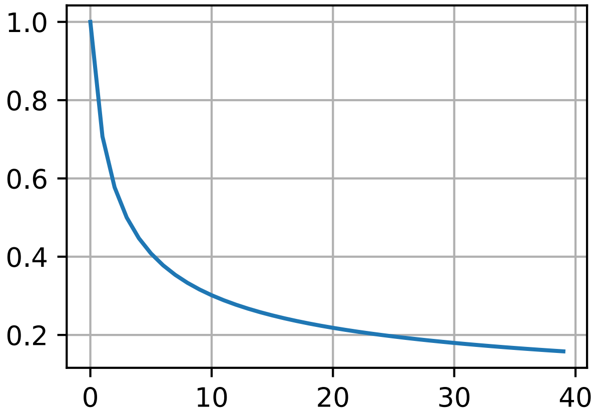
```
Learning rate is now 0.10
```

More generally we want to define a scheduler. When invoked with the number of updates it returns the appropriate value of the learning rate. Let's define a simple one that sets the learning rate to $\eta = \eta_0(t+1)^{-\frac{1}{2}}$.

```
class SquareRootScheduler(object):
    def __init__(self, lr=0.1):
        self.lr = lr

    def __call__(self, num_update):
        return self.lr * pow(num_update + 1.0, -0.5)
```

Let's plot its behavior over a range of values.

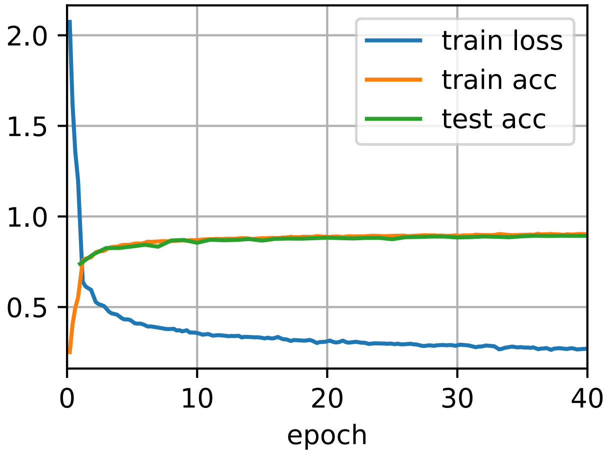
```
scheduler = SquareRootScheduler(lr=1.0)
d2l.plot(np.arange(num_epochs), [scheduler(t) for t in range(num_epochs)])
```



Now let's see how this plays out for training on FashionMNIST. We simply provide the scheduler as an additional argument to the training algorithm.

```
trainer = gluon.Trainer(net.collect_params(), 'sgd',
                        {'lr_scheduler': scheduler})
train(net, train_iter, test_iter, num_epochs, loss, trainer, ctx)
```

```
train loss 0.271, train acc 0.902, test acc 0.893
```



This worked quite a bit better than previously. Two things stand out: the curve was rather more smooth than previously. Secondly, there was less overfitting. Unfortunately it is not a well-resolved question as to why certain strategies lead to less overfitting in *theory*. There is some argument that a smaller stepsize will lead to parameters that are closer to zero and thus simpler. However, this doesn't explain the phenomenon entirely since we don't really stop early but simply reduce the learning rate gently.

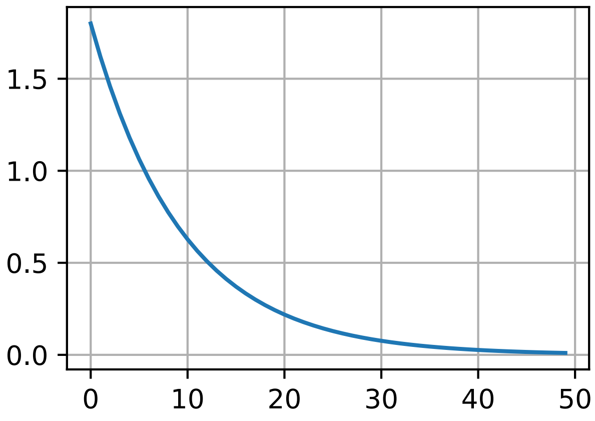
11.11.3 Policies

While we cannot possibly cover the entire variety of learning rate schedulers, we attempt to give a brief overview of popular policies below. Common choices are polynomial decay and piecewise constant schedules. Beyond that, cosine learning rate schedules have been found to work well empirically on some problems. Lastly, on some problems it is beneficial to warm up the optimizer prior to using large learning rates.

Factor Scheduler

One alternative to a polynomial decay would be a multiplicative one, that is $\eta_{t+1} \leftarrow \eta_t \cdot \alpha$ for $\alpha \in (0, 1)$. To prevent the learning rate from decaying beyond a reasonable lower bound the update equation is often modified to $\eta_{t+1} \leftarrow \max(\eta_{\min}, \eta_t \cdot \alpha)$.

```python
class FactorScheduler(object):
    def __init__(self, factor=1, stop_factor_lr=1e-7, base_lr=0.1):
        self.factor = factor
        self.stop_factor_lr = stop_factor_lr
        self.base_lr = base_lr

    def __call__(self, num_update):
        self.base_lr = max(self.stop_factor_lr, self.base_lr * self.factor)
        return self.base_lr

scheduler = FactorScheduler(factor=0.9, stop_factor_lr=1e-2, base_lr=2.0)
d2l.plot(np.arange(50), [scheduler(t) for t in range(50)])
```

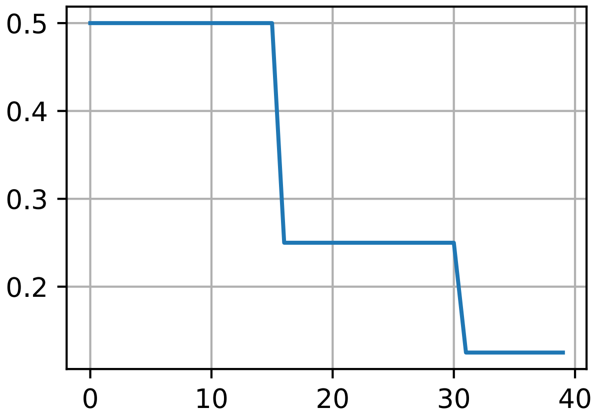


This can also be accomplished by a built-in scheduler in MXNet via the `lr_scheduler.FactorScheduler` object. It takes a few more parameters, such as warmup period, warmup mode (linear or constant), the maximum number of desired updates, etc.; Going forward we will use the built-in schedulers as appropriate and only explain their functionality here. As illustrated, it is fairly straightforward to build your own scheduler if needed.

Multi Factor Scheduler

A common strategy for training deep networks is to keep the learning rate piecewise constant and to decrease it by a given amount every so often. That is, given a set of times when to decrease the rate, such as $s = \{5, 10, 20\}$ decrease $\eta_{t+1} \leftarrow \eta_t \cdot \alpha$ whenever $t \in s$. Assuming that the values are halved at each step we can implement this as follows.

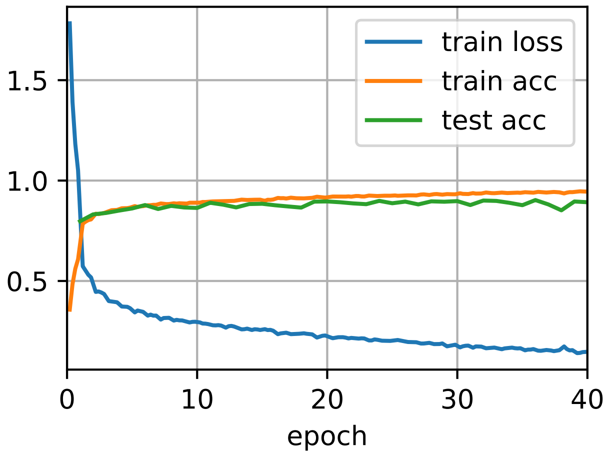
```
scheduler = lr_scheduler.MultiFactorScheduler(step=[15, 30], factor=0.5,
                                              base_lr=0.5)
d2l.plot(np.arange(num_epochs), [scheduler(t) for t in range(num_epochs)])
```



The intuition behind this piecewise constant learning rate schedule is that one lets optimization proceed until a stationary point has been reached in terms of the distribution of weight vectors. Then (and only then) do we decrease the rate such as to obtain a higher quality proxy to a good local minimum. The example below shows how this can produce ever slightly better solutions.

```
trainer = gluon.Trainer(net.collect_params(), 'sgd',
                        {'lr_scheduler': scheduler})
train(net, train_iter, test_iter, num_epochs, loss, trainer, ctx)
```

```
train loss 0.149, train acc 0.945, test acc 0.892
```



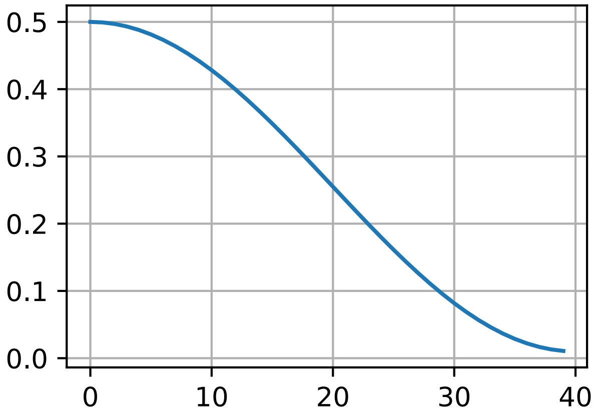
Cosine Scheduler

A rather perplexing heuristic was proposed by (Loshchilov & Hutter, 2016). It relies on the observation that we might not want to decrease the learning rate too drastically in the beginning and moreover, that we might want to "refine" the solution in the end using a very small learning rate. This results in a cosine-like schedule with the following functional form for learning rates in the range $t \in [0, T]$.

$$\eta_t = \eta_T + \frac{\eta_0 - \eta_T}{2} \left(1 + \cos(\pi t/T)\right) \tag{11.11.1}$$

Here η_0 is the initial learning rate, η_T is the target rate at time T. Furthermore, for $t > T$ we simply pin the value to η_T without increasing it again. In the following example, we set the max update step $T = 40$.

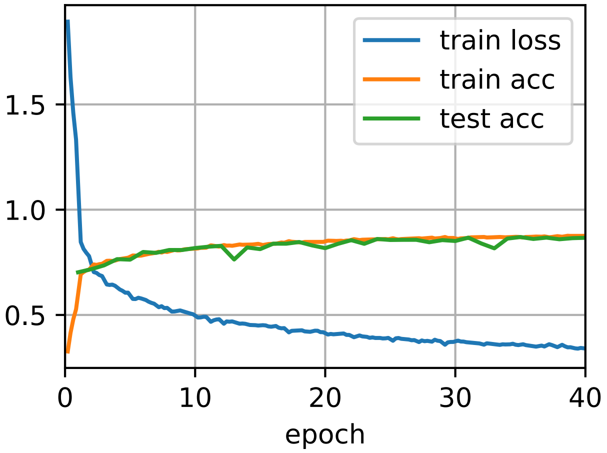
```
scheduler = lr_scheduler.CosineScheduler(max_update=40, base_lr=0.5,
                                         final_lr=0.01)
d2l.plot(np.arange(num_epochs), [scheduler(t) for t in range(num_epochs)])
```



In the context of computer vision this schedule *can* lead to improved results. Note, though, that such improvements are not guaranteed (as can be seen below).

```
trainer = gluon.Trainer(net.collect_params(), 'sgd',
                        {'lr_scheduler': scheduler})
train(net, train_iter, test_iter, num_epochs, loss, trainer, ctx)
```

```
train loss 0.342, train acc 0.876, test acc 0.867
```

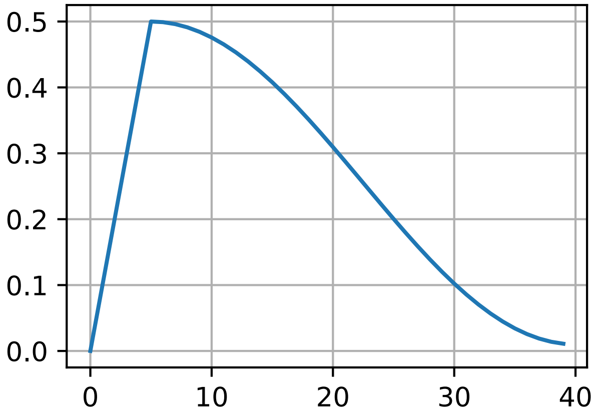


Warmup

In some cases initializing the parameters is not sufficient to guarantee a good solution. This particularly a problem for some advanced network designs that may lead to unstable optimization problems. We could address this by choosing a sufficiently small learning rate to prevent divergence in the beginning. Unfortunately this means that progress is slow. Conversely, a large learning rate initially leads to divergence.

A rather simple fix for this dilemma is to use a warmup period during which the learning rate *increases* to its initial maximum and to cool down the rate until the end of the optimization process. For simplicity one typically uses a linear increase for this purpose. This leads to a schedule of the form indicated below.

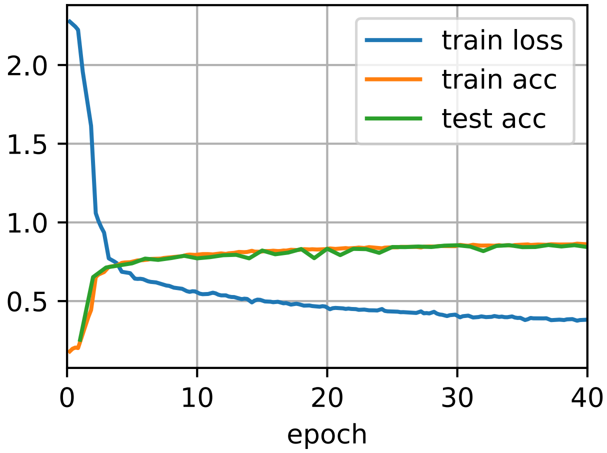
```
scheduler = lr_scheduler.CosineScheduler(40, warmup_steps=5, base_lr=0.5,
                                         final_lr=0.01)
d2l.plot(np.arange(num_epochs), [scheduler(t) for t in range(num_epochs)])
```



Note that the network converges better initially (in particular observe the performance during the first 5 epochs).

```
trainer = gluon.Trainer(net.collect_params(), 'sgd',
                        {'lr_scheduler': scheduler})
train(net, train_iter, test_iter, num_epochs, loss, trainer, ctx)
```

```
train loss 0.380, train acc 0.862, test acc 0.843
```



Warmup can be applied to any scheduler (not just cosine). For a more detailed discussion of learning rate schedules and many more experiments see also (Gotmare et al., 2018). In particular they find that a warmup phase limits the amount of divergence of parameters in very deep networks. This makes intuitively sense since we would expect significant divergence due to random initialization in those parts of the network that take the most time to make progress in the beginning.

Summary

- Decreasing the learning rate during training can lead to improved accuracy and (most perplexingly) reduced overfitting of the model.
- A piecewise decrease of the learning rate whenever progress has plateaued is effective in practice. Essentially this ensures that we converge efficiently to a suitable solution and only then reduce the inherent variance of the parameters by reducing the learning rate.
- Cosine schedulers are popular for some computer vision problems. See e.g., GluonCV[164] for details of such a scheduler.
- A warmup period before optimization can prevent divergence.
- Optimization serves multiple purposes in deep learning. Besides minimizing the training objective, different choices of optimization algorithms and learning rate scheduling can lead to rather different amounts of generalization and overfitting on the test set (for the same amount of training error).

[164] http://gluon-cv.mxnet.io

Exercises

1. Experiment with the optimization behavior for a given fixed learning rate. What is the best model you can obtain this way?
2. How does convergence change if you change the exponent of the decrease in the learning rate? Use `PolyScheduler` for your convenience in the experiments.
3. Apply the cosine scheduler to large computer vision problems, e.g., training ImageNet. How does it affect performance relative to other schedulers?
4. How long should warmup last?
5. Can you connect optimization and sampling? Start by using results from (Welling & Teh, 2011) on Stochastic Gradient Langevin Dynamics.

12 | Computational Performance

In deep learning, datasets are usually large and model computation is complex. Therefore, we are always very concerned about computing performance. This chapter will focus on the important factors that affect computing performance: imperative programming, symbolic programming, asynchronous programing, automatic parallel computation, and multi-GPU computation. By studying this chapter, you should be able to further improve the computing performance of the models that have been implemented in the previous chapters, for example, by reducing the model training time without affecting the accuracy of the model.

12.1 Compilers and Interpreters

So far, this book has focused on imperative programming, which makes use of statements such as `print`, `+` or `if` to change a program's state. Consider the following example of a simple imperative program.

```
def add(a, b):
    return a + b

def fancy_func(a, b, c, d):
    e = add(a, b)
    f = add(c, d)
    g = add(e, f)
    return g

print(fancy_func(1, 2, 3, 4))
```

```
10
```

Python is an interpreted language. When evaluating `fancy_func` it performs the operations making up the function's body *in sequence*. That is, it will evaluate `e = add(a, b)` and it will store the results as variable `e`, thereby changing the program's state. The next two statements `f = add(c, d)` and `g = add(e, f)` will be excecuted similarly, performing additions and storing the results as variables. Fig. 12.1.1 illustrates the flow of data.

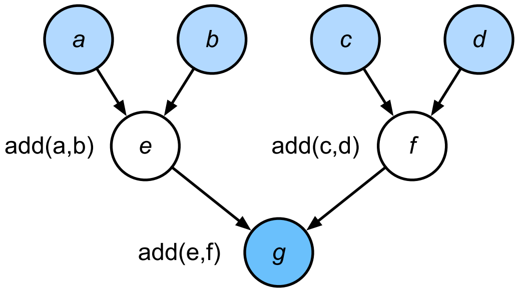


Fig. 12.1.1: Data flow in an imperative program.

Although imperative programming is convenient, it may be inefficient. On one hand, even if the `add` function is repeatedly called throughout `fancy_func`, Python will execute the three function calls individually. If these are executed, say, on a GPU (or even on multiple GPUs), the overhead arising from the Python interpreter can become overwhelming. Moreover, it will need to save the variable values of `e` and `f` until all the statements in `fancy_func` have been executed. This is because we do not know whether the variables `e` and `f` will be used by other parts of the program after the statements `e = add(a, b)` and `f = add(c, d)` have been executed.

12.1.1 Symbolic Programming

Consider the alternative, symbolic programming where computation is usually performed only once the process has been fully defined. This strategy is used by multiple deep learning frameworks, including Theano, Keras and TensorFlow (the latter two have since acquired imperative extensions). It usually involves the following steps:

1. Define the operations to be executed.
2. Compile the operations into an executable program.
3. Provide the required inputs and call the compiled program for execution.

This allows for a significant amount of optimization. First off, we can skip the Python interpreter in many cases, thus removing a performance bottleneck that can become significant on multiple fast GPUs paired with a single Python thread on a CPU. Secondly, a compiler might optimize and rewrite the above code into `print((1 + 2) + (3 + 4))` or even `print(10)`. This is possible since a compiler gets to see the full code before turning it into machine instructions. For instance, it can release memory (or never allocate it) whenever a variable is no longer needed. Or it can transform the code entirely into an equivalent piece. To get a better idea consider the following simulation of imperative programming (it's Python after all) below.

```
def add_():
    return '''
def add(a, b):
    return a + b
'''

def fancy_func_():
    return '''
def fancy_func(a, b, c, d):
    e = add(a, b)
    f = add(c, d)
```

(continues on next page)

(continued from previous page)

```
    g = add(e, f)
    return g
'''

def evoke_():
    return add_() + fancy_func_() + 'print(fancy_func(1, 2, 3, 4))'

prog = evoke_()
print(prog)
y = compile(prog, '', 'exec')
exec(y)
```

```
def add(a, b):
    return a + b

def fancy_func(a, b, c, d):
    e = add(a, b)
    f = add(c, d)
    g = add(e, f)
    return g
print(fancy_func(1, 2, 3, 4))
10
```

The differences between imperative (interpreted) programming and symbolic programming are as follows:

- Imperative programming is easier. When imperative programming is used in Python, the majority of the code is straightforward and easy to write. It is also easier to debug imperative programming code. This is because it is easier to obtain and print all relevant intermediate variable values, or use Python's built-in debugging tools.
- Symbolic programming is more efficient and easier to port. It makes it easier to optimize the code during compilation, while also having the ability to port the program into a format independent of Python. This allows the program to be run in a non-Python environment, thus avoiding any potential performance issues related to the Python interpreter.

12.1.2 Hybrid Programming

Historically most deep learning frameworks choose between an imperative or a symbolic approach. For example, Theano, TensorFlow (inspired by the latter), Keras and CNTK formulate models symbolically. Conversely Chainer and PyTorch take an imperative approach. An imperative mode was added TensorFlow 2.0 (via Eiger) and Keras in later revisions. When designing Gluon, developers considered whether it would be possible to combine the benefits of both programming models. This led to a hybrid model that lets users develop and debug using pure imperative programming, while having the ability to convert most programs into symbolic programs to be run when product-level computing performance and deployment are required.

In practice this means that we build models using either the `HybridBlock` or the `HybridSequential` and `HybridConcurrent` classes. By default, they are executed in the same way `Block` or `Sequential` and `Concurrent` classes are executed in imperative programming. `HybridSequential` is a subclass of `HybridBlock` (just like `Sequential` subclasses `Block`). When the `hybridize` function is called, Gluon compiles the model into the form used in symbolic programming. This allows one to optimize the compute-intensive components without sacrifices in the way a model is implemented. We will illustrate the benefits below, focusing on sequential models and blocks only (the concurrent composition works analogously).

12.1.3 HybridSequential

The easiest way to get a feel for how hybridization works is to consider deep networks with multiple layers. Conventionally the Python interpreter will need to execute the code for all layers to generate an instruction that can then be forwarded to a CPU or a GPU. For a single (fast) compute device this doesn't cause any major issues. On the other hand, if we use an advanced 8-GPU server such as an AWS P3dn.24xlarge instance Python will struggle to keep all GPUs busy. The single-threaded Python interpreter becomes the bottleneck here. Let's see how we can address this for significant parts of the code by replacing `Sequential` by `HybridSequential`. We begin by defining a simple MLP.

```
import d2l
from mxnet import np, npx
from mxnet.gluon import nn
npx.set_np()

# factory for networks
def get_net():
    net = nn.HybridSequential()
    net.add(nn.Dense(256, activation='relu'),
            nn.Dense(128, activation='relu'),
            nn.Dense(2))
    net.initialize()
    return net

x = np.random.normal(size=(1, 512))
net = get_net()
net(x)
```

```
array([[ 0.16526176, -0.14005631]])
```

By calling the `hybridize` function, we are able to compile and optimize the computation in the MLP. The model's computation result remains unchanged.

```
net.hybridize()
net(x)
```

```
array([[ 0.16526176, -0.14005631]])
```

This seems almost too good to be true: simply designate a block to be `HybridSequential`, write the same code as before and invoke `hybridize`. Once this happens the network is optimized (we will benchmark the performance below). Unfortunately this doesn't work magically for every layer. That said, the blocks provided by Gluon are by default subclasses of `HybridBlock` and thus hybridizable. A layer will not be optimized if it inherits from the `Block` instead.

Acceleration by Hybridization

To demonstrate the performance improvement gained by compilation we compare the time needed to evaluate `net(x)` before and after hybridization. Let's define a function to measure this time first. It will come handy throughout the chapter as we set out to measure (and improve) performance.

```
# Saved in the d2l package for later use
class benchmark:
    def __init__(self, description = 'Done in %.4f sec'):
        self.description = description

    def __enter__(self):
        self.timer = d2l.Timer()
        return self

    def __exit__(self, *args):
        print(self.description % self.timer.stop())
```

Now we can invoke the network twice, once with and once without hybridization.

```
net = get_net()
with benchmark('Without hybridization: %.4f sec'):
    for i in range(1000): net(x)
    npx.waitall()

net.hybridize()
with benchmark('With    hybridization: %.4f sec'):
    for i in range(1000): net(x)
    npx.waitall()
```

```
Without hybridization: 0.5402 sec
With    hybridization: 0.2650 sec
```

As is observed in the above results, after a HybridSequential instance calls the `hybridize` function, computing performance is improved through the use of symbolic programming.

Serialization

One of the benefits of compiling the models is that we can serialize (save) the model and its parameters to disk. This allows us to store a model in a manner that is independent of the front-end language of choice. This allows us to deploy trained models to other devices and easily use other front-end programming languages. At the same time the code is often faster than what can be achieved in imperative programming. Let's see the `export` method in action.

```
net.export('my_mlp')
!ls -lh my_mlp*
```

```
-rw-r--r-- 1 jenkins jenkins 643K Dec 17 22:22 my_mlp-0000.params
-rw-r--r-- 1 jenkins jenkins 3.0K Dec 17 22:22 my_mlp-symbol.json
```

The model is decomposed into a (large binary) parameter file and a JSON description of the program required to execute to compute the model. The files can be read by other front-end languages supported by Python or MXNet, such as C++, R, Scala, and Perl. Let's have a look at the model description.

```
!head my_mlp-symbol.json
```

```
{
  "nodes": [
    {
      "op": "null",
      "name": "data",
      "inputs": []
    },
    {
      "op": "null",
      "name": "dense3_weight",
```

Things are slightly more tricky when it comes to models that resemble code more closely. Basically hybridization needs to deal with control flow and Python overhead in a much more immediate manner. Moreover,

Contrary to the Block instance, which needs to use the `forward` function, for a HybridBlock instance we need to use the `hybrid_forward` function.

Earlier, we demonstrated that, after calling the `hybridize` method, the model is able to achieve superior computing performance and portability. Note, though that hybridization can affect model flexibility, in particular in terms of control flow. We will illustrate how to design more general models and also how compilation will remove spurious Python elements.

```
class HybridNet(nn.HybridBlock):
    def __init__(self, **kwargs):
        super(HybridNet, self).__init__(**kwargs)
        self.hidden = nn.Dense(4)
        self.output = nn.Dense(2)

    def hybrid_forward(self, F, x):
        print('module F: ', F)
        print('value  x: ', x)
        x = F.npx.relu(self.hidden(x))
        print('result  : ', x)
        return self.output(x)
```

The code above implements a simple network with 4 hidden units and 2 outputs. `hybrid_forward` takes an additional argument - the module `F`. This is needed since, depending on whether the code has been hybridized or not, it will use a slightly different library (`ndarray` or `symbol`) for processing. Both classes perform very similar functions and MXNet automatically determines the argument. To understand what is going on we print the arguments as part of the function invocation.

```
net = HybridNet()
net.initialize()
x = np.random.normal(size=(1, 3))
net(x)
```

```
module F:  <module 'mxnet.ndarray' from '/var/lib/jenkins/miniconda3/envs/d2l-en-1/lib/
↪python3.7/site-packages/mxnet/ndarray/__init__.py'>
value  x:  [[-0.6338663   0.40156594  0.46456942]]
result  :  [[0.01641375 0.         0.         0.        ]]
```

```
array([[0.00097611, 0.00019453]])
```

Repeating the forward computation will lead to the same output (we omit details). Now let's see what happens if we invoke the `hybridize` method.

```
net.hybridize()
net(x)
```

```
module F:  <module 'mxnet.symbol' from '/var/lib/jenkins/miniconda3/envs/d2l-en-1/lib/
↪python3.7/site-packages/mxnet/symbol/__init__.py'>
value  x:  <_Symbol data>
result  :  <_Symbol hybridnet0_relu0>
```

```
array([[0.00097611, 0.00019453]])
```

Instead of using `ndarray` we now use the `symbol` module for `F`. Moreover, even though the input is of `ndarray` type, the data flowing through the network is now converted to `symbol` type as part of the compilation process. Repeating the function call leads to a surprising outcome:

```
net(x)
```

```
array([[0.00097611, 0.00019453]])
```

This is quite different from what we saw previously. All print statements, as defined in `hybrid_forward` are omitted. Indeed, after hybridization the execution of `net(x)` does not involve the Python interpreter any longer. This means that any spurious Python code is omitted (such as print statements) in favor of a much more streamlined execution and better performance. Instead, MXNet directly calls the C++ backend. Also note that some functions are not supported in the `symbol` module (like `asnumpy`) and operations in-place like `a += b` and `a[:] = a + b` must be rewritten as `a = a + b`. Nonetheless, compilation of models is worth the effort whenever speed matters. The benefit can range from small percentage points to more than twice the speed, depending on the complexity of the model, the speed of the CPU and the speed and number of GPUs.

Summary

- Imperative programming makes it easy to design new models since it is possible to write code with control flow and the ability to use a large amount of the Python software ecosystem.
- Symbolic programming requires that we specify the program and compile it before executing it. The benefit is improved performance.
- MXNet is able to combine the advantages of both approaches as needed.

- Models constructed by the `HybridSequential` and `HybridBlock` classes are able to convert imperative programs into symbolic programs by calling the `hybridize` method.

Exercises

1. Design a network using the `HybridConcurrent` class. Alternatively look at *Networks with Parallel Concatenations (GoogLeNet)* (page 273) for a network to compose.
2. Add `x.asnumpy()` to the first line of the `hybrid_forward` function of the HybridNet class in this section. Execute the code and observe the errors you encounter. Why do they happen?
3. What happens if we add control flow, i.e. the Python statements `if` and `for` in the `hybrid_forward` function?
4. Review the models that interest you in the previous chapters and use the HybridBlock class or HybridSequential class to implement them.

12.2 Asynchronous Computation

Today's computers are highly parallel systems, consisting of multiple CPU cores (often multiple threads per core), multiple processing elements per GPU and often multiple GPUs per device. In short, we can process many different things at the same time, often on different devices. Unfortunately Python is not a great way of writing parallel and asynchronous code, at least not with some extra help. After all, Python is single-threaded and this is unlikely to change in the future. Deep learning frameworks such as MXNet and TensorFlow utilize an asynchronous programming model to improve performance (PyTorch uses Python's own scheduler leading to a different performance trade-off). Hence, understanding how asynchronous programming works helps us to develop more efficient programs, by proactively reducing computational requirements and mutual dependencies. This allows us to reduce memory overhead and increase processor utilization. We begin by importing the necessary libraries.

```
import d2l, numpy, os, subprocess
from mxnet import autograd, gluon, np, npx
from mxnet.gluon import nn
npx.set_np()
```

12.2.1 Asynchrony via Backend

For a warmup consider the following toy problem - we want to generate a random matrix and multiply it. Let's do that both in NumPy and in MXNet NP to see the difference.

```
with d2l.benchmark('numpy   : %.4f sec'):
    for _ in range(10):
        a = numpy.random.normal(size=(1000, 1000))
        b = numpy.dot(a, a)

with d2l.benchmark('mxnet.np: %.4f sec'):
    for _ in range(10):
        a = np.random.normal(size=(1000, 1000))
        b = np.dot(a, a)
```

```
numpy   : 0.5854 sec
mxnet.np: 0.0031 sec
```

This is orders of magnitude faster. At least it seems to be so. Since both are executed on the same processor something else must be going on. Forcing MXNet to finish all computation prior to returning shows what happened previously: computation is being executed by the backend while the frontend returns control to Python.

```
with d2l.benchmark():
    for _ in range(10):
        a = np.random.normal(size=(1000, 1000))
        b = np.dot(a, a)
    npx.waitall()
```

```
Done in 0.6616 sec
```

Broadly speaking, MXNet has a frontend for direct interaction with the users, e.g., via Python, as well as a backend used by the system to perform the computation. The backend possesses its own threads that continuously collect and execute queued tasks. Note that for this to work the backend must be able to keep track of the dependencies between various steps in the computational graph. Hence it is ony possible to parallelize operations that do not depend on each other.

As shown in Fig. 12.2.1, users can write MXNet programs in various frontend languages, such as Python, R, Scala and C++. Regardless of the front-end programming language used, the execution of MXNet programs occurs primarily in the back-end of C++ implementations. Operations issued by the frontend language are passed on to the backend for execution. The backend manages its own threads that continuously collect and execute queued tasks. Note that for this to work the backend must be able to keep track of the dependencies between various steps in the computational graph. That is, it is not possible to parallelize operations that depend on each other.

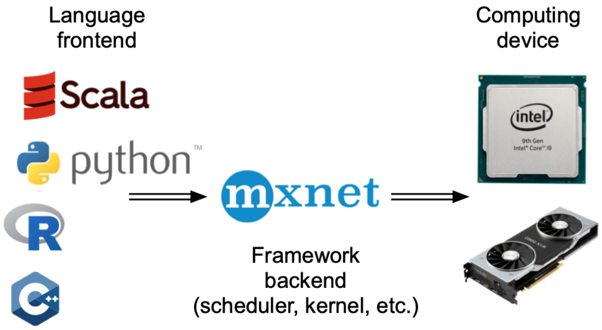


Fig. 12.2.1: Programming Frontends.

Let's look at another toy example to understand the dependency graph a bit better.

```
x = np.ones((1, 2))
y = np.ones((1, 2))
z = x * y + 2
z
```

```
array([[3., 3.]])
```

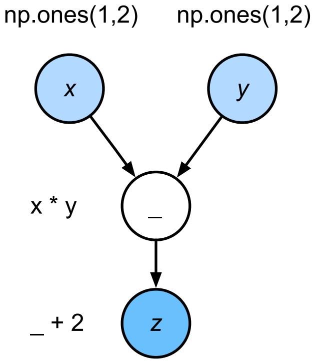


Fig. 12.2.2: Dependencies.

The code snippet above is also illustrated in Fig. 12.2.2. Whenever the Python frontend thread executes one of the first three statements, it simply returns the task to the backend queue. When the last statement's results need to be printed, the Python frontend thread will wait for the C++ backend thread to finish computing result of the variable z. One benefit of this design is that the Python frontend thread does not need to perform actual computations. Thus, there is little impact on the program's overall performance, regardless of Python's performance. Fig. 12.2.3 illustrates how frontend and backend interact.

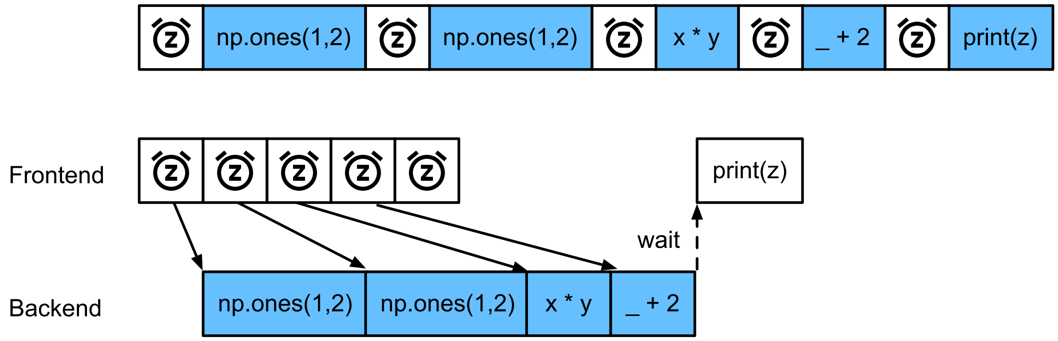


Fig. 12.2.3: Frontend and Backend.

12.2.2 Barriers and Blockers

There are a number of operations that will force Python to wait for completion: * Most obviously `npx.waitall()` waits until all computation has completed, regardless of when the compute instructions were issued. In practice it is a bad idea to use this operator unless absolutely necessary since it can lead to poor performance. * If we just want to wait until a specific variable is available we can call `z.wait_to_read()`. In this case MXNet blocks return to Python until the variable z has been computed. Other computation may well continue afterwards.

Let's see how this works in practice:

```
with d2l.benchmark('waitall     : %.4f sec'):
    b = np.dot(a, a)
    npx.waitall()

with d2l.benchmark('wait_to_read: %.4f sec'):
    b = np.dot(a, a)
    b.wait_to_read()
```

```
waitall     : 0.0105 sec
wait_to_read: 0.0042 sec
```

Both operations take approximately the same time to complete. Besides the obvious blocking operations we recommend that the reader is aware of *implicit* blockers. Printing a variable clearly requires the variable to be avaialable and is thus a blocker. Lastly, conversions to NumPy via `z.asnumpy()` and conversions to scalars via `z.item()` are blocking, since NumPy has no notion of asynchrony. It needs access to the values just like the `print` function. Copying small amounts of data frequently from MXNet's scope to NumPy and back can destroy performance of an otherwise efficient code, since each such operation requires the compute graph to evaluate all intermediate results needed to get the relevant term *before* anything else can be done.

```
with d2l.benchmark('numpy  conversion: %.4f sec'):
    b = np.dot(a, a)
    b.asnumpy()

with d2l.benchmark('scalar conversion: %.4f sec'):
    b = np.dot(a, a)
    b.sum().item()
```

```
numpy  conversion: 0.0064 sec
scalar conversion: 0.0134 sec
```

12.2.3 Improving Computation

On a heavily multithreaded system (even regular laptops have 4 threads or more and on multi-socket servers this number can exceed 256) the overhead of scheduling operations can become significant. This is why it's highly desirable to have computation and scheduling occur asynchronously and in parallel. To illustrate the benefit of doing this let's see what happens if we increment a variable by 1 multiple times, both in sequence or asynchronously. We simulate synchronous execution by inserting a `wait_to_read()` barrier in between each addition.

```
with d2l.benchmark('Synchronous : %.4f sec'):
    for _ in range(1000):
        y = x + 1
        y.wait_to_read()

with d2l.benchmark('Asynchronous: %.4f sec'):
    for _ in range(1000):
        y = x + 1
    y.wait_to_read()
```

```
Synchronous : 0.0553 sec
Asynchronous: 0.0537 sec
```

A slightly simplified interaction between the Python front-end thread and the C++ back-end thread can be summarized as follows:

1. The front-end orders the back-end to insert the calculation task `y = x + 1` into the queue.
2. The back-end then receives the computation tasks from the queue and performs the actual computations.
3. The back-end then returns the computation results to the front-end.

Assume that the durations of these three stages are t_1, t_2 and t_3, respectively. If we do not use asynchronous programming, the total time taken to perform 1000 computations is approximately $1000(t_1 + t_2 + t_3)$. If asynchronous programming is used, the total time taken to perform 1000 computations can be reduced to $t_1 + 1000t_2 + t_3$ (assuming $1000t_2 > 999t_1$), since the front-end does not have to wait for the back-end to return computation results for each loop.

12.2.4 Improving Memory Footprint

Imagine a situation where we keep on inserting operations into the backend by executing Python code on the frontend. For instance, the frontend might insert a large number of minibatch tasks within a very short time. After all, if no meaningful computation happens in Python this can be done quite quickly. If each of these tasks can be launched quickly at the same time this may cause a spike in memory usage. Given a finite amount of memory available on GPUs (and even on CPUs) this can lead to resource contention or even program crashes. Some readers might have noticed that previous training routines made use of synchronization methods such as `item` or even `asnumpy`.

We recommend to use these operations carefully, e.g., for each minibatch, such as to balance computational efficiency and memory footprint. To illustrate what happens let's implement a simple training loop for a deep network and measure its memory consumption and timing. Below is the mock data generator and deep network.

```
def data_iter():
    timer = d2l.Timer()
    num_batches, batch_size = 150, 1024
    for i in range(num_batches):
        X = np.random.normal(size=(batch_size, 512))
        y = np.ones((batch_size,))
        yield X, y
        if (i + 1) % 50 == 0:
            print('batch %d, time %.4f sec' % (i + 1, timer.stop()))

net = nn.Sequential()
net.add(nn.Dense(2048, activation='relu'),
        nn.Dense(512, activation='relu'), nn.Dense(1))
net.initialize()
trainer = gluon.Trainer(net.collect_params(), 'sgd')
loss = gluon.loss.L2Loss()
```

Next we need a tool to measure the memory footprint of our code. We use a relatively primitive ps call to accomplish this (note that the latter only works on Linux and MacOS). For a much more detailed analysis of what is going on here use e.g., Nvidia's Nsight[167] or Intel's vTune[168].

```
def get_mem():
    res = subprocess.check_output(['ps', 'u', '-p', str(os.getpid())])
    return int(str(res).split()[15]) / 1e3
```

Before we can begin testing we need to initialize the parameters of the network and process one batch. Otherwise it would be tricky to see what the additional memory consumption is. See Section 5.3 for further details related to initialization.

```
for X, y in data_iter():
    break
loss(y, net(X)).wait_to_read()
```

To ensure that we don't overflow the task buffer on the backend we insert a `wait_to_read` call for the loss function at the end of each loop. This forces the forward pass to complete before a new forward pass is commenced. Note that a (possibly more elegant) alternative would have been to track the loss in a scalar variable and to force a barrier via the `item` call.

```
mem = get_mem()
with d2l.benchmark('Time per epoch: %.4f sec'):
    for X, y in data_iter():
        with autograd.record():
            l = loss(y, net(X))
        l.backward()
        trainer.step(X.shape[0])
        l.wait_to_read() # barrier before new batch
```

(continues on next page)

[167] https://developer.nvidia.com/nsight-compute-2019_5
[168] https://software.intel.com/en-us/vtune

(continued from previous page)

```
    npx.waitall()
print('increased memory: %f MB' % (get_mem() - mem))
```

```
batch 50, time 2.3893 sec
batch 100, time 4.7343 sec
batch 150, time 7.2224 sec
Time per epoch: 7.2520 sec
increased memory: 7.216000 MB
```

As we see, the timing of the minibatches lines up quite nicely with the overall runtime of the optimization code. Moreover, memory footprint only increases slightly. Now let's see what happens if we drop the barrier at the end of each minibatch.

```
mem = get_mem()
with d2l.benchmark('Time per epoch: %.4f sec'):
    for X, y in data_iter():
        with autograd.record():
            l = loss(y, net(X))
        l.backward()
        trainer.step(X.shape[0])
    npx.waitall()
print('increased memory: %f MB' % (get_mem() - mem))
```

```
batch 50, time 0.1010 sec
batch 100, time 0.1986 sec
batch 150, time 0.2984 sec
Time per epoch: 7.8668 sec
increased memory: 0.020000 MB
```

Even though the time to issue instructions for the backend is an order of magnitude smaller, we still need to perform computation. Consequently a large amount of intermediate results cannot be released and may pile up in memory. While this didn't cause any issues in the toy example above, it might well have resulted in out of memory situations when left unchecked in real world scenarios.

Summary

- MXNet decouples the Python frontend from an execution backend. This allows for fast asynchronous insertion of commands into the backend and associated parallelism.
- Asynchrony leads to a rather responsive frontend. However, use caution not to overfill the task queue since it may lead to excessive memory consumption.
- It is recommended to synchronize for each minibatch to keep frontend and backend approximately synchronized.
- Be aware of the fact that conversions from MXNet's memory management to Python will force the backend to wait until the specific variable is ready. `print`, `asnumpy` and `item` all have this effect. This can be desirable but a carless use of synchronization can ruin performance.
- Chip vendors offer sophisticated performance analysis tools to obtain a much more fine-grained insight into the efficiency of deep learning.

Exercises

1. We mentioned above that using asynchronous computation can reduce the total amount of time needed to perform 1000 computations to $t_1 + 1000t_2 + t_3$. Why do we have to assume $1000t_2 > 999t_1$ here?
2. How would you need to modify the training loop if you wanted to have an overlap of one minibatch each? I.e., if you wanted to ensure that batch b_t finishes before batch b_{t+2} commences?
3. What might happen if we want to execute code on CPUs and GPUs simultaneously? Should you still insist on synchronizing after every minibatch has been issued?
4. Measure the difference between `waitall` and `wait_to_read`. Hint - perform a number of instructions and synchronize for an intermediate result.

12.3 Automatic Parallelism

MXNet automatically constructs computational graphs at the backend. Using a computational graph, the system is aware of all the dependencies, and can selectively execute multiple non-interdependent tasks in parallel to improve speed. For instance, Fig. 12.2.2 in Section 12.2 initializes two variables independently. Consequently the system can choose to execute them in parallel.

Typically, a single operator will use all the computational resources on all CPUs or on a single GPU. For example, the `dot` operator will use all cores (and threads) on all CPUs, even if there are multiple CPU processors on a single machine. The same applies to a single GPU. Hence parallelization isn't quite so useful single-device computers. With multiple devices things matter more. While parallelization is typically most relevant between multiple GPUs, adding the local CPU will increase performance slightly. See e.g., (Hadjis et al., 2016) for a paper that focuses on training computer vision models combining a GPU and a CPU. With the convenience of an automatically parallelizing framework we can accomplish the same goal in a few lines of Python code. More broadly, our discussion of automatic parallel computation focuses on parallel computation using both CPUs and GPUs, as well as the parallelization of computation and communication. We begin by importing the required packages and modules. Note that we need at least one GPU to run the experiments in this section.

```
import d2l
from mxnet import np, npx
npx.set_np()
```

12.3.1 Parallel Computation on CPUs and GPUs

Let's start by defining a reference workload to test - the run function below performs 10 matrix-matrix multiplications on the device of our choosing using data allocated into two variables, x_cpu and x_gpu.

```
def run(x):
    return [x.dot(x) for _ in range(10)]

x_cpu = np.random.uniform(size=(2000, 2000))
x_gpu = np.random.uniform(size=(6000, 6000), ctx=d2l.try_gpu())
```

Now we apply the function to the data. To ensure that caching doesn't play a role in the results we warm up the devices by performing a single pass on each of them prior to measuring.

```
run(x_cpu)  # Warm-up both devices
run(x_gpu)
npx.waitall()

with d2l.benchmark('CPU time: %.4f sec'):
    run(x_cpu)
    npx.waitall()

with d2l.benchmark('GPU time: %.4f sec'):
    run(x_gpu)
    npx.waitall()
```

```
CPU time: 0.3072 sec
GPU time: 0.3030 sec
```

If we remove the waitall() between both tasks the system is free to parallelize computation on both devices automatically.

```
with d2l.benchmark('CPU&GPU : %.4f sec'):
    run(x_cpu)
    run(x_gpu)
    npx.waitall()
```

```
CPU&GPU : 0.3074 sec
```

In the above case the total execution time is less than the sum of its parts, since MXNet automatically schedules computation on both CPU and GPU devices without the need for sophisticated code on behalf of the user.

12.3.2 Parallel Computation and Communication

In many cases we need to move data between different devices, say between CPU and GPU, or between different GPUs. This occurs e.g., when we want to perform distributed optimization where we need to aggregate the gradients over multiple accelerator cards. Let's simulate this by computing on the GPU and then copying the results back to the CPU.

```
def copy_to_cpu(x):
    return [y.copyto(npx.cpu()) for y in x]

with d2l.benchmark('Run  on GPU: %.4f sec'):
    y = run(x_gpu)
    npx.waitall()

with d2l.benchmark('Copy to CPU: %.4f sec'):
    y_cpu = copy_to_cpu(y)
    npx.waitall()
```

```
Run  on GPU: 0.3125 sec
Copy to CPU: 1.0149 sec
```

This is somewhat inefficient. Note that we could already start copying parts of `y` to the CPU while the remainder of the list is still being computed. This situatio occurs, e.g., when we compute the (backprop) gradient on a minibatch. The gradients of some of the parameters will be available earlier than that of others. Hence it works to our advantage to start using PCI-Express bus bandwidth while the GPU is still running. Removing `waitall` between both parts allows us to simulate this scenario.

```
with d2l.benchmark('Run on GPU and copy to CPU: %.4f sec'):
    y = run(x_gpu)
    y_cpu = copy_to_cpu(y)
    npx.waitall()
```

```
Run on GPU and copy to CPU: 1.0901 sec
```

The total time required for both operations is (as expected) significantly less than the sum of their parts. Note that this task is different from parallel computation as it uses a different resource: the bus between CPU and GPUs. In fact, we could compute on both devices and communicate, all at the same time. As noted above, there is a dependency between computation and communication: `y[i]` must be computed before it can be copied to the CPU. Fortunately, the system can copy `y[i-1]` while computing `y[i]` to reduce the total running time.

We conclude with an illustration of the computational graph and its dependencies for a simple two-layer MLP when training on a CPU and two GPUs, as depicted in Fig. 12.3.1. It would be quite painful to schedule the parallel program resulting from this manually. This is where it is advantageous to have a graph based compute backend for optimization.

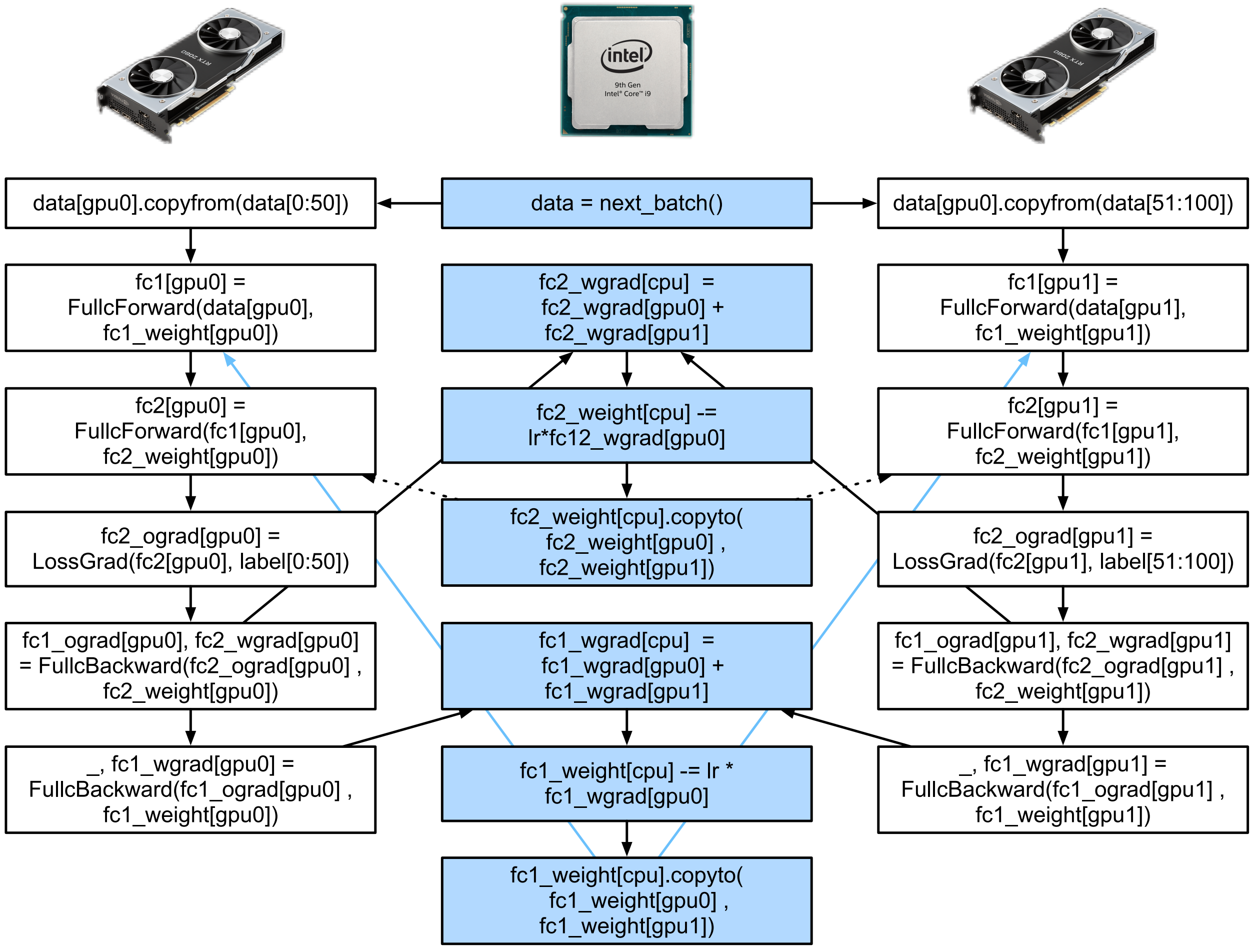

data[gpu0].copyfrom(data[0:50])
data = next_batch()
data[gpu0].copyfrom(data[51:100])
fc1[gpu0] = FullcForward(data[gpu0], fc1_weight[gpu0])
fc2_wgrad[cpu] = fc2_wgrad[gpu0] + fc2_wgrad[gpu1]
fc1[gpu1] = FullcForward(data[gpu1], fc1_weight[gpu1])
fc2[gpu0] = FullcForward(fc1[gpu0], fc2_weight[gpu0])
fc2_weight[cpu] -= lr*fc12_wgrad[gpu0]
fc2[gpu1] = FullcForward(fc1[gpu1], fc2_weight[gpu1])
fc2_ograd[gpu0] = LossGrad(fc2[gpu0], label[0:50])
fc2_weight[cpu].copyto(fc2_weight[gpu0] , fc2_weight[gpu1])
fc2_ograd[gpu1] = LossGrad(fc2[gpu1], label[51:100])
fc1_ograd[gpu0], fc2_wgrad[gpu0] = FullcBackward(fc2_ograd[gpu0] , fc2_weight[gpu0])
fc1_wgrad[cpu] = fc1_wgrad[gpu0] + fc1_wgrad[gpu1]
fc1_ograd[gpu1], fc2_wgrad[gpu1] = FullcBackward(fc2_ograd[gpu1] , fc2_weight[gpu1])
_, fc1_wgrad[gpu0] = FullcBackward(fc1_ograd[gpu0] , fc1_weight[gpu0])
fc1_weight[cpu] -= lr * fc1_wgrad[gpu0]
_, fc1_wgrad[gpu1] = FullcBackward(fc1_ograd[gpu1] , fc1_weight[gpu1])
fc1_weight[cpu].copyto(fc1_weight[gpu0] , fc1_weight[gpu1])

Fig. 12.3.1: Two layer MLP on a CPU and 2 GPUs.

Summary

- Modern systems have a variety of devices, such as multiple GPUs and CPUs. They can be used in parallel, asynchronously.
- Modern systems also have a variety of resources for communication, such as PCI Express, storage (typically SSD or via network), and network bandwidth. They can be used in parallel for peak efficiency.
- The backend can improve performance through through automatic parallel computation and communication.

Exercises

1. 10 operations were performed in the `run` function defined in this section. There are no dependencies between them. Design an experiment to see if MXNet will automatically execute them in parallel.
2. When the workload of an individual operator is sufficiently small, parallelization can help even on a single CPU or GPU. Design an experiment to verify this.
3. Design an experiment that uses parallel computation on CPU, GPU and communication between both devices.
4. Use a debugger such as NVIDIA's Nsight to verify that your code is efficient.
5. Designing computation tasks that include more complex data dependencies, and run experiments to see if you can obtain the correct results while improving performance.

12.4 Hardware

Building systems with great performance requires a good understanding of the algorithms and models to capture the statistical aspects of the problem. At the same time it is also indispensable to have at least a modicum of knowledge of the underlying hardware. The current section is no substitute for a proper course on hardware and systems design. Instead, it might serve as a starting point for understanding why some algorithms are more efficient than others and how to achieve good throughput. Good design can easily make a difference of an order of magnitude and, in turn, this can make the difference between being able to train a network (e.g., in a week) or not at all (in 3 months, thus missing the deadline). We'll start by looking at computers. Then we will zoom in to look more carefully at CPUs and GPUs. Lastly we zoom out to review how multiple computers are connected in a server center or in the cloud. This is not a GPU purchase guide. For this review Section 18.5. An introduction to cloud computing with AWS can be found in Section 18.3.

Impatient readers may be able to get by with Fig. 12.4.1. It is taken from Colin Scott's interactive post[171] which gives a good overview of the progress over the past decade. The original numbers are due to Jeff Dean's Stanford talk from 2010[172]. The discussion below explains some of the rationale for these numbers and how they can guide us in designing algorithms. The discussion below is very high level and cursory. It is clearly *no substitute* for a proper course but rather just meant to provide enough information for a statistical modeler to make suitable design decisions. For an in-depth overview of computer architecture we refer the reader to (Hennessy & Patterson, 2011) or a recent course on the subject, such as the one by Arste Asanovic[173].

[171] https://people.eecs.berkeley.edu/~rcs/research/interactive_latency.html

[172] https://static.googleusercontent.com/media/research.google.com/en//people/jeff/Stanford-DL-Nov-2010.pdf

[173] http://inst.eecs.berkeley.edu/~cs152/sp19/

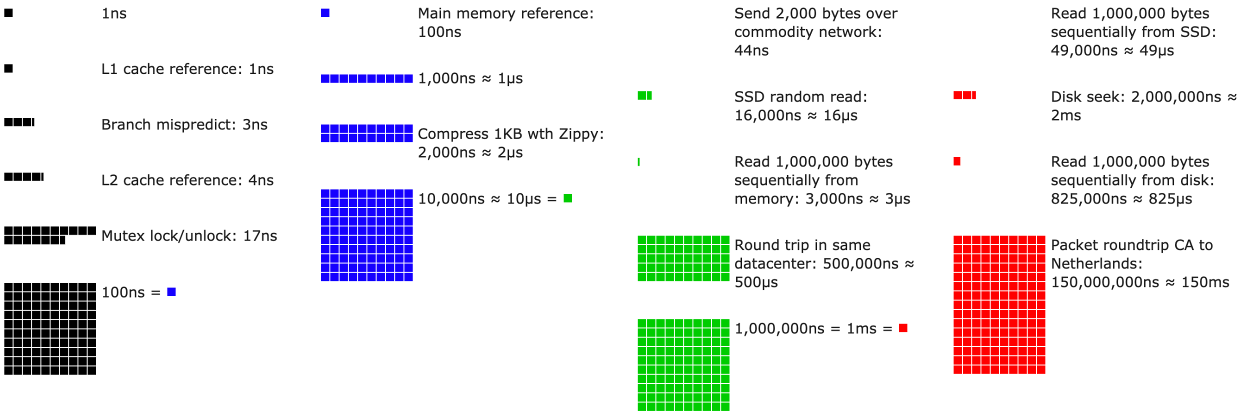


Fig. 12.4.1: Latency Numbers every Programmer should know.

12.4.1 Computers

Most deep learning researchers have access to a computer with a fair amount of memory, compute, some form of an accelerator such as a GPU, or multiples thereof. It consists of several key components:

- A processor, also referred to as CPU which is able to execute the programs we give it (in addition to running an operating system and many other things), typically consisting of 8 or more cores.
- Memory (RAM) to store and retrieve the results from computation, such as weight vectors, activations, often training data.
- An Ethernet network connection (sometimes multiple) with speeds ranging from 1Gbit/s to 100Gbit/s (on high end servers more advanced interconnects can be found).
- A high speed expansion bus (PCIe) to connect the system to one or more GPUs. Severs have up to 8 accelerators, often connected in an advanced topology, desktop systems have 1-2, depending on the budget of the user and the size of the power supply.
- Durable storage, such as a magnetic harddrive (HDD), solid state (SSD), in many cases connected using the PCIe bus, provides efficient transfer of training data to the system and storage of intermediate checkpoints as needed.

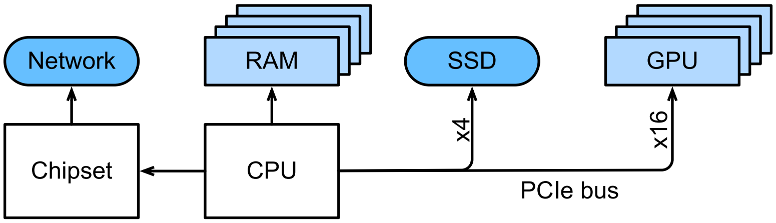


Fig. 12.4.2: Connectivity of components

As Fig. 12.4.2 indicates, most components (network, GPU, storage) are connected to the CPU across the PCI Express bus. It consists of multiple lanes that are directly attached to the CPU. For instance AMD's Threadripper 3 has 64 PCIe 4.0 lanes, each of which is capable 16 Gbit/s data transfer in both directions. The memory is directly attached to the CPU with a total bandwidth of up to 100 GB/s.

When we run code on a computer we need to shuffle data to the processors (CPU or GPU), perform computation and then move the results off the processor back to RAM and durable storage. Hence, in order to get good performance we need to make sure that this works seamlessly without any one of the systems becoming a major bottleneck. For instance, if we cannot load images quickly enough the processor won't have any work to do. Likewise, if we cannot move matrices quickly enough to the CPU (or GPU), its processing elements will starve. Finally, if we want to synchronize multiple computers across the network, the latter shouldn't slow down computation. One option is to interleave communication and computation. Let's have a look at the various components in more detail.

12.4.2 Memory

At its most basic memory is used to store data that needs to be readily accessible. At present CPU RAM is typically of the DDR4[174] variety, offering 20-25GB/s bandwidth per module. Each module has a 64 bit wide bus. Typically pairs of memory modules are used to allow for multiple channels. CPUs have between 2 and 4 memory channels, i.e., they have between 40GB/s and 100GB/s peak memory bandwidth. Often there are two banks per channel. For instance AMD's Zen 3 Threadripper has 8 slots.

While these numbers are impressive, indeed, they only tell part of the story. When we want to read a portion from memory we first need to tell the memory module where the information can be found. That is, we first need to send the *address* to RAM. Once this accomplished we can choose to read just a single 64bit record or a long sequence of records. The latter is called *burst read*. In a nutshell, sending an address to memory and setting up the transfer takes approximately 100ns (details depend on the specific timing coefficients of the memory chips used), every subsequent transfer takes only 0.2ns. In short, the first read is 500 times as expensive as subsequent ones! We could perform up to $10,000,000$ random reads per second. This suggests that we avoid random memory access as far as possible and use burst reads (and writes) instead.

Matters are a bit more complex when we take into account that we have multiple banks. Each bank can read memory largely independently. This means two things: the effective number of random reads is up to 4x higher, provided that they are spread evenly across memory. It also means that it's still a bad idea to perform random reads since burst reads are 4x faster, too. Secondly, due to memory alignment to 64 bit boundaries it is a good idea to align any datastructures with the same boundaries. Compilers do this pretty much automatically[175] when the appropriate flags are set. Curious readers are encouraged to review a lecture on DRAMs such as the one by Zeshan Chishti[176].

GPU memory is subject to even higher bandwidth requirements since they have many more processing elements than CPUs. By and large there are two options to address them. One is to make the memory bus significantly wider. For instance NVIDIA's RTX 2080 Ti has a 352 bit wide bus. This allows for much more information to be transferred at the same time. Secondly, GPUs use specific high-performance memory. Consumer grade devices, such as NVIDIA's RTX and Titan series typically use GDDR6[177] chips with over 500 GB/s aggregate bandwidth. An alternative is to use HBM (high bandwidth memory) modules. They use a very different interface and connect directly with GPUs on a dedicated silicon wafer. This makes them very expensive and their use is typically limited to high end server chips, such as the NVIDIA Volta V100 series of accelerators. Quite unsurprisingly GPU memory is *much* smaller than CPU memory due to its higher cost. For

[174] https://en.wikipedia.org/wiki/DDR4_SDRAM
[175] https://en.wikipedia.org/wiki/Data_structure_alignment
[176] http://web.cecs.pdx.edu/~zeshan/ece585_lec5.pdf
[177] https://en.wikipedia.org/wiki/GDDR6_SDRAM

our purposes, by and large their performance characteristics are similar, just a lot faster. We can safely ignore the details for the purpose of this book. They only matter when tuning GPU kernels for high throughput.

12.4.3 Storage

We saw that some of the key characteristics of RAM were *bandwidth* and *latency*. The same is true for storage devices, just that the differences can be even more extreme.

Hard Disks have been in use for over half a century. In a nutshell they contain a number of spinning platters with heads that can be positioned to read / write at any given track. High end end disks hold up to 16TB on 9 platters. One of the key benefits of HDDs is that they are relatively inexpensive. One of their many downsides are their typically catastrophic failure modes and their relatively high read latency.

To understand the latter, consider the fact that HDDs spin at around 7,200 RPM. If they were much faster they would shatter due to the centrifugal force exerted on the platters. This has a major downside when it comes to accessing a specific sector on the disk: we need to wait until the platter has rotated in position (we can move the heads but not accelerate the actual disks). Hence it can take over 8ms until the requested data is available. A common way this is expressed is to say that HDDs can operate at approximately 100 IOPs. This number has essentially remained unchanged for the past two decades. Worse still, it is equally difficult to increase bandwidth (it is in the order of 100-200 MB/s). After all, each head reads a track of bits, hence the bit rate only scales with the square root of the information density. As a result HDDs are quickly becoming relegated to archival storage and low-grade storage for very large datasets.

Solid State Drives use Flash memory to store information persistently. This allows for *much faster* access to stored records. Modern SSDs can operate at 100,000 to 500,000 IOPs, i.e., up to 3 orders of magnitude faster than HDDs. Furthermore, their bandwidth can reach 1-3GB/s, i.e., one order of magnitude faster than HDDs. These improvements sound almost too good to be true. Indeed, they come with a number of caveats, due to the way SSDs are designed.

- SSDs store information in blocks (256 KB or larger). They can only be written as a whole, which takes significant time. Consequently bit-wise random writes on SSD have very poor performance. Likewise, writing data in general takes significant time since the block has to be read, erased and then rewritten with new information. By now SSD controllers and firmware have developed algorithms to mitigate this. Nonetheless writes can be much slower, in particular for QLC (quad level cell) SSDs. The key for improved performance is to maintain a *queue* of operations, to prefer reads and to write in large blocks if possible.
- The memory cells in SSDs wear out relatively quickly (often already after a few thousand writes). Wear-level protection algorithms are able to spread the degradation over many cells. That said, it is not recommended to use SSDs for swap files or for large aggregations of logfiles.
- Lastly, the massive increase in bandwidth has forced computer designers to attach SSDs directly to the PCIe bus. The drives capable of handling this, referred to as NVMe (Non Volatile Memory enhanced), can use up to 4 PCIe lanes. This amounts to up to 8GB/s on PCIe 4.0.

Cloud Storage provides a configurable range of performance. That is, the assignment of storage to virtual machines is dynamic, both in terms of quantity and in terms speed, as chosen by the user. We recommend that the user increase the provisioned number of IOPs whenever latency is too high, e.g., during training with many small records.

12.4.4 CPUs

Central Processing Units (CPUs) are the centerpiece of any computer (as before we give a very high level description focusing primarily on what matters for efficient deep learning models). They consist of a number of key components: processor cores which are able to execute machine code, a bus connecting them (the specific topology differs significantly between processor models, generations and vendors), and caches to allow for higher bandwidth and lower latency memory access than what is possible by reads from main memory. Lastly, almost all modern CPUs contain vector processing units to aid with high performance linear algebra and convolutions, as they are common in media processing and machine learning.

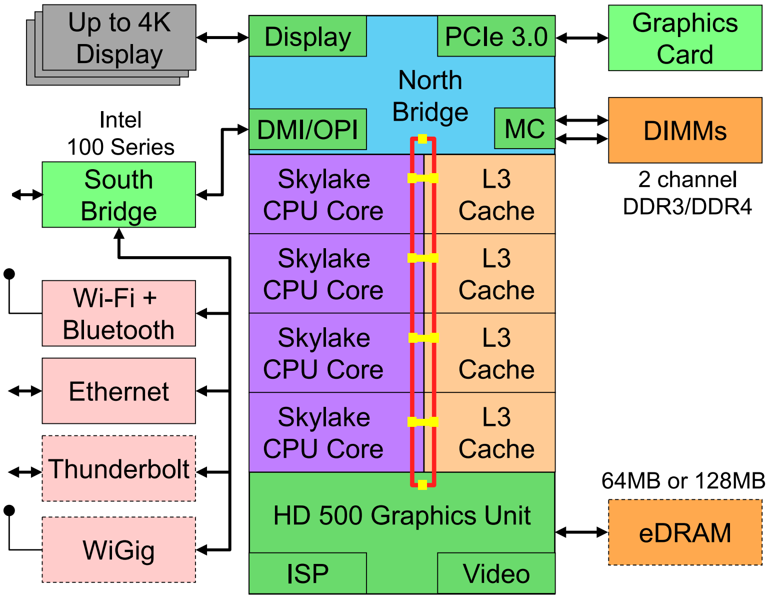


Fig. 12.4.3: Intel Skylake consumer quad-core CPU

Fig. 12.4.3 depicts an Intel Skylake consumer grade quad-core CPU. It has an integrated GPU, caches, and a ringbus connecting the four cores. Peripherals (Ethernet, WiFi, Bluetooth, SSD controller, USB, etc.) are either part of the chipset or directly attached (PCIe) to the CPU.

Microarchitecture

Each of the processor cores consists of a rather sophisticated set of components. While details differ between generations and vendors, the basic functionality is pretty much standard. The front end loads instructions and tries to predict which path will be taken (e.g., for control flow). Instructions are then decoded from assembly code to microinstructions. Assembly code is often not the lowest level code that a processor executes. Instead, complex instructions may be decoded into a set of more lower level operations. These are then processed by the actual execution core. Often the latter is capable of performing many operations simultaneously. For instance, the ARM Cortex A77 core of Fig. 12.4.4 is able to perform up to 8 operations simultaneously.

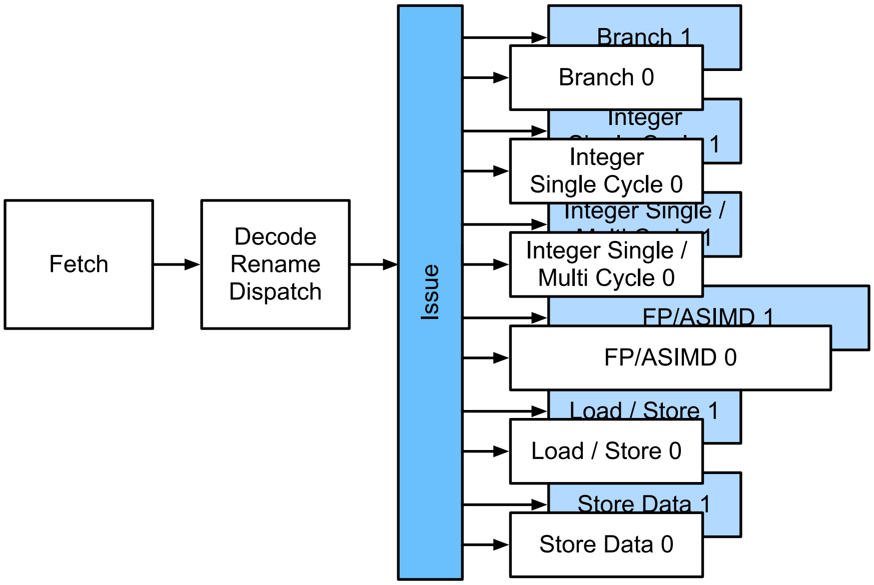


Fig. 12.4.4: ARM Cortex A77 Microarchitecture Overview

This means that efficient programs might be able to perform more than one instruction per clock cycle, *provided that* they can be carried out independently. Not all units are created equal. Some specialize in integer instructions whereas others are optimized for floating point performcne. To increase throughput the processor might also follow multiple codepaths simultaneously in a branching instruction and then discard the results of the branch not taken. This is why branch prediction units matter (on the frontend) such that only the most promising paths are pursued.

Vectorization

Deep learning is extremely compute hungry. Hence, to make CPUs suitable for machine learning one needs to perform many operations in one clock cycle. This is achieved via vector units. They have different names: on ARM they're called NEON, on x86 the latest generation is referred to as AVX2[178] units. A common aspect is that they are able to perform SIMD (single instruction multiple data) operations. Fig. 12.4.5 shows how 8 short integers can be added in one clock cycle on ARM.

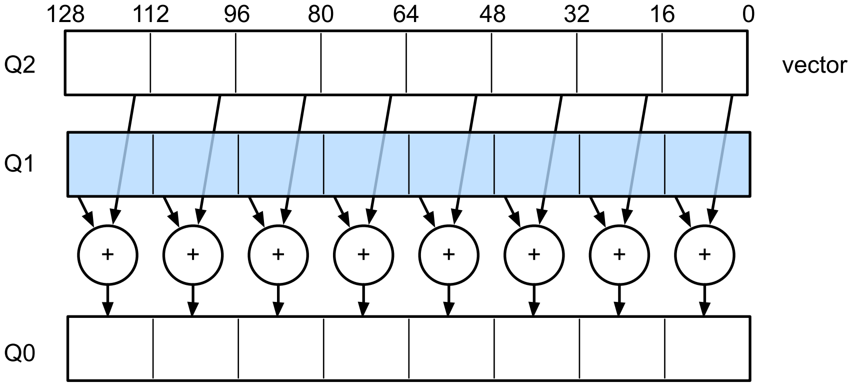


Fig. 12.4.5: 128 bit NEON vectorization

[178] https://en.wikipedia.org/wiki/Advanced_Vector_Extensions

Depending on architecture choices such registers are up to 512 bit long, allowing for the combination of up to 64 pairs of numbers. For instance, we might be multiplying two numbers and adding them to a third, which is also known as a fused multiply-add. Intel's OpenVino[179] uses these to achieve respectable throughput for deep learning on server grade CPUs. Note, though, that this number is entirely dwarved by what GPUs are capable of achieving. For instance, NVIDIA's RTX 2080 Ti has 4,352 CUDA cores, each of which is capable of processing such an operation at any time.

Cache

Consider the following situation: we have a modest CPU core with 4 cores as depicted in Fig. 12.4.3 above, running at 2GHz frequency. Moreover, let's assume that we have an IPC (instructions per clock) count of 1 and that the units have AVX2 with 256bit width enabled. Let's furthermore assume that at least one of the registers used for AVX2 operations needs to be retrieved from memory. This means that the CPU consumes 4x256bit = 1kbit of data per clock cycle. Unless we are able to transfer $2 \cdot 10^9 \cdot 128 = 256 \cdot 10^9$ bytes to the processor per second the processing elements are going to starve. Unfortunately the memory interface of such a chip only supports 20-40 GB/s data transfer, i.e., one order of magnitude less. The fix is to avoid loading *new* data from memory as far as possible and rather to cache it locally on the CPU. This is where caches come in handy (see this Wikipedia article[180] for a primer). Commonly the following names / concepts are used:

- **Registers** are strictly speaking not part of the cache. They help stage instructions. That said, CPU registers are memory locations that a CPU can access at clock speed without any delay penalty. CPUs have tens of registers. It is up to the compiler (or programmer) to use registers efficiently. For instance the C programming language has a `register` keyword.
- **L1** caches are the first line of defense against high memory bandwidth requirements. L1 caches are tiny (typical sizes might be 32-64kB) and often split into data and instructions caches. When data is found in the L1 cache access is very fast. If it cannot be found there, the search progresses down the cache hierarchy.
- **L2** caches are the next stop. Depending on architecture design and processor size they might be exclusive. They might be accessible only by a given core or shared between multiple cores. L2 caches are larger (typically 256-512kB per core) and slower than L1. Furthermore, to access something in L2 we first need to check to realize that the data isn't in L1, which adds a small amount of extra latency.
- **L3** caches are shared between multiple cores and can be quite large. AMD's Epyc 3 server CPUs have a whopping 256MB of cache spread across multiple chiplets. More typical numbers are in the 4-8MB range.

Predicting which memory elements will be needed next is one of the key optimization parameters in chip design. For instance, it is advisable to traverse memory in a *forward* direction since most caching algorithms will try to *read ahead* rather than backwards. Likewise, keeping memory access patterns local is agood way of improving performance. Adding caches is a double-edge sword. On one hand they ensure that the processor cores don't starve of data. At the same time they increase chip size, using up area that otherwise could have been spent on increasing processing power. Moreover, *cache misses* can be expensive. Consider the worst case scenario, depicted in Fig. 12.4.6. A memory location is cached on processor 0 when a thread on processor 1 requests the data. To obtain it, processor 0 needs to stop what it's doing, write the information back to main

[179] https://01.org/openvinotoolkit

[180] https://en.wikipedia.org/wiki/Cache_hierarchy

memory and then let processor 1 read it from memoruy. During this operation both processors wait. Quite potentially such code runs *more slowly* on multiple processors when compared to an efficient single-processor implementation. This is one more reason for why there is a practical limit to cache sizes (besides their physical size).

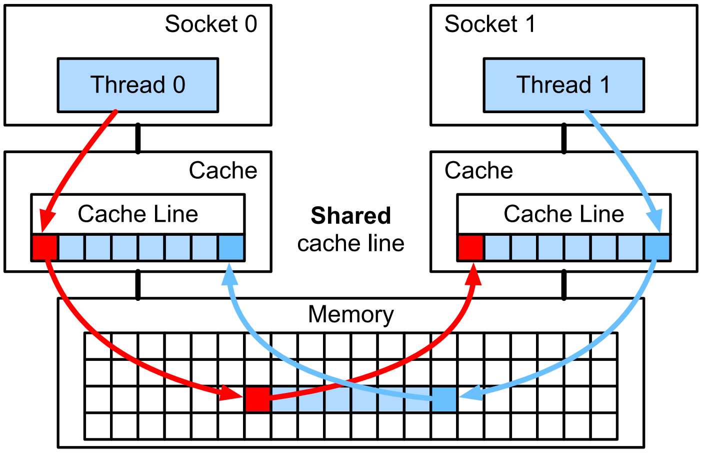


Fig. 12.4.6: False sharing (image courtesy of Intel)

12.4.5 GPUs and other Accelerators

It is not an exaggeration to claim that deep learning would not have been successful without GPUs. By the same token, it is quite reasonable to argue that GPU manufacturers' fortunes have been increased significantly due to deep learning. This co-evolution of hardware and algorithms has led to a situation where for better or worse deep learning is the preferable statistical modeling paradigm. Hence it pays to understand the specific benefits that GPUs and related accelerators such as the TPU (Jouppi et al., 2017) offer.

Of note is a distinction that is often made in practice: accelerators are optimized either for training or inference. For the latter we only need to compute the forward pass in a network. No storage of intermediate data is needed for backpropagation. Moreover, we may not need very precise computation (FP16 or INT8 typically suffice). On the other hand, during training all intermediate results need storing to compute gradients. Moreover, accumulating gradients requires higher precision to avoid numerical underflow (or overflow). This means that FP16 (or mixed precision with FP32) is the minimum required. All of this necessitates faster and larger memory (HBM2 vs. GDDR6) and more processing power. For instance, NVIDIA's Turing[181] T4 GPUs are optimized for inference whereas the V100 GPUs are preferable for training.

Recall Fig. 12.4.5. Adding vector units to a processor core allowed us to increase throughput significantly (in the example in the figure we were able to perform 16 operations simultaneously). What if we added operations that optimized not just operations between vectors but also between matrices? This strategy led to Tensor Cores (more on this shortly). Secondly, what if we added many more cores? In a nutshell, these two strategies summarize the design decisions in GPUs. Fig. 12.4.7 gives an overview over a basic processing block. It contains 16 integer and 16 floating point units. In addition to that, two Tensor Cores accelerate a narrow subset of additional operations relevant for deep learning. Each Streaming Multiprocessor (SM) consists of four such blocks.

[181] https://devblogs.nvidia.com/nvidia-turing-architecture-in-depth/

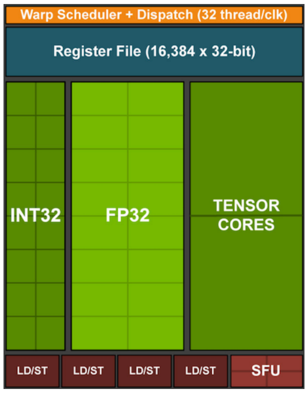

Warp Scheduler + Dispatch (32 thread/clk)
Register File (16,384 x 32-bit)
INT32
FP32
TENSOR CORES
LD/ST
LD/ST
LD/ST
LD/ST
SFU

Fig. 12.4.7: NVIDIA Turing Processing Block (image courtesy of NVIDIA)

12 streaming multiprocessors are then grouped into graphics processing clusters which make up the high-end TU102 processors. Ample memory channels and an L2 cache complement the setup. Fig. 12.4.8 has the relevant details. One of the reasons for designing such a device is that individual blocks can be added or removed as needed to allow for more compact chips and to deal with yield issues (faulty modules might not be activated). Fortunately programming such devices is well hidden from the casual deep learning researcher beneath layers of CUDA and framework code. In particular, more than one of the programs might well be executed simultaneously on the GPU, provided that there are available resources. Nonetheless it pays to be aware of the limitations of the devices to avoid picking models that do not fit into device memory.

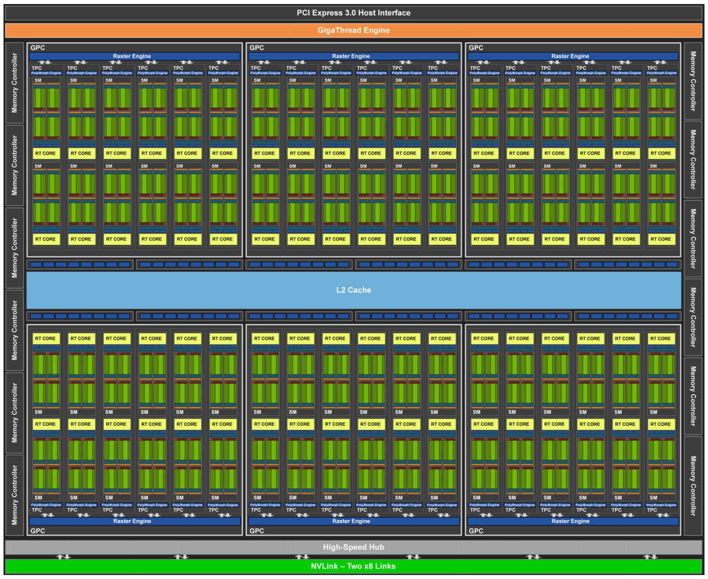


Fig. 12.4.8: NVIDIA Turing Architecture (image courtesy of NVIDIA)

A last aspect that is worth mentioning in more detail are TensorCores. They are an example of a recent trend of adding more optimized circuits that are specifically effective for deep learning. For instance, the TPU added a systolic array (Kung, 1988) for fast matrix multiplication. There the design was to support a very small number (one for the first generation of TPUs) of large operations. TensorCores are at the other end. They are optimized for small operations involving between 4x4 and 16x16 matrices, depending on their numerical precision. Fig. 12.4.9 gives an overview of the optimizations.

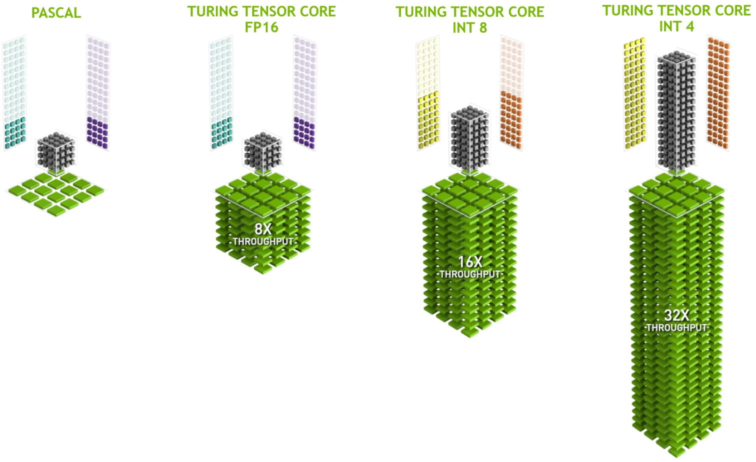


Fig. 12.4.9: NVIDIA TensorCores in Turing (image courtesy of NVIDIA)

Obviously when optimizing for computation we end up making certain compomises. One of them is that GPUs are not very good at handling interrupts and sparse data. While there are notable exceptions, such as Gunrock[182] (Wang et al., 2016), the access pattern of sparse matrices and vectors do not go well with the high bandwidth burst read operations where GPUs excel. Matching both goals is an area of active research. See e.g., DGL[183], a library tuned for deep learning on graphs.

12.4.6 Networks and Buses

Whenever a single device is insufficient for optimization we need to transfer data to and from it to synchronize processing. This is where networks and buses come in handy. We have a number of design parameters: bandwidth, cost, distance and flexibility. On one end we have WiFi which has a pretty good range, is very easy to use (no wires, after all), cheap but it offers comparatively mediocre bandwidth and latency. No machine learning researcher within their right mind would use it to build a cluster of servers. In what follows we focus on interconnects that are suitable for deep learning.

- **PCIe** is a dedicated bus for very high bandwidth point to point connections (up to 16 Gbs on PCIe 4.0) per lane. Latency is in the order of single-digit microseconds (5 μs). PCIe links are precious. Processors only have a limited number of them: AMD's EPYC 3 has 128 lanes, Intel's Xeon has up to 48 lanes per chip; on desktop grade CPUs the numbers are 20 (Ryzen 9) and 16 (Core i9) respectively. Since GPUs have typically 16 lanes this limits the number of GPUs that can connect to the CPU at full bandwidth. After all, they need to share the links with other high bandwidth peripherals such as storage and Ethernet. Just like with RAM access, large bulk transfers are preferable due to reduced packet overhead.
- **Ethernet** is the most commonly used way of connecting computers. While it is significantly slower than PCIe, it is very cheap and resilient to install and covers much longer distances. Typical bandwidth for low-grade servers is 1 GBit/s. Higher end devices (e.g., C5 instances[184] in the cloud) offer between 10 and 100 GBit/s bandwidth. As in all previous cases data transmission has significant overheads. Note that we almost never use raw Ethernet directly but rather a protocol that is executed on top of the physical interconnect (such as UDP or TCP/IP).

[182] https://github.com/gunrock/gunrock
[183] http://dgl.ai
[184] https://aws.amazon.com/ec2/instance-types/c5/

This adds further overhead. Like PCIe, Ethernet is designed to connect two devices, e.g., a computer and a switch.

- **Switches** allow us to connect multiple devices in a manner where any pair of them can carry out a (typically full bandwidth) point to point connection simultaneously. For instance, Ethernet switches might connect 40 servers at high cross-sectional bandwidth. Note that switches are not unique to traditional computer networks. Even PCIe lanes can be switched[185]. This occurs e.g., to connect a large number of GPUs to a host processor, as is the case for the P2 instances[186].
- **NVLink** is an alternative to PCIe when it comes to very high bandwidth interconnects. It offers up to 300 Gbit/s data transfer rate per link. Server GPUs (Volta V100) have 6 links whereas consumer grade GPUs (RTX 2080 Ti) have only one link, operating at a reduced 100 Gbit/s rate. We recommend to use NCCL[187] to achieve high data transfer between GPUs.

Summary

- Devices have overheads for operations. Hence it is important to aim for a small number of large transfers rather than many small ones. This applies to RAM, SSDs, Networks and GPUs.
- Vectorization is key for performance. Make sure you're aware of the specific abilities of your accelerator. E.g., some Intel Xeon CPUs are particularly good for INT8 operations, NVIDIA Volta GPUs excel at FP16 matrix-matrix operations and NVIDIA Turing shines at FP16, INT8 and INT4 operations.
- Numerical overflow due to small datatypes can be a problem during training (and to a lesser extent during inference).
- Aliasing can significantly degrade performance. For instance, memory alignment on 64 bit CPUs should be done with respect to 64 bit boundaries. On GPUs it's a good idea to keep convolution sizes aligned e.g., to TensorCores.
- Match your algorithms to the hardware (memory footprint, bandwidth, etc.). Great speedup (orders of magnitude) can be achieved when fitting the parameters into caches.
- We recommend that you sketch out the performance of a novel algorithm on paper before verifying the experimental results. Discrepancies of an order-of-magnitude or more are reasons for concern.
- Use profilers to debug performance bottlenecks.
- Training and inference hardware have different sweet spots in terms of price / performance.

[185] https://www.broadcom.com/products/pcie-switches-bridges/pcie-switches

[186] https://aws.amazon.com/ec2/instance-types/p2/

[187] https://github.com/NVIDIA/nccl

12.4.7 More Latency Numbers

The summary in Table 12.4.1 and Table 12.4.2 are due to Eliot Eshelman[188] who maintains an updated version of the numbers as a GitHub Gist[189].

Table 12.4.1: Common Latency Numbers.

Action	Time	Notes
L1 cache reference/hit	1.5 ns	4 cycles
Floating-point add/mult/FMA	1.5 ns	4 cycles
L2 cache reference/hit	5 ns	12 ~ 17 cycles
Branch mispredict	6 ns	15 ~ 20 cycles
L3 cache hit (unshared cache)	16 ns	42 cycles
L3 cache hit (shared in another core)	25 ns	65 cycles
Mutex lock/unlock	25 ns	
L3 cache hit (modified in another core)	29 ns	75 cycles
L3 cache hit (on a remote CPU socket)	40 ns	100 ~ 300 cycles (40 ~ 116 ns)
QPI hop to a another CPU (per hop)	40 ns	
64MB memory ref. (local CPU)	46 ns	TinyMemBench on Broadwell E5-2690v4
64MB memory ref. (remote CPU)	70 ns	TinyMemBench on Broadwell E5-2690v4
256MB memory ref. (local CPU)	75 ns	TinyMemBench on Broadwell E5-2690v4
Intel Optane random write	94 ns	UCSD Non-Volatile Systems Lab
256MB memory ref. (remote CPU)	120 ns	TinyMemBench on Broadwell E5-2690v4
Intel Optane random read	305 ns	UCSD Non-Volatile Systems Lab
Send 4KB over 100 Gbps HPC fabric	1 µs	MVAPICH2 over Intel Omni-Path
Compress 1KB with Google Snappy	3 µs	
Send 4KB over 10 Gbps ethernet	10 µs	
Write 4KB randomly to NVMe SSD	30 µs	DC P3608 NVMe SSD (QOS 99% is 500µs)
Transfer 1MB to/from NVLink GPU	30 µs	~33GB/s on NVIDIA 40GB NVLink
Transfer 1MB to/from PCI-E GPU	80 µs	~12GB/s on PCIe 3.0 x16 link
Read 4KB randomly from NVMe SSD	120 µs	DC P3608 NVMe SSD (QOS 99%)
Read 1MB sequentially from NVMe SSD	208 µs	~4.8GB/s DC P3608 NVMe SSD
Write 4KB randomly to SATA SSD	500 µs	DC S3510 SATA SSD (QOS 99.9%)
Read 4KB randomly from SATA SSD	500 µs	DC S3510 SATA SSD (QOS 99.9%)
Round trip within same datacenter	500 µs	One-way ping is ~250µs
Read 1MB sequentially from SATA SSD	2 ms	~550MB/s DC S3510 SATA SSD
Read 1MB sequentially from disk	5 ms	~200MB/s server HDD
Random Disk Access (seek+rotation)	10 ms	
Send packet CA->Netherlands->CA	150 ms	

Table 12.4.2: Latency Numbers for NVIDIA Tesla GPUs.

Action	Time	Notes
GPU Shared Memory access	30 ns	30~90 cycles (bank conflicts add latency)
GPU Global Memory access	200 ns	200~800 cycles
Launch CUDA kernel on GPU	10 µs	Host CPU instructs GPU to start kernel
Transfer 1MB to/from NVLink GPU	30 µs	~33GB/s on NVIDIA 40GB NVLink
Transfer 1MB to/from PCI-E GPU	80 µs	~12GB/s on PCI-Express x16 link

[188] https://gist.github.com/eshelman

[189] https://gist.github.com/eshelman/343a1c46cb3fba142c1afdcdeec17646

Exercises

1. Write C code to test whether there is any difference in speed between accessing memory aligned or misaligned relative to the external memory interface. Hint - be careful of caching effects.
2. Test the difference in speed between accessing memory in sequence or with a given stride.
3. How could you measure the cache sizes on a CPU?
4. How would you lay out data across multiple memory channels for maximum bandwidth? How would you lay it out if you had many small threads?
5. An enterprise class HDD is spinning at 10,000 rpm. What is the absolutely minimum time an HDD needs to spend worst case before it can read data (you can assume that heads move almost instantaneously)? Why are 2.5" HDDs becoming popular for commercial servers (relative to 3.5" and 5.25" drives)?
6. Assume that an HDD manufacturer increases the storage density from 1 Tbit per square inch to 5 Tbit per square inch. How much information can you store on a ring on a 2.5" HDD? Is there a difference between the inner and outer tracks?
7. The AWS P2 instances have 16 K80 Kepler GPUs. Use `lspci` on a p2.16xlarge and a p2.8xlarge instance to understand how the GPUs are connected to the CPUs. Hint - keep your eye out for PCI PLX bridges.
8. Going from 8 bit to 16 bit datatypes increases the amount of silicon approximately by 4x. Why? Why might NVIDIA have added INT4 operations to their Turing GPUs.
9. Given 6 high speed links between GPUs (such as for the Volta V100 GPUs), how would you connect 8 of them? Look up the connectivity used in the P3.16xlarge servers.
10. How much faster is it to read forward through memory vs. reading backwards? Does this number differ between different computers and CPU vendors? Why? Write C code and experiment with it.
11. Can you measure the cache size of your disk? What is it for a typical HDD? Do SSDs need a cache?
12. Measure the packet overhead when sending messages across the Ethernet. Look up the difference between UDP and TCP/IP connections.
13. Direct Memory Access allows devices other than the CPU to write (and read) directly to (from) memory. Why is this a good idea?
14. Look at the performance numbers for the Turing T4 GPU. Why does the performance 'only' double as you go from FP16 to INT8 and INT4?
15. What is the shortest time it should take for a packet on a roundtrip between San Francisco and Amsterdam? Hint - you can assume that the distance is 10,000km.

12.5 Training on Multiple GPUs

So far we discussed how to train models efficiently on CPUs and GPUs. We even showed how deep learning frameworks such as MXNet (and TensorFlow) allow one to parallelize computation and communication automatically between them in Section 12.3. Lastly, we showed in Section 5.6 how to list all available GPUs on a computer using `nvidia-smi`. What we did *not* discuss is how to actually parallelize deep learning training (we omit any discussion of *inference* on multiple GPUs here as it's a rather rarely used and advanced topic that goes beyond the scope of this book). Instead, we implied in passing that one would somehow split the data across multiple devices and make it work. The present section fills in the details and shows how to train a network in parallel when starting from scratch. Details on how to take advantage of functionality in Gluon is relegated to Section 12.6. We assume that the reader is familiar with minibatch SGD algorithms such as the ones described in Section 11.5.

12.5.1 Splitting the Problem

Let's start with a simple computer vision problem and a slightly archaic network, e.g., with multiple layers of convolutions, pooling, and possibly a few dense layers in the end. That is, let's start with a network that looks quite similar to LeNet (LeCun et al., 1998) or AlexNet (Krizhevsky et al., 2012). Given multiple GPUs (2 if it's a desktop server, 4 on a g4dn.12xlarge, 8 on an AWS p3.16xlarge, or 16 on a p2.16xlarge), we want to partition training in a manner as to achieve good speedup while simultaneously benefitting from simple and reproducible design choices. Multiple GPUs, after all, increase both *memory* and *compute* ability. In a nutshell, we have a number of choices, given a minibatch of training data that we want to classify.

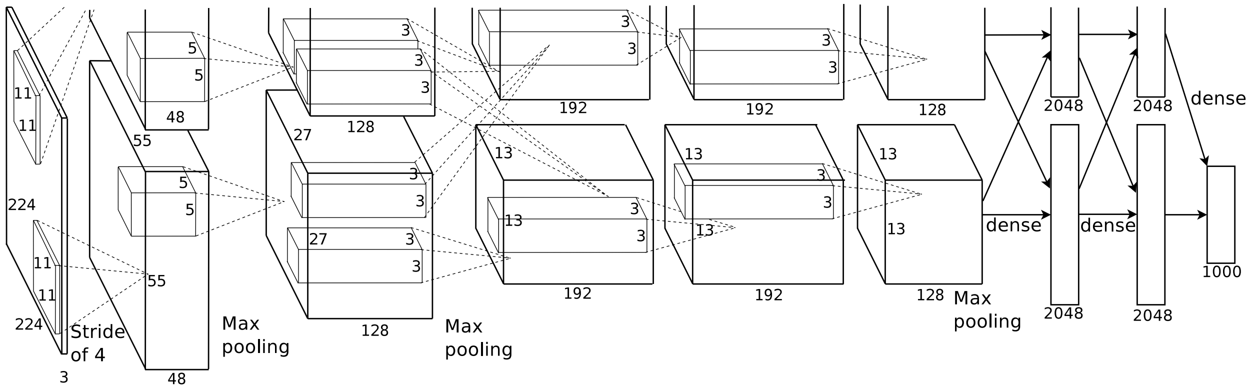


Fig. 12.5.1: Model parallelism in the original AlexNet design due to limited GPU memory.

- We could partition the network layers across multiple GPUs. That is, each GPU takes as input the data flowing into a particular layer, processes data across a number of subsequent layers and then sends the data to the next GPU.
  - This allows us to process data with larger networks when compared to what a single GPU could handle.
  - Memory footprint per GPU can be well controlled (it's a fraction of the total network footprint)
  - The interface between layers (and thus GPUs) requires tight synchronization. This can be tricky, in particular if the computational workloads are not properly matched between layers. The problem is exacerbated for large numbers of GPUs.

  - The interface between layers requires large amounts of data transfer (activations, gradients). This may overwhelm the bandwidth of the GPU buses.
  - Compute intensive, yet sequential operations are nontrivial to partition. See e.g., (Mirhoseini et al., 2017) for a best effort in this regard. It remains a difficult problem and it is unclear whether it is possible to achieve good (linear) scaling on nontrivial problems. We do not recommend it unless there is excellent framework / OS support for chaining together multiple GPUs.
- We could split the work required by individual layers. For instance, rather than computing 64 channels on a single GPU we could split up the problem across 4 GPUs, each of which generate data for 16 channels. Likewise, for a dense layer we could split the number of output neurons. Fig. 12.5.1 illustrates this design. The figure is taken from (Krizhevsky et al., 2012) where this strategy was used to deal with GPUs that had a very small memory footprint (2GB at the time).
  - This allows for good scaling in terms of computation, provided that the number of channels (or neurons) is not too small.
  - Multiple GPUs can process increasingly larger networks since the memory available scales linearly.
  - We need a *very large* number of synchronization / barrier operations since each layer depends on the results from all other layers.
  - The amount of data that needs to be transferred is potentially even larger than when distributing layers across GPUs. We do not recommend this approach due to its bandwidth cost and complexity.
- Lastly we could partition data across multiple GPUs. This way all GPUs perform the same type of work, albeit on different observations. Gradients are aggregated between GPUs after each minibatch.
  - This is the simplest approach and it can be applied in any situation.
  - Adding more GPUs does not allow us to train larger models.
  - We only need to synchronize after each minibatch. That said, it's highly desirable to start exchanging gradients parameters already while others are still being computed.
  - Large numbers of GPUs lead to very large minibatch sizes, thus reducing training efficiency.

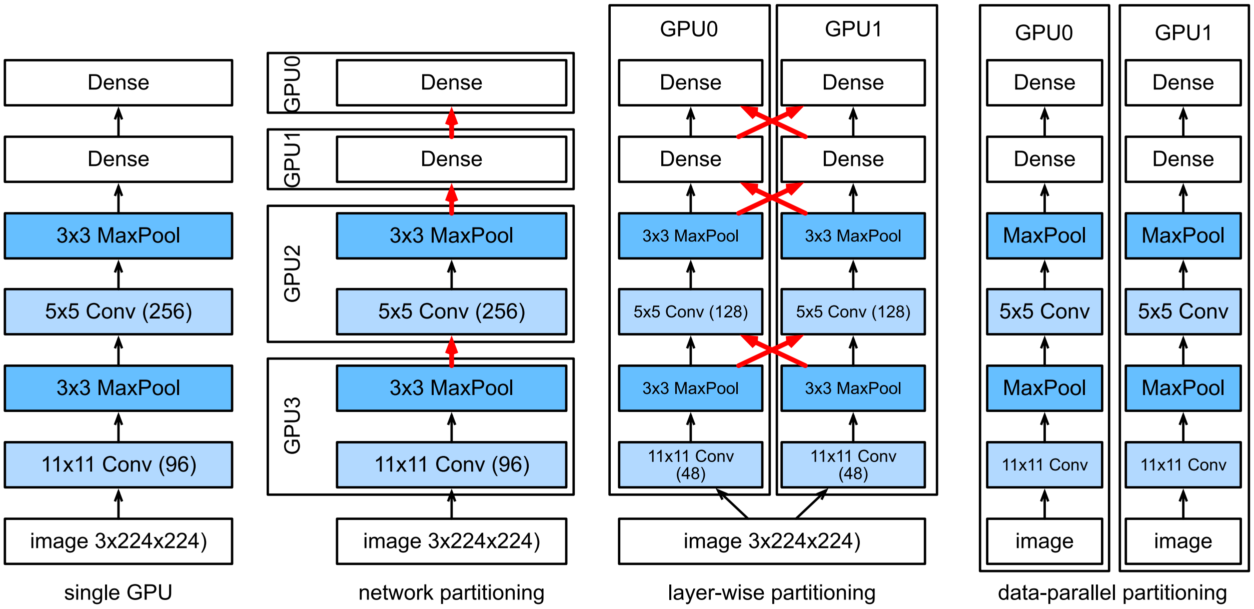


Fig. 12.5.2: Parallelization on multiple GPUs. From left to right - original problem, network partitioning, layer partitioning, data parallelism.

By and large, data parallelism is the most convenient way to proceed, provided that we have access to GPUs with sufficiently large memory. See also (Li et al., 2014) for a detailed description of partitioning for distributed training. GPU memory used to be a problem in the early days of deep learning. By now this issue has been resolved for all but the most unusual cases. We focus on data parallelism in what follows.

12.5.2 Data Parallelism

Assume that there are k GPUs on a machine. Given the model to be trained, each GPU will maintain a complete set of model parameters independently. Training proceeds as follows (see Fig. 12.5.3 for details on data parallel training on two GPUs):

- In any iteration of training, given a random minibatch, we split the examples in the batch into k portions and distribute them evenly across the GPUs.
- Each GPU calculates loss and gradient of the model parameters based on the minibatch subset it was assigned and the model parameters it maintains.
- The local gradients of each of the k GPUs are aggregated to obtain the current minibatch stochastic gradient.
- The aggregate gradient is re-distributed to each GPU.
- Each GPU uses this minibatch stochastic gradient to update the complete set of model parameters that it maintains.

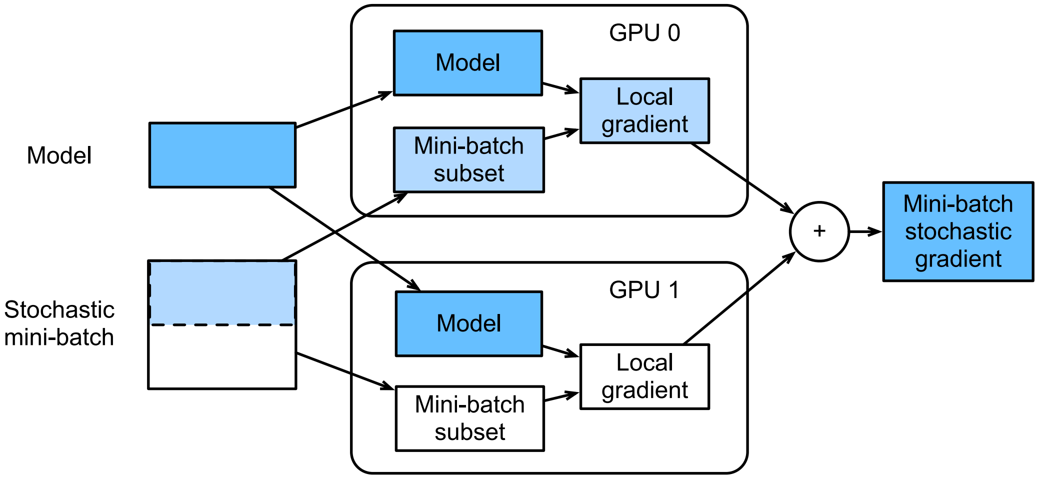


Fig. 12.5.3: Calculation of minibatch stochastic gradient using data parallelism and two GPUs.

A comparison of different ways of parallelization on multiple GPUs is depicted in Fig. 12.5.2. Note that in practice we *increase* the minibatch size k-fold when training on k GPUs such that each GPU has the same amount of work to do as if we were training on a single GPU only. On a 16 GPU server this can increase the minibatch size considerably and we may have to increase the learning rate accordingly. Also note that Section 7.5 needs to be adjusted (e.g., by keeping a separate batch norm coefficient per GPU). In what follows we will use Section 6.6 as the toy network to illustrate multi-GPU training. As always we begin by importing the relevant packages and modules.

```
%matplotlib inline
import d2l
from mxnet import autograd, gluon, np, npx
npx.set_np()
```

12.5.3 A Toy Network

We use LeNet as introduced in Section 6.6. We define it from scratch to illustrate parameter exchange and synchronization in detail.

```
# Initialize model parameters
scale = 0.01
W1 = np.random.normal(scale=scale, size=(20, 1, 3, 3))
b1 = np.zeros(20)
W2 = np.random.normal(scale=scale, size=(50, 20, 5, 5))
b2 = np.zeros(50)
W3 = np.random.normal(scale=scale, size=(800, 128))
b3 = np.zeros(128)
W4 = np.random.normal(scale=scale, size=(128, 10))
b4 = np.zeros(10)
params = [W1, b1, W2, b2, W3, b3, W4, b4]

# Define the model
def lenet(X, params):
    h1_conv = npx.convolution(data=X, weight=params[0], bias=params[1],
                              kernel=(3, 3), num_filter=20)
    h1_activation = npx.relu(h1_conv)
```

(continues on next page)

(continued from previous page)

```
    h1 = npx.pooling(data=h1_activation, pool_type='avg', kernel=(2, 2),
                     stride=(2, 2))
    h2_conv = npx.convolution(data=h1, weight=params[2], bias=params[3],
                              kernel=(5, 5), num_filter=50)
    h2_activation = npx.relu(h2_conv)
    h2 = npx.pooling(data=h2_activation, pool_type='avg', kernel=(2, 2),
                     stride=(2, 2))
    h2 = h2.reshape(h2.shape[0], -1)
    h3_linear = np.dot(h2, params[4]) + params[5]
    h3 = npx.relu(h3_linear)
    y_hat = np.dot(h3, params[6]) + params[7]
    return y_hat

# Cross-entropy loss function
loss = gluon.loss.SoftmaxCrossEntropyLoss()
```

12.5.4 Data Synchronization

For efficient multi-GPU training we need two basic operations: firstly we need to have the ability to distribute a list of parameters to multiple devices and to attach gradients (`get_params`). Without parameters it's impossible to evaluate the network on a GPU. Secondly, we need the ability to sum parameters across multiple devices, i.e., we need an `allreduce` function.

```
def get_params(params, ctx):
    new_params = [p.copyto(ctx) for p in params]
    for p in new_params:
        p.attach_grad()
    return new_params
```

Let's try it out by copying the model parameters of lenet to gpu(0).

```
new_params = get_params(params, d2l.try_gpu(0))
print('b1 weight:', new_params[1])
print('b1 grad:', new_params[1].grad)
```

```
b1 weight: [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.] @gpu(0)
b1 grad: [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.] @gpu(0)
```

Since we didn't perform any computation yet, the gradient with regard to the bias weights is still 0. Now let's assume that we have a vector distributed across multiple GPUs. The following allreduce function adds up all vectors and broadcasts the result back to all GPUs. Note that for this to work we need to copy the data to the device accumulating the results.

```
def allreduce(data):
    for i in range(1, len(data)):
        data[0][:] += data[i].copyto(data[0].context)
    for i in range(1, len(data)):
        data[0].copyto(data[i])
```

Let's test this by creating vectors with different values on different devices and aggregate them.

```
data = [np.ones((1, 2), ctx=d2l.try_gpu(i)) * (i + 1) for i in range(2)]
print('before allreduce:\n', data[0], '\n', data[1])
allreduce(data)
print('after allreduce:\n', data[0], '\n', data[1])
```

```
before allreduce:
 [[1. 1.]] @gpu(0)
 [[2. 2.]] @gpu(1)
after allreduce:
 [[3. 3.]] @gpu(0)
 [[3. 3.]] @gpu(1)
```

12.5.5 Distributing Data

We need a simple utility function to distribute a minibatch evenly across multiple GPUs. For instance, on 2 GPUs we'd like to have half of the data to be copied to each of the GPUs. Since it's more convenient and more concise, we use the built-in split and load function in Gluon (to try it out on a 4×5 matrix).

```
data = np.arange(20).reshape(4, 5)
ctx = [npx.gpu(0), npx.gpu(1)]
split = gluon.utils.split_and_load(data, ctx)
print('input :', data)
print('load into', ctx)
print('output:', split)
```

```
input : [[ 0.  1.  2.  3.  4.]
 [ 5.  6.  7.  8.  9.]
 [10. 11. 12. 13. 14.]
 [15. 16. 17. 18. 19.]]
load into [gpu(0), gpu(1)]
output: [array([[0., 1., 2., 3., 4.],
       [5., 6., 7., 8., 9.]], ctx=gpu(0)), array([[10., 11., 12., 13., 14.],
       [15., 16., 17., 18., 19.]], ctx=gpu(1))]
```

For later reuse we define a `split_batch` function which splits both data and labels.

```
# Saved in the d2l package for later use
def split_batch(X, y, ctx_list):
    """Split X and y into multiple devices specified by ctx."""
    assert X.shape[0] == y.shape[0]
    return (gluon.utils.split_and_load(X, ctx_list),
            gluon.utils.split_and_load(y, ctx_list))
```

12.5.6 Training

Now we can implement multi-GPU training on a single minibatch. Its implementation is primarily based on the data parallelism approach described in this section. We will use the auxiliary functions we just discussed, `allreduce` and `split_and_load`, to synchronize the data among multiple GPUs. Note that we don't need to write any specific code to achieve parallelism. Since the compute graph doesn't have any dependencies across devices within a minibatch, it is executed in parallel *automatically*.

```
def train_batch(X, y, gpu_params, ctx_list, lr):
    gpu_Xs, gpu_ys = split_batch(X, y, ctx_list)
    with autograd.record():  # Loss is calculated separately on each GPU
        losses = [loss(lenet(gpu_X, gpu_W), gpu_y)
                  for gpu_X, gpu_y, gpu_W in zip(gpu_Xs, gpu_ys, gpu_params)]
    for l in losses:  # Back Propagation is performed separately on each GPU
        l.backward()
    # Sum all gradients from each GPU and broadcast them to all GPUs
    for i in range(len(gpu_params[0])):
        allreduce([gpu_params[c][i].grad for c in range(len(ctx_list))])
    # The model parameters are updated separately on each GPU
    for param in gpu_params:
        d2l.sgd(param, lr, X.shape[0])  # Here, we use a full-size batch
```

Now, we can define the training function. It is slightly different from the ones used in the previous chapters: we need to allocate the GPUs and copy all the model parameters to all devices. Obviously each batch is processed using `train_batch` to deal with multiple GPUs. For convenience (and conciseness of code) we compute the accuracy on a single GPU (this is *inefficient* since the other GPUs are idle).

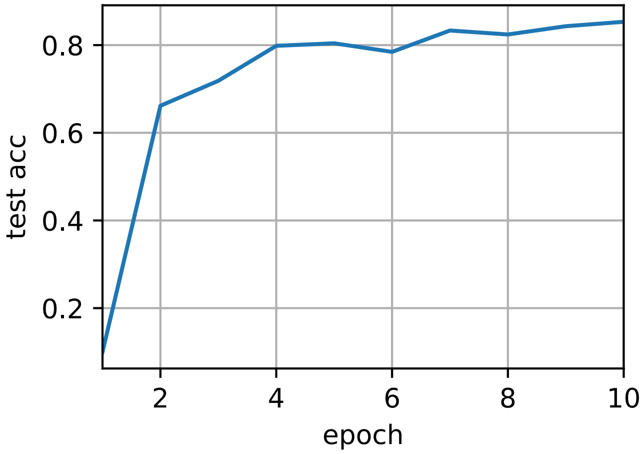
```
def train(num_gpus, batch_size, lr):
    train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)
    ctx_list = [d2l.try_gpu(i) for i in range(num_gpus)]
    # Copy model parameters to num_gpus GPUs
    gpu_params = [get_params(params, c) for c in ctx_list]
    # num_epochs, times, acces = 10, [], []
    num_epochs = 10
    animator = d2l.Animator('epoch', 'test acc', xlim=[1, num_epochs])
    timer = d2l.Timer()
    for epoch in range(num_epochs):
        timer.start()
        for X, y in train_iter:
            # Perform multi-GPU training for a single minibatch
            train_batch(X, y, gpu_params, ctx_list, lr)
            npx.waitall()
        timer.stop()
        # Verify the model on GPU 0
        animator.add(epoch+1, (d2l.evaluate_accuracy_gpu(
            lambda x: lenet(x, gpu_params[0]), test_iter, ctx[0]),))
    print('test acc: %.2f, %.1f sec/epoch on %s' % (
        animator.Y[0][-1], timer.avg(), ctx_list))
```

12.5.7 Experiment

Let's see how well this works on a single GPU. We use a batch size of 256 and a learning rate of 0.2.

```
train(num_gpus=1, batch_size=256, lr=0.2)
```

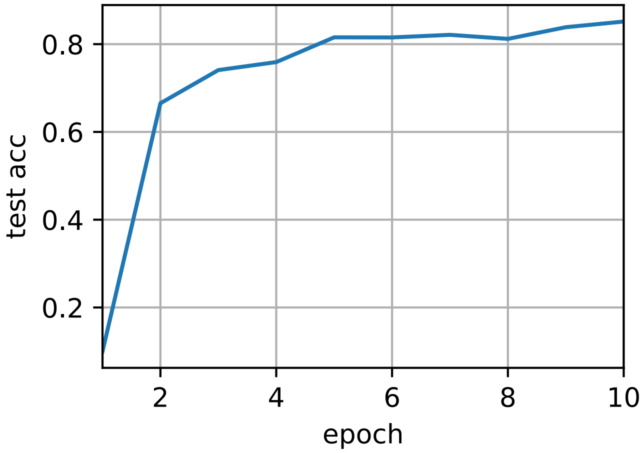
```
test acc: 0.85, 2.0 sec/epoch on [gpu(0)]
```



By keeping the batch size and learning rate unchanged and changing the number of GPUs to 2, we can see that the improvement in test accuracy is roughly the same as in the results from the previous experiment. In terms of the optimization algorithms, they are identical. Unfortunately there's no meaningful speedup to be gained here: the model is simply too small; moreover we only have a small dataset, where our slightly unsophisticated approach to implementing multi-GPU training suffered from significant Python overhead. We will encounter more complex models and more sophisticated ways of parallelization going forward. Let's see what happens nonetheless for MNIST.

```
train(num_gpus=2, batch_size=256, lr=0.2)
```

```
test acc: 0.85, 3.9 sec/epoch on [gpu(0), gpu(1)]
```



Summary

- There are multiple ways to split deep network training over multiple GPUs. We could split them between layers, across layers, or across data. The former two require tightly choreographed data transfers. Data parallelism is the simplest strategy.
- Data parallel training is straightforward. However, it increases the effective minibatch size to be efficient.
- Data is split across multiple GPUs, each GPU executes its own forward and backward operation and subsequently gradients are aggregated and results broadcast back to the GPUs.
- Large minibatches may require a slightly increased learning rate.

Exercises

1. When training on multiple GPUs, change the minibatch size from b to $k \cdot b$, i.e., scale it up by the number of GPUs.
2. Compare accuracy for different learning rates. How does it scale with the number of GPUs.
3. Implement a more efficient allreduce that aggregates different parameters on different GPUs (why is this more efficient in the first place).
4. Implement multi-GPU test accuracy computation.

12.6 Concise Implementation for Multiple GPUs

Implementing parallelism from scratch for every new model is no fun. Moreover, there's significant benefit in optimizing synchronization tools for high performance. In the following we'll show how to do this using Gluon. The math and the algorithms are the same as in Section 12.5. As before we begin by importing the required modules (quite unsurprisingly you'll need at least two GPUs to run this notebook).

```
import d2l
from mxnet import autograd, gluon, init, np, npx
from mxnet.gluon import nn
npx.set_np()
```

12.6.1 A Toy Network

Let's use a slightly more meaningful network than LeNet from the previous section that's still sufficiently easy and quick to train. We pick a ResNet-18 variant (He et al., 2016a). Since the input images are tiny we modify it slightly. In particular, the difference to Section 7.6 is that we use a smaller convolution kernel, stride, and padding at the beginning. Moreover, we remove the max-pooling layer.

```
# Saved in the d2l package for later use
def resnet18(num_classes):
    """A slightly modified ResNet-18 model."""
    def resnet_block(num_channels, num_residuals, first_block=False):
        blk = nn.Sequential()
        for i in range(num_residuals):
            if i == 0 and not first_block:
                blk.add(d2l.Residual(
                    num_channels, use_1x1conv=True, strides=2))
            else:
                blk.add(d2l.Residual(num_channels))
        return blk

    net = nn.Sequential()
    # This model uses a smaller convolution kernel, stride, and padding and
    # removes the maximum pooling layer
    net.add(nn.Conv2D(64, kernel_size=3, strides=1, padding=1),
            nn.BatchNorm(), nn.Activation('relu'))
    net.add(resnet_block(64, 2, first_block=True),
            resnet_block(128, 2),
            resnet_block(256, 2),
            resnet_block(512, 2))
    net.add(nn.GlobalAvgPool2D(), nn.Dense(num_classes))
    return net
```

12.6.2 Parameter Initialization and Logistics

The `initialize` method allows us to set initial defaults for parameters on a device of our choice. For a refresher see Section 4.8. What is particularly convenient is that it also lets us initialize the network on *multiple* devices simultaneously. Let's try how this works in practice.

```
net = resnet18(10)
# get a list of GPUs
ctx = d2l.try_all_gpus()
# initialize the network on all of them
net.initialize(init=init.Normal(sigma=0.01), ctx=ctx)
```

Using the `split_and_load` function introduced in the previous section we can divide a minibatch of data and copy portions to the list of devices provided by the context variable. The network object *automatically* uses the appropriate GPU to compute the value of the forward pass. As before we generate 4 observations and split them over the GPUs.

```
x = np.random.uniform(size=(4, 1, 28, 28))
gpu_x = gluon.utils.split_and_load(x, ctx)
net(gpu_x[0]), net(gpu_x[1])
```

```
(array([[ 2.2610193e-06,  2.2045974e-06, -5.4046782e-06,  1.2869954e-06,
          5.1373149e-06, -3.8298003e-06,  1.4339014e-07,  5.4683451e-06,
         -2.8279194e-06, -3.9651113e-06],
        [ 2.0698667e-06,  2.0084665e-06, -5.6382501e-06,  1.0498469e-06,
          5.5506416e-06, -4.1065468e-06,  6.0830143e-07,  5.4521765e-06,
         -3.7365030e-06, -4.1891640e-06]], ctx=gpu(0)),
 array([[ 2.4629794e-06,  2.6015521e-06, -5.4362622e-06,  1.2938231e-06,
          5.6387898e-06, -4.1360104e-06,  3.5758922e-07,  5.5125238e-06,
         -3.1957329e-06, -4.2976321e-06],
        [ 1.9431675e-06,  2.2600425e-06, -5.2698206e-06,  1.4807410e-06,
          5.4830930e-06, -3.9678889e-06,  7.5752268e-08,  5.6764361e-06,
         -3.2530238e-06, -4.0943960e-06]], ctx=gpu(1)))
```

Once data passes through the network, the corresponding parameters are initialized *on the device the data passed through*. This means that initialization happens on a per-device basis. Since we picked GPU 0 and GPU 1 for initialization, the network is initialized only there, and not on the CPU. In fact, the parameters don't even exist on the device. We can verify this by printing out the parameters and observing any errors that might arise.

```
weight = net[0].params.get('weight')

try:
    weight.data()
except RuntimeError:
    print('not initialized on cpu')
weight.data(ctx[0])[0], weight.data(ctx[1])[0]
```

```
not initialized on cpu
```

```
(array([[[ 0.01382882, -0.01183044,  0.01417866],
         [-0.00319718,  0.00439528,  0.02562625],
         [-0.00835081,  0.01387452, -0.01035946]]], ctx=gpu(0)),
 array([[[ 0.01382882, -0.01183044,  0.01417866],
         [-0.00319718,  0.00439528,  0.02562625],
         [-0.00835081,  0.01387452, -0.01035946]]], ctx=gpu(1)))
```

Lastly let's replace the code to evaluate the accuracy by one that works in parallel across multiple devices. This serves as a replacement of the evaluate_accuracy_gpu function from Section 6.6. The main difference is that we split a batch before invoking the network. All else is essentially identical.

```
# Saved in the d2l package for later use
def evaluate_accuracy_gpus(net, data_iter, split_f=d2l.split_batch):
    # Query the list of devices
    ctx = list(net.collect_params().values())[0].list_ctx()
    metric = d2l.Accumulator(2)  # num_corrected_examples, num_examples
    for features, labels in data_iter:
        Xs, ys = split_f(features, labels, ctx)
        pys = [net(X) for X in Xs]  # Run in parallel
        metric.add(sum(float(d2l.accuracy(py, y)) for py, y in zip(pys, ys)),
                   labels.size)
    return metric[0]/metric[1]
```

12.6.3 Training

As before, the training code needs to perform a number of basic functions for efficient parallelism:

- Network parameters need to be initialized across all devices.
- While iterating over the dataset minibatches are to be divided across all devices.
- We compute the loss and its gradient in parallel across devices.
- Losses are aggregated (by the trainer method) and parameters are updated accordingly.

In the end we compute the accuracy (again in parallel) to report the final value of the network. The training routine is quite similar to implementations in previous chapters, except that we need to split and aggregate data.

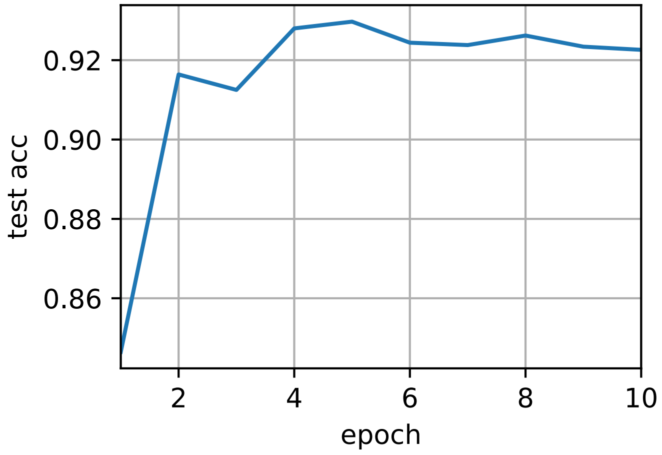
```
def train(num_gpus, batch_size, lr):
    train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)
    ctx = [d2l.try_gpu(i) for i in range(num_gpus)]
    net.initialize(init=init.Normal(sigma=0.01), ctx=ctx, force_reinit=True)
    trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': lr})
    loss = gluon.loss.SoftmaxCrossEntropyLoss()
    timer, num_epochs = d2l.Timer(), 10
    animator = d2l.Animator('epoch', 'test acc', xlim=[1, num_epochs])
    for epoch in range(num_epochs):
        timer.start()
        for features, labels in train_iter:
            Xs, ys = d2l.split_batch(features, labels, ctx)
            with autograd.record():
                losses = [loss(net(X), y) for X, y in zip(Xs, ys)]
            for l in losses:
                l.backward()
            trainer.step(batch_size)
        npx.waitall()
        timer.stop()
        animator.add(epoch+1, (evaluate_accuracy_gpus(net, test_iter),))
    print('test acc: %.2f, %.1f sec/epoch on %s' % (
        animator.Y[0][-1], timer.avg(), ctx))
```

12.6.4 Experiments

Let's see how this works in practice. As a warmup we train the network on a single GPU.

```
train(num_gpus=1, batch_size=256, lr=0.1)
```

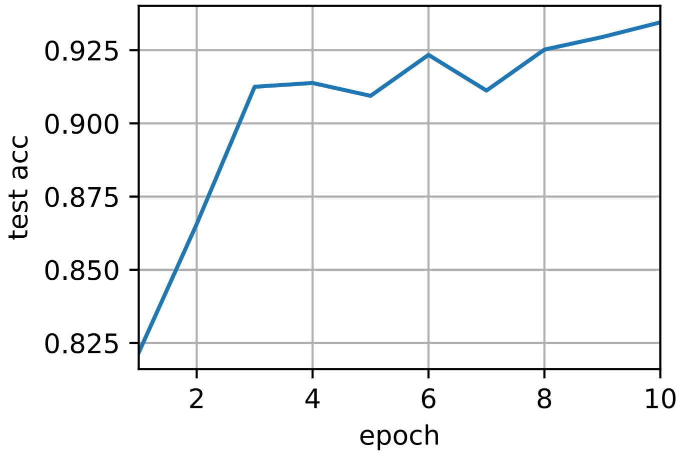
```
test acc: 0.92, 13.1 sec/epoch on [gpu(0)]
```



Next we use 2 GPUs for training. Compared to LeNet the model for ResNet-18 is considerably more complex. This is where parallelization shows its advantage. The time for computation is meaningfully larger than the time for synchronizing parameters. This improves scalability since the overhead for parallelization is less relevant.

```
train(num_gpus=2, batch_size=512, lr=0.2)
```

```
test acc: 0.93, 6.7 sec/epoch on [gpu(0), gpu(1)]
```



Summary

- Gluon provides primitives for model initialization across multiple devices by providing a context list.
- Data is automatically evaluated on the devices where the data can be found.
- Take care to initialize the networks on each device before trying to access the parameters on that device. Otherwise you will encounter an error.
- The optimization algorithms automatically aggregate over multiple GPUs.

Exercises

1. This section uses ResNet-18. Try different epochs, batch sizes, and learning rates. Use more GPUs for computation. What happens if you try this on a p2.16xlarge instance with 16 GPUs?
2. Sometimes, different devices provide different computing power. We could use the GPUs and the CPU at the same time. How should we divide the work? Is it worth the effort? Why? Why not?
3. What happens if we drop `npx.waitall()`? How would you modify training such that you have an overlap of up to two steps for parallelism?

12.7 Parameter Servers

As we move from single GPUs to multiple GPUs and then to multiple servers containing multiple GPUs, possibly all spread out across multiple racks and network switches our algorithms for distributed and parallel training need to become much more sophisticated. Details matter since different interconnects have very different bandwidth (e.g. NVLink can offer up to 100GB/s across 6 links in an appropriate setting, PCIe 3.0 16x lanes offer 16GB/s while even high speed 100 GbE Ethernet only amounts to 10GB/s). At the same time it's unreasonable to expect that a statistical modeler be an expert in networking and systems.

The core idea of the parameter server was introduced in (Smola & Narayanamurthy, 2010) in the context of distributed latent variable models. A description of the push and pull semantics then followed in (Ahmed et al., 2012) and a description of the system and an open source library followed in (Li et al., 2014). In the following we will motivate the components needed for efficiency.

12.7.1 Data Parallel Training

Let's review the data parallel training approach to distributed training. We will use this to the exclusion of all others in this section since it's significantly simpler to implement in practice. There are virtually no use cases (besides deep learning on graphs) where any other strategy for parallelism is preferred since GPUs have plenty of memory nowadays. Fig. 12.7.1 describes the variant of data parallelism that we implemented in the previous section. The key aspect in it is that the aggregation of gradients occurs on GPU0 before the updated parameters are rebroadcast to all GPUs.

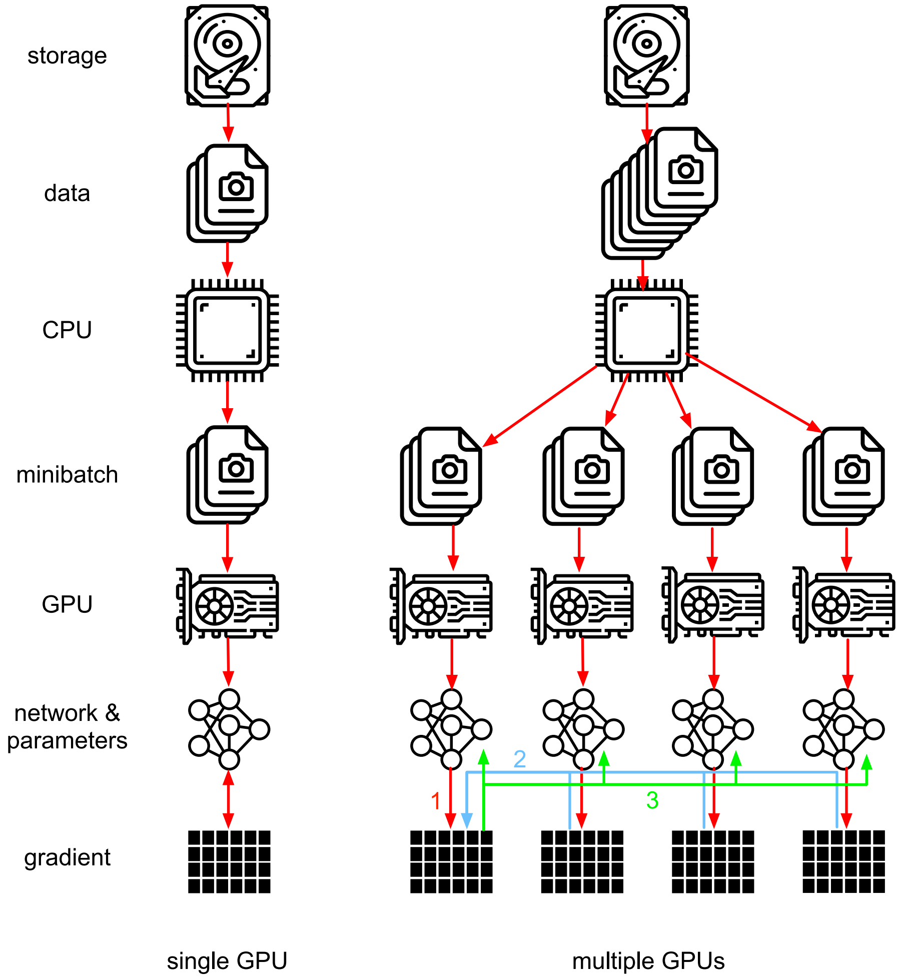


Fig. 12.7.1: Left: single GPU training; Right: a variant of multi-GPU training. It proceeds as follows. (1) we compute loss and gradient, (2) all gradients are aggregated on one GPU, (3) parameter update happens and the parameters are re-distributed to all GPUs.

In retrospect, the decision to aggregate on GPU0 seems rather ad-hoc. After all, we might just as well aggregate on the CPU. In fact, we could even decide to aggregate some of the parameters on one GPU and some others on another. Provided that the optimization algorithm supports this, there's no real reason for why we couldn't. For instance, if we have four parameter vectors $\mathbf{v}_1, \ldots \mathbf{v}_4$ with associated gradients $\mathbf{g}_1, \ldots \mathbf{g}_4$ we could aggregate the gradients on one GPU each.

$$\mathbf{g}_i = \sum_{j \in \mathrm{GPUs}} \mathbf{g}_{ij} \tag{12.7.1}$$

This reasoning seems arbitrary and frivolous. After all, the math is the same throughout. However, we are dealing with real physical hardware where different buses have different bandwidth as discussed in Section 12.4. Consider a real 4-way GPU server as described in Fig. 12.7.2. If it's particularly well connected, it might have a 100 GbE network card. More typical numbers are in the 1-10 GbE range with an effective bandwidth of 100MB/s to 1GB/s. Since the CPUs have too few PCIe lanes to connect to all GPUs directly (e.g. consumer grade Intel CPUs have 24 lanes) we need a multiplexer[193]. The bandwidth from the CPU on a 16x Gen3 link is 16GB/s. This is also the speed at which *each* of the GPUs is connected to the switch. This means that it's more effective to communicate between the

[193] https://www.broadcom.com/products/pcie-switches-bridges/pcie-switches

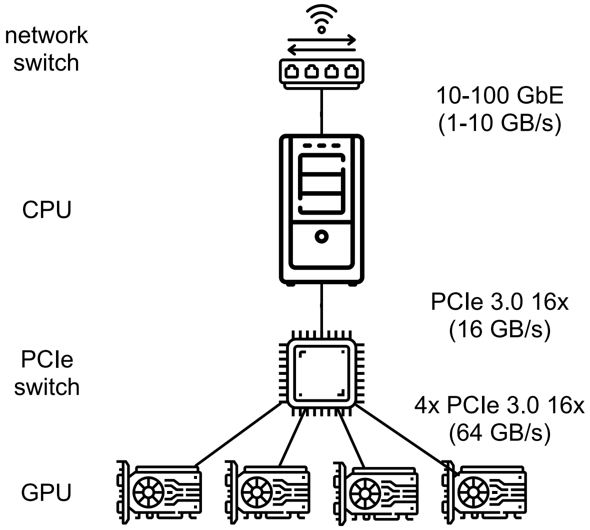


Fig. 12.7.2: Foo.

For the sake of the argument let's assume that the gradients 'weigh' 160MB. In this case it takes 30ms to send the gradients from all 3 remaining GPUs to the fourth one (each transfer takes 10ms = 160MB / 16 GB/s). Add another 30ms to transmit the weight vectors back we arrive at a total of 60ms. If we send all data to the CPU we incur a penalty of 40ms since *each* of the four GPUs needs to send the data to the CPU, yielding a total of 80ms. Lastly assume that we are able to split the gradients into 4 parts of 40MB each. Now we can aggregate each of the parts on a different GPU *simultaneously* since the PCIe switch offers a full-bandwidth operation between all links. Instead of 30ms this takes 7.5ms, yielding a total of 15ms for a synchronization operation. In short, depending on how we synchronize parameters the same operation can take anywhere from 15ms to 80ms. Fig. 12.7.3 depicts the different strategies for exchanging parameters.

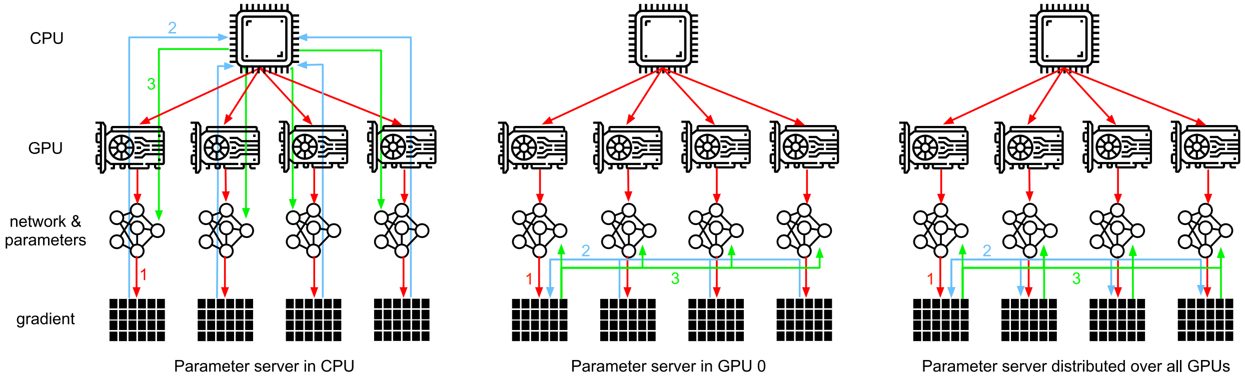

CPU
GPU
network & parameters
gradient
1
2
3
Parameter server in CPU
Parameter server in GPU 0
Parameter server distributed over all GPUs

Fig. 12.7.3: Synchronization strategies.

Note that we have yet another tool at our disposal when it comes to improving performance: in a deep network it takes some time to compute all gradients from the top to the bottom. We can begin synchronizing gradients for some parameter groups even while we're still busy computing them for others (the technical details for that are somewhat involved). See e.g. (Sergeev & DelBalso, 2018) for details on how to do this in Horovod[194].

[194] https://github.com/horovod/horovod

12.7.2 Ring Synchronization

When it comes to synchronization on modern deep learning hardware we often encounter significantly bespoke network connectivity. For instance, the AWS P3.16xlarge and NVIDIA DGX-2 instances share the connectivity structure of Fig. 12.7.4. Each GPU connects to a host CPU via a PCIe link which operates at best at 16 GB/s. Additionally each GPU also has 6 NVLink connections, each of which is capable of transferring 300 Gbit/s bidirectionally. This amounts to around 18 GB/s per link per direction. In short, the aggregate NVLink bandwidth is significantly higher than the PCIe bandwidth. The question is how to use it most efficiently.

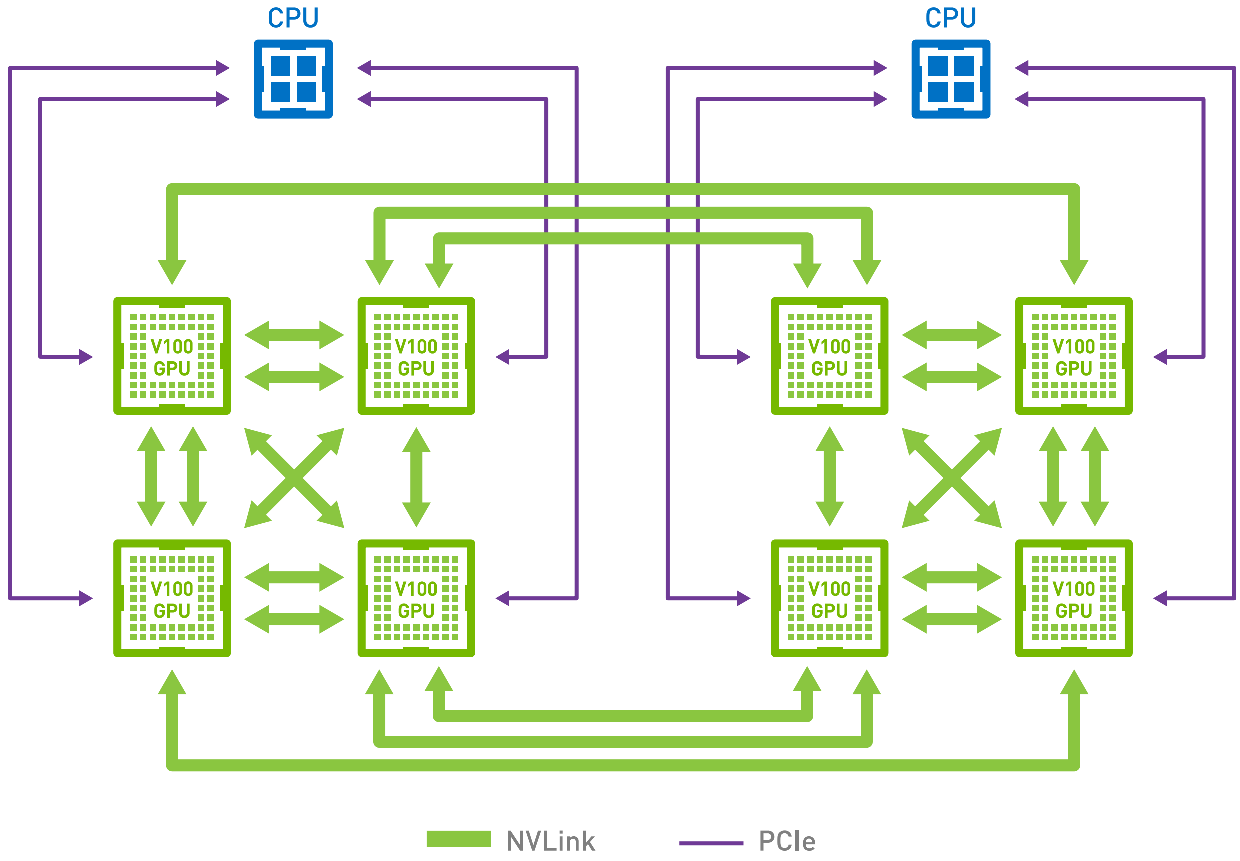


Fig. 12.7.4: NVLink connectivity on 8GPU V100 servers (image courtesy of NVIDIA).

It turns out (Wang et al., 2018) that the optimal synchronization strategy is to decompose the network into two rings and to use them to synchronize data directly. Fig. 12.7.5 illustrates that the network can be decomposed into one ring (1-2-3-4-5-6-7-8-1) with double NVLink bandwidth and into one (1-4-6-3-5-8-2-7-1) with regular bandwidth. Designing an efficient synchronization protocol in this case is nontrivial.

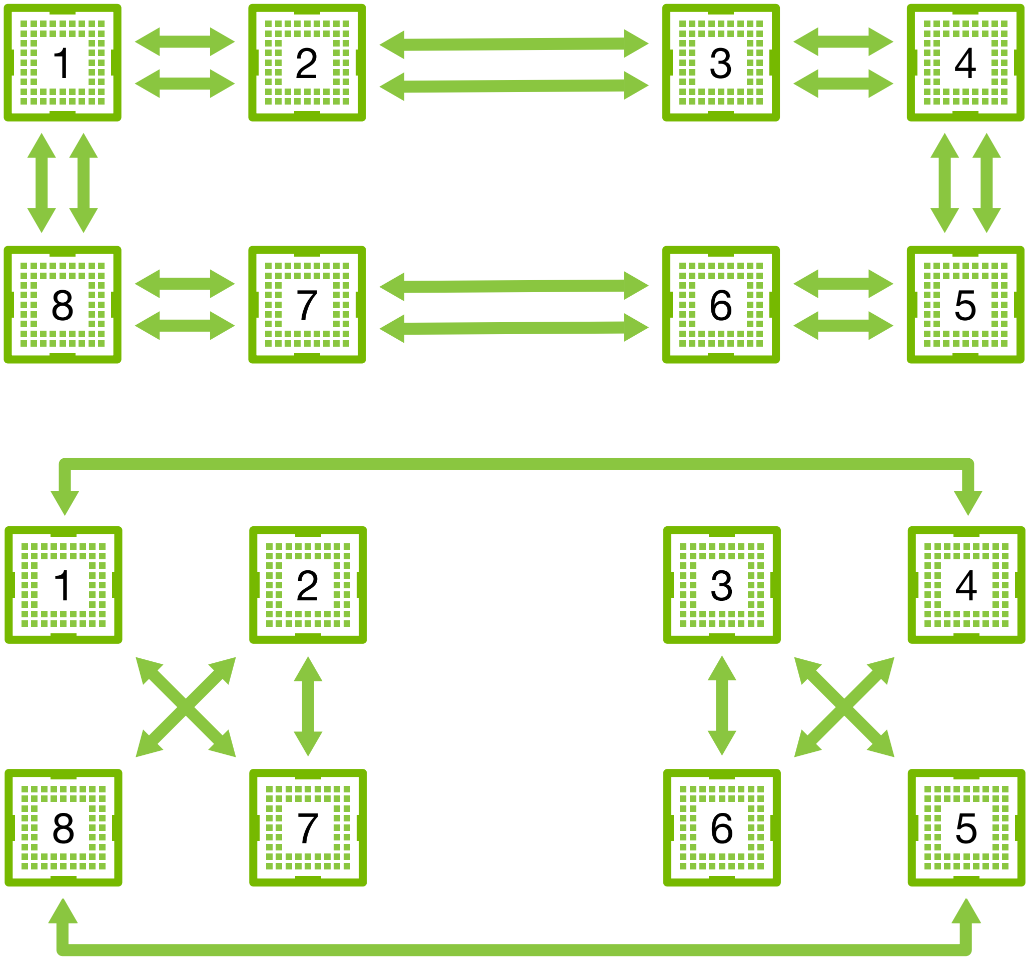


Fig. 12.7.5: Decomposition of the NVLink network into two rings.

Consider the following thought experiment: given a ring of n compute nodes (or GPUs) we can send gradients from the first to the second node. There it is added to the local gradient and sent on to the third node, and so on. After $n-1$ steps the aggregate gradient can be found in the last-visited node. That is, the time to aggregate gradients grows linearly with the number of nodes. But if we do this the algorithm is quite inefficient. After all, at any time there's only one of the nodes communicating. What if we broke the gradients into n chunks and started synchronizing chunk i starting at node i. Since each chunk is of site $1/n$ the total time is now $(n-1)/n \approx 1$. In other words, the time spent to aggregate gradients *does not grow* as we increase the size of the ring. This is quite an astonishing result. Fig. 12.7.6 illustrates the sequence of steps on $n = 4$ nodes.

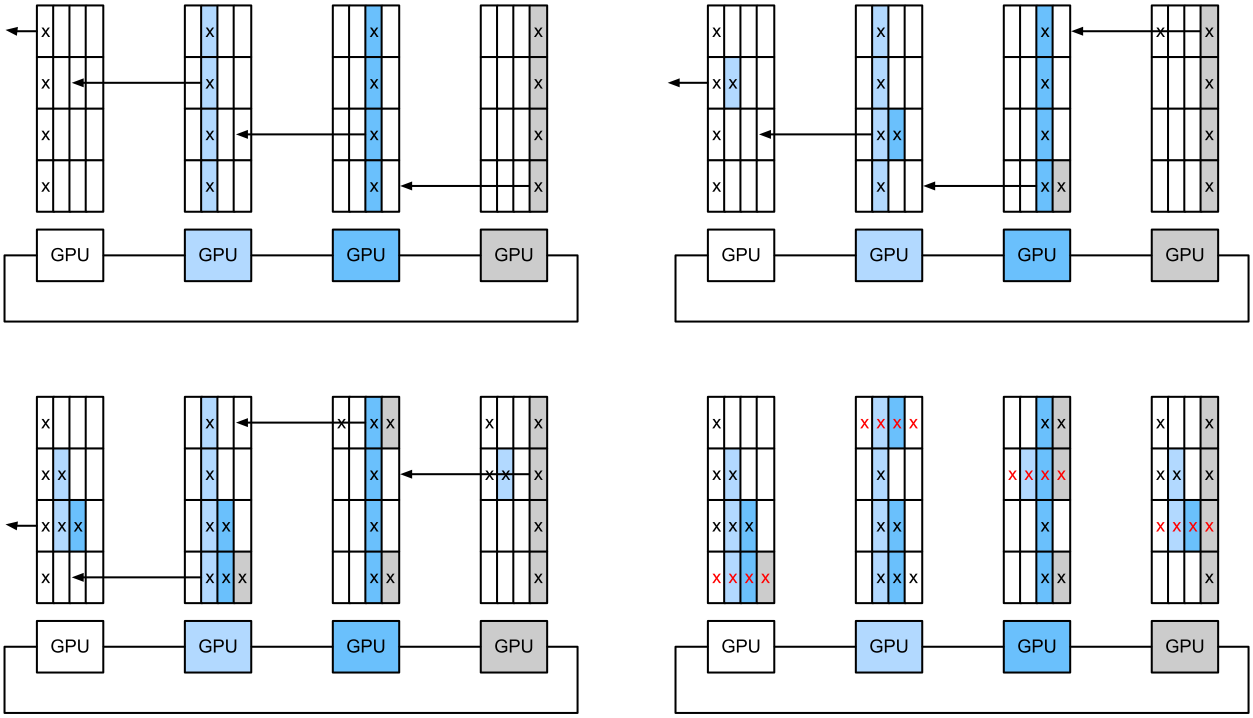


Fig. 12.7.6: Ring synchronization across 4 nodes. Each node starts transmitting parts of gradients to its left neighbor until the assembled gradient can be found in its right neighbor.

If we use the same example of synchronizing 160MB across 8 V100 GPUs we arrive at approximately $2 \cdot 160\text{MB}/(3 \cdot 18\text{GB/s}) \approx 6\text{ms}$ This is quite a bit better than using the PCIe bus, even though we are now using 8 GPUs. Note that in practice these numbers are quite a bit worse, since deep learning frameworks often fail to assemble communication into large burst transfers. Moreover, timing is critical. Note that there is a common misconception that ring synchronization is fundamentally different from other synchronization algorithms. The only difference is that the synchronization path is somewhat more elaborate when compared to a simple tree.

12.7.3 Multi-Machine Training

Distributed training on multiple machines adds a further challenge: we need to communicate with servers that are only connected across a comparatively lower bandwidth fabric which can be over an order of magnitude slower in some cases. Synchronization across devices is tricky. After all, different machines running training code will have subtly different speed. Hence we need to *synchronize* them if we want to use synchronous distributed optimization. Fig. 12.7.7 illustrates how distributed parallel training occurs.

1. A (different) batch of data is read on each machine, split across multiple GPUs and transferred to GPU memory. There predictions and gradients are computed on each GPU batch separately.
2. The gradients from all local GPUs are aggregated on one GPU (or alternatively parts of it are aggregated over different GPUs.
3. The gradients are sent to the CPU.
4. The CPU sends the gradients to a central parameter server which aggregates all the gradients.

5. The aggregate gradients are then used to update the weight vectors and the updated weight vectors are broadcast back to the individual CPUs.
6. The information is sent to one (or multiple) GPUs.
7. The updated weight vectors are spread across all GPUs.

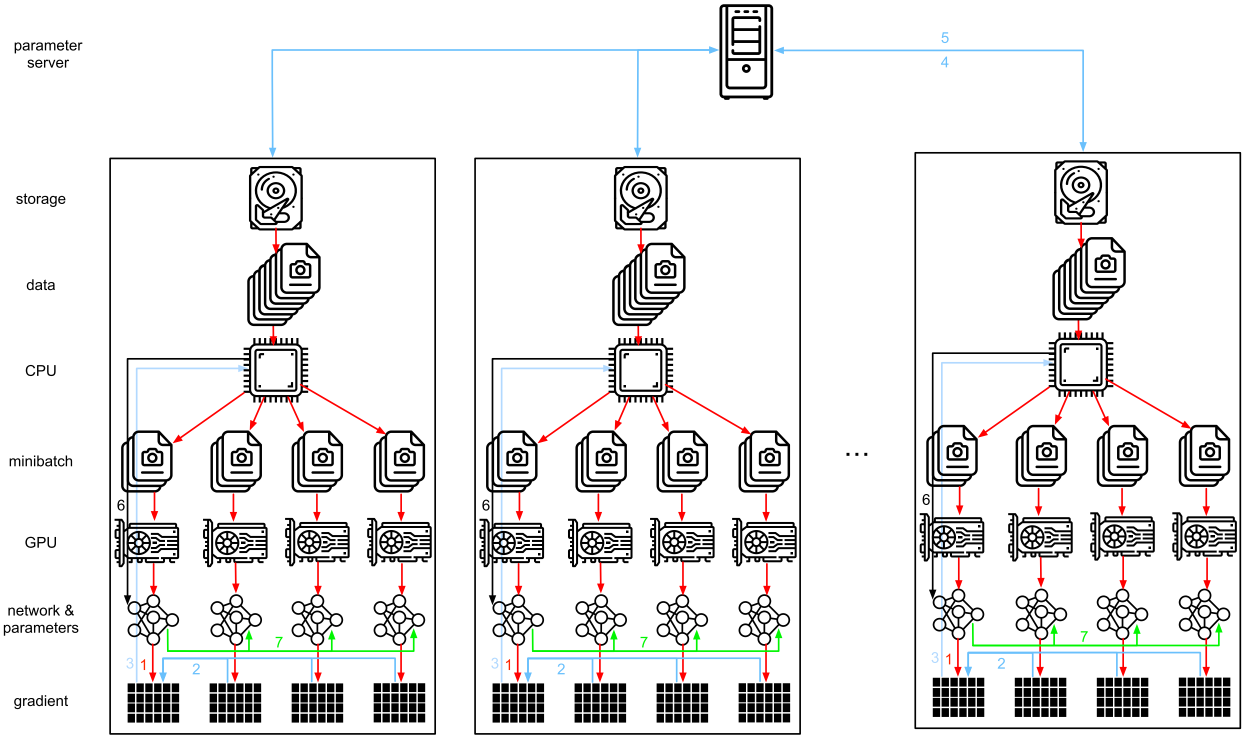
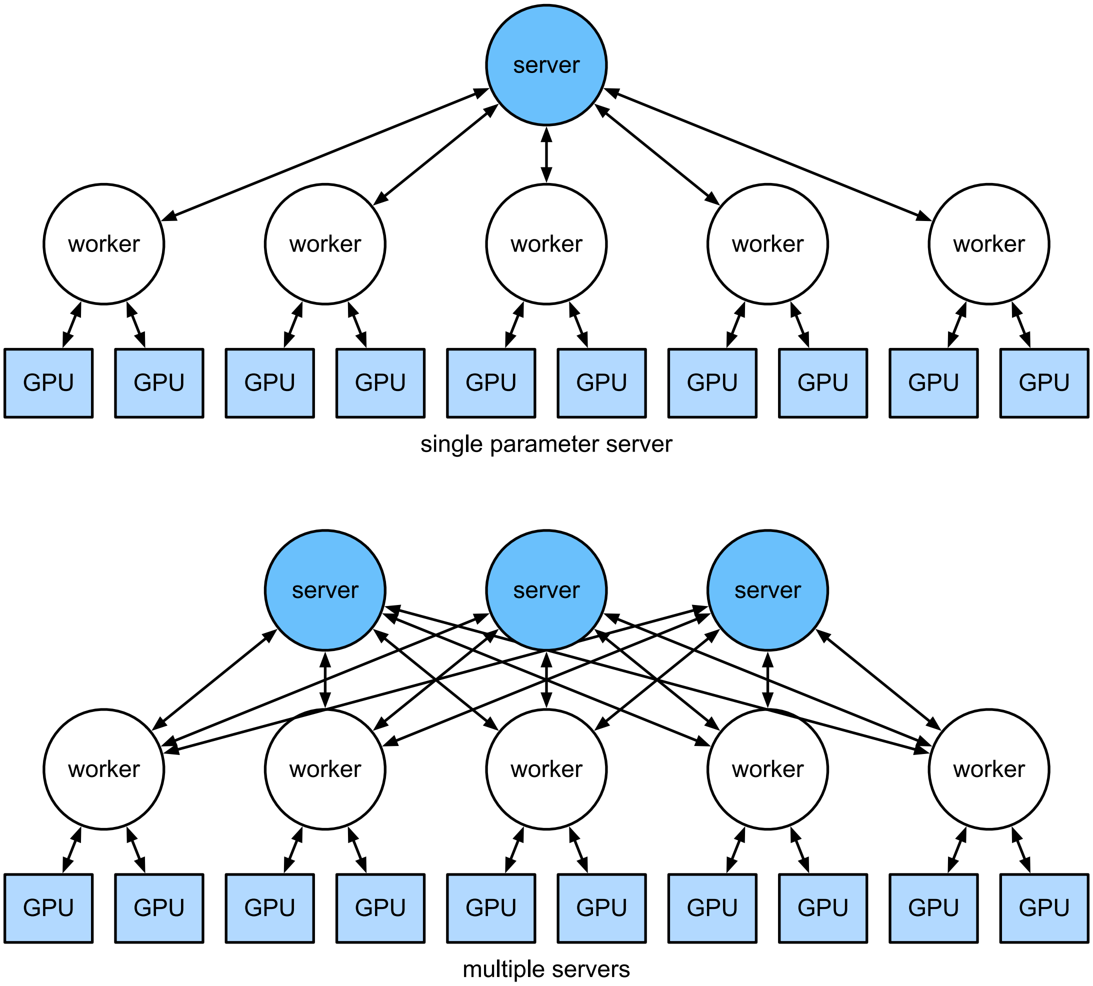


Fig. 12.7.7: Multi-machine multi-GPU distributed parallel training.

Each of these operations seems rather straightforward. And, indeed, they can be carried out efficiently *within* a single machine. Once we look at multiple machines, though, we can see that the central parameter server becomes the bottleneck. After all, the bandwidth per server is limited, hence for m workers the time it takes to send all gradients to the server is $O(m)$. We can break through this barrier by increasing the number of servers to n. At this point each server only needs to store $O(1/n)$ of the parameters, hence the total time for updates and optimization becomes $O(m/n)$. Matching both numbers yields constant scaling regardless of how many workers we are dealing with. In practice we use the *same* machines both as workers and as servers. Fig. 12.7.8 illustrates the design. See also (Li et al., 2014) for details. In particular, ensuring that multiple machines work without unreasonable delays is nontrivial. We omit details on barriers and will only briefly touch on synchronous and asynchronous updates below.



single parameter server

multiple servers

Fig. 12.7.8: Top - a single parameter server is a bottleneck since its bandwidth is finite. Bottom - multiple parameter servers store parts of the parameters with aggregate bandwidth.

12.7.4 (key,value) Stores

Implementing the steps required for distributed multi-GPU training in practice is nontrivial. In particular, given the many different choices that we might encounter. This is why it pays to use a common abstraction, namely that of a (key,value) store with redefined update semantics. Across many servers and many GPUs the gradient computation can be defined as

$$\mathbf{g}_{i} = \sum_{k \in \text{workers}} \sum_{j \in \text{GPUs}} \mathbf{g}_{ijk}. \tag{12.7.2}$$

The key aspect in this operation is that it is a *commutative reduction*, that is, it turns many vectors into one and the order in which the operation is applied doesn't matter. This is great for our purposes since we don't (need to) have fine grained control over when which gradient is received. Note that it's possible for us to perform the reduction stagewise. Furthermore, note that this operation is independent between blocks i pertaining to different parameters (and gradients).

This allows us to define the following two operations: push, which accumulates gradients, and pull, which retrieves aggregate gradients. Since we have many different sets of gradients (after all, we have many layers), we need to index the gradients with a key i. This similarity to (key,value) stores, such as the one introduced in Dynamo (DeCandia et al., 2007) is not by coincidence. They, too, satisfy many similar characteristics, in particular when it comes to distributing the parameters across multiple servers.

- **push(key, value)** sends a particular gradient (the value) from a worker to a common storage. There the parameter is aggregated, e.g. by summing it up.
- **pull(key, value)** retrieves an aggregate parameter from common storage, e.g. after combining the gradients from all workers.

By hiding all the complexity about synchronization behind a simple push and pull operation we can decouple the concerns of the statistical modeler who wants to be able to express optimization in simple terms and the systems engineer who needs to deal with the complexity inherent in distributed synchronization. In the next section we will experiment with such a (key,value) store in practice.

Summary

- Synchronization needs to be highly adaptive to specific network infrastructure and connectivity within a server. This can make a significant difference to the time it takes to synchronize.
- Ring-synchronization can be optimal for P3 and DGX-2 servers. For others possibly not so much.
- A hierarchical synchronization strategy works well when adding multiple parameter servers for increased bandwidth.
- Asynchronous communication (while computation is still ongoing) can improve performance.

Exercises

1. Can you increase the ring synchronization even further? Hint - you can send messages in both directions.
2. Fully asynchronous. Some delays permitted?
3. Fault tolerance. How? What if we lose a server? Is this a problem?
4. Checkpointing
5. Tree aggregation. Can you do it faster?
6. Other reductions (commutative semiring).

13 | Computer Vision

Many applications in the area of computer vision are closely related to our daily lives, now and in the future, whether medical diagnostics, driverless vehicles, camera monitoring, or smart filters. In recent years, deep learning technology has greatly enhanced computer vision systems' performance. It can be said that the most advanced computer vision applications are nearly inseparable from deep learning.

We have introduced deep learning models commonly used in the area of computer vision in the chapter "Convolutional Neural Networks" and have practiced simple image classification tasks. In this chapter, we will introduce image augmentation and fine tuning methods and apply them to image classification. Then, we will explore various methods of object detection. After that, we will learn how to use fully convolutional networks to perform semantic segmentation on images. Then, we explain how to use style transfer technology to generate images that look like the cover of this book. Finally, we will perform practice exercises on two important computer vision datasets to review the content of this chapter and the previous chapters.

13.1 Image Augmentation

We mentioned that large-scale datasets are prerequisites for the successful application of deep neural networks in Section 7.1. Image augmentation technology expands the scale of training datasets by making a series of random changes to the training images to produce similar, but different, training examples. Another way to explain image augmentation is that randomly changing training examples can reduce a model's dependence on certain properties, thereby improving its capability for generalization. For example, we can crop the images in different ways, so that the objects of interest appear in different positions, reducing the model's dependence on the position where objects appear. We can also adjust the brightness, color, and other factors to reduce model's sensitivity to color. It can be said that image augmentation technology contributed greatly to the success of AlexNet. In this section we will discuss this technology, which is widely used in computer vision.

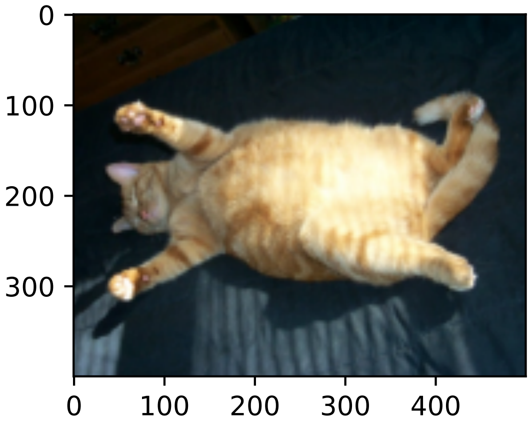
First, import the packages or modules required for the experiment in this section.

```
%matplotlib inline
import d2l
from mxnet import autograd, gluon, image, init, np, npx
from mxnet.gluon import nn

npx.set_np()
```

13.1.1 Common Image Augmentation Method

In this experiment, we will use an image with a shape of 400×500 as an example.

```
d2l.set_figsize((3.5, 2.5))
img = image.imread('../img/cat1.jpg')
d2l.plt.imshow(img.asnumpy());
```



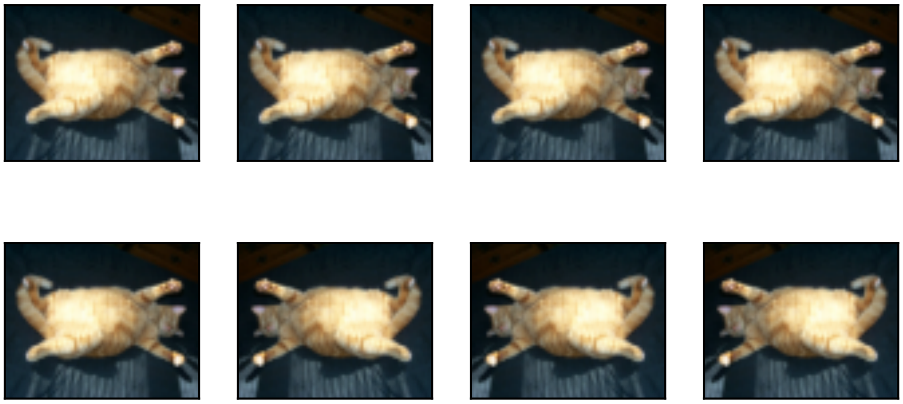
Most image augmentation methods have a certain degree of randomness. To make it easier for us to observe the effect of image augmentation, we next define the auxiliary function `apply`. This function runs the image augmentation method `aug` multiple times on the input image `img` and shows all results.

```
def apply(img, aug, num_rows=2, num_cols=4, scale=1.5):
    Y = [aug(img) for _ in range(num_rows * num_cols)]
    d2l.show_images(Y, num_rows, num_cols, scale=scale)
```

Flipping and Cropping

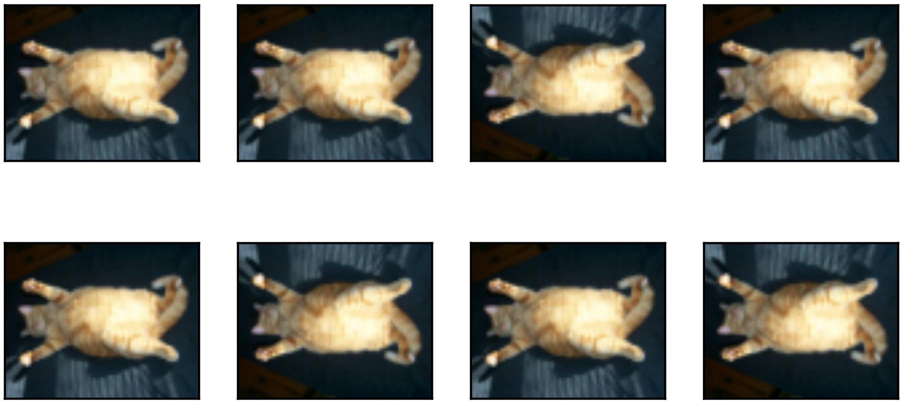
Flipping the image left and right usually does not change the category of the object. This is one of the earliest and most widely used methods of image augmentation. Next, we use the `transforms` module to create the `RandomFlipLeftRight` instance, which introduces a 50% chance that the image is flipped left and right.

```
apply(img, gluon.data.vision.transforms.RandomFlipLeftRight())
```



Flipping up and down is not as commonly used as flipping left and right. However, at least for this example image, flipping up and down does not hinder recognition. Next, we create a `RandomFlipTopBottom` instance for a 50% chance of flipping the image up and down.

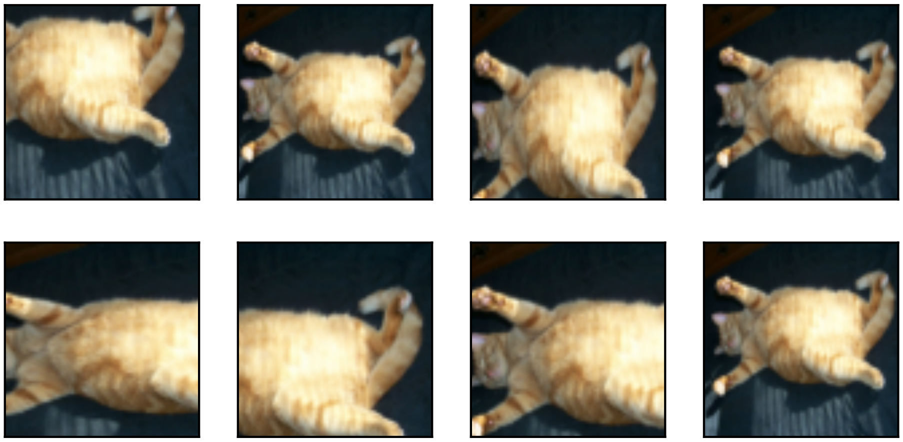
```
apply(img, gluon.data.vision.transforms.RandomFlipTopBottom())
```



In the example image we used, the cat is in the middle of the image, but this may not be the case for all images. In Section 6.5, we explained that the pooling layer can reduce the sensitivity of the convolutional layer to the target location. In addition, we can make objects appear at different positions in the image in different proportions by randomly cropping the image. This can also reduce the sensitivity of the model to the target position.

In the following code, we randomly crop a region with an area of 10% to 100% of the original area, and the ratio of width to height of the region is randomly selected from between 0.5 and 2. Then, the width and height of the region are both scaled to 200 pixels. Unless otherwise stated, the random number between a and b in this section refers to a continuous value obtained by uniform sampling in the interval $[a, b]$.

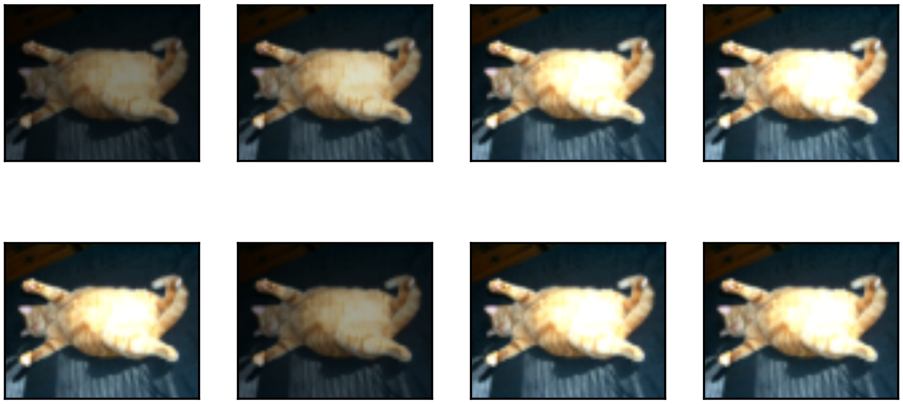
```
shape_aug = gluon.data.vision.transforms.RandomResizedCrop(
    (200, 200), scale=(0.1, 1), ratio=(0.5, 2))
apply(img, shape_aug)
```



Changing the Color

Another augmentation method is changing colors. We can change four aspects of the image color: brightness, contrast, saturation, and hue. In the example below, we randomly change the brightness of the image to a value between 50% $(1 - 0.5)$ and 150% $(1 + 0.5)$ of the original image.

```
apply(img, gluon.data.vision.transforms.RandomBrightness(0.5))
```

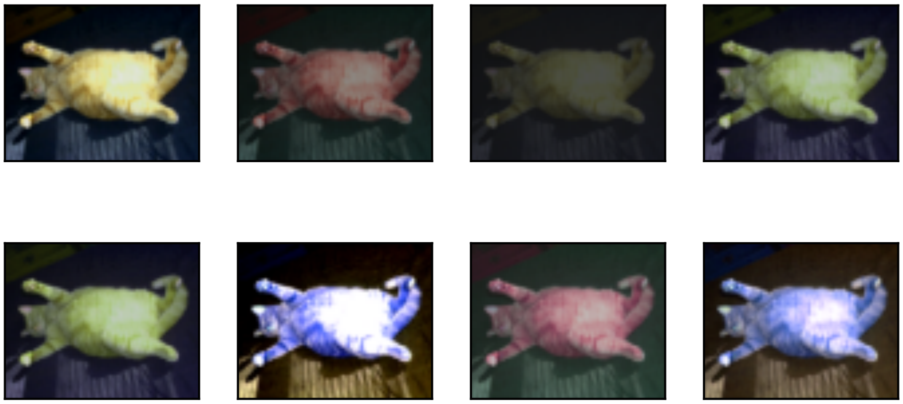


Similarly, we can randomly change the hue of the image.

```
apply(img, gluon.data.vision.transforms.RandomHue(0.5))
```

We can also create a `RandomColorJitter` instance and set how to randomly change the `brightness`, `contrast`, `saturation`, and `hue` of the image at the same time.

```
color_aug = gluon.data.vision.transforms.RandomColorJitter(
    brightness=0.5, contrast=0.5, saturation=0.5, hue=0.5)
apply(img, color_aug)
```



Overlying Multiple Image Augmentation Methods

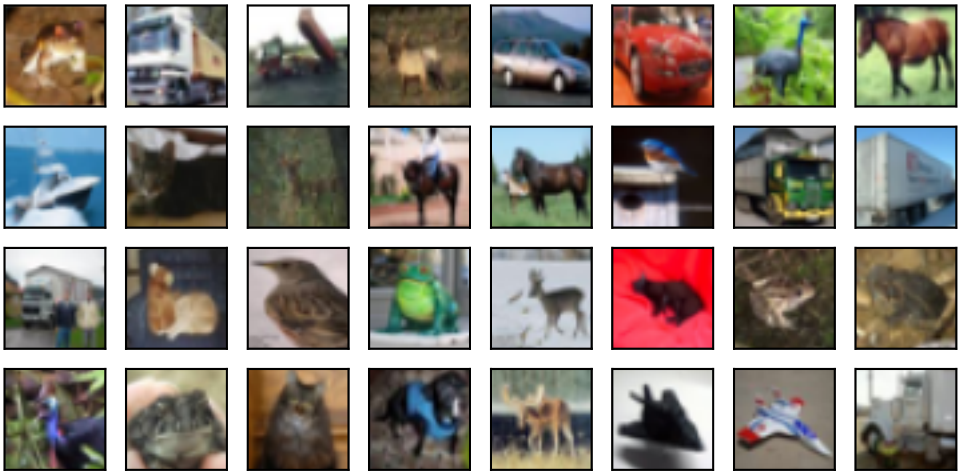
In practice, we will overlay multiple image augmentation methods. We can overlay the different image augmentation methods defined above and apply them to each image by using a `Compose` instance.

```
augs = gluon.data.vision.transforms.Compose([
    gluon.data.vision.transforms.RandomFlipLeftRight(), color_aug, shape_aug])
apply(img, augs)
```

13.1.2 Using an Image Augmentation Training Model

Next, we will look at how to apply image augmentation in actual training. Here, we use the CIFAR-10 dataset, instead of the Fashion-MNIST dataset we have been using. This is because the position and size of the objects in the Fashion-MNIST dataset have been normalized, and the differences in color and size of the objects in CIFAR-10 dataset are more significant. The first 32 training images in the CIFAR-10 dataset are shown below.

```
d2l.show_images(gluon.data.vision.CIFAR10(
    train=True)[0:32][0], 4, 8, scale=0.8);
```



In order to obtain a definitive results during prediction, we usually only apply image augmentation to the training example, and do not use image augmentation with random operations during prediction. Here, we only use the simplest random left-right flipping method. In addition, we use a `ToTensor` instance to convert minibatch images into the format required by MXNet, i.e., 32-bit floating point numbers with the shape of (batch size, number of channels, height, width) and value range between 0 and 1.

```
train_augs = gluon.data.vision.transforms.Compose([
    gluon.data.vision.transforms.RandomFlipLeftRight(),
    gluon.data.vision.transforms.ToTensor()])

test_augs = gluon.data.vision.transforms.Compose([
    gluon.data.vision.transforms.ToTensor()])
```

Next, we define an auxiliary function to make it easier to read the image and apply image augmentation. The `transform_first` function provided by Gluon's dataset applies image augmentation to the first element of each training example (image and label), i.e., the element at the top of the image. For detailed description of `DataLoader`, refer to Section 3.5.

```
def load_cifar10(is_train, augs, batch_size):
    return gluon.data.DataLoader(
        gluon.data.vision.CIFAR10(train=is_train).transform_first(augs),
        batch_size=batch_size, shuffle=is_train,
        num_workers=d2l.get_dataloader_workers())
```

Using a Multi-GPU Training Model

We train the ResNet-18 model described in Section 7.6 on the CIFAR-10 dataset. We will also apply the methods described in Section 12.6 and use a multi-GPU training model.

Next, we define the training function to train and evaluate the model using multiple GPUs.

```
# Saved in the d2l package for later use
def train_batch_ch13(net, features, labels, loss, trainer, ctx_list,
                     split_f=d2l.split_batch):
    Xs, ys = split_f(features, labels, ctx_list)
    with autograd.record():
        pys = [net(X) for X in Xs]
        ls = [loss(py, y) for py, y in zip(pys, ys)]
    for l in ls:
        l.backward()
    trainer.step(features.shape[0])
    train_loss_sum = sum([float(l.sum()) for l in ls])
    train_acc_sum = sum(d2l.accuracy(py, y) for py, y in zip(pys, ys))
    return train_loss_sum, train_acc_sum
```

```
# Saved in the d2l package for later use
def train_ch13(net, train_iter, test_iter, loss, trainer, num_epochs,
               ctx_list=d2l.try_all_gpus(), split_f=d2l.split_batch):
    num_batches, timer = len(train_iter), d2l.Timer()
    animator = d2l.Animator(xlabel='epoch', xlim=[0, num_epochs], ylim=[0, 1],
                            legend=['train loss', 'train acc', 'test acc'])
    for epoch in range(num_epochs):
        # Store training_loss, training_accuracy, num_examples, num_features
        metric = d2l.Accumulator(4)
        for i, (features, labels) in enumerate(train_iter):
            timer.start()
            l, acc = train_batch_ch13(
                net, features, labels, loss, trainer, ctx_list, split_f)
```

(continues on next page)

(continued from previous page)

```
            metric.add(l, acc, labels.shape[0], labels.size)
            timer.stop()
            if (i+1) % (num_batches // 5) == 0:
                animator.add(epoch+i/num_batches,
                             (metric[0]/metric[2], metric[1]/metric[3], None))
        test_acc = d2l.evaluate_accuracy_gpus(net, test_iter, split_f)
        animator.add(epoch+1, (None, None, test_acc))
    print('loss %.3f, train acc %.3f, test acc %.3f' % (
        metric[0]/metric[2], metric[1]/metric[3], test_acc))
    print('%.1f exampes/sec on %s' % (
        metric[2]*num_epochs/timer.sum(), ctx_list))
```

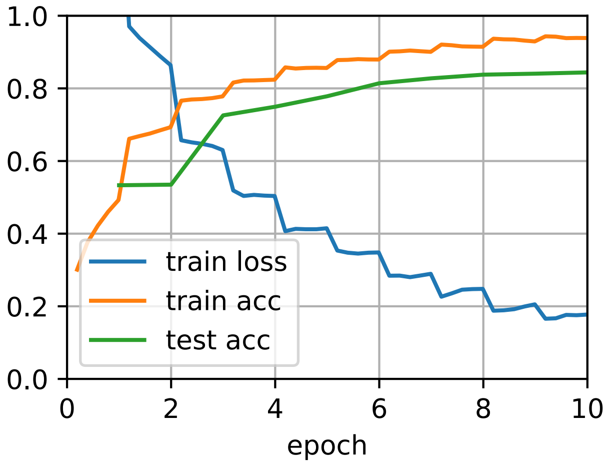
Now, we can define the `train_with_data_aug` function to use image augmentation to train the model. This function obtains all available GPUs and uses Adam as the optimization algorithm for training. It then applies image augmentation to the training dataset, and finally calls the `train` function just defined to train and evaluate the model.

```
batch_size, ctx, net = 256, d2l.try_all_gpus(), d2l.resnet18(10)
net.initialize(init=init.Xavier(), ctx=ctx)

def train_with_data_aug(train_augs, test_augs, net, lr=0.001):
    train_iter = load_cifar10(True, train_augs, batch_size)
    test_iter = load_cifar10(False, test_augs, batch_size)
    loss = gluon.loss.SoftmaxCrossEntropyLoss()
    trainer = gluon.Trainer(net.collect_params(), 'adam',
                            {'learning_rate': lr})
    train_ch13(net, train_iter, test_iter, loss, trainer, 10, ctx)
```

Now we train the model using image augmentation of random flipping left and right.

```
train_with_data_aug(train_augs, test_augs, net)
```

```
loss 0.177, train acc 0.939, test acc 0.844
5011.9 exampes/sec on [gpu(0), gpu(1)]
```



Summary

- Image augmentation generates random images based on existing training data to cope with overfitting.
- In order to obtain a definitive results during prediction, we usually only apply image augmentation to the training example, and do not use image augmentation with random operations during prediction.
- We can obtain classes related to image augmentation from Gluon's `transforms` module.

Exercises

1. Train the model without using image augmentation: `train_with_data_aug(no_aug, no_aug)`. Compare training and testing accuracy when using and not using image augmentation. Can this comparative experiment support the argument that image augmentation can mitigate overfitting? Why?
2. Add different image augmentation methods in model training based on the CIFAR-10 dataset. Observe the implementation results.
3. With reference to the MXNet documentation, what other image augmentation methods are provided in Gluon's `transforms` module?

13.2 Fine Tuning

In earlier chapters, we discussed how to train models on the Fashion-MNIST training dataset, which only has 60,000 images. We also described ImageNet, the most widely used large-scale image dataset in the academic world, with more than 10 million images and objects of over 1000 categories. However, the size of datasets that we often deal with is usually larger than the first, but smaller than the second.

Assume we want to identify different kinds of chairs in images and then push the purchase link to the user. One possible method is to first find a hundred common chairs, take one thousand different images with different angles for each chair, and then train a classification model on the collected image dataset. Although this dataset may be larger than Fashion-MNIST, the number of examples is still less than one tenth of ImageNet. This may result in the overfitting of the complicated model applicable to ImageNet on this dataset. At the same time, because of the limited amount of data, the accuracy of the final trained model may not meet the practical requirements.

In order to deal with the above problems, an obvious solution is to collect more data. However, collecting and labeling data can consume a lot of time and money. For example, in order to collect the ImageNet datasets, researchers have spent millions of dollars of research funding. Although, recently, data collection costs have dropped significantly, the costs still cannot be ignored.

Another solution is to apply transfer learning to migrate the knowledge learned from the source dataset to the target dataset. For example, although the images in ImageNet are mostly unrelated to chairs, models trained on this dataset can extract more general image features that can help identify edges, textures, shapes, and object composition. These similar features may be equally effective for recognizing a chair.

In this section, we introduce a common technique in transfer learning: fine tuning. As shown in Fig. 13.2.1, fine tuning consists of the following four steps:

1. Pre-train a neural network model, i.e., the source model, on a source dataset (e.g., the ImageNet dataset).
2. Create a new neural network model, i.e., the target model. This replicates all model designs and their parameters on the source model, except the output layer. We assume that these model parameters contain the knowledge learned from the source dataset and this knowledge will be equally applicable to the target dataset. We also assume that the output layer of the source model is closely related to the labels of the source dataset and is therefore not used in the target model.
3. Add an output layer whose output size is the number of target dataset categories to the target model, and randomly initialize the model parameters of this layer.
4. Train the target model on a target dataset, such as a chair dataset. We will train the output layer from scratch, while the parameters of all remaining layers are fine tuned based on the parameters of the source model.

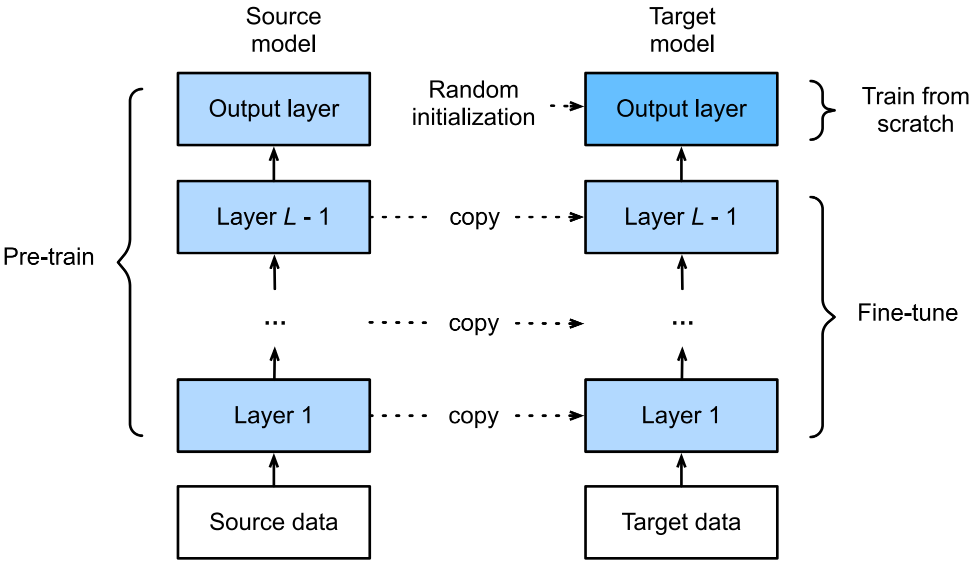


Fig. 13.2.1: Fine tuning.

13.2.1 Hot Dog Recognition

Next, we will use a specific example for practice: hot dog recognition. We will fine tune the ResNet model trained on the ImageNet dataset based on a small dataset. This small dataset contains thousands of images, some of which contain hot dogs. We will use the model obtained by fine tuning to identify whether an image contains a hot dog.

First, import the packages and modules required for the experiment. Gluon's `model_zoo` package provides a common pre-trained model. If you want to get more pre-trained models for computer vision, you can use the GluonCV Toolkit[197].

```
%matplotlib inline
import d2l
from mxnet import gluon, init, np, npx
from mxnet.gluon import nn
import os
import zipfile

npx.set_np()
```

Obtaining the Dataset

The hot dog dataset we use was taken from online images and contains $1,400$ positive images containing hot dogs and same number of negative images containing other foods. $1,000$ images of various classes are used for training and the rest are used for testing.

We first download the compressed dataset to the path `../data`. Then, we unzip the downloaded dataset in this path and get two folders, `hotdog/train` and `hotdog/test`. Both folders have `hotdog` and `not-hotdog` category subfolders, each of which has corresponding image files.

```
data_dir = '../data'
base_url = 'https://apache-mxnet.s3-accelerate.amazonaws.com/'
fname = gluon.utils.download(
    base_url + 'gluon/dataset/hotdog.zip',
    path=data_dir, sha1_hash='fba480ffa8aa7e0febbb511d181409f899b9baa5')
with zipfile.ZipFile(fname, 'r') as z:
    z.extractall(data_dir)
```

```
Downloading ../data/hotdog.zip from https://apache-mxnet.s3-accelerate.amazonaws.com/gluon/
↪dataset/hotdog.zip...
```

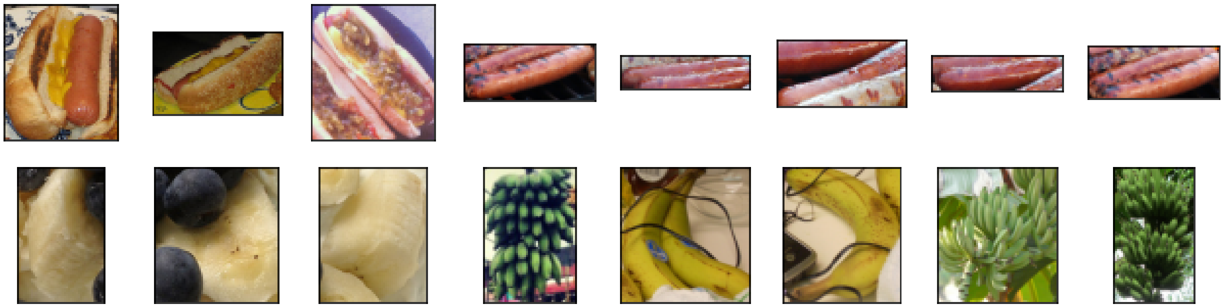
We create two `ImageFolderDataset` instances to read all the image files in the training dataset and testing dataset, respectively.

```
train_imgs = gluon.data.vision.ImageFolderDataset(
    os.path.join(data_dir, 'hotdog/train'))
test_imgs = gluon.data.vision.ImageFolderDataset(
    os.path.join(data_dir, 'hotdog/test'))
```

The first 8 positive examples and the last 8 negative images are shown below. As you can see, the images vary in size and aspect ratio.

```
hotdogs = [train_imgs[i][0] for i in range(8)]
not_hotdogs = [train_imgs[-i - 1][0] for i in range(8)]
d2l.show_images(hotdogs + not_hotdogs, 2, 8, scale=1.4);
```

[197] https://gluon-cv.mxnet.io



During training, we first crop a random area with random size and random aspect ratio from the image and then scale the area to an input with a height and width of 224 pixels. During testing, we scale the height and width of images to 256 pixels, and then crop the center area with height and width of 224 pixels to use as the input. In addition, we normalize the values of the three RGB (red, green, and blue) color channels. The average of all values of the channel is subtracted from each value and then the result is divided by the standard deviation of all values of the channel to produce the output.

```
# We specify the mean and variance of the three RGB channels to normalize the
# image channel
normalize = gluon.data.vision.transforms.Normalize(
    [0.485, 0.456, 0.406], [0.229, 0.224, 0.225])

train_augs = gluon.data.vision.transforms.Compose([
    gluon.data.vision.transforms.RandomResizedCrop(224),
    gluon.data.vision.transforms.RandomFlipLeftRight(),
    gluon.data.vision.transforms.ToTensor(),
    normalize])

test_augs = gluon.data.vision.transforms.Compose([
    gluon.data.vision.transforms.Resize(256),
    gluon.data.vision.transforms.CenterCrop(224),
    gluon.data.vision.transforms.ToTensor(),
    normalize])
```

Defining and Initializing the Model

We use ResNet-18, which was pre-trained on the ImageNet dataset, as the source model. Here, we specify `pretrained=True` to automatically download and load the pre-trained model parameters. The first time they are used, the model parameters need to be downloaded from the Internet.

```
pretrained_net = gluon.model_zoo.vision.resnet18_v2(pretrained=True)
```

The pre-trained source model instance contains two member variables: `features` and `output`. The former contains all layers of the model, except the output layer, and the latter is the output layer of the model. The main purpose of this division is to facilitate the fine tuning of the model parameters of all layers except the output layer. The member variable `output` of source model is given below. As a fully connected layer, it transforms ResNet's final global average pooling layer output into 1000 class output on the ImageNet dataset.

```
pretrained_net.output
```

```
Dense(512 -> 1000, linear)
```

We then build a new neural network to use as the target model. It is defined in the same way as the pre-trained source model, but the final number of outputs is equal to the number of categories in the target dataset. In the code below, the model parameters in the member variable `features` of the target model instance `finetune_net` are initialized to model parameters of the corresponding layer of the source model. Because the model parameters in `features` are obtained by pre-training on the ImageNet dataset, it is good enough. Therefore, we generally only need to use small learning rates to "fine tune" these parameters. In contrast, model parameters in the member variable `output` are randomly initialized and generally require a larger learning rate to learn from scratch. Assume the learning rate in the `Trainer` instance is η and use a learning rate of 10η to update the model parameters in the member variable `output`.

```
finetune_net = gluon.model_zoo.vision.resnet18_v2(classes=2)
finetune_net.features = pretrained_net.features
finetune_net.output.initialize(init.Xavier())
# The model parameters in output will be updated using a learning rate ten
# times greater
finetune_net.output.collect_params().setattr('lr_mult', 10)
```

Fine Tuning the Model

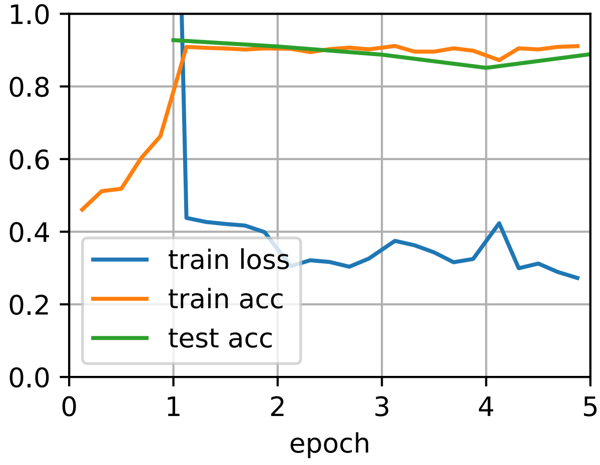
We first define a training function `train_fine_tuning` that uses fine tuning so it can be called multiple times.

```
def train_fine_tuning(net, learning_rate, batch_size=128, num_epochs=5):
    train_iter = gluon.data.DataLoader(
        train_imgs.transform_first(train_augs), batch_size, shuffle=True)
    test_iter = gluon.data.DataLoader(
        test_imgs.transform_first(test_augs), batch_size)
    ctx = d2l.try_all_gpus()
    net.collect_params().reset_ctx(ctx)
    net.hybridize()
    loss = gluon.loss.SoftmaxCrossEntropyLoss()
    trainer = gluon.Trainer(net.collect_params(), 'sgd', {
        'learning_rate': learning_rate, 'wd': 0.001})
    d2l.train_ch13(net, train_iter, test_iter, loss, trainer, num_epochs, ctx)
```

We set the learning rate in the `Trainer` instance to a smaller value, such as 0.01, in order to fine tune the model parameters obtained in pre-training. Based on the previous settings, we will train the output layer parameters of the target model from scratch using a learning rate ten times greater.

```
train_fine_tuning(finetune_net, 0.01)
```

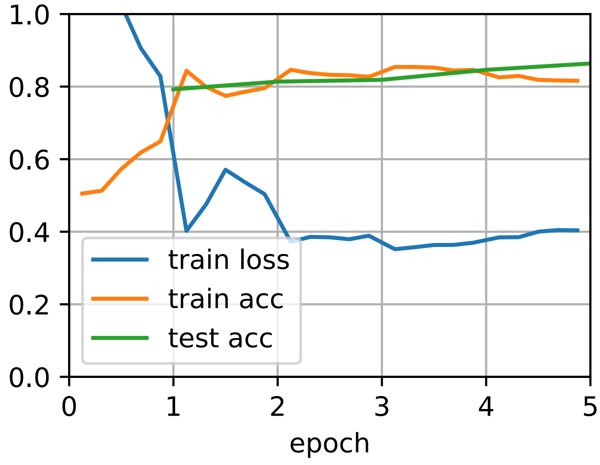
```
loss 0.276, train acc 0.909, test acc 0.889
696.4 exampes/sec on [gpu(0), gpu(1)]
```



For comparison, we define an identical model, but initialize all of its model parameters to random values. Since the entire model needs to be trained from scratch, we can use a larger learning rate.

```
scratch_net = gluon.model_zoo.vision.resnet18_v2(classes=2)
scratch_net.initialize(init=init.Xavier())
train_fine_tuning(scratch_net, 0.1)
```

```
loss 0.401, train acc 0.818, test acc 0.864
739.8 exampes/sec on [gpu(0), gpu(1)]
```



As you can see, the fine-tuned model tends to achieve higher precision in the same epoch because the initial values of the parameters are better.

Summary

- Transfer learning migrates the knowledge learned from the source dataset to the target dataset. Fine tuning is a common technique for transfer learning.
- The target model replicates all model designs and their parameters on the source model, except the output layer, and fine tunes these parameters based on the target dataset. In contrast, the output layer of the target model needs to be trained from scratch.
- Generally, fine tuning parameters use a smaller learning rate, while training the output layer from scratch can use a larger learning rate.

Exercises

1. Keep increasing the learning rate of `finetune_net`. How does the precision of the model change?
2. Further tune the hyper-parameters of `finetune_net` and `scratch_net` in the comparative experiment. Do they still have different precisions?
3. Set the parameters in `finetune_net.features` to the parameters of the source model and do not update them during training. What will happen? You can use the following code.

```
finetune_net.features.collect_params().setattr('grad_req', 'null')
```

4. In fact, there is also a "hotdog" class in the `ImageNet` dataset. Its corresponding weight parameter at the output layer can be obtained by using the following code. How can we use this parameter?

```
weight = pretrained_net.output.weight
hotdog_w = np.split(weight.data(), 1000, axis=0)[713]
hotdog_w.shape
```

```
(1, 512)
```

13.3 Object Detection and Bounding Boxes

In the previous section, we introduced many models for image classification. In image classification tasks, we assume that there is only one main target in the image and we only focus on how to identify the target category. However, in many situations, there are multiple targets in the image that we are interested in. We not only want to classify them, but also want to obtain their specific positions in the image. In computer vision, we refer to such tasks as object detection (or object recognition).

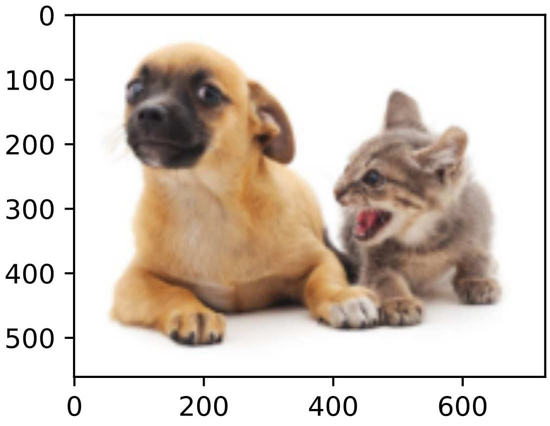
Object detection is widely used in many fields. For example, in self-driving technology, we need to plan routes by identifying the locations of vehicles, pedestrians, roads, and obstacles in the captured video image. Robots often perform this type of task to detect targets of interest. Systems in the security field need to detect abnormal targets, such as intruders or bombs.

In the next few sections, we will introduce multiple deep learning models used for object detection. Before that, we should discuss the concept of target location. First, import the packages and modules required for the experiment.

```
%matplotlib inline
import d2l
from mxnet import image, npx

npx.set_np()
```

Next, we will load the sample images that will be used in this section. We can see there is a dog on the left side of the image and a cat on the right. They are the two main targets in this image.

```
d2l.set_figsize((3.5, 2.5))
img = image.imread('../img/catdog.jpg').asnumpy()
d2l.plt.imshow(img);
```



13.3.1 Bounding Box

In object detection, we usually use a bounding box to describe the target location. The bounding box is a rectangular box that can be determined by the x and y axis coordinates in the upper-left corner and the x and y axis coordinates in the lower-right corner of the rectangle. We will define the bounding boxes of the dog and the cat in the image based on the coordinate information in the above image. The origin of the coordinates in the above image is the upper left corner of the image, and to the right and down are the positive directions of the x axis and the y axis, respectively.

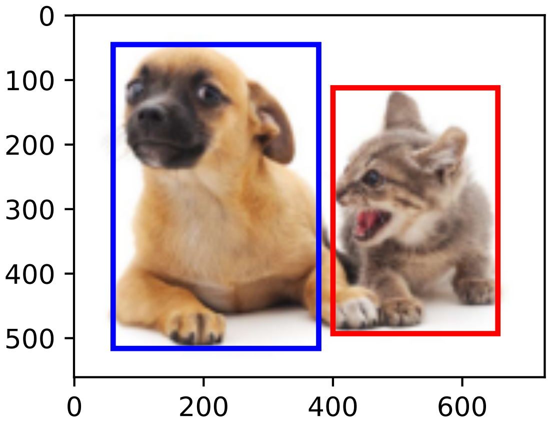
```
# bbox is the abbreviation for bounding box
dog_bbox, cat_bbox = [60, 45, 378, 516], [400, 112, 655, 493]
```

We can draw the bounding box in the image to check if it is accurate. Before drawing the box, we will define a helper function `bbox_to_rect`. It represents the bounding box in the bounding box format of `matplotlib`.

```
# Saved in the d2l package for later use
def bbox_to_rect(bbox, color):
    """Convert bounding box to matplotlib format."""
    # Convert the bounding box (top-left x, top-left y, bottom-right x,
    # bottom-right y) format to matplotlib format: ((upper-left x,
    # upper-left y), width, height)
    return d2l.plt.Rectangle(
        xy=(bbox[0], bbox[1]), width=bbox[2]-bbox[0], height=bbox[3]-bbox[1],
        fill=False, edgecolor=color, linewidth=2)
```

After loading the bounding box on the image, we can see that the main outline of the target is basically inside the box.

```
fig = d2l.plt.imshow(img)
fig.axes.add_patch(bbox_to_rect(dog_bbox, 'blue'))
fig.axes.add_patch(bbox_to_rect(cat_bbox, 'red'));
```



Summary

- In object detection, we not only need to identify all the objects of interest in the image, but also their positions. The positions are generally represented by a rectangular bounding box.

Exercises

1. Find some images and try to label a bounding box that contains the target. Compare the difference between the time it takes to label the bounding box and label the category.

13.4 Anchor Boxes

Object detection algorithms usually sample a large number of regions in the input image, determine whether these regions contain objects of interest, and adjust the edges of the regions so as to predict the ground-truth bounding box of the target more accurately. Different models may use different region sampling methods. Here, we introduce one such method: it generates multiple bounding boxes with different sizes and aspect ratios while centering on each pixel. These bounding boxes are called anchor boxes. We will practice object detection based on anchor boxes in the following sections.

First, import the packages or modules required for this section. Here, we have introduced the `contrib` package, and modified the printing accuracy of NumPy. Because printing `ndarrays` actually calls the print function of NumPy, the floating-point numbers in `ndarrays` printed in this section are more concise.

```
%matplotlib inline
import d2l
from mxnet import contrib, gluon, image, np, npx

np.set_printoptions(2)
npx.set_np()
```

13.4.1 Generating Multiple Anchor Boxes

Assume that the input image has a height of h and width of w. We generate anchor boxes with different shapes centered on each pixel of the image. Assume the size is $s \in (0, 1]$, the aspect ratio is $r > 0$, and the width and height of the anchor box are $ws\sqrt{r}$ and $hs/\sqrt{r}$, respectively. When the center position is given, an anchor box with known width and height is determined.

Below we set a set of sizes $s_1, \ldots, s_n$ and a set of aspect ratios $r_1, \ldots, r_m$. If we use a combination of all sizes and aspect ratios with each pixel as the center, the input image will have a total of $whnm$ anchor boxes. Although these anchor boxes may cover all ground-truth bounding boxes, the computational complexity is often excessive. Therefore, we are usually only interested in a combination containing s_1 or r_1 sizes and aspect ratios, that is:

$$(s_1, r_1), (s_1, r_2), \ldots, (s_1, r_m), (s_2, r_1), (s_3, r_1), \ldots, (s_n, r_1). \tag{13.4.1}$$

That is, the number of anchor boxes centered on the same pixel is $n + m - 1$. For the entire input image, we will generate a total of $wh(n + m - 1)$ anchor boxes.

The above method of generating anchor boxes has been implemented in the `MultiBoxPrior` function. We specify the input, a set of sizes, and a set of aspect ratios, and this function will return all the anchor boxes entered.

```
img = image.imread('../img/catdog.jpg').asnumpy()
h, w = img.shape[0:2]

print(h, w)
X = np.random.uniform(size=(1, 3, h, w))  # Construct input data
Y = npx.multibox_prior(X, sizes=[0.75, 0.5, 0.25], ratios=[1, 2, 0.5])
Y.shape
```

```
561 728
```

```
(1, 2042040, 4)
```

We can see that the shape of the returned anchor box variable y is (batch size, number of anchor boxes, 4). After changing the shape of the anchor box variable y to (image height, image width, number of anchor boxes centered on the same pixel, 4), we can obtain all the anchor boxes centered on a specified pixel position. In the following example, we access the first anchor box centered on (250, 250). It has four elements: the x, y axis coordinates in the upper-left corner and the x, y axis coordinates in the lower-right corner of the anchor box. The coordinate values of the x and y axis are divided by the width and height of the image, respectively, so the value range is between 0 and 1.

```
boxes = Y.reshape(h, w, 5, 4)
boxes[250, 250, 0, :]
```

```
array([0.06, 0.07, 0.63, 0.82])
```

In order to describe all anchor boxes centered on one pixel in the image, we first define the show_bboxes function to draw multiple bounding boxes on the image.

```
# Saved in the d2l package for later use
def show_bboxes(axes, bboxes, labels=None, colors=None):
    """Show bounding boxes."""
    def _make_list(obj, default_values=None):
        if obj is None:
            obj = default_values
        elif not isinstance(obj, (list, tuple)):
            obj = [obj]
        return obj
    labels = _make_list(labels)
    colors = _make_list(colors, ['b', 'g', 'r', 'm', 'c'])
    for i, bbox in enumerate(bboxes):
        color = colors[i % len(colors)]
        rect = d2l.bbox_to_rect(bbox.asnumpy(), color)
        axes.add_patch(rect)
        if labels and len(labels) > i:
            text_color = 'k' if color == 'w' else 'w'
            axes.text(rect.xy[0], rect.xy[1], labels[i],
                      va='center', ha='center', fontsize=9, color=text_color,
                      bbox=dict(facecolor=color, lw=0))
```

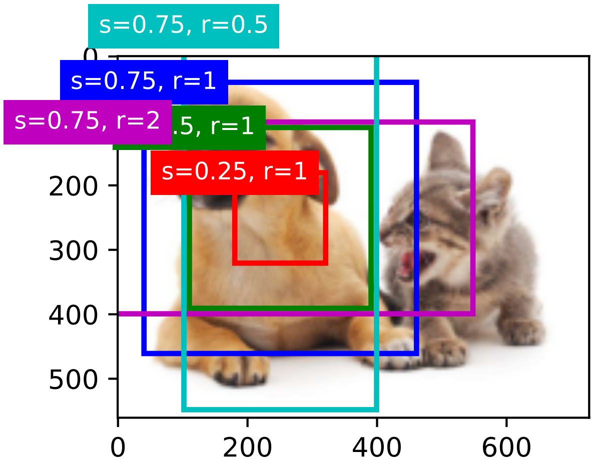
As we just saw, the coordinate values of the x and y axis in the variable boxes have been divided by the width and height of the image, respectively. When drawing images, we need to restore the original coordinate values of the anchor boxes and therefore define the variable bbox_scale. Now, we can draw all the anchor boxes centered on (250, 250) in the image. As you can see, the blue anchor box with a size of 0.75 and an aspect ratio of 1 covers the dog in the image well.

```
d2l.set_figsize((3.5, 2.5))
bbox_scale = np.array((w, h, w, h))
fig = d2l.plt.imshow(img)
```

(continues on next page)

(continued from previous page)

```
show_bboxes(fig.axes, boxes[250, 250, :, :] * bbox_scale,
            ['s=0.75, r=1', 's=0.5, r=1', 's=0.25, r=1', 's=0.75, r=2',
             's=0.75, r=0.5'])
```



13.4.2 Intersection over Union

We just mentioned that the anchor box covers the dog in the image well. If the ground-truth bounding box of the target is known, how can "well" here be quantified? An intuitive method is to measure the similarity between anchor boxes and the ground-truth bounding box. We know that the Jaccard index can measure the similarity between two sets. Given sets $\mathcal{A}$ and $\mathcal{B}$, their Jaccard index is the size of their intersection divided by the size of their union:

$$J(\mathcal{A},\mathcal{B}) = \frac{\left|\mathcal{A} \cap \mathcal{B}\right|}{\left| \mathcal{A} \cup \mathcal{B}\right|}. \tag{13.4.2}$$

In fact, we can consider the pixel area of a bounding box as a collection of pixels. In this way, we can measure the similarity of the two bounding boxes by the Jaccard index of their pixel sets. When we measure the similarity of two bounding boxes, we usually refer the Jaccard index as intersection over union (IoU), which is the ratio of the intersecting area to the union area of the two bounding boxes, as shown in Fig. 13.4.1. The value range of IoU is between 0 and 1: 0 means that there are no overlapping pixels between the two bounding boxes, while 1 indicates that the two bounding boxes are equal.

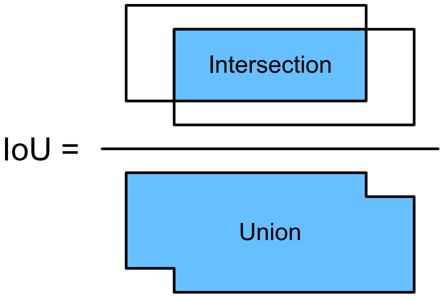


Fig. 13.4.1: IoU is the ratio of the intersecting area to the union area of two bounding boxes.

For the remainder of this section, we will use IoU to measure the similarity between anchor boxes and ground-truth bounding boxes, and between different anchor boxes.

13.4.3 Labeling Training Set Anchor Boxes

In the training set, we consider each anchor box as a training example. In order to train the object detection model, we need to mark two types of labels for each anchor box: first, the category of the target contained in the anchor box (category) and, second, the offset of the ground-truth bounding box relative to the anchor box (offset). In object detection, we first generate multiple anchor boxes, predict the categories and offsets for each anchor box, adjust the anchor box position according to the predicted offset to obtain the bounding boxes to be used for prediction, and finally filter out the prediction bounding boxes that need to be output.

We know that, in the object detection training set, each image is labelled with the location of the ground-truth bounding box and the category of the target contained. After the anchor boxes are generated, we primarily label anchor boxes based on the location and category information of the ground-truth bounding boxes similar to the anchor boxes. So how do we assign ground-truth bounding boxes to anchor boxes similar to them?

Assume that the anchor boxes in the image are $A_1, A_2, \ldots, A_{n_a}$ and the ground-truth bounding boxes are $B_1, B_2, \ldots, B_{n_b}$ and $n_a \geq n_b$. Define matrix $\mathbf{X} \in \mathbb{R}^{n_a \times n_b}$, where element x_{ij} in the i^th row and j^th column is the IoU of the anchor box A_i to the ground-truth bounding box B_j. First, we find the largest element in the matrix $\mathbf{X}$ and record the row index and column index of the element as i_1, j_1. We assign the ground-truth bounding box B_{j_1} to the anchor box A_{i_1}. Obviously, anchor box A_{i_1} and ground-truth bounding box B_{j_1} have the highest similarity among all the "anchor box–ground-truth bounding box" pairings. Next, discard all elements in the i_1th row and the j_1th column in the matrix $\mathbf{X}$. Find the largest remaining element in the matrix $\mathbf{X}$ and record the row index and column index of the element as i_2, j_2. We assign ground-truth bounding box B_{j_2} to anchor box A_{i_2} and then discard all elements in the i_2th row and the j_2th column in the matrix $\mathbf{X}$. At this point, elements in two rows and two columns in the matrix $\mathbf{X}$ have been discarded.

We proceed until all elements in the n_b column in the matrix $\mathbf{X}$ are discarded. At this time, we have assigned a ground-truth bounding box to each of the n_b anchor boxes. Next, we only traverse the remaining $n_a - n_b$ anchor boxes. Given anchor box A_i, find the bounding box B_j with the largest IoU with A_i according to the i^th row of the matrix $\mathbf{X}$, and only assign ground-truth bounding box B_j to anchor box A_i when the IoU is greater than the predetermined threshold.

As shown in Fig. 13.4.2 (left), assuming that the maximum value in the matrix $\mathbf{X}$ is x_{23}, we will assign ground-truth bounding box B_3 to anchor box A_2. Then, we discard all the elements in row 2 and column 3 of the matrix, find the largest element x_{71} of the remaining shaded area, and assign ground-truth bounding box B_1 to anchor box A_7. Then, as shown in Fig. 13.4.2 (middle), discard all the elements in row 7 and column 1 of the matrix, find the largest element x_{54} of the remaining shaded area, and assign ground-truth bounding box B_4 to anchor box A_5. Finally, as shown in Fig. 13.4.2 (right), discard all the elements in row 5 and column 4 of the matrix, find the largest element x_{92} of the remaining shaded area, and assign ground-truth bounding box B_2 to anchor box A_9. After that, we only need to traverse the remaining anchor boxes of A_2, A_5, A_7, A_9 and determine whether to assign ground-truth bounding boxes to the remaining anchor boxes according to the threshold.

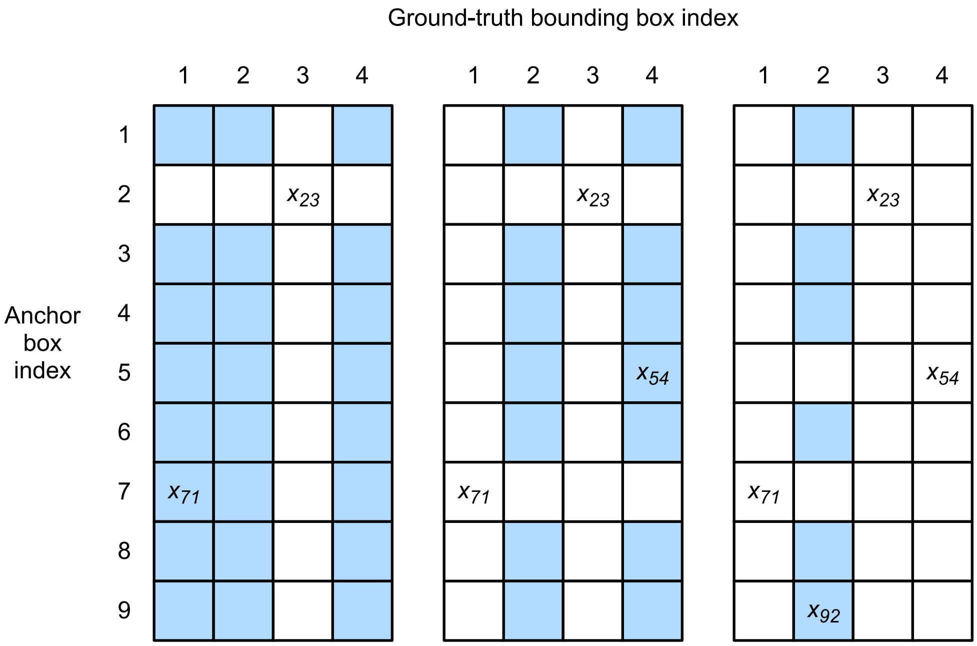


Fig. 13.4.2: Assign ground-truth bounding boxes to anchor boxes.

Now we can label the categories and offsets of the anchor boxes. If an anchor box A is assigned ground-truth bounding box B, the category of the anchor box A is set to the category of B. And the offset of the anchor box A is set according to the relative position of the central coordinates of B and A and the relative sizes of the two boxes. Because the positions and sizes of various boxes in the dataset may vary, these relative positions and relative sizes usually require some special transformations to make the offset distribution more uniform and easier to fit. Assume the center coordinates of anchor box A and its assigned ground-truth bounding box B are $(x_a, y_a), (x_b, y_b)$, the widths of A and B are w_a, w_b, and their heights are h_a, h_b, respectively. In this case, a common technique is to label the offset of A as

$$\left(\frac{\frac{x_b - x_a}{w_a} - \mu_x}{\sigma_x}, \frac{\frac{y_b - y_a}{h_a} - \mu_y}{\sigma_y}, \frac{\log \frac{w_b}{w_a} - \mu_w}{\sigma_w}, \frac{\log \frac{h_b}{h_a} - \mu_h}{\sigma_h} \right), \tag{13.4.3}$$

The default values of the constant are $\mu_x = \mu_y = \mu_w = \mu_h = 0, \sigma_x = \sigma_y = 0.1, and \sigma_w = \sigma_h = 0.2$. If an anchor box is not assigned a ground-truth bounding box, we only need to set the category of the anchor box to background. Anchor boxes whose category is background are often referred to as negative anchor boxes, and the rest are referred to as positive anchor boxes.

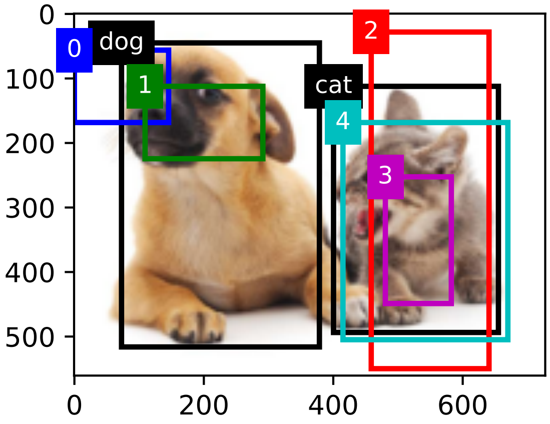
Below we demonstrate a detailed example. We define ground-truth bounding boxes for the cat and dog in the read image, where the first element is category (0 for dog, 1 for cat) and the remaining four elements are the x, y axis coordinates at top-left corner and x, y axis coordinates at lower-right corner (the value range is between 0 and 1). Here, we construct five anchor boxes to be labeled by the coordinates of the upper-left corner and the lower-right corner, which are recorded as $A_0, \ldots, A_4$, respectively (the index in the program starts from 0). First, draw the positions of these anchor boxes and the ground-truth bounding boxes in the image.

```
ground_truth = np.array([[0, 0.1, 0.08, 0.52, 0.92],
                         [1, 0.55, 0.2, 0.9, 0.88]])
anchors = np.array([[0, 0.1, 0.2, 0.3], [0.15, 0.2, 0.4, 0.4],
                    [0.63, 0.05, 0.88, 0.98], [0.66, 0.45, 0.8, 0.8],
```

(continues on next page)

(continued from previous page)

```
                    [0.57, 0.3, 0.92, 0.9]])

fig = d2l.plt.imshow(img)
show_bboxes(fig.axes, ground_truth[:, 1:] * bbox_scale, ['dog', 'cat'], 'k')
show_bboxes(fig.axes, anchors * bbox_scale, ['0', '1', '2', '3', '4']);
```



We can label categories and offsets for anchor boxes by using the `MultiBoxTarget` function in the `contrib.nd` module. This function sets the background category to 0 and increments the integer index of the target category from zero by 1 (1 for dog and 2 for cat). We add example dimensions to the anchor boxes and ground-truth bounding boxes and construct random predicted results with a shape of (batch size, number of categories including background, number of anchor boxes) by using the `expand_dims` function.

```
labels = npx.multibox_target(np.expand_dims(anchors, axis=0),
                             np.expand_dims(ground_truth, axis=0),
                             np.zeros((1, 3, 5)))
```

There are three items in the returned result, all of which are in the `ndarray` format. The third item is represented by the category labeled for the anchor box.

```
labels[2]
```

```
array([[0., 1., 2., 0., 2.]])
```

We analyze these labelled categories based on positions of anchor boxes and ground-truth bounding boxes in the image. First, in all "anchor box–ground-truth bounding box" pairs, the IoU of anchor box A_4 to the ground-truth bounding box of the cat is the largest, so the category of anchor box A_4 is labeled as cat. Without considering anchor box A_4 or the ground-truth bounding box of the cat, in the remaining "anchor box–ground-truth bounding box" pairs, the pair with the largest IoU is anchor box A_1 and the ground-truth bounding box of the dog, so the category of anchor box A_1 is labeled as dog. Next, traverse the remaining three unlabeled anchor boxes. The category of the ground-truth bounding box with the largest IoU with anchor box A_0 is dog, but the IoU is smaller than the threshold (the default is 0.5), so the category is labeled as background; the category of the ground-truth bounding box with the largest IoU with anchor box A_2 is cat and the IoU is greater than the threshold, so the category is labeled as cat; the category of the ground-truth bounding box with the largest IoU with anchor box A_3 is cat, but the IoU is smaller than the threshold, so the category is labeled as background.

The second item of the return value is a mask variable, with the shape of (batch size, four times the number of anchor boxes). The elements in the mask variable correspond one-to-one with the four offset values of each anchor box. Because we do not care about background detection, offsets of the negative class should not affect the target function. By multiplying by element, the 0 in the mask variable can filter out negative class offsets before calculating target function.

```
labels[1]
```

```
array([[0., 0., 0., 0., 1., 1., 1., 1., 1., 1., 1., 1., 0., 0., 0., 0.,
        1., 1., 1., 1.]])
```

The first item returned is the four offset values labeled for each anchor box, with the offsets of negative class anchor boxes labeled as 0.

```
labels[0]
```

```
array([[ 0.00e+00,  0.00e+00,  0.00e+00,  0.00e+00,  1.40e+00,  1.00e+01,
         2.59e+00,  7.18e+00, -1.20e+00,  2.69e-01,  1.68e+00, -1.57e+00,
         0.00e+00,  0.00e+00,  0.00e+00,  0.00e+00, -5.71e-01, -1.00e+00,
        -8.94e-07,  6.26e-01]])
```

13.4.4 Bounding Boxes for Prediction

During model prediction phase, we first generate multiple anchor boxes for the image and then predict categories and offsets for these anchor boxes one by one. Then, we obtain prediction bounding boxes based on anchor boxes and their predicted offsets. When there are many anchor boxes, many similar prediction bounding boxes may be output for the same target. To simplify the results, we can remove similar prediction bounding boxes. A commonly used method is called non-maximum suppression (NMS).

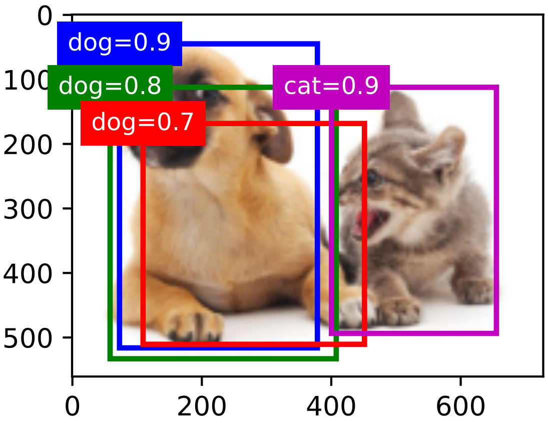
Let's take a look at how NMS works. For a prediction bounding box B, the model calculates the predicted probability for each category. Assume the largest predicted probability is p, the category corresponding to this probability is the predicted category of B. We also refer to p as the confidence level of prediction bounding box B. On the same image, we sort the prediction bounding boxes with predicted categories other than background by confidence level from high to low, and obtain the list L. Select the prediction bounding box B_1 with highest confidence level from L as a baseline and remove all non-benchmark prediction bounding boxes with an IoU with B_1 greater than a certain threshold from L. The threshold here is a preset hyperparameter. At this point, L retains the prediction bounding box with the highest confidence level and removes other prediction bounding boxes similar to it. Next, select the prediction bounding box B_2 with the second highest confidence level from L as a baseline, and remove all non-benchmark prediction bounding boxes with an IoU with B_2 greater than a certain threshold from L. Repeat this process until all prediction bounding boxes in L have been used as a baseline. At this time, the IoU of any pair of prediction bounding boxes in L is less than the threshold. Finally, output all prediction bounding boxes in the list L.

Next, we will look at a detailed example. First, construct four anchor boxes. For the sake of simplicity, we assume that predicted offsets are all 0. This means that the prediction bounding boxes are anchor boxes. Finally, we construct a predicted probability for each category.

```
anchors = np.array([[0.1, 0.08, 0.52, 0.92], [0.08, 0.2, 0.56, 0.95],
                    [0.15, 0.3, 0.62, 0.91], [0.55, 0.2, 0.9, 0.88]])
offset_preds = np.array([0] * anchors.size)
cls_probs = np.array([[0] * 4,  # Predicted probability for background
                      [0.9, 0.8, 0.7, 0.1],  # Predicted probability for dog
                      [0.1, 0.2, 0.3, 0.9]])  # Predicted probability for cat
```

Print prediction bounding boxes and their confidence levels on the image.

```
fig = d2l.plt.imshow(img)
show_bboxes(fig.axes, anchors * bbox_scale,
            ['dog=0.9', 'dog=0.8', 'dog=0.7', 'cat=0.9'])
```



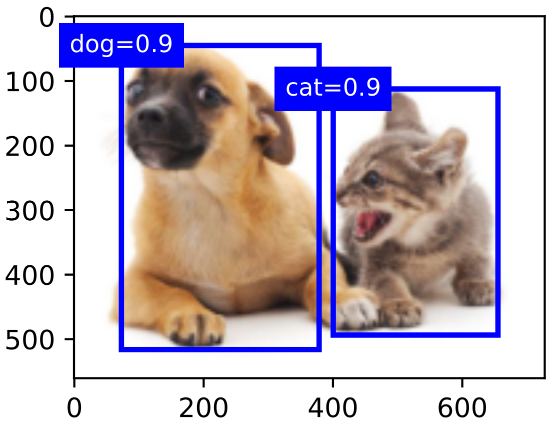
We use the MultiBoxDetection function of the contrib.nd module to perform NMS and set the threshold to 0.5. This adds an example dimension to the ndarray input. We can see that the shape of the returned result is (batch size, number of anchor boxes, 6). The 6 elements of each row represent the output information for the same prediction bounding box. The first element is the predicted category index, which starts from 0 (0 is dog, 1 is cat). The value -1 indicates background or removal in NMS. The second element is the confidence level of prediction bounding box. The remaining four elements are the x, y axis coordinates of the upper-left corner and the x, y axis coordinates of the lower-right corner of the prediction bounding box (the value range is between 0 and 1).

```
output = npx.multibox_detection(
    np.expand_dims(cls_probs, axis=0),
    np.expand_dims(offset_preds, axis=0),
    np.expand_dims(anchors, axis=0),
    nms_threshold=0.5)
output
```

```
array([[[ 0.  ,  0.9 ,  0.1 ,  0.08,  0.52,  0.92],
        [ 1.  ,  0.9 ,  0.55,  0.2 ,  0.9 ,  0.88],
        [-1.  ,  0.8 ,  0.08,  0.2 ,  0.56,  0.95],
        [-1.  ,  0.7 ,  0.15,  0.3 ,  0.62,  0.91]]])
```

We remove the prediction bounding boxes of category -1 and visualize the results retained by NMS.

```
fig = d2l.plt.imshow(img)
for i in output[0].asnumpy():
    if i[0] == -1:
        continue
    label = ('dog=', 'cat=')[int(i[0])] + str(i[1])
    show_bboxes(fig.axes, [np.array(i[2:]) * bbox_scale], label)
```



In practice, we can remove prediction bounding boxes with lower confidence levels before performing NMS, thereby reducing the amount of computation for NMS. We can also filter the output of NMS, for example, by only retaining results with higher confidence levels as the final output.

Summary

- We generate multiple anchor boxes with different sizes and aspect ratios, centered on each pixel.
- IoU, also called Jaccard index, measures the similarity of two bounding boxes. It is the ratio of the intersecting area to the union area of two bounding boxes.
- In the training set, we mark two types of labels for each anchor box: one is the category of the target contained in the anchor box and the other is the offset of the ground-truth bounding box relative to the anchor box.
- When predicting, we can use non-maximum suppression (NMS) to remove similar prediction bounding boxes, thereby simplifying the results.

Exercises

1. Change the `sizes` and `ratios` values in `contrib.nd.MultiBoxPrior` and observe the changes to the generated anchor boxes.
2. Construct two bounding boxes with and IoU of 0.5, and observe their coincidence.
3. Verify the output of offset `labels[0]` by marking the anchor box offsets as defined in this section (the constant is the default value).
4. Modify the variable `anchors` in the "Labeling Training Set Anchor Boxes" and "Output Bounding Boxes for Prediction" sections. How do the results change?

13.5 Multiscale Object Detection

In Section 13.4, we generated multiple anchor boxes centered on each pixel of the input image. These anchor boxes are used to sample different regions of the input image. However, if anchor boxes are generated centered on each pixel of the image, soon there will be too many anchor boxes for us to compute. For example, we assume that the input image has a height and a width of 561 and 728 pixels respectively. If five different shapes of anchor boxes are generated centered on each pixel, over two million anchor boxes ($561 \times 728 \times 5$) need to be predicted and labeled on the image.

It is not difficult to reduce the number of anchor boxes. An easy way is to apply uniform sampling on a small portion of pixels from the input image and generate anchor boxes centered on the sampled pixels. In addition, we can generate anchor boxes of varied numbers and sizes on multiple scales. Notice that smaller objects are more likely to be positioned on the image than larger ones. Here, we will use a simple example: Objects with shapes of 1×1, 1×2, and 2×2 may have 4, 2, and 1 possible position(s) on an image with the shape 2×2. Therefore, when using smaller anchor boxes to detect smaller objects, we can sample more regions; when using larger anchor boxes to detect larger objects, we can sample fewer regions.

To demonstrate how to generate anchor boxes on multiple scales, let's read an image first. It has a height and width of 561×728 pixels.

```
%matplotlib inline
import d2l
from mxnet import contrib, image, np, npx

npx.set_np()

img = image.imread('../img/catdog.jpg')
h, w = img.shape[0:2]
h, w
```

```
(561, 728)
```

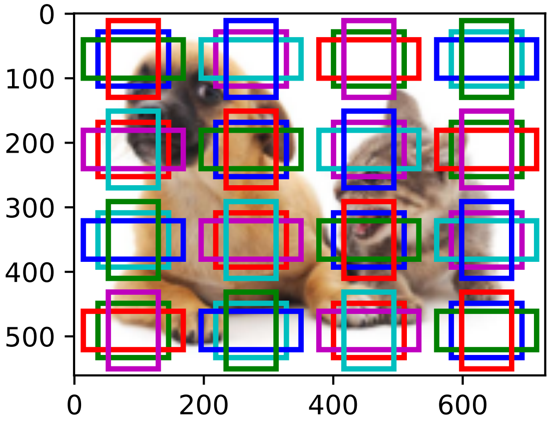
In Section 6.2, the 2D array output of the convolutional neural network (CNN) is called a feature map. We can determine the midpoints of anchor boxes uniformly sampled on any image by defining the shape of the feature map.

The function `display_anchors` is defined below. We are going to generate anchor boxes `anchors` centered on each unit (pixel) on the feature map `fmap`. Since the coordinates of axes x and y in anchor boxes `anchors` have been divided by the width and height of the feature map `fmap`, values between 0 and 1 can be used to represent relative positions of anchor boxes in the feature map. Since the midpoints of anchor boxes `anchors` overlap with all the units on feature map `fmap`, the relative spatial positions of the midpoints of the `anchors` on any image must have a uniform distribution. Specifically, when the width and height of the feature map are set to `fmap_w` and `fmap_h` respectively, the function will conduct uniform sampling for `fmap_h` rows and `fmap_w` columns of pixels and use them as midpoints to generate anchor boxes with size s (we assume that the length of list s is 1) and different aspect ratios (ratios).

```
def display_anchors(fmap_w, fmap_h, s):
    d2l.set_figsize((3.5, 2.5))
    # The values from the first two dimensions will not affect the output
    fmap = np.zeros((1, 10, fmap_w, fmap_h))
    anchors = npx.multibox_prior(fmap, sizes=s, ratios=[1, 2, 0.5])
    bbox_scale = np.array((w, h, w, h))
    d2l.show_bboxes(d2l.plt.imshow(img.asnumpy()).axes,
                    anchors[0] * bbox_scale)
```

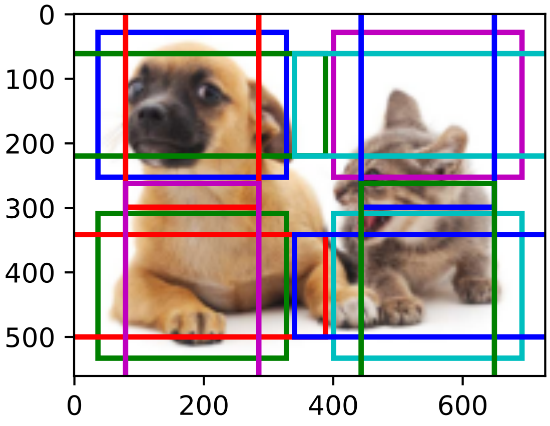
We will first focus on the detection of small objects. In order to make it easier to distinguish upon display, the anchor boxes with different midpoints here do not overlap. We assume that the size of the anchor boxes is 0.15 and the height and width of the feature map are 4. We can see that the midpoints of anchor boxes from the 4 rows and 4 columns on the image are uniformly distributed.

```
display_anchors(fmap_w=4, fmap_h=4, s=[0.15])
```



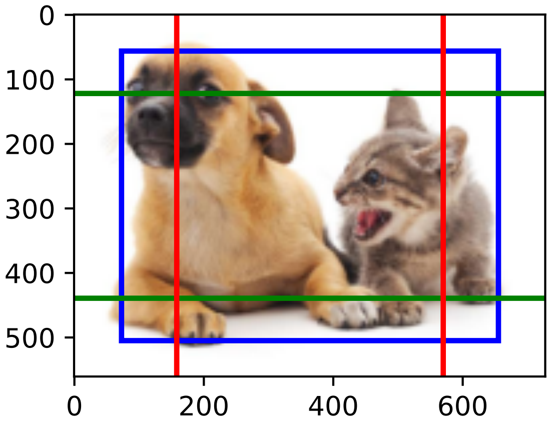
We are going to reduce the height and width of the feature map by half and use a larger anchor box to detect larger objects. When the size is set to 0.4, overlaps will occur between regions of some anchor boxes.

```
display_anchors(fmap_w=2, fmap_h=2, s=[0.4])
```



Finally, we are going to reduce the height and width of the feature map by half and increase the anchor box size to 0.8. Now the midpoint of the anchor box is the center of the image.

```
display_anchors(fmap_w=1, fmap_h=1, s=[0.8])
```



Since we have generated anchor boxes of different sizes on multiple scales, we will use them to detect objects of various sizes at different scales. Now we are going to introduce a method based on convolutional neural networks (CNNs).

At a certain scale, suppose we generate $h \times w$ sets of anchor boxes with different midpoints based on c_i feature maps with the shape $h \times w$ and the number of anchor boxes in each set is a. For example, for the first scale of the experiment, we generate 16 sets of anchor boxes with different midpoints based on 10 (number of channels) feature maps with a shape of 4×4, and each set contains 3 anchor boxes. Next, each anchor box is labeled with a category and offset based on the classification and position of the ground-truth bounding box. At the current scale, the object detection model needs to predict the category and offset of $h \times w$ sets of anchor boxes with different midpoints based on the input image.

We assume that the c_i feature maps are the intermediate output of the CNN based on the input image. Since each feature map has $h \times w$ different spatial positions, the same position will have c_i units. According to the definition of receptive field in the Section 6.2, the c_i units of the feature map at the same spatial position have the same receptive field on the input image. Thus, they represent the information of the input image in this same receptive field. Therefore, we can transform the c_i units of the feature map at the same spatial position into the categories and offsets of the a anchor boxes generated using that position as a midpoint. It is not hard to see that, in essence, we use the information of the input image in a certain receptive field to predict the category and offset of the anchor boxes close to the field on the input image.

When the feature maps of different layers have receptive fields of different sizes on the input image, they are used to detect objects of different sizes. For example, we can design a network to have a wider receptive field for each unit in the feature map that is closer to the output layer, to detect objects with larger sizes in the input image.

We will implement a multiscale object detection model in the following section.

Summary

- We can generate anchor boxes with different numbers and sizes on multiple scales to detect objects of different sizes on multiple scales.
- The shape of the feature map can be used to determine the midpoint of the anchor boxes that uniformly sample any image.
- We use the information for the input image from a certain receptive field to predict the category and offset of the anchor boxes close to that field on the image.

Exercises

1. Given an input image, assume $1 \times c_i \times h \times w$ to be the shape of the feature map while c_i, h, w are the number, height, and width of the feature map. What methods can you think of to convert this variable into the anchor box's category and offset? What is the shape of the output?

13.6 The Object Detection Dataset (Pikachu)

There are no small datasets, like MNIST or Fashion-MNIST, in the object detection field. In order to quickly test models, we are going to assemble a small dataset. First, we generate 1000 Pikachu images of different angles and sizes using an open source 3D Pikachu model. Then, we collect a series of background images and place a Pikachu image at a random position on each image. We use the im2rec tool[202] provided by MXNet to convert the images to binary RecordIO format[1]. This format can reduce the storage overhead of the dataset on the disk and improve the reading efficiency. If you want to learn more about how to read images, refer to the documentation for the GluonCV Toolkit[203].

13.6.1 Downloading the Dataset

The Pikachu dataset in RecordIO format can be downloaded directly from the Internet. The operation for downloading the dataset is defined in the function `_download_pikachu`.

```
%matplotlib inline
import d2l
from mxnet import gluon, image, np, npx
import os

npx.set_np()
```

(continues on next page)

[202] https://github.com/apache/incubator-mxnet/blob/master/tools/im2rec.py
[203] https://gluon-cv.mxnet.io/

(continued from previous page)

```
# Saved in the d2l package for later use
def download_pikachu(data_dir):
    root_url = ('https://apache-mxnet.s3-accelerate.amazonaws.com/'
                'gluon/dataset/pikachu/')
    dataset = {'train.rec': 'e6bcb6ffba1ac04ff8a9b1115e650af56ee969c8',
               'train.idx': 'dcf7318b2602c06428b9988470c731621716c393',
               'val.rec': 'd6c33f799b4d058e82f2cb5bd9a976f69d72d520'}
    for k, v in dataset.items():
        gluon.utils.download(
            root_url + k, os.path.join(data_dir, k), sha1_hash=v)
```

13.6.2 Reading the Dataset

We are going to read the object detection dataset by creating the instance `ImageDetIter`. The "Det" in the name refers to Detection. We will read the training dataset in random order. Since the format of the dataset is RecordIO, we need the image index file `'train.idx'` to read random minibatches. In addition, for each image of the training set, we will use random cropping and require the cropped image to cover at least 95% of each object. Since the cropping is random, this requirement is not always satisfied. We preset the maximum number of random cropping attempts to 200. If none of them meets the requirement, the image will not be cropped. To ensure the certainty of the output, we will not randomly crop the images in the test dataset. We also do not need to read the test dataset in random order.

```
# Saved in the d2l package for later use
def load_data_pikachu(batch_size, edge_size=256):
    """Load the pikachu dataset"""
    data_dir = '../data/pikachu'
    download_pikachu(data_dir)
    train_iter = image.ImageDetIter(
        path_imgrec=os.path.join(data_dir, 'train.rec'),
        path_imgidx=os.path.join(data_dir, 'train.idx'),
        batch_size=batch_size,
        data_shape=(3, edge_size, edge_size),  # The shape of the output image
        shuffle=True,  # Read the dataset in random order
        rand_crop=1,  # The probability of random cropping is 1
        min_object_covered=0.95, max_attempts=200)
    val_iter = image.ImageDetIter(
        path_imgrec=os.path.join(data_dir, 'val.rec'), batch_size=batch_size,
        data_shape=(3, edge_size, edge_size), shuffle=False)
    return train_iter, val_iter
```

Below, we read a minibatch and print the shape of the image and label. The shape of the image is the same as in the previous experiment (batch size, number of channels, height, width). The shape of the label is (batch size, m, 5), where m is equal to the maximum number of bounding boxes contained in a single image in the dataset. Although computation for the minibatch is very efficient, it requires each image to contain the same number of bounding boxes so that they can be placed in the same batch. Since each image may have a different number of bounding boxes, we can add illegal bounding boxes to images that have less than m bounding boxes until each image contains m bounding boxes. Thus, we can read a minibatch of images each time. The label of each bounding box in the image is represented by an array of length 5. The first element in the array is the category of the object contained in the bounding box. When the value is -1, the bounding box is an illegal bounding box for filling purpose. The remaining four elements of the array represent the x, y axis coordinates of the upper-left corner of the bounding box and the x, y axis coordinates of the lower-right corner of the bounding box (the value range is between 0 and 1). The Pikachu dataset here has only one bounding box per image, so $m = 1$.

```
batch_size, edge_size = 32, 256
train_iter, _ = load_data_pikachu(batch_size, edge_size)
batch = train_iter.next()
batch.data[0].shape, batch.label[0].shape
```

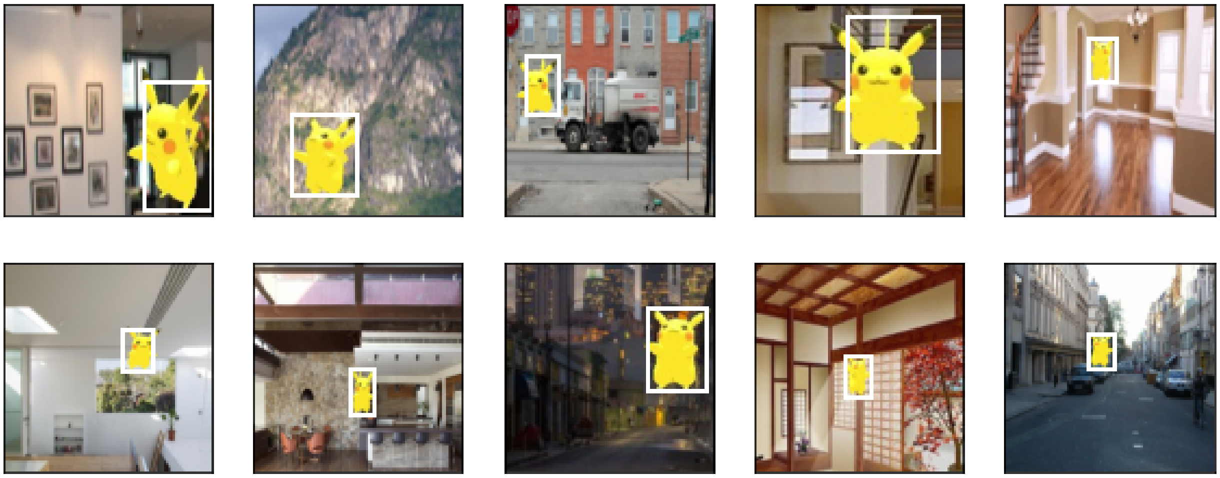
```
Downloading ../data/pikachu/train.rec from https://apache-mxnet.s3-accelerate.amazonaws.com/
↪gluon/dataset/pikachu/train.rec...
Downloading ../data/pikachu/train.idx from https://apache-mxnet.s3-accelerate.amazonaws.com/
↪gluon/dataset/pikachu/train.idx...
Downloading ../data/pikachu/val.rec from https://apache-mxnet.s3-accelerate.amazonaws.com/
↪gluon/dataset/pikachu/val.rec...
```

```
((32, 3, 256, 256), (32, 1, 5))
```

13.6.3 Demonstration

We have ten images with bounding boxes on them. We can see that the angle, size, and position of Pikachu are different in each image. Of course, this is a simple artificial dataset. In actual practice, the data are usually much more complicated.

```
imgs = (batch.data[0][0:10].transpose(0, 2, 3, 1)) / 255
axes = d2l.show_images(imgs, 2, 5, scale=2)
for ax, label in zip(axes, batch.label[0][0:10]):
    d2l.show_bboxes(ax, [label[0][1:5] * edge_size], colors=['w'])
```



Summary

- The Pikachu dataset we synthesized can be used to test object detection models.
- The data reading for object detection is similar to that for image classification. However, after we introduce bounding boxes, the label shape and image augmentation (e.g., random cropping) are changed.

Exercises

1. Referring to the MXNet documentation, what are the parameters for the constructors of the `image.ImageDetIter` and `image.CreateDetAugmenter` classes? What is their significance?

13.7 Single Shot Multibox Detection (SSD)

In the previous few sections, we have introduced bounding boxes, anchor boxes, multiscale object detection, and datasets. Now, we will use this background knowledge to construct an object detection model: single shot multibox detection (SSD) (Liu et al., 2016). This quick and easy model is already widely used. Some of the design concepts and implementation details of this model are also applicable to other object detection models.

13.7.1 Model

Fig. 13.7.1 shows the design of an SSD model. The model's main components are a base network block and several multiscale feature blocks connected in a series. Here, the base network block is used to extract features of original images, and it generally takes the form of a deep convolutional neural network. The paper on SSDs chooses to place a truncated VGG before the classification layer (Liu et al., 2016), but this is now commonly replaced by ResNet. We can design the base network so that it outputs larger heights and widths. In this way, more anchor boxes are generated based on this feature map, allowing us to detect smaller objects. Next, each multiscale feature block reduces the height and width of the feature map provided by the previous layer (for example, it may reduce the sizes by half). The blocks then use each element in the feature map to expand the receptive field on the input image. In this way, the closer a multiscale feature block is to the top of Fig. 13.7.1 the smaller its output feature map, and the fewer the anchor boxes that are generated based on the feature map. In addition, the closer a feature block is to the top, the larger the receptive field of each element in the feature map and the better suited it is to detect larger objects. As the SSD generates different numbers of anchor boxes of different sizes based on the base network block and each multiscale feature block and then predicts the categories and offsets (i.e., predicted bounding boxes) of the anchor boxes in order to detect objects of different sizes, SSD is a multiscale object detection model.

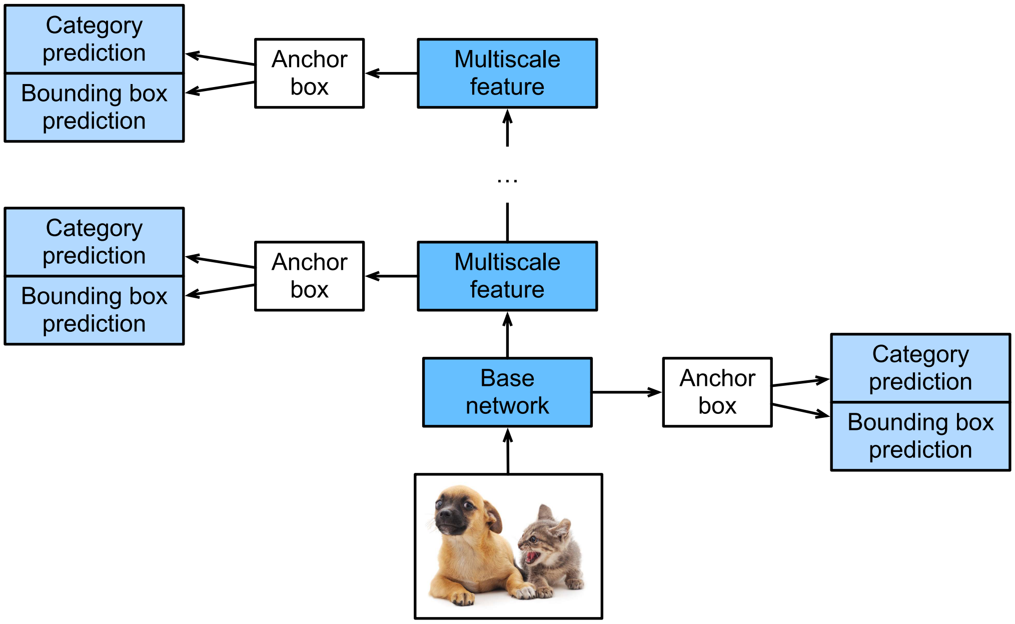


Fig. 13.7.1: The SSD is composed of a base network block and several multiscale feature blocks connected in a series.

Next, we will describe the implementation of the modules in Fig. 13.7.1. First, we need to discuss the implementation of category prediction and bounding box prediction.

Category Prediction Layer

Set the number of object categories to q. In this case, the number of anchor box categories is $q + 1$, with 0 indicating an anchor box that only contains background. For a certain scale, set the height and width of the feature map to h and w, respectively. If we use each element as the center to generate a anchor boxes, we need to classify a total of hwa anchor boxes. If we use a fully connected layer (FCN) for the output, this will likely result in an excessive number of model parameters. Recall how we used convolutional layer channels to output category predictions in Section 7.3. SSD uses the same method to reduce the model complexity.

Specifically, the category prediction layer uses a convolutional layer that maintains the input height and width. Thus, the output and input have a one-to-one correspondence to the spatial coordinates along the width and height of the feature map. Assuming that the output and input have the same spatial coordinates (x, y), the channel for the coordinates (x, y) on the output feature map contains the category predictions for all anchor boxes generated using the input feature map coordinates (x, y) as the center. Therefore, there are $a(q + 1)$ output channels, with the output channels indexed as $i(q+1)+j$ $(0 \leq j \leq q)$ representing the predictions of the category index j for the anchor box index i.

Now, we will define a category prediction layer of this type. After we specify the parameters a and q, it uses a 3×3 convolutional layer with a padding of 1. The heights and widths of the input and output of this convolutional layer remain unchanged.

```
%matplotlib inline
import d2l
from mxnet import autograd, contrib, gluon, image, init, np, npx
```

(continues on next page)

(continued from previous page)

```
from mxnet.gluon import nn

npx.set_np()

def cls_predictor(num_anchors, num_classes):
    return nn.Conv2D(num_anchors * (num_classes + 1), kernel_size=3,
                     padding=1)
```

Bounding Box Prediction Layer

The design of the bounding box prediction layer is similar to that of the category prediction layer. The only difference is that, here, we need to predict 4 offsets for each anchor box, rather than $q+1$ categories.

```
def bbox_predictor(num_anchors):
    return nn.Conv2D(num_anchors * 4, kernel_size=3, padding=1)
```

Concatenating Predictions for Multiple Scales

As we mentioned, SSD uses feature maps based on multiple scales to generate anchor boxes and predict their categories and offsets. Because the shapes and number of anchor boxes centered on the same element differ for the feature maps of different scales, the prediction outputs at different scales may have different shapes.

In the following example, we use the same batch of data to construct feature maps of two different scales, `Y1` and `Y2`. Here, `Y2` has half the height and half the width of `Y1`. Using category prediction as an example, we assume that each element in the `Y1` and `Y2` feature maps generates five (Y1) or three (Y2) anchor boxes. When there are 10 object categories, the number of category prediction output channels is either $5 \times (10 + 1) = 55$ or $3 \times (10 + 1) = 33$. The format of the prediction output is (batch size, number of channels, height, width). As you can see, except for the batch size, the sizes of the other dimensions are different. Therefore, we must transform them into a consistent format and concatenate the predictions of the multiple scales to facilitate subsequent computation.

```
def forward(x, block):
    block.initialize()
    return block(x)

Y1 = forward(np.zeros((2, 8, 20, 20)), cls_predictor(5, 10))
Y2 = forward(np.zeros((2, 16, 10, 10)), cls_predictor(3, 10))
(Y1.shape, Y2.shape)
```

```
((2, 55, 20, 20), (2, 33, 10, 10))
```

The channel dimension contains the predictions for all anchor boxes with the same center. We first move the channel dimension to the final dimension. Because the batch size is the same for all scales, we can convert the prediction results to binary format (batch size, height $\times$ width $\times$ number of channels) to facilitate subsequent concatenation on the 1^{st} dimension.

```
def flatten_pred(pred):
    return npx.batch_flatten(pred.transpose(0, 2, 3, 1))

def concat_preds(preds):
    return np.concatenate([flatten_pred(p) for p in preds], axis=1)
```

Thus, regardless of the different shapes of Y1 and Y2, we can still concatenate the prediction results for the two different scales of the same batch.

```
concat_preds([Y1, Y2]).shape
```

```
(2, 25300)
```

Height and Width Downsample Block

For multiscale object detection, we define the following down_sample_blk block, which reduces the height and width by 50%. This block consists of two 3×3 convolutional layers with a padding of 1 and a 2×2 maximum pooling layer with a stride of 2 connected in a series. As we know, 3×3 convolutional layers with a padding of 1 do not change the shape of feature maps. However, the subsequent pooling layer directly reduces the size of the feature map by half. Because $1 \times 2 + (3 - 1) + (3 - 1) = 6$, each element in the output feature map has a receptive field on the input feature map of the shape 6×6. As you can see, the height and width downsample block enlarges the receptive field of each element in the output feature map.

```
def down_sample_blk(num_channels):
    blk = nn.Sequential()
    for _ in range(2):
        blk.add(nn.Conv2D(num_channels, kernel_size=3, padding=1),
                nn.BatchNorm(in_channels=num_channels),
                nn.Activation('relu'))
    blk.add(nn.MaxPool2D(2))
    return blk
```

By testing forward computation in the height and width downsample block, we can see that it changes the number of input channels and halves the height and width.

```
forward(np.zeros((2, 3, 20, 20)), down_sample_blk(10)).shape
```

```
(2, 10, 10, 10)
```

Base Network Block

The base network block is used to extract features from original images. To simplify the computation, we will construct a small base network. This network consists of three height and width downsample blocks connected in a series, so it doubles the number of channels at each step. When we input an original image with the shape 256×256, the base network block outputs a feature map with the shape 32×32.

```
def base_net():
    blk = nn.Sequential()
    for num_filters in [16, 32, 64]:
        blk.add(down_sample_blk(num_filters))
    return blk

forward(np.zeros((2, 3, 256, 256)), base_net()).shape
```

```
(2, 64, 32, 32)
```

The Complete Model

The SSD model contains a total of five modules. Each module outputs a feature map used to generate anchor boxes and predict the categories and offsets of these anchor boxes. The first module is the base network block, modules two to four are height and width downsample blocks, and the fifth module is a global maximum pooling layer that reduces the height and width to 1. Therefore, modules two to five are all multiscale feature blocks shown in Fig. 13.7.1.

```
def get_blk(i):
    if i == 0:
        blk = base_net()
    elif i == 4:
        blk = nn.GlobalMaxPool2D()
    else:
        blk = down_sample_blk(128)
    return blk
```

Now, we will define the forward computation process for each module. In contrast to the previously-described convolutional neural networks, this module not only returns feature map Y output by convolutional computation, but also the anchor boxes of the current scale generated from Y and their predicted categories and offsets.

```
def blk_forward(X, blk, size, ratio, cls_predictor, bbox_predictor):
    Y = blk(X)
    anchors = npx.multibox_prior(Y, sizes=size, ratios=ratio)
    cls_preds = cls_predictor(Y)
    bbox_preds = bbox_predictor(Y)
    return (Y, anchors, cls_preds, bbox_preds)
```

As we mentioned, the closer a multiscale feature block is to the top in Fig. 13.7.1, the larger the objects it detects and the larger the anchor boxes it must generate. Here, we first divide the interval from 0.2 to 1.05 into five equal parts to determine the sizes of smaller anchor boxes at different scales: 0.2, 0.37, 0.54, etc. Then, according to $\sqrt{0.2 \times 0.37} = 0.272$, $\sqrt{0.37 \times 0.54} = 0.447$, and similar formulas, we determine the sizes of larger anchor boxes at the different scales.

```
sizes = [[0.2, 0.272], [0.37, 0.447], [0.54, 0.619], [0.71, 0.79],
         [0.88, 0.961]]
ratios = [[1, 2, 0.5]] * 5
num_anchors = len(sizes[0]) + len(ratios[0]) - 1
```

Now, we can define the complete model, TinySSD.

```
class TinySSD(nn.Block):
    def __init__(self, num_classes, **kwargs):
        super(TinySSD, self).__init__(**kwargs)
        self.num_classes = num_classes
        for i in range(5):
            # The assignment statement is self.blk_i = get_blk(i)
            setattr(self, 'blk_%d' % i, get_blk(i))
            setattr(self, 'cls_%d' % i, cls_predictor(num_anchors,
                                                      num_classes))
            setattr(self, 'bbox_%d' % i, bbox_predictor(num_anchors))

    def forward(self, X):
        anchors, cls_preds, bbox_preds = [None] * 5, [None] * 5, [None] * 5
        for i in range(5):
            # getattr(self, 'blk_%d' % i) accesses self.blk_i
            X, anchors[i], cls_preds[i], bbox_preds[i] = blk_forward(
                X, getattr(self, 'blk_%d' % i), sizes[i], ratios[i],
                getattr(self, 'cls_%d' % i), getattr(self, 'bbox_%d' % i))
        # In the reshape function, 0 indicates that the batch size remains
        # unchanged
        anchors = np.concatenate(anchors, axis=1)
        cls_preds = concat_preds(cls_preds)
        cls_preds = cls_preds.reshape(
            cls_preds.shape[0], -1, self.num_classes + 1)
        bbox_preds = concat_preds(bbox_preds)
        return anchors, cls_preds, bbox_preds
```

We now create an SSD model instance and use it to perform forward computation on image minibatch X, which has a height and width of 256 pixels. As we verified previously, the first module outputs a feature map with the shape 32×32. Because modules two to four are height and width downsample blocks, module five is a global pooling layer, and each element in the feature map is used as the center for 4 anchor boxes, a total of $(32^2 + 16^2 + 8^2 + 4^2 + 1) \times 4 = 5444$ anchor boxes are generated for each image at the five scales.

```
net = TinySSD(num_classes=1)
net.initialize()
X = np.zeros((32, 3, 256, 256))
anchors, cls_preds, bbox_preds = net(X)

print('output anchors:', anchors.shape)
print('output class preds:', cls_preds.shape)
print('output bbox preds:', bbox_preds.shape)
```

```
output anchors: (1, 5444, 4)
output class preds: (32, 5444, 2)
output bbox preds: (32, 21776)
```

13.7.2 Training

Now, we will explain, step by step, how to train the SSD model for object detection.

Data Reading and Initialization

We read the Pikachu dataset we created in the previous section.

```
batch_size = 32
train_iter, _ = d2l.load_data_pikachu(batch_size)
```

There is 1 category in the Pikachu dataset. After defining the module, we need to initialize the model parameters and define the optimization algorithm.

```
ctx, net = d2l.try_gpu(), TinySSD(num_classes=1)
net.initialize(init=init.Xavier(), ctx=ctx)
trainer = gluon.Trainer(net.collect_params(), 'sgd',
                        {'learning_rate': 0.2, 'wd': 5e-4})
```

Defining Loss and Evaluation Functions

Object detection is subject to two types of losses. The first is anchor box category loss. For this, we can simply reuse the cross-entropy loss function we used in image classification. The second loss is positive anchor box offset loss. Offset prediction is a normalization problem. However, here, we do not use the squared loss introduced previously. Rather, we use the L_1 norm loss, which is the absolute value of the difference between the predicted value and the ground-truth value. The mask variable bbox_masks removes negative anchor boxes and padding anchor boxes from the loss calculation. Finally, we add the anchor box category and offset losses to find the final loss function for the model.

```
cls_loss = gluon.loss.SoftmaxCrossEntropyLoss()
bbox_loss = gluon.loss.L1Loss()

def calc_loss(cls_preds, cls_labels, bbox_preds, bbox_labels, bbox_masks):
    cls = cls_loss(cls_preds, cls_labels)
    bbox = bbox_loss(bbox_preds * bbox_masks, bbox_labels * bbox_masks)
    return cls + bbox
```

We can use the accuracy rate to evaluate the classification results. As we use the L_1 norm loss, we will use the average absolute error to evaluate the bounding box prediction results.

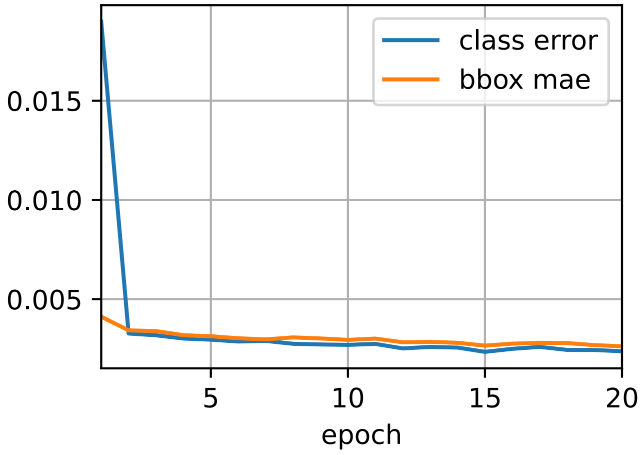
```
def cls_eval(cls_preds, cls_labels):
    # Because the category prediction results are placed in the final
    # dimension, argmax must specify this dimension
    return float((cls_preds.argmax(axis=-1) == cls_labels).sum())

def bbox_eval(bbox_preds, bbox_labels, bbox_masks):
    return float((np.abs((bbox_labels - bbox_preds) * bbox_masks)).sum())
```

Training the Model

During model training, we must generate multiscale anchor boxes (anchors) in the model's forward computation process and predict the category (cls_preds) and offset (bbox_preds) for each anchor box. Afterwards, we label the category (cls_labels) and offset (bbox_labels) of each generated anchor box based on the label information Y. Finally, we calculate the loss function using the predicted and labeled category and offset values. To simplify the code, we do not evaluate the training dataset here.

```
num_epochs, timer = 20, d2l.Timer()
animator = d2l.Animator(xlabel='epoch', xlim=[1, num_epochs],
                        legend=['class error', 'bbox mae'])
for epoch in range(num_epochs):
    # accuracy_sum, mae_sum, num_examples, num_labels
    metric = d2l.Accumulator(4)
    train_iter.reset()  # Read data from the start.
    for batch in train_iter:
        timer.start()
        X = batch.data[0].as_in_context(ctx)
        Y = batch.label[0].as_in_context(ctx)
        with autograd.record():
            # Generate multiscale anchor boxes and predict the category and
            # offset of each
            anchors, cls_preds, bbox_preds = net(X)
            # Label the category and offset of each anchor box
            bbox_labels, bbox_masks, cls_labels = npx.multibox_target(
                anchors, Y, cls_preds.transpose(0, 2, 1))
            # Calculate the loss function using the predicted and labeled
            # category and offset values
            l = calc_loss(cls_preds, cls_labels, bbox_preds, bbox_labels,
                          bbox_masks)
        l.backward()
        trainer.step(batch_size)
        metric.add(cls_eval(cls_preds, cls_labels), cls_labels.size,
                   bbox_eval(bbox_preds, bbox_labels, bbox_masks),
                   bbox_labels.size)
    cls_err, bbox_mae = 1-metric[0]/metric[1], metric[2]/metric[3]
    animator.add(epoch+1, (cls_err, bbox_mae))
print('class err %.2e, bbox mae %.2e' % (cls_err, bbox_mae))
print('%.1f exampes/sec on %s' % (train_iter.num_image/timer.stop(), ctx))
```

```
class err 2.38e-03, bbox mae 2.63e-03
4249.8 exampes/sec on gpu(0)
```



13.7.3 Prediction

In the prediction stage, we want to detect all objects of interest in the image. Below, we read the test image and transform its size. Then, we convert it to the four-dimensional format required by the convolutional layer.

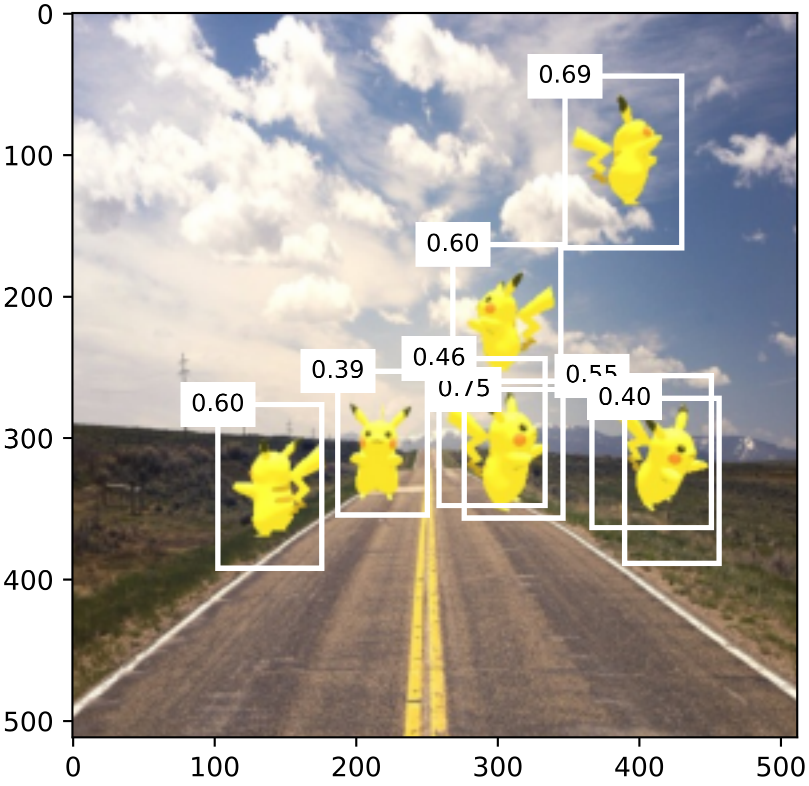
```
img = image.imread('../img/pikachu.jpg')
feature = image.imresize(img, 256, 256).astype('float32')
X = np.expand_dims(feature.transpose(2, 0, 1), axis=0)
```

Using the `MultiBoxDetection` function, we predict the bounding boxes based on the anchor boxes and their predicted offsets. Then, we use non-maximum suppression to remove similar bounding boxes.

```
def predict(X):
    anchors, cls_preds, bbox_preds = net(X.as_in_context(ctx))
    cls_probs = npx.softmax(cls_preds).transpose(0, 2, 1)
    output = npx.multibox_detection(cls_probs, bbox_preds, anchors)
    idx = [i for i, row in enumerate(output[0]) if row[0] != -1]
    return output[0, idx]

output = predict(X)
```

Finally, we take all the bounding boxes with a confidence level of at least 0.3 and display them as the final output.

```
def display(img, output, threshold):
    d2l.set_figsize((5, 5))
    fig = d2l.plt.imshow(img.asnumpy())
    for row in output:
        score = float(row[1])
        if score < threshold:
            continue
        h, w = img.shape[0:2]
        bbox = [row[2:6] * np.array((w, h, w, h), ctx=row.context)]
        d2l.show_bboxes(fig.axes, bbox, '%.2f' % score, 'w')

display(img, output, threshold=0.3)
```



Summary

- SSD is a multiscale object detection model. This model generates different numbers of anchor boxes of different sizes based on the base network block and each multiscale feature block and predicts the categories and offsets of the anchor boxes to detect objects of different sizes.
- During SSD model training, the loss function is calculated using the predicted and labeled category and offset values.

Exercises

1. Due to space limitations, we have ignored some of the implementation details of SSD models in this experiment. Can you further improve the model in the following areas?

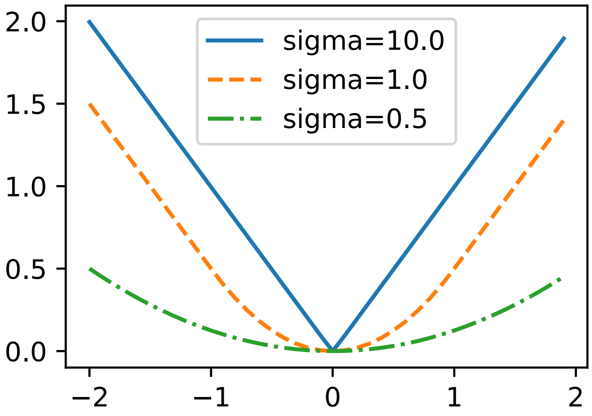
Loss Function

For the predicted offsets, replace L_1 norm loss with L_1 regularization loss. This loss function uses a square function around zero for greater smoothness. This is the regularized area controlled by the hyperparameter σ:

$$f(x) = \begin{cases} (\sigma x)^2/2, & \text{if } |x| < 1/\sigma^2 \\ |x| - 0.5/\sigma^2, & \text{otherwise} \end{cases} \tag{13.7.1}$$

When σ is large, this loss is similar to the L_1 norm loss. When the value is small, the loss function is smoother.

```
sigmas = [10, 1, 0.5]
lines = ['-', '--', '-.']
x = np.arange(-2, 2, 0.1)
d2l.set_figsize()

for l, s in zip(lines, sigmas):
    y = npx.smooth_l1(x, scalar=s)
    d2l.plt.plot(x.asnumpy(), y.asnumpy(), l, label='sigma=%.1f' % s)
d2l.plt.legend();
```

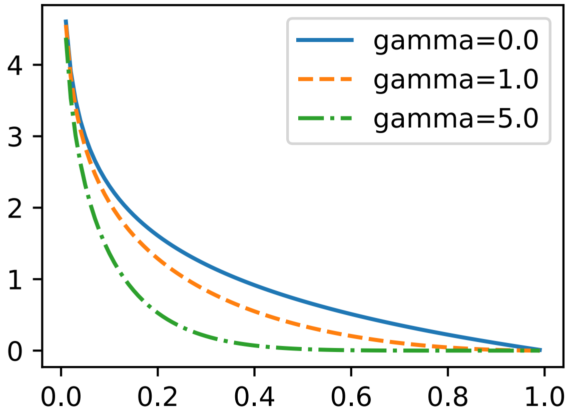


In the experiment, we used cross-entropy loss for category prediction. Now, assume that the prediction probability of the actual category j is p_j and the cross-entropy loss is $-\log p_j$. We can also use the focal loss (Lin et al., 2017). Given the positive hyper-parameters γ and α, this loss is defined as:

$$-\alpha(1-p_j)^{\gamma}\log p_j. \tag{13.7.2}$$

As you can see, by increasing γ, we can effectively reduce the loss when the probability of predicting the correct category is high.

```
def focal_loss(gamma, x):
    return -(1 - x) ** gamma * np.log(x)

x = np.arange(0.01, 1, 0.01)
for l, gamma in zip(lines, [0, 1, 5]):
    y = d2l.plt.plot(x.asnumpy(), focal_loss(gamma, x).asnumpy(), l,
                     label='gamma=%.1f' % gamma)
d2l.plt.legend();
```



Training and Prediction

2. When an object is relatively large compared to the image, the model normally adopts a larger input image size.
3. This generally produces a large number of negative anchor boxes when labeling anchor box categories. We can sample the negative anchor boxes to better balance the data categories. To do this, we can set the `MultiBoxTarget` function's `negative_mining_ratio` parameter.
4. Assign hyper-parameters with different weights to the anchor box category loss and positive anchor box offset loss in the loss function.
5. Refer to the SSD paper. What methods can be used to evaluate the precision of object detection models (Liu et al., 2016)?

13.8 Region-based CNNs (R-CNNs)

Region-based convolutional neural networks or regions with CNN features (R-CNNs) are a pioneering approach that applies deep models to object detection (Girshick et al., 2014). In this section, we will discuss R-CNNs and a series of improvements made to them: Fast R-CNN (Girshick, 2015), Faster R-CNN (Ren et al., 2015), and Mask R-CNN (He et al., 2017a). Due to space limitations, we will confine our discussion to the designs of these models.

13.8.1 R-CNNs

R-CNN models first select several proposed regions from an image (for example, anchor boxes are one type of selection method) and then label their categories and bounding boxes (e.g., offsets). Then, they use a CNN to perform forward computation to extract features from each proposed area. Afterwards, we use the features of each proposed region to predict their categories and bounding boxes. Fig. 13.8.1 shows an R-CNN model.

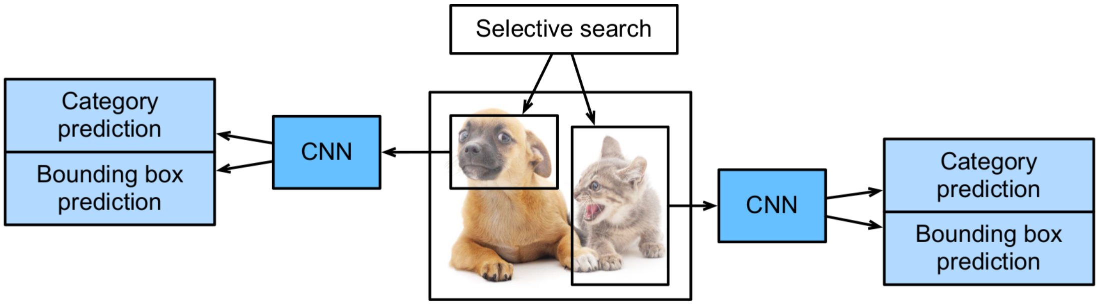


Fig. 13.8.1: R-CNN model.

Specifically, R-CNNs are composed of four main parts:

1. Selective search is performed on the input image to select multiple high-quality proposed regions (Uijlings et al., 2013). These proposed regions are generally selected on multiple scales and have different shapes and sizes. The category and ground-truth bounding box of each proposed region is labeled.
2. A pre-trained CNN is selected and placed, in truncated form, before the output layer. It transforms each proposed region into the input dimensions required by the network and uses forward computation to output the features extracted from the proposed regions.
3. The features and labeled category of each proposed region are combined as an example to train multiple support vector machines for object classification. Here, each support vector machine is used to determine whether an example belongs to a certain category.
4. The features and labeled bounding box of each proposed region are combined as an example to train a linear regression model for ground-truth bounding box prediction.

Although R-CNN models use pre-trained CNNs to effectively extract image features, the main downside is the slow speed. As you can imagine, we can select thousands of proposed regions from a single image, requiring thousands of forward computations from the CNN to perform object detection. This massive computing load means that R-CNNs are not widely used in actual applications.

13.8.2 Fast R-CNN

The main performance bottleneck of an R-CNN model is the need to independently extract features for each proposed region. As these regions have a high degree of overlap, independent feature extraction results in a high volume of repetitive computations. Fast R-CNN improves on the R-CNN by only performing CNN forward computation on the image as a whole.

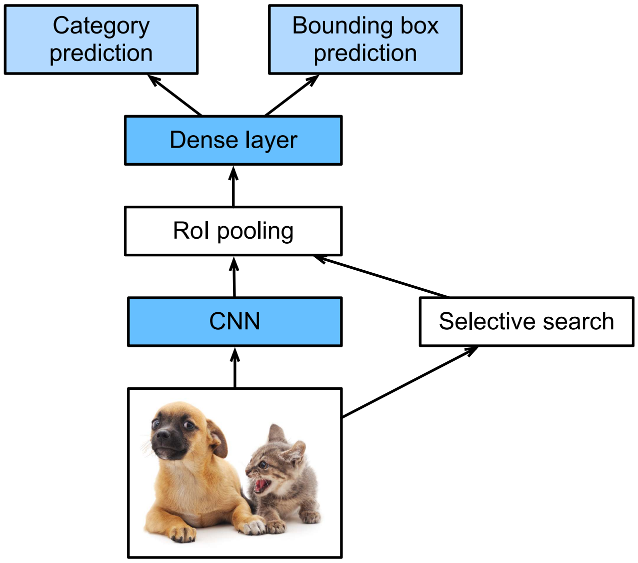


Fig. 13.8.2: Fast R-CNN model.

Fig. 13.8.2 shows a Fast R-CNN model. It is primary computation steps are described below:

1. Compared to an R-CNN model, a Fast R-CNN model uses the entire image as the CNN input for feature extraction, rather than each proposed region. Moreover, this network is generally trained to update the model parameters. As the input is an entire image, the CNN output shape is $1 \times c \times h_1 \times w_1$.

2. Assuming selective search generates n proposed regions, their different shapes indicate regions of interests (RoIs) of different shapes on the CNN output. Features of the same shapes must be extracted from these RoIs (here we assume that the height is h_2 and the width is w_2). Fast R-CNN introduces RoI pooling, which uses the CNN output and RoIs as input to output a concatenation of the features extracted from each proposed region with the shape $n \times c \times h_2 \times w_2$.

3. A fully connected layer is used to transform the output shape to $n \times d$, where d is determined by the model design.

4. During category prediction, the shape of the fully connected layer output is again transformed to $n \times q$ and we use softmax regression (q is the number of categories). During bounding box prediction, the shape of the fully connected layer output is again transformed to $n \times 4$. This means that we predict the category and bounding box for each proposed region.

The RoI pooling layer in Fast R-CNN is somewhat different from the pooling layers we have discussed before. In a normal pooling layer, we set the pooling window, padding, and stride to control the output shape. In an RoI pooling layer, we can directly specify the output shape of each region, such as specifying the height and width of each region as h_2, w_2. Assuming that the height and width of the RoI window are h and w, this window is divided into a grid of sub-windows with the shape $h_2 \times w_2$. The size of each sub-window is about $(h/h_2) \times (w/w_2)$. The sub-window height and width must always be integers and the largest element is used as the output for a given sub-window. This allows the RoI pooling layer to extract features of the same shape from RoIs of different shapes.

In Fig. 13.8.3, we select an 3×3 region as an RoI of the 4×4 input. For this RoI, we use a 2×2 RoI pooling layer to obtain a single 2×2 output. When we divide the region into four sub-windows, they respectively contain the elements 0, 1, 4, and 5 (5 is the largest); 2 and 6 (6 is the largest); 8 and 9 (9 is the largest); and 10.

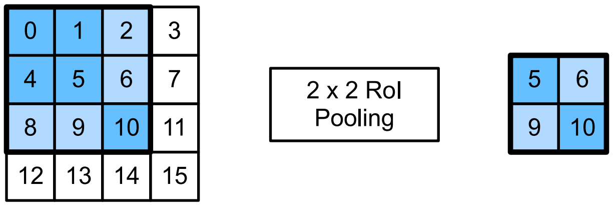


Fig. 13.8.3: 2×2 RoI pooling layer.

We use the `ROIPooling` function to demonstrate the RoI pooling layer computation. Assume that the CNN extracts the feature `X` with both a height and width of 4 and only a single channel.

```
from mxnet import np, npx

npx.set_np()

X = np.arange(16).reshape(1, 1, 4, 4)
X
```

```
array([[[[ 0.,  1.,  2.,  3.],
         [ 4.,  5.,  6.,  7.],
         [ 8.,  9., 10., 11.],
         [12., 13., 14., 15.]]]])
```

Assume that the height and width of the image are both 40 pixels and that selective search generates two proposed regions on the image. Each region is expressed as five elements: the region's object category and the x, y coordinates of its upper-left and bottom-right corners.

```
rois = np.array([[0, 0, 0, 20, 20], [0, 0, 10, 30, 30]])
```

Because the height and width of `X` are $1/10$ of the height and width of the image, the coordinates of the two proposed regions are multiplied by 0.1 according to the `spatial_scale`, and then the RoIs are labeled on `X` as `X[:, :, 0:3, 0:3]` and `X[:, :, 1:4, 0:4]`, respectively. Finally, we divide the two RoIs into a sub-window grid and extract features with a height and width of 2.

```
npx.roi_pooling(X, rois, pooled_size=(2, 2), spatial_scale=0.1)
```

```
array([[[[ 5.,  6.],
         [ 9., 10.]]],


       [[[ 9., 11.],
         [13., 15.]]]])
```

13.8.3 Faster R-CNN

In order to obtain precise object detection results, Fast R-CNN generally requires that many proposed regions be generated in selective search. Faster R-CNN replaces selective search with a region proposal network. This reduces the number of proposed regions generated, while ensuring precise object detection.

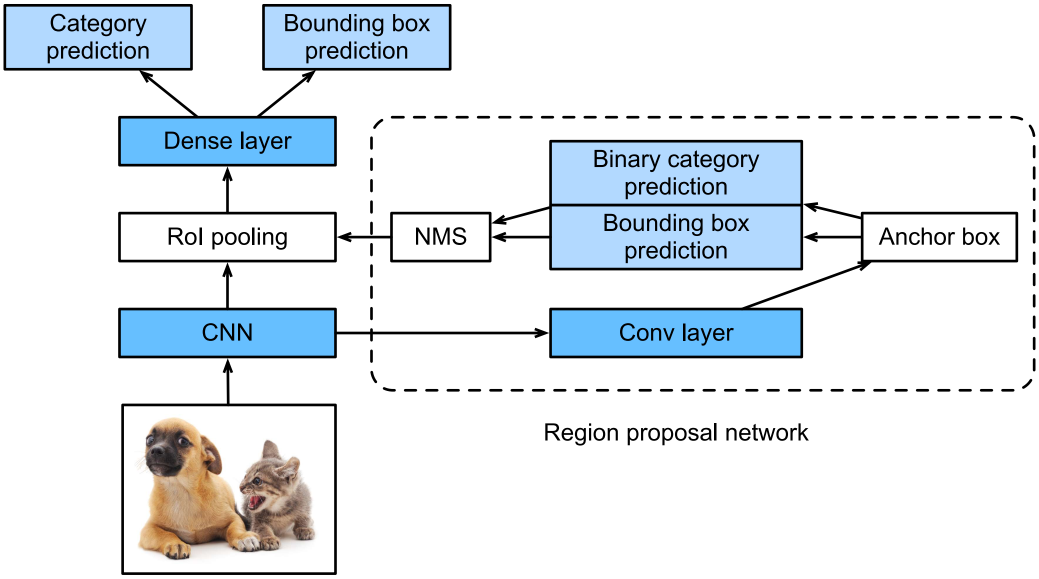


Fig. 13.8.4: Faster R-CNN model.

Fig. 13.8.4 shows a Faster R-CNN model. Compared to Fast R-CNN, Faster R-CNN only changes the method for generating proposed regions from selective search to region proposal network. The other parts of the model remain unchanged. The detailed region proposal network computation process is described below:

1. We use a 3×3 convolutional layer with a padding of 1 to transform the CNN output and set the number of output channels to c. This way, each element in the feature map the CNN extracts from the image is a new feature with a length of c.

2. We use each element in the feature map as a center to generate multiple anchor boxes of different sizes and aspect ratios and then label them.

3. We use the features of the elements of length c at the center on the anchor boxes to predict the binary category (object or background) and bounding box for their respective anchor boxes.

4. Then, we use non-maximum suppression to remove similar bounding box results that correspond to category predictions of "object". Finally, we output the predicted bounding boxes as the proposed regions required by the RoI pooling layer.

It is worth noting that, as a part of the Faster R-CNN model, the region proposal network is trained together with the rest of the model. In addition, the Faster R-CNN object functions include the category and bounding box predictions in object detection, as well as the binary category and bounding box predictions for the anchor boxes in the region proposal network. Finally, the region proposal network can learn how to generate high-quality proposed regions, which reduces the number of proposed regions while maintaining the precision of object detection.

13.8.4 Mask R-CNN

If training data is labeled with the pixel-level positions of each object in an image, a Mask R-CNN model can effectively use these detailed labels to further improve the precision of object detection.

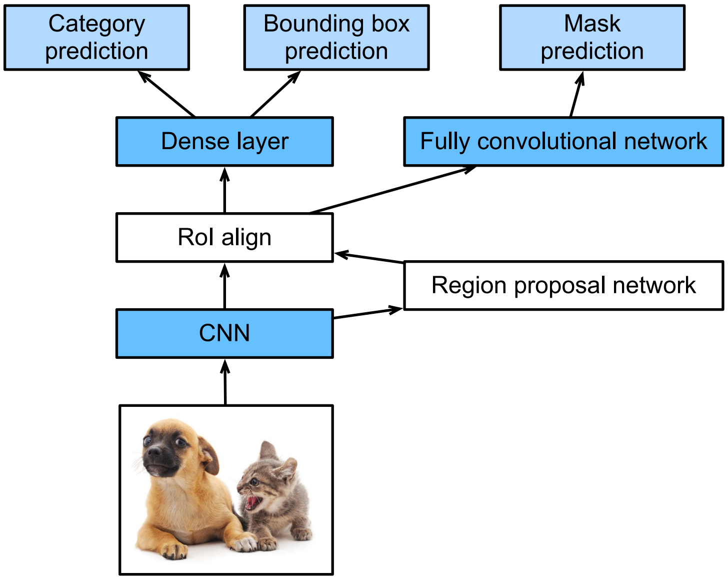


Fig. 13.8.5: Mask R-CNN model.

As shown in Fig. 13.8.5, Mask R-CNN is a modification to the Faster R-CNN model. Mask R-CNN models replace the RoI pooling layer with an RoI alignment layer. This allows the use of bilinear interpolation to retain spatial information on feature maps, making Mask R-CNN better suited for pixel-level predictions. The RoI alignment layer outputs feature maps of the same shape for all RoIs. This not only predicts the categories and bounding boxes of RoIs, but allows us to use an additional fully convolutional network to predict the pixel-level positions of objects. We will describe how to use fully convolutional networks to predict pixel-level semantics in images later in this chapter.

Summary

- An R-CNN model selects several proposed regions and uses a CNN to perform forward computation and extract the features from each proposed region. It then uses these features to predict the categories and bounding boxes of proposed regions.
- Fast R-CNN improves on the R-CNN by only performing CNN forward computation on the image as a whole. It introduces an RoI pooling layer to extract features of the same shape from RoIs of different shapes.
- Faster R-CNN replaces the selective search used in Fast R-CNN with a region proposal network. This reduces the number of proposed regions generated, while ensuring precise object detection.
- Mask R-CNN uses the same basic structure as Faster R-CNN, but adds a fully convolution layer to help locate objects at the pixel level and further improve the precision of object detection.

Exercises

1. Study the implementation of each model in the GluonCV toolkit[206] related to this section.

13.9 Semantic Segmentation and the Dataset

In our discussion of object detection issues in the previous sections, we only used rectangular bounding boxes to label and predict objects in images. In this section, we will look at semantic segmentation, which attempts to segment images into regions with different semantic categories. These semantic regions label and predict objects at the pixel level. Fig. 13.9.1 shows a semantically-segmented image, with areas labeled "dog", "cat", and "background". As you can see, compared to object detection, semantic segmentation labels areas with pixel-level borders, for significantly greater precision.

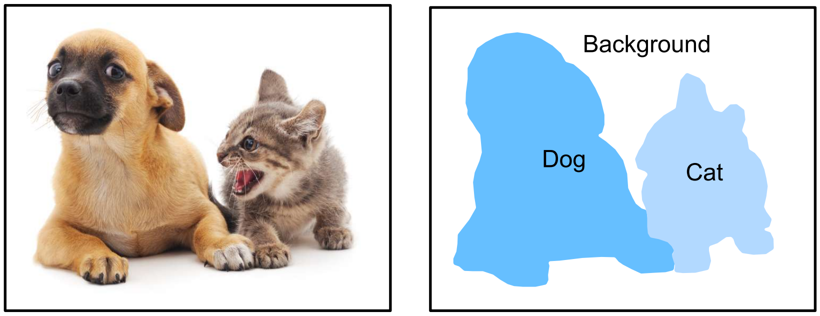


Fig. 13.9.1: Semantically-segmented image, with areas labeled "dog", "cat", and "background".

13.9.1 Image Segmentation and Instance Segmentation

In the computer vision field, there are two important methods related to semantic segmentation: image segmentation and instance segmentation. Here, we will distinguish these concepts from semantic segmentation as follows:

- Image segmentation divides an image into several constituent regions. This method generally uses the correlations between pixels in an image. During training, labels are not needed for image pixels. However, during prediction, this method cannot ensure that the segmented regions have the semantics we want. If we input the image in 9.10, image segmentation might divide the dog into two regions, one covering the dog's mouth and eyes where black is the prominent color and the other covering the rest of the dog where yellow is the prominent color.
- Instance segmentation is also called simultaneous detection and segmentation. This method attempts to identify the pixel-level regions of each object instance in an image. In contrast to semantic segmentation, instance segmentation not only distinguishes semantics,

[206] https://github.com/dmlc/gluon-cv/

but also different object instances. If an image contains two dogs, instance segmentation will distinguish which pixels belong to which dog.

13.9.2 The Pascal VOC2012 Semantic Segmentation Dataset

In the semantic segmentation field, one important dataset is Pascal VOC2012[208]. To better understand this dataset, we must first import the package or module needed for the experiment.

```
%matplotlib inline
import d2l
from mxnet import gluon, image, np, npx
import os
import tarfile

npx.set_np()
```

The original site might be unstable, we download the data from a mirror site. We download the archive to the `../data` path. The archive is about 2GB, so it will take some time to download. After you decompress the archive, the dataset is located in the `../data/VOCdevkit/VOC2012` path.

```
# Saved in the d2l package for later use
def download_voc_pascal(data_dir='../data'):
    """Download the VOC2012 segmentation dataset."""
    voc_dir = os.path.join(data_dir, 'VOCdevkit/VOC2012')
    url = ('http://data.mxnet.io/data/VOCtrainval_11-May-2012.tar')
    sha1 = '4e443f8a2eca6b1dac8a6c57641b67dd40621a49'
    fname = gluon.utils.download(url, data_dir, sha1_hash=sha1)
    with tarfile.open(fname, 'r') as f:
        f.extractall(data_dir)
    return voc_dir

voc_dir = download_voc_pascal()
```

Go to `../data/VOCdevkit/VOC2012` to see the different parts of the dataset. The `ImageSets/Segmentation` path contains text files that specify the training and testing examples. The `JPEGImages` and `SegmentationClass` paths contain the example input images and labels, respectively. These labels are also in image format, with the same dimensions as the input images to which they correspond. In the labels, pixels with the same color belong to the same semantic category. The `read_voc_images` function defined below reads all input images and labels to the memory.

```
# Saved in the d2l package for later use
def read_voc_images(root='../data/VOCdevkit/VOC2012', is_train=True):
    """Read all VOC feature and label images."""
    txt_fname = '%s/ImageSets/Segmentation/%s' % (
        root, 'train.txt' if is_train else 'val.txt')
    with open(txt_fname, 'r') as f:
        images = f.read().split()
    features, labels = [None] * len(images), [None] * len(images)
    for i, fname in enumerate(images):
        features[i] = image.imread('%s/JPEGImages/%s.jpg' % (root, fname))
        labels[i] = image.imread(
```

(continues on next page)

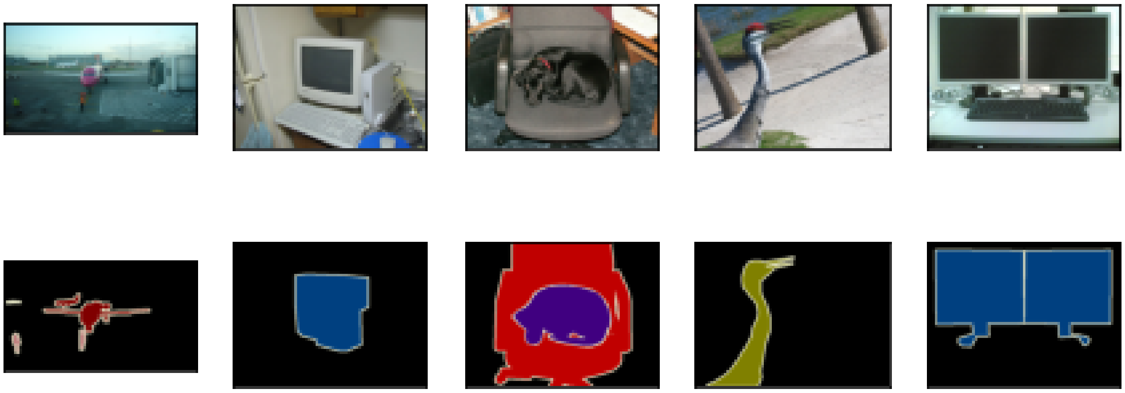
[208] http://host.robots.ox.ac.uk/pascal/VOC/voc2012/

(continued from previous page)

```
            '%s/SegmentationClass/%s.png' % (root, fname))
    return features, labels

train_features, train_labels = read_voc_images(voc_dir, True)
```

We draw the first five input images and their labels. In the label images, white represents borders and black represents the background. Other colors correspond to different categories.

```
n = 5
imgs = train_features[0:n] + train_labels[0:n]
d2l.show_images(imgs, 2, n);
```



Next, we list each RGB color value in the labels and the categories they label.

```
# Saved in the d2l package for later use
VOC_COLORMAP = [[0, 0, 0], [128, 0, 0], [0, 128, 0], [128, 128, 0],
                [0, 0, 128], [128, 0, 128], [0, 128, 128], [128, 128, 128],
                [64, 0, 0], [192, 0, 0], [64, 128, 0], [192, 128, 0],
                [64, 0, 128], [192, 0, 128], [64, 128, 128], [192, 128, 128],
                [0, 64, 0], [128, 64, 0], [0, 192, 0], [128, 192, 0],
                [0, 64, 128]]
# Saved in the d2l package for later use
VOC_CLASSES = ['background', 'aeroplane', 'bicycle', 'bird', 'boat',
               'bottle', 'bus', 'car', 'cat', 'chair', 'cow',
               'diningtable', 'dog', 'horse', 'motorbike', 'person',
               'potted plant', 'sheep', 'sofa', 'train', 'tv/monitor']
```

After defining the two constants above, we can easily find the category index for each pixel in the labels.

```
# Saved in the d2l package for later use
def build_colormap2label():
    """Build a RGB color to label mapping for segmentation."""
    colormap2label = np.zeros(256 ** 3)
    for i, colormap in enumerate(VOC_COLORMAP):
        colormap2label[(colormap[0]*256 + colormap[1])*256 + colormap[2]] = i
    return colormap2label
```

(continues on next page)

(continued from previous page)

```
# Saved in the d2l package for later use
def voc_label_indices(colormap, colormap2label):
    """Map a RGB color to a label."""
    colormap = colormap.astype(np.int32)
    idx = ((colormap[:, :, 0] * 256 + colormap[:, :, 1]) * 256
           + colormap[:, :, 2])
    return colormap2label[idx]
```

For example, in the first example image, the category index for the front part of the airplane is 1 and the index for the background is 0.

```
y = voc_label_indices(train_labels[0], build_colormap2label())
y[105:115, 130:140], VOC_CLASSES[1]
```

```
(array([[0., 0., 0., 0., 0., 0., 0., 0., 0., 1.],
        [0., 0., 0., 0., 0., 0., 0., 1., 1., 1.],
        [0., 0., 0., 0., 0., 0., 1., 1., 1., 1.],
        [0., 0., 0., 0., 0., 1., 1., 1., 1., 1.],
        [0., 0., 0., 0., 0., 1., 1., 1., 1., 1.],
        [0., 0., 0., 0., 1., 1., 1., 1., 1., 1.],
        [0., 0., 0., 0., 0., 1., 1., 1., 1., 1.],
        [0., 0., 0., 0., 0., 1., 1., 1., 1., 1.],
        [0., 0., 0., 0., 0., 0., 1., 1., 1., 1.],
        [0., 0., 0., 0., 0., 0., 0., 0., 1., 1.]]), 'aeroplane')
```

Data Preprocessing

In the preceding chapters, we scaled images to make them fit the input shape of the model. In semantic segmentation, this method would require us to re-map the predicted pixel categories back to the original-size input image. It would be very difficult to do this precisely, especially in segmented regions with different semantics. To avoid this problem, we crop the images to set dimensions and do not scale them. Specifically, we use the random cropping method used in image augmentation to crop the same region from input images and their labels.

```
# Saved in the d2l package for later use
def voc_rand_crop(feature, label, height, width):
    """Randomly crop for both feature and label images."""
    feature, rect = image.random_crop(feature, (width, height))
    label = image.fixed_crop(label, *rect)
    return feature, label

imgs = []
for _ in range(n):
    imgs += voc_rand_crop(train_features[0], train_labels[0], 200, 300)
d2l.show_images(imgs[::2] + imgs[1::2], 2, n);
```

Dataset Classes for Custom Semantic Segmentation

We use the inherited `Dataset` class provided by Gluon to customize the semantic segmentation dataset class `VOCSegDataset`. By implementing the `__getitem__` function, we can arbitrarily access the input image with the index `idx` and the category indexes for each of its pixels from the dataset. As some images in the dataset may be smaller than the output dimensions specified for random cropping, we must remove these example by using a custom `filter` function. In addition, we define the `normalize_image` function to normalize each of the three RGB channels of the input images.

```
# Saved in the d2l package for later use
class VOCSegDataset(gluon.data.Dataset):
    """A customized dataset to load VOC dataset."""

    def __init__(self, is_train, crop_size, voc_dir):
        self.rgb_mean = np.array([0.485, 0.456, 0.406])
        self.rgb_std = np.array([0.229, 0.224, 0.225])
        self.crop_size = crop_size
        features, labels = read_voc_images(root=voc_dir, is_train=is_train)
        self.features = [self.normalize_image(feature)
                         for feature in self.filter(features)]
        self.labels = self.filter(labels)
        self.colormap2label = build_colormap2label()
        print('read ' + str(len(self.features)) + ' examples')

    def normalize_image(self, img):
        return (img.astype('float32') / 255 - self.rgb_mean) / self.rgb_std

    def filter(self, imgs):
        return [img for img in imgs if (
            img.shape[0] >= self.crop_size[0] and
            img.shape[1] >= self.crop_size[1])]

    def __getitem__(self, idx):
        feature, label = voc_rand_crop(self.features[idx], self.labels[idx],
                                       *self.crop_size)
        return (feature.transpose(2, 0, 1),
                voc_label_indices(label, self.colormap2label))

    def __len__(self):
        return len(self.features)
```

Reading the Dataset

Using the custom `VOCSegDataset` class, we create the training set and testing set instances. We assume the random cropping operation output images in the shape 320×480. Below, we can see the number of examples retained in the training and testing sets.

```
crop_size = (320, 480)
voc_train = VOCSegDataset(True, crop_size, voc_dir)
voc_test = VOCSegDataset(False, crop_size, voc_dir)
```

```
read 1114 examples
read 1078 examples
```

We set the batch size to 64 and define the iterators for the training and testing sets. Print the shape of the first minibatch. In contrast to image classification and object recognition, labels here are three-dimensional arrays.

```
batch_size = 64
train_iter = gluon.data.DataLoader(voc_train, batch_size, shuffle=True,
                                   last_batch='discard',
                                   num_workers=d2l.get_dataloader_workers())
for X, Y in train_iter:
    print(X.shape)
    print(Y.shape)
    break
```

```
(64, 3, 320, 480)
(64, 320, 480)
```

Putting All Things Together

Finally, we define a function `load_data_voc` that downloads and loads this dataset, and then returns the data loaders.

```
# Saved in the d2l package for later use
def load_data_voc(batch_size, crop_size):
    """Download and load the VOC2012 semantic dataset."""
    voc_dir = d2l.download_voc_pascal()
    num_workers = d2l.get_dataloader_workers()
    train_iter = gluon.data.DataLoader(
        VOCSegDataset(True, crop_size, voc_dir), batch_size,
        shuffle=True, last_batch='discard', num_workers=num_workers)
    test_iter = gluon.data.DataLoader(
        VOCSegDataset(False, crop_size, voc_dir), batch_size,
        last_batch='discard', num_workers=num_workers)
    return train_iter, test_iter
```

Summary

- Semantic segmentation looks at how images can be segmented into regions with different semantic categories.
- In the semantic segmentation field, one important dataset is Pascal VOC2012.
- Because the input images and labels in semantic segmentation have a one-to-one correspondence at the pixel level, we randomly crop them to a fixed size, rather than scaling them.

Exercises

1. Recall the content we covered in Section 13.1. Which of the image augmentation methods used in image classification would be hard to use in semantic segmentation?

13.10 Transposed Convolution

The layers we introduced so far for convolutional neural networks, including convolutional layers (Section 6.2) and pooling layers (Section 6.5), often reduce the input width and height, or keep them unchanged. Applications such as semantic segmentation (Section 13.9) and generative adversarial networks (Section 16.2), however, require to predict values for each pixel and therefore needs to increase input width and height. Transposed convolution, also named fractionally-strided convolution (Dumoulin & Visin, 2016) or deconvolution (Long et al., 2015), serves this purpose.

```
from mxnet import np, npx, init
from mxnet.gluon import nn
import d2l

npx.set_np()
```

13.10.1 Basic 2D Transposed Convolution

Let's consider a basic case that both input and output channels are 1, with 0 padding and 1 stride. Fig. 13.10.1 illustrates how transposed convolution with a 2×2 kernel is computed on the 2×2 input matrix.

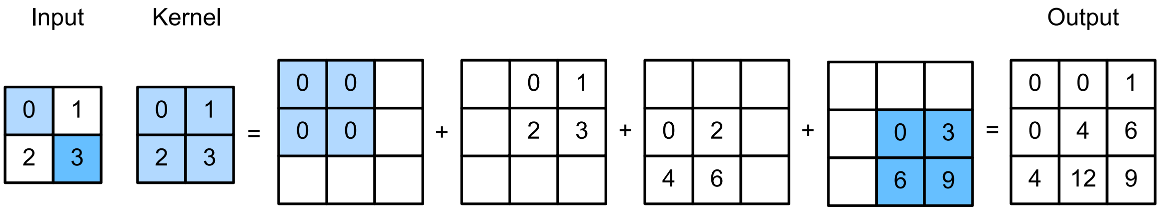


Fig. 13.10.1: Transposed convolution layer with a 2×2 kernel.

We can implement this operation by giving matrix kernel K and matrix input X.

```python
def trans_conv(X, K):
    h, w = K.shape
    Y = np.zeros((X.shape[0] + h - 1, X.shape[1] + w - 1))
    for i in range(X.shape[0]):
        for j in range(X.shape[1]):
            Y[i: i + h, j: j + w] += X[i, j] * K
    return Y
```

Remember the convolution computes results by `Y[i, j] = (X[i: i + h, j: j + w] * K).sum()` (refer to `corr2d` in Section 6.2), which summarizes input values through the kernel. While the transposed convolution broadcasts input values through the kernel, which results in a larger output shape.

Verify the results in Fig. 13.10.1.

```python
X = np.array([[0, 1], [2, 3]])
K = np.array([[0, 1], [2, 3]])
trans_conv(X, K)
```

```
array([[ 0.,  0.,  1.],
       [ 0.,  4.,  6.],
       [ 4., 12.,  9.]])
```

Or we can use `nn.Conv2DTranspose` to obtain the same results. As `nn.Conv2D`, both input and kernel should be 4-D tensors.

```python
X, K = X.reshape(1, 1, 2, 2), K.reshape(1, 1, 2, 2)
tconv = nn.Conv2DTranspose(1, kernel_size=2)
tconv.initialize(init.Constant(K))
tconv(X)
```

```
array([[[[ 0.,  0.,  1.],
         [ 0.,  4.,  6.],
         [ 4., 12.,  9.]]]])
```

13.10.2 Padding, Strides, and Channels

We apply padding elements to the input in convolution, while they are applied to the output in transposed convolution. A 1×1 padding means we first compute the output as normal, then remove the first/last rows and columns.

```
tconv = nn.Conv2DTranspose(1, kernel_size=2, padding=1)
tconv.initialize(init.Constant(K))
tconv(X)
```

```
array([[[[4.]]]])
```

Similarly, strides are applied to outputs as well.

```
tconv = nn.Conv2DTranspose(1, kernel_size=2, strides=2)
tconv.initialize(init.Constant(K))
tconv(X)
```

```
array([[[[0., 0., 0., 1.],
         [0., 0., 2., 3.],
         [0., 2., 0., 3.],
         [4., 6., 6., 9.]]]])
```

The multi-channel extension of the transposed convolution is the same as the convolution. When the input has multiple channels, denoted by c_i, the transposed convolution assigns a $k_h \times k_w$ kernel matrix to each input channel. If the output has a channel size c_o, then we have a $c_i \times k_h \times k_w$ kernel for each output channel.

As a result, if we feed X into a convolutional layer f to compute $Y = f(X)$ and create a transposed convolution layer g with the same hyper-parameters as f except for the output channel set to be the channel size of X, then $g(Y)$ should has the same shape as X. Let's verify this statement.

```
X = np.random.uniform(size=(1, 10, 16, 16))
conv = nn.Conv2D(20, kernel_size=5, padding=2, strides=3)
tconv = nn.Conv2DTranspose(10, kernel_size=5, padding=2, strides=3)
conv.initialize()
tconv.initialize()
tconv(conv(X)).shape == X.shape
```

```
True
```

13.10.3 Analogy to Matrix Transposition

The transposed convolution takes its name from the matrix transposition. In fact, convolution operations can also be achieved by matrix multiplication. In the example below, we define a $3\times$ input X with a 2×2 kernel K, and then use `corr2d` to compute the convolution output.

```
X = np.arange(9).reshape(3, 3)
K = np.array([[0, 1], [2, 3]])
```

(continues on next page)

(continued from previous page)

```
Y = d2l.corr2d(X, K)
Y
```

```
array([[19., 25.],
       [37., 43.]])
```

Next, we rewrite convolution kernel K as a matrix W. Its shape will be $(4, 9)$, where the i^th row present applying the kernel to the input to generate the i^th output element.

```
def kernel2matrix(K):
    k, W = np.zeros(5), np.zeros((4, 9))
    k[:2], k[3:5] = K[0, :], K[1, :]
    W[0, :5], W[1, 1:6], W[2, 3:8], W[3, 4:] = k, k, k, k
    return W

W = kernel2matrix(K)
W
```

```
array([[0., 1., 0., 2., 3., 0., 0., 0., 0.],
       [0., 0., 1., 0., 2., 3., 0., 0., 0.],
       [0., 0., 0., 0., 1., 0., 2., 3., 0.],
       [0., 0., 0., 0., 0., 1., 0., 2., 3.]])
```

Then the convolution operator can be implemented by matrix multiplication with proper reshaping.

```
Y == np.dot(W, X.reshape(-1)).reshape(2, 2)
```

```
array([[ True,  True],
       [ True,  True]])
```

We can implement transposed convolution as a matrix multiplication as well by reusing `kernel2matrix`. To reuse the generated W, we construct a 2×2 input, so the corresponding weight matrix will have a shape $(9, 4)$, which is $W^\top$. Let's verify the results.

```
X = np.array([[0, 1], [2, 3]])
Y = trans_conv(X, K)
Y == np.dot(W.T, X.reshape(-1)).reshape(3, 3)
```

```
array([[ True,  True,  True],
       [ True,  True,  True],
       [ True,  True,  True]])
```

Summary

- Compared to convolutions that reduce inputs through kernels, transposed convolutions broadcast inputs.
- If a convolution layer reduces the input width and height by n_w and h_h time, respectively. Then a transposed convolution layer with the same kernel sizes, padding and strides will increase the input width and height by n_w and n_h, respectively.
- We can implement convolution operations by the matrix multiplication, the corresponding transposed convolutions can be done by transposed matrix multiplication.

Exercises

1. Is it efficient to use matrix multiplication to implement convolution operations? Why?

13.11 Fully Convolutional Networks (FCN)

We previously discussed semantic segmentation using each pixel in an image for category prediction. A fully convolutional network (FCN) (Long et al., 2015) uses a convolutional neural network to transform image pixels to pixel categories. Unlike the convolutional neural networks previously introduced, an FCN transforms the height and width of the intermediate layer feature map back to the size of input image through the transposed convolution layer, so that the predictions have a one-to-one correspondence with input image in spatial dimension (height and width). Given a position on the spatial dimension, the output of the channel dimension will be a category prediction of the pixel corresponding to the location.

We will first import the package or module needed for the experiment and then explain the transposed convolution layer.

```
%matplotlib inline
import d2l
from mxnet import gluon, image, init, np, npx
from mxnet.gluon import nn

npx.set_np()
```

13.11.1 Constructing a Model

Here, we demonstrate the most basic design of a fully convolutional network model. As shown in Fig. 13.11.1, the fully convolutional network first uses the convolutional neural network to extract image features, then transforms the number of channels into the number of categories through the 1×1 convolution layer, and finally transforms the height and width of the feature map to the size of the input image by using the transposed convolution layer Section 13.10. The model output has the same height and width as the input image and has a one-to-one correspondence in spatial positions. The final output channel contains the category prediction of the pixel of the corresponding spatial position.

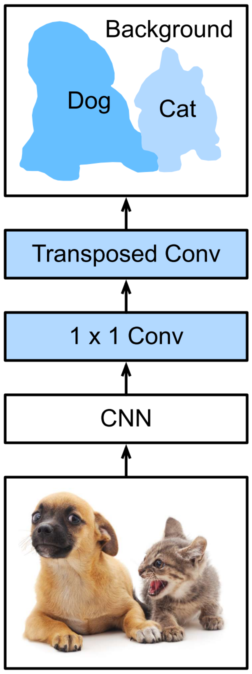


Fig. 13.11.1: Fully convolutional network.

Below, we use a ResNet-18 model pre-trained on the ImageNet dataset to extract image features and record the network instance as `pretrained_net`. As you can see, the last two layers of the model member variable `features` are the global maximum pooling layer `GlobalAvgPool2D` and example flattening layer `Flatten`. The `output` module contains the fully connected layer used for output. These layers are not required for a fully convolutional network.

```
pretrained_net = gluon.model_zoo.vision.resnet18_v2(pretrained=True)
pretrained_net.features[-4:], pretrained_net.output
```

```
(HybridSequential(
   (0): BatchNorm(axis=1, eps=1e-05, momentum=0.9, fix_gamma=False, use_global_stats=False,␣
↪in_channels=512)
   (1): Activation(relu)
   (2): GlobalAvgPool2D(size=(1, 1), stride=(1, 1), padding=(0, 0), ceil_mode=True, global_
↪pool=True, pool_type=avg, layout=NCHW)
   (3): Flatten
 ), Dense(512 -> 1000, linear))
```

Next, we create the fully convolutional network instance `net`. It duplicates all the neural layers except the last two layers of the instance member variable features of pretrained_net and the model parameters obtained after pre-training.

```
net = nn.HybridSequential()
for layer in pretrained_net.features[:-2]:
    net.add(layer)
```

Given an input of a height and width of 320 and 480 respectively, the forward computation of net will reduce the height and width of the input to $1/32$ of the original, i.e., 10 and 15.

```
X = np.random.uniform(size=(1, 3, 320, 480))
net(X).shape
```

```
(1, 512, 10, 15)
```

Next, we transform the number of output channels to the number of categories of Pascal VOC2012 (21) through the 1×1 convolution layer. Finally, we need to magnify the height and width of the feature map by a factor of 32 to change them back to the height and width of the input image. Recall the calculation method for the convolution layer output shape described in Section 6.3. Because $(320 - 64 + 16 \times 2 + 32)/32 = 10$ and $(480 - 64 + 16 \times 2 + 32)/32 = 15$, we construct a transposed convolution layer with a stride of 32 and set the height and width of the convolution kernel to 64 and the padding to 16. It is not difficult to see that, if the stride is s, the padding is $s/2$ (assuming $s/2$ is an integer), and the height and width of the convolution kernel are $2s$, the transposed convolution kernel will magnify both the height and width of the input by a factor of s.

```
num_classes = 21
net.add(nn.Conv2D(num_classes, kernel_size=1),
        nn.Conv2DTranspose(
            num_classes, kernel_size=64, padding=16, strides=32))
```

13.11.2 Initializing the Transposed Convolution Layer

We already know that the transposed convolution layer can magnify a feature map. In image processing, sometimes we need to magnify the image, i.e., upsampling. There are many methods for upsampling, and one common method is bilinear interpolation. Simply speaking, in order to get the pixel of the output image at the coordinates (x, y), the coordinates are first mapped to the coordinates of the input image (x', y'). This can be done based on the ratio of the size of thee input to the size of the output. The mapped values x' and y' are usually real numbers. Then, we find the four pixels closest to the coordinate (x', y') on the input image. Finally, the pixels of the output image at coordinates (x, y) are calculated based on these four pixels on the input image and their relative distances to (x', y'). Upsampling by bilinear interpolation can be implemented by transposed convolution layer of the convolution kernel constructed using the following bilinear_kernel function. Due to space limitations, we only give the implementation of the bilinear_kernel function and will not discuss the principles of the algorithm.

```
def bilinear_kernel(in_channels, out_channels, kernel_size):
    factor = (kernel_size + 1) // 2
    if kernel_size % 2 == 1:
```

(continues on next page)

(continued from previous page)

```
        center = factor - 1
    else:
        center = factor - 0.5
    og = (np.arange(kernel_size).reshape(-1, 1),
          np.arange(kernel_size).reshape(1, -1))
    filt = (1 - np.abs(og[0] - center) / factor) * \
           (1 - np.abs(og[1] - center) / factor)
    weight = np.zeros((in_channels, out_channels, kernel_size, kernel_size))
    weight[range(in_channels), range(out_channels), :, :] = filt
    return np.array(weight)
```

Now, we will experiment with bilinear interpolation upsampling implemented by transposed convolution layers. Construct a transposed convolution layer that magnifies height and width of input by a factor of 2 and initialize its convolution kernel with the `bilinear_kernel` function.

```
conv_trans = nn.Conv2DTranspose(3, kernel_size=4, padding=1, strides=2)
conv_trans.initialize(init.Constant(bilinear_kernel(3, 3, 4)))
```

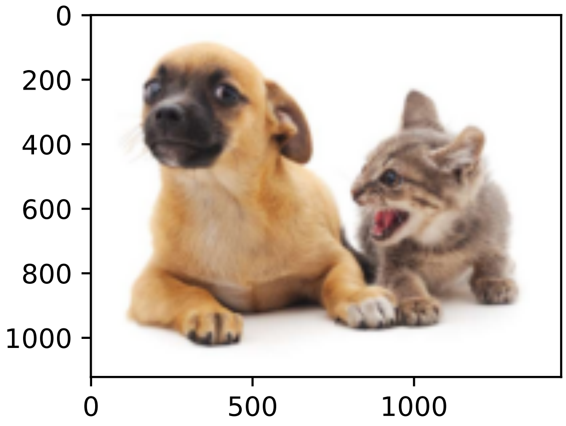
Read the image `X` and record the result of upsampling as `Y`. In order to print the image, we need to adjust the position of the channel dimension.

```
img = image.imread('../img/catdog.jpg')
X = np.expand_dims(img.astype('float32').transpose(2, 0, 1), axis=0) / 255
Y = conv_trans(X)
out_img = Y[0].transpose(1, 2, 0)
```

As you can see, the transposed convolution layer magnifies both the height and width of the image by a factor of 2. It is worth mentioning that, besides to the difference in coordinate scale, the image magnified by bilinear interpolation and original image printed in Section 13.3 look the same.

```
d2l.set_figsize((3.5, 2.5))
print('input image shape:', img.shape)
d2l.plt.imshow(img.asnumpy());
print('output image shape:', out_img.shape)
d2l.plt.imshow(out_img.asnumpy());
```

```
input image shape: (561, 728, 3)
output image shape: (1122, 1456, 3)
```



In a fully convolutional network, we initialize the transposed convolution layer for upsampled bilinear interpolation. For a 1×1 convolution layer, we use Xavier for randomly initialization.

```
W = bilinear_kernel(num_classes, num_classes, 64)
net[-1].initialize(init.Constant(W))
net[-2].initialize(init=init.Xavier())
```

13.11.3 Reading the Dataset

We read the dataset using the method described in the previous section. Here, we specify shape of the randomly cropped output image as 320×480, so both the height and width are divisible by 32.

```
batch_size, crop_size = 32, (320, 480)
train_iter, test_iter = d2l.load_data_voc(batch_size, crop_size)
```

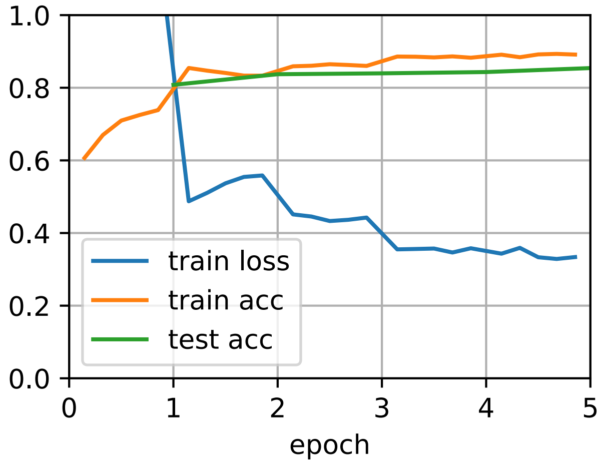
```
Downloading ../data/VOCtrainval_11-May-2012.tar from http://data.mxnet.io/data/VOCtrainval_
↪11-May-2012.tar...
read 1114 examples
read 1078 examples
```

13.11.4 Training

Now we can start training the model. The loss function and accuracy calculation here are not substantially different from those used in image classification. Because we use the channel of the transposed convolution layer to predict pixel categories, the axis=1 (channel dimension) option is specified in SoftmaxCrossEntropyLoss. In addition, the model calculates the accuracy based on whether the prediction category of each pixel is correct.

```
num_epochs, lr, wd, ctx = 5, 0.1, 1e-3, d2l.try_all_gpus()
loss = gluon.loss.SoftmaxCrossEntropyLoss(axis=1)
net.collect_params().reset_ctx(ctx)
trainer = gluon.Trainer(net.collect_params(), 'sgd',
                        {'learning_rate': lr, 'wd': wd})
d2l.train_ch13(net, train_iter, test_iter, loss, trainer, num_epochs, ctx)
```

```
loss 0.335, train acc 0.891, test acc 0.854
305.6 exampes/sec on [gpu(0), gpu(1)]
```



13.11.5 Prediction

During predicting, we need to standardize the input image in each channel and transform them into the four-dimensional input format required by the convolutional neural network.

```
def predict(img):
    X = test_iter._dataset.normalize_image(img)
    X = np.expand_dims(X.transpose(2, 0, 1), axis=0)
    pred = net(X.as_in_context(ctx[0])).argmax(axis=1)
    return pred.reshape(pred.shape[1], pred.shape[2])
```

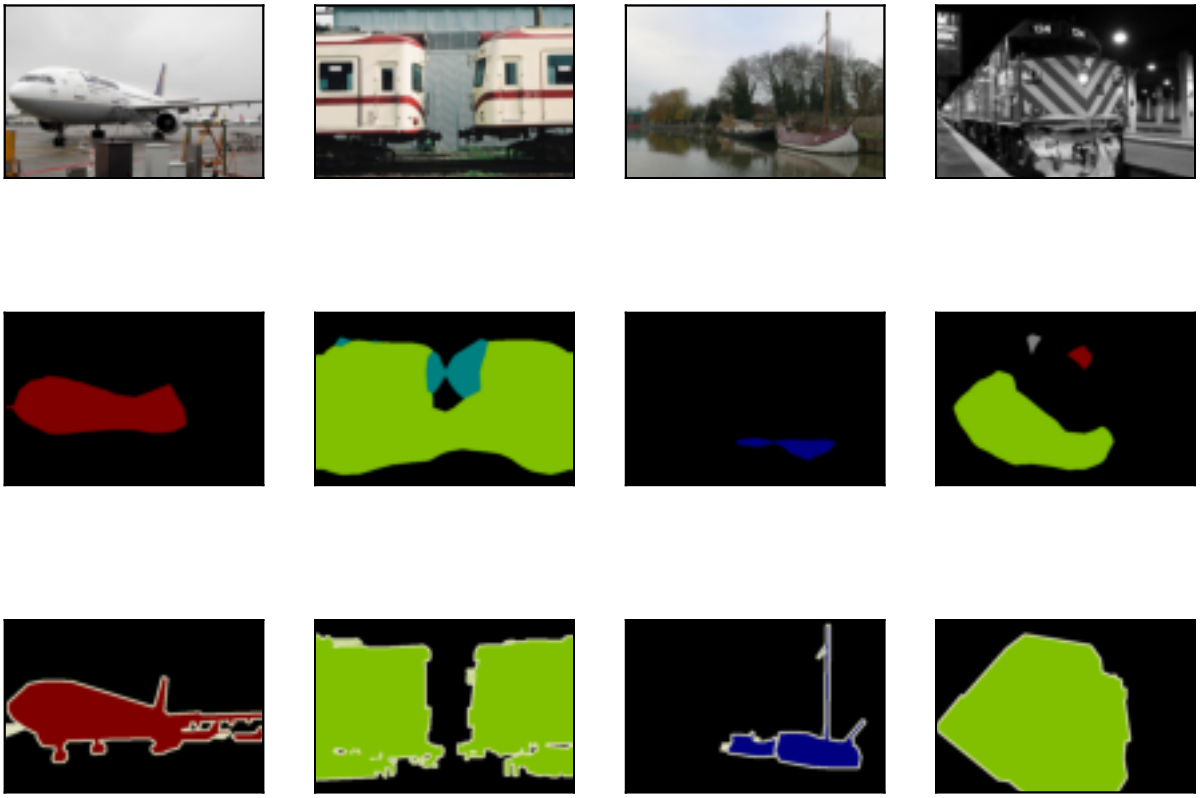
To visualize the predicted categories for each pixel, we map the predicted categories back to their labeled colors in the dataset.

```
def label2image(pred):
    colormap = np.array(d2l.VOC_COLORMAP, ctx=ctx[0], dtype='uint8')
    X = pred.astype('int32')
    return colormap[X, :]
```

The size and shape of the images in the test dataset vary. Because the model uses a transposed convolution layer with a stride of 32, when the height or width of the input image is not divisible by 32, the height or width of the transposed convolution layer output deviates from the size of the input image. In order to solve this problem, we can crop multiple rectangular areas in the image with heights and widths as integer multiples of 32, and then perform forward computation on the pixels in these areas. When combined, these areas must completely cover the input image. When a pixel is covered by multiple areas, the average of the transposed convolution layer output in the forward computation of the different areas can be used as an input for the softmax operation to predict the category.

For the sake of simplicity, we only read a few large test images and crop an area with a shape of 320×480 from the top-left corner of the image. Only this area is used for prediction. For the input image, we print the cropped area first, then print the predicted result, and finally print the labeled category.

```
test_images, test_labels = d2l.read_voc_images(is_train=False)
n, imgs = 4, []
for i in range(n):
    crop_rect = (0, 0, 480, 320)
    X = image.fixed_crop(test_images[i], *crop_rect)
    pred = label2image(predict(X))
    imgs += [X, pred, image.fixed_crop(test_labels[i], *crop_rect)]
d2l.show_images(imgs[::3] + imgs[1::3] + imgs[2::3], 3, n, scale=2);
```



Summary

- The fully convolutional network first uses the convolutional neural network to extract image features, then transforms the number of channels into the number of categories through the 1×1 convolution layer, and finally transforms the height and width of the feature map to the size of the input image by using the transposed convolution layer to output the category of each pixel.
- In a fully convolutional network, we initialize the transposed convolution layer for upsampled bilinear interpolation.

Exercises

1. If we use Xavier to randomly initialize the transposed convolution layer, what will happen to the result?
2. Can you further improve the accuracy of the model by tuning the hyper-parameters?
3. Predict the categories of all pixels in the test image.
4. The outputs of some intermediate layers of the convolutional neural network are also used in the paper on fully convolutional networks[1]. Try to implement this idea.

13.12 Neural Style Transfer

If you use social sharing apps or happen to be an amateur photographer, you are familiar with filters. Filters can alter the color styles of photos to make the background sharper or people's faces whiter. However, a filter generally can only change one aspect of a photo. To create the ideal photo, you often need to try many different filter combinations. This process is as complex as tuning the hyper-parameters of a model.

In this section, we will discuss how we can use convolution neural networks (CNNs) to automatically apply the style of one image to another image, an operation known as style transfer (Gatys et al., 2016). Here, we need two input images, one content image and one style image. We use a neural network to alter the content image so that its style mirrors that of the style image. In Fig. 13.12.1, the content image is a landscape photo the author took in Mount Rainier National Part near Seattle. The style image is an oil painting of oak trees in autumn. The output composite image retains the overall shapes of the objects in the content image, but applies the oil painting brushwork of the style image and makes the overall color more vivid.

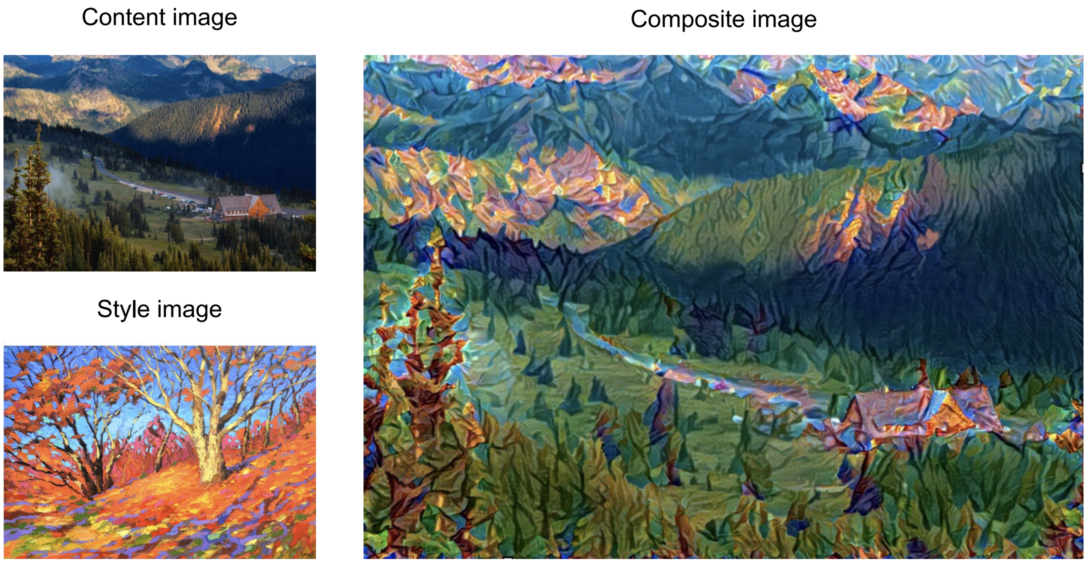


Fig. 13.12.1: Content and style input images and composite image produced by style transfer.

13.12.1 Technique

The CNN-based style transfer model is shown in Fig. 13.12.2. First, we initialize the composite image. For example, we can initialize it as the content image. This composite image is the only variable that needs to be updated in the style transfer process, i.e., the model parameter to be updated in style transfer. Then, we select a pre-trained CNN to extract image features. These model parameters do not need to be updated during training. The deep CNN uses multiple neural layers that successively extract image features. We can select the output of certain layers to use as content features or style features. If we use the structure in Fig. 13.12.2, the pre-trained neural network contains three convolutional layers. The second layer outputs the image content features, while the outputs of the first and third layers are used as style features. Next, we use forward propagation (in the direction of the solid lines) to compute the style transfer loss function and backward propagation (in the direction of the dotted lines) to update the model parameter, constantly updating the composite image. The loss functions used in style transfer generally have three parts: 1. Content loss is used to make the composite image approximate the content image as regards content features. 2. Style loss is used to make the composite image approximate the style image in terms of style features. 3. Total variation loss helps reduce the noise in the composite image. Finally, after we finish training the model, we output the style transfer model parameters to obtain the final composite image.

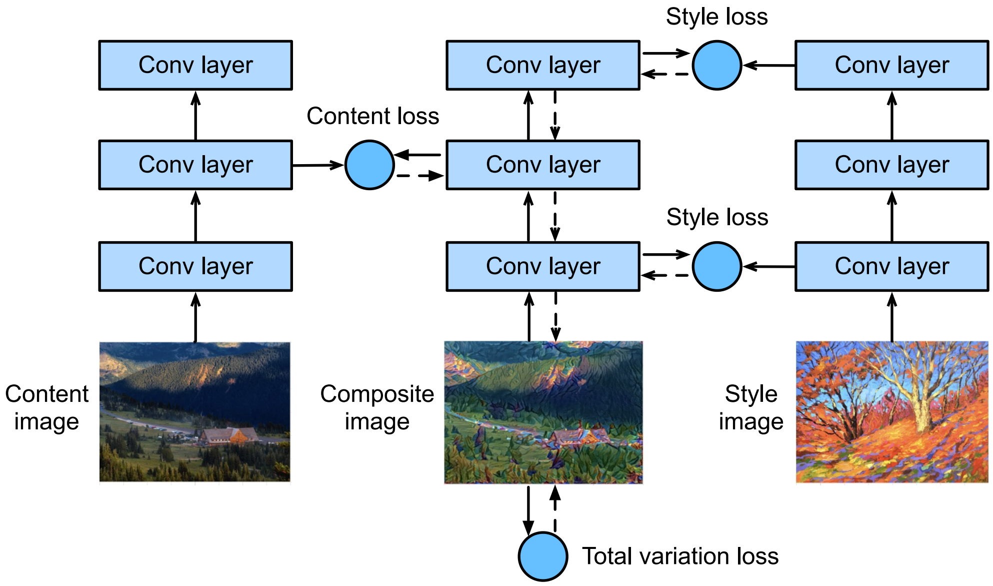


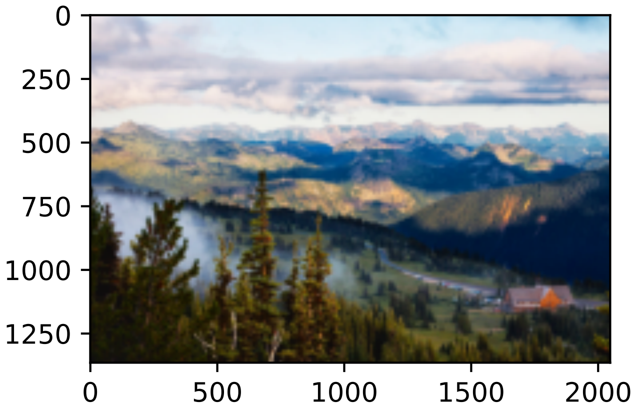
Fig. 13.12.2: CNN-based style transfer process. Solid lines show the direction of forward propagation and dotted lines show backward propagation.

Next, we will perform an experiment to help us better understand the technical details of style transfer.

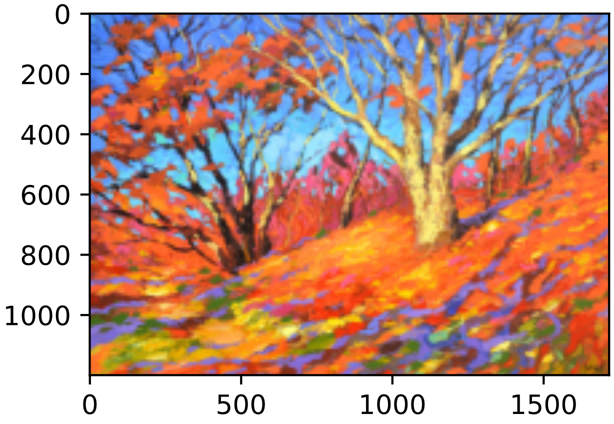
13.12.2 Reading the Content and Style Images

First, we read the content and style images. By printing out the image coordinate axes, we can see that they have different dimensions.

```
%matplotlib inline
import d2l
from mxnet import autograd, gluon, image, init, np, npx
from mxnet.gluon import nn

npx.set_np()

d2l.set_figsize((3.5, 2.5))
content_img = image.imread('../img/rainier.jpg')
d2l.plt.imshow(content_img.asnumpy());
```



```
style_img = image.imread('../img/autumn_oak.jpg')
d2l.plt.imshow(style_img.asnumpy());
```



13.12.3 Preprocessing and Postprocessing

Below, we define the functions for image preprocessing and postprocessing. The preprocess function normalizes each of the three RGB channels of the input images and transforms the results to a format that can be input to the CNN. The postprocess function restores the pixel values in the output image to their original values before normalization. Because the image printing function requires that each pixel has a floating point value from 0 to 1, we use the clip function to replace values smaller than 0 or greater than 1 with 0 or 1, respectively.

```
rgb_mean = np.array([0.485, 0.456, 0.406])
rgb_std = np.array([0.229, 0.224, 0.225])

def preprocess(img, image_shape):
    img = image.imresize(img, *image_shape)
    img = (img.astype('float32') / 255 - rgb_mean) / rgb_std
    return np.expand_dims(img.transpose(2, 0, 1), axis=0)

def postprocess(img):
    img = img[0].as_in_context(rgb_std.context)
    return (img.transpose(1, 2, 0) * rgb_std + rgb_mean).clip(0, 1)
```

13.12.4 Extracting Features

We use the VGG-19 model pre-trained on the ImageNet dataset to extract image features[1].

```
pretrained_net = gluon.model_zoo.vision.vgg19(pretrained=True)
```

To extract image content and style features, we can select the outputs of certain layers in the VGG network. In general, the closer an output is to the input layer, the easier it is to extract image detail information. The farther away an output is, the easier it is to extract global information. To prevent the composite image from retaining too many details from the content image, we select a VGG network layer near the output layer to output the image content features. This layer is called the content layer. We also select the outputs of different layers from the VGG network for matching local and global styles. These are called the style layers. As we mentioned in Section 7.2, VGG networks have five convolutional blocks. In this experiment, we select the last convolutional layer of the fourth convolutional block as the content layer and the first layer of each block as style layers. We can obtain the indexes for these layers by printing the `pretrained_net` instance.

```
style_layers, content_layers = [0, 5, 10, 19, 28], [25]
```

During feature extraction, we only need to use all the VGG layers from the input layer to the content or style layer nearest the output layer. Below, we build a new network, `net`, which only retains the layers in the VGG network we need to use. We then use `net` to extract features.

```
net = nn.Sequential()
for i in range(max(content_layers + style_layers) + 1):
    net.add(pretrained_net.features[i])
```

Given input `X`, if we simply call the forward computation `net(X)`, we can only obtain the output of the last layer. Because we also need the outputs of the intermediate layers, we need to perform layer-by-layer computation and retain the content and style layer outputs.

```
def extract_features(X, content_layers, style_layers):
    contents = []
    styles = []
    for i in range(len(net)):
        X = net[i](X)
        if i in style_layers:
            styles.append(X)
        if i in content_layers:
            contents.append(X)
    return contents, styles
```

Next, we define two functions: The `get_contents` function obtains the content features extracted from the content image, while the `get_styles` function obtains the style features extracted from the style image. Because we do not need to change the parameters of the pre-trained VGG model during training, we can extract the content features from the content image and style features from the style image before the start of training. As the composite image is the model parameter that must be updated during style transfer, we can only call the `extract_features` function during training to extract the content and style features of the composite image.

```
def get_contents(image_shape, ctx):
    content_X = preprocess(content_img, image_shape).copyto(ctx)
```

(continues on next page)

(continued from previous page)

```
    contents_Y, _ = extract_features(content_X, content_layers, style_layers)
    return content_X, contents_Y

def get_styles(image_shape, ctx):
    style_X = preprocess(style_img, image_shape).copyto(ctx)
    _, styles_Y = extract_features(style_X, content_layers, style_layers)
    return style_X, styles_Y
```

13.12.5 Defining the Loss Function

Next, we will look at the loss function used for style transfer. The loss function includes the content loss, style loss, and total variation loss.

Content Loss

Similar to the loss function used in linear regression, content loss uses a square error function to measure the difference in content features between the composite image and content image. The two inputs of the square error function are both content layer outputs obtained from the extract_features function.

```
def content_loss(Y_hat, Y):
    return np.square(Y_hat - Y).mean()
```

Style Loss

Style loss, similar to content loss, uses a square error function to measure the difference in style between the composite image and style image. To express the styles output by the style layers, we first use the extract_features function to compute the style layer output. Assuming that the output has 1 example, c channels, and a height and width of h and w, we can transform the output into the matrix $\mathbf{X}$, which has c rows and $h \cdot w$ columns. You can think of matrix $\mathbf{X}$ as the combination of the c vectors $\mathbf{x}_1, \ldots, \mathbf{x}_c$, which have a length of hw. Here, the vector $\mathbf{x}_i$ represents the style feature of channel i. In the Gram matrix of these vectors $\mathbf{X}\mathbf{X}^\top \in \mathbb{R}^{c \times c}$, element x_{ij} in row i column j is the inner product of vectors $\mathbf{x}_i$ and $\mathbf{x}_j$. It represents the correlation of the style features of channels i and j. We use this type of Gram matrix to represent the style output by the style layers. You must note that, when the $h \cdot w$ value is large, this often leads to large values in the Gram matrix. In addition, the height and width of the Gram matrix are both the number of channels c. To ensure that the style loss is not affected by the size of these values, we define the gram function below to divide the Gram matrix by the number of its elements, i.e., $c \cdot h \cdot w$.

```
def gram(X):
    num_channels, n = X.shape[1], X.size // X.shape[1]
    X = X.reshape(num_channels, n)
    return np.dot(X, X.T) / (num_channels * n)
```

Naturally, the two Gram matrix inputs of the square error function for style loss are taken from the composite image and style image style layer outputs. Here, we assume that the Gram matrix of the style image, gram_Y, has been computed in advance.

```
def style_loss(Y_hat, gram_Y):
    return np.square(gram(Y_hat) - gram_Y).mean()
```

Total Variance Loss

Sometimes, the composite images we learn have a lot of high-frequency noise, particularly bright or dark pixels. One common noise reduction method is total variation denoising. We assume that $x_{i,j}$ represents the pixel value at the coordinate (i, j), so the total variance loss is:

$$\sum_{i,j} \left|x_{i,j} - x_{i+1,j}\right| + \left|x_{i,j} - x_{i,j+1}\right|. \tag{13.12.1}$$

We try to make the values of neighboring pixels as similar as possible.

```
def tv_loss(Y_hat):
    return 0.5 * (np.abs(Y_hat[:, :, 1:, :] - Y_hat[:, :, :-1, :]).mean() +
                  np.abs(Y_hat[:, :, :, 1:] - Y_hat[:, :, :, :-1]).mean())
```

The Loss Function

The loss function for style transfer is the weighted sum of the content loss, style loss, and total variance loss. By adjusting these weight hyper-parameters, we can balance the retained content, transferred style, and noise reduction in the composite image according to their relative importance.

```
content_weight, style_weight, tv_weight = 1, 1e3, 10

def compute_loss(X, contents_Y_hat, styles_Y_hat, contents_Y, styles_Y_gram):
    # Calculate the content, style, and total variance losses respectively
    contents_l = [content_loss(Y_hat, Y) * content_weight for Y_hat, Y in zip(
        contents_Y_hat, contents_Y)]
    styles_l = [style_loss(Y_hat, Y) * style_weight for Y_hat, Y in zip(
        styles_Y_hat, styles_Y_gram)]
    tv_l = tv_loss(X) * tv_weight
    # Add up all the losses
    l = sum(styles_l + contents_l + [tv_l])
    return contents_l, styles_l, tv_l, l
```

13.12.6 Creating and Initializing the Composite Image

In style transfer, the composite image is the only variable that needs to be updated. Therefore, we can define a simple model, `GeneratedImage`, and treat the composite image as a model parameter. In the model, forward computation only returns the model parameter.

```
class GeneratedImage(nn.Block):
    def __init__(self, img_shape, **kwargs):
        super(GeneratedImage, self).__init__(**kwargs)
        self.weight = self.params.get('weight', shape=img_shape)
```

(continues on next page)

(continued from previous page)

```
    def forward(self):
        return self.weight.data()
```

Next, we define the get_inits function. This function creates a composite image model instance and initializes it to the image X. The Gram matrix for the various style layers of the style image, styles_Y_gram, is computed prior to training.

```
def get_inits(X, ctx, lr, styles_Y):
    gen_img = GeneratedImage(X.shape)
    gen_img.initialize(init.Constant(X), ctx=ctx, force_reinit=True)
    trainer = gluon.Trainer(gen_img.collect_params(), 'adam',
                            {'learning_rate': lr})
    styles_Y_gram = [gram(Y) for Y in styles_Y]
    return gen_img(), styles_Y_gram, trainer
```

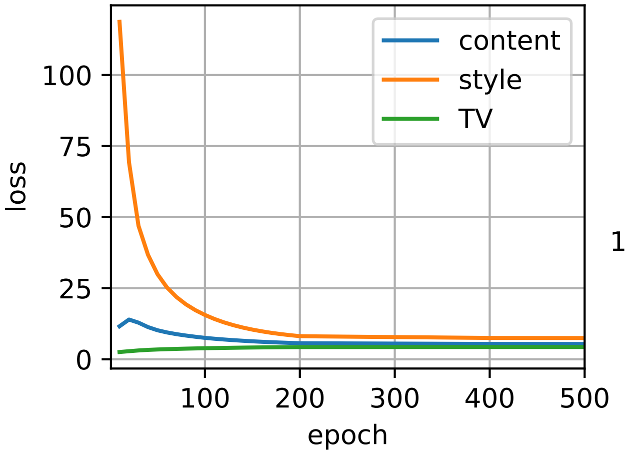
13.12.7 Training

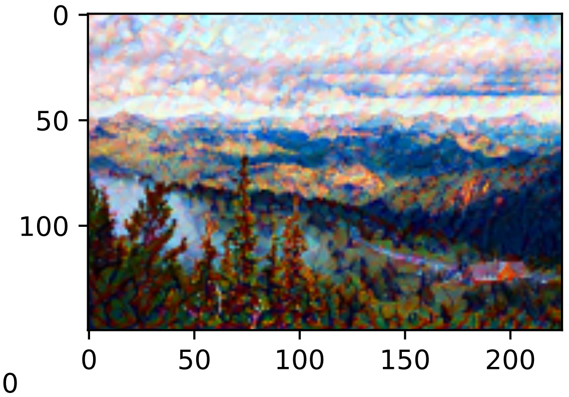
During model training, we constantly extract the content and style features of the composite image and calculate the loss function. Recall our discussion of how synchronization functions force the front end to wait for computation results in Section 12.2. Because we only call the asscalar synchronization function every 50 epochs, the process may occupy a great deal of memory. Therefore, we call the waitall synchronization function during every epoch.

```
def train(X, contents_Y, styles_Y, ctx, lr, num_epochs, lr_decay_epoch):
    X, styles_Y_gram, trainer = get_inits(X, ctx, lr, styles_Y)
    animator = d2l.Animator(xlabel='epoch', ylabel='loss',
                            xlim=[1, num_epochs],
                            legend=['content', 'style', 'TV'],
                            ncols=2, figsize=(7, 2.5))
    for epoch in range(1, num_epochs+1):
        with autograd.record():
            contents_Y_hat, styles_Y_hat = extract_features(
                X, content_layers, style_layers)
            contents_l, styles_l, tv_l, l = compute_loss(
                X, contents_Y_hat, styles_Y_hat, contents_Y, styles_Y_gram)
        l.backward()
        trainer.step(1)
        npx.waitall()
        if epoch % lr_decay_epoch == 0:
            trainer.set_learning_rate(trainer.learning_rate * 0.1)
        if epoch % 10 == 0:
            animator.axes[1].imshow(postprocess(X).asnumpy())
            animator.add(epoch, [float(sum(contents_l)),
                                 float(sum(styles_l)),
                                 float(tv_l)])
    return X
```

Next, we start to train the model. First, we set the height and width of the content and style images to 150 by 225 pixels. We use the content image to initialize the composite image.

```
ctx, image_shape = d2l.try_gpu(), (225, 150)
net.collect_params().reset_ctx(ctx)
content_X, contents_Y = get_contents(image_shape, ctx)
_, styles_Y = get_styles(image_shape, ctx)
output = train(content_X, contents_Y, styles_Y, ctx, 0.01, 500, 200)
```

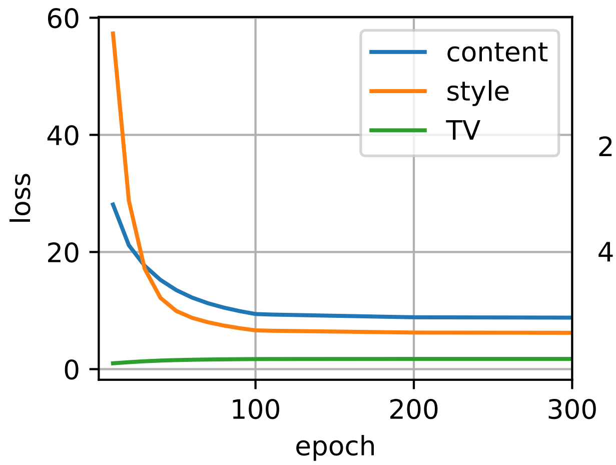


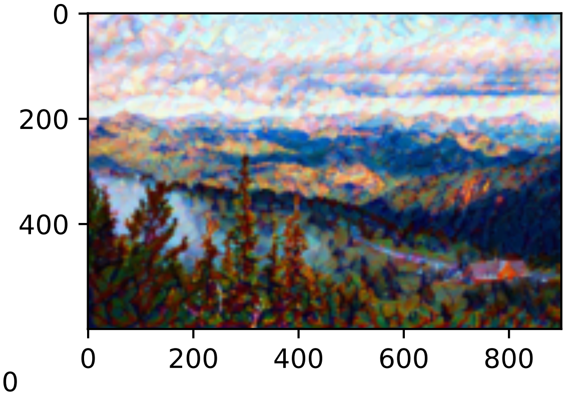


As you can see, the composite image retains the scenery and objects of the content image, while introducing the color of the style image. Because the image is relatively small, the details are a bit fuzzy.

To obtain a clearer composite image, we train the model using a larger image size: 900×600. We increase the height and width of the image used before by a factor of four and initialize a larger composite image.

```
image_shape = (900, 600)
_, content_Y = get_contents(image_shape, ctx)
_, style_Y = get_styles(image_shape, ctx)
X = preprocess(postprocess(output) * 255, image_shape)
output = train(X, content_Y, style_Y, ctx, 0.01, 300, 100)
d2l.plt.imsave('../img/neural-style.png', postprocess(output).asnumpy())
```





As you can see, each epoch takes more time due to the larger image size. As shown in Fig. 13.12.3, the composite image produced retains more detail due to its larger size. The composite image not only has large blocks of color like the style image, but these blocks even have the subtle texture of brush strokes.

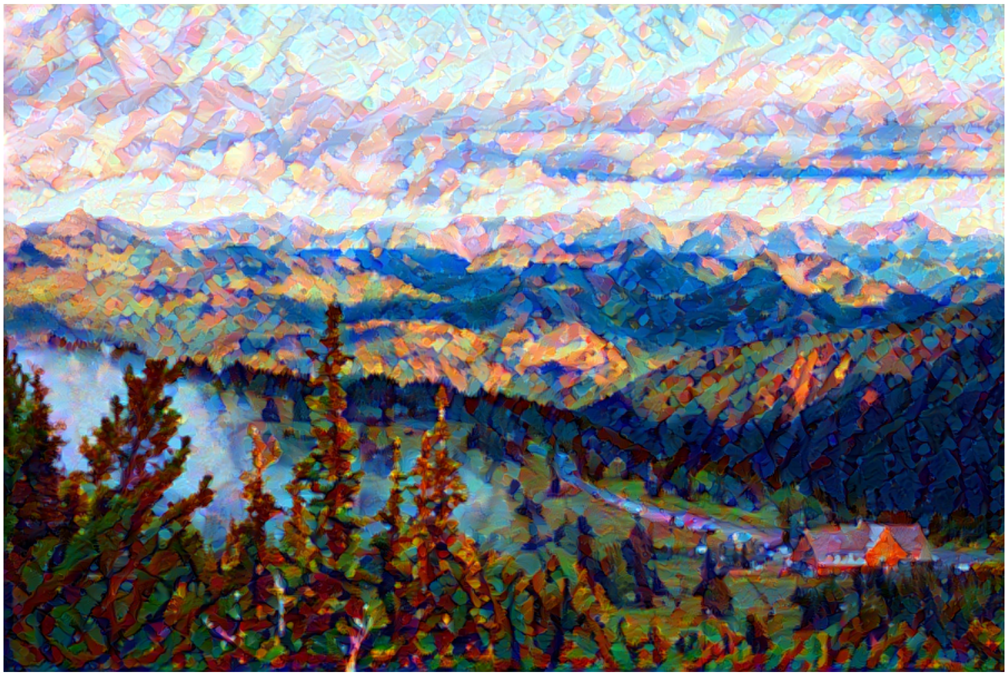


Fig. 13.12.3: 900×600 composite image.

Summary

- The loss functions used in style transfer generally have three parts: 1. Content loss is used to make the composite image approximate the content image as regards content features. 2. Style loss is used to make the composite image approximate the style image in terms of style features. 3. Total variation loss helps reduce the noise in the composite image.
- We can use a pre-trained CNN to extract image features and minimize the loss function to continuously update the composite image.
- We use a Gram matrix to represent the style output by the style layers.

Exercises

1. How does the output change when you select different content and style layers?
2. Adjust the weight hyper-parameters in the loss function. Does the output retain more content or have less noise?
3. Use different content and style images. Can you create more interesting composite images?

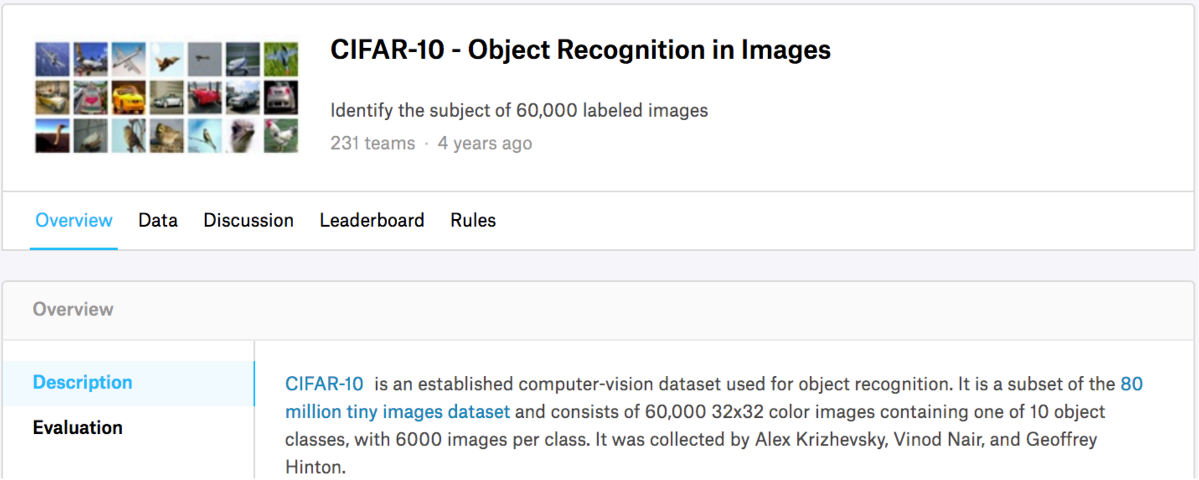
13.13 Image Classification (CIFAR-10) on Kaggle

So far, we have been using Gluon's `data` package to directly obtain image datasets in the `ndarray` format. In practice, however, image datasets often exist in the format of image files. In this section, we will start with the original image files and organize, read, and convert the files to the `ndarray` format step by step.

We performed an experiment on the CIFAR-10 dataset in Section 13.1. This is an important data set in the computer vision field. Now, we will apply the knowledge we learned in the previous sections in order to participate in the Kaggle competition, which addresses CIFAR-10 image classification problems. The competition's web address is

> https://www.kaggle.com/c/cifar-10

Fig. 13.13.1 shows the information on the competition's webpage. In order to submit the results, please register an account on the Kaggle website first.


CIFAR-10 - Object Recognition in Images

Identify the subject of 60,000 labeled images

231 teams · 4 years ago

CIFAR-10 is an established computer-vision dataset used for object recognition. It is a subset of the 80 million tiny images dataset and consists of 60,000 32x32 color images containing one of 10 object classes, with 6000 images per class. It was collected by Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton.

Fig. 13.13.1: CIFAR-10 image classification competition webpage information. The dataset for the competition can be accessed by clicking the "Data" tab.

First, import the packages or modules required for the competition.

```python
import d2l
from mxnet import autograd, gluon, init, npx
from mxnet.gluon import nn
import os
import pandas as pd
import shutil
import time

npx.set_np()
```

13.13.1 Obtaining and Organizing the Dataset

The competition data is divided into a training set and testing set. The training set contains $50,000$ images. The testing set contains $300,000$ images, of which $10,000$ images are used for scoring, while the other $290,000$ non-scoring images are included to prevent the manual labeling of the testing set and the submission of labeling results. The image formats in both datasets are PNG, with heights and widths of 32 pixels and three color channels (RGB). The images cover 10$ categories: planes, cars, birds, cats, deer, dogs, frogs, horses, boats, and trucks. The upper-left corner of Figure 9.16 shows some images of planes, cars, and birds in the dataset.

Downloading the Dataset

After logging in to Kaggle, we can click on the "Data" tab on the CIFAR-10 image classification competition webpage shown in Figure 9.16 and download the training dataset "train.7z", the testing dataset "test.7z", and the training dataset labels "trainlabels.csv".

Unzipping the Dataset

The training dataset "train.7z" and the test dataset "test.7z" need to be unzipped after downloading. After unzipping the datasets, store the training dataset, test dataset, and training dataset labels in the following respective paths:

- ../data/kaggle_cifar10/train/[1-50000].png
- ../data/kaggle_cifar10/test/[1-300000].png
- ../data/kaggle_cifar10/trainLabels.csv

To make it easier to get started, we provide a small-scale sample of the dataset mentioned above. "train_tiny.zip" contains 100 training examples, while "test_tiny.zip" contains only one test example. Their unzipped folder names are "train_tiny" and "test_tiny", respectively. In addition, unzip the zip file of the training dataset labels to obtain the file "trainlabels.csv". If you are going to use the full dataset of the Kaggle competition, you will also need to change the following `demo` variable to `False`.

```python
# If you use the full dataset downloaded for the Kaggle competition, change
# the demo variable to False
demo = True
if demo:
    import zipfile
    for f in ['train_tiny.zip', 'test_tiny.zip', 'trainLabels.csv.zip']:
        with zipfile.ZipFile('../data/kaggle_cifar10/' + f, 'r') as z:
            z.extractall('../data/kaggle_cifar10/')
```

Organizing the Dataset

We need to organize datasets to facilitate model training and testing. The following read_label_file function will be used to read the label file for the training dataset. The parameter valid_ratio in this function is the ratio of the number of examples in the validation set to the number of examples in the original training set.

```
def read_label_file(data_dir, label_file, train_dir, valid_ratio):
    with open(os.path.join(data_dir, label_file), 'r') as f:
        # Skip the file header line (column name)
        lines = f.readlines()[1:]
        tokens = [l.rstrip().split(',') for l in lines]
        idx_label = dict(((int(idx), label) for idx, label in tokens))
    labels = set(idx_label.values())
    n_train_valid = len(os.listdir(os.path.join(data_dir, train_dir)))
    n_train = int(n_train_valid * (1 - valid_ratio))
    assert 0 < n_train < n_train_valid
    return n_train // len(labels), idx_label
```

Below we define a helper function to create a path only if the path does not already exist.

```
# save to the d2l package.
def mkdir_if_not_exist(path):
    if not os.path.exists(os.path.join(*path)):
        os.makedirs(os.path.join(*path))
```

Next, we define the reorg_train_valid function to segment the validation set from the original training set. Here, we use valid_ratio=0.1 as an example. Since the original training set has $50,000$ images, there will be $45,000$ images used for training and stored in the path "input_dir/train" when tuning hyper-parameters, while the other $5,000$ images will be stored as validation set in the path "input_dir/valid". After organizing the data, images of the same type will be placed under the same folder so that we can read them later.

```
def reorg_train_valid(data_dir, train_dir, input_dir, n_train_per_label,
                      idx_label):
    label_count = {}
    for train_file in os.listdir(os.path.join(data_dir, train_dir)):
        idx = int(train_file.split('.')[0])
        label = idx_label[idx]
        mkdir_if_not_exist([data_dir, input_dir, 'train_valid', label])
        shutil.copy(os.path.join(data_dir, train_dir, train_file),
                    os.path.join(data_dir, input_dir, 'train_valid', label))
        if label not in label_count or label_count[label] < n_train_per_label:
            mkdir_if_not_exist([data_dir, input_dir, 'train', label])
            shutil.copy(os.path.join(data_dir, train_dir, train_file),
                        os.path.join(data_dir, input_dir, 'train', label))
            label_count[label] = label_count.get(label, 0) + 1
        else:
            mkdir_if_not_exist([data_dir, input_dir, 'valid', label])
            shutil.copy(os.path.join(data_dir, train_dir, train_file),
                        os.path.join(data_dir, input_dir, 'valid', label))
```

The reorg_test function below is used to organize the testing set to facilitate the reading during prediction.

```
def reorg_test(data_dir, test_dir, input_dir):
    mkdir_if_not_exist([data_dir, input_dir, 'test', 'unknown'])
    for test_file in os.listdir(os.path.join(data_dir, test_dir)):
        shutil.copy(os.path.join(data_dir, test_dir, test_file),
                    os.path.join(data_dir, input_dir, 'test', 'unknown'))
```

Finally, we use a function to call the previously defined reorg_test, reorg_train_valid, and reorg_test functions.

```
def reorg_cifar10_data(data_dir, label_file, train_dir, test_dir, input_dir,
                       valid_ratio):
    n_train_per_label, idx_label = read_label_file(data_dir, label_file,
                                                    train_dir, valid_ratio)
    reorg_train_valid(data_dir, train_dir, input_dir, n_train_per_label,
                      idx_label)
    reorg_test(data_dir, test_dir, input_dir)
```

We use only 100 training example and one test example here. The folder names for the training and testing datasets are "train_tiny" and "test_tiny", respectively. Accordingly, we only set the batch size to 1. During actual training and testing, the complete dataset of the Kaggle competition should be used and batch_size should be set to a larger integer, such as 128. We use 10% of the training examples as the validation set for tuning hyper-parameters.

```
if demo:
    # Note: Here, we use small training sets and small testing sets and the
    # batch size should be set smaller. When using the complete dataset for
    # the Kaggle competition, the batch size can be set to a large integer
    train_dir, test_dir, batch_size = 'train_tiny', 'test_tiny', 1
else:
    train_dir, test_dir, batch_size = 'train', 'test', 128
data_dir, label_file = '../data/kaggle_cifar10', 'trainLabels.csv'
input_dir, valid_ratio = 'train_valid_test', 0.1
reorg_cifar10_data(data_dir, label_file, train_dir, test_dir, input_dir,
                   valid_ratio)
```

13.13.2 Image Augmentation

To cope with overfitting, we use image augmentation. For example, by adding transforms.RandomFlipLeftRight(), the images can be flipped at random. We can also perform normalization for the three RGB channels of color images using transforms.Normalize(). Below, we list some of these operations that you can choose to use or modify depending on requirements.

```
transform_train = gluon.data.vision.transforms.Compose([
    # Magnify the image to a square of 40 pixels in both height and width
    gluon.data.vision.transforms.Resize(40),
    # Randomly crop a square image of 40 pixels in both height and width to
    # produce a small square of 0.64 to 1 times the area of the original
    # image, and then shrink it to a square of 32 pixels in both height and
    # width
    gluon.data.vision.transforms.RandomResizedCrop(32, scale=(0.64, 1.0),
                                                    ratio=(1.0, 1.0)),
    gluon.data.vision.transforms.RandomFlipLeftRight(),
```

(continues on next page)

(continued from previous page)

```
    gluon.data.vision.transforms.ToTensor(),
    # Normalize each channel of the image
    gluon.data.vision.transforms.Normalize([0.4914, 0.4822, 0.4465],
                                           [0.2023, 0.1994, 0.2010])])
```

In order to ensure the certainty of the output during testing, we only perform normalization on the image.

```
transform_test = gluon.data.vision.transforms.Compose([
    gluon.data.vision.transforms.ToTensor(),
    gluon.data.vision.transforms.Normalize([0.4914, 0.4822, 0.4465],
                                           [0.2023, 0.1994, 0.2010])])
```

13.13.3 Reading the Dataset

Next, we can create the `ImageFolderDataset` instance to read the organized dataset containing the original image files, where each data instance includes the image and label.

```
# Read the original image file. Flag=1 indicates that the input image has
# three channels (color)
train_ds = gluon.data.vision.ImageFolderDataset(
    os.path.join(data_dir, input_dir, 'train'), flag=1)
valid_ds = gluon.data.vision.ImageFolderDataset(
    os.path.join(data_dir, input_dir, 'valid'), flag=1)
train_valid_ds = gluon.data.vision.ImageFolderDataset(
    os.path.join(data_dir, input_dir, 'train_valid'), flag=1)
test_ds = gluon.data.vision.ImageFolderDataset(
    os.path.join(data_dir, input_dir, 'test'), flag=1)
```

We specify the defined image augmentation operation in `DataLoader`. During training, we only use the validation set to evaluate the model, so we need to ensure the certainty of the output. During prediction, we will train the model on the combined training set and validation set to make full use of all labelled data.

```
train_iter = gluon.data.DataLoader(train_ds.transform_first(transform_train),
                                   batch_size, shuffle=True,
                                   last_batch='keep')
valid_iter = gluon.data.DataLoader(valid_ds.transform_first(transform_test),
                                   batch_size, shuffle=True,
                                   last_batch='keep')
train_valid_iter = gluon.data.DataLoader(train_valid_ds.transform_first(
    transform_train), batch_size, shuffle=True, last_batch='keep')
test_iter = gluon.data.DataLoader(test_ds.transform_first(transform_test),
                                  batch_size, shuffle=False,
                                  last_batch='keep')
```

13.13.4 Defining the Model

Here, we build the residual blocks based on the HybridBlock class, which is slightly different than the implementation described in Section 7.6. This is done to improve execution efficiency.

```
class Residual(nn.HybridBlock):
    def __init__(self, num_channels, use_1x1conv=False, strides=1, **kwargs):
        super(Residual, self).__init__(**kwargs)
        self.conv1 = nn.Conv2D(num_channels, kernel_size=3, padding=1,
                               strides=strides)
        self.conv2 = nn.Conv2D(num_channels, kernel_size=3, padding=1)
        if use_1x1conv:
            self.conv3 = nn.Conv2D(num_channels, kernel_size=1,
                                   strides=strides)
        else:
            self.conv3 = None
        self.bn1 = nn.BatchNorm()
        self.bn2 = nn.BatchNorm()

    def hybrid_forward(self, F, X):
        Y = F.npx.relu(self.bn1(self.conv1(X)))
        Y = self.bn2(self.conv2(Y))
        if self.conv3:
            X = self.conv3(X)
        return F.npx.relu(Y + X)
```

Next, we define the ResNet-18 model.

```
def resnet18(num_classes):
    net = nn.HybridSequential()
    net.add(nn.Conv2D(64, kernel_size=3, strides=1, padding=1),
            nn.BatchNorm(), nn.Activation('relu'))

    def resnet_block(num_channels, num_residuals, first_block=False):
        blk = nn.HybridSequential()
        for i in range(num_residuals):
            if i == 0 and not first_block:
                blk.add(Residual(num_channels, use_1x1conv=True, strides=2))
            else:
                blk.add(Residual(num_channels))
        return blk

    net.add(resnet_block(64, 2, first_block=True),
            resnet_block(128, 2),
            resnet_block(256, 2),
            resnet_block(512, 2))
    net.add(nn.GlobalAvgPool2D(), nn.Dense(num_classes))
    return net
```

The CIFAR-10 image classification challenge uses 10 categories. We will perform Xavier random initialization on the model before training begins.

```
def get_net(ctx):
    num_classes = 10
    net = resnet18(num_classes)
```

(continues on next page)

(continued from previous page)

```
    net.initialize(ctx=ctx, init=init.Xavier())
    return net

loss = gluon.loss.SoftmaxCrossEntropyLoss()
```

13.13.5 Defining the Training Functions

We will select the model and tune hyper-parameters according to the model's performance on the validation set. Next, we define the model training function `train`. We record the training time of each epoch, which helps us compare the time costs of different models.

```
def train(net, train_iter, valid_iter, num_epochs, lr, wd, ctx, lr_period,
          lr_decay):
    trainer = gluon.Trainer(net.collect_params(), 'sgd',
                            {'learning_rate': lr, 'momentum': 0.9, 'wd': wd})
    for epoch in range(num_epochs):
        train_l_sum, train_acc_sum, n, start = 0.0, 0.0, 0, time.time()
        if epoch > 0 and epoch % lr_period == 0:
            trainer.set_learning_rate(trainer.learning_rate * lr_decay)
        for X, y in train_iter:
            y = y.astype('float32').as_in_context(ctx)
            with autograd.record():
                y_hat = net(X.as_in_context(ctx))
                l = loss(y_hat, y).sum()
            l.backward()
            trainer.step(batch_size)
            train_l_sum += float(l)
            train_acc_sum += float((y_hat.argmax(axis=1) == y).sum())
            n += y.size
        time_s = "time %.2f sec" % (time.time() - start)
        if valid_iter is not None:
            valid_acc = d2l.evaluate_accuracy_gpu(net, valid_iter)
            epoch_s = ("epoch %d, loss %f, train acc %f, valid acc %f, "
                       % (epoch + 1, train_l_sum / n, train_acc_sum / n,
                          valid_acc))
        else:
            epoch_s = ("epoch %d, loss %f, train acc %f, " %
                       (epoch + 1, train_l_sum / n, train_acc_sum / n))
        print(epoch_s + time_s + ', lr ' + str(trainer.learning_rate))
```

13.13.6 Training and Validating the Model

Now, we can train and validate the model. The following hyper-parameters can be tuned. For example, we can increase the number of epochs. Because `lr_period` and `lr_decay` are set to 80 and 0.1 respectively, the learning rate of the optimization algorithm will be multiplied by 0.1 after every 80 epochs. For simplicity, we only train one epoch here.

```
ctx, num_epochs, lr, wd = d2l.try_gpu(), 1, 0.1, 5e-4
lr_period, lr_decay, net = 80, 0.1, get_net(ctx)
net.hybridize()
```

(continues on next page)

(continued from previous page)

```
train(net, train_iter, valid_iter, num_epochs, lr, wd, ctx, lr_period,
      lr_decay)
```

```
epoch 1, loss 25.188189, train acc 0.077778, valid acc 0.200000, time 1.41 sec, lr 0.1
```

13.13.7 Classifying the Testing Set and Submitting Results on Kaggle

After obtaining a satisfactory model design and hyper-parameters, we use all training datasets (including validation sets) to retrain the model and classify the testing set.

```
net, preds = get_net(ctx), []
net.hybridize()
train(net, train_valid_iter, None, num_epochs, lr, wd, ctx, lr_period,
      lr_decay)

for X, _ in test_iter:
    y_hat = net(X.as_in_context(ctx))
    preds.extend(y_hat.argmax(axis=1).astype(int).asnumpy())
sorted_ids = list(range(1, len(test_ds) + 1))
sorted_ids.sort(key=lambda x: str(x))
df = pd.DataFrame({'id': sorted_ids, 'label': preds})
df['label'] = df['label'].apply(lambda x: train_valid_ds.synsets[x])
df.to_csv('submission.csv', index=False)
```

```
epoch 1, loss 6.266124, train acc 0.070000, time 1.09 sec, lr 0.1
```

After executing the above code, we will get a "submission.csv" file. The format of this file is consistent with the Kaggle competition requirements. The method for submitting results is similar to method in Section 4.10.

Summary

- We can create an `ImageFolderDataset` instance to read the dataset containing the original image files.
- We can use convolutional neural networks, image augmentation, and hybrid programming to take part in an image classification competition.

Exercises

1. Use the complete CIFAF-10 dataset for the Kaggle competition. Change the `batch_size` and number of epochs `num_epochs` to 128 and 100, respectively. See what accuracy and ranking you can achieve in this competition.
2. What accuracy can you achieve when not using image augmentation?
3. Scan the QR code to access the relevant discussions and exchange ideas about the methods used and the results obtained with the community. Can you come up with any better techniques?

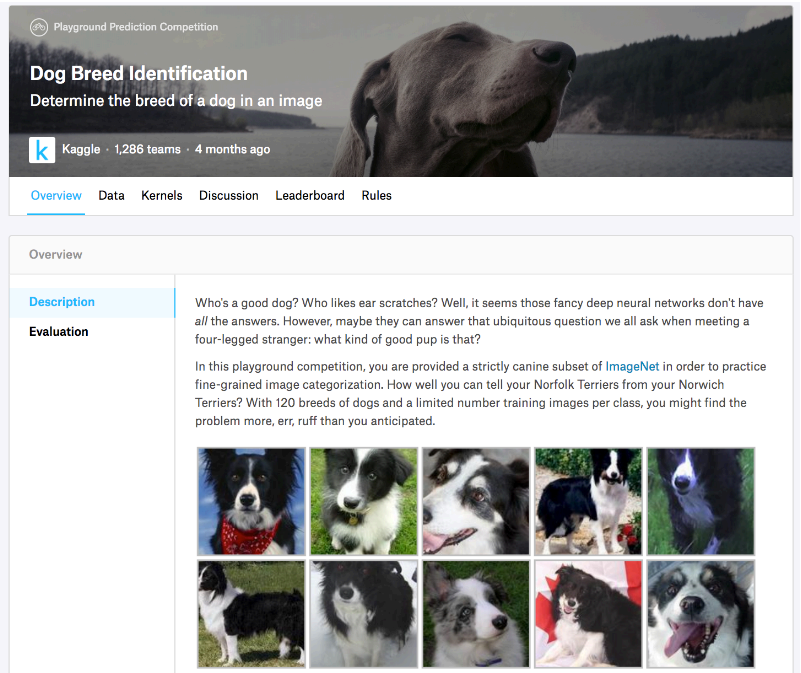
13.14 Dog Breed Identification (ImageNet Dogs) on Kaggle

In this section, we will tackle the dog breed identification challenge in the Kaggle Competition. The competition's web address is

https://www.kaggle.com/c/dog-breed-identification

In this competition, we attempt to identify 120 different breeds of dogs. The dataset used in this competition is actually a subset of the famous ImageNet dataset. Different from the images in the CIFAR-10 dataset used in the previous section, the images in the ImageNet dataset are higher and wider and their dimensions are inconsistent.

Fig. 13.14.1 shows the information on the competition's webpage. In order to submit the results, please register an account on the Kaggle website first.


Playground Prediction Competition
Dog Breed Identification
Determine the breed of a dog in an image
Kaggle · 1,286 teams · 4 months ago
Overview
Data
Kernels
Discussion
Leaderboard
Rules
Overview
Description
Evaluation

Who's a good dog? Who likes ear scratches? Well, it seems those fancy deep neural networks don't have *all* the answers. However, maybe they can answer that ubiquitous question we all ask when meeting a four-legged stranger: what kind of good pup is that?

In this playground competition, you are provided a strictly canine subset of ImageNet in order to practice fine-grained image categorization. How well you can tell your Norfolk Terriers from your Norwich Terriers? With 120 breeds of dogs and a limited number training images per class, you might find the problem more, err, ruff than you anticipated.

Fig. 13.14.1: Dog breed identification competition website. The dataset for the competition can be accessed by clicking the "Data" tab.

First, import the packages or modules required for the competition.

```
import collections
import d2l
import math
from mxnet import autograd, gluon, init, npx
from mxnet.gluon import nn
```

(continues on next page)

(continued from previous page)

```
import os
import shutil
import time
import zipfile

npx.set_np()
```

13.14.1 Obtaining and Organizing the Dataset

The competition data is divided into a training set and testing set. The training set contains $10,222$ images and the testing set contains $10,357$ images. The images in both sets are in JPEG format. These images contain three RGB channels (color) and they have different heights and widths. There are 120 breeds of dogs in the training set, including Labradors, Poodles, Dachshunds, Samoyeds, Huskies, Chihuahuas, and Yorkshire Terriers.

Downloading the Dataset

After logging in to Kaggle, we can click on the "Data" tab on the dog breed identification competition webpage shown in Fig. 13.14.1 and download the training dataset "train.zip", the testing dataset "test.zip", and the training dataset labels "label.csv.zip". After downloading the files, place them in the three paths below:

- ../data/kaggle_dog/train.zip
- ../data/kaggle_dog/test.zip
- ../data/kaggle_dog/labels.csv.zip

To make it easier to get started, we provide a small-scale sample of the dataset mentioned above, "train_valid_test_tiny.zip". If you are going to use the full dataset for the Kaggle competition, you will also need to change the `demo` variable below to `False`.

```
# If you use the full dataset downloaded for the Kaggle competition, change
# the variable below to False
demo = True
data_dir = '../data/kaggle_dog'
if demo:
    zipfiles = ['train_valid_test_tiny.zip']
else:
    zipfiles = ['train.zip', 'test.zip', 'labels.csv.zip']
for f in zipfiles:
    with zipfile.ZipFile(data_dir + '/' + f, 'r') as z:
        z.extractall(data_dir)
```

Organizing the Dataset

Next, we define the reorg_train_valid function to segment the validation set from the original Kaggle competition training set. The parameter valid_ratio in this function is the ratio of the number of examples of each dog breed in the validation set to the number of examples of the breed with the least examples (66) in the original training set. After organizing the data, images of the same breed will be placed in the same folder so that we can read them later.

```python
def reorg_train_valid(data_dir, train_dir, input_dir, valid_ratio, idx_label):
    # The number of examples of the least represented breed in the training
    # set
    min_n_train_per_label = (
        collections.Counter(idx_label.values()).most_common()[:-2:-1][0][1])
    # The number of examples of each breed in the validation set
    n_valid_per_label = math.floor(min_n_train_per_label * valid_ratio)
    label_count = {}
    for train_file in os.listdir(os.path.join(data_dir, train_dir)):
        idx = train_file.split('.')[0]
        label = idx_label[idx]
        d2l.mkdir_if_not_exist([data_dir, input_dir, 'train_valid', label])
        shutil.copy(os.path.join(data_dir, train_dir, train_file),
                    os.path.join(data_dir, input_dir, 'train_valid', label))
        if label not in label_count or label_count[label] < n_valid_per_label:
            d2l.mkdir_if_not_exist([data_dir, input_dir, 'valid', label])
            shutil.copy(os.path.join(data_dir, train_dir, train_file),
                        os.path.join(data_dir, input_dir, 'valid', label))
            label_count[label] = label_count.get(label, 0) + 1
        else:
            d2l.mkdir_if_not_exist([data_dir, input_dir, 'train', label])
            shutil.copy(os.path.join(data_dir, train_dir, train_file),
                        os.path.join(data_dir, input_dir, 'train', label))
```

The reorg_dog_data function below is used to read the training data labels, segment the validation set, and organize the training set.

```python
def reorg_dog_data(data_dir, label_file, train_dir, test_dir, input_dir,
                   valid_ratio):
    # Read the training data labels
    with open(os.path.join(data_dir, label_file), 'r') as f:
        # Skip the file header line (column name)
        lines = f.readlines()[1:]
        tokens = [l.rstrip().split(',') for l in lines]
        idx_label = dict(((idx, label) for idx, label in tokens))
    reorg_train_valid(data_dir, train_dir, input_dir, valid_ratio, idx_label)
    # Organize the training set
    d2l.mkdir_if_not_exist([data_dir, input_dir, 'test', 'unknown'])
    for test_file in os.listdir(os.path.join(data_dir, test_dir)):
        shutil.copy(os.path.join(data_dir, test_dir, test_file),
                    os.path.join(data_dir, input_dir, 'test', 'unknown'))
```

Because we are using a small dataset, we set the batch size to 1. During actual training and testing, we would use the entire Kaggle Competition dataset and call the reorg_dog_data function to organize the dataset. Likewise, we would need to set the batch_size to a larger integer, such as 128.

```
if demo:
    # Note: Here, we use a small dataset and the batch size should be set
    # smaller. When using the complete dataset for the Kaggle competition, we
    # can set the batch size to a larger integer
    input_dir, batch_size = 'train_valid_test_tiny', 1
else:
    label_file, train_dir, test_dir = 'labels.csv', 'train', 'test'
    input_dir, batch_size, valid_ratio = 'train_valid_test', 128, 0.1
    reorg_dog_data(data_dir, label_file, train_dir, test_dir, input_dir,
                   valid_ratio)
```

13.14.2 Image Augmentation

The size of the images in this section are larger than the images in the previous section. Here are some more image augmentation operations that might be useful.

```
transform_train = gluon.data.vision.transforms.Compose([
    # Randomly crop the image to obtain an image with an area of 0.08 to 1 of
    # the original area and height to width ratio between 3/4 and 4/3. Then,
    # scale the image to create a new image with a height and width of 224
    # pixels each
    gluon.data.vision.transforms.RandomResizedCrop(224, scale=(0.08, 1.0),
                                                   ratio=(3.0/4.0, 4.0/3.0)),
    gluon.data.vision.transforms.RandomFlipLeftRight(),
    # Randomly change the brightness, contrast, and saturation
    gluon.data.vision.transforms.RandomColorJitter(brightness=0.4,
                                                   contrast=0.4,
                                                   saturation=0.4),
    # Add random noise
    gluon.data.vision.transforms.RandomLighting(0.1),
    gluon.data.vision.transforms.ToTensor(),
    # Standardize each channel of the image
    gluon.data.vision.transforms.Normalize([0.485, 0.456, 0.406],
                                           [0.229, 0.224, 0.225])])
```

During testing, we only use definite image preprocessing operations.

```
transform_test = gluon.data.vision.transforms.Compose([
    gluon.data.vision.transforms.Resize(256),
    # Crop a square of 224 by 224 from the center of the image
    gluon.data.vision.transforms.CenterCrop(224),
    gluon.data.vision.transforms.ToTensor(),
    gluon.data.vision.transforms.Normalize([0.485, 0.456, 0.406],
                                           [0.229, 0.224, 0.225])])
```

13.14.3 Reading the Dataset

As in the previous section, we can create an ImageFolderDataset instance to read the dataset containing the original image files.

```
train_ds = gluon.data.vision.ImageFolderDataset(
    os.path.join(data_dir, input_dir, 'train'), flag=1)
valid_ds = gluon.data.vision.ImageFolderDataset(
    os.path.join(data_dir, input_dir, 'valid'), flag=1)
train_valid_ds = gluon.data.vision.ImageFolderDataset(
    os.path.join(data_dir, input_dir, 'train_valid'), flag=1)
test_ds = gluon.data.vision.ImageFolderDataset(
    os.path.join(data_dir, input_dir, 'test'), flag=1)
```

Here, we create a DataLoader instance, just like in the previous section.

```
train_iter = gluon.data.DataLoader(train_ds.transform_first(transform_train),
                                   batch_size, shuffle=True,
                                   last_batch='keep')
valid_iter = gluon.data.DataLoader(valid_ds.transform_first(transform_test),
                                   batch_size, shuffle=True,
                                   last_batch='keep')
train_valid_iter = gluon.data.DataLoader(train_valid_ds.transform_first(
    transform_train), batch_size, shuffle=True, last_batch='keep')
test_iter = gluon.data.DataLoader(test_ds.transform_first(transform_test),
                                  batch_size, shuffle=False,
                                  last_batch='keep')
```

13.14.4 Defining the Model

The dataset for this competition is a subset of the ImageNet data set. Therefore, we can use the approach discussed in Section 13.2 to select a model pre-trained on the entire ImageNet dataset and use it to extract image features to be input in the custom small-scale output network. Gluon provides a wide range of pre-trained models. Here, we will use the pre-trained ResNet-34 model. Because the competition dataset is a subset of the pre-training dataset, we simply reuse the input of the pre-trained model's output layer, i.e., the extracted features. Then, we can replace the original output layer with a small custom output network that can be trained, such as two fully connected layers in a series. Different from the experiment in Section 13.2, here, we do not retrain the pre-trained model used for feature extraction. This reduces the training time and the memory required to store model parameter gradients.

You must note that, during image augmentation, we use the mean values and standard deviations of the three RGB channels for the entire ImageNet dataset for normalization. This is consistent with the normalization of the pre-trained model.

```
def get_net(ctx):
    finetune_net = gluon.model_zoo.vision.resnet34_v2(pretrained=True)
    # Define a new output network
    finetune_net.output_new = nn.HybridSequential(prefix='')
    finetune_net.output_new.add(nn.Dense(256, activation='relu'))
    # There are 120 output categories
    finetune_net.output_new.add(nn.Dense(120))
```

(continues on next page)

(continued from previous page)

```
    # Initialize the output network
    finetune_net.output_new.initialize(init.Xavier(), ctx=ctx)
    # Distribute the model parameters to the CPUs or GPUs used for computation
    finetune_net.collect_params().reset_ctx(ctx)
    return finetune_net
```

When calculating the loss, we first use the member variable features to obtain the input of the pre-trained model's output layer, i.e., the extracted feature. Then, we use this feature as the input for our small custom output network and compute the output.

```
loss = gluon.loss.SoftmaxCrossEntropyLoss()

def evaluate_loss(data_iter, net, ctx):
    l_sum, n = 0.0, 0
    for X, y in data_iter:
        y = y.as_in_context(ctx)
        output_features = net.features(X.as_in_context(ctx))
        outputs = net.output_new(output_features)
        l_sum += float(loss(outputs, y).sum())
        n += y.size
    return l_sum / n
```

13.14.5 Defining the Training Functions

We will select the model and tune hyper-parameters according to the model's performance on the validation set. The model training function train only trains the small custom output network.

```
def train(net, train_iter, valid_iter, num_epochs, lr, wd, ctx, lr_period,
          lr_decay):
    # Only train the small custom output network
    trainer = gluon.Trainer(net.output_new.collect_params(), 'sgd',
                            {'learning_rate': lr, 'momentum': 0.9, 'wd': wd})
    for epoch in range(num_epochs):
        train_l_sum, n, start = 0.0, 0, time.time()
        if epoch > 0 and epoch % lr_period == 0:
            trainer.set_learning_rate(trainer.learning_rate * lr_decay)
        for X, y in train_iter:
            y = y.as_in_context(ctx)
            output_features = net.features(X.as_in_context(ctx))
            with autograd.record():
                outputs = net.output_new(output_features)
                l = loss(outputs, y).sum()
            l.backward()
            trainer.step(batch_size)
            train_l_sum += float(l)
            n += y.size
        time_s = "time %.2f sec" % (time.time() - start)
        if valid_iter is not None:
            valid_loss = evaluate_loss(valid_iter, net, ctx)
            epoch_s = ("epoch %d, train loss %f, valid loss %f, "
                       % (epoch + 1, train_l_sum / n, valid_loss))
        else:
```

(continues on next page)

(continued from previous page)

```
            epoch_s = ("epoch %d, train loss %f, "
                       % (epoch + 1, train_l_sum / n))
        print(epoch_s + time_s + ', lr ' + str(trainer.learning_rate))
```

13.14.6 Training and Validating the Model

Now, we can train and validate the model. The following hyper-parameters can be tuned. For example, we can increase the number of epochs. Because `lr_period` and `lr_decay` are set to 10 and 0.1 respectively, the learning rate of the optimization algorithm will be multiplied by 0.1 after every 10 epochs.

```
ctx, num_epochs, lr, wd = d2l.try_gpu(), 1, 0.01, 1e-4
lr_period, lr_decay, net = 10, 0.1, get_net(ctx)
net.hybridize()
train(net, train_iter, valid_iter, num_epochs, lr, wd, ctx, lr_period,
      lr_decay)
```

```
epoch 1, train loss 5.227054, valid loss 4.753427, time 1.46 sec, lr 0.01
```

13.14.7 Classifying the Testing Set and Submit Results on Kaggle

After obtaining a satisfactory model design and hyper-parameters, we use all training datasets (including validation sets) to retrain the model and then classify the testing set. Note that predictions are made by the output network we just trained.

```
net = get_net(ctx)
net.hybridize()
train(net, train_valid_iter, None, num_epochs, lr, wd, ctx, lr_period,
      lr_decay)

preds = []
for data, label in test_iter:
    output_features = net.features(data.as_in_context(ctx))
    output = npx.softmax(net.output_new(output_features))
    preds.extend(output.asnumpy())
ids = sorted(os.listdir(os.path.join(data_dir, input_dir, 'test/unknown')))
with open('submission.csv', 'w') as f:
    f.write('id,' + ','.join(train_valid_ds.synsets) + '\n')
    for i, output in zip(ids, preds):
        f.write(i.split('.')[0] + ',' + ','.join(
            [str(num) for num in output]) + '\n')
```

```
epoch 1, train loss 5.054451, time 2.43 sec, lr 0.01
```

After executing the above code, we will generate a “submission.csv” file. The format of this file is consistent with the Kaggle competition requirements. The method for submitting results is similar to method in Section 4.10.

Summary

- We can use a model pre-trained on the ImageNet dataset to extract features and only train a small custom output network. This will allow us to classify a subset of the ImageNet dataset with lower computing and storage overhead.

Exercises

1. When using the entire Kaggle dataset, what kind of results do you get when you increase the `batch_size` (batch size) and `num_epochs` (number of epochs)?
2. Do you get better results if you use a deeper pre-trained model?
3. Scan the QR code to access the relevant discussions and exchange ideas about the methods used and the results obtained with the community. Can you come up with any better techniques?

14 | Natural Language Processing

Natural language processing is concerned with interactions between computers and humans that use natural language. In practice, it is very common for us to use this technique to process and analyze large amounts of natural language data, like the language models from Chapter 8

In this chapter, we will discuss how to use vectors to represent words and train the word vectors on a corpus. We will also use word vectors pre-trained on a larger corpus to find synonyms and analogies. Then, in the text classification task, we will use word vectors to analyze the emotion of a text and explain the important ideas of timing data classification based on recurrent neural networks and the convolutional neural networks.

14.1 Word Embedding (word2vec)

A natural language is a complex system that we use to express meanings. In this system, words are the basic unit of linguistic meaning. As its name implies, a word vector is a vector used to represent a word. It can also be thought of as the feature vector of a word. The technique of mapping words to vectors of real numbers is also known as word embedding. Over the last few years, word embedding has gradually become basic knowledge in natural language processing.

14.1.1 Why Not Use One-hot Vectors?

We used one-hot vectors to represent words (characters are words) in Section 8.5 . Recall that when we assume the number of different words in a dictionary (the dictionary size) is N, each word can correspond one-to-one with consecutive integers from 0 to $N-1$. These integers that correspond to words are called the indices of the words. We assume that the index of a word is i. In order to get the one-hot vector representation of the word, we create a vector of all 0s with a length of N and set element i to 1. In this way, each word is represented as a vector of length N that can be used directly by the neural network.

Although one-hot word vectors are easy to construct, they are usually not a good choice. One of the major reasons is that the one-hot word vectors cannot accurately express the similarity between different words, such as the cosine similarity that we commonly use. For the vectors $\mathbf{x}, \mathbf{y} \in \mathbb{R}^d$, their cosine similarities are the cosines of the angles between them:

$$\frac{\mathbf{x}^\top \mathbf{y}}{\|\mathbf{x}\| \|\mathbf{y}\|} \in [-1, 1]. \tag{14.1.1}$$

Since the cosine similarity between the one-hot vectors of any two different words is 0, it is difficult to use the one-hot vector to accurately represent the similarity between multiple different words.

Word2vec[215] is a tool that we came up with to solve the problem above. It represents each word with a fixed-length vector and uses these vectors to better indicate the similarity and analogy relationships between different words. The Word2vec tool contains two models: skip-gram (Mikolov et al., 2013b) and continuous bag of words (CBOW) (Mikolov et al., 2013a). Next, we will take a look at the two models and their training methods.

14.1.2 The Skip-Gram Model

The skip-gram model assumes that a word can be used to generate the words that surround it in a text sequence. For example, we assume that the text sequence is "the", "man", "loves", "his", and "son". We use "loves" as the central target word and set the context window size to 2. As shown in Fig. 14.1.1, given the central target word "loves", the skip-gram model is concerned with the conditional probability for generating the context words, "the", "man", "his" and "son", that are within a distance of no more than 2 words, which is

$$P(\text{"the"}, \text{"man"}, \text{"his"}, \text{"son"} \mid \text{"loves"}). \tag{14.1.2}$$

We assume that, given the central target word, the context words are generated independently of each other. In this case, the formula above can be rewritten as

$$P(\text{"the"} \mid \text{"loves"}) \cdot P(\text{"man"} \mid \text{"loves"}) \cdot P(\text{"his"} \mid \text{"loves"}) \cdot P(\text{"son"} \mid \text{"loves"}). \tag{14.1.3}$$

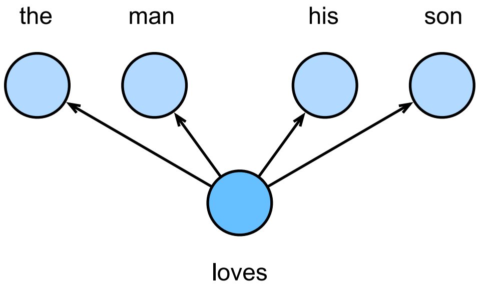


Fig. 14.1.1: The skip-gram model cares about the conditional probability of generating context words for a given central target word.

In the skip-gram model, each word is represented as two d-dimension vectors, which are used to compute the conditional probability. We assume that the word is indexed as i in the dictionary, its vector is represented as $\mathbf{v}_i \in \mathbb{R}^d$ when it is the central target word, and $\mathbf{u}_i \in \mathbb{R}^d$ when it is a context word. Let the central target word w_c and context word w_o be indexed as c and o respectively in the dictionary. The conditional probability of generating the context word for the given central target word can be obtained by performing a softmax operation on the vector inner product:

$$P(w_o \mid w_c) = \frac{\exp(\mathbf{u}_o^\top \mathbf{v}_c)}{\sum_{i \in \mathcal{V}} \exp(\mathbf{u}_i^\top \mathbf{v}_c)}, \tag{14.1.4}$$

where vocabulary index set $\mathcal{V} = \{0, 1, \dots, |\mathcal{V}|-1\}$. Assume that a text sequence of length T is given, where the word at timestep t is denoted as $w^{(t)}$. Assume that context words are independently

[215] https://code.google.com/archive/p/word2vec/

generated given center words. When context window size is m, the likelihood function of the skip-gram model is the joint probability of generating all the context words given any center word

$$\prod_{t=1}^{T} \prod_{-m\leq j\leq m,\ j\neq 0} P(w^{(t+j)} \mid w^{(t)}), \tag{14.1.5}$$

Here, any timestep that is less than 1 or greater than T can be ignored.

Skip-Gram Model Training

The skip-gram model parameters are the central target word vector and context word vector for each individual word. In the training process, we are going to learn the model parameters by maximizing the likelihood function, which is also known as maximum likelihood estimation. This is equivalent to minimizing the following loss function:

$$-\sum_{t=1}^{T} \sum_{-m\leq j\leq m,\ j\neq 0} \log P(w^{(t+j)} \mid w^{(t)}). \tag{14.1.6}$$

If we use the SGD, in each iteration we are going to pick a shorter subsequence through random sampling to compute the loss for that subsequence, and then compute the gradient to update the model parameters. The key of gradient computation is to compute the gradient of the logarithmic conditional probability for the central word vector and the context word vector. By definition, we first have

$$\log P(w_o \mid w_c) = \mathbf{u}_o^\top \mathbf{v}_c - \log\left(\sum_{i\in\mathcal{V}} \text{exp}(\mathbf{u}_i^\top \mathbf{v}_c)\right). \tag{14.1.7}$$

Through differentiation, we can get the gradient $\mathbf{v}_c$ from the formula above.

$$\begin{aligned}\frac{\partial \text{log}\, P(w_o \mid w_c)}{\partial \mathbf{v}_c} &= \mathbf{u}_o - \frac{\sum_{j\in\mathcal{V}} \exp(\mathbf{u}_j^\top \mathbf{v}_c)\mathbf{u}_j}{\sum_{i\in\mathcal{V}} \exp(\mathbf{u}_i^\top \mathbf{v}_c)}\\ &= \mathbf{u}_o - \sum_{j\in\mathcal{V}} \left(\frac{\exp(\mathbf{u}_j^\top \mathbf{v}_c)}{\sum_{i\in\mathcal{V}} \exp(\mathbf{u}_i^\top \mathbf{v}_c)}\right)\mathbf{u}_j\\ &= \mathbf{u}_o - \sum_{j\in\mathcal{V}} P(w_j \mid w_c)\mathbf{u}_j.\end{aligned} \tag{14.1.8}$$

Its computation obtains the conditional probability for all the words in the dictionary given the central target word w_c. We then use the same method to obtain the gradients for other word vectors.

After the training, for any word in the dictionary with index i, we are going to get its two word vector sets $\mathbf{v}_i$ and $\mathbf{u}_i$. In applications of natural language processing (NLP), the central target word vector in the skip-gram model is generally used as the representation vector of a word.

14.1.3 The Continuous Bag of Words (CBOW) Model

The continuous bag of words (CBOW) model is similar to the skip-gram model. The biggest difference is that the CBOW model assumes that the central target word is generated based on the context words before and after it in the text sequence. With the same text sequence "the", "man", "loves", "his" and "son", in which "loves" is the central target word, given a context window size of 2, the CBOW model is concerned with the conditional probability of generating the target word "loves" based on the context words "the", "man", "his" and "son"(as shown in Fig. 14.1.2), such as

$$P(\text{"loves"} \mid \text{"the"}, \text{"man"}, \text{"his"}, \text{"son"}). \tag{14.1.9}$$

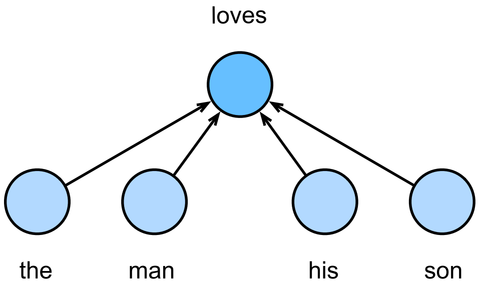


Fig. 14.1.2: The CBOW model cares about the conditional probability of generating the central target word from given context words.

Since there are multiple context words in the CBOW model, we will average their word vectors and then use the same method as the skip-gram model to compute the conditional probability. We assume that $\mathbf{v_i} \in \mathbb{R}^d$ and $\mathbf{u_i} \in \mathbb{R}^d$ are the context word vector and central target word vector of the word with index i in the dictionary (notice that the symbols are opposite to the ones in the skip-gram model). Let central target word w_c be indexed as c, and context words $w_{o_1}, \ldots, w_{o_{2m}}$ be indexed as $o_1, \ldots, o_{2m}$ in the dictionary. Thus, the conditional probability of generating a central target word from the given context word is

$$P(w_c \mid w_{o_1}, \ldots, w_{o_{2m}}) = \frac{\text{exp}\left(\frac{1}{2m}\mathbf{u}_c^\top (\mathbf{v}_{o_1} + \ldots + \mathbf{v}_{o_{2m}})\right)}{\sum_{i \in \mathcal{V}} \text{exp}\left(\frac{1}{2m}\mathbf{u}_i^\top (\mathbf{v}_{o_1} + \ldots + \mathbf{v}_{o_{2m}})\right)}. \tag{14.1.10}$$

For brevity, denote $\mathcal{W}_o = \{w_{o_1}, \ldots, w_{o_{2m}}\}$, and $\bar{\mathbf{v}}_o = \left(\mathbf{v}_{o_1} + \ldots + \mathbf{v}_{o_{2m}}\right)/(2m)$. The equation above can be simplified as

$$P(w_c \mid \mathcal{W}_o) = \frac{\text{exp}\left(\mathbf{u}_c^\top \bar{\mathbf{v}}_o\right)}{\sum_{i \in \mathcal{V}} \text{exp}\left(\mathbf{u}_i^\top \bar{\mathbf{v}}_o\right)}. \tag{14.1.11}$$

Given a text sequence of length T, we assume that the word at timestep t is $w^{(t)}$, and the context window size is m. The likelihood function of the CBOW model is the probability of generating any central target word from the context words.

$$\prod_{t=1}^{T} P(w^{(t)} \mid w^{(t-m)}, \ldots, w^{(t-1)}, w^{(t+1)}, \ldots, w^{(t+m)}). \tag{14.1.12}$$

CBOW Model Training

CBOW model training is quite similar to skip-gram model training. The maximum likelihood estimation of the CBOW model is equivalent to minimizing the loss function.

$$-\sum_{t=1}^{T} \log P(w^{(t)} \mid w^{(t-m)}, \ldots, w^{(t-1)}, w^{(t+1)}, \ldots, w^{(t+m)}). \tag{14.1.13}$$

Notice that

$$\log\ P(w_c \mid \mathcal{W}_o) = \mathbf{u}_c^\top \bar{\mathbf{v}}_o - \log\left(\sum_{i\in\mathcal{V}} \exp\left(\mathbf{u}_i^\top \bar{\mathbf{v}}_o\right)\right). \tag{14.1.14}$$

Through differentiation, we can compute the logarithm of the conditional probability of the gradient of any context word vector $\mathbf{v}_{o_i}(i = 1, \ldots, 2m)$ in the formula above.

$$\frac{\partial \log\ P(w_c \mid \mathcal{W}_o)}{\partial \mathbf{v}_{o_i}} = \frac{1}{2m}\left(\mathbf{u}_c - \sum_{j\in\mathcal{V}} \frac{\exp(\mathbf{u}_j^\top \bar{\mathbf{v}}_o)\mathbf{u}_j}{\sum_{i\in\mathcal{V}} \exp(\mathbf{u}_i^\top \bar{\mathbf{v}}_o)}\right) = \frac{1}{2m}\left(\mathbf{u}_c - \sum_{j\in\mathcal{V}} P(w_j \mid \mathcal{W}_o)\mathbf{u}_j\right). \tag{14.1.15}$$

We then use the same method to obtain the gradients for other word vectors. Unlike the skip-gram model, we usually use the context word vector as the representation vector for a word in the CBOW model.

Summary

- A word vector is a vector used to represent a word. The technique of mapping words to vectors of real numbers is also known as word embedding.
- Word2vec includes both the continuous bag of words (CBOW) and skip-gram models. The skip-gram model assumes that context words are generated based on the central target word. The CBOW model assumes that the central target word is generated based on the context words.

Exercises

1. What is the computational complexity of each gradient? If the dictionary contains a large volume of words, what problems will this cause?
2. There are some fixed phrases in the English language which consist of multiple words, such as “new york”. How can you train their word vectors? Hint: See section 4 in the Word2vec paper[2].
3. Use the skip-gram model as an example to think about the design of a word2vec model. What is the relationship between the inner product of two word vectors and the cosine similarity in the skip-gram model? For a pair of words with close semantical meaning, why it is likely for their word vector cosine similarity to be high?

14.2 Approximate Training for Word2vec

Recall content of the last section. The core feature of the skip-gram model is the use of softmax operations to compute the conditional probability of generating context word w_o based on the given central target word w_c.

$$P(w_o \mid w_c) = \frac{\text{exp}(\mathbf{u}_o^\top \mathbf{v}_c)}{\sum_{i \in \mathcal{V}} \text{exp}(\mathbf{u}_i^\top \mathbf{v}_c)}. \tag{14.2.1}$$

The logarithmic loss corresponding to the conditional probability is given as

$$-\log P(w_o \mid w_c) = -\mathbf{u}_o^\top \mathbf{v}_c + \log \left(\sum_{i \in \mathcal{V}} \text{exp}(\mathbf{u}_i^\top \mathbf{v}_c) \right). \tag{14.2.2}$$

Because the softmax operation has considered that the context word could be any word in the dictionary $\mathcal{V}$, the loss mentioned above actually includes the sum of the number of items in the dictionary size. From the last section, we know that for both the skip-gram model and CBOW model, because they both get the conditional probability using a softmax operation, the gradient computation for each step contains the sum of the number of items in the dictionary size. For larger dictionaries with hundreds of thousands or even millions of words, the overhead for computing each gradient may be too high. In order to reduce such computational complexity, we will introduce two approximate training methods in this section: negative sampling and hierarchical softmax. Since there is no major difference between the skip-gram model and the CBOW model, we will only use the skip-gram model as an example to introduce these two training methods in this section.

14.2.1 Negative Sampling

Negative sampling modifies the original objective function. Given a context window for the central target word w_c, we will treat it as an event for context word w_o to appear in the context window and compute the probability of this event from

$$P(D = 1 \mid w_c, w_o) = \sigma(\mathbf{u}_o^\top \mathbf{v}_c), \tag{14.2.3}$$

Here, the σ function has the same definition as the sigmoid activation function:

$$\sigma(x) = \frac{1}{1 + \text{exp}(-x)}. \tag{14.2.4}$$

We will first consider training the word vector by maximizing the joint probability of all events in the text sequence. Given a text sequence of length T, we assume that the word at timestep t is $w^{(t)}$ and the context window size is m. Now we consider maximizing the joint probability

$$\prod_{t=1}^{T} \prod_{-m \leq j \leq m,\ j \neq 0} P(D = 1 \mid w^{(t)}, w^{(t+j)}). \tag{14.2.5}$$

However, the events included in the model only consider positive examples. In this case, only when all the word vectors are equal and their values approach infinity can the joint probability above be maximized to 1. Obviously, such word vectors are meaningless. Negative sampling makes the objective function more meaningful by sampling with an addition of negative examples. Assume that event P occurs when context word w_o appears in the context window of central target word w_c, and we sample K words that do not appear in the context window according to the distribution $P(w)$ to act as noise words. We assume the event for noise word $w_k (k = 1, \ldots, K)$ to not appear in the context window of central target word w_c is N_k. Suppose that events P and $N_1, \ldots, N_K$ for both positive and negative examples are independent of each other. By considering negative sampling, we can rewrite the joint probability above, which only considers the positive examples, as

$$\prod_{t=1}^{T} \prod_{-m \leq j \leq m,\ j \neq 0} P(w^{(t+j)} \mid w^{(t)}), \tag{14.2.6}$$

Here, the conditional probability is approximated to be

$$P(w^{(t+j)} \mid w^{(t)}) = P(D = 1 \mid w^{(t)}, w^{(t+j)}) \prod_{k=1,\ w_k \sim P(w)}^{K} P(D = 0 \mid w^{(t)}, w_k). \tag{14.2.7}$$

Let the text sequence index of word $w^{(t)}$ at timestep t be i_t and h_k for noise word w_k in the dictionary. The logarithmic loss for the conditional probability above is

$$\begin{aligned} -\log P(w^{(t+j)} \mid w^{(t)}) &= -\log P(D = 1 \mid w^{(t)}, w^{(t+j)}) - \sum_{k=1,\ w_k \sim P(w)}^{K} \log P(D = 0 \mid w^{(t)}, w_k) \\ &= -\log \sigma\left(\mathbf{u}_{i_{t+j}}^\top \mathbf{v}_{i_t}\right) - \sum_{k=1,\ w_k \sim P(w)}^{K} \log\left(1 - \sigma\left(\mathbf{u}_{h_k}^\top \mathbf{v}_{i_t}\right)\right) \\ &= -\log \sigma\left(\mathbf{u}_{i_{t+j}}^\top \mathbf{v}_{i_t}\right) - \sum_{k=1,\ w_k \sim P(w)}^{K} \log \sigma\left(-\mathbf{u}_{h_k}^\top \mathbf{v}_{i_t}\right). \end{aligned} \tag{14.2.8}$$

Here, the gradient computation in each step of the training is no longer related to the dictionary size, but linearly related to K. When K takes a smaller constant, the negative sampling has a lower computational overhead for each step.

14.2.2 Hierarchical Softmax

Hierarchical softmax is another type of approximate training method. It uses a binary tree for data structure as illustrated in Fig. 14.2.1, with the leaf nodes of the tree representing every word in the dictionary $\mathcal{V}$.

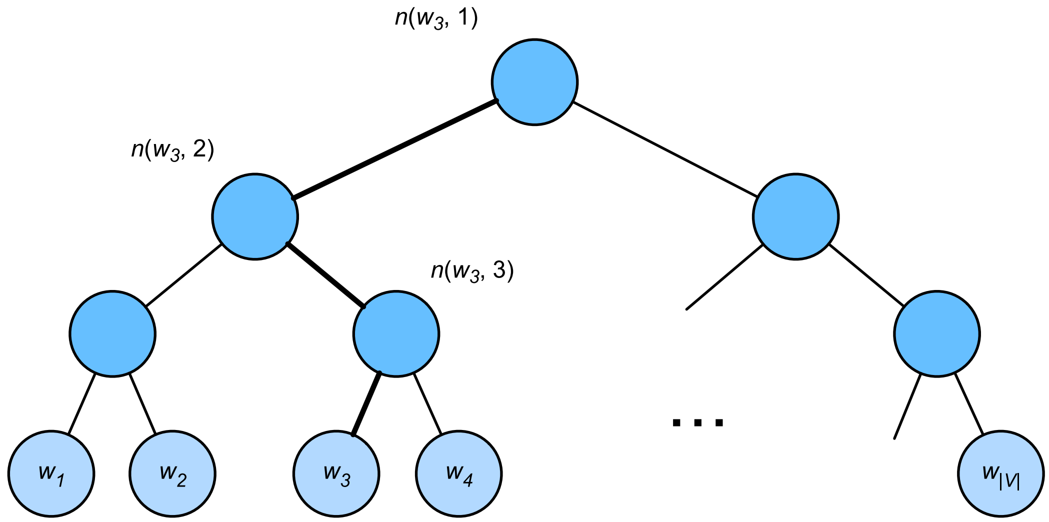


Fig. 14.2.1: Hierarchical Softmax. Each leaf node of the tree represents a word in the dictionary.

We assume that $L(w)$ is the number of nodes on the path (including the root and leaf nodes) from the root node of the binary tree to the leaf node of word w. Let $n(w, j)$ be the j^{th} node on this path, with the context word vector $\mathbf{u}_{n(w,j)}$. We use Figure 10.3 as an example, so $L(w_3) = 4$. Hierarchical softmax will approximate the conditional probability in the skip-gram model as

$$P(w_o \mid w_c) = \prod_{j=1}^{L(w_o)-1} \sigma\left([\![n(w_o, j+1) = \text{leftChild}(n(w_o, j))]\!] \cdot \mathbf{u}_{n(w_o,j)}^\top \mathbf{v}_c \right), \tag{14.2.9}$$

Here the σ function has the same definition as the sigmoid activation function, and leftChild(n) is the left child node of node n. If x is true, $[\![x]\!] = 1$; otherwise $[\![x]\!] = -1$. Now, we will compute the conditional probability of generating word w_3 based on the given word w_c in Figure 10.3. We need to find the inner product of word vector $\mathbf{v}_c$ (for word w_c) and each non-leaf node vector on the path from the root node to w_3. Because, in the binary tree, the path from the root node to leaf node w_3 needs to be traversed left, right, and left again (the path with the bold line in Figure 10.3), we get

$$P(w_3 \mid w_c) = \sigma(\mathbf{u}_{n(w_3,1)}^\top \mathbf{v}_c) \cdot \sigma(-\mathbf{u}_{n(w_3,2)}^\top \mathbf{v}_c) \cdot \sigma(\mathbf{u}_{n(w_3,3)}^\top \mathbf{v}_c). \tag{14.2.10}$$

Because $\sigma(x) + \sigma(-x) = 1$, the condition that the sum of the conditional probability of any word generated based on the given central target word w_c in dictionary $\mathcal{V}$ be 1 will also suffice:

$$\sum_{w \in \mathcal{V}} P(w \mid w_c) = 1. \tag{14.2.11}$$

In addition, because the order of magnitude for $L(w_o) - 1$ is $\mathcal{O}(\log_2 |\mathcal{V}|)$, when the size of dictionary $\mathcal{V}$ is large, the computational overhead for each step in the hierarchical softmax training is greatly reduced compared to situations where we do not use approximate training.

Summary

- Negative sampling constructs the loss function by considering independent events that contain both positive and negative examples. The gradient computational overhead for each step in the training process is linearly related to the number of noise words we sample.
- Hierarchical softmax uses a binary tree and constructs the loss function based on the path from the root node to the leaf node. The gradient computational overhead for each step in the training process is related to the logarithm of the dictionary size.

Exercises

1. Before reading the next section, think about how we should sample noise words in negative sampling.
2. What makes the last formula in this section hold?
3. How can we apply negative sampling and hierarchical softmax in the skip-gram model?

14.3 The Dataset for Word2vec

In this section, we will introduce how to preprocess a dataset with negative sampling Section 14.2 and load into minibatches for word2vec training. The dataset we use is Penn Tree Bank (PTB)[218], which is a small but commonly-used corpus. It takes samples from Wall Street Journal articles and includes training sets, validation sets, and test sets.

First, import the packages and modules required for the experiment.

```
import d2l
import math
from mxnet import gluon, np
import random
import zipfile
```

14.3.1 Reading and Preprocessing the Dataset

This dataset has already been preprocessed. Each line of the dataset acts as a sentence. All the words in a sentence are separated by spaces. In the word embedding task, each word is a token.

```
# Saved in the d2l package for later use
def read_ptb():
    with zipfile.ZipFile('../data/ptb.zip', 'r') as f:
        raw_text = f.read('ptb/ptb.train.txt').decode("utf-8")
```

(continues on next page)

[218] https://catalog.ldc.upenn.edu/LDC99T42

(continued from previous page)

```
    return [line.split() for line in raw_text.split('\n')]

sentences = read_ptb()
'# sentences: %d' % len(sentences)
```

```
'# sentences: 42069'
```

Next we build a vocabulary with words appeared not greater than 10 times mapped into a "<unk>" token. Note that the preprocessed PTB data also contains "<unk>" tokens presenting rare words.

```
vocab = d2l.Vocab(sentences, min_freq=10)
'vocab size: %d' % len(vocab)
```

```
'vocab size: 6719'
```

14.3.2 Subsampling

In text data, there are generally some words that appear at high frequencies, such "the", "a", and "in" in English. Generally speaking, in a context window, it is better to train the word embedding model when a word (such as "chip") and a lower-frequency word (such as "microprocessor") appear at the same time, rather than when a word appears with a higher-frequency word (such as "the"). Therefore, when training the word embedding model, we can perform subsampling[2] on the words. Specifically, each indexed word w_i in the dataset will drop out at a certain probability. The dropout probability is given as:

$$P(w_i) = \max\left(1 - \sqrt{\frac{t}{f(w_i)}}, 0\right), \tag{14.3.1}$$

Here, $f(w_i)$ is the ratio of the instances of word w_i to the total number of words in the dataset, and the constant t is a hyperparameter (set to 10^{-4} in this experiment). As we can see, it is only possible to drop out the word w_i in subsampling when $f(w_i) > t$. The higher the word's frequency, the higher its dropout probability.

```
# Saved in the d2l package for later use
def subsampling(sentences, vocab):
    # Map low frequency words into <unk>
    sentences = [[vocab.idx_to_token[vocab[tk]] for tk in line]
                 for line in sentences]
    # Count the frequency for each word
    counter = d2l.count_corpus(sentences)
    num_tokens = sum(counter.values())

    # Return True if to keep this token during subsampling
    def keep(token):
        return(random.uniform(0, 1) <
               math.sqrt(1e-4 / counter[token] * num_tokens))

    # Now do the subsampling
    return [[tk for tk in line if keep(tk)] for line in sentences]
```

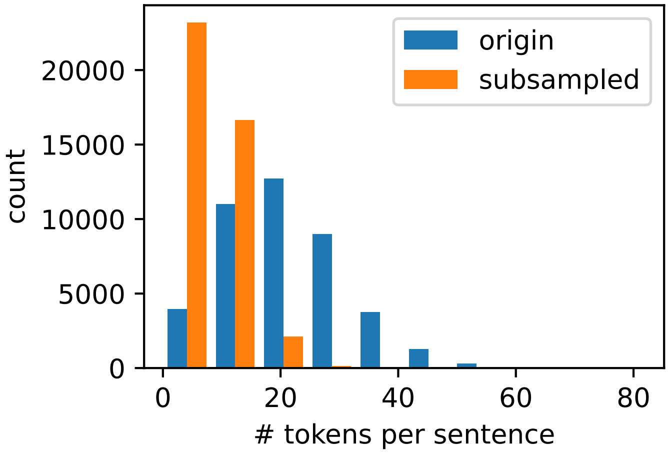
(continues on next page)

(continued from previous page)

```
subsampled = subsampling(sentences, vocab)
```

Compare the sequence lengths before and after sampling, we can see subsampling significantly reduced the sequence length.

```
d2l.set_figsize((3.5, 2.5))
d2l.plt.hist([[len(line) for line in sentences],
              [len(line) for line in subsampled]])
d2l.plt.xlabel('# tokens per sentence')
d2l.plt.ylabel('count')
d2l.plt.legend(['origin', 'subsampled']);
```



For individual tokens, the sampling rate of the high-frequency word "the" is less than 1/20.

```
def compare_counts(token):
    return '# of "%s": before=%d, after=%d' % (token, sum(
        [line.count(token) for line in sentences]), sum(
        [line.count(token) for line in subsampled]))

compare_counts('the')
```

```
'# of "the": before=50770, after=2107'
```

But the low-frequency word "join" is completely preserved.

```
compare_counts('join')
```

```
'# of "join": before=45, after=45'
```

Last, we map each token into an index to construct the corpus.

```
corpus = [vocab[line] for line in subsampled]
corpus[0:3]
```

```
[[], [71, 392, 2132, 18, 275, 406], [0, 140, 0, 5464, 3080, 1595, 95]]
```

14.3.3 Loading the Dataset

Next we read the corpus with token indicies into data batches for training.

Extracting Central Target Words and Context Words

We use words with a distance from the central target word not exceeding the context window size as the context words of the given center target word. The following definition function extracts all the central target words and their context words. It uniformly and randomly samples an integer to be used as the context window size between integer 1 and the `max_window_size` (maximum context window).

```
# Saved in the d2l package for later use
def get_centers_and_contexts(corpus, max_window_size):
    centers, contexts = [], []
    for line in corpus:
        # Each sentence needs at least 2 words to form a
        # "central target word - context word” pair
        if len(line) < 2:
            continue
        centers += line
        for i in range(len(line)):  # Context window centered at i
            window_size = random.randint(1, max_window_size)
            indices = list(range(max(0, i - window_size),
                                 min(len(line), i + 1 + window_size)))
            # Exclude the central target word from the context words
            indices.remove(i)
            contexts.append([line[idx] for idx in indices])
    return centers, contexts
```

Next, we create an artificial dataset containing two sentences of 7 and 3 words, respectively. Assume the maximum context window is 2 and print all the central target words and their context words.

```
tiny_dataset = [list(range(7)), list(range(7, 10))]
print('dataset', tiny_dataset)
for center, context in zip(*get_centers_and_contexts(tiny_dataset, 2)):
    print('center', center, 'has contexts', context)
```

```
dataset [[0, 1, 2, 3, 4, 5, 6], [7, 8, 9]]
center 0 has contexts [1, 2]
center 1 has contexts [0, 2]
center 2 has contexts [0, 1, 3, 4]
center 3 has contexts [1, 2, 4, 5]
center 4 has contexts [3, 5]
center 5 has contexts [3, 4, 6]
center 6 has contexts [4, 5]
center 7 has contexts [8, 9]
```

(continues on next page)

(continued from previous page)

```
center 8 has contexts [7, 9]
center 9 has contexts [7, 8]
```

We set the maximum context window size to 5. The following extracts all the central target words and their context words in the dataset.

```
all_centers, all_contexts = get_centers_and_contexts(corpus, 5)
'# center-context pairs: %d' % len(all_centers)
```

```
'# center-context pairs: 353705'
```

Negative Sampling

We use negative sampling for approximate training. For a central and context word pair, we randomly sample K noise words ($K = 5$ in the experiment). According to the suggestion in the Word2vec paper, the noise word sampling probability $P(w)$ is the ratio of the word frequency of w to the total word frequency raised to the power of 0.75 [2].

We first define a class to draw a candidate according to the sampling weights. It caches a 10000 size random number bank instead of calling `random.choices` every time.

```
# Saved in the d2l package for later use
class RandomGenerator(object):
    """Draw a random int in [0, n] according to n sampling weights"""
    def __init__(self, sampling_weights):
        self.population = list(range(len(sampling_weights)))
        self.sampling_weights = sampling_weights
        self.candidates = []
        self.i = 0

    def draw(self):
        if self.i == len(self.candidates):
            self.candidates = random.choices(
                self.population, self.sampling_weights, k=10000)
            self.i = 0
        self.i += 1
        return self.candidates[self.i-1]

generator = RandomGenerator([2, 3, 4])
[generator.draw() for _ in range(10)]
```

```
[0, 2, 0, 2, 2, 1, 0, 0, 2, 2]
```

```
# Saved in the d2l package for later use
def get_negatives(all_contexts, corpus, K):
    counter = d2l.count_corpus(corpus)
    sampling_weights = [counter[i]**0.75 for i in range(len(counter))]
    all_negatives, generator = [], RandomGenerator(sampling_weights)
    for contexts in all_contexts:
```

(continues on next page)

(continued from previous page)

```
        negatives = []
        while len(negatives) < len(contexts) * K:
            neg = generator.draw()
            # Noise words cannot be context words
            if neg not in contexts:
                negatives.append(neg)
        all_negatives.append(negatives)
    return all_negatives

all_negatives = get_negatives(all_contexts, corpus, 5)
```

Reading into Batches

We extract all central target words `all_centers`, and the context words `all_contexts` and noise words `all_negatives` of each central target word from the dataset. We will read them in random minibatches.

In a minibatch of data, the i^{th} example includes a central word and its corresponding n_i context words and m_i noise words. Since the context window size of each example may be different, the sum of context words and noise words, $n_i + m_i$, will be different. When constructing a minibatch, we concatenate the context words and noise words of each example, and add 0s for padding until the length of the concatenations are the same, that is, the length of all concatenations is $\max_i n_i + m_i$(`max_len`). In order to avoid the effect of padding on the loss function calculation, we construct the mask variable `masks`, each element of which corresponds to an element in the concatenation of context and noise words, `contexts_negatives`. When an element in the variable `contexts_negatives` is a padding, the element in the mask variable `masks` at the same position will be 0. Otherwise, it takes the value 1. In order to distinguish between positive and negative examples, we also need to distinguish the context words from the noise words in the `contexts_negatives` variable. Based on the construction of the mask variable, we only need to create a label variable `labels` with the same shape as the `contexts_negatives` variable and set the elements corresponding to context words (positive examples) to 1, and the rest to 0.

Next, we will implement the minibatch reading function `batchify`. Its minibatch input `data` is a list whose length is the batch size, each element of which contains central target words `center`, context words `context`, and noise words `negative`. The minibatch data returned by this function conforms to the format we need, for example, it includes the mask variable.

```
# Saved in the d2l package for later use
def batchify(data):
    max_len = max(len(c) + len(n) for _, c, n in data)
    centers, contexts_negatives, masks, labels = [], [], [], []
    for center, context, negative in data:
        cur_len = len(context) + len(negative)
        centers += [center]
        contexts_negatives += [context + negative + [0] * (max_len - cur_len)]
        masks += [[1] * cur_len + [0] * (max_len - cur_len)]
        labels += [[1] * len(context) + [0] * (max_len - len(context))]
    return (np.array(centers).reshape(-1, 1), np.array(contexts_negatives),
            np.array(masks), np.array(labels))
```

Construct two simple examples:

```
x_1 = (1, [2, 2], [3, 3, 3, 3])
x_2 = (1, [2, 2, 2], [3, 3])
batch = batchify((x_1, x_2))

names = ['centers', 'contexts_negatives', 'masks', 'labels']
for name, data in zip(names, batch):
    print(name, '=', data)
```

```
centers = [[1.]
 [1.]]
contexts_negatives = [[2. 2. 3. 3. 3. 3.]
 [2. 2. 2. 3. 3. 0.]]
masks = [[1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 0.]]
labels = [[1. 1. 0. 0. 0. 0.]
 [1. 1. 1. 0. 0. 0.]]
```

We use the batchify function just defined to specify the minibatch reading method in the DataLoader instance.

14.3.4 Putting All Things Together

Last, we define the load_data_ptb function that read the PTB dataset and return the data loader.

```
# Saved in the d2l package for later use
def load_data_ptb(batch_size, max_window_size, num_noise_words):
    sentences = read_ptb()
    vocab = d2l.Vocab(sentences, min_freq=10)
    subsampled = subsampling(sentences, vocab)
    corpus = [vocab[line] for line in subsampled]
    all_centers, all_contexts = get_centers_and_contexts(
        corpus, max_window_size)
    all_negatives = get_negatives(all_contexts, corpus, num_noise_words)
    dataset = gluon.data.ArrayDataset(
        all_centers, all_contexts, all_negatives)
    data_iter = gluon.data.DataLoader(dataset, batch_size, shuffle=True,
                                      batchify_fn=batchify)
    return data_iter, vocab
```

Let’s print the first minibatch of the data iterator.

```
data_iter, vocab = load_data_ptb(512, 5, 5)
for batch in data_iter:
    for name, data in zip(names, batch):
        print(name, 'shape:', data.shape)
    break
```

```
centers shape: (512, 1)
contexts_negatives shape: (512, 60)
masks shape: (512, 60)
labels shape: (512, 60)
```

Summary

- Subsampling attempts to minimize the impact of high-frequency words on the training of a word embedding model.
- We can pad examples of different lengths to create minibatches with examples of all the same length and use mask variables to distinguish between padding and non-padding elements, so that only non-padding elements participate in the calculation of the loss function.

Exercises

1. We use the `batchify` function to specify the minibatch reading method in the `DataLoader` instance and print the shape of each variable in the first batch read. How should these shapes be calculated?

14.4 Implementation of Word2vec

In this section, we will train a skip-gram model defined in Section 14.1.

First, import the packages and modules required for the experiment, and load the PTB dataset.

```
import d2l
from mxnet import autograd, gluon, np, npx
from mxnet.gluon import nn
npx.set_np()

batch_size, max_window_size, num_noise_words = 512, 5, 5
data_iter, vocab = d2l.load_data_ptb(512, 5, 5)
```

14.4.1 The Skip-Gram Model

We will implement the skip-gram model by using embedding layers and minibatch multiplication. These methods are also often used to implement other natural language processing applications.

Embedding Layer

The layer in which the obtained word is embedded is called the embedding layer, which can be obtained by creating an `nn.Embedding` instance in Gluon. The weight of the embedding layer is a matrix whose number of rows is the dictionary size (`input_dim`) and whose number of columns is the dimension of each word vector (`output_dim`). We set the dictionary size to 20 and the word vector dimension to 4.

```
embed = nn.Embedding(input_dim=20, output_dim=4)
embed.initialize()
embed.weight
```

```
Parameter embedding0_weight (shape=(20, 4), dtype=float32)
```

The input of the embedding layer is the index of the word. When we enter the index i of a word, the embedding layer returns the i^{th} row of the weight matrix as its word vector. Below we enter an index of shape $(2, 3)$ into the embedding layer. Because the dimension of the word vector is 4, we obtain a word vector of shape $(2, 3, 4)$.

```
x = np.array([[1, 2, 3], [4, 5, 6]])
embed(x)
```

```
array([[[ 0.01438687,  0.05011239,  0.00628365,  0.04861524],
        [-0.01068833,  0.01729892,  0.02042518, -0.01618656],
        [-0.00873779, -0.02834515,  0.05484822, -0.06206018]],

       [[ 0.06491279, -0.03182812, -0.01631819, -0.00312688],
        [ 0.0408415 ,  0.04370362,  0.00404529, -0.0028032 ],
        [ 0.00952624, -0.01501013,  0.05958354,  0.04705103]]])
```

Minibatch Multiplication

We can multiply the matrices in two minibatches one by one, by the minibatch multiplication operation batch_dot. Suppose the first batch contains n matrices $\mathbf{X}_1, \ldots, \mathbf{X}_n$ with a shape of $a \times b$, and the second batch contains n matrices $\mathbf{Y}_1, \ldots, \mathbf{Y}_n$ with a shape of $b \times c$. The output of matrix multiplication on these two batches are n matrices $\mathbf{X}_1\mathbf{Y}_1, \ldots, \mathbf{X}_n\mathbf{Y}_n$ with a shape of $a \times c$. Therefore, given two ndarrays of shape (n, a, b) and (n, b, c), the shape of the minibatch multiplication output is (n, a, c).

```
X = np.ones((2, 1, 4))
Y = np.ones((2, 4, 6))
npx.batch_dot(X, Y).shape
```

```
(2, 1, 6)
```

Skip-gram Model Forward Calculation

In forward calculation, the input of the skip-gram model contains the central target word index center and the concatenated context and noise word index contexts_and_negatives. In which, the center variable has the shape (batch size, 1), while the contexts_and_negatives variable has the shape (batch size, max_len). These two variables are first transformed from word indexes to word vectors by the word embedding layer, and then the output of shape (batch size, 1, max_len) is obtained by minibatch multiplication. Each element in the output is the inner product of the central target word vector and the context word vector or noise word vector.

```
def skip_gram(center, contexts_and_negatives, embed_v, embed_u):
    v = embed_v(center)
    u = embed_u(contexts_and_negatives)
    pred = npx.batch_dot(v, u.swapaxes(1, 2))
    return pred
```

Verify that the output shape should be (batch size, 1, `max_len`).

```
skip_gram(np.ones((2, 1)), np.ones((2, 4)), embed, embed).shape
```

```
(2, 1, 4)
```

14.4.2 Training

Before training the word embedding model, we need to define the loss function of the model.

Binary Cross Entropy Loss Function

According to the definition of the loss function in negative sampling, we can directly use Gluon's binary cross-entropy loss function `SigmoidBinaryCrossEntropyLoss`.

```
loss = gluon.loss.SigmoidBinaryCrossEntropyLoss()
```

It is worth mentioning that we can use the mask variable to specify the partial predicted value and label that participate in loss function calculation in the minibatch: when the mask is 1, the predicted value and label of the corresponding position will participate in the calculation of the loss function; When the mask is 0, the predicted value and label of the corresponding position do not participate in the calculation of the loss function. As we mentioned earlier, mask variables can be used to avoid the effect of padding on loss function calculations.

Given two identical examples, different masks lead to different loss values.

```
pred = np.array([[.5]*4]*2)
label = np.array([[1, 0, 1, 0]]*2)
mask = np.array([[1, 1, 1, 1], [1, 1, 0, 0]])
loss(pred, label, mask)
```

```
array([0.724077 , 0.3620385])
```

We can normalize the loss in each example due to various lengths in each example.

```
loss(pred, label, mask) / mask.sum(axis=1) * mask.shape[1]
```

```
array([0.724077, 0.724077])
```

Initializing Model Parameters

We construct the embedding layers of the central and context words, respectively, and set the hyperparameter word vector dimension embed_size to 100.

```
embed_size = 100
net = nn.Sequential()
net.add(nn.Embedding(input_dim=len(vocab), output_dim=embed_size),
        nn.Embedding(input_dim=len(vocab), output_dim=embed_size))
```

Training

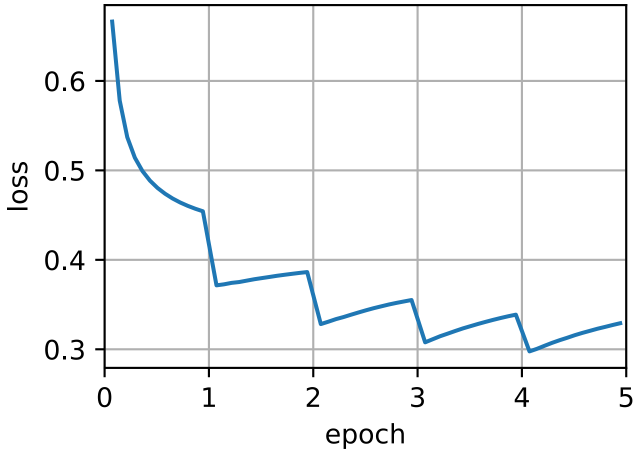
The training function is defined below. Because of the existence of padding, the calculation of the loss function is slightly different compared to the previous training functions.

```
def train(net, data_iter, lr, num_epochs, ctx=d2l.try_gpu()):
    net.initialize(ctx=ctx, force_reinit=True)
    trainer = gluon.Trainer(net.collect_params(), 'adam',
                            {'learning_rate': lr})
    animator = d2l.Animator(xlabel='epoch', ylabel='loss',
                            xlim=[0, num_epochs])
    for epoch in range(num_epochs):
        timer = d2l.Timer()
        metric = d2l.Accumulator(2)  # loss_sum, num_tokens
        for i, batch in enumerate(data_iter):
            center, context_negative, mask, label = [
                data.as_in_context(ctx) for data in batch]
            with autograd.record():
                pred = skip_gram(center, context_negative, net[0], net[1])
                l = (loss(pred.reshape(label.shape), label, mask)
                     / mask.sum(axis=1) * mask.shape[1])
            l.backward()
            trainer.step(batch_size)
            metric.add(l.sum(), l.size)
            if (i+1) % 50 == 0:
                animator.add(epoch+(i+1)/len(data_iter),
                             (metric[0]/metric[1],))
    print('loss %.3f, %d tokens/sec on %s ' % (
        metric[0]/metric[1], metric[1]/timer.stop(), ctx))
```

Now, we can train a skip-gram model using negative sampling.

```
lr, num_epochs = 0.01, 5
train(net, data_iter, lr, num_epochs)
```

```
loss 0.330, 27378 tokens/sec on gpu(0)
```



14.4.3 Applying the Word Embedding Model

After training the word embedding model, we can represent similarity in meaning between words based on the cosine similarity of two word vectors. As we can see, when using the trained word embedding model, the words closest in meaning to the word "chip" are mostly related to chips.

```
def get_similar_tokens(query_token, k, embed):
    W = embed.weight.data()
    x = W[vocab[query_token]]
    # Compute the cosine similarity. Add 1e-9 for numerical stability
    cos = np.dot(W, x) / np.sqrt(np.sum(W * W, axis=1) * np.sum(x * x) + 1e-9)
    topk = npx.topk(cos, k=k+1, ret_typ='indices').asnumpy().astype('int32')
    for i in topk[1:]:  # Remove the input words
        print('cosine sim=%.3f: %s' % (cos[i], (vocab.idx_to_token[i])))

get_similar_tokens('chip', 3, net[0])
```

```
cosine sim=0.584: intel
cosine sim=0.558: computer
cosine sim=0.524: mips
```

Summary

- We can use Gluon to train a skip-gram model through negative sampling.

Exercises

1. Set `sparse_grad=True` when creating an instance of `nn.Embedding`. Does it accelerate training? Look up MXNet documentation to learn the meaning of this argument.
2. Try to find synonyms for other words.
3. Tune the hyper-parameters and observe and analyze the experimental results.
4. When the dataset is large, we usually sample the context words and the noise words for the central target word in the current minibatch only when updating the model parameters. In other words, the same central target word may have different context words or noise words in different epochs. What are the benefits of this sort of training? Try to implement this training method.

14.5 Subword Embedding (fastText)

English words usually have internal structures and formation methods. For example, we can deduce the relationship between “dog”, “dogs”, and “dogcatcher” by their spelling. All these words have the same root, “dog”, but they use different suffixes to change the meaning of the word. Moreover, this association can be extended to other words. For example, the relationship between “dog” and “dogs” is just like the relationship between “cat” and “cats”. The relationship between “boy” and “boyfriend” is just like the relationship between “girl” and “girlfriend”. This characteristic is not unique to English. In French and Spanish, a lot of verbs can have more than 40 different forms depending on the context. In Finnish, a noun may have more than 15 forms. In fact, morphology, which is an important branch of linguistics, studies the internal structure and formation of words.

In word2vec, we did not directly use morphology information. In both the skip-gram model and continuous bag-of-words model, we use different vectors to represent words with different forms. For example, “dog” and “dogs” are represented by two different vectors, while the relationship between these two vectors is not directly represented in the model. In view of this, fastText (Bojanowski et al., 2017) proposes the method of subword embedding, thereby attempting to introduce morphological information in the skip-gram model in word2vec.

In fastText, each central word is represented as a collection of subwords. Below we use the word “where” as an example to understand how subwords are formed. First, we add the special characters “<” and “>” at the beginning and end of the word to distinguish the subwords used as prefixes and suffixes. Then, we treat the word as a sequence of characters to extract the n-grams. For example, when $n = 3$, we can get all subwords with a length of 3:

$$\text{"<wh", "whe", "her", "ere", "re>"}, \tag{14.5.1}$$

and the special subword "<where>".

In fastText, for a word w, we record the union of all its subwords with length of 3 to 6 and special subwords as $\mathcal{G}_w$. Thus, the dictionary is the union of the collection of subwords of all words. Assume the vector of the subword g in the dictionary is $\mathbf{z}_g$. Then, the central word vector $\mathbf{u}_w$ for the word w in the skip-gram model can be expressed as

$$\mathbf{u}_w = \sum_{g \in \mathcal{G}_w} \mathbf{z}_g. \tag{14.5.2}$$

The rest of the fastText process is consistent with the skip-gram model, so it is not repeated here. As we can see, compared with the skip-gram model, the dictionary in fastText is larger, resulting in more model parameters. Also, the vector of one word requires the summation of all subword vectors, which results in higher computation complexity. However, we can obtain better vectors for more uncommon complex words, even words not existing in the dictionary, by looking at other words with similar structures.

Summary

- FastText proposes a subword embedding method. Based on the skip-gram model in word2vec, it represents the central word vector as the sum of the subword vectors of the word.
- Subword embedding utilizes the principles of morphology, which usually improves the quality of representations of uncommon words.

Exercises

1. When there are too many subwords (for example, 6 words in English result in about 3×10^8 combinations), what problems arise? Can you think of any methods to solve them? Hint: Refer to the end of section 3.2 of the fastText paper[1].
2. How can you design a subword embedding model based on the continuous bag-of-words model?

14.6 Word Embedding with Global Vectors (GloVe)

First, we should review the skip-gram model in word2vec. The conditional probability $P(w_j \mid w_i)$ expressed in the skip-gram model using the softmax operation will be recorded as q_{ij}, that is:

$$q_{ij} = \frac{\exp(\mathbf{u}_j^\top \mathbf{v}_i)}{\sum_{k \in \mathcal{V}} \exp(\mathbf{u}_k^\top \mathbf{v}_i)}, \tag{14.6.1}$$

where $\mathbf{v}_i$ and $\mathbf{u}_i$ are the vector representations of word w_i of index i as the center word and context word respectively, and $\mathcal{V} = \{0, 1, \ldots, |\mathcal{V}| - 1\}$ is the vocabulary index set.

For word w_i, it may appear in the dataset for multiple times. We collect all the context words every time when w_i is a center word and keep duplicates, denoted as multiset $\mathcal{C}_i$. The number of an element in a multiset is called the multiplicity of the element. For instance, suppose that word w_i appears twice in the dataset: the context windows when these two w_i become center words in the text sequence contain context word indices $2, 1, 5, 2$ and $2, 3, 2, 1$. Then, multiset $\mathcal{C}_i = \{1, 1, 2, 2, 2, 2, 3, 5\}$, where multiplicity of element 1 is 2, multiplicity of element 2 is 4, and multiplicities of elements 3 and 5 are both 1. Denote multiplicity of element j in multiset $\mathcal{C}_i$ as x_{ij}: it is the number of word w_j in all the context windows for center word w_i in the entire dataset. As a result, the loss function of the skip-gram model can be expressed in a different way:

$$-\sum_{i\in\mathcal{V}}\sum_{j\in\mathcal{V}} x_{ij} \log\, q_{ij}. \tag{14.6.2}$$

We add up the number of all the context words for the central target word w_i to get x_i, and record the conditional probability x_{ij}/x_i for generating context word w_j based on central target word w_i as p_{ij}. We can rewrite the loss function of the skip-gram model as

$$-\sum_{i\in\mathcal{V}} x_i \sum_{j\in\mathcal{V}} p_{ij} \log\, q_{ij}. \tag{14.6.3}$$

In the formula above, $\sum_{j\in\mathcal{V}} p_{ij} \log\, q_{ij}$ computes the conditional probability distribution p_{ij} for context word generation based on the central target word w_i and the cross-entropy of conditional probability distribution q_{ij} predicted by the model. The loss function is weighted using the sum of the number of context words with the central target word w_i. If we minimize the loss function from the formula above, we will be able to allow the predicted conditional probability distribution to approach as close as possible to the true conditional probability distribution.

However, although the most common type of loss function, the cross-entropy loss function is sometimes not a good choice. On the one hand, as we mentioned in Section 14.2 the cost of letting the model prediction q_{ij} become the legal probability distribution has the sum of all items in the entire dictionary in its denominator. This can easily lead to excessive computational overhead. On the other hand, there are often a lot of uncommon words in the dictionary, and they appear rarely in the dataset. In the cross-entropy loss function, the final prediction of the conditional probability distribution on a large number of uncommon words is likely to be inaccurate.

14.6.1 The GloVe Model

To address this, GloVe (Pennington et al., 2014), a word embedding model that came after word2vec, adopts square loss and makes three changes to the skip-gram model based on this loss.

1. Here, we use the non-probability distribution variables $p'_{ij} = x_{ij}$ and $q'_{ij} = \exp(\mathbf{u}_j^\top \mathbf{v}_i)$ and take their logs. Therefore, we get the square loss $\left(\log\, p'_{ij} - \log\, q'_{ij}\right)^2 = \left(\mathbf{u}_j^\top \mathbf{v}_i - \log\, x_{ij}\right)^2$.
2. We add two scalar model parameters for each word w_i: the bias terms b_i (for central target words) and c_i(for context words).
3. Replace the weight of each loss with the function $h(x_{ij})$. The weight function $h(x)$ is a monotone increasing function with the range $[0, 1].

Therefore, the goal of GloVe is to minimize the loss function.

$$\sum_{i\in\mathcal{V}}\sum_{j\in\mathcal{V}} h(x_{ij}) \left(\mathbf{u}_j^\top \mathbf{v}_i + b_i + c_j - \log\, x_{ij}\right)^2. \tag{14.6.4}$$

Here, we have a suggestion for the choice of weight function $h(x)$: when $x < c$ (e.g $c = 100$), make $h(x) = (x/c)^{\alpha}$ (e.g $\alpha = 0.75$), otherwise make $h(x) = 1$. Because $h(0) = 0$, the squared loss term for $x_{ij} = 0$ can be simply ignored. When we use minibatch SGD for training, we conduct random sampling to get a non-zero minibatch x_{ij} from each timestep and compute the gradient to update the model parameters. These non-zero x_{ij} are computed in advance based on the entire dataset and they contain global statistics for the dataset. Therefore, the name GloVe is taken from "Global Vectors".

Notice that if word w_i appears in the context window of word w_j, then word w_j will also appear in the context window of word w_i. Therefore, $x_{ij} = x_{ji}$. Unlike word2vec, GloVe fits the symmetric $\log x_{ij}$ in lieu of the asymmetric conditional probability p_{ij}. Therefore, the central target word vector and context word vector of any word are equivalent in GloVe. However, the two sets of word vectors that are learned by the same word may be different in the end due to different initialization values. After learning all the word vectors, GloVe will use the sum of the central target word vector and the context word vector as the final word vector for the word.

14.6.2 Understanding GloVe from Conditional Probability Ratios

We can also try to understand GloVe word embedding from another perspective. We will continue the use of symbols from earlier in this section, $P(w_j \mid w_i)$ represents the conditional probability of generating context word w_j with central target word w_i in the dataset, and it will be recorded as p_{ij}. From a real example from a large corpus, here we have the following two sets of conditional probabilities with "ice" and "steam" as the central target words and the ratio between them:

w_k=	"solid"	"gas"	"water"	"fashion"
$p_1 = P(w_k \mid$ "ice" $)$	0.00019	0.000066	0.003	0.000017
$p_2 = P(w_k \mid$ "steam" $)$	0.000022	0.00078	0.0022	0.000018
p_1/p_2	8.9	0.085	1.36	0.96

We will be able to observe phenomena such as:

- For a word w_k that is related to "ice" but not to "steam", such as $w_k =$"solid", we would expect a larger conditional probability ratio, like the value 8.9 in the last row of the table above.
- For a word w_k that is related to "steam" but not to "ice", such as $w_k =$"gas", we would expect a smaller conditional probability ratio, like the value 0.085 in the last row of the table above.
- For a word w_k that is related to both "ice" and "steam", such as $w_k =$"water", we would expect a conditional probability ratio close to 1, like the value 1.36 in the last row of the table above.
- For a word w_k that is related to neither "ice" or "steam", such as $w_k =$"fashion", we would expect a conditional probability ratio close to 1, like the value 0.96 in the last row of the table above.

We can see that the conditional probability ratio can represent the relationship between different words more intuitively. We can construct a word vector function to fit the conditional probability ratio more effectively. As we know, to obtain any ratio of this type requires three words w_i, w_j, and w_k. The conditional probability ratio with w_i as the central target word is p_{ij}/p_{ik}. We can find a function that uses word vectors to fit this conditional probability ratio.

$$f(\mathbf{u}_j, \mathbf{u}_k, \mathbf{v}_i) \approx \frac{p_{ij}}{p_{ik}}. \tag{14.6.5}$$

The possible design of function f here will not be unique. We only need to consider a more reasonable possibility. Notice that the conditional probability ratio is a scalar, we can limit f to be a scalar function: $f(\mathbf{u}_j, \mathbf{u}_k, \mathbf{v}_i) = f\left((\mathbf{u}_j - \mathbf{u}_k)^\top \mathbf{v}_i\right)$. After exchanging index j with k, we will be able to see that function f satisfies the condition $f(x)f(-x) = 1$, so one possibility could be $f(x) = \exp(x)$. Thus:

$$f(\mathbf{u}_j, \mathbf{u}_k, \mathbf{v}_i) = \frac{\exp\left(\mathbf{u}_j^\top \mathbf{v}_i\right)}{\exp\left(\mathbf{u}_k^\top \mathbf{v}_i\right)} \approx \frac{p_{ij}}{p_{ik}}. \tag{14.6.6}$$

One possibility that satisfies the right side of the approximation sign is $\exp\left(\mathbf{u}_j^\top \mathbf{v}_i\right) \approx \alpha p_{ij}$, where α is a constant. Considering that $p_{ij} = x_{ij}/x_i$, after taking the logarithm we get $\mathbf{u}_j^\top \mathbf{v}_i \approx \log\, \alpha + \log\, x_{ij} - \log\, x_i$. We use additional bias terms to fit $-\log\, \alpha + \log\, x_i$, such as the central target word bias term b_i and context word bias term c_j:

$$\mathbf{u}_j^\top \mathbf{v}_i + b_i + c_j \approx \log(x_{ij}). \tag{14.6.7}$$

By taking the square error and weighting the left and right sides of the formula above, we can get the loss function of GloVe.

Summary

- In some cases, the cross-entropy loss function may have a disadvantage. GloVe uses squared loss and the word vector to fit global statistics computed in advance based on the entire dataset.
- The central target word vector and context word vector of any word are equivalent in GloVe.

Exercises

1. If a word appears in the context window of another word, how can we use the distance between them in the text sequence to redesign the method for computing the conditional probability p_{ij}? Hint: See section 4.2 from the paper GloVe (Pennington et al., 2014).
2. For any word, will its central target word bias term and context word bias term be equivalent to each other in GloVe? Why?

14.7 Finding Synonyms and Analogies

In Section 14.4 we trained a word2vec word embedding model on a small-scale dataset and searched for synonyms using the cosine similarity of word vectors. In practice, word vectors pre-trained on a large-scale corpus can often be applied to downstream natural language processing tasks. This section will demonstrate how to use these pre-trained word vectors to find synonyms and analogies. We will continue to apply pre-trained word vectors in subsequent sections.

14.7.1 Using Pre-Trained Word Vectors

MXNet's `contrib.text` package provides functions and classes related to natural language processing (see the GluonNLP[223] tool package for more details). Next, let's check out names of the provided pre-trained word embeddings.

```
from mxnet import np, npx
from mxnet.contrib import text
npx.set_np()

text.embedding.get_pretrained_file_names().keys()
```

```
dict_keys(['glove', 'fasttext'])
```

Given the name of the word embedding, we can see which pre-trained models are provided by the word embedding. The word vector dimensions of each model may be different or obtained by pre-training on different datasets.

```
print(text.embedding.get_pretrained_file_names('glove'))
```

```
['glove.42B.300d.txt', 'glove.6B.50d.txt', 'glove.6B.100d.txt', 'glove.6B.200d.txt', 'glove.
↪6B.300d.txt', 'glove.840B.300d.txt', 'glove.twitter.27B.25d.txt', 'glove.twitter.27B.50d.
↪txt', 'glove.twitter.27B.100d.txt', 'glove.twitter.27B.200d.txt']
```

The general naming conventions for pre-trained GloVe models are "model.(dataset.)number of words in dataset.word vector dimension.txt". For more information, please refer to the GloVe and fastText project sites [2, 3]. Below, we use a 50-dimensional GloVe word vector based on Wikipedia subset pre-training. The corresponding word vector is automatically downloaded the first time we create a pre-trained word vector instance.

```
glove_6b50d = text.embedding.create(
    'glove', pretrained_file_name='glove.6B.50d.txt')
```

Print the dictionary size. The dictionary contains $400,000$ words and a special unknown token.

```
len(glove_6b50d)
```

```
400001
```

We can use a word to get its index in the dictionary, or we can get the word from its index.

```
glove_6b50d.token_to_idx['beautiful'], glove_6b50d.idx_to_token[3367]
```

```
(3367, 'beautiful')
```

[223] https://gluon-nlp.mxnet.io/

14.7.2 Applying Pre-Trained Word Vectors

Below, we demonstrate the application of pre-trained word vectors, using GloVe as an example.

Finding Synonyms

Here, we re-implement the algorithm used to search for synonyms by cosine similarity introduced in Section 14.1

In order to reuse the logic for seeking the k nearest neighbors when seeking analogies, we encapsulate this part of the logic separately in the knn (k-nearest neighbors) function.

```
def knn(W, x, k):
    # The added 1e-9 is for numerical stability
    cos = np.dot(W, x.reshape(-1,)) / (
        np.sqrt(np.sum(W * W, axis=1) + 1e-9) * np.sqrt((x * x).sum()))
    topk = npx.topk(cos, k=k, ret_typ='indices')
    return topk, [cos[int(i)] for i in topk]
```

Then, we search for synonyms by pre-training the word vector instance embed.

```
def get_similar_tokens(query_token, k, embed):
    topk, cos = knn(embed.idx_to_vec,
                    embed.get_vecs_by_tokens([query_token]), k+1)
    for i, c in zip(topk[1:], cos[1:]):  # Remove input words
        print('cosine sim=%.3f: %s' % (c, (embed.idx_to_token[int(i)])))
```

The dictionary of pre-trained word vector instance glove_6b50d already created contains 400,000 words and a special unknown token. Excluding input words and unknown words, we search for the three words that are the most similar in meaning to "chip".

```
get_similar_tokens('chip', 3, glove_6b50d)
```

```
cosine sim=0.856: chips
cosine sim=0.749: intel
cosine sim=0.749: electronics
```

Next, we search for the synonyms of "baby" and "beautiful".

```
get_similar_tokens('baby', 3, glove_6b50d)
```

```
cosine sim=0.839: babies
cosine sim=0.800: boy
cosine sim=0.792: girl
```

```
get_similar_tokens('beautiful', 3, glove_6b50d)
```

```
cosine sim=0.921: lovely
cosine sim=0.893: gorgeous
cosine sim=0.830: wonderful
```

Finding Analogies

In addition to seeking synonyms, we can also use the pre-trained word vector to seek the analogies between words. For example, “man”:“woman”::“son”:“daughter” is an example of analogy, “man” is to “woman” as “son” is to “daughter”. The problem of seeking analogies can be defined as follows: for four words in the analogical relationship $a : b :: c : d$, given the first three words, a, b and c, we want to find d. Assume the word vector for the word w is $\text{vec}(w)$. To solve the analogy problem, we need to find the word vector that is most similar to the result vector of $\text{vec}(c) + \text{vec}(b) - \text{vec}(a)$.

```
def get_analogy(token_a, token_b, token_c, embed):
    vecs = embed.get_vecs_by_tokens([token_a, token_b, token_c])
    x = vecs[1] - vecs[0] + vecs[2]
    topk, cos = knn(embed.idx_to_vec, x, 1)
    return embed.idx_to_token[int(topk[0])]  # Remove unknown words
```

Verify the “male-female” analogy.

```
get_analogy('man', 'woman', 'son', glove_6b50d)
```

```
'daughter'
```

“Capital-country” analogy: “beijing” is to “china” as “tokyo” is to what? The answer should be “japan”.

```
get_analogy('beijing', 'china', 'tokyo', glove_6b50d)
```

```
'japan'
```

“Adjective-superlative adjective” analogy: “bad” is to “worst” as “big” is to what? The answer should be “biggest”.

```
get_analogy('bad', 'worst', 'big', glove_6b50d)
```

```
'biggest'
```

“Present tense verb-past tense verb” analogy: “do” is to “did” as “go” is to what? The answer should be “went”.

```
get_analogy('do', 'did', 'go', glove_6b50d)
```

```
'went'
```

Summary

- Word vectors pre-trained on a large-scale corpus can often be applied to downstream natural language processing tasks.
- We can use pre-trained word vectors to seek synonyms and analogies.

Exercises

1. Test the fastText results.
2. If the dictionary is extremely large, how can we accelerate finding synonyms and analogies?

14.8 Text Classification and the Dataset

Text classification is a common task in natural language processing, which transforms a sequence of text of indefinite length into a category of text. It is similar to the image classification, the most frequently used application in this book, e.g., Section 17.8. The only difference is that, rather than an image, text classification's example is a text sentence.

This section will focus on loading data for one of the sub-questions in this field: using text sentiment classification to analyze the emotions of the text's author. This problem is also called sentiment analysis and has a wide range of applications. For example, we can analyze user reviews of products to obtain user satisfaction statistics, or analyze user sentiments about market conditions and use it to predict future trends.

```
import d2l
from mxnet import gluon, np, npx
import os
import tarfile
npx.set_np()
```

14.8.1 The Text Sentiment Classification Dataset

We use Stanford's Large Movie Review Dataset as the dataset for text sentiment classification[1]. This dataset is divided into two datasets for training and testing purposes, each containing 25,000 movie reviews downloaded from IMDb. In each dataset, the number of comments labeled as "positive" and "negative" is equal.

Reading the Dataset

We first download this dataset to the "../data" path and extract it to "../data/aclImdb".

```
# Saved in the d2l package for later use
def download_imdb(data_dir='../data'):
    url = 'http://ai.stanford.edu/~amaas/data/sentiment/aclImdb_v1.tar.gz'
    fname = gluon.utils.download(url, data_dir)
    with tarfile.open(fname, 'r') as f:
        f.extractall(data_dir)

download_imdb()
```

Next, read the training and test datasets. Each example is a review and its corresponding label: 1 indicates "positive" and 0 indicates "negative".

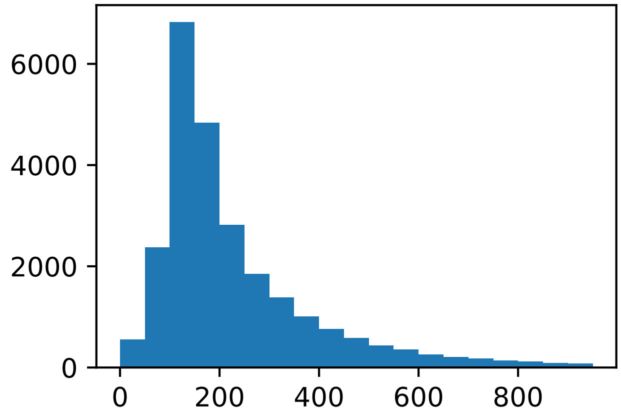
```
# Saved in the d2l package for later use
def read_imdb(folder='train', data_dir='../data'):
    data, labels = [], []
    for label in ['pos', 'neg']:
        folder_name = os.path.join(data_dir, 'aclImdb', folder, label)
        for file in os.listdir(folder_name):
            with open(os.path.join(folder_name, file), 'rb') as f:
                review = f.read().decode('utf-8').replace('\n', '')
                data.append(review)
                labels.append(1 if label == 'pos' else 0)
    return data, labels

train_data = read_imdb('train')
print('# trainings:', len(train_data[0]))
for x, y in zip(train_data[0][:3], train_data[1][:3]):
    print('label:', y, 'review:', x[0:60])
```

```
# trainings: 25000
label: 1 review: Normally the best way to annoy me in a film is to include so
label: 1 review: The Bible teaches us that the love of money is the root of a
label: 1 review: Being someone who lists Night of the Living Dead at number t
```

Tokenization and Vocabulary

We use a word as a token, and then create a dictionary based on the training dataset.

```
train_tokens = d2l.tokenize(train_data[0], token='word')
vocab = d2l.Vocab(train_tokens, min_freq=5)

d2l.set_figsize((3.5, 2.5))
d2l.plt.hist([len(line) for line in train_tokens], bins=range(0, 1000, 50));
```



Padding to the Same Length

Because the reviews have different lengths, so they cannot be directly combined into minibatches. Here we fix the length of each comment to 500 by truncating or adding "<unk>" indices.

```
num_steps = 500  # sequence length
train_features = np.array([d2l.trim_pad(vocab[line], num_steps, vocab.unk)
                           for line in train_tokens])
train_features.shape
```

```
(25000, 500)
```

Creating the Data Iterator

Now, we will create a data iterator. Each iteration will return a minibatch of data.

```
train_iter = d2l.load_array((train_features, train_data[1]), 64)

for X, y in train_iter:
    print('X', X.shape, 'y', y.shape)
    break
'# batches:', len(train_iter)
```

```
X (64, 500) y (64,)
```

```
('# batches:', 391)
```

14.8.2 Putting All Things Together

Last, we will save a function load_data_imdb into d2l, which returns the vocabulary and data iterators.

```
# Saved in the d2l package for later use
def load_data_imdb(batch_size, num_steps=500):
    download_imdb()
    train_data, test_data = read_imdb('train'), read_imdb('test')
    train_tokens = d2l.tokenize(train_data[0], token='word')
    test_tokens = d2l.tokenize(test_data[0], token='word')
    vocab = d2l.Vocab(train_tokens, min_freq=5)
    train_features = np.array([d2l.trim_pad(vocab[line], num_steps, vocab.unk)
                               for line in train_tokens])
    test_features = np.array([d2l.trim_pad(vocab[line], num_steps, vocab.unk)
                              for line in test_tokens])
    train_iter = d2l.load_array((train_features, train_data[1]), batch_size)
    test_iter = d2l.load_array((test_features, test_data[1]), batch_size,
                               is_train=False)
    return train_iter, test_iter, vocab
```

Summary

- Text classification can classify a text sequence into a category.
- To classify a text sentiment, we load an IMDb dataset and tokenize its words. Then we pad the text sequence for short reviews and create a data iterator.

Exercises

1. Discover a different natural language dataset (such as Amazon reviews[225]) and build a similar data_loader function as load_data_imdb.

14.9 Text Sentiment Classification: Using Recurrent Neural Networks

Similar to search synonyms and analogies, text classification is also a downstream application of word embedding. In this section, we will apply pre-trained word vectors and bidirectional recurrent neural networks with multiple hidden layers (Maas et al., 2011). We will use them to determine whether a text sequence of indefinite length contains positive or negative emotion. Import the required package or module before starting the experiment.

[225] https://snap.stanford.edu/data/web-Amazon.html

```python
import d2l
from mxnet import gluon, init, np, npx
from mxnet.gluon import nn, rnn
from mxnet.contrib import text
npx.set_np()

batch_size = 64
train_iter, test_iter, vocab = d2l.load_data_imdb(batch_size)
```

14.9.1 Using a Recurrent Neural Network Model

In this model, each word first obtains a feature vector from the embedding layer. Then, we further encode the feature sequence using a bidirectional recurrent neural network to obtain sequence information. Finally, we transform the encoded sequence information to output through the fully connected layer. Specifically, we can concatenate hidden states of bidirectional long-short term memory in the initial timestep and final timestep and pass it to the output layer classification as encoded feature sequence information. In the `BiRNN` class implemented below, the `Embedding` instance is the embedding layer, the `LSTM` instance is the hidden layer for sequence encoding, and the `Dense` instance is the output layer for generated classification results.

```python
class BiRNN(nn.Block):
    def __init__(self, vocab_size, embed_size, num_hiddens,
                 num_layers, **kwargs):
        super(BiRNN, self).__init__(**kwargs)
        self.embedding = nn.Embedding(vocab_size, embed_size)
        # Set Bidirectional to True to get a bidirectional recurrent neural
        # network
        self.encoder = rnn.LSTM(num_hiddens, num_layers=num_layers,
                                bidirectional=True, input_size=embed_size)
        self.decoder = nn.Dense(2)

    def forward(self, inputs):
        # The shape of inputs is (batch size, number of words). Because LSTM
        # needs to use sequence as the first dimension, the input is
        # transformed and the word feature is then extracted. The output shape
        # is (number of words, batch size, word vector dimension).
        embeddings = self.embedding(inputs.T)
        # Since the input (embeddings) is the only argument passed into
        # rnn.LSTM, it only returns the hidden states of the last hidden layer
        # at different timestep (outputs). The shape of outputs is
        # (number of words, batch size, 2 * number of hidden units).
        outputs = self.encoder(embeddings)
        # Concatenate the hidden states of the initial timestep and final
        # timestep to use as the input of the fully connected layer. Its
        # shape is (batch size, 4 * number of hidden units)
        encoding = np.concatenate((outputs[0], outputs[-1]), axis=1)
        outs = self.decoder(encoding)
        return outs
```

Create a bidirectional recurrent neural network with two hidden layers.

```
embed_size, num_hiddens, num_layers, ctx = 100, 100, 2, d2l.try_all_gpus()
net = BiRNN(len(vocab), embed_size, num_hiddens, num_layers)
net.initialize(init.Xavier(), ctx=ctx)
```

Loading Pre-trained Word Vectors

Because the training dataset for sentiment classification is not very large, in order to deal with overfitting, we will directly use word vectors pre-trained on a larger corpus as the feature vectors of all words. Here, we load a 100-dimensional GloVe word vector for each word in the dictionary vocab.

```
glove_embedding = text.embedding.create(
    'glove', pretrained_file_name='glove.6B.100d.txt')
```

Query the word vectors that in our vocabulary.

```
embeds = glove_embedding.get_vecs_by_tokens(vocab.idx_to_token)
embeds.shape
```

```
(49339, 100)
```

Then, we will use these word vectors as feature vectors for each word in the reviews. Note that the dimensions of the pre-trained word vectors need to be consistent with the embedding layer output size embed_size in the created model. In addition, we no longer update these word vectors during training.

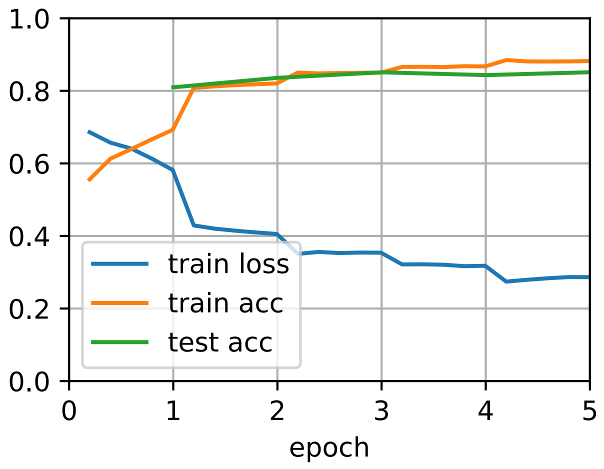
```
net.embedding.weight.set_data(embeds)
net.embedding.collect_params().setattr('grad_req', 'null')
```

Training and Evaluating the Model

Now, we can start training.

```
lr, num_epochs = 0.01, 5
trainer = gluon.Trainer(net.collect_params(), 'adam', {'learning_rate': lr})
loss = gluon.loss.SoftmaxCrossEntropyLoss()
d2l.train_ch13(net, train_iter, test_iter, loss, trainer, num_epochs, ctx)
```

```
loss 0.286, train acc 0.882, test acc 0.851
644.1 exampes/sec on [gpu(0), gpu(1)]
```



Finally, define the prediction function.

```
# Saved in the d2l package for later use
def predict_sentiment(net, vocab, sentence):
    sentence = np.array(vocab[sentence.split()], ctx=d2l.try_gpu())
    label = np.argmax(net(sentence.reshape(1, -1)), axis=1)
    return 'positive' if label == 1 else 'negative'
```

Then, use the trained model to classify the sentiments of two simple sentences.

```
predict_sentiment(net, vocab, 'this movie is so great')
```

```
'positive'
```

```
predict_sentiment(net, vocab, 'this movie is so bad')
```

```
'negative'
```

Summary

- Text classification transforms a sequence of text of indefinite length into a category of text. This is a downstream application of word embedding.
- We can apply pre-trained word vectors and recurrent neural networks to classify the emotions in a text.

Exercises

1. Increase the number of epochs. What accuracy rate can you achieve on the training and testing datasets? What about trying to re-tune other hyper-parameters?
2. Will using larger pre-trained word vectors, such as 300-dimensional GloVe word vectors, improve classification accuracy?
3. Can we improve the classification accuracy by using the spaCy word tokenization tool? You need to install spaCy: `pip install spacy` and install the English package: `python -m spacy download en`. In the code, first import spacy: `import spacy`. Then, load the spacy English package: `spacy_en = spacy.load('en')`. Finally, define the function `def tokenizer(text): return [tok.text for tok in spacy_en.tokenizer(text)]` and replace the original `tokenizer` function. It should be noted that GloVe's word vector uses "-" to connect each word when storing noun phrases. For example, the phrase "new york" is represented as "new-york" in GloVe. After using spaCy tokenization, "new york" may be stored as "new york".

14.10 Text Sentiment Classification: Using Convolutional Neural Networks (textCNN)

In Chapter 6, we explored how to process two-dimensional image data with two-dimensional convolutional neural networks. In the previous language models and text classification tasks, we treated text data as a time series with only one dimension, and naturally, we used recurrent neural networks to process such data. In fact, we can also treat text as a one-dimensional image, so that we can use one-dimensional convolutional neural networks to capture associations between adjacent words. This section describes a groundbreaking approach to applying convolutional neural networks to text analysis: textCNN (Kim, 2014). First, import the packages and modules required for the experiment.

```
import d2l
from mxnet import gluon, init, np, npx
from mxnet.contrib import text
from mxnet.gluon import nn
npx.set_np()

batch_size = 64
train_iter, test_iter, vocab = d2l.load_data_imdb(batch_size)
```

```
Downloading ../data/aclImdb_v1.tar.gz from http://ai.stanford.edu/~amaas/data/sentiment/
↪aclImdb_v1.tar.gz...
```

14.10.1 One-Dimensional Convolutional Layer

Before introducing the model, let's explain how a one-dimensional convolutional layer works. Like a two-dimensional convolutional layer, a one-dimensional convolutional layer uses a one-dimensional cross-correlation operation. In the one-dimensional cross-correlation operation, the convolution window starts from the leftmost side of the input array and slides on the input array from left to right successively. When the convolution window slides to a certain position, the input subarray in the window and kernel array are multiplied and summed by element to get the element at the corresponding location in the output array. As shown in Fig. 14.10.1, the input is a one-dimensional array with a width of 7 and the width of the kernel array is 2. As we can see, the output width is $7 - 2 + 1 = 6$ and the first element is obtained by performing multiplication by element on the leftmost input subarray with a width of 2 and kernel array and then summing the results.

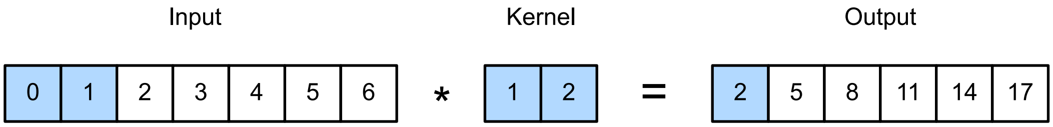


Fig. 14.10.1: One-dimensional cross-correlation operation. The shaded parts are the first output element as well as the input and kernel array elements used in its calculation: $0 \times 1 + 1 \times 2 = 2$.

Next, we implement one-dimensional cross-correlation in the `corr1d` function. It accepts the input array `X` and kernel array `K` and outputs the array `Y`.

```
def corr1d(X, K):
    w = K.shape[0]
    Y = np.zeros((X.shape[0] - w + 1))
    for i in range(Y.shape[0]):
        Y[i] = (X[i: i + w] * K).sum()
    return Y
```

Now, we will reproduce the results of the one-dimensional cross-correlation operation in Fig. 14.10.1.

```
X, K = np.array([0, 1, 2, 3, 4, 5, 6]), np.array([1, 2])
corr1d(X, K)
```

```
array([ 2.,  5.,  8., 11., 14., 17.])
```

The one-dimensional cross-correlation operation for multiple input channels is also similar to the two-dimensional cross-correlation operation for multiple input channels. On each channel, it performs the one-dimensional cross-correlation operation on the kernel and its corresponding input and adds the results of the channels to get the output. Fig. 14.10.2 shows a one-dimensional cross-correlation operation with three input channels.

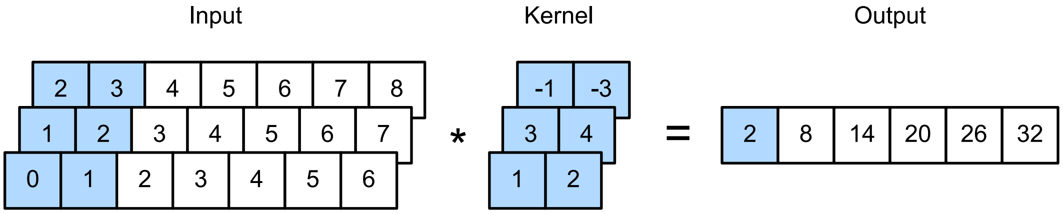


Fig. 14.10.2: One-dimensional cross-correlation operation with three input channels. The shaded parts are the first output element as well as the input and kernel array elements used in its calculation: $0 \times 1 + 1 \times 2 + 1 \times 3 + 2 \times 4 + 2 \times (-1) + 3 \times (-3) = 2$.

Now, we reproduce the results of the one-dimensional cross-correlation operation with multi-input channel in Fig. 14.10.2.

```
def corr1d_multi_in(X, K):
    # First, we traverse along the 0th dimension (channel dimension) of X and
    # K. Then, we add them together by using * to turn the result list into a
    # positional argument of the add_n function
    return sum(corr1d(x, k) for x, k in zip(X, K))

X = np.array([[0, 1, 2, 3, 4, 5, 6],
              [1, 2, 3, 4, 5, 6, 7],
              [2, 3, 4, 5, 6, 7, 8]])
K = np.array([[1, 2], [3, 4], [-1, -3]])
corr1d_multi_in(X, K)
```

```
array([ 2.,  8., 14., 20., 26., 32.])
```

The definition of a two-dimensional cross-correlation operation tells us that a one-dimensional cross-correlation operation with multiple input channels can be regarded as a two-dimensional cross-correlation operation with a single input channel. As shown in Fig. 14.10.3, we can also present the one-dimensional cross-correlation operation with multiple input channels in Fig. 14.10.2 as the equivalent two-dimensional cross-correlation operation with a single input channel. Here, the height of the kernel is equal to the height of the input.

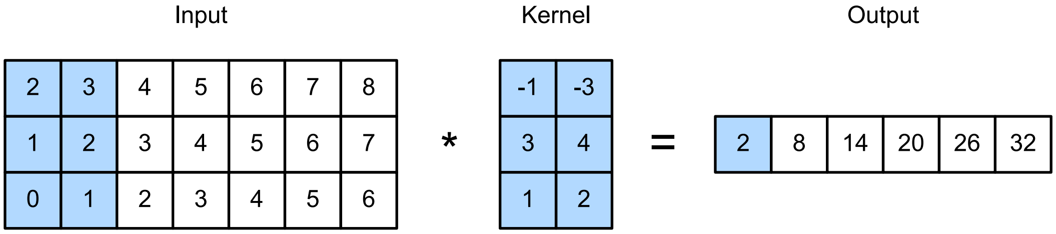


Fig. 14.10.3: Two-dimensional cross-correlation operation with a single input channel. The highlighted parts are the first output element and the input and kernel array elements used in its calculation: $2 \times (-1) + 3 \times (-3) + 1 \times 3 + 2 \times 4 + 0 \times 1 + 1 \times 2 = 2$.

Both the outputs in Fig. 14.10.1 and Fig. 14.10.2 have only one channel. We discussed how to specify multiple output channels in a two-dimensional convolutional layer in Section 6.4. Similarly, we can also specify multiple output channels in the one-dimensional convolutional layer to extend the model parameters in the convolutional layer.

14.10.2 Max-Over-Time Pooling Layer

Similarly, we have a one-dimensional pooling layer. The max-over-time pooling layer used in TextCNN actually corresponds to a one-dimensional global maximum pooling layer. Assuming that the input contains multiple channels, and each channel consists of values on different timesteps, the output of each channel will be the largest value of all timesteps in the channel. Therefore, the input of the max-over-time pooling layer can have different timesteps on each channel.

To improve computing performance, we often combine timing examples of different lengths into a minibatch and make the lengths of each timing example in the batch consistent by appending special characters (such as 0) to the end of shorter examples. Naturally, the added special characters have no intrinsic meaning. Because the main purpose of the max-over-time pooling layer is to capture the most important features of timing, it usually allows the model to be unaffected by the manually added characters.

14.10.3 The TextCNN Model

TextCNN mainly uses a one-dimensional convolutional layer and max-over-time pooling layer. Suppose the input text sequence consists of n words, and each word is represented by a d-dimension word vector. Then the input example has a width of n, a height of 1, and d input channels. The calculation of textCNN can be mainly divided into the following steps:

1. Define multiple one-dimensional convolution kernels and use them to perform convolution calculations on the inputs. Convolution kernels with different widths may capture the correlation of different numbers of adjacent words.
2. Perform max-over-time pooling on all output channels, and then concatenate the pooling output values of these channels in a vector.
3. The concatenated vector is transformed into the output for each category through the fully connected layer. A dropout layer can be used in this step to deal with overfitting.

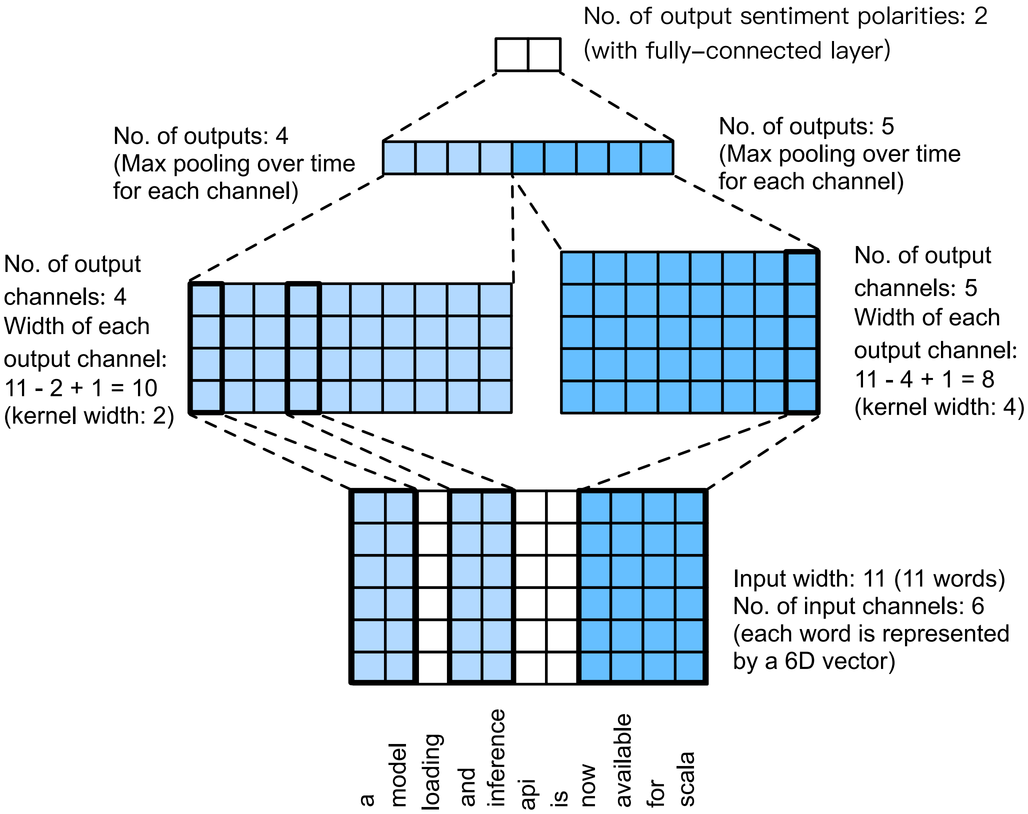


Fig. 14.10.4: TextCNN design.

Fig. 14.10.4 gives an example to illustrate the textCNN. The input here is a sentence with 11 words, with each word represented by a 6-dimensional word vector. Therefore, the input sequence has a width of 11 and 6 input channels. We assume there are two one-dimensional convolution kernels with widths of 2 and 4, and 4 and 5 output channels, respectively. Therefore, after one-dimensional convolution calculation, the width of the four output channels is $11 - 2 + 1 = 10$, while the width of the other five channels is $11 - 4 + 1 = 8$. Even though the width of each channel is different, we can still perform max-over-time pooling for each channel and concatenate the pooling outputs of the 9 channels into a 9-dimensional vector. Finally, we use a fully connected layer to transform the 9-dimensional vector into a 2-dimensional output: positive sentiment and negative sentiment predictions.

Next, we will implement a textCNN model. Compared with the previous section, in addition to replacing the recurrent neural network with a one-dimensional convolutional layer, here we use two embedding layers, one with a fixed weight and another that participates in training.

```
class TextCNN(nn.Block):
    def __init__(self, vocab_size, embed_size, kernel_sizes, num_channels,
                 **kwargs):
        super(TextCNN, self).__init__(**kwargs)
        self.embedding = nn.Embedding(vocab_size, embed_size)
        # The embedding layer does not participate in training
        self.constant_embedding = nn.Embedding(vocab_size, embed_size)
        self.dropout = nn.Dropout(0.5)
        self.decoder = nn.Dense(2)
        # The max-over-time pooling layer has no weight, so it can share an
```

(continues on next page)

(continued from previous page)

```
        # instance
        self.pool = nn.GlobalMaxPool1D()
        # Create multiple one-dimensional convolutional layers
        self.convs = nn.Sequential()
        for c, k in zip(num_channels, kernel_sizes):
            self.convs.add(nn.Conv1D(c, k, activation='relu'))

    def forward(self, inputs):
        # Concatenate the output of two embedding layers with shape of
        # (batch size, number of words, word vector dimension) by word vector
        embeddings = np.concatenate((
            self.embedding(inputs), self.constant_embedding(inputs)), axis=2)
        # According to the input format required by Conv1D, the word vector
        # dimension, that is, the channel dimension of the one-dimensional
        # convolutional layer, is transformed into the previous dimension
        embeddings = embeddings.transpose(0, 2, 1)
        # For each one-dimensional convolutional layer, after max-over-time
        # pooling, an ndarray with the shape of (batch size, channel size, 1)
        # can be obtained. Use the flatten function to remove the last
        # dimension and then concatenate on the channel dimension
        encoding = np.concatenate([
            np.squeeze(self.pool(conv(embeddings)), axis=-1)
            for conv in self.convs], axis=1)
        # After applying the dropout method, use a fully connected layer to
        # obtain the output
        outputs = self.decoder(self.dropout(encoding))
        return outputs
```

Create a TextCNN instance. It has 3 convolutional layers with kernel widths of 3, 4, and 5, all with 100 output channels.

```
embed_size, kernel_sizes, nums_channels = 100, [3, 4, 5], [100, 100, 100]
ctx = d2l.try_all_gpus()
net = TextCNN(len(vocab), embed_size, kernel_sizes, nums_channels)
net.initialize(init.Xavier(), ctx=ctx)
```

Load Pre-trained Word Vectors

As in the previous section, load pre-trained 100-dimensional GloVe word vectors and initialize the embedding layers `embedding` and `constant_embedding`. Here, the former participates in training while the latter has a fixed weight.

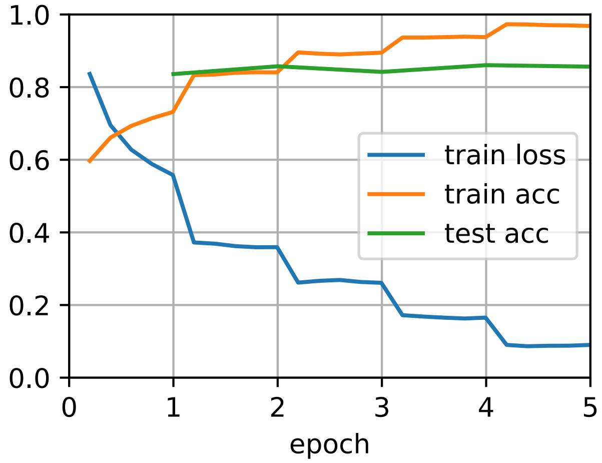
```
glove_embedding = text.embedding.create(
    'glove', pretrained_file_name='glove.6B.100d.txt')
embeds = glove_embedding.get_vecs_by_tokens(vocab.idx_to_token)
net.embedding.weight.set_data(embeds)
net.constant_embedding.weight.set_data(embeds)
net.constant_embedding.collect_params().setattr('grad_req', 'null')
```

Train and Evaluate the Model

Now we can train the model.

```
lr, num_epochs = 0.001, 5
trainer = gluon.Trainer(net.collect_params(), 'adam', {'learning_rate': lr})
loss = gluon.loss.SoftmaxCrossEntropyLoss()
d2l.train_ch13(net, train_iter, test_iter, loss, trainer, num_epochs, ctx)
```

```
loss 0.090, train acc 0.968, test acc 0.856
4205.6 exampes/sec on [gpu(0), gpu(1)]
```



Below, we use the trained model to classify sentiments of two simple sentences.

```
d2l.predict_sentiment(net, vocab, 'this movie is so great')
```

```
'positive'
```

```
d2l.predict_sentiment(net, vocab, 'this movie is so bad')
```

```
'negative'
```

Summary

- We can use one-dimensional convolution to process and analyze timing data.
- A one-dimensional cross-correlation operation with multiple input channels can be regarded as a two-dimensional cross-correlation operation with a single input channel.
- The input of the max-over-time pooling layer can have different numbers of timesteps on each channel.
- TextCNN mainly uses a one-dimensional convolutional layer and max-over-time pooling layer.

Exercises

1. Tune the hyper-parameters and compare the two sentiment analysis methods, using recurrent neural networks and using convolutional neural networks, as regards accuracy and operational efficiency.
2. Can you further improve the accuracy of the model on the test set by using the three methods introduced in the previous section: tuning hyper-parameters, using larger pre-trained word vectors, and using the spaCy word tokenization tool?
3. What other natural language processing tasks can you use textCNN for?

15 | Recommender Systems

Shuai Zhang (*Amazon*), **Aston Zhang** (*Amazon*), and **Yi Tay** (*Nanyang Technological University*)

Recommender systems are widely employed in industry and are ubiquitous in our daily lives. These systems are utilized in a number of areas such as online shopping sites (e.g., amazon.com), music/movie services site (e.g., Netflix and Spotify), mobile application stores (e.g., IOS app store and google play), online advertising, just to name a few.

The major goal of recommender systems is to help users discover relevant items such as movies to watch, text to read or products to buy, so as to create a delightful user experience. Moreover, recommender systems are among the most powerful machine learning systems that online retailers implement in order to drive incremental revenue. Recommender systems are replacements of search engines by reducing the efforts in proactive searches and surprising users with offers they never searched for. Many companies managed to position themselves ahead of their competitors with the help of more effective recommender systems. As such, recommender systems are central to not only our everyday lives but also highly indispensable in some industries.

In this chapter, we will cover the fundamentals and advancements of recommender systems, along with exploring some common fundamental techniques for building recommender systems with different data sources available and their implementations. Specifically, you will learn how to predict the rating a user might give to a prospective item, how to generate a recommendation list of items and how to predict the click-through rate from abundant features. These tasks are commonplace in real-world applications. By studying this chapter, you will get hands-on experience pertaining to solving real world recommendation problems with not only classical methods but the more advanced deep learning based models as well.

15.1 Overview of Recommender Systems

In the last decade, the Internet has evolved into a platform for large-scale online services, which profoundly changed the way we communicate, read news, buy products, and watch movies. In the meanwhile, the unprecedented number of items (we use the term *item* to refer to movies, news, books, and products.) offered online requires a system that can help us discover items that we preferred. Recommender systems are therefore powerful information filtering tools that can facilitate personalized services and provide tailored experience to individual users. In short, recommender systems play a pivotal role in utilizing the wealth of data available to make choices manageable. Nowadays, recommender systems are at the core of a number of online services providers such as Amazon, Netflix, and YouTube. Recall the example of Deep learning books recommended by Amazon in Fig. 1.3.3. The benefits of employing recommender systems are two-folds: On the one hand, it can largely reduce users' effort in finding items and alleviate the issue of information overload. On the other hand, it can add business value to online service providers and is an important source of revenue. This chapter will introduce the fundamental concepts, classic models and recent advances with deep learning in the field of recommender systems, together with implemented examples.

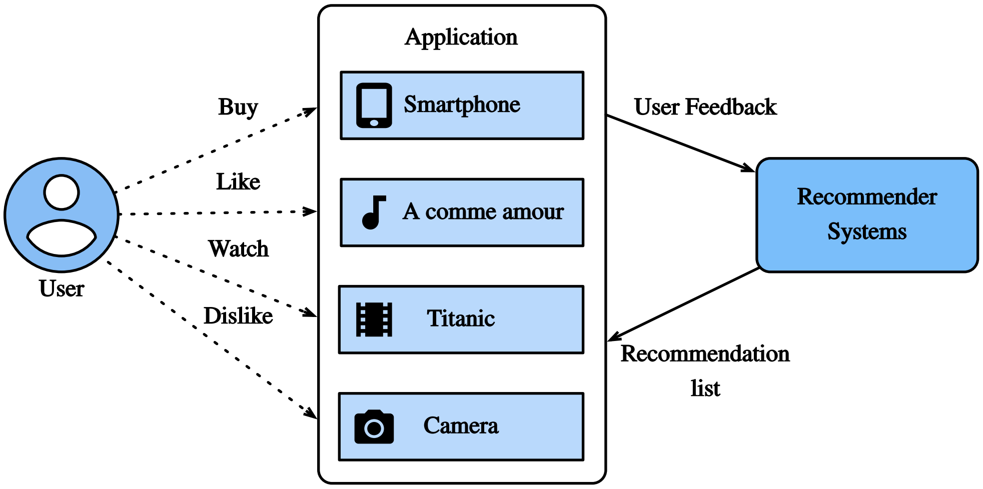


Fig. 15.1.1: Illustration of the Recommendation Process

15.1.1 Collaborative Filtering

We start the journey with the important concept in recommender systems—collaborative filtering (CF), which was first coined by the Tapestry system (Goldberg et al., 1992), referring to "people collaborate to help one another perform the filtering process in order to handle the large amounts of email and messages posted to newsgroups". This term has been enriched with more senses. In a broad sense, it is the process of filtering for information or patterns using techniques involving collaboration among multiple users, agents, and data sources. CF has many forms and numerous CF methods proposed since its advent.

Overall, CF techniques can be categorized into: memory-based CF, model-based CF, and their hybrid (Su & Khoshgoftaar, 2009). Representative memory-based CF techniques are nearest neighbor-based CF such as user-based CF and item-based CF (Sarwar et al., 2001). Latent factor models such as matrix factorization are examples of model-based CF. Memory-based CF has limitations in dealing with sparse and large-scale data since it computes the similarity values based on common items. Model-based methods become more popular with its better capability in dealing with sparsity and scalability. Many model-based CF approaches can be extended with neural networks, leading to more flexible and scalable models with the computation acceleration in deep learning (Zhang et al., 2019). In general, CF only uses the user-item interaction data to make predictions and recommendations. Besides CF, content-based and context-based recommender systems are also useful in incorporating the content descriptions of items/users and contextual signals such as timestamps and locations. Obviously, we may need to adjust the model types/structures when different input data is available.

15.1.2 Explicit Feedback and Implicit Feedback

To learn the preference of users, the system shall collect feedback from them. The feedback can be either explicit or implicit (Hu et al., 2008). For example, IMDB[229] collects star ratings ranging from one to ten stars for movies. YouTube provides the thumbs-up and thumbs-down buttons for users to show their preferences. It is apparent that gathering explicit feedback requires users to indicate their interests proactively. Nonetheless, explicit feedback is not always readily available as many users may be reluctant to rate products. Relatively speaking, implicit feedback is often readily available since it is mainly concerned with modeling implicit behavior such user clicks. As such, many recommender systems are centered on implicit feedback which indirectly reflects user's opinion through observing user behavior. There are diverse forms of implicit feedback including purchase history, browsing history, watches and even mouse movements. For example, a user that purchased many books by the same author probably likes that author. Note that implicit feedback is inherently noisy. We can only *guess* their preferences and true motives. A user watched a movie does not necessarily indicate a positive view of that movie.

15.1.3 Recommendation Tasks

A number of recommendation tasks have been investigated in the past decades. Based on the domain of applications, there are movies recommendation, news recommendations, point-of-interest recommendation (Ye et al., 2011) and so forth. It is also possible to differentiate the tasks based on the types of feedback and input data, for example, the rating prediction task aims to predict the explicit ratings. Top-n recommendation (item ranking) ranks all items for each user personally based on the implicit feedback. If time-stamp information is also included, we can build sequence-aware recommendation (Quadrana et al., 2018). Another popular task is called click-through rate prediction, which is also based on implicit feedback, but various categorical features can be utilized. Recommending for new users and recommending new items to existing users are called cold-start recommendation (Schein et al., 2002).

Summary

- Recommender systems are important for individual users and industries. Collaborative filtering is a key concept in recommendation.
- There are two types of feedbacks: implicit feedback and explicit feedback. A number recommendation tasks have been explored during the last decade.

Exercises

1. Can you explain how recommender systems influence your daily life?
2. What interesting recommendation tasks do you think can be investigated?

[229] https://www.imdb.com/

15.2 The MovieLens Dataset

There are a number of datasets that are available for recommendation research. Amongst them, the MovieLens[231] dataset is probably the one of the more popular ones. MovieLens is a non-commercial web-based movie recommender system. It is created in 1997 and run by GroupLens, a research lab at the University of Minnesota, in order to gather movie rating data for research purposes. MovieLens data has been critical for several research studies including personalized recommendation and social psychology.

15.2.1 Getting the Data

The MovieLens dataset is hosted by the GroupLens[232] website. Several versions are available. We will use the MovieLens 100K dataset (Herlocker et al., 1999). This dataset is comprised of $100,000$ ratings, ranging from 1 to 5 stars, from 943 users on 1682 movies. It has been cleaned up so that each user has rated at least 20 movies. Some simple demographic information such as age, gender, genres for the users and items are also available. We can download the ml-100k.zip[233] and extract the `u.data` file, which contains all the $100,000$ ratings in the csv format. There are many other files in the folder, a detailed description for each file can be found in the README[234] file of the dataset.

To begin with, let's import the packages required to run this section's experiments.

```
import d2l
from mxnet import gluon, np
import pandas as pd
import zipfile
```

Then, we download the MovieLens 100k dataset and load the interactions as `DataFrame`.

```
# Saved in the d2l package for later use
def read_data_ml100k(path="../data/", member="ml-100k/u.data",
                     names=['user_id', 'item_id', 'rating', 'timestamp'],
                     sep="\t"):
    fname = gluon.utils.download(
        'http://files.grouplens.org/datasets/movielens/ml-100k.zip',
        path=path)
    with zipfile.ZipFile(fname, 'r') as inzipfile:
        inzipfile.extract(member, path)
        data = pd.read_csv(path + member, sep, names=names, engine='python')
        num_users = data.user_id.unique().shape[0]
        num_items = data.item_id.unique().shape[0]
        return data, num_users, num_items
```

[231] https://movielens.org/
[232] https://grouplens.org/datasets/movielens/
[233] http://files.grouplens.org/datasets/movielens/ml-100k.zip
[234] http://files.grouplens.org/datasets/movielens/ml-100k-README.txt

15.2.2 Statistics of the Dataset

Let's load up the data and inspect the first five records manually. It is an effective way to learn the data structure and verify that they have been loaded properly.

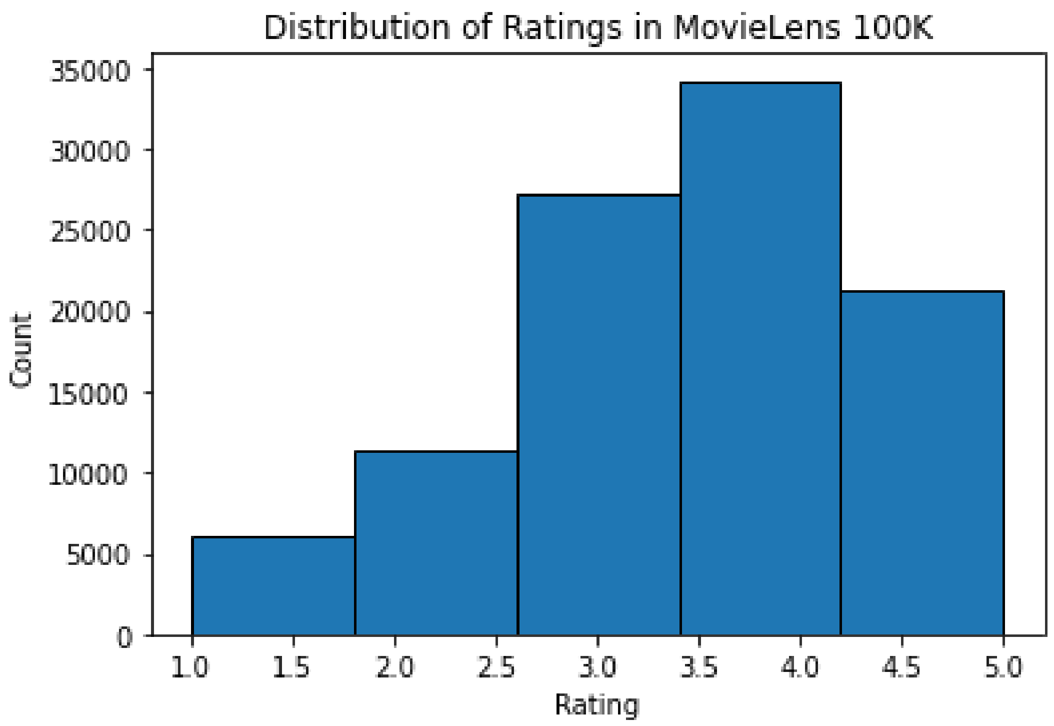
```
data, num_users, num_items = read_data_ml100k()
sparsity = 1 - len(data) / (num_users * num_items)
print('number of users: %d, number of items: %d.' % (num_users, num_items))
print('matrix sparsity: %f' % sparsity)
print(data.head(5))
```

```
number of users: 943, number of items: 1682.
matrix sparsity: 0.936953
   user_id  item_id  rating  timestamp
0      196      242       3  881250949
1      186      302       3  891717742
2       22      377       1  878887116
3      244       51       2  880606923
4      166      346       1  886397596
```

We can see that each line consists of four columns, including "user id" 1-943, "item id" 1-1682, "rating" 1-5 and "timestamp". We can construct an interaction matrix of size $n \times m$, where n and m are the number of users and the number of items respectively. This dataset only records the existing ratings, so we can also call it rating matrix and we will use interaction matrix and rating matrix interchangeably in case that the values of this matrix represent exact ratings. Most of the values in the rating matrix are unknown as users have not rated the majority of movies. We also show the sparsity of this dataset. The sparsity is defined as `1 - number of nonzero entries / ( number of users * number of items)`. Clearly, the interaction matrix is extremely sparse (i.e., sparsity = 93.695%). Real world datasets may suffer from a greater extent of sparsity and has been a long-standing challenge in building recommender systems. A viable solution is to use additional side information such as user/item features to alleviate the sparsity.

We then plot the distribution of the count of different ratings. As expected, it appears to be a normal distribution, with most ratings centered at 3-4.

```
d2l.plt.hist(data['rating'], bins=5, ec='black')
d2l.plt.xlabel("Rating")
d2l.plt.ylabel("Count")
d2l.plt.title("Distribution of Ratings in MovieLens 100K")
d2l.plt.show()
```



15.2.3 Splitting the dataset

We split the dataset into training and test sets. The following function provides two split modes including `random` and `seq-aware`. In the `random` mode, the function splits the 100k interactions randomly without considering timestamp and uses the 90% of the data as training samples and the rest 10% as test samples by default. In the `seq-aware` mode, we leave out the item that a user rated most recently for test, and users' historical interactions as training set. User historical interactions are sorted from oldest to newest based on timestamp. This mode will be used in the sequence-aware recommendation section.

```python
# Saved in the d2l package for later use
def split_data_ml100k(data, num_users, num_items,
                      split_mode="random", test_ratio=0.1):
    """Split the dataset in random mode or seq-aware mode."""
    if split_mode == "seq-aware":
        train_items, test_items, train_list = {}, {}, []
        for line in data.itertuples():
            u, i, rating, time = line[1], line[2], line[3], line[4]
            train_items.setdefault(u, []).append((u, i, rating, time))
            if u not in test_items or test_items[u][-1] < time:
                test_items[u] = (i, rating, time)
        for u in range(1, num_users + 1):
            train_list.extend(sorted(train_items[u], key=lambda k: k[3]))
        test_data = [(key, *value) for key, value in test_items.items()]
        train_data = [item for item in train_list if item not in test_data]
        train_data = pd.DataFrame(train_data)
        test_data = pd.DataFrame(test_data)
    else:
        mask = [True if x == 1 else False for x in np.random.uniform(
            0, 1, (len(data))) < 1 - test_ratio]
```

(continues on next page)

(continued from previous page)

```
        neg_mask = [not x for x in mask]
        train_data, test_data = data[mask], data[neg_mask]
    return train_data, test_data
```

Note that it is good practice to use a validation set in practice, apart from only a test set. However, we omit that for the sake of brevity. In this case, our test set can be regarded as our held-out validation set.

15.2.4 Loading the data

After dataset splitting, we will convert the training set and test set into lists and dictionaries/matrix for the sake of convenience. The following function reads the dataframe line by line and enumerates the index of users/items start from zero. The function then returns lists of users, items, ratings and a dictionary/matrix that records the interactions. We can specify the type of feedback to either `explicit` or `implicit`.

```
# Saved in the d2l package for later use
def load_data_ml100k(data, num_users, num_items, feedback="explicit"):
    users, items, scores = [], [], []
    inter = np.zeros((num_items, num_users)) if feedback == "explicit" else {}
    for line in data.itertuples():
        user_index, item_index = int(line[1] - 1), int(line[2] - 1)
        score = int(line[3]) if feedback == "explicit" else 1
        users.append(user_index)
        items.append(item_index)
        scores.append(score)
        if feedback == "implicit":
            inter.setdefault(user_index, []).append(item_index)
        else:
            inter[item_index, user_index] = score
    return users, items, scores, inter
```

Afterwards, we put the above steps together and it will be used in the next section. The results are wrapped with `Dataset` and `DataLoader`. Note that the `last_batch` of `DataLoader` for training data is set to the `rollover` mode (The remaining samples are rolled over to the next epoch.) and orders are shuffled.

```
# Saved in the d2l package for later use
def split_and_load_ml100k(split_mode="seq-aware", feedback="explicit",
                          test_ratio=0.1, batch_size=256):
    data, num_users, num_items = read_data_ml100k()
    train_data, test_data = split_data_ml100k(
        data, num_users, num_items, split_mode, test_ratio)
    train_u, train_i, train_r, _ = load_data_ml100k(
        train_data, num_users, num_items, feedback)
    test_u, test_i, test_r, _ = load_data_ml100k(
        test_data, num_users, num_items, feedback)
    train_set = gluon.data.ArrayDataset(
        np.array(train_u), np.array(train_i), np.array(train_r))
    test_set = gluon.data.ArrayDataset(
        np.array(test_u), np.array(test_i), np.array(test_r))
```

(continues on next page)

(continued from previous page)

```
    train_iter = gluon.data.DataLoader(
        train_set, shuffle=True, last_batch="rollover",
        batch_size=batch_size)
    test_iter = gluon.data.DataLoader(
        test_set, batch_size=batch_size)
    return num_users, num_items, train_iter, test_iter
```

Summary

- MovieLens datasets are widely used for recommendation research. It is public available and free to use.
- We define functions to download and preprocess the MovieLens 100k dataset for further use in later sections.

Exercises

- What other similar recommendation datasets can you find?
- Go through the https://movielens.org/ site for more information about MovieLens.

15.3 Matrix Factorization

Matrix Factorization (Koren et al., 2009) is a well-established algorithm in the recommender systems literature. The first version of matrix factorization model is proposed by Simon Funk in a famous blog post[236] in which he described the idea of factorizing the interaction matrix. It then became widely known due to the Netflix contest which was held in 2006. At that time, Netflix, a media-streaming and video-rental company, announced a contest to improve its recommender system performance. The best team that can improve on the Netflix baseline, i.e., Cinematch), by 10 percent would win a one million USD prize. As such, this contest attracted a lot of attention to the field of recommender system research. Subsequently, the grand prize was won by the BellKor's Pragmatic Chaos team, a combined team of BellKor, Pragmatic Theory, and BigChaos (you do not need to worry about these algorithms now). Although the final score was the result of an ensemble solution (i.e., a combination of many algorithms), the matrix factorization algorithm played a critical role in the final blend. The technical report the Netflix Grand Prize solution (Toscher et al., 2009) provides a detailed introduction to the adopted model. In this section, we will dive into the details of the matrix factorization model and its implementation.

[236] https://sifter.org/~simon/journal/20061211.html

15.3.1 The Matrix Factorization Model

Matrix factorization is a class of collaborative filtering models. Specifically, the model factorizes the user-item interaction matrix (e.g., rating matrix) into the product of two lower-rank matrices, capturing the low-rank structure of the user-item interactions.

Let $\mathbf{R} \in \mathbb{R}^{m\times n}$ denote the interaction matrix with m users and n items, and the values of $\mathbf{R}$ represent explicit ratings. The user-item interaction will be factorized into a user latent matrix $\mathbf{P} \in \mathbb{R}^{m\times k}$ and an item latent matrix $\mathbf{Q} \in \mathbb{R}^{n\times k}$, where $k \ll m, n$, is the latent factor size. Let $\mathbf{p}_u$ denote the $u^\textrm{th}$ row of $\mathbf{P}$ and $\mathbf{q}_i$ denote the $i^\textrm{th}$ row of $\mathbf{Q}$. For a given item i, the elements of $\mathbf{q}_i$ measure the extent to which the item possesses those characteristics such as the genres and languages of a movie. For a given user u, the elements of $\mathbf{p}_u$ measure the extent of interest the user has in items' corresponding characteristics. These latent factors might measure obvious dimensions as mentioned in those examples or are completely uninterpretable. The predicted ratings can be estimated by

$$\hat{\mathbf{R}} = \mathbf{P}\mathbf{Q}^\top \tag{15.3.1}$$

where $\hat{\mathbf{R}} \in \mathbb{R}^{m\times n}$ is the predicted rating matrix which has the same shape as $\mathbf{R}$. One major problem of this prediction rule is that users/items biases can not be modeled. For example, some users tend to give higher ratings or some items always get lower ratings due to poorer quality. These biases are commonplace in real-world applications. To capture these biases, user specific and item specific bias terms are introduced. Specifically, the predicted rating user u gives to item i is calculated by

$$\hat{\mathbf{R}}_{ui} = \mathbf{p}_u\mathbf{q}^\top_i + b_u + b_i \tag{15.3.2}$$

Then, we train the matrix factorization model by minimizing the mean squared error between predicted rating scores and real rating scores. The objective function is defined as follows:

$$\underset{\mathbf{P}, \mathbf{Q}, b}{\mathrm{argmin}} \sum_{(u, i) \in \mathcal{K}} \| \mathbf{R}_{ui} - \hat{\mathbf{R}}_{ui} \|^2 + \lambda (\| \mathbf{P} \|^2_F + \| \mathbf{Q} \|^2_F + b_u^2 + b_i^2) \tag{15.3.3}$$

where λ denotes the regularization rate. The regularizing term $\lambda (\| \mathbf{P} \|^2_F + \| \mathbf{Q} \|^2_F + b_u^2 + b_i^2)$ is used to avoid over-fitting by penalizing the magnitude of the parameters. The (u, i) pairs for which $\mathbf{R}_{ui}$ is known are stored in the set $\mathcal{K}=\{(u, i) \mid \mathbf{R}_{ui} \textrm{ is known}\}$. The model parameters can be learned with an optimization algorithm, such as Stochastic Gradient Descent and Adam.

An intuitive illustration of the matrix factorization model is shown below:

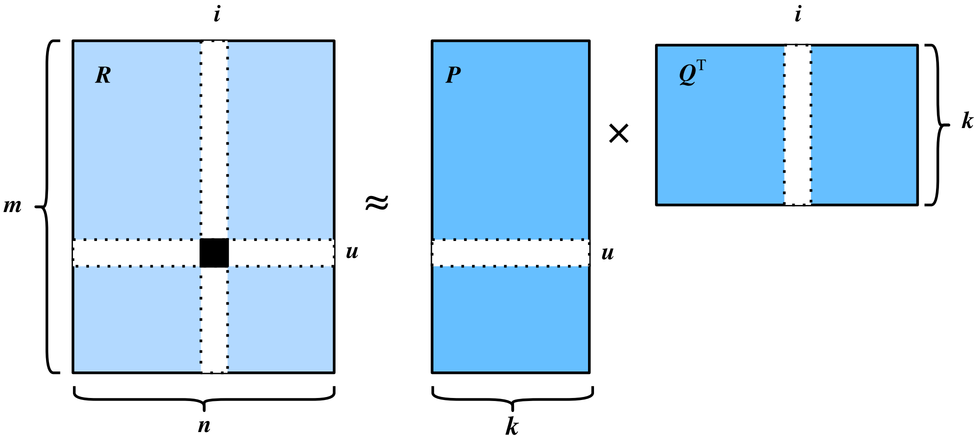


Fig. 15.3.1: Illustration of matrix factorization model

In the rest of this section, we will explain the implementation of matrix factorization and train the model on the MovieLens dataset.

```
import d2l
from mxnet import autograd, gluon, np, npx
from mxnet.gluon import nn
import mxnet as mx
npx.set_np()
```

15.3.2 Model Implementation

First, we implement the matrix factorization model described above. The user and item latent factors can be created with the `nn.Embedding`. The `input_dim` is the number of items/users and the (`output_dim`) is the dimension of the latent factors (k). We can also use `nn.Embedding` to create the user/item biases by setting the `output_dim` to one. In the `forward` function, user and item ids are used to look up the embeddings.

```
class MF(nn.Block):
    def __init__(self, num_factors, num_users, num_items, **kwargs):
        super(MF, self).__init__(**kwargs)
        self.P = nn.Embedding(input_dim=num_users, output_dim=num_factors)
        self.Q = nn.Embedding(input_dim=num_items, output_dim=num_factors)
        self.user_bias = nn.Embedding(num_users, 1)
        self.item_bias = nn.Embedding(num_items, 1)

    def forward(self, user_id, item_id):
        P_u = self.P(user_id)
        Q_i = self.Q(item_id)
        b_u = self.user_bias(user_id)
        b_i = self.item_bias(item_id)
        outputs = (P_u * Q_i).sum(axis=1) + np.squeeze(b_u) + np.squeeze(b_i)
        return outputs.flatten()
```

15.3.3 Evaluation Measures

We then implement the RMSE (root-mean-square error) measure, which is commonly used to measure the differences between rating scores predicted by the model and the actually observed ratings (ground truth) (Gunawardana & Shani, 2015). RMSE is defined as:

$$\text{RMSE} = \sqrt{\frac{1}{|\mathcal{T}|}\sum_{(u,i)\in\mathcal{T}}(\mathbf{R}_{ui} - \hat{\mathbf{R}}_{ui})^2} \tag{15.3.4}$$

where $\mathcal{T}$ is the set consisting of pairs of users and items that you want to evaluate on. $|\mathcal{T}|$ is the size of this set. We can use the RMSE function provided by `mx.metric`.

```
def evaluator(net, test_iter, ctx):
    rmse = mx.metric.RMSE()  # Get the RMSE
    rmse_list = []
    for idx, (users, items, ratings) in enumerate(test_iter):
        u = gluon.utils.split_and_load(users, ctx, even_split=False)
```

(continues on next page)

(continued from previous page)

```
        i = gluon.utils.split_and_load(items, ctx, even_split=False)
        r_ui = gluon.utils.split_and_load(ratings, ctx, even_split=False)
        r_hat = [net(u, i) for u, i in zip(u, i)]
        rmse.update(labels=r_ui, preds=r_hat)
        rmse_list.append(rmse.get()[1])
    return float(np.mean(np.array(rmse_list)))
```

15.3.4 Training and Evaluating the Model

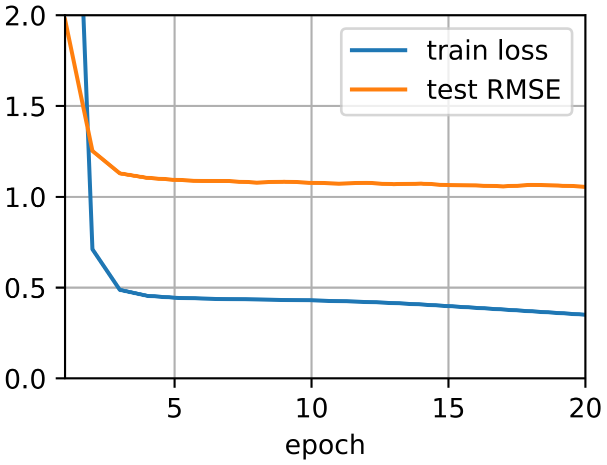
In the training function, we adopt the L_2 loss with weight decay. The weight decay mechanism has the same effect as the L_2 regularization.

```
# Saved in the d2l package for later use
def train_recsys_rating(net, train_iter, test_iter, loss, trainer, num_epochs,
                        ctx_list=d2l.try_all_gpus(), evaluator=None,
                        **kwargs):
    timer = d2l.Timer()
    animator = d2l.Animator(xlabel='epoch', xlim=[1, num_epochs], ylim=[0, 2],
                            legend=['train loss', 'test RMSE'])
    for epoch in range(num_epochs):
        metric, l = d2l.Accumulator(3), 0.
        for i, values in enumerate(train_iter):
            timer.start()
            input_data = []
            values = values if isinstance(values, list) else [values]
            for v in values:
                input_data.append(gluon.utils.split_and_load(v, ctx_list))
            train_feat = input_data[0:-1] if len(values) > 1 else input_data
            train_label = input_data[-1]
            with autograd.record():
                preds = [net(*t) for t in zip(*train_feat)]
                ls = [loss(p, s) for p, s in zip(preds, train_label)]
            [l.backward() for l in ls]
            l += sum([l.asnumpy() for l in ls]).mean() / len(ctx_list)
            trainer.step(values[0].shape[0])
            metric.add(l, values[0].shape[0], values[0].size)
            timer.stop()
        if len(kwargs) > 0:  # it will be used in section AutoRec.
            test_rmse = evaluator(net, test_iter, kwargs['inter_mat'],
                                  ctx_list)
        else:
            test_rmse = evaluator(net, test_iter, ctx_list)
        train_l = l / (i + 1)
        animator.add(epoch + 1, (train_l, test_rmse))
    print('train loss %.3f, test RMSE %.3f'
          % (metric[0] / metric[1], test_rmse))
    print('%.1f examples/sec on %s'
          % (metric[2] * num_epochs / timer.sum(), ctx_list))
```

Finally, let's put all things together and train the model. Here, we set the latent factor dimension to 30.

```
ctx = d2l.try_all_gpus()
num_users, num_items, train_iter, test_iter = d2l.split_and_load_ml100k(
    test_ratio=0.1, batch_size=512)
net = MF(30, num_users, num_items)
net.initialize(ctx=ctx, force_reinit=True, init=mx.init.Normal(0.01))
lr, num_epochs, wd, optimizer = 0.002, 20, 1e-5, 'adam'
loss = gluon.loss.L2Loss()
trainer = gluon.Trainer(net.collect_params(), optimizer,
                        {"learning_rate": lr, 'wd': wd})
train_recsys_rating(net, train_iter, test_iter, loss, trainer, num_epochs,
                    ctx, evaluator)
```

```
train loss 0.066, test RMSE 1.055
78035.0 examples/sec on [gpu(0), gpu(1)]
```



Below, we use the trained model to predict the rating that a user (ID 20) might give to an item (ID 30).

```
scores = net(np.array([20], dtype='int', ctx=d2l.try_gpu()),
             np.array([30], dtype='int', ctx=d2l.try_gpu()))
scores
```

```
array([3.1662967], ctx=gpu(0))
```

Summary

- The matrix factorization model is widely used in recommender systems. It can be used to predict ratings that a user might give to an item.
- We can implement and train matrix factorization for recommender systems.

Exercise

- Vary the size of latent factors. How does the size of latent factors influence the model performance?
- Try different optimizers, learning rates, and weight decay rates.
- Check the predicted rating scores of other users for a specific movie.

15.4 AutoRec: Rating Prediction with Autoencoders

Although the matrix factorization model achieves decent performance on the rating prediction task, it is essentially a linear model. Thus, such models are not capable of capturing complex nonlinear and intricate relationships that may be predictive of users' preferences. In this section, we introduce a nonlinear neural network collaborative filtering model, AutoRec (Sedhain et al., 2015). It identifies collaborative filtering (CF) with an autoencoder architecture and aims to integrate nonlinear transformations into CF on the basis of explicit feedback. Neural networks have been proven to be capable of approximating any continuous function, making it suitable to address the limitation of matrix factorization and enrich the expressiveness of matrix factorization.

On one hand, AutoRec has the same structure as an autoencoder which consists of an input layer, a hidden layer, and a reconstruction (output) layer. An autoencoder is a neural network that learns to copy its input to its output in order to code the inputs into the hidden (and usually low-dimensional) representations. In AutoRec, instead of explicitly embedding users/items into low-dimensional space, it uses the column/row of the interaction matrix as the input, then reconstructs the interaction matrix in the output layer.

On the other hand, AutoRec differs from a traditional autoencoder: rather than learning the hidden representations, AutoRec focuses on learning/reconstructing the output layer. It uses a partially observed interaction matrix as the input, aiming to reconstruct a completed rating matrix. In the meantime, the missing entries of the input are filled in the output layer via reconstruction for the purpose of recommendation.

There are two variants of AutoRec: user-based and item-based. For brevity, here we only introduce the item-based AutoRec. User-based AutoRec can be derived accordingly.

15.4.1 Model

Let $\mathbf{R}_{*i}$ denote the i^th column of the rating matrix, where unknown ratings are set to zeros by default. The neural architecture is defined as:

$$h(\mathbf{R}_{*i}) = f(\mathbf{W} \cdot g(\mathbf{V}\mathbf{R}_{*i} + \mu) + b) \tag{15.4.1}$$

where $f(\cdot)$ and $g(\cdot)$ represent activation functions, $\mathbf{W}$ and $\mathbf{V}$ are weight matrices, μ and b are biases. Let $h(\cdot)$ denote the whole network of AutoRec. The output $h(\mathbf{R}_{*i})$ is the reconstruction of the i^th column of the rating matrix.

The following objective function aims to minimize the reconstruction error:

$$\underset{\mathbf{W},\mathbf{V},\mu,b}{\operatorname{argmin}} \sum_{i=1}^{M} \| \mathbf{R}_{*i} - h(\mathbf{R}_{*i}) \|_{\mathcal{O}}^2 + \lambda(\|\mathbf{W}\|_F^2 + \|\mathbf{V}\|_F^2) \tag{15.4.2}$$

where $\| \cdot \|_{\mathcal{O}}$ means only the contribution of observed ratings are considered, that is, only weights that are associated with observed inputs are updated during back-propagation.

```
import d2l
from mxnet import autograd, gluon, np, npx
from mxnet.gluon import nn
import mxnet as mx
import sys
npx.set_np()
```

15.4.2 Implementing the Model

A typical autoencoder consists of an encoder and a decoder. The encoder projects the input to hidden representations and the decoder maps the hidden layer to the reconstruction layer. We follow this practice and create the encoder and decoder with dense layers. The activation of encoder is set to `sigmoid` by default and no activation is applied for decoder. Dropout is included after the encoding transformation to reduce over-fitting. The gradients of unobserved inputs are masked out to ensure that only observed ratings contribute to the model learning process.

```
class AutoRec(nn.Block):
    def __init__(self, num_hidden, num_users, dropout_rate=0.05):
        super(AutoRec, self).__init__()
        self.encoder = gluon.nn.Dense(num_hidden, activation='sigmoid',
                                      use_bias=True)
        self.decoder = gluon.nn.Dense(num_users, use_bias=True)
        self.dropout_layer = gluon.nn.Dropout(dropout_rate)

    def forward(self, input):
        hidden = self.dropout_layer(self.encoder(input))
        pred = self.decoder(hidden)
        if autograd.is_training():  # mask the gradient during training.
            return pred * np.sign(input)
        else:
            return pred
```

15.4.3 Reimplementing the Evaluator

Since the input and output have been changed, we need to reimplement the evaluation function, while we still use RMSE as the accuracy measure.

```
def evaluator(network, inter_matrix, test_data, ctx):
    scores = []
    for values in inter_matrix:
        feat = gluon.utils.split_and_load(values, ctx, even_split=False)
        scores.extend([network(i).asnumpy() for i in feat])
```

(continues on next page)

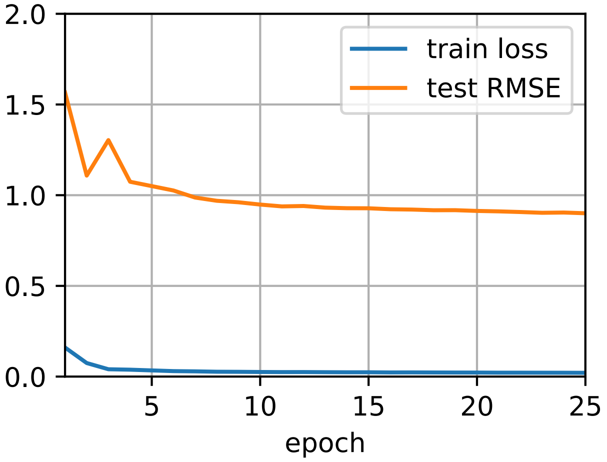
```
    recons = np.array([item for sublist in scores for item in sublist])
    # Calculate the test RMSE.
    rmse = np.sqrt(np.sum(np.square(test_data - np.sign(test_data) * recons))
                   / np.sum(np.sign(test_data)))
    return float(rmse)
```

15.4.4 Training and Evaluating the Model

Now, let's train and evaluate AutoRec on the MovieLens dataset. We can clearly see that the test RMSE is lower than the matrix factorization model, confirming the effectiveness of neural networks in the rating prediction task.

```
ctx = d2l.try_all_gpus()
# Load the MovieLens 100K dataset
df, num_users, num_items = d2l.read_data_ml100k()
train_data, test_data = d2l.split_data_ml100k(df, num_users, num_items)
_, _, _, train_inter_mat = d2l.load_data_ml100k(train_data, num_users,
                                                num_items)
_, _, _, test_inter_mat = d2l.load_data_ml100k(test_data, num_users,
                                               num_items)
num_workers = 0 if sys.platform.startswith("win") else 4
train_iter = gluon.data.DataLoader(train_inter_mat, shuffle=True,
                                   last_batch="rollover", batch_size=256,
                                   num_workers=num_workers)
test_iter = gluon.data.DataLoader(np.array(train_inter_mat), shuffle=False,
                                  last_batch="keep", batch_size=1024,
                                  num_workers=num_workers)
# Model initialization, training, and evaluation
net = AutoRec(500, num_users)
net.initialize(ctx=ctx, force_reinit=True, init=mx.init.Normal(0.01))
lr, num_epochs, wd, optimizer = 0.002, 25, 1e-5, 'adam'
loss = gluon.loss.L2Loss()
trainer = gluon.Trainer(net.collect_params(), optimizer,
                        {"learning_rate": lr, 'wd': wd})
d2l.train_recsys_rating(net, train_iter, test_iter, loss, trainer, num_epochs,
                        ctx, evaluator, inter_mat=test_inter_mat)
```

```
train loss 0.000, test RMSE 0.900
45418222.5 examples/sec on [gpu(0), gpu(1)]
```



Summary

- We can frame the matrix factorization algorithm with autoencoders, while integrating nonlinear layers and dropout regularization.
- Experiments on the MovieLens 100K dataset show that AutoRec achieves superior performance than matrix factorization.

Exercises

- Vary the hidden dimension of AutoRec to see its impact on the model performance.
- Try to add more hidden layers. Is it helpful to improve the model performance?
- Can you find a better combination of decoder and encoder activation functions?

15.5 Personalized Ranking for Recommender Systems

In the former sections, only explicit feedback was considered and models were trained and tested on observed ratings only. There are two demerits of such methods: First, most feedback is not explicit but implicit in real-world scenarios, and explicit feedback can be more expensive to collect. Second, non-observed user-item pairs which may be predictive for users' interests are totally ignored, making these methods unsuitable for cases where ratings are not missing at random but because of users' preferences. Non-observed user-item pairs are a mixture of real negative feedback (users are not interested in the items) and missing values (the user might interact with the items in the future). We simply ignore the non-observed pairs in matrix factorization and AutoRec. Clearly, these models are incapable of distinguishing between observed and non-observed pairs and are usually not suitable for personalized ranking tasks.

To this end, a class of recommendation models targeting at generating ranked recommendation lists from implicit feedback have gained popularity. In general, personalized ranking models can be optimized with pointwise, pairwise or Listwise approaches. Pointwise approaches considers a single interaction at a time and train a classifier/regressor to predict individual preferences. Matrix factorization and AutoRec are optimized with pointwise objectives. Pairwise approaches consider a pair of items for each user and aim to approximate the optimal ordering for that pair. Usually, pairwise approaches are more suitable for the ranking task because predicting relative order is reminiscent to the nature of ranking. Listwise approaches approximate the ordering of the entire list of items, for example, direct optimizing the ranking measures such as Normalized Discounted Cumulative Gain (NDCG[239]). However, listwise approaches are more complex and compute-intensive than pointwise or pairwise approaches. In this section, we will introduce two pairwise objectives/losses, Bayesian Personalized Ranking loss and Hinge loss, and their respective implementations.

15.5.1 Bayesian Personalized Ranking Loss and its Implementation

Bayesian personalized ranking (BPR) (Rendle et al., 2009) is a pairwise personalized ranking loss that is derived from the maximum posterior estimator. It has been widely used in many existing recommendation models. The training data of BPR consists of both positive and negative pairs (missing values). It assumes that the user prefers the positive item over all other non-observed items.

In formal, the training data is constructed by tuples in the form of (u, i, j), which represents that the user u prefers the item i over the item j. The Bayesian formulation of BPR which aims to maximize the posterior probability is given below:

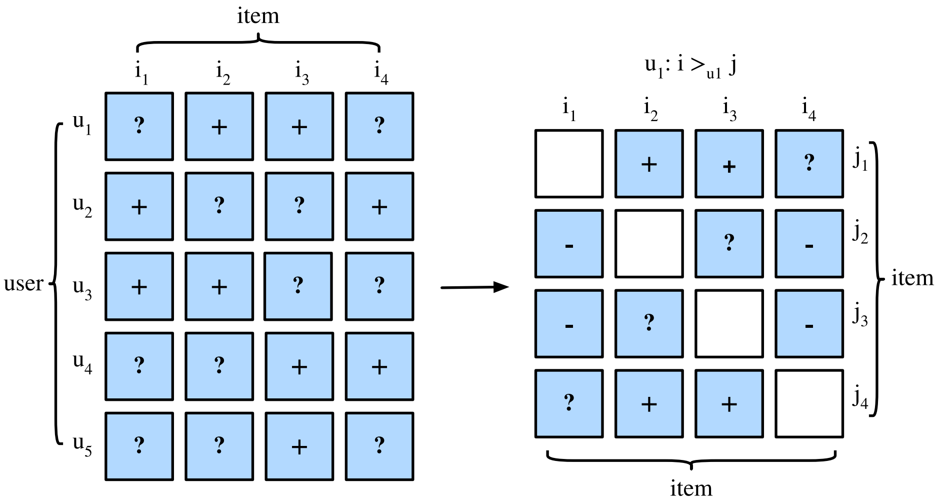
$$p(\Theta \mid >_u) \propto p(>_u \mid \Theta) p(\Theta) \tag{15.5.1}$$

Where Θ represents the parameters of an arbitrary recommendation model, $>_u$ represents the desired personalized total ranking of all items for user u. We can formulate the maximum posterior estimator to derive the generic optimization criterion for the personalized ranking task.

$$\begin{aligned}
\text{BPR-OPT} : &= \ln p(\Theta \mid >_u) \\
&= \ln p(>_u \mid \Theta) p(\Theta) \\
&= \ln \prod_{(u,i,j \in D)} \sigma(\hat{y}_{ui} - \hat{y}_{uj}) p(\Theta) \\
&= \sum_{(u,i,j \in D)} \ln \sigma(\hat{y}_{ui} - \hat{y}_{uj}) + \ln p(\Theta) \\
&= \sum_{(u,i,j \in D)} \ln \sigma(\hat{y}_{ui} - \hat{y}_{uj}) - \lambda_\Theta \|\Theta\|^2
\end{aligned} \tag{15.5.2}$$

where $D := \{(u, i, j) \mid i \in I^+_u \wedge j \in I \backslash I^+_u\}$ is the training set, with I^+_u denoting the items the user u liked, I denoting all items, and $I \backslash I^+_u$ indicating all other items excluding items the user liked. $\hat{y}_{ui}$ and $\hat{y}_{uj}$ are the predicted scores of the user u to item i and j, respectively. The prior $p(\Theta)$ is a normal distribution with zero mean and variance-covariance matrix Σ_Θ. Here, we let $\Sigma_\Theta = \lambda_\Theta I$.

[239] https://en.wikipedia.org/wiki/Discounted_cumulative_gain



We will implement the base class `mxnet.gluon.loss.Loss` and override the `forward` method to construct the Bayesian personalized ranking loss. We begin by importing the Loss class and the np module.

```
from mxnet import gluon, np, npx
npx.set_np()
```

The implementation of BPR loss is as follows.

```
# Saved in the d2l package for later use
class BPRLoss(gluon.loss.Loss):
    def __init__(self, weight=None, batch_axis=0, **kwargs):
        super(BPRLoss, self).__init__(weight=None, batch_axis=0, **kwargs)

    def forward(self, positive, negative):
        distances = positive - negative
        loss = - np.sum(np.log(npx.sigmoid(distances)), 0, keepdims=True)
        return loss
```

15.5.2 Hinge Loss and its Implementation

The Hinge loss for ranking has different form to the hinge loss[240] provided within the gluon library that is often used in classifiers such as SVMs. The loss used for ranking in recommender systems has the following form.

$$\sum_{(u,i,j\in D)} (\max(m - \hat{y}_{ui} + \hat{y}_{uj}), 0) \tag{15.5.3}$$

where m is the safety margin size. It aims to push negative items away from positive items. Similar to BPR, it aims to optimize for relevant distance between positive and negative samples instead of absolute outputs, making it well suited to recommender systems.

```
# Saved in the d2l package for later use
class HingeLossbRec(gluon.loss.Loss):
```

(continues on next page)

[240] https://mxnet.incubator.apache.org/api/python/gluon/loss.html#mxnet.gluon.loss.HingeLoss

(continued from previous page)

```
    def __init__(self, weight=None, batch_axis=0, **kwargs):
        super(HingeLossbRec, self).__init__(weight=None, batch_axis=0,
                                            **kwargs)

    def forward(self, positive, negative, margin=1):
        distances = positive - negative
        loss = np.sum(np.maximum(- distances + margin, 0))
        return loss
```

These two losses are interchangeable for personalized ranking in recommendation.

Summary

- There are three types of ranking losses available for the personalized ranking task in recommender systems, namely, pointwise, pairwise and listwise methods.
- The two pairwise loses, Bayesian personalized ranking loss and hinge loss, can be used interchangeably.

Exercises

- Are there any variants of BPR and hinge loss available?
- Can you find any recommendation models that use BPR or hinge loss?

15.6 Neural Collaborative Filtering for Personalized Ranking

This section moves beyond explicit feedback, introducing the neural collaborative filtering (NCF) framework for recommendation with implicit feedback. Implicit feedback is pervasive in recommender systems. Actions such as Clicks, buys, and watches are common implicit feedback which are easy to collect and indicative of users' preferences. The model we will introduce, titled NeuMF (He et al., 2017b), short for neural matrix factorization, aims to address the personalized ranking task with implicit feedback. This model leverages the flexibility and non-linearity of neural networks to replace dot products of matrix factorization, aiming at enhancing the model expressiveness. In specific, this model is structured with two subnetworks including generalized matrix factorization (GMF) and multilayer perceptron (MLP) and models the interactions from two pathways instead of simple inner products. The outputs of these two networks are concatenated for the final prediction scores calculation. Unlike the rating prediction task in AutoRec, this model generates a ranked recommendation list to each user based on the implicit feedback. We will use the personalized ranking loss introduced in the last section to train this model.

15.6.1 The NeuMF model

As aforementioned, NeuMF fuses two subnetworks. The GMF is a generic neural network version of matrix factorization where the input is the elementwise product of user and item latent factors. It consists of two neural layers:

$$\begin{aligned} \mathbf{x} &= \mathbf{p}_u \odot \mathbf{q}_i \\ \hat{y}_{ui} &= \alpha(\mathbf{h}^\top \mathbf{x}), \end{aligned} \tag{15.6.1}$$

where $\odot$ denotes the Hadamard product of vectors. $\mathbf{P} \in \mathbb{R}^{m \times k}$ and $\mathbf{Q} \in \mathbb{R}^{n \times k}$ corespond to user and item latent matrix respectively. $\mathbf{p}_u \in \mathbb{R}^k$ is the u^th row of P and $\mathbf{q}_i \in \mathbb{R}^k$ is the i^th row of Q. α and h denote the activation function and weight of the output layer. $\hat{y}_{ui}$ is the prediction score of the user u might give to the item i.

Another component of this model is MLP. To enrich model flexibility, the MLP subnetwork does not share user and item embeddings with GMF. It uses the concatenation of user and item embeddings as input. With the complicated connections and nonlinear transformations, it is capable of eastimating the intricate interactions between users and items. More precisely, the MLP subnetwork is defined as:

$$\begin{aligned} z^{(1)} &= \phi_1(\mathbf{U}_u, \mathbf{V}_i) = [\mathbf{U}_u, \mathbf{V}_i] \\ \phi^{(2)}(z^{(1)}) &= \alpha^1(\mathbf{W}^{(2)} z^{(1)} + b^{(2)}) \\ &\ldots \\ \phi^{(L)}(z^{(L-1)}) &= \alpha^L(\mathbf{W}^{(L)} z^{(L-1)} + b^{(L)})) \\ \hat{y}_{ui} &= \alpha(\mathbf{h}^\top \phi^L(z^{(L)})) \end{aligned} \tag{15.6.2}$$

where $\mathbf{W}^*, \mathbf{b}^*$ and α^* denote the weight matrix, bias vector, and activation function. ϕ^* denotes the function of the corresponding layer. $\mathbf{z}^*$ denotes the output of corresponding layer.

To fuse the results of GMF and MLP, instead of simple addition, NeuMF concatenates the second last layers of two subnetworks to create a feature vector which can be passed to the further layers. Afterwards, the ouputs are projected with matrix $\mathbf{h}$ and a sigmoid activation function. The prediction layer is formulated as:

$$\hat{y}_{ui} = \sigma(\mathbf{h}^\top [\mathbf{x}, \phi^L(z^{(L)})]). \tag{15.6.3}$$

The following figure illustrates the model architecture of NeuMF.

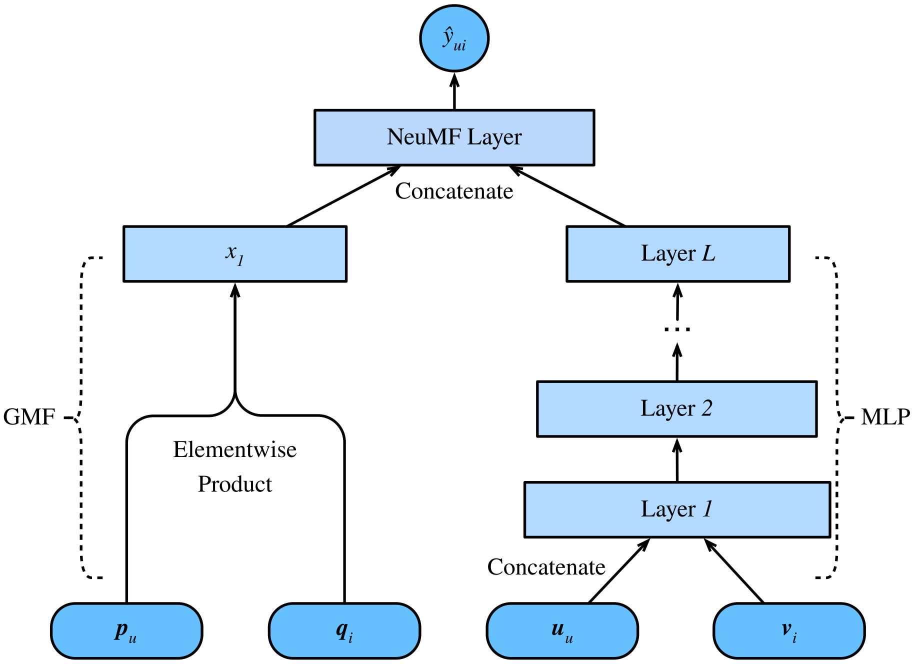


Fig. 15.6.1: Illustration of the NeuMF model

```
import d2l
from mxnet import autograd, gluon, np, npx
from mxnet.gluon import nn
import mxnet as mx
import random
import sys
npx.set_np()
```

15.6.2 Model Implementation

The following code implements the NeuMF model. It consists of a generalized matrix factorization model and a multi-layered perceptron with different user and item embedding vectors. The structure of the MLP is controlled with the parameter mlp_layers. ReLU is used as the default activation function.

```
class NeuMF(nn.Block):
    def __init__(self, num_factors, num_users, num_items, mlp_layers,
                 **kwargs):
        super(NeuMF, self).__init__(**kwargs)
        self.P = nn.Embedding(num_users, num_factors)
        self.Q = nn.Embedding(num_items, num_factors)
        self.U = nn.Embedding(num_users, num_factors)
        self.V = nn.Embedding(num_items, num_factors)
        self.mlp = nn.Sequential()  # The MLP layers
        for i in mlp_layers:
            self.mlp.add(gluon.nn.Dense(i, activation='relu', use_bias=True))

    def forward(self, user_id, item_id):
        p_mf = self.P(user_id)
```

(continues on next page)

(continued from previous page)

```
        q_mf = self.Q(item_id)
        gmf = p_mf * q_mf
        p_mlp = self.U(user_id)
        q_mlp = self.V(item_id)
        mlp = self.mlp(np.concatenate([p_mlp, q_mlp], axis=1))
        con_res = np.concatenate([gmf, mlp], axis=1)
        return np.sum(con_res, axis=-1)
```

15.6.3 Customized Dataset with Negative Sampling

For pairwise ranking loss, an important step is negative sampling. For each user, the items that a user has not interacted with are candidate items (unobserved entries). The following function takes users identity and candidate items as input, and samples negative items randomly for each user from the candidate set of that user. During the training stage, the model ensures that the items that a user likes to be ranked higher than items she dislikes or has not interacted with.

```
class PRDataset(gluon.data.Dataset):
    def __init__(self, users, items, candidates, num_items):
        self.users = users
        self.items = items
        self.cand = candidates
        self.all = set([i for i in range(num_items)])

    def __len__(self):
        return len(self.users)

    def __getitem__(self, idx):
        neg_items = list(self.all - set(self.cand[int(self.users[idx])]))
        indices = random.randint(0, len(neg_items) - 1)
        return self.users[idx], self.items[idx], neg_items[indices]
```

15.6.4 Evaluator

In this section, we adopt the splitting by time strategy to construct the training and test sets. Two evaluation measures including hit rate at given cutting off ℓ (Hit@ℓ) and area under the ROC curve (AUC) are used to assess the model effectiveness. Hit rate at given position ℓ for each user indicates that whether the recommended item is included in the top ℓ ranked list. The formal definition is as follows:

$$\text{Hit@}\ell = \frac{1}{m} \sum_{u \in \mathcal{U}} \mathbf{1}(rank_{u,g_u} <= \ell), \tag{15.6.4}$$

where $\mathbf{1}$ denotes an indicator function that is equal to one if the ground truth item is ranked in the top ℓ list, otherwise it is equal to zero. $rank_{u,g_u}$ denotes the ranking of the ground truth item g_u of the user u in the recommendation list (The ideal ranking is 1). m is the number of users. $\mathcal{U}$ is the user set.

The definition of AUC is as follows:

$$\text{AUC} = \frac{1}{m} \sum_{u \in \mathcal{U}} \frac{1}{|\mathcal{I} \backslash S_u|} \sum_{j \in I \backslash S_u} \mathbf{1}(rank_{u,g_u} < rank_{u,j}), \tag{15.6.5}$$

where $\mathcal{I}$ is the item set. S_u is the candidate items of user u. Note that many other evaluation protocols such as precision, recall and normalized discounted cumulative gain (NDCG) can also be used.

The following function calculates the hit counts and AUC for each user.

```
# Saved in the d2l package for later use
def hit_and_auc(rankedlist, test_matrix, k):
    hits_k = [(idx, val) for idx, val in enumerate(rankedlist[:k])
              if val in set(test_matrix)]
    hits_all = [(idx, val) for idx, val in enumerate(rankedlist)
                if val in set(test_matrix)]
    max = len(rankedlist) - 1
    auc = 1.0 * (max - hits_all[0][0]) / max if len(hits_all) > 0 else 0
    return len(hits_k), auc
```

Then, the overall Hit rate and AUC are calculated as follows.

```
# Saved in the d2l package for later use
def evaluate_ranking(net, test_input, seq, candidates, num_users, num_items,
                     ctx):
    ranked_list, ranked_items, hit_rate, auc = {}, {}, [], []
    all_items = set([i for i in range(num_users)])
    for u in range(num_users):
        neg_items = list(all_items - set(candidates[int(u)]))
        user_ids, item_ids, x, scores = [], [], [], []
        [item_ids.append(i) for i in neg_items]
        [user_ids.append(u) for _ in neg_items]
        x.extend([np.array(user_ids)])
        if seq is not None:
            x.append(seq[user_ids, :])
        x.extend([np.array(item_ids)])
        test_data_iter = gluon.data.DataLoader(gluon.data.ArrayDataset(*x),
                                               shuffle=False,
                                               last_batch="keep",
                                               batch_size=1024)
        for index, values in enumerate(test_data_iter):
            x = [gluon.utils.split_and_load(v, ctx, even_split=False)
                 for v in values]
            scores.extend([list(net(*t).asnumpy()) for t in zip(*x)])
        scores = [item for sublist in scores for item in sublist]
        item_scores = list(zip(item_ids, scores))
        ranked_list[u] = sorted(item_scores, key=lambda t: t[1], reverse=True)
        ranked_items[u] = [r[0] for r in ranked_list[u]]
        temp = hit_and_auc(ranked_items[u], test_input[u], 50)
        hit_rate.append(temp[0])
        auc.append(temp[1])
    return np.mean(np.array(hit_rate)), np.mean(np.array(auc))
```

15.6.5 Training and Evaluating the Model

The training function is defined below. We train the model in the pairwise manner.

```
# Saved in the d2l package for later use
def train_ranking(net, train_iter, test_iter, loss, trainer, test_seq_iter,
                  num_users, num_items, num_epochs, ctx_list, evaluator,
                  candidates, eval_step=1):
    timer, hit_rate, auc = d2l.Timer(), 0, 0
    animator = d2l.Animator(xlabel='epoch', xlim=[1, num_epochs], ylim=[0, 1],
                            legend=['test hit rate', 'test AUC'])
    for epoch in range(num_epochs):
        metric, l = d2l.Accumulator(3), 0.
        for i, values in enumerate(train_iter):
            input_data = []
            for v in values:
                input_data.append(gluon.utils.split_and_load(v, ctx_list))
            with autograd.record():
                p_pos = [net(*t) for t in zip(*input_data[0:-1])]
                p_neg = [net(*t) for t in zip(*input_data[0:-2],
                                              input_data[-1])]
                ls = [loss(p, n) for p, n in zip(p_pos, p_neg)]
            [l.backward(retain_graph=False) for l in ls]
            l += sum([l.asnumpy() for l in ls]).mean()/len(ctx_list)
            trainer.step(values[0].shape[0])
            metric.add(l, values[0].shape[0], values[0].size)
            timer.stop()
        with autograd.predict_mode():
            if (epoch + 1) % eval_step == 0:
                hit_rate, auc = evaluator(net, test_iter, test_seq_iter,
                                          candidates, num_users, num_items,
                                          ctx_list)
                animator.add(epoch + 1, (hit_rate, auc))
    print('train loss %.3f, test hit rate %.3f, test AUC %.3f'
          % (metric[0] / metric[1], hit_rate, auc))
    print('%.1f examples/sec on %s'
          % (metric[2] * num_epochs / timer.sum(), ctx_list))
```

Now, we can load the MovieLens 100k dataset and train the model. Since there are only ratings in the MovieLens dataset, with some losses of accuracy, we binarize these ratings to zeros and ones. If a user rated an item, we consider the implicit feedback as one, otherwise as zero. The action of rating an item can be treated as a form of providing implicit feedback. Here, we split the dataset in the seq-aware mode where users' latest interacted items are left out for test.

```
batch_size = 1024
df, num_users, num_items = d2l.read_data_ml100k()
train_data, test_data = d2l.split_data_ml100k(df, num_users, num_items,
                                              'seq-aware')
users_train, items_train, ratings_train, candidates = d2l.load_data_ml100k(
    train_data, num_users, num_items, feedback="implicit")
users_test, items_test, ratings_test, test_iter = d2l.load_data_ml100k(
    test_data, num_users, num_items, feedback="implicit")
num_workers = 0 if sys.platform.startswith("win") else 4
train_iter = gluon.data.DataLoader(PRDataset(users_train, items_train,
                                             candidates, num_items ),
```

(continues on next page)

(continued from previous page)

```
                                   batch_size, True,
                                   last_batch="rollover",
                                   num_workers=num_workers)
```

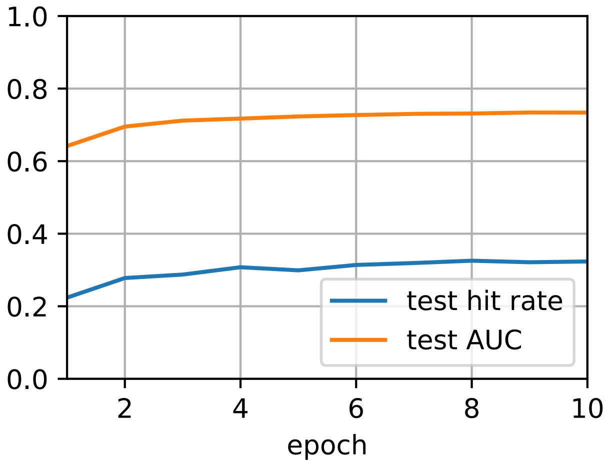
We then create and initialize the model. we use a three-layer MLP with constant hidden size 10.

```
ctx = d2l.try_all_gpus()
net = NeuMF(10, num_users, num_items, mlp_layers=[10, 10, 10])
net.initialize(ctx=ctx, force_reinit=True, init=mx.init.Normal(0.01))
```

The following code trains the model.

```
lr, num_epochs, wd, optimizer = 0.01, 10, 1e-5, 'adam'
loss = d2l.BPRLoss()
trainer = gluon.Trainer(net.collect_params(), optimizer,
                        {"learning_rate": lr, 'wd': wd})
train_ranking(net, train_iter, test_iter, loss, trainer, None, num_users,
              num_items, num_epochs, ctx, evaluate_ranking, candidates)
```

```
train loss 4.381, test hit rate 0.323, test AUC 0.734
15.6 examples/sec on [gpu(0), gpu(1)]
```



Summary

- Adding nonlinearity to matrix factorization model is beneficial for improving the model capability and effectiveness.
- NeuMF is a combination of matrix factorization and Multilayer perceptron. The multilayer perceptron takes the concatenation of user and item embeddings as the input.

Exercises

- Vary the size of latent factors. How the size of latent factors impact the model performance?
- Vary the architectures (e.g., number of layers, number of neurons of each layer) of the MLP to check the its impact on the performance.
- Try different optimizers, learning rate and weight decay rate.
- Try to use hinge loss defined in the last section to optimize this model.

15.7 Sequence-Aware Recommender Systems

In previous sections, we abstract the recommendation task as a matrix completion problem without considering users' short-term behaviors. In this section, we will introduce a recommendation model that takes the sequentially-ordered user interaction logs into account. It is a sequence-aware recommender (Quadrana et al., 2018) where the input is an ordered and often timestamped list of past user actions. A number of recent literatures have demonstrated the usefulness of incorporating such information in modeling users' temporal behavioral patterns and discovering their interest drift.

The model we will introduce, Caser (Sedhain et al., 2015), short for convolutional sequence embedding recommendation model, adopts convolutional neural networks capture the dynamic pattern influences of users' recent activities. The main component of Caser consists of a horizontal convolutional network and a vertical convolutional network, aiming to uncover the union-level and point-level sequence patterns, respectively. Point-level pattern indicates the impact of single item in the historical sequence on the target item, while union level pattern implies the influences of several previous actions on the subsequent target. For example, buying both milk and butter together leads to higher probability of buying flour than just buying one of them. Moreover, users' general interests, or long term preferences are also modeled in the last fully-connected layers, resulting in a more comprehensive modeling of user interests. Details of the model are described as follows.

15.7.1 Model Architectures

In sequence-aware recommendation system, each user is associated with a sequence of some items from the item set. Let $S^u = (S_1^u, ...S_{|S_u|}^u)$ denotes the ordered sequence. The goal of Caser is to recommend item by considering user general tastes as well as short-term intention. Suppose we take the previous L items into consideration, an embedding matrix that represents the former interactions for timestep t can be constructed:

$$\mathbf{E}^{(u,t)} = [\mathbf{q}_{S_{t-L}^u}, ..., \mathbf{q}_{S_{t-2}^u}, \mathbf{q}_{S_{t-1}^u}]^\top, \tag{15.7.1}$$

where $\mathbf{Q} \in \mathbb{R}^{n \times k}$ represents item embeddings and $\mathbf{q}_i$ denotes the i^th row. $\mathbf{E}^{(u,t)} \in \mathbb{R}^{L \times k}$ can be used to infer the transient interest of user u at time-step t. We can view the input matrix $\mathbf{E}^{(u,t)}$ as an image which is the input of the subsequent two convolutional components.

The horizontal convolutional layer has d horizontal filters $\mathbf{F}^j \in \mathbb{R}^{h\times k}, 1 \leq j \leq d, h = \{1, ..., L\}$, and the vertical convolutional layer has d' vertical filters $\mathbf{G}^j \in \mathbb{R}^{L\times 1}, 1 \leq j \leq d'$. After a series of convolutional and pool operations, we get the two outputs:

$$\begin{aligned}\mathbf{o} &= \text{HConv}(\mathbf{E}^{(u,t)}, \mathbf{F}) \\ \mathbf{o}' &= \text{VConv}(\mathbf{E}^{(u,t)}, \mathbf{G}),\end{aligned} \tag{15.7.2}$$

where $\mathbf{o} \in \mathbb{R}^d$ is the output of horizontal convolutional network and $\mathbf{o}' \in \mathbb{R}^{kd'}$ is the output of vertical convolutional network. For simplicity, we omit the details of convolution and pool operations. They are concatenated and fed into a fully-connected neural network layer to get more high-level representations.

$$\mathbf{z} = \phi(\mathbf{W}[\mathbf{o}, \mathbf{o}']^\top + \mathbf{b}), \tag{15.7.3}$$

where $\mathbf{W} \in \mathbb{R}^{k\times(d+kd')}$ is the weight matrix and $\mathbf{b} \in \mathbb{R}^k$ is the bias. The learned vector $\mathbf{z} \in \mathbb{R}^k$ is the representation of user's short-term intent.

At last, the prediction function combines users' short-term and general taste together, which is defined as:

$$\hat{y}_{uit} = \mathbf{v}_i \cdot [\mathbf{z}, \mathbf{p}_u]^\top + \mathbf{b}'_i, \tag{15.7.4}$$

where $\mathbf{V} \in \mathbb{R}^{n\times 2k}$ is another item embedding matrix. $\mathbf{b}' \in \mathbb{R}^n$ is the item specific bias. $\mathbf{P} \in \mathbb{R}^{m\times k}$ is the user embedding matrix for users' general tastes. $\mathbf{p}_u \in \mathbb{R}^k$ is the u^th row of P and $\mathbf{v}_i \in \mathbb{R}^{2k}$ is the i^th row of $\mathbf{V}$.

The model can be learned with BPR or Hinge loss. The architecture of Caser is shown below:

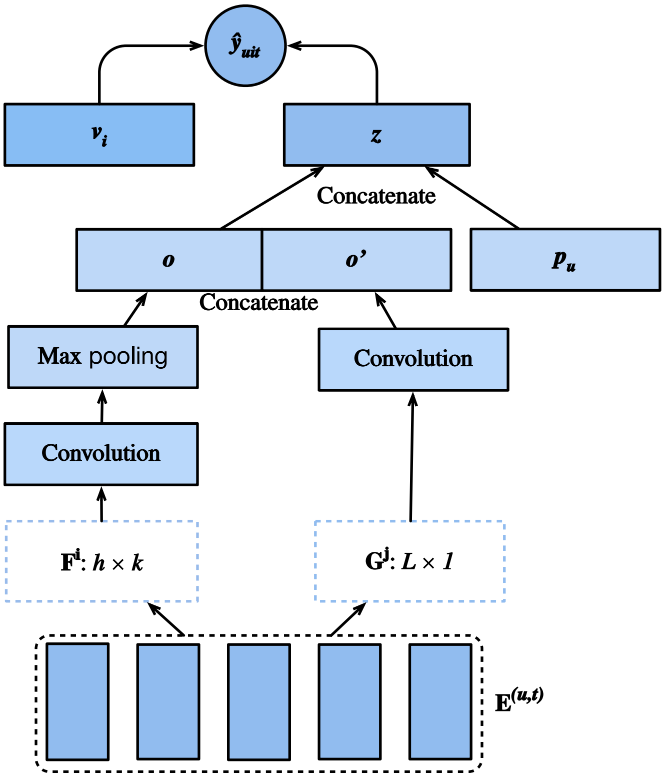


Fig. 15.7.1: Illustration of the Caser Model

We first import the required libraries.

```python
import d2l
from mxnet import gluon, np, npx
from mxnet.gluon import nn
import mxnet as mx
import random
import sys
npx.set_np()
```

15.7.2 Model Implementation

The following code implements the Caser model. It consists of a vertical convolutional layer, a horizontal convolutional layer, and a full-connected layer.

```python
class Caser(nn.Block):
    def __init__(self, num_factors, num_users, num_items, L=5, d=16,
                 d_prime=4, drop_ratio=0.05, **kwargs):
        super(Caser, self).__init__(**kwargs)
        self.P = nn.Embedding(num_users, num_factors)
        self.Q = nn.Embedding(num_items, num_factors)
        self.d_prime, self.d = d_prime, d
        # Vertical convolution layer
        self.conv_v = nn.Conv2D(d_prime, (L, 1), in_channels=1)
        # Horizontal convolution layer
        h = [i + 1 for i in range(L)]
        self.conv_h, self.max_pool = nn.Sequential(), nn.Sequential()
        for i in h:
            self.conv_h.add(nn.Conv2D(d, (i, num_factors), in_channels=1))
            self.max_pool.add(nn.MaxPool1D(L - i + 1))
        # Fully-connected layer
        self.fc1_dim_v, self.fc1_dim_h = d_prime * num_factors, d * len(h)
        self.fc = nn.Dense(in_units=d_prime * num_factors + d * L,
                           activation='relu', units=num_factors)
        self.Q_prime = nn.Embedding(num_items, num_factors * 2)
        self.b = nn.Embedding(num_items, 1)
        self.dropout = nn.Dropout(drop_ratio)

    def forward(self, user_id, seq, item_id):
        item_embs = np.expand_dims(self.Q(seq), 1)
        user_emb = self.P(user_id)
        out, out_h, out_v, out_hs = None, None, None, []
        if self.d_prime:
            out_v = self.conv_v(item_embs)
            out_v = out_v.reshape(out_v.shape[0], self.fc1_dim_v)
        if self.d:
            for conv, maxp in zip(self.conv_h, self.max_pool):
                conv_out = np.squeeze(npx.relu(conv(item_embs)), axis=3)
                t = maxp(conv_out)
                pool_out = np.squeeze(t, axis=2)
                out_hs.append(pool_out)
            out_h = np.concatenate(out_hs, axis=1)
        out = np.concatenate([out_v, out_h], axis=1)
        z = self.fc(self.dropout(out))
        x = np.concatenate([z, user_emb], axis=1)
```

(continues on next page)

(continued from previous page)

```
        q_prime_i = np.squeeze(self.Q_prime(item_id))
        b = np.squeeze(self.b(item_id))
        res = (x * q_prime_i).sum(1) + b
        return res
```

15.7.3 Sequential Dataset with Negative Sampling

To process the sequential interaction data, we need to reimplement the Dataset class. The following code creates a new dataset class named SeqDataset. In each sample, it outputs the user identity, her previous L interacted items as a sequence and the next item she interacts as the target. The following figure demonstrates the data loading process for one user. Suppose that this user liked 8 movies, we organize these eight movies in chronological order. The latest movie is left out as the test item. For the remaining seven movies, we can get three training samples, with each sample containing a sequence of five ($L = 5$) movies and its subsequent item as the target item. Negative samples are also included in the Customized dataset.

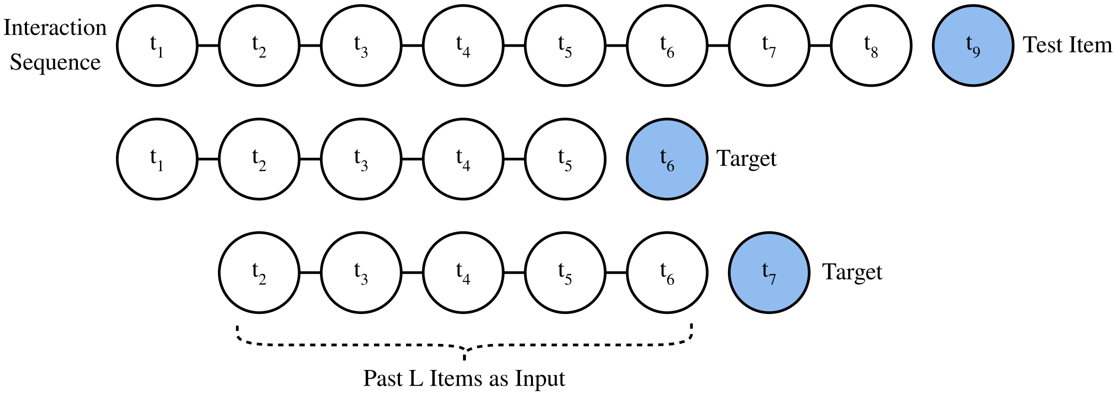


Fig. 15.7.2: Illustration of the data generation process

```
class SeqDataset(gluon.data.Dataset):
    def __init__(self, user_ids, item_ids, L, num_users, num_items,
                 candidates):
        user_ids, item_ids = np.array(user_ids), np.array(item_ids)
        sort_idx = np.array(sorted(range(len(user_ids)),
                                   key=lambda k: user_ids[k]))
        u_ids, i_ids = user_ids[sort_idx], item_ids[sort_idx]
        temp, u_ids, self.cand = {}, u_ids.asnumpy(), candidates
        self.all_items = set([i for i in range(num_items)])
        [temp.setdefault(u_ids[i], []).append(i) for i, _ in enumerate(u_ids)]
        temp = sorted(temp.items(), key=lambda x: x[0])
        u_ids = np.array([i[0] for i in temp])
        idx = np.array([i[1][0] for i in temp])
        self.ns = ns = int(sum([c - L if c >= L + 1 else 1 for c
                                in np.array([len(i[1]) for i in temp])]))
        self.seq_items = np.zeros((ns, L))
        self.seq_users = np.zeros(ns, dtype='int32')
        self.seq_tgt = np.zeros((ns, 1))
        self.test_seq = np.zeros((num_users, L))
```

(continues on next page)

(continued from previous page)

```
        test_users, _uid = np.empty(num_users), None
        for i, (uid, i_seq) in enumerate(self._seq(u_ids, i_ids, idx, L + 1)):
            if uid != _uid:
                self.test_seq[uid][:] = i_seq[-L:]
                test_users[uid], _uid = uid, uid
            self.seq_tgt[i][:] = i_seq[-1:]
            self.seq_items[i][:], self.seq_users[i] = i_seq[:L], uid

    def _win(self, tensor, window_size, step_size=1):
        if len(tensor) - window_size >= 0:
            for i in range(len(tensor), 0, - step_size):
                if i - window_size >= 0:
                    yield tensor[i - window_size:i]
                else:
                    break
        else:
            yield tensor

    def _seq(self, u_ids, i_ids, idx, max_len):
        for i in range(len(idx)):
            stop_idx = None if i >= len(idx) - 1 else int(idx[i + 1])
            for s in self._win(i_ids[int(idx[i]):stop_idx], max_len):
                yield (int(u_ids[i]), s)

    def __len__(self):
        return self.ns

    def __getitem__(self, i):
        neg = list(self.all_items - set(self.cand[int(self.seq_users[i])]))
        idx = random.randint(0, len(neg) - 1)
        return self.seq_users[i], self.seq_items[i], self.seq_tgt[i], neg[idx]
```

15.7.4 Load the MovieLens 100K dataset

Afterwards, we read and split the MovieLens 100K dataset in sequence-aware mode and load the training data with sequential dataloader implemented above.

```
TARGET_NUM, L, batch_size = 1, 3, 4096
df, num_users, num_items = d2l.read_data_ml100k()
train_data, test_data = d2l.split_data_ml100k(df, num_users, num_items,
                                              'seq-aware')
users_train, items_train, ratings_train, candidates = d2l.load_data_ml100k(
    train_data, num_users, num_items, feedback="implicit")
users_test, items_test, ratings_test, test_iter = d2l.load_data_ml100k(
    test_data, num_users, num_items, feedback="implicit")
train_seq_data = SeqDataset(users_train, items_train, L, num_users,
                            num_items, candidates)
num_workers = 0 if sys.platform.startswith("win") else 4
train_iter = gluon.data.DataLoader(train_seq_data, batch_size, True,
                                   last_batch="rollover",
                                   num_workers=num_workers)
test_seq_iter = train_seq_data.test_seq
train_seq_data[0]
```

```
(array(0, dtype=int32), array([110., 255.,   4.]), array([101.]), 657)
```

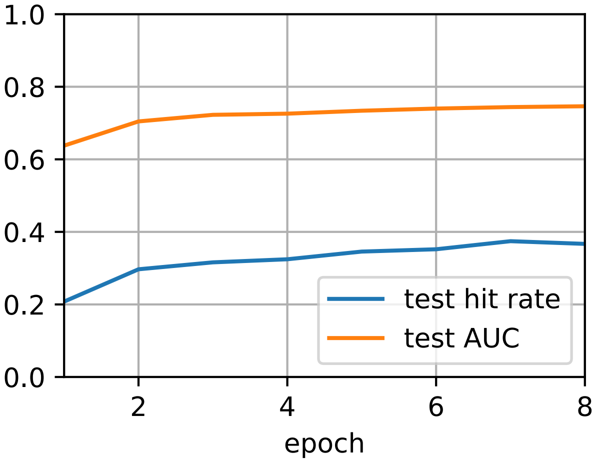
The training data structure is shown above. The first element is the user identity, the next list indicates the last five items this user liked, and the last element is the item this user liked after the five items.

15.7.5 Train the Model

Now, let's train the model. We use the same setting as NeuMF, including learning rate, optimizer, and k, in the last section so that the results are comparable.

```
ctx = d2l.try_all_gpus()
net = Caser(10, num_users, num_items, L)
net.initialize(ctx=ctx, force_reinit=True, init=mx.init.Normal(0.01))
lr, num_epochs, wd, optimizer = 0.04, 8, 1e-5, 'adam'
loss = d2l.BPRLoss()
trainer = gluon.Trainer(net.collect_params(), optimizer,
                        {"learning_rate": lr, 'wd': wd})

d2l.train_ranking(net, train_iter, test_iter, loss, trainer, test_seq_iter,
                  num_users, num_items, num_epochs, ctx, d2l.evaluate_ranking,
                  candidates, eval_step=1)
```

```
train loss 0.859, test hit rate 0.367, test AUC 0.746
35.0 examples/sec on [gpu(0), gpu(1)]
```



Summary

- Inferring a user's short-term and long-term interests can make prediction of the next item that she preferred more effectively.
- Convolutional neural networks can be utilized to capture users' short-term interests from sequential interactions.

Exercises

- Conduct an ablation study by removing one of the horizontal and vertical convolutional networks, which component is the more important ?
- Vary the hyper-parameter L. Does longer historical interactions bring higher accuracy?
- Apart from the sequence-aware recommendation task we introduced above, there is another type of sequence-aware recommendation task called session-based recommendation (Hidasi et al., 2015). Can you explain the differences between these two tasks?

15.8 Feature-Rich Recommender Systems

Interaction data is the most basic indication of users' preferences and interests. It plays a critical role in former introduced models. Yet, interaction data is usually extremely sparse and can be noisy at times. To address this issue, we can integrate side information such as features of items, profiles of users, and even in which context that the interaction occurred into the recommendation model. Utilizing these features are helpful in making recommendations in that these features can be an effective predictor of users interests especially when interaction data is lacking. As such, it is essential for recommendation models also have the capability to deal with those features and give the model some content/context awareness. To demonstrate this type of recommendation models, we introduce another task on click-through rate (CTR) for online advertisement recommendations (McMahan et al., 2013) and present an anonymous advertising data. Targeted advertisement services have attracted widespread attention and are often framed as recommendation engines. Recommending advertisements that match users' personal taste and interest is important for click-through rate improvement.

Digital marketers use online advertising to display advertisements to customers. Click-through rate is a metric that measures the number of clicks advertisers receive on their ads per number of impressions and it is expressed as a percentage calculated with the formula:

$$\text{CTR} = \frac{\#\text{Clicks}}{\#\text{Impressions}} \times 100\%. \tag{15.8.1}$$

Click-through rate is an important signal that indicates the effectiveness of prediction algorithms. Click-through rate prediction is a task of predicting the likelihood that something on a website will be clicked. Models on CTR prediction can not only be employed in targeted advertising systems but also in general item (e.g., movies, news, products) recommender systems, email campaigns, and even search engines. It is also closely related to user satisfaction, conversion rate, and can be helpful in setting campaign goals as it can help advertisers to set realistic expectations.

15.8.1 An Online Advertising Dataset

With the considerable advancements of Internet and mobile technology, online advertising has become an important income resource and generates vast majority of revenue in the Internet industry. It is important to display relevant advertisements or advertisements that pique users' interests so that casual visitors can be converted into paying customers. The dataset we introduced is an online advertising dataset. It consists of 34 fields, with the first column representing the target variable that indicates if an ad was clicked (1) or not (0). All the other columns are categorical features. The columns might represent the advertisement id, site or application id, device id, time, user profiles and so on. The real semantics of the features are undisclosed due to anonymization and privacy concern.

The following code downloads the dataset from our server and saves it into the local data folder.

```
from collections import defaultdict
from mxnet import gluon, np
import os

# Saved in the d2l package for later use
def read_data_ctr(path="../data/", train="ctr/train.csv",
                  test="ctr/test.csv"):
    data_path = ("https://apache-mxnet.s3-accelerate.amazonaws.com/"
                 "gluon/dataset/")
    train_sha1 = "6dec3052e49ce0d1cec5ebc6f5ded1172be0befb"
    test_sha1 = "c265e3c1fad0ed4caf8c1a373c580465a8096eb0"
    ctr_path = path+"ctr"
    os.makedirs(ctr_path, exist_ok=True)
    gluon.utils.download(data_path + train, ctr_path, train_sha1)
    gluon.utils.download(data_path + test, ctr_path, test_sha1)

read_data_ctr()
```

```
Downloading ../data/ctr/train.csv from https://apache-mxnet.s3-accelerate.amazonaws.com/
↪gluon/dataset/ctr/train.csv...
Downloading ../data/ctr/test.csv from https://apache-mxnet.s3-accelerate.amazonaws.com/gluon/
↪dataset/ctr/test.csv...
```

There are a training set and a test set, consisting of 15000 and 3000 samples/lines, respectively.

15.8.2 Dataset Wrapper

For the convenience of data loading, we implement a `CTRDataset` which loads the advertising dataset from the CSV file and can be used by `DataLoader`.

```
# Saved in the d2l package for later use
class CTRDataset(gluon.data.Dataset):
    def __init__(self, data_path, feat_mapper=None, defaults=None,
                 min_threshold=4, num_feat=34):
```

(continues on next page)

(continued from previous page)

```
        self.NUM_FEATS, self.count, self.data = num_feat, 0, {}
        feat_cnts = defaultdict(lambda: defaultdict(int))
        self.feat_mapper, self.defaults = feat_mapper, defaults
        self.field_dims = np.zeros(self.NUM_FEATS, dtype=np.int64)
        with open(data_path) as f:
            for line in f:
                instance = {}
                values = line.rstrip('\n').split('\t')
                if len(values) != self.NUM_FEATS + 1:
                    continue
                label = np.float32([0, 0])
                label[int(values[0])] = 1
                instance['y'] = [np.float32(values[0])]
                for i in range(1, self.NUM_FEATS + 1):
                    feat_cnts[i][values[i]] += 1
                    instance.setdefault('x', []).append(values[i])
                self.data[self.count] = instance
                self.count = self.count + 1
        if self.feat_mapper is None and self.defaults is None:
            feat_mapper = {i: {feat for feat, c in cnt.items() if c >=
                               min_threshold} for i, cnt in feat_cnts.items()}
            self.feat_mapper = {i: {feat: idx for idx, feat in enumerate(cnt)}
                                for i, cnt in feat_mapper.items()}
            self.defaults = {i: len(cnt) for i, cnt in feat_mapper.items()}
        for i, fm in self.feat_mapper.items():
            self.field_dims[i - 1] = len(fm) + 1
        self.offsets = np.array((0, *np.cumsum(self.field_dims).asnumpy()
                                 [:-1]))

    def __len__(self):
        return self.count

    def __getitem__(self, idx):
        feat = np.array([self.feat_mapper[i + 1].get(v, self.defaults[i + 1])
                         for i, v in enumerate(self.data[idx]['x'])])
        return feat + self.offsets, self.data[idx]['y']
```

The following example loads the training data and print out the first record.

```
train_data = CTRDataset(data_path="../data/ctr/train.csv")
train_data[0]
```

```
(array([ 143.,  145.,  227.,  236.,  957., 1250., 1471., 1566., 1624.,
        1933., 2008., 2061., 2256., 2304., 2305., 2360., 2745., 2746.,
        2747., 2748., 2892., 2988., 3165., 3184., 3194., 3195., 3527.,
        3643., 3687., 3697., 3732., 3740., 3773., 3797.]), [1.0])
```

As can be seen, all the 34 fields are categorical features. Each value represents the one-hot index of the corresponding entry. The label 0 means that it is not clicked. This `CTRDataset` can also be used to load other datasets such as the Criteo display advertising challenge Dataset[244] and the Avazu click-through rate prediction Dataset[245].

[244] https://labs.criteo.com/2014/02/kaggle-display-advertising-challenge-dataset/

[245] https://www.kaggle.com/c/avazu-ctr-prediction

Summary

- Click-through rate is an important metric that is used to measure the effectiveness of advertising systems and recommender systems.
- Click-through rate prediction is usually converted to a binary classification problem. The target is to predict whether an ad/item will be clicked or not based on given features.

Exercise

- Can you load the Criteo and Avazu dataset with the provided `CTRDataset`. It is worth noting that the Criteo dataset consisting of real-valued features so you may have to revise the code a bit.

15.9 Factorization Machines

Factorization machines (FM) (Rendle, 2010), proposed by Steffen Rendle in 2010, is a supervised algorithm that can be used for classification, regression, and ranking tasks. It quickly took notice and became a popular and impactful method for making predictions and recommendations. Particularly, it is a generalization of the linear regression model and the matrix factorization model. Moreover, it is reminiscent of support vector machines with a polynomial kernel. The strengths of factorization machines over the linear regression and matrix factorization are: (1) it can model χ-way variable interactions, where χ is the number of polynomial order and is usually set to two. (2) A fast optimization algorithm associated with factorization machines can reduce the polynomial computation time to linear complexity, making it extremely efficient especially for high dimensional sparse inputs. For these reasons, factorization machines are widely employed in modern advertisement and products recommendations. The technical details and implementations are described below.

15.9.1 2-Way Factorization Machines

Formally, let $x \in \mathbb{R}^d$ denote the feature vectors of one sample, and y denote the corresponding label which can be real-valued label or class label such as binary class "click/non-click". The model for a factorization machine of degree two is defined as:

$$\hat{y}(x) = \mathbf{w}_0 + \sum_{i=1}^d \mathbf{w}_i x_i + \sum_{i=1}^d \sum_{j=i+1}^d \langle \mathbf{v}_i, \mathbf{v}_j \rangle x_i x_j \tag{15.9.1}$$

where $\mathbf{w}_0 \in \mathbb{R}$ is the global bias; $\mathbf{w} \in \mathbb{R}^d$ denotes the weights of the i-th variable; $\mathbf{V} \in \mathbb{R}^{d \times k}$ represents the feature embeddings; $\mathbf{v}_i$ represents the i^th row of $\mathbf{V}$; k is the dimensionality of latent factors; $\langle \cdot, \cdot \rangle$ is the dot product of two vectors. $\langle \mathbf{v}_i, \mathbf{v}_j \rangle$ model the interaction between the i^th and j^th feature. Some feature interactions can be easily understood so they can be designed by experts. However, most other feature interactions are hidden in data and difficult to identify. So modeling feature interactions automatically can greatly reduce the efforts in feature engineering. It is obvious that the first two terms correspond to the linear regression model and the last term is an extension of the matrix factorization model. If the feature i represents a item and the feature j represents a user, the third term is exactly the dot product between user and item embeddings. It is worth noting that FM can also generalize to higher orders (degree > 2). Nevertheless, the numerical stability might weaken the generalization.

15.9.2 An Efficient Optimization Criterion

Optimizing the factorization machines in a straight forward method leads to a complexity of $\mathcal{O}(kd^2)$ as all pairwise interactions require to be computed. To solve this inefficiency problem, we can reorganize the third term of FM which could greatly reduce the computation cost, leading to a linear time complexity ($\mathcal{O}(kd)$). The reformulation of the pairwise interaction term is as follows:

$$\begin{aligned}
&\sum_{i=1}^d \sum_{j=i+1}^d \langle\mathbf{v}_i, \mathbf{v}_j\rangle x_i x_j \\
&= \frac{1}{2} \sum_{i=1}^d \sum_{j=1}^d\langle\mathbf{v}_i, \mathbf{v}_j\rangle x_i x_j - \frac{1}{2}\sum_{i=1}^d \langle\mathbf{v}_i, \mathbf{v}_i\rangle x_i x_i \\
&= \frac{1}{2} \big(\sum_{i=1}^d \sum_{j=1}^d \sum_{l=1}^k\mathbf{v}_{i, l} \mathbf{v}_{j, l} x_i x_j - \sum_{i=1}^d \sum_{l=1}^k \mathbf{v}_{i, l} \mathbf{v}_{j, l} x_i x_i \big)\\
&= \frac{1}{2} \sum_{l=1}^k \big((\sum_{i=1}^d \mathbf{v}_{i, l} x_i) (\sum_{j=1}^d \mathbf{v}_{j, l}x_j) - \sum_{i=1}^d \mathbf{v}_{i, l}^2 x_i^2 \big) \\
&= \frac{1}{2} \sum_{l=1}^k \big((\sum_{i=1}^d \mathbf{v}_{i, l} x_i)^2 - \sum_{i=1}^d \mathbf{v}_{i, l}^2 x_i^2\big)
\end{aligned} \tag{15.9.2}$$

With this reformulation, the model complexity are decreased greatly. Moreover, for sparse features, only non-zero elements needs to be computed so that the overall complexity is linear to the number of non-zero features.

To learn the FM model, we can use the MSE loss for regression task, the cross entropy loss for classification tasks, and the BPR loss for ranking task. Standard optimizers such as SGD and Adam are viable for optimization.

```
import d2l
from mxnet import init, gluon, np, npx
from mxnet.gluon import nn
import sys
npx.set_np()
```

15.9.3 Model Implementation

The following code implement the factorization machines. It is clear to see that FM consists a linear regression block and an efficient feature interaction block. We apply a sigmoid function over the final score since we treat the CTR prediction as a classification task.

```
class FM(nn.Block):
    def __init__(self, field_dims, num_factors):
        super(FM, self).__init__()
        input_size = int(sum(field_dims))
        self.embedding = nn.Embedding(input_size, num_factors)
        self.fc = nn.Embedding(input_size, 1)
        self.linear_layer = gluon.nn.Dense(1, use_bias=True)

    def forward(self, x):
        square_of_sum = np.sum(self.embedding(x), axis=1) ** 2
        sum_of_square = np.sum(self.embedding(x) ** 2, axis=1)
        x = self.linear_layer(self.fc(x).sum(1)) \
            + 0.5 * (square_of_sum - sum_of_square).sum(1, keepdims=True)
        x = npx.sigmoid(x)
        return x
```

15.9.4 Load the Advertising Dataset

We use the CTR data wrapper from the last section to load the online advertising dataset.

```
batch_size = 2048
d2l.read_data_ctr()
train_data = d2l.CTRDataset(data_path="../data/ctr/train.csv")
test_data = d2l.CTRDataset(data_path="../data/ctr/test.csv",
                           feat_mapper=train_data.feat_mapper,
                           defaults=train_data.defaults)
num_workers = 0 if sys.platform.startswith("win") else 4
train_iter = gluon.data.DataLoader(train_data, shuffle=True,
                                   last_batch="rollover",
                                   batch_size=batch_size,
                                   num_workers=num_workers)
test_iter = gluon.data.DataLoader(test_data, shuffle=False,
                                  last_batch="rollover",
                                  batch_size=batch_size,
                                  num_workers=num_workers)
```

```
Downloading ../data/ctr/train.csv from https://apache-mxnet.s3-accelerate.amazonaws.com/
↪gluon/dataset/ctr/train.csv...
```

```
/var/lib/jenkins/miniconda3/envs/d2l-en-1/lib/python3.7/site-packages/mxnet/gluon/utils.
↪py:341: UserWarning: File ../data/ctr/train.csv exists in file system so the downloaded␣
↪file is deleted
  'File {} exists in file system so the downloaded file is deleted'.format(fname))
/var/lib/jenkins/miniconda3/envs/d2l-en-1/lib/python3.7/site-packages/mxnet/gluon/utils.
↪py:341: UserWarning: File ../data/ctr/test.csv exists in file system so the downloaded␣
↪file is deleted
  'File {} exists in file system so the downloaded file is deleted'.format(fname))
```

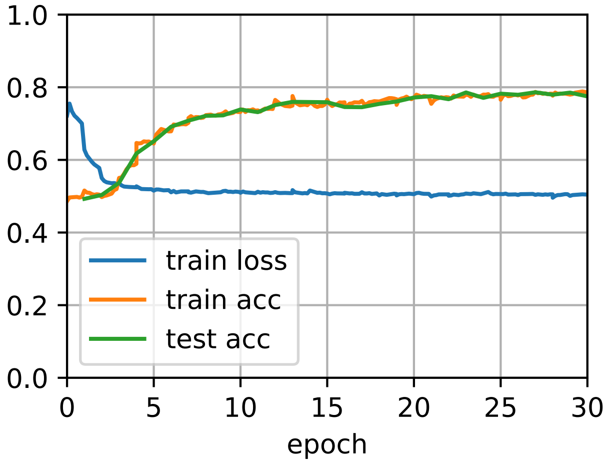
```
Downloading ../data/ctr/test.csv from https://apache-mxnet.s3-accelerate.amazonaws.com/gluon/
↪dataset/ctr/test.csv...
```

15.9.5 Train the Model

Afterwards, we train the model. The learning rate is set to 0.01 and the embedding size is set to 20 by default. The Adam optimizer and the SigmoidBinaryCrossEntropyLoss loss are used for model training.

```
ctx = d2l.try_all_gpus()
net = FM(train_data.field_dims, num_factors=20)
net.initialize(init.Xavier(), ctx=ctx)
lr, num_epochs, optimizer = 0.02, 30, 'adam'
trainer = gluon.Trainer(net.collect_params(), optimizer,
                        {'learning_rate': lr})
loss = gluon.loss.SigmoidBinaryCrossEntropyLoss()
d2l.train_ch13(net, train_iter, test_iter, loss, trainer, num_epochs, ctx)
```

```
loss 0.505, train acc 0.787, test acc 0.775
217142.6 exampes/sec on [gpu(0), gpu(1)]
```



Summary

- FM is a general framework that can be applied on a variety of tasks such as regression, classification, and ranking.
- Feature interaction/crossing is important for prediction tasks and the 2-way interaction can be efficiently modeled with FM.

Exercise

- Can you test FM on other dataset such as Avazu, MovieLens, and Criteo datasets?
- Vary the embedding size to check its impact on performance, can you observe a similar pattern as that of matrix factorization?

15.10 Deep Factorization Machines

Learning effective feature combinations is critical to the success of click-through rate prediction task. Factorization machines model feature interactions in a linear paradigm (e.g., bilinear interactions). This is often insufficient for real-world data where inherent feature crossing structures are usually very complex and nonlinear. What's worse, second-order feature interactions are generally used in factorization machines in practice. Modeling higher degrees of feature combinations with factorization machines is possible theoretically but it is usually not adopted due to numerical instability and high computational complexity.

One effective solution is using deep neural networks. Deep neural networks are powerful in feature representation learning and have the potential to learn sophisticated feature interactions. As such, it is natural to integrate deep neural networks to factorization machines. Adding nonlinear transformation layers to factorization machines gives it the capability to model both low-order feature combinations and high-order feature combinations. Moreover, non-linear inherent structures from inputs can also be captured with deep neural networks. In this section, we will introduce a representative model named deep factorization machines (DeepFM) (Guo et al., 2017) which combine FM and deep neural networks.

15.10.1 Model Architectures

DeepFM consists of an FM component and a deep component which are integrated in a parallel structure. The FM component is the same as the 2-way factorization machines which is used to model the low-order feature interactions. The deep component is a multi-layered perceptron that is used to capture high-order feature interactions and nonlinearities. These two components share the same inputs/embeddings and their outputs are summed up as the final prediction. It is worth pointing out that the spirit of DeepFM resembles that of the Wide & Deep architecture which can capture both memorization and generalization. The advantages of DeepFM over the Wide & Deep model is that it reduces the effort of hand-crafted feature engineering by identifying feature combinations automatically.

We omit the description of the FM component for brevity and denote the output as $\hat{y}^{(FM)}$. Readers are referred to the last section for more details. Let $\mathbf{e}_i \in \mathbb{R}^k$ denote the latent feature vector of the i^th field. The input of the deep component is the concatenation of the dense embeddings of all fields that are looked up with the sparse categorical feature input, denoted as:

$$\mathbf{z}^{(0)} = [\mathbf{e}_1, \mathbf{e}_2, ..., \mathbf{e}_f], \tag{15.10.1}$$

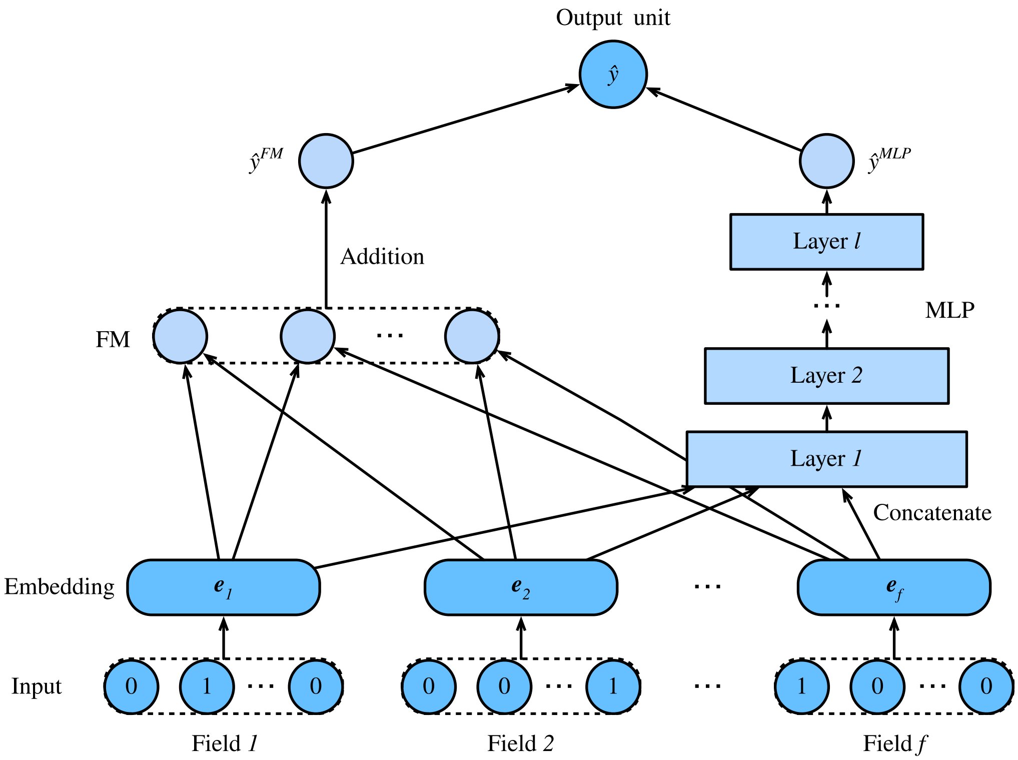
where f is the number of fields. It is then fed into the following neural network:

$$\mathbf{z}^{(l)} = \alpha(\mathbf{W}^{(l)}\mathbf{z}^{(l-1)} + \mathbf{b}^{(l)}), \tag{15.10.2}$$

where α is the activation function. $\mathbf{W}_l$ and $\mathbf{b}_l$ are the weight and bias at the l^{th} layer. Let y_{DNN} denote the output of the prediction. The ultimate prediction of DeepFM is the summation of the outputs from both FM and DNN. So we have:

$$\hat{y} = \sigma(\hat{y}^{(FM)} + \hat{y}^{(DNN)}), \tag{15.10.3}$$

where σ is the sigmoid function. The architecture of DeepFM is illustrated below.



It is worth noting that DeepFM is not the only way to combine deep neural networks with FM. We can also add nonlinear layers over the feature interactions (He & Chua, 2017).

```python
import d2l
from mxnet import init, gluon, np, npx
from mxnet.gluon import nn
import sys
npx.set_np()
```

15.10.2 Implemenation of DeepFM

The implementation of DeepFM is similar to that of FM. We keep the FM part unchanged and use an MLP block with relu as the activation function. Dropout is also used to regularize the model. The number of neurons of the MLP can be adjusted with the mlp_dims hyper-parameter.

```
class DeepFM(nn.Block):
    def __init__(self, field_dims, num_factors, mlp_dims, drop_rate=0.1):
        super(DeepFM, self).__init__()
        input_size = int(sum(field_dims))
        self.embedding = nn.Embedding(input_size, num_factors)
        self.fc = nn.Embedding(input_size, 1)
        self.linear_layer = gluon.nn.Dense(1, use_bias=True)
        input_dim = self.embed_output_dim = len(field_dims) * num_factors
        self.mlp = nn.Sequential()
        for dim in mlp_dims:
            self.mlp.add(nn.Dense(dim, 'relu', True, in_units=input_dim))
            self.mlp.add(nn.Dropout(rate=drop_rate))
            input_dim = dim
        self.mlp.add(nn.Dense(in_units=input_dim, units=1))

    def forward(self, x):
        embed_x = self.embedding(x)
        square_of_sum = np.sum(embed_x, axis=1) ** 2
        sum_of_square = np.sum(embed_x ** 2, axis=1)
        inputs = np.reshape(embed_x, (-1, self.embed_output_dim))
        x = self.linear_layer(self.fc(x).sum(1)) \
            + 0.5 * (square_of_sum - sum_of_square).sum(1, keepdims=True) \
            + self.mlp(inputs)
        x = npx.sigmoid(x)
        return x
```

15.10.3 Training and Evaluating the Model

The data loading process is the same as that of FM. We set the MLP component of DeepFM to a three-layered dense network with the a pyramid structure (30-20-10). All other hyper-parameters remain the same as FM.

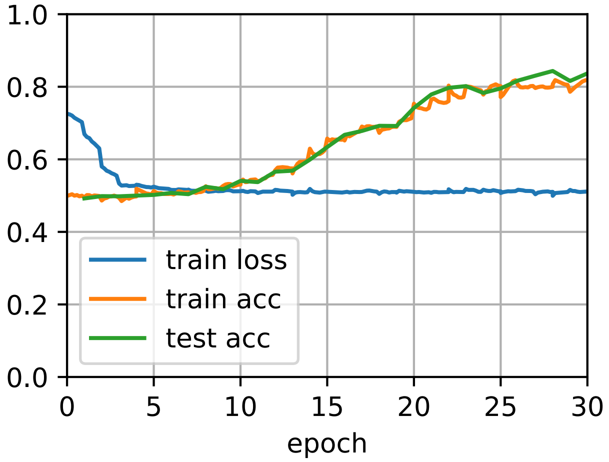
```
batch_size = 2048
d2l.read_data_ctr()
train_data = d2l.CTRDataset(data_path="../data/ctr/train.csv")
test_data = d2l.CTRDataset(data_path="../data/ctr/test.csv",
                           feat_mapper=train_data.feat_mapper,
                           defaults=train_data.defaults)
field_dims = train_data.field_dims
num_workers = 0 if sys.platform.startswith("win") else 4
train_iter = gluon.data.DataLoader(train_data, shuffle=True,
                                   last_batch="rollover",
                                   batch_size=batch_size,
                                   num_workers=num_workers)
test_iter = gluon.data.DataLoader(test_data, shuffle=False,
                                  last_batch="rollover",
                                  batch_size=batch_size,
```

(continues on next page)

(continued from previous page)

```
                                  num_workers=num_workers)
ctx = d2l.try_all_gpus()
net = DeepFM(field_dims, num_factors=10, mlp_dims=[30, 20, 10])
net.initialize(init.Xavier(), ctx=ctx)
lr, num_epochs, optimizer = 0.01, 30, 'adam'
trainer = gluon.Trainer(net.collect_params(), optimizer,
                        {'learning_rate': lr})
loss = gluon.loss.SigmoidBinaryCrossEntropyLoss()
d2l.train_ch13(net, train_iter, test_iter, loss, trainer, num_epochs, ctx)
```

```
loss 0.511, train acc 0.817, test acc 0.837
124928.7 exampes/sec on [gpu(0), gpu(1)]
```



Compared with FM, DeepFM converges faster and achieves better performance.

Summary

- Integrating neural networks to FM enables it to model complex and high-order interactions.
- DeepFM outperforms the original FM on the advertising dataset.

Exercise

- Vary the structure of the MLP to check its impact on model performance.
- Change the dataset to Criteo and compare it with the original FM model.

16 | Generative Adversarial Networks

16.1 Generative Adversarial Networks

Throughout most of this book, we have talked about how to make predictions. In some form or another, we used deep neural networks learned mappings from data points to labels. This kind of learning is called discriminative learning, as in, we'd like to be able to discriminate between photos cats and photos of dogs. Classifiers and regressors are both examples of discriminative learning. And neural networks trained by backpropagation have upended everything we thought we knew about discriminative learning on large complicated datasets. Classification accuracies on high-res images has gone from useless to human-level (with some caveats) in just 5-6 years. We will spare you another spiel about all the other discriminative tasks where deep neural networks do astoundingly well.

But there is more to machine learning than just solving discriminative tasks. For example, given a large dataset, without any labels, we might want to learn a model that concisely captures the characteristics of this data. Given such a model, we could sample synthetic data points that resemble the distribution of the training data. For example, given a large corpus of photographs of faces, we might want to be able to generate a new photorealistic image that looks like it might plausibly have come from the same dataset. This kind of learning is called generative modeling.

Until recently, we had no method that could synthesize novel photorealistic images. But the success of deep neural networks for discriminative learning opened up new possibilities. One big trend over the last three years has been the application of discriminative deep nets to overcome challenges in problems that we do not generally think of as supervised learning problems. The recurrent neural network language models are one example of using a discriminative network (trained to predict the next character) that once trained can act as a generative model.

In 2014, a breakthrough paper introduced Generative adversarial networks (GANs) (Goodfellow et al., 2014), a clever new way to leverage the power of discriminative models to get good generative models. At their heart, GANs rely on the idea that a data generator is good if we cannot tell fake data apart from real data. In statistics, this is called a two-sample test - a test to answer the question whether datasets $X = \{x_1, \ldots, x_n\}$ and $X' = \{x'_1, \ldots, x'_n\}$ were drawn from the same distribution. The main difference between most statistics papers and GANs is that the latter use this idea in a constructive way. In other words, rather than just training a model to say "hey, these two datasets do not look like they came from the same distribution", they use the two-sample test[249] to provide training signals to a generative model. This allows us to improve the data generator until it generates something that resembles the real data. At the very least, it needs to fool the classifier. Even if our classifier is a state of the art deep neural network.

[249] https://en.wikipedia.org/wiki/Two-sample_hypothesis_testing

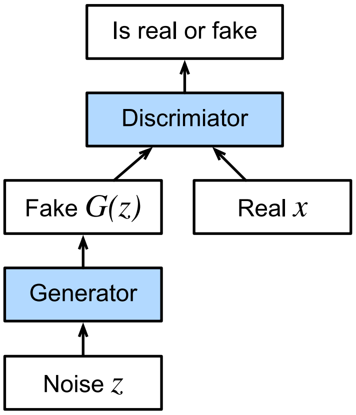


Fig. 16.1.1: Generative Adversarial Networks

The GAN architecture is illustrated in Fig. 16.1.1. As you can see, there are two pieces in GAN architecture - first off, we need a device (say, a deep network but it really could be anything, such as a game rendering engine) that might potentially be able to generate data that looks just like the real thing. If we are dealing with images, this needs to generate images. If we are dealing with speech, it needs to generate audio sequences, and so on. We call this the generator network. The second component is the discriminator network. It attempts to distinguish fake and real data from each other. Both networks are in competition with each other. The generator network attempts to fool the discriminator network. At that point, the discriminator network adapts to the new fake data. This information, in turn is used to improve the generator network, and so on.

The discriminator is a binary classifier to distinguish if the input x is real (from real data) or fake (from the generator). Typically, the discriminator outputs a scalar prediction $o \in \mathbb{R}$ for input $\mathbf{x}$, such as using a dense layer with hidden size 1, and then applies sigmoid function to obtain the predicted probability $D(\mathbf{x}) = 1/(1 + e^{-o})$. Assume the label y for the true data is 1 and 0 for the fake data. We train the discriminator to minimize the cross-entropy loss, *i.e.*,

$$\min_D \{ - y \log D(\mathbf{x}) - (1-y)\log(1-D(\mathbf{x})) \}, \tag{16.1.1}$$

For the generator, it first draws some parameter $\mathbf{z} \in \mathbb{R}^d$ from a source of randomness, *e.g.*, a normal distribution $\mathbf{z} \sim \mathcal{N}(0, 1)$. We often call $\mathbf{z}$ as the latent variable. It then applies a function to generate $\mathbf{x}' = G(\mathbf{z})$. The goal of the generator is to fool the discriminator to classify $\mathbf{x}' = G(\mathbf{z})$ as true data, *i.e.*, we want $D(G(\mathbf{z})) \approx 1$. In other words, for a given discriminator D, we update the parameters of the generator G to maximize the cross-entropy loss when $y = 0$, *i.e.*,

$$\max_G \{ - (1-y) \log(1-D(G(\mathbf{z}))) \} = \max_G \{ - \log(1-D(G(\mathbf{z}))) \}. \tag{16.1.2}$$

If the generator does a perfect job, then $D(\mathbf{x}') \approx 1$ so the above loss near 0, which results the gradients are too small to make a good progress for the discriminator. So commonly we minimize the following loss:

$$\min_G \{ - y \log(D(G(\mathbf{z}))) \} = \min_G \{ - \log(D(G(\mathbf{z}))) \}, \tag{16.1.3}$$

which is just feed $\mathbf{x}' = G(\mathbf{z})$ into the discriminator but giving label $y = 1$.

To sum up, D and G are playing a "minimax" game with the comprehensive objective function:

$$min_D max_G \{ -E_{x \sim \text{Data}} log D(\mathbf{x}) - E_{z \sim \text{Noise}} log(1 - D(G(\mathbf{z}))) \}. \tag{16.1.4}$$

Many of the GANs applications are in the context of images. As a demonstration purpose, we are going to content ourselves with fitting a much simpler distribution first. We will illustrate what happens if we use GANs to build the world's most inefficient estimator of parameters for a Gaussian. Let's get started.

```
%matplotlib inline
import d2l
from mxnet import autograd, gluon, init, np, npx
from mxnet.gluon import nn
npx.set_np()
```

16.1.1 Generate some "real" data

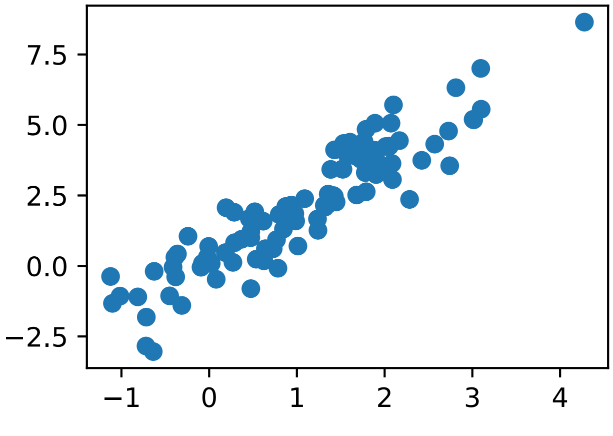
Since this is going to be the world's lamest example, we simply generate data drawn from a Gaussian.

```
X = np.random.normal(size=(1000, 2))
A = np.array([[1, 2], [-0.1, 0.5]])
b = np.array([1, 2])
data = X.dot(A) + b
```

Let's see what we got. This should be a Gaussian shifted in some rather arbitrary way with mean b and covariance matrix A^TA.

```
d2l.set_figsize((3.5, 2.5))
d2l.plt.scatter(data[:100, 0].asnumpy(), data[:100, 1].asnumpy());
print("The covariance matrix is\n%s" % np.dot(A.T, A))
```

```
The covariance matrix is
[[1.01 1.95]
 [1.95 4.25]]
```



```
batch_size = 8
data_iter = d2l.load_array((data,), batch_size)
```

16.1.2 Generator

Our generator network will be the simplest network possible - a single layer linear model. This is since we will be driving that linear network with a Gaussian data generator. Hence, it literally only needs to learn the parameters to fake things perfectly.

```
net_G = nn.Sequential()
net_G.add(nn.Dense(2))
```

16.1.3 Discriminator

For the discriminator we will be a bit more discriminating: we will use an MLP with 3 layers to make things a bit more interesting.

```
net_D = nn.Sequential()
net_D.add(nn.Dense(5, activation='tanh'),
          nn.Dense(3, activation='tanh'),
          nn.Dense(1))
```

16.1.4 Training

First we define a function to update the discriminator.

```
# Saved in the d2l package for later use
def update_D(X, Z, net_D, net_G, loss, trainer_D):
    """Update discriminator"""
    batch_size = X.shape[0]
    ones = np.ones((batch_size,), ctx=X.context)
    zeros = np.zeros((batch_size,), ctx=X.context)
    with autograd.record():
        real_Y = net_D(X)
        fake_X = net_G(Z)
        # Do not need to compute gradient for net_G, detach it from
        # computing gradients.
        fake_Y = net_D(fake_X.detach())
        loss_D = (loss(real_Y, ones) + loss(fake_Y, zeros)) / 2
    loss_D.backward()
    trainer_D.step(batch_size)
    return float(loss_D.sum())
```

The generator is updated similarly. Here we reuse the cross-entropy loss but change the label of the fake data from 0 to 1.

```
# Saved in the d2l package for later use
def update_G(Z, net_D, net_G, loss, trainer_G):  # saved in d2l
    """Update generator"""
    batch_size = Z.shape[0]
    ones = np.ones((batch_size,), ctx=Z.context)
    with autograd.record():
        # We could reuse fake_X from update_D to save computation.
        fake_X = net_G(Z)
```

(continues on next page)

(continued from previous page)

```
        # Recomputing fake_Y is needed since net_D is changed.
        fake_Y = net_D(fake_X)
        loss_G = loss(fake_Y, ones)
    loss_G.backward()
    trainer_G.step(batch_size)
    return float(loss_G.sum())
```

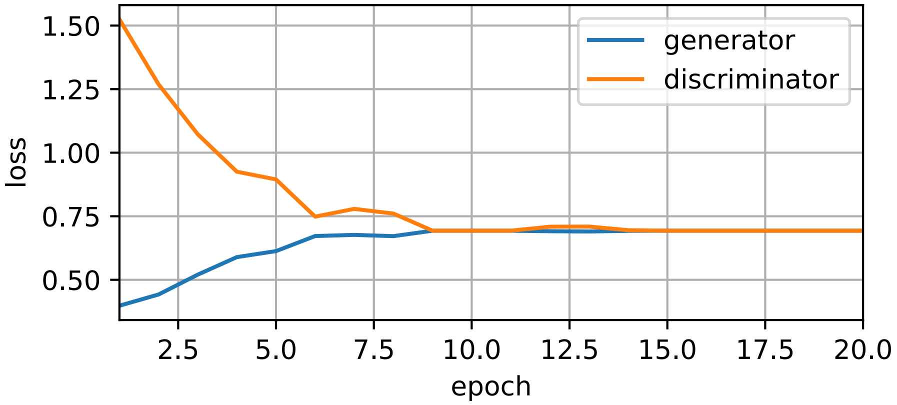
Both the discriminator and the generator performs a binary logistic regression with the cross-entropy loss. We use Adam to smooth the training process. In each iteration, we first update the discriminator and then the generator. We visualize both losses and generated examples.

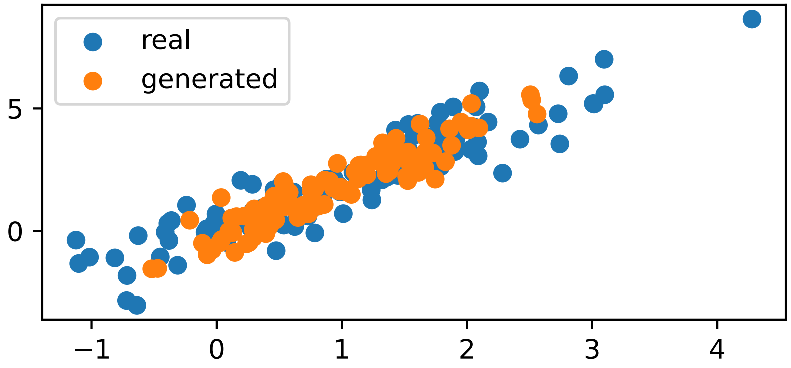
```
def train(net_D, net_G, data_iter, num_epochs, lr_D, lr_G, latent_dim, data):
    loss = gluon.loss.SigmoidBCELoss()
    net_D.initialize(init=init.Normal(0.02), force_reinit=True)
    net_G.initialize(init=init.Normal(0.02), force_reinit=True)
    trainer_D = gluon.Trainer(net_D.collect_params(),
                              'adam', {'learning_rate': lr_D})
    trainer_G = gluon.Trainer(net_G.collect_params(),
                              'adam', {'learning_rate': lr_G})
    animator = d2l.Animator(xlabel='epoch', ylabel='loss',
                            xlim=[1, num_epochs], nrows=2, figsize=(5, 5),
                            legend=['generator', 'discriminator'])
    animator.fig.subplots_adjust(hspace=0.3)
    for epoch in range(1, num_epochs+1):
        # Train one epoch
        timer = d2l.Timer()
        metric = d2l.Accumulator(3)  # loss_D, loss_G, num_examples
        for X in data_iter:
            batch_size = X.shape[0]
            Z = np.random.normal(0, 1, size=(batch_size, latent_dim))
            metric.add(update_D(X, Z, net_D, net_G, loss, trainer_D),
                       update_G(Z, net_D, net_G, loss, trainer_G),
                       batch_size)
        # Visualize generated examples
        Z = np.random.normal(0, 1, size=(100, latent_dim))
        fake_X = net_G(Z).asnumpy()
        animator.axes[1].cla()
        animator.axes[1].scatter(data[:, 0], data[:, 1])
        animator.axes[1].scatter(fake_X[:, 0], fake_X[:, 1])
        animator.axes[1].legend(['real', 'generated'])
        # Show the losses
        loss_D, loss_G = metric[0]/metric[2], metric[1]/metric[2]
        animator.add(epoch, (loss_D, loss_G))
    print('loss_D %.3f, loss_G %.3f, %d examples/sec' % (
        loss_D, loss_G, metric[2]/timer.stop()))
```

Now we specify the hyper-parameters to fit the Gaussian distribution.

```
lr_D, lr_G, latent_dim, num_epochs = 0.05, 0.005, 2, 20
train(net_D, net_G, data_iter, num_epochs, lr_D, lr_G,
      latent_dim, data[:100].asnumpy())
```

```
loss_D 0.693, loss_G 0.693, 628 examples/sec
```





Summary

- Generative adversarial networks (GANs) composes of two deep networks, the generator and the discriminator.
- The generator generates the image as much closer to the true image as possible to fool the discriminator, via maximizing the cross-entropy loss, *i.e.*, $\max \log(D(\mathbf{x}'))$.
- The discriminator tries to distinguish the generated images from the true images, via minimizing the cross-entropy loss, *i.e.*, $\min -y \log D(\mathbf{x}) - (1-y)\log(1-D(\mathbf{x}))$.

Exercises

- Does an equilibrium exist where the generator wins, *i.e.* the discriminator ends up unable to distinguish the two distributions on finite samples?

16.2 Deep Convolutional Generative Adversarial Networks

In Section 16.1, we introduced the basic ideas behind how GANs work. We showed that they can draw samples from some simple, easy-to-sample distribution, like a uniform or normal distribution, and transform them into samples that appear to match the distribution of some dataset. And while our example of matching a 2D Gaussian distribution got the point across, it is not especially exciting.

In this section, we will demonstrate how you can use GANs to generate photorealistic images. We will be basing our models on the deep convolutional GANs (DCGAN) introduced in (Radford et al., 2015). We will borrow the convolutional architecture that have proven so successful for discriminative computer vision problems and show how via GANs, they can be leveraged to generate photorealistic images.

```
from mxnet import gluon, init, np, npx
from mxnet.gluon import nn
import d2l
import zipfile

npx.set_np()
```

16.2.1 The Pokemon Dataset

The dataset we will use is a collection of Pokemon sprites obtained from pokemondb[251]. First download, extract and load this dataset.

```
data_dir = '../data/'
url = 'http://data.mxnet.io/data/pokemon.zip'
sha1 = 'c065c0e2593b8b161a2d7873e42418bf6a21106c'
fname = gluon.utils.download(url, data_dir, sha1_hash=sha1)
with zipfile.ZipFile(fname) as f:
    f.extractall(data_dir)
pokemon = gluon.data.vision.datasets.ImageFolderDataset(data_dir+'pokemon')
```

```
Downloading ../data/pokemon.zip from http://data.mxnet.io/data/pokemon.zip...
```

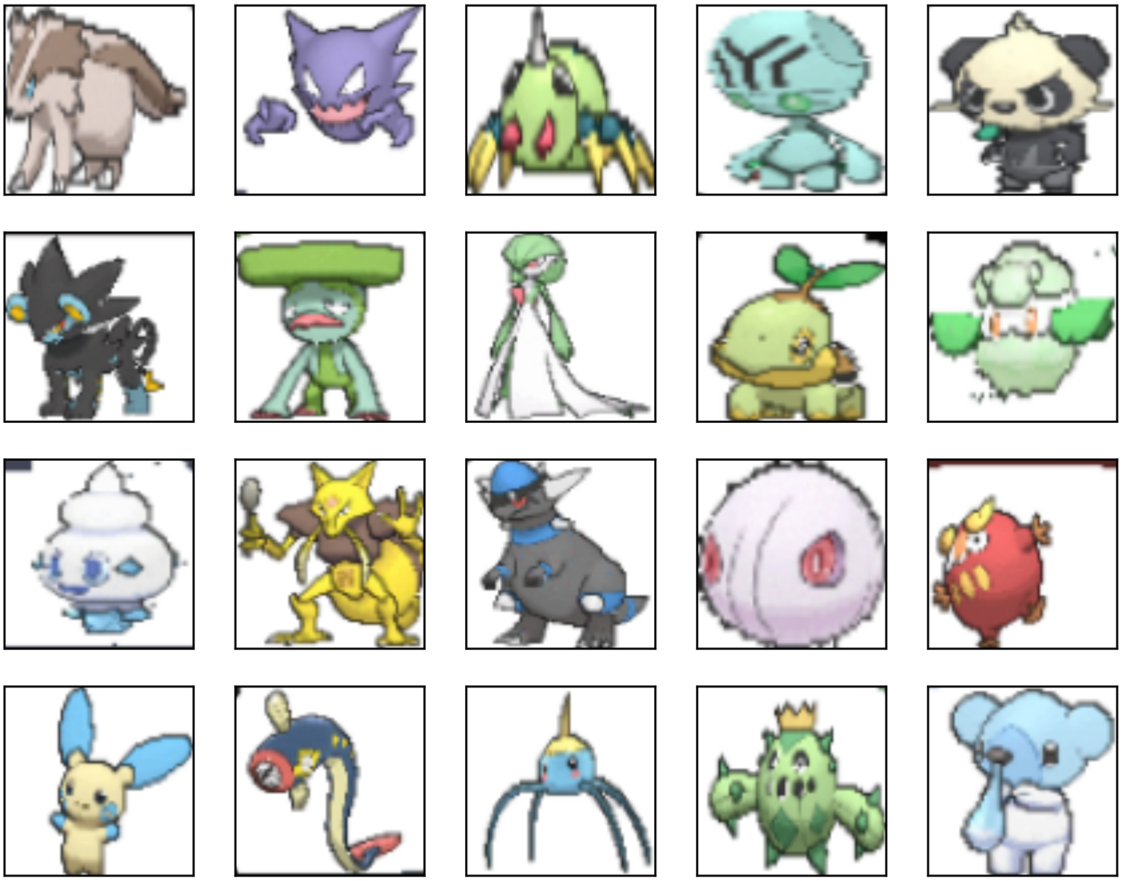
We resize each image into 64×64. The ToTensor transformation will project the pixel value into $[0, 1]$, while our generator will use the tanh function to obtain outputs in $[-1, 1]$. Therefore we normalize the data with 0.5 mean and 0.5 standard deviation to match the value range.

```
batch_size = 256
transformer = gluon.data.vision.transforms.Compose([
    gluon.data.vision.transforms.Resize(64),
    gluon.data.vision.transforms.ToTensor(),
    gluon.data.vision.transforms.Normalize(0.5, 0.5)
])
data_iter = gluon.data.DataLoader(
    pokemon.transform_first(transformer), batch_size=batch_size,
    shuffle=True, num_workers=d2l.get_dataloader_workers())
```

[251] https://pokemondb.net/sprites

Let’s visualize the first 20 images.

```
d2l.set_figsize((4, 4))
for X, y in data_iter:
    imgs = X[0:20,:,:,:].transpose(0, 2, 3, 1)/2+0.5
    d2l.show_images(imgs, num_rows=4, num_cols=5)
    break
```



16.2.2 The Generator

The generator needs to map the noise variable $\mathbf{z} \in \mathbb{R}^d$, a length-$d$ vector, to a RGB image with width and height to be 64×64 . In Section 13.11 we introduced the fully convolutional network that uses transposed convolution layer (refer to Section 13.10) to enlarge input size. The basic block of the generator contains a transposed convolution layer followed by the batch normalization and ReLU activation.

```
class G_block(nn.Block):
    def __init__(self, channels, kernel_size=4,
                 strides=2, padding=1, **kwargs):
        super(G_block, self).__init__(**kwargs)
        self.conv2d_trans = nn.Conv2DTranspose(
            channels, kernel_size, strides, padding, use_bias=False)
        self.batch_norm = nn.BatchNorm()
        self.activation = nn.Activation('relu')
```

(continues on next page)

(continued from previous page)

```
    def forward(self, X):
        return self.activation(self.batch_norm(self.conv2d_trans(X)))
```

In default, the transposed convolution layer uses a $k_h = k_w = 4$ kernel, a $s_h = s_w = 2$ strides, and a $p_h = p_w = 1$ padding. With a input shape of $n_h^{'} \times n_w^{'} = 16 \times 16$, the generator block will double input's width and height.

$$
\begin{aligned}
n_h^{'} \times n_w^{'} &= [(n_h k_h - (n_h - 1)(k_h - s_h) - 2p_h] \times [(n_w k_w - (n_w - 1)(k_w - s_w) - 2p_w] \\
&= [(k_h + s_h(n_h - 1) - 2p_h] \times [(k_w + s_w(n_w - 1) - 2p_w] \\
&= [(4 + 2 \times (16 - 1) - 2 \times 1] \times [(4 + 2 \times (16 - 1) - 2 \times 1] \\
&= 32 \times 32.
\end{aligned}
\tag{16.2.1}
$$

```
x = np.zeros((2, 3, 16, 16))
g_blk = G_block(20)
g_blk.initialize()
g_blk(x).shape
```

```
(2, 20, 32, 32)
```

If changing the transposed convolution layer to a 4×4 kernel, 1×1 strides and zero padding. With a input size of 1×1, the output will have its width and height increased by 3 respectively.

```
x = np.zeros((2, 3, 1, 1))
g_blk = G_block(20, strides=1, padding=0)
g_blk.initialize()
g_blk(x).shape
```

```
(2, 20, 4, 4)
```

The generator consists of four basic blocks that increase input's both width and height from 1 to 32. At the same time, it first projects the latent variable into 64×8 channels, and then halve the channels each time. At last, a transposed convolution layer is used to generate the output. It further doubles the width and height to match the desired 64×64 shape, and reduces the channel size to 3. The tanh activation function is applied to project output values into the $(-1, 1)$ range.

```
n_G = 64
net_G = nn.Sequential()
net_G.add(G_block(n_G*8, strides=1, padding=0),  # output: (64*8, 4, 4)
          G_block(n_G*4),  # output: (64*4, 8, 8)
          G_block(n_G*2),  # output: (64*2, 16, 16)
          G_block(n_G),   # output: (64, 32, 32)
          nn.Conv2DTranspose(
              3, kernel_size=4, strides=2, padding=1, use_bias=False,
              activation='tanh'))  # output: (3, 64, 64)
```

Generate a 100 dimensional latent variable to verify the generator's output shape.

```
x = np.zeros((1, 100, 1, 1))
net_G.initialize()
net_G(x).shape
```

```
(1, 3, 64, 64)
```

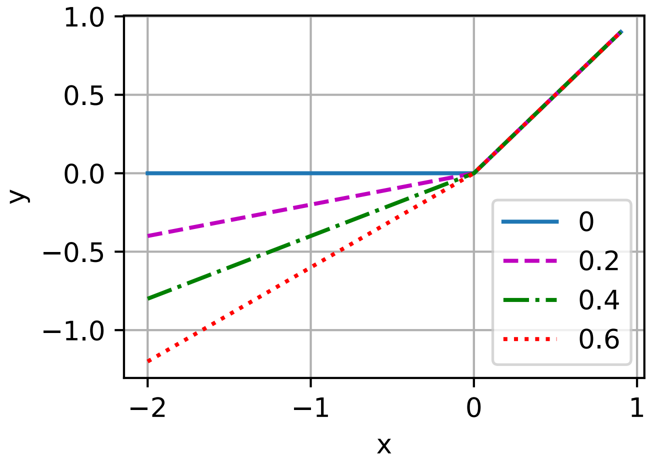
16.2.3 Discriminator

The discriminator is a normal convolutional network network except that it uses a leaky ReLU as its activation function. Given $\alpha \in [0, 1]$, its definition is

$$\text{leaky ReLU}(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha x & \text{otherwise} \end{cases}. \tag{16.2.2}$$

As it can be seen, it is normal ReLU if $\alpha = 0$, and an identity function if $\alpha = 1$. For $\alpha \in (0, 1)$, leaky ReLU is a nonlinear function that give a non-zero output for a negative input. It aims to fix the "dying ReLU" problem that a neuron might always output a negative value and therefore cannot make any progress since the gradient of ReLU is 0.

```
alphas = [0, 0.2, 0.4, .6, .8, 1]
x = np.arange(-2, 1, 0.1)
Y = [nn.LeakyReLU(alpha)(x).asnumpy() for alpha in alphas]
d2l.plot(x.asnumpy(), Y, 'x', 'y', alphas)
```



The basic block of the discriminator is a convolution layer followed by a batch normalization layer and a leaky ReLU activation. The hyper-parameters of the convolution layer are similar to the transpose convolution layer in the generator block.

```
class D_block(nn.Block):
    def __init__(self, channels, kernel_size=4, strides=2,
                 padding=1, alpha=0.2, **kwargs):
        super(D_block, self).__init__(**kwargs)
        self.conv2d = nn.Conv2D(
            channels, kernel_size, strides, padding, use_bias=False)
```

(continues on next page)

(continued from previous page)

```
        self.batch_norm = nn.BatchNorm()
        self.activation = nn.LeakyReLU(alpha)

    def forward(self, X):
        return self.activation(self.batch_norm(self.conv2d(X)))
```

A basic block with default settings will halve the width and height of the inputs, as we demonstrated in Section 6.3. For example, given a input shape $n_h = n_w = 16 $, with a kernel shape $k_h = k_w = 4$, a stride shape $s_h = s_w = 2$, and a padding shape $p_h = p_w = 1$, the output shape will be:

$$\begin{aligned}
n_h^{'} \times n_w^{'} &= \lfloor(n_h - k_h + 2p_h + s_h)/s_h\rfloor \times \lfloor(n_w - k_w + 2p_w + s_w)/s_w\rfloor \\
&= \lfloor(16 - 4 + 2 \times 1 + 2)/2\rfloor \times \lfloor(16 - 4 + 2 \times 1 + 2)/2\rfloor \\
&= 8 \times 8.
\end{aligned} \tag{16.2.3}$$

```
x = np.zeros((2, 3, 16, 16))
d_blk = D_block(20)
d_blk.initialize()
d_blk(x).shape
```

```
(2, 20, 8, 8)
```

The discriminator is a mirror of the generator.

```
n_D = 64
net_D = nn.Sequential()
net_D.add(D_block(n_D),   # output: (64, 32, 32)
          D_block(n_D*2),  # output: (64*2, 16, 16)
          D_block(n_D*4),  # output: (64*4, 8, 8)
          D_block(n_D*8),  # output: (64*8, 4, 4)
          nn.Conv2D(1, kernel_size=4, use_bias=False))  # output: (1, 1, 1)
```

It uses a convolution layer with output channel 1 as the last layer to obtain a single prediction value.

```
x = np.zeros((1, 3, 64, 64))
net_D.initialize()
net_D(x).shape
```

```
(1, 1, 1, 1)
```

16.2.4 Training

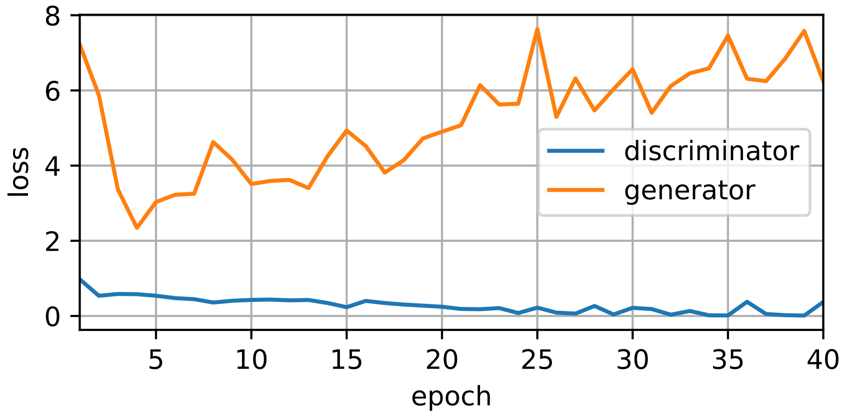
Compared to the basic GAN in Section 16.1, we use the same learning rate for both generator and discriminator since they are similar to each other. In addition, we change β_1 in Adam (Section 11.10) from 0.9 to 0.5. It decreases the smoothness of the momentum, the exponentially weighted moving average of past gradients, to take care of the rapid changing gradients because the generator and the discriminator fight with each other. Besides, the random generated noise Z, is a 4-D tensor and we are using GPU to accelerate the computation.

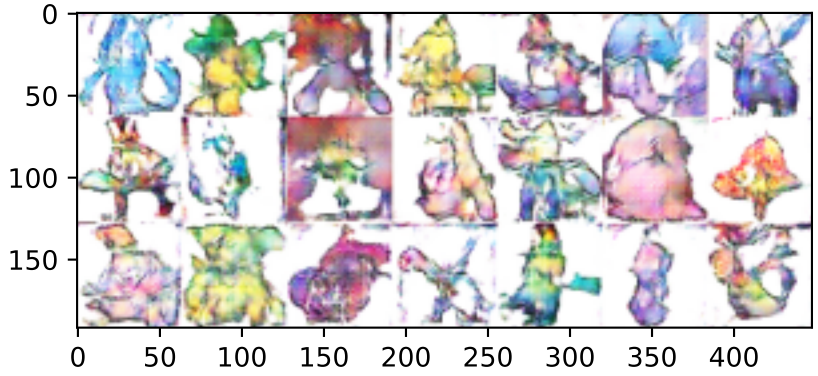
```
def train(net_D, net_G, data_iter, num_epochs, lr, latent_dim,
          ctx=d2l.try_gpu()):
    loss = gluon.loss.SigmoidBCELoss()
    net_D.initialize(init=init.Normal(0.02), force_reinit=True, ctx=ctx)
    net_G.initialize(init=init.Normal(0.02), force_reinit=True, ctx=ctx)
    trainer_hp = {'learning_rate': lr, 'beta1': 0.5}
    trainer_D = gluon.Trainer(net_D.collect_params(), 'adam', trainer_hp)
    trainer_G = gluon.Trainer(net_G.collect_params(), 'adam', trainer_hp)
    animator = d2l.Animator(xlabel='epoch', ylabel='loss',
                            xlim=[1, num_epochs], nrows=2, figsize=(5, 5),
                            legend=['discriminator', 'generator'])
    animator.fig.subplots_adjust(hspace=0.3)
    for epoch in range(1, num_epochs+1):
        # Train one epoch
        timer = d2l.Timer()
        metric = d2l.Accumulator(3)  # loss_D, loss_G, num_examples
        for X, _ in data_iter:
            batch_size = X.shape[0]
            Z = np.random.normal(0, 1, size=(batch_size, latent_dim, 1, 1))
            X, Z = X.as_in_context(ctx), Z.as_in_context(ctx),
            metric.add(d2l.update_D(X, Z, net_D, net_G, loss, trainer_D),
                       d2l.update_G(Z, net_D, net_G, loss, trainer_G),
                       batch_size)
        # Show generated examples
        Z = np.random.normal(0, 1, size=(21, latent_dim, 1, 1), ctx=ctx)
        # Noramlize the synthetic data to N(0, 1)
        fake_x = net_G(Z).transpose(0, 2, 3, 1)/2+0.5
        imgs = np.concatenate(
            [np.concatenate([fake_x[i * 7 + j] for j in range(7)], axis=1)
             for i in range(len(fake_x)//7)], axis=0)
        animator.axes[1].cla()
        animator.axes[1].imshow(imgs.asnumpy())
        # Show the losses
        loss_D, loss_G = metric[0]/metric[2], metric[1]/metric[2]
        animator.add(epoch, (loss_D, loss_G))
    print('loss_D %.3f, loss_G %.3f, %d examples/sec on %s' % (
        loss_D, loss_G, metric[2]/timer.stop(), ctx))
```

Now let's train the model.

```
latent_dim, lr, num_epochs = 100, 0.005, 40
train(net_D, net_G, data_iter, num_epochs, lr, latent_dim)
```

```
loss_D 0.373, loss_G 6.242, 2687 examples/sec on gpu(0)
```





Summary

- DCGAN architecture has four convolutional layers for the Discriminator and four "fractionally-strided" convolutional layers for the Generator.
- The Discriminator is a 4-layer strided convolutions with batch normalization (except its input layer) and leaky ReLU activations.
- Leaky ReLU is a nonlinear function that give a non-zero output for a negative input. It aims to fix the "dying ReLU" problem and helps the gradients flow easier through the architecture.

Exercises

- What will happen if we use standard ReLU activation rather than leaky ReLU?
- Apply DCGAN on Fashion-MNIST and see which category works well and which does not.

17 | Appendix: Mathematics for Deep Learning

Brent Werness (*Amazon*), **Rachel Hu** (*Amazon*), and authors of this book

One of the wonderful parts of modern deep learning is the fact that much of it can be understood and used without a full understanding of the mathematics below it. This is a sign that the field is maturing. Just as most software developers no longer need to worry about the theory of computable functions, neither should deep learning practitioners need to worry about the theoretical foundations of maximum likelihood learning.

But, we are not quite there yet.

In practice, you will sometimes need to understand how architectural choices influence gradient flow, or the implicit assumptions you make by training with a certain loss function. You might need to know what in the world entropy measures, and how it can help you understand exactly what bits-per-character means in your model. These all require deeper mathematical understanding.

This appendix aims to provide you the mathematical background you need to understand the core theory of modern deep learning, but it is not exhaustive. We will begin with examining linear algebra in greater depth. We develop a geometric understanding of all the common linear algebraic objects and operations that will enable us to visualize the effects of various transformations on our data. A key element is the development of the basics of eigen-decompositions.

We next develop the theory of differential calculus to the point that we can fully understand why the gradient is the direction of steepest descent, and why back-propagation takes the form it does. Integral calculus is then discussed to the degree needed to support our next topic, probability theory.

Problems encountered in practice frequently are not certain, and thus we need a language to speak about uncertain things. We review the theory of random variables and the most commonly encountered distributions so we may discuss models probabilistically. This provides the foundation for the naive Bayes classifier, a probabilistic classification technique.

Closely related to probability theory is the study of statistics. While statistics is far too large a field to do justice in a short section, we will introduce fundamental concepts that all machine learning practitioners should be aware of, in particular: evaluating and comparing estimators, conducting hypothesis tests, and constructing confidence intervals.

Last, we turn to the topic of information theory, which is the mathematical study of information storage and transmission. This provides the core language by which we may discuss quantitatively how much information a model holds on a domain of discourse.

Taken together, these form the core of the mathematical concepts needed to begin down the path towards a deep understanding of deep learning.

17.1 Geometry and Linear Algebraic Operations

In Section 2.3, we encountered the basics of linear algebra and saw how it could be used to express common operations for transforming our data. Linear algebra is one of the key mathematical pillars underlying much of the work that we do deep learning and in machine learning more broadly. While Section 2.3 contained enough machinery to communicate the mechanics of modern deep learning models, there is a lot more to the subject. In this section, we will go deeper, highlighting some geometric interpretations of linear algebra operations, and introducing a few fundamental concepts, including of eigenvalues and eigenvectors.

17.1.1 Geometry of Vectors

First, we need to discuss the two common geometric interpretations of vectors, as either points or directions in space. Fundamentally, a vector is a list of numbers such as the Python list below.

```
v = [1, 7, 0, 1]
```

Mathematicians most often write this as either a *column* or *row* vector, which is to say either as

$$\mathbf{x} = \begin{bmatrix}1\\7\\0\\1\end{bmatrix}, \tag{17.1.1}$$

or

$$\mathbf{x}^\top = \begin{bmatrix}1 & 7 & 0 & 1\end{bmatrix}. \tag{17.1.2}$$

These often have different interpretations, where data points are column vectors and weights used to form weighted sums are row vectors. However, it can be beneficial to be flexible. Matrices are useful data structures: they allow us to organize data that have different modalities of variation. For example, rows in our matrix might correspond to different houses (data points), while columns might correspond to different attributes. This should sound familiar if you have ever used spreadsheet software or have read Section 2.2. Thus, although the default orientation of a single vector is a column vector, in a matrix that represents a tabular dataset, it is more conventional to treat each data point as a row vector in the matrix. And, as we will see in later chapters, this convention will enable common deep learning practices. For example, along the outermost axis of an `ndarray`, we can access or enumerate minibatches of data points, or just data points if no minibatch exists.

Given a vector, the first interpretation that we should give it is as a point in space. In two or three dimensions, we can visualize these points by using the components of the vectors to define the location of the points in space compared to a fixed reference called the *origin*. This can be seen in Fig. 17.1.1.

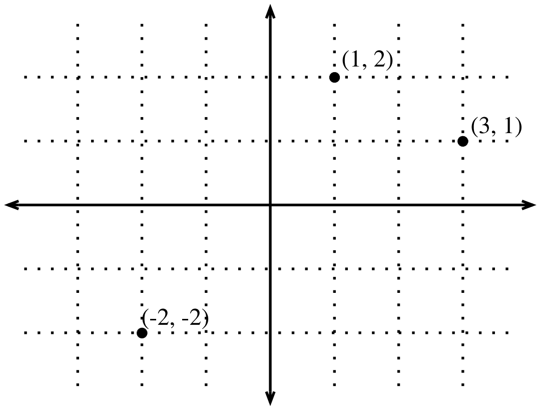


Fig. 17.1.1: An illustration of visualizing vectors as points in the plane. The first component of the vector gives the x-coordinate, the second component gives the y-coordinate. Higher dimensions are analogous, although much harder to visualize.

This geometric point of view allows us to consider the problem on a more abstract level. No longer faced with some insurmountable seeming problem like classifying pictures as either cats or dogs, we can start considering tasks abstractly as collections of points in space and picturing the task as discovering how to separate two distinct clusters of points.

In parallel, there is a second point of view that people often take of vectors: as directions in space. Not only can we think of the vector $\mathbf{v} = [2, 3]^\top$ as the location 2 units to the right and 3 units up from the origin, we can also think of it as the direction itself to take 2 steps to the right and 3 steps up. In this way, we consider all the vectors in figure Fig. 17.1.2 the same.

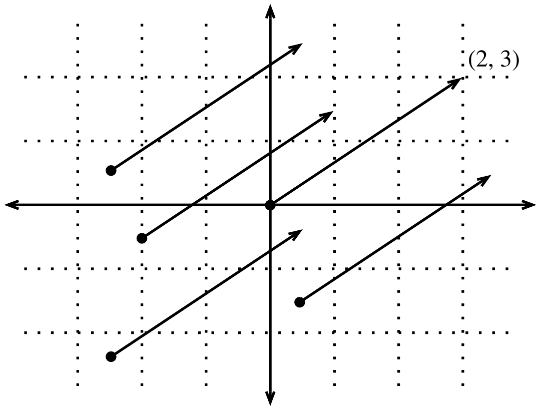


Fig. 17.1.2: Any vector can be visualized as an arrow in the plane. In this case, every vector drawn is a representation of the vector $(2, 3)$.

One of the benefits of this shift is that we can make visual sense of the act of vector addition. In particular, we follow the directions given by one vector, and then follow the directions given by the other, as is seen in Fig. 17.1.3.

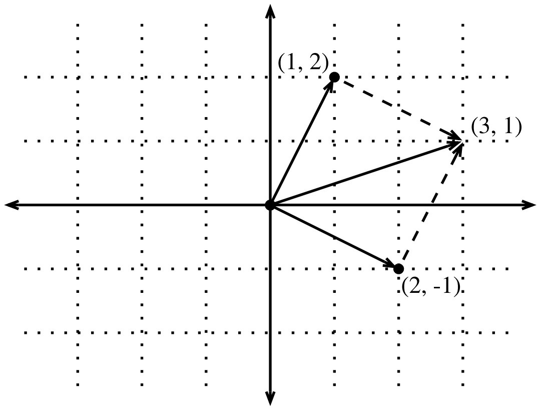


Fig. 17.1.3: We can visualize vector addition by first following one vector, and then another.

Vector subtraction has a similar interpretation. By considering the identity that $\mathbf{u} = \mathbf{v} + (\mathbf{u} - \mathbf{v})$, we see that the vector $\mathbf{u} - \mathbf{v}$ is the direction that takes us from the point $\mathbf{u}$ to the point $\mathbf{v}$.

17.1.2 Dot Products and Angles

As we saw in Section 2.3, if we take two column vectors say $\mathbf{u}$ and $\mathbf{v}$, we can form their dot product by computing:

$$\mathbf{u}^\top\mathbf{v} = \sum_i u_i \cdot v_i. \tag{17.1.3}$$

Because (17.1.3) is symmetric, we will mirror the notation of classical multiplication and write

$$\mathbf{u} \cdot \mathbf{v} = \mathbf{u}^\top\mathbf{v} = \mathbf{v}^\top\mathbf{u}, \tag{17.1.4}$$

to highlight the fact that exchanging the order of the vectors will yield the same answer.

The dot product (17.1.3) also admits a geometric interpretation: it is closely related to the angle between two vectors. Consider the angle shown in Fig. 17.1.4.

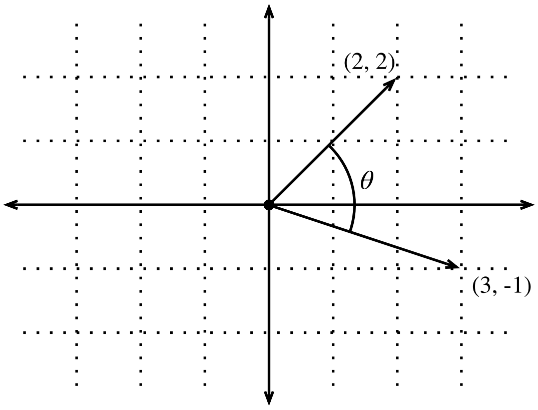


Fig. 17.1.4: Between any two vectors in the plane there is a well defined angle θ. We will see this angle is intimately tied to the dot product.

To start, let's consider two specific vectors:

$$\mathbf{v} = (r, 0) \text{ and } \mathbf{w} = (s\cos(\theta), s\sin(\theta)). \tag{17.1.5}$$

The vector $\mathbf{v}$ is length r and runs parallel to the x-axis, and the vector $\mathbf{w}$ is of length s and at angle θ with the x-axis.

If we compute the dot product of these two vectors, we see that

$$\mathbf{v} \cdot \mathbf{w} = rs\cos(\theta) = \|\mathbf{v}\|\|\mathbf{w}\|\cos(\theta). \tag{17.1.6}$$

With some simple algebraic manipulation, we can rearrange terms to obtain

$$\theta = \arccos\left(\frac{\mathbf{v}\cdot\mathbf{w}}{\|\mathbf{v}\|\|\mathbf{w}\|}\right). \tag{17.1.7}$$

In short, for these two specific vectors, the dot product combined with the norms tell us the angle between the two vectors. This same fact is true in general. We will not derive the expression here, however, if we consider writing $\|\mathbf{v} - \mathbf{w}\|^2$ in two ways: one with the dot product, and the other geometrically using the law of cosines, we can obtain the full relationship. Indeed, for any two vectors $\mathbf{v}$ and $\mathbf{w}$, the angle between the two vectors is

$$\theta = \arccos\left(\frac{\mathbf{v}\cdot\mathbf{w}}{\|\mathbf{v}\|\|\mathbf{w}\|}\right). \tag{17.1.8}$$

This is a nice result since nothing in the computation references two-dimensions. Indeed, we can use this in three or three million dimensions without issue.

As a simple example, let's see how to compute the angle between a pair of vectors:

```
%matplotlib inline
import d2l
from IPython import display
from mxnet import gluon, np, npx
npx.set_np()

def angle(v, w):
    return np.arccos(v.dot(w) / (np.linalg.norm(v) * np.linalg.norm(w)))

angle(np.array([0, 1, 2]), np.array([2, 3, 4]))
```

```
array(0.41899002)
```

We will not use it right now, but it is useful to know that we will refer to vectors for which the angle is $\pi/2$ (or equivalently $90°$) as being *orthogonal*. By examining the equation above, we see that this happens when $\theta = \pi/2$, which is the same thing as $\cos(\theta) = 0$. The only way this can happen is if the dot product itself is zero, and two vectors are orthogonal if and only if $\mathbf{v}\cdot\mathbf{w} = 0$. This will prove to be a helpful formula when understanding objects geometrically.

It is reasonable to ask: why is computing the angle useful? The answer comes in the kind of invariance we expect data to have. Consider an image, and a duplicate image, where every pixel value is the same but 10% the brightness. The values of the individual pixels are in general far from the original values. Thus, if one computed the distance between the original image and the darker one, the distance can be large.

However, for most ML applications, the *content* is the same—it is still an image of a cat as far as a cat/dog classifier is concerned. However, if we consider the angle, it is not hard to see that for any vector $\mathbf{v}$, the angle between $\mathbf{v}$ and $0.1 \cdot \mathbf{v}$ is zero. This corresponds to the fact that scaling vectors keeps the same direction and just changes the length. The angle considers the darker image identical.

Examples like this are everywhere. In text, we might want the topic being discussed to not change if we write twice as long of document that says the same thing. For some encoding (such as counting the number of occurrences of words in some vocabulary), this corresponds to a doubling of the vector encoding the document, so again we can use the angle.

Cosine Similarity

In ML contexts where the angle is employed to measure the closeness of two vectors, practitioners adopt the term *cosine similarity* to refer to the portion

$$\cos(\theta) = \frac{\mathbf{v}\cdot\mathbf{w}}{\|\mathbf{v}\|\|\mathbf{w}\|}. \tag{17.1.9}$$

The cosine takes a maximum value of 1 when the two vectors point in the same direction, a minimum value of -1 when they point in opposite directions, and a value of 0 when the two vectors are orthogonal. Note that if the components of high-dimensional vectors are sampled randomly with mean 0, their cosine will nearly always be close to 0.

17.1.3 Hyperplanes

In addition to working with vectors, another key object that you must understand to go far in linear algebra is the *hyperplane*, a generalization to higher dimensions of a line (two dimensions) or of a plane (three dimensions). In an d-dimensional vector space, a hyperplane has $d - 1$ dimensions and divides the space into two half-spaces.

Let's start with an example. Suppose that we have a column vector $\mathbf{w} = [2, 1]^\top$. We want to know, "what are the points $\mathbf{v}$ with $\mathbf{w}\cdot\mathbf{v} = 1$?" By recalling the connection between dot products and angles above (17.1.8), we can see that this is equivalent to

$$\|\mathbf{v}\|\|\mathbf{w}\|\cos(\theta) = 1 \iff \|\mathbf{v}\|\cos(\theta) = \frac{1}{\|\mathbf{w}\|} = \frac{1}{\sqrt{5}}. \tag{17.1.10}$$

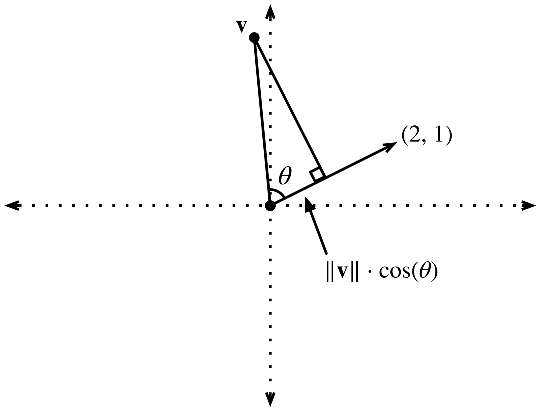


Fig. 17.1.5: Recalling trigonometry, we see the formula $\|\mathbf{v}\|\cos(\theta)$ is the length of the projection of the vector $\mathbf{v}$ onto the direction of $\mathbf{w}$

If we consider the geometric meaning of this expression, we see that this is equivalent to saying that the length of the projection of $\mathbf{v}$ onto the direction of $\mathbf{w}$ is exactly $1/\|\mathbf{w}\|$, as is shown in Fig. 17.1.5. The set of all points where this is true is a line at right angles to the vector $\mathbf{w}$. If we wanted, we could find the equation for this line and see that it is $2x + y = 1$ or equivalently $y = 1 - 2x$.

If we now look at what happens when we ask about the set of points with $\mathbf{w} \cdot \mathbf{v} > 1$ or $\mathbf{w} \cdot \mathbf{v} < 1$, we can see that these are cases where the projections are longer or shorter than $1/\|\mathbf{w}\|$, respectively. Thus, those two inequalities define either side of the line. In this way, we have found a way to cut our space into two halves, where all the points on one side have dot product below a threshold, and the other side above as we see in Fig. 17.1.6.

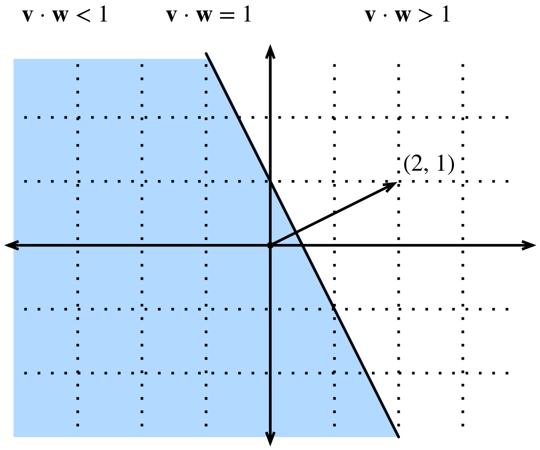

$\mathbf{v} \cdot \mathbf{w} < 1$
$\mathbf{v} \cdot \mathbf{w} = 1$
$\mathbf{v} \cdot \mathbf{w} > 1$
(2, 1)

Fig. 17.1.6: If we now consider the inequality version of the expression, we see that our hyperplane (in this case: just a line) separates the space into two halves.

The story in higher dimension is much the same. If we now take $\mathbf{w} = [1, 2, 3]^\top$ and ask about the points in three dimensions with $\mathbf{w} \cdot \mathbf{v} = 1$, we obtain a plane at right angles to the given vector $\mathbf{w}$. The two inequalities again define the two sides of the plane as is shown in Fig. 17.1.7.

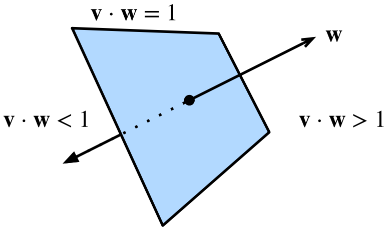

$\mathbf{v} \cdot \mathbf{w} = 1$
$\mathbf{w}$
$\mathbf{v} \cdot \mathbf{w} < 1$
$\mathbf{v} \cdot \mathbf{w} > 1$

Fig. 17.1.7: Hyperplanes in any dimension separate the space into two halves.

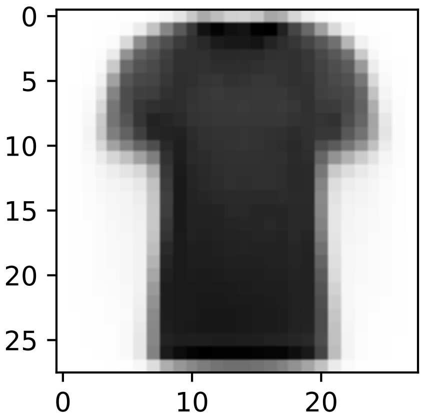
While our ability to visualize runs out at this point, nothing stops us from doing this in tens, hundreds, or billions of dimensions. This occurs often when thinking about machine learned models. For instance, we can understand linear classification models like those from Section 3.4, as methods to find hyperplanes that separate the different target classes. In this context, such hyperplanes are often referred to as *decision planes*. The majority of deep learned classification models end with a linear layer fed into a softmax, so one can interpret the role of the deep neural network to be to find a non-linear embedding such that the target classes can be separated cleanly by hyperplanes.

To give a hand-built example, notice that we can produce a reasonable model to classify tiny images of t-shirts and trousers from the Fashion MNIST dataset (seen in Section 3.5) by just taking the vector between their means to define the decision plane and eyeball a crude threshold. First we will load the data and compute the averages.

```
# Load in the dataset
train = gluon.data.vision.FashionMNIST(train=True)
test = gluon.data.vision.FashionMNIST(train=False)

X_train_0 = np.stack([x[0] for x in train if x[1] == 0]).astype(float)
X_train_1 = np.stack([x[0] for x in train if x[1] == 1]).astype(float)
X_test = np.stack(
    [x[0] for x in test if x[1] == 0 or x[1] == 1]).astype(float)
y_test = np.stack(
    [x[1] for x in test if x[1] == 0 or x[1] == 1]).astype(float)

# Compute averages
ave_0 = np.mean(X_train_0, axis=0)
ave_1 = np.mean(X_train_1, axis=0)
```

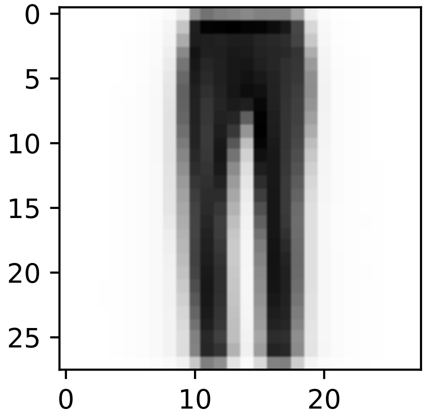
It can be informative to examine these averages in detail, so let's plot what they look like. In this case, we see that the average indeed resembles a blurry image of a t-shirt.

```
# Plot average t-shirt
d2l.set_figsize()
d2l.plt.imshow(ave_0.reshape(28, 28).tolist(), cmap='Greys')
d2l.plt.show()
```



In the second case, we again see that the average resembles a blurry image of trousers.

```
# Plot average trousers
d2l.plt.imshow(ave_1.reshape(28, 28).tolist(), cmap='Greys')
d2l.plt.show()
```



In a fully machine learned solution, we would learn the threshold from the dataset. In this case, I simply eyeballed a threshold that looked good on the training data by hand.

```
# Print test set accuracy with eyeballed threshold
w = (ave_1 - ave_0).T
predictions = X_test.reshape(2000, -1).dot(w.flatten()) > -1500000

# Accuracy
np.mean(predictions.astype(y_test.dtype) == y_test, dtype=np.float64)
```

```
array(0.801, dtype=float64)
```

17.1.4 Geometry of Linear Transformations

Through Section 2.3 and the above discussions, we have a solid understanding of the geometry of vectors, lengths, and angles. However, there is one important object we have omitted discussing, and that is a geometric understanding of linear transformations represented by matrices. Fully internalizing what matrices can do to transform data between two potentially different high dimensional spaces takes significant practice, and is beyond the scope of this appendix. However, we can start building up intuition in two dimensions.

Suppose that we have some matrix:

$$\mathbf{A} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}. \tag{17.1.11}$$

If we want to apply this to an arbitrary vector $\mathbf{v} = [x, y]^\top$, we multiply and see that

$$\begin{aligned} \mathbf{A}\mathbf{v} & = \begin{bmatrix} a & b \\ c & d \end{bmatrix}\begin{bmatrix} x \\ y \end{bmatrix} \\ & = \begin{bmatrix} ax+by \\ cx+dy \end{bmatrix} \\ & = x\begin{bmatrix} a \\ c \end{bmatrix} + y\begin{bmatrix} b \\ d \end{bmatrix} \\ & = x\left\{\mathbf{A}\begin{bmatrix} 1 \\ 0 \end{bmatrix}\right\} + y\left\{\mathbf{A}\begin{bmatrix} 0 \\ 1 \end{bmatrix}\right\}. \end{aligned} \tag{17.1.12}$$

This may seem like an odd computation, where something clear became somewhat impenetrable. However, it tells us that we can write the way that a matrix transforms *any* vector in terms of how it transforms *two specific vectors*: $[1,0]^\top$ and $[0,1]^\top$. This is worth considering for a moment. We have essentially reduced an infinite problem (what happens to any pair of real numbers) to a finite one (what happens to these specific vectors). These vectors are an example a *basis*, where we can write any vector in our space as a weighted sum of these *basis vectors*.

Let's draw what happens when we use the specific matrix

$$\mathbf{A} = \begin{bmatrix} 1 & 2 \\ -1 & 3 \end{bmatrix}. \tag{17.1.13}$$

If we look at the specific vector $\mathbf{v} = [2,-1]^\top$, we see this is $2 \cdot [1,0]^\top + -1 \cdot [0,1]^\top$, and thus we know that the matrix A will send this to $2(\mathbf{A}[1,0]^\top) + -1(\mathbf{A}[0,1])^\top = 2[1,-1]^\top - [2,3]^\top = [0,-5]^\top$. If we follow this logic through carefully, say by considering the grid of all integer pairs of points, we see that what happens is that the matrix multiplication can skew, rotate, and scale the grid, but the grid structure must remain as you see in Fig. 17.1.8.

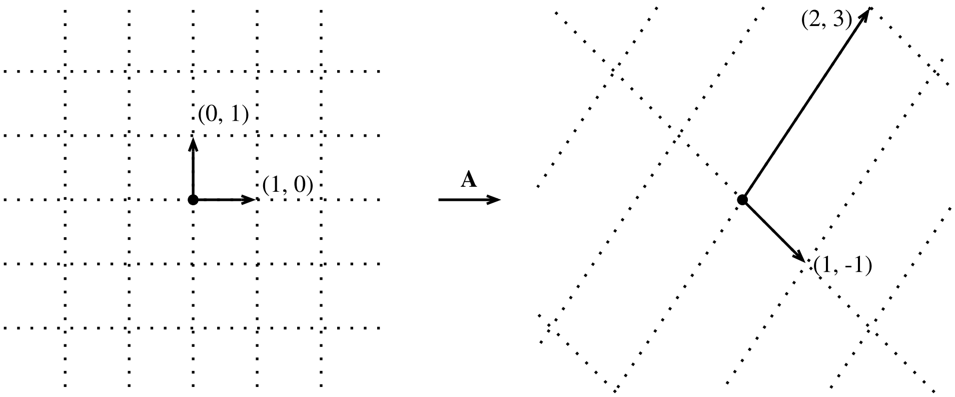


Fig. 17.1.8: The matrix **A** acting on the given basis vectors. Notice how the entire grid is transported along with it.

This is the most important intuitive point to internalize about linear transformations represented by matrices. Matrices are incapable of distorting some parts of space differently than others. All they can do is take the original coordinates on our space and skew, rotate, and scale them.

Some distortions can be severe. For instance the matrix

$$\mathbf{B} = \begin{bmatrix} 2 & -1 \\ 4 & -2 \end{bmatrix}, \tag{17.1.14}$$

compresses the entire two-dimensional plane down to a single line. Identifying and working with such transformations are the topic of a later section, but geometrically we can see that this is fundamentally different from the types of transformations we saw above. For instance, the result from matrix **A** can be "bent back" to the original grid. The results from matrix **B** cannot because we will never know where the vector $[1,2]^\top$ came from—was it $[1,1]^\top$ or $[0,-1]^\top$?

While this picture was for a 2×2 matrix, nothing prevents us from taking the lessons learned into higher dimensions. If we take similar basis vectors like $[1,0,\ldots,0]$ and see where our matrix sends them, we can start to get a feeling for how the matrix multiplication distorts the entire space in whatever dimension space we are dealing with.

17.1.5 Linear Dependence

Consider again the matrix

$$\mathbf{B} = \begin{bmatrix}2 & -1\\ 4 & -2\end{bmatrix}. \tag{17.1.15}$$

This compresses the entire plane down to live on the single line $y = 2x$. The question now arises: is there some way we can detect this just looking at the matrix itself? The answer is that indeed we can. Let's take $\mathbf{b}_1 = [2, 4]^\top$ and $\mathbf{b}_2 = [-1, -2]^\top$ be the two columns of **B**. Remember that we can write everything transformed by the matrix **B** as a weighted sum of the columns of the matrix: like $a_1\mathbf{b}_1 + a_2\mathbf{b}_2$. We call this a *linear combination*. The fact that $\mathbf{b}_1 = -2 \cdot \mathbf{b}_2$ means that we can write any linear combination of those two columns entirely in terms of say $\mathbf{b}_2$ since

$$a_1\mathbf{b}_1 + a_2\mathbf{b}_2 = -2a_1\mathbf{b}_2 + a_2\mathbf{b}_2 = (a_2 - 2a_1)\mathbf{b}_2. \tag{17.1.16}$$

This means that one of the columns is, in a sense, redundant because it does not define a unique direction in space. This should not surprise us too much since we already saw that this matrix collapses the entire plane down into a single line. Moreover, we see that the linear dependence $\mathbf{b}_1 = -2 \cdot \mathbf{b}_2$ captures this. To make this more symmetrical between the two vectors, we will write this as

$$\mathbf{b}_1 + 2 \cdot \mathbf{b}_2 = 0. \tag{17.1.17}$$

In general, we will say that a collection of vectors $\mathbf{v}_1, \dots \mathbf{v}_k$ are *linearly dependent* if there exist coefficients $a_1, \dots, a_k$ *not all equal to zero* so that

$$\sum_{i=1}^{k} a_i\mathbf{v_i} = 0. \tag{17.1.18}$$

In this case, we can solve for one of the vectors in terms of some combination of the others, and effectively render it redundant. Thus, a linear dependence in the columns of a matrix is a witness to the fact that our matrix is compressing the space down to some lower dimension. If there is no linear dependence we say the vectors are *linearly independent*. If the columns of a matrix are linearly independent, no compression occurs and the operation can be undone.

17.1.6 Rank

If we have a general $n \times m$ matrix, it is reasonable to ask what dimension space the matrix maps into. A concept known as the *rank* will be our answer. In the previous section, we noted that a linear dependence bears witness to compression of space into a lower dimension and so we will be able to use this to define the notion of rank. In particular, the rank of a matrix **A** is the largest number of linearly independent columns amongst all subsets of columns. For example, the matrix

$$\mathbf{B} = \begin{bmatrix}2 & 4\\ -1 & -2\end{bmatrix}, \tag{17.1.19}$$

has $\mathrm{rank}(B) = 1$, since the two columns are linearly dependent, but either column by itself is not linearly dependent. For a more challenging example, we can consider

$$\mathbf{C} = \begin{bmatrix}1 & 3 & 0 & -1 & 0\\ -1 & 0 & 1 & 1 & -1\\ 0 & 3 & 1 & 0 & -1\\ 2 & 3 & -1 & -2 & 1\end{bmatrix}, \tag{17.1.20}$$

and show that $\mathbf{C}$ has rank two since, for instance, the first two columns are linearly independent, however any of the four collections of three columns are dependent.

This procedure, as described, is very inefficient. It requires looking at every subset of the columns of our given matrix, and thus is potentially exponential in the number of columns. Later we will see a more computationally efficient way to compute the rank of a matrix, but for now, this is sufficient to see that the concept is well defined and understand the meaning.

17.1.7 Invertibility

We have seen above that multiplication by a matrix with linearly dependent columns cannot be undone, i.e., there is no inverse operation that can always recover the input. However, multiplication by a full-rank matrix (i.e., some $\mathbf{A}$ that is $n \times n$ matrix with rank n), we should always be able to undo it. Consider the matrix

$$\mathbf{I} = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix}. \tag{17.1.21}$$

which is the matrix with ones along the diagonal, and zeros elsewhere. We call this the *identity* matrix. It is the matrix which leaves our data unchanged when applied. To find a matrix which undoes what our matrix $\mathbf{A}$ has done, we want to find a matrix $\mathbf{A}^{-1}$ such that

$$\mathbf{A}^{-1}\mathbf{A} = \mathbf{A}\mathbf{A}^{-1} = \mathbf{I}. \tag{17.1.22}$$

If we look at this as a system, we have $n \times n$ unknowns (the entries of $\mathbf{A}^{-1}$) and $n \times n$ equations (the equality that needs to hold between every entry of the product $\mathbf{A}^{-1}\mathbf{A}$ and every entry of $\mathbf{I}$) so we should generically expect a solution to exist. Indeed, in the next section we will see a quantity called the *determinant*, which has the property that as long as the determinant is not zero, we can find a solution. We call such a matrix $\mathbf{A}^{-1}$ the *inverse* matrix. As an example, if $\mathbf{A}$ is the general 2×2 matrix

$$\mathbf{A} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}, \tag{17.1.23}$$

then we can see that the inverse is

$$\frac{1}{ad-bc} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix}. \tag{17.1.24}$$

We can test to see this by seeing that multiplying by the inverse given by the formula above works in practice.

```
M = np.array([[1, 2], [1, 4]])
M_inv = np.array([[2, -1], [-0.5, 0.5]])
M_inv.dot(M)
```

```
array([[1., 0.],
       [0., 1.]])
```

Numerical Issues

While the inverse of a matrix is useful in theory, we must say that most of the time we do not wish to *use* the matrix inverse to solve a problem in practice. In general, there are far more numerically stable algorithms for solving linear equations like

$$\mathbf{A}\mathbf{x} = \mathbf{b}, \tag{17.1.25}$$

than computing the inverse and multiplying to get

$$\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}. \tag{17.1.26}$$

Just as division by a small number can lead to numerical instability, so can inversion of a matrix which is close to having low rank.

Moreover, it is common that the matrix $\mathbf{A}$ is *sparse*, which is to say that it contains only a small number of non-zero values. If we were to explore examples, we would see that this does not mean the inverse is sparse. Even if $\mathbf{A}$ was a 1 million by 1 million matrix with only 5 million non-zero entries (and thus we need only store those 5 million), the inverse will typically have almost every entry non-negative, requiring us to store all 1M^2 entries—that is 1 trillion entries!

While we do not have time to dive all the way into the thorny numerical issues frequently encountered when working with linear algebra, we want to provide you with some intuition about when to proceed with caution, and generally avoiding inversion in practice is a good rule of thumb.

17.1.8 Determinant

The geometric view of linear algebra gives an intuitive way to interpret a a fundamental quantity known as the *determinant*. Consider the grid image from before, but now with a highlighted region (Fig. 17.1.9).

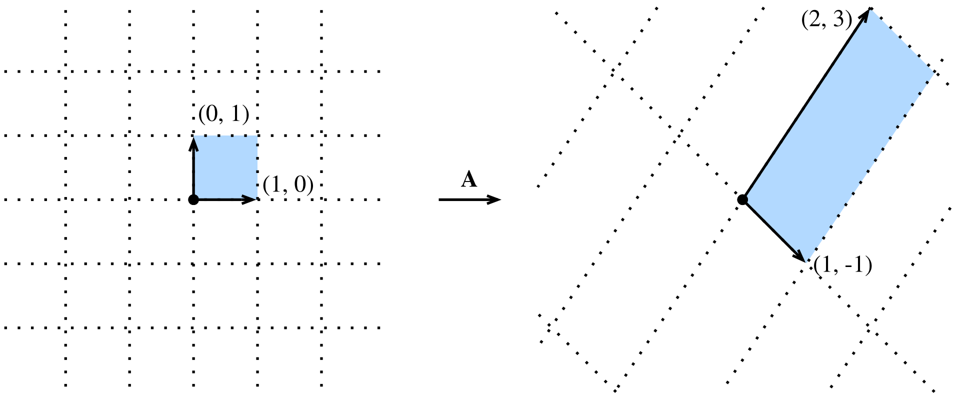


Fig. 17.1.9: The matrix $\mathbf{A}$ again distorting the grid. This time, I want to draw particular attention to what happens to the highlighted square.

Look at the highlighted square. This is a square with edges given by $(0, 1)$ and $(1, 0)$ and thus it has area one. After $\mathbf{A}$ transforms this square, we see that it becomes a parallelogram. There is no reason this parallelogram should have the same area that we started with, and indeed in the specific case shown here of

$$\mathbf{A} = \begin{bmatrix} 1 & -1 \\ 2 & 3 \end{bmatrix}, \tag{17.1.27}$$

it is an exercise in coordinate geometry to compute the area of this parallelogram and obtain that the area is 5.

In general, if we have a matrix

$$\mathbf{A} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}, \tag{17.1.28}$$

we can see with some computation that the area of the resulting parallelogram is $ad - bc$. This area is referred to as the *determinant*.

Let's check this quickly with some example code.

```
import numpy as np
np.linalg.det(np.array([[1, -1], [2, 3]]))
```

```
5.000000000000001
```

The eagle-eyed amongst us will notice that this expression can be zero or even negative. For the negative term, this is a matter of convention taken generally in mathematics: if the matrix flips the figure, we say the area is negated. Let's see now that when the determinant is zero, we learn more.

Let's consider

$$\mathbf{B} = \begin{bmatrix} 2 & 4 \\ -1 & -2 \end{bmatrix}. \tag{17.1.29}$$

If we compute the determinant of this matrix, we get $2\cdot(-2)-4\cdot(-1) = 0$. Given our understanding above, this makes sense. **B** compresses the square from the original image down to a line segment, which has zero area. And indeed, being compressed into a lower dimensional space is the only way to have zero area after the transformation. Thus we see the following result is true: a matrix A is invertible if and only if the determinant is not equal to zero.

As a final comment, imagine that we have any figure drawn on the plane. Thinking like computer scientists, we can decompose that figure into a collection of little squares so that the area of the figure is in essence just the number of squares in the decomposition. If we now transform that figure by a matrix, we send each of these squares to parallelograms, each one of which has area given by the determinant. We see that for any figure, the determinant gives the (signed) number that a matrix scales the area of any figure.

Computing determinants for larger matrices can be laborious, but the intuition is the same. The determinant remains the factor that $n \times n$ matrices scale n-dimensional volumes.

17.1.9 Tensors and Common Linear Algebra Operations

In Section 2.3 the concept of tensors was introduced. In this section, we will dive more deeply into tensor contractions (the tensor equivalent of matrix multiplication), and see how it can provide a unified view on a number of matrix and vector operations.

With matrices and vectors we knew how to multiply them to transform data. We need to have a similar definition for tensors if they are to be useful to us. Think about matrix multiplication:

$$\mathbf{C} = \mathbf{A}\mathbf{B}, \tag{17.1.30}$$

or equivalently

$$c_{i,j} = \sum_k a_{i,k}b_{k,j}. \tag{17.1.31}$$

This pattern is one we can repeat for tensors. For tensors, there is no one case of what to sum over that can be universally chosen, so we need specify exactly which indices we want to sum over. For instance we could consider

$$y_{il} = \sum_{jk} x_{ijkl}a_{jk}. \tag{17.1.32}$$

Such a transformation is called a *tensor contraction*. It can represent a far more flexible family of transformations that matrix multiplication alone.

As a often-used notational simplification, we can notice that the sum is over exactly those indices that occur more than once in the expression, thus people often work with *Einstein notation*, where the summation is implicitly taken over all repeated indices. This gives the compact expression:

$$y_{il} = x_{ijkl}a_{jk}. \tag{17.1.33}$$

Common Examples from Linear Algebra

Let's see how many of the linear algebraic definitions we have seen before can be expressed in this compressed tensor notation:

- $\mathbf{v} \cdot \mathbf{w} = \sum_i v_iw_i$
- $\|\mathbf{v}\|_2^2 = \sum_i v_iv_i$
- $(\mathbf{A}\mathbf{v})_i = \sum_j a_{ij}v_j$
- $(\mathbf{A}\mathbf{B})_{ik} = \sum_j a_{ij}b_{jk}$
- $\mathrm{tr}(\mathbf{A}) = \sum_i a_{ii}$

In this way, we can replace a myriad of specialized notations with short tensor expressions.

Expressing in Code

Tensors may flexibly be operated on in code as well. As seen in Section 2.3, we can create tensors as is shown below.

```
# Define tensors
B = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])
A = np.array([[1, 2], [3, 4]])
v = np.array([1, 2])

# Print out the shapes
A.shape, B.shape, v.shape
```

```
((2, 2), (2, 2, 3), (2,))
```

Einstein summation has been implemented directly via `np.einsum`. The indices that occurs in the Einstein summation can be passed as a string, followed by the tensors that are being acted upon. For instance, to implement matrix multiplication, we can consider the Einstein summation seen above ($\mathbf{A}\mathbf{v} = a_{ij}v_j$) and strip out the indices themselves to get the implementation:

```
# Reimplement matrix multiplication
np.einsum("ij, j -> i", A, v), A.dot(v)
```

```
(array([ 5, 11]), array([ 5, 11]))
```

This is a highly flexible notation. For instance if we want to compute what would be traditionally written as

$$c_{kl} = \sum_{ij} \mathbf{B}_{ijk}\mathbf{A}_{il}v_j. \tag{17.1.34}$$

it can be implemented via Einstein summation as:

```
np.einsum("ijk, il, j -> kl", B, A, v)
```

```
array([[ 90, 126],
       [102, 144],
       [114, 162]])
```

This notation is readable and efficient for humans, however bulky if for whatever reason we need to generate a tensor contraction programmatically. For this reason, `einsum` provides an alternative notation by providing integer indices for each tensor. For example, the same tensor contraction can also be written as:

```
np.einsum(B, [0, 1, 2], A, [0, 3], v, [1], [2, 3])
```

```
array([[ 90, 126],
       [102, 144],
       [114, 162]])
```

Either notation allows for concise and efficient representation of tensor contractions in code.

Summary

- Vectors can be interpreted geometrically as either points or directions in space.
- Dot products define the notion of angle to arbitrarily high-dimensional spaces.
- Hyperplanes are high-dimensional generalizations of lines and planes. They can be used to define decision planes that are often used as the last step in a classification task.
- Matrix multiplication can be geometrically interpreted as uniform distortions of the underlying coordinates. They represent a very restricted, but mathematically clean, way to transform vectors.
- Linear dependence is a way to tell when a collection of vectors are in a lower dimensional space than we would expect (say you have 3 vectors living in a 2-dimensional space). The rank of a matrix is the size of the largest subset of its columns that are linearly independent.

- When a matrix's inverse is defined, matrix inversion allows us to find another matrix that undoes the action of the first. Matrix inversion is useful in theory, but requires care in practice owing to numerical instability.
- Determinants allow us to measure how much a matrix expands or contracts a space. A nonzero determinant implies an invertible (non-singular) matrix and a zero-valued determinant means that the matrix is non-invertible (singular).
- Tensor contractions and Einstein summation provide for a neat and clean notation for expressing many of the computations that are seen in machine learning.

Exercises

1. What is the angle between

$$\vec{v}_1 = \begin{bmatrix} 1 \\ 0 \\ -1 \\ 2 \end{bmatrix}, \qquad \vec{v}_2 = \begin{bmatrix} 3 \\ 1 \\ 0 \\ 1 \end{bmatrix}? \tag{17.1.35}$$

2. True or false: $\begin{bmatrix} 1 & 2 \\ 0 & 1 \end{bmatrix}$ and $\begin{bmatrix} 1 & -2 \\ 0 & 1 \end{bmatrix}$ are inverses of one another?
3. Suppose that we draw a shape in the plane with area $100\textrm{m}^2$. What is the area after transforming the figure by the matrix

$$\begin{bmatrix} 2 & 3 \\ 1 & 2 \end{bmatrix}. \tag{17.1.36}$$

4. Which of the following sets of vectors are linearly independent?

- $\left\{\begin{pmatrix} 1 \\ 0 \\ -1 \end{pmatrix}, \begin{pmatrix} 2 \\ 1 \\ -1 \end{pmatrix}, \begin{pmatrix} 3 \\ 1 \\ 1 \end{pmatrix}\right\}$
- $\left\{\begin{pmatrix} 3 \\ 1 \\ 1 \end{pmatrix}, \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}\right\}$
- $\left\{\begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \\ -1 \end{pmatrix}, \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix}\right\}$

5. Suppose that you have a matrix written as $A = \begin{bmatrix} c \\ d \end{bmatrix} \cdot \begin{bmatrix} a & b \end{bmatrix}$ for some choice of values a, b, c, and d. True or false: the determinant of such a matrix is always 0?
6. The vectors $e_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$ and $e_2 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$ are orthogonal. What is the condition on a matrix A so that Ae_1 and Ae_2 are orthogonal?
7. How can you write $\textrm{tr}(\mathbf{A}^4)$ in Einstein notation for an arbitrary matrix A?

17.2 Eigendecompositions

Eigenvalues are often one of the most useful notions we will encounter when studying linear algebra, however, as a beginner, it is easy to overlook their importance. Below, we introduce eigendecomposition and try to convey some sense of just why it is so important.

Suppose that we have a matrix A with the following entries:

$$\mathbf{A} = \begin{bmatrix} 2 & 0 \\ 0 & -1 \end{bmatrix}. \tag{17.2.1}$$

If we apply A to any vector $\mathbf{v} = [x, y]^\top$, we obtain a vector $\mathbf{v}A = [2x, -y]^\top$. This has an intuitive interpretation: stretch the vector to be twice as wide in the x-direction, and then flip it in the y-direction.

However, there are *some* vectors for which something remains unchanged. Namely $[1, 0]^\top$ gets sent to $[2, 0]^\top$ and $[0, 1]^\top$ gets sent to $[0, -1]^\top$. These vectors are still in the same line, and the only modification is that the matrix stretches them by a factor of 2 and -1 respectively. We call such vectors *eigenvectors* and the factor they are stretched by *eigenvalues*.

In general, if we can find a number λ and a vector $\mathbf{v}$ such that

$$\mathbf{A}\mathbf{v} = \lambda\mathbf{v}. \tag{17.2.2}$$

We say that $\mathbf{v}$ is an eigenvector for A and λ is an eigenvalue.

17.2.1 Finding Eigenvalues

Let's figure out how to find them.
By subtracting off the $\lambda\vec{v}$ from both sides, and then factoring out the vector, we see the above is equivalent to:

$$(\mathbf{A} - \lambda\mathbf{I})\mathbf{v} = 0. \tag{17.2.3}$$

For (17.2.3) to happen, we see that $(\mathbf{A} - \lambda\mathbf{I})$ must compress some direction down to zero, hence it is not invertible, and thus the determinant is zero. Thus, we can find the *eigenvalues* by finding for what λ is $\det(\mathbf{A} - \lambda\mathbf{I}) = 0$. Once we find the eigenvalues, we can solve $\mathbf{A}\mathbf{v} = \lambda\mathbf{v}$ to find the associated *eigenvector(s)*.

An Example

Let's see this with a more challenging matrix

$$\mathbf{A} = \begin{bmatrix} 2 & 1 \\ 2 & 3 \end{bmatrix}. \tag{17.2.4}$$

If we consider $\det(\mathbf{A} - \lambda \mathbf{I}) = 0$, we see this is equivalent to the polynomial equation $0 = (2-\lambda)(3-\lambda) - 2 = (4-\lambda)(1-\lambda)$. Thus, two eigenvalues are 4 and 1. To find the associated vectors, we then need to solve

$$\begin{bmatrix} 2 & 1 \\ 2 & 3 \end{bmatrix}\begin{bmatrix}x \\ y\end{bmatrix} = \begin{bmatrix}x \\ y\end{bmatrix} \; \text{and} \; \begin{bmatrix} 2 & 2 \\ 1 & 3 \end{bmatrix}\begin{bmatrix}x \\ y\end{bmatrix} = \begin{bmatrix}4x \\ 4y\end{bmatrix}. \tag{17.2.5}$$

We can solve this with the vectors $[1, -1]^\top$ and $[1, 2]^\top$ respectively.

We can check this in code using the built-in `numpy.linalg.eig` routine.

```
%matplotlib inline
import d2l
from IPython import display
import numpy as np

np.linalg.eig(np.array([[2, 1], [2, 3]]))
```

```
(array([1., 4.]), array([[-0.70710678, -0.4472136 ],
        [ 0.70710678, -0.89442719]]))
```

Note that `numpy` normalizes the eigenvectors to be of length one, whereas we took ours to be of arbitrary length. Additionally, the choice of sign is arbitrary. However, the vectors computed are parallel to the ones we found by hand with the same eigenvalues.

17.2.2 Decomposing Matrices

Let's continue the previous example one step further. Let

$$\mathbf{W} = \begin{bmatrix} 1 & 1 \\ -1 & 2 \end{bmatrix}, \tag{17.2.6}$$

be the matrix where the columns are the eigenvectors of the matrix $\mathbf{A}$. Let

$$\boldsymbol{\Sigma} = \begin{bmatrix} 1 & 0 \\ 0 & 4 \end{bmatrix}, \tag{17.2.7}$$

be the matrix with the associated eigenvalues on the diagonal. Then the definition of eigenvalues and eigenvectors tells us that

$$\mathbf{A}\mathbf{W} = \mathbf{W} \boldsymbol{\Sigma}. \tag{17.2.8}$$

The matrix W is invertible, so we may multiply both sides by W^{-1} on the right, we see that we may write

$$\mathbf{A} = \mathbf{W} \boldsymbol{\Sigma} \mathbf{W}^{-1}. \tag{17.2.9}$$

In the next section we will see some nice consequences of this, but for now we need only know that such a decomposition will exist as long as we can find a full collection of linearly independent eigenvectors (so that W is invertible).

17.2.3 Operations on Eigendecompositions

One nice thing about eigendecompositions (17.2.9) is that we can write many operations we usually encounter cleanly in terms of the eigendecomposition.
As a first example, consider:

$$\mathbf{A}^n = \overbrace{\mathbf{A} \cdots \mathbf{A}}^{n \text{ times}} = \overbrace{(\mathbf{W}\boldsymbol{\Sigma}\mathbf{W}^{-1}) \cdots (\mathbf{W}\boldsymbol{\Sigma}\mathbf{W}^{-1})}^{n \text{ times}} = \mathbf{W}\overbrace{\boldsymbol{\Sigma} \cdots \boldsymbol{\Sigma}}^{n \text{ times}}\mathbf{W}^{-1} = \mathbf{W}\boldsymbol{\Sigma}^n\mathbf{W}^{-1}. \tag{17.2.10}$$

This tells us that for any positive power of a matrix, the eigendecomposition is obtained by just raising the eigenvalues to the same power. The same can be shown for negative powers, so if we want to invert a matrix we need only consider

$$\mathbf{A}^{-1} = \mathbf{W}\boldsymbol{\Sigma}^{-1}\mathbf{W}^{-1}, \tag{17.2.11}$$

or in other words, just invert each eigenvalue. This will work as long as each eigenvalue is non-zero, so we see that invertible is the same as having no zero eigenvalues.

Indeed, additional work can show that if $\lambda_1, \ldots, \lambda_n$ are the eigenvalues of a matrix, then the determinant of that matrix is

$$\det(\mathbf{A}) = \lambda_1 \cdots \lambda_n, \tag{17.2.12}$$

or the product of all the eigenvalues. This makes sense intuitively because whatever stretching **W** does, W^{-1} undoes it, so in the end the only stretching that happens is by multiplication by the diagonal matrix $\boldsymbol{\Sigma}$, which stretches volumes by the product of the diagonal elements.

Finally, recall that the rank was the maximum number of linearly independent columns of your matrix. By examining the eigendecomposition closely, we can see that the rank is the same as the number of non-zero eigenvalues of **A**.

The examples could continue, but hopefully the point is clear: eigendecompositions can simplify many linear-algebraic computations and are a fundamental operation underlying many numerical algorithms and much of the analysis that we do in linear algebra.

17.2.4 Eigendecompositions of Symmetric Matrices

It is not always possible to find enough linearly independent eigenvectors for the above process to work. For instance the matrix

$$\mathbf{A} = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}, \tag{17.2.13}$$

has only a single eigenvector, namely $(0, 1)$. To handle such matrices, we require more advanced techniques than we can cover (such as the Jordan Normal Form, or Singular Value Decomposition). We will often need to restrict our attention to those matrices where we can guarantee the existence of a full set of eigenvectors.

The most commonly encountered family are the *symmetric matrices*, which are those matrices where $\mathbf{A} = \mathbf{A}^\top$. In this case, we may take W to be an *orthogonal matrix*—a matrix whose columns are all length one vectors that are at right angles to one another, where $\mathbf{W}^\top = \mathbf{W}^{-1}$—and all the eigenvalues will be real.

Thus, in this special case, we can write (17.2.9) as

$$\mathbf{A} = \mathbf{W}\boldsymbol{\Sigma}\mathbf{W}^\top. \tag{17.2.14}$$

17.2.5 Gershgorin Circle Theorem

Eigenvalues are often difficult to reason with intuitively. If presented an arbitrary matrix, there is little that can be said about what the eigenvalues are without computing them. There is, however, one theorem that can make it easy to approximate well if the largest values are on the diagonal.

Let $\mathbf{A} = (a_{ij})$ be any square matrix ($n \times n$). We will define $r_i = \sum_{j \neq i} |a_{ij}|$. Let $\mathcal{D}_i$ represent the disc in the complex plane with center a_{ii} radius r_i. Then, every eigenvalue of $\mathbf{A}$ is contained in one of the $\mathcal{D}_i$.

This can be a bit to unpack, so let's look at an example.
Consider the matrix:

$$\mathbf{A} = \begin{bmatrix} 1.0 & 0.1 & 0.1 & 0.1 \\ 0.1 & 3.0 & 0.2 & 0.3 \\ 0.1 & 0.2 & 5.0 & 0.5 \\ 0.1 & 0.3 & 0.5 & 9.0 \end{bmatrix}. \tag{17.2.15}$$

We have $r_1 = 0.3$, $r_2 = 0.6$, $r_3 = 0.8$ and $r_4 = 0.9$. The matrix is symmetric, so all eigenvalues are real. This means that all of our eigenvalues will be in one of the ranges of

$$[a_{11} - r_1, a_{11} + r_1] = [0.7, 1.3], \tag{17.2.16}$$

$$[a_{22} - r_2, a_{22} + r_2] = [2.4, 3.6], \tag{17.2.17}$$

$$[a_{33} - r_3, a_{33} + r_3] = [4.2, 5.8], \tag{17.2.18}$$

$$[a_{44} - r_4, a_{44} + r_4] = [8.1, 9.9]. \tag{17.2.19}$$

Performing the numerical computation shows that the eigenvalues are approximately 0.99, 2.97, 4.95, 9.08, all comfortably inside the ranges provided.

```
A = np.array([[1.0, 0.1, 0.1, 0.1],
              [0.1, 3.0, 0.2, 0.3],
              [0.1, 0.2, 5.0, 0.5],
              [0.1, 0.3, 0.5, 9.0]])

v, _ = np.linalg.eig(A)
v
```

```
array([9.08033648, 0.99228545, 4.95394089, 2.97343718])
```

In this way, eigenvalues can be approximated, and the approximations will be fairly accurate in the case that the diagonal is significantly larger than all the other elements.

It is a small thing, but with a complex and subtle topic like eigendecomposition, it is good to get any intuitive grasp we can.

17.2.6 A Useful Application: The Growth of Iterated Maps

Now that we understand what eigenvectors are in principle, let's see how they can be used to provide a deep understanding of a problem central to neural network behavior: proper weight initialization.

Eigenvectors as Long Term Behavior

The full mathematical investigation of the initialization of deep neural networks is beyond the scope of the text, but we can see a toy version here to understand how eigenvalues can help us see how these models work. As we know, neural networks operate by interspersing layers of linear transformations with non-linear operations. For simplicity here, we will assume that there is no non-linearity, and that the transformation is a single repeated matrix operation A, so that the output of our model is

$$\mathbf{v}_{out} = \mathbf{A}\cdot\mathbf{A}\cdots\mathbf{A}\mathbf{v}_{in} = \mathbf{A}^N\mathbf{v}_{in}. \tag{17.2.20}$$

When these models are initialized, A is taken to be a random matrix with Gaussian entries, so let's make one of those. To be concrete, we start with a mean zero, variance one Gaussian distributed 5×5 matrix.

```
np.random.seed(8675309)

k = 5
A = np.random.randn(k, k)
A
```

```
array([[ 0.58902366,  0.73311856, -1.1621888 , -0.55681601, -0.77248843],
       [-0.16822143, -0.41650391, -1.37843129,  0.74925588,  0.17888446],
       [ 0.69401121, -1.9780535 , -0.83381434,  0.56437344,  0.31201299],
       [-0.87334496,  0.15601291, -0.38710108, -0.23920821,  0.88850104],
       [ 1.29385371, -0.76774106,  0.20131613,  0.91800842,  0.38974115]])
```

Behavior on Random Data

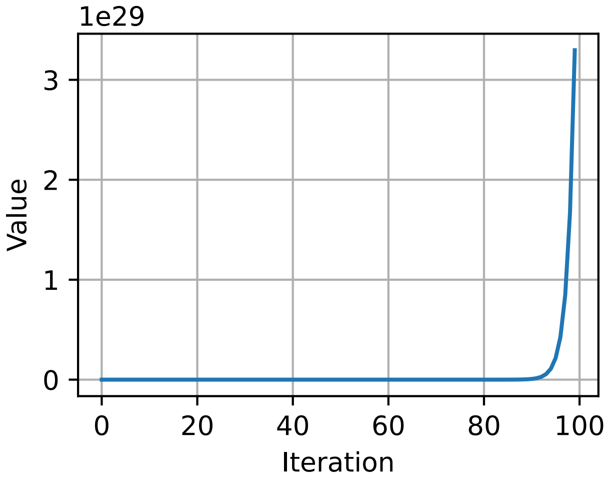
For simplicity in our toy model, we will assume that the data vector we feed in $\mathbf{v}_{in}$ is a random five dimensional Gaussian vector. Let's think about what we want to have happen. For context, lets think of a generic ML problem, where we are trying to turn input data, like an image, into a prediction, like the probability the image is a picture of a cat. If repeated application of $\mathbf{A}$ stretches a random vector out to be very long, then small changes in input will be amplified into large changes in output—tiny modifications of the input image would lead to vastly different predictions. This does not seem right!

On the flip side, if $\mathbf{A}$ shrinks random vectors to be shorter, then after running through many layers, the vector will essentially shrink to nothing, and the output will not depend on the input. This is also clearly not right either!

We need to walk the narrow line between growth and decay to make sure that our output changes depending on our input, but not much!

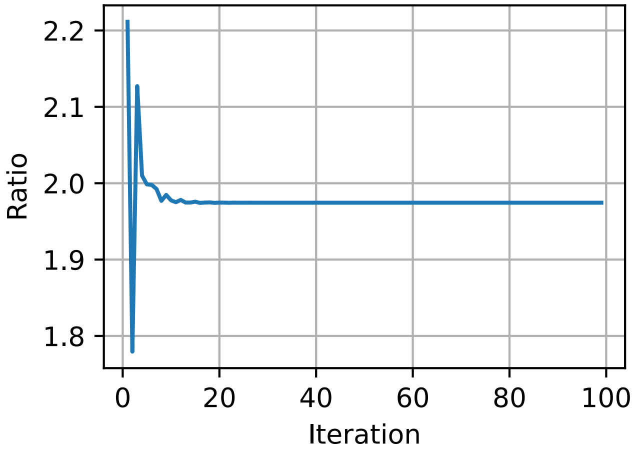
Let's see what happens when we repeatedly multiply our matrix $\mathbf{A}$ against a random input vector, and keep track of the norm.

```
# Calculate the sequence of norms after repeatedly applying A
v_in = np.random.randn(k, 1)

norm_list = [np.linalg.norm(v_in)]
for i in range(1, 100):
    v_in = A.dot(v_in)
    norm_list.append(np.linalg.norm(v_in))

d2l.plot(np.arange(0, 100), norm_list, 'Iteration', 'Value')
```



The norm is growing uncontrollably! Indeed if we take the list of quotients, we will see a pattern.

```
# Compute the scaling factor of the norms
norm_ratio_list = []
for i in range(1, 100):
    norm_ratio_list.append(norm_list[i]/norm_list[i - 1])

d2l.plot(np.arange(1, 100), norm_ratio_list, 'Iteration', 'Ratio')
```



If we look at the last portion of the above computation, we see that the random vector is stretched by a factor of `1.974459321485[...]`, where the portion at the end shifts a little, but the stretching factor is stable.

Relating Back to Eigenvectors

We have seen that eigenvectors and eigenvalues correspond to the amount something is stretched, but that was for specific vectors, and specific stretches. Let's take a look at what they are for $\mathbf{A}$. A bit of a caveat here: it turns out that to see them all, we will need to go to complex numbers. You can think of these as stretches and rotations. By taking the norm of the complex number (square root of the sums of squares of real and imaginary parts) we can measure that stretching factor. Let's also sort them.

```
# Compute the eigenvalues
eigs = np.linalg.eigvals(A).tolist()
norm_eigs = [np.absolute(x) for x in eigs]
norm_eigs.sort()
"Norms of eigenvalues: {}".format(norm_eigs)
```

```
'Norms of eigenvalues: [0.8786205280381857, 1.2757952665062624, 1.4983381517710659, 1.
↪4983381517710659, 1.974459321485074]'
```

An Observation

We see something a bit unexpected happening here: that number we identified before for the long term stretching of our matrix $\mathbf{A}$ applied to a random vector is *exactly* (accurate to thirteen decimal places!) the largest eigenvalue of $\mathbf{A}$. This is clearly not a coincidence!

But, if we now think about what is happening geometrically, this starts to make sense. Consider a random vector. This random vector points a little in every direction, so in particular, it points at least a little bit in the same direction as the eigenvector of $\mathbf{A}$ associated with the largest eigenvalue. This is so important that it is called the *principle eigenvalue* and *principle eigenvector*. After applying $\mathbf{A}$, our random vector gets stretched in every possible direction, as is associated with every possible eigenvector, but it is stretched most of all in the direction associated with this principle eigenvector. What this means is that after apply in A, our random vector is longer, and points in a direction closer to being aligned with the principle eigenvector. After applying the matrix many times, the alignment with the principle eigenvector becomes closer and closer until, for all practical purposes, our random vector has been transformed into the principle eigenvector! Indeed this algorithm is the basis for what is known as the *power iteration* for finding the largest eigenvalue and eigenvector of a matrix. For details see, for example, (VanLoan & Golub, 1983).

Fixing the Normalization

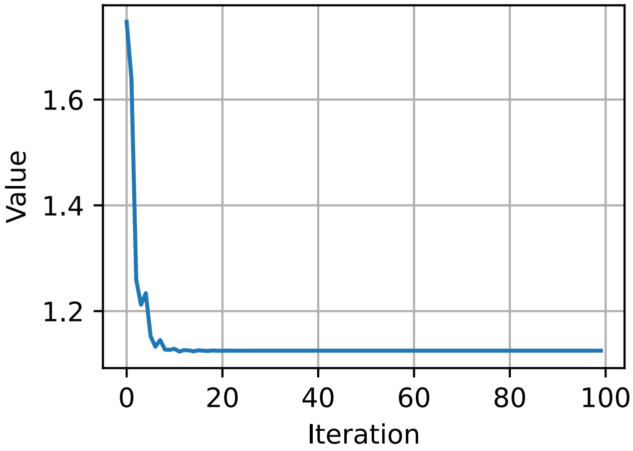
Now, from above discussions, we concluded that we do not want a random vector to be stretched or squished at all, we would like random vectors to stay about the same size throughout the entire process. To do so, we now rescale our matrix by this principle eigenvalue so that the largest eigenvalue is instead now just one. Let's see what happens in this case.

```
# Rescale the matrix A
A /= norm_eigs[-1]
```

(continues on next page)

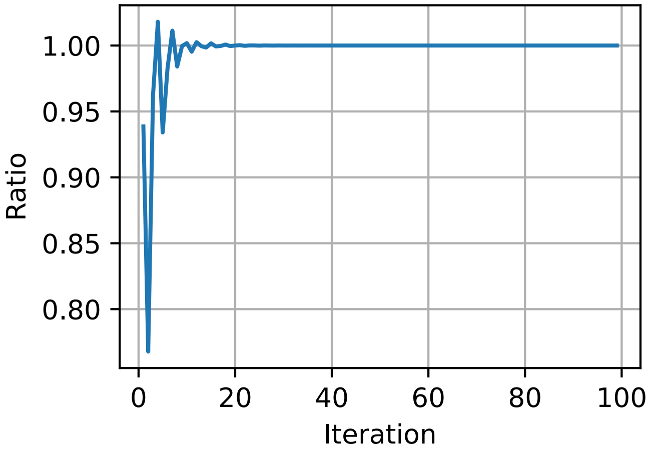
(continued from previous page)

```
# Do the same experiment again
v_in = np.random.randn(k, 1)

norm_list = [np.linalg.norm(v_in)]
for i in range(1, 100):
    v_in = A.dot(v_in)
    norm_list.append(np.linalg.norm(v_in))

d2l.plot(np.arange(0, 100), norm_list, 'Iteration', 'Value')
```



We can also plot the ration between consecutive norms as before and see that indeed it stabilizes.

```
# Also plot the ratio
norm_ratio_list = []
for i in range(1, 100):
    norm_ratio_list.append(norm_list[i]/norm_list[i-1])

d2l.plot(np.arange(1, 100), norm_ratio_list, 'Iteration', 'Ratio')
```



17.2.7 Conclusions

We now see exactly what we hoped for! After normalizing the matrices by the principle eigenvalue, we see that the random data does not explode as before, but rather eventually equilibrates to a specific value. It would be nice to be able to do these things from first principles, and it turns out that if we look deeply at the mathematics of it, we can see that the largest eigenvalue of a large random matrix with independent mean zero, variance one Gaussian entries is on average about $\sqrt{n}$, or in our case $\sqrt{5} \approx 2.2$, due to a fascinating fact known as the *circular law* (Ginibre, 1965). The relationship between the eigenvalues (and a related object called singular values) of random matrices has been shown to have deep connections to proper initialization of neural networks as was discussed in (Pennington et al., 2017) and subsequent works.

Summary

- Eigenvectors are vectors which are stretched by a matrix without changing direction.
- Eigenvalues are the amount that the eigenvectors are stretched by the application of the matrix.
- The eigendecomposition of a matrix can allow for many operations to be reduced to operations on the eigenvalues.
- The Gershgorin Circle Theorem can provide approximate values for the eigenvalues of a matrix.
- The behavior of iterated matrix powers depends primarily on the size of the largest eigenvalue. This understanding has many applications in the theory of neural network initialization.

Exercises

1. What are the eigenvalues and eigenvectors of

$$\mathbf{A} = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}? \tag{17.2.21}$$

2. What are the eigenvalues and eigenvectors of the following matrix, and what is strange about this example compared to the previous one?

$$\mathbf{A} = \begin{bmatrix} 2 & 1 \\ 0 & 2 \end{bmatrix}. \tag{17.2.22}$$

3. Without computing the eigenvalues, is it possible that the smallest eigenvalue of the following matrix is less that 0.5? *Note*: this problem can be done in your head.

$$\mathbf{A} = \begin{bmatrix} 3.0 & 0.1 & 0.3 & 1.0 \\ 0.1 & 1.0 & 0.1 & 0.2 \\ 0.3 & 0.1 & 5.0 & 0.0 \\ 1.0 & 0.2 & 0.0 & 1.8 \end{bmatrix}. \tag{17.2.23}$$

17.3 Single Variable Calculus

In Section 2.4, we saw the basic elements of differential calculus. This section takes a deeper dive into the fundamentals of calculus and how we can understand and apply it in the context of machine learning.

17.3.1 Differential Calculus

Differential calculus is fundamentally the study of how functions behave under small changes. To see why this is so core to deep learning, let's consider an example.

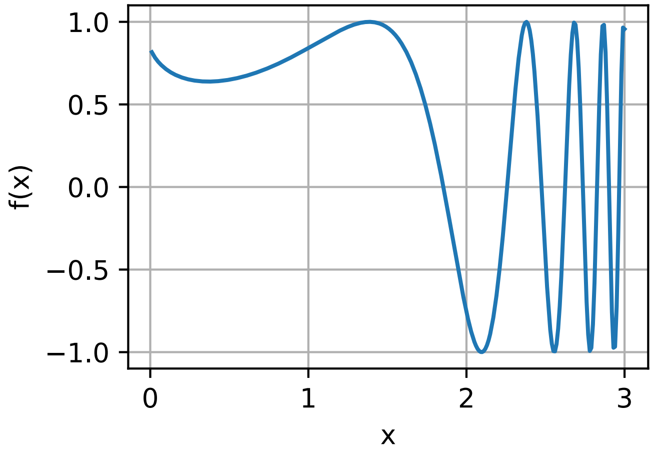
Suppose that we have a deep neural network where the weights are, for convenience, concatenated into a single vector $\mathbf{w} = (w_1, \ldots, w_n)$. Given a training dataset, we consider the loss of our neural network on this dataset, which we will write as $\mathcal{L}(\mathbf{w})$.

This function is extraordinarily complex, encoding the performance of all possible models of the given architecture on this dataset, so it is nearly impossible to tell what set of weights $\mathbf{w}$ will minimize the loss. Thus, in practice, we often start by initializing our weights *randomly*, and then iteratively take small steps in the direction which makes the loss decrease as rapidly as possible.

The question then becomes something that on the surface is no easier: how do we find the direction which makes the weights decrease as quickly as possible? To dig into this, let's first examine the case with only a single weight: $L(\mathbf{w}) = L(x)$ for a single real value x.

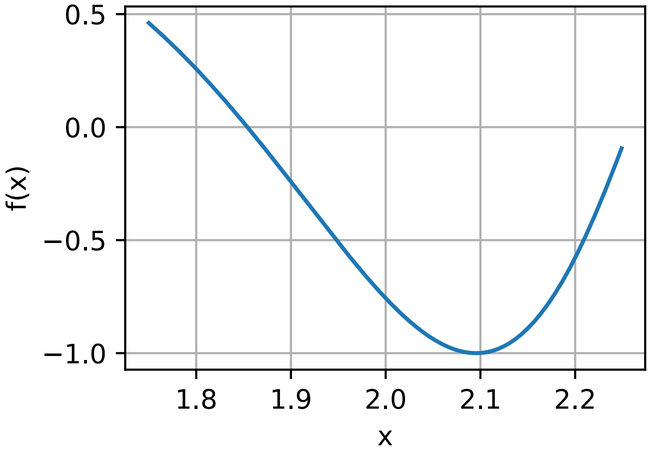
Let's take x and try to understand what happens when we change it by a small amount to $x + \epsilon$. If you wish to be concrete, think a number like $\epsilon = 0.0000001$. To help us visualize what happens, let's graph an example function, $f(x) = \sin(x^x)$, over the $[0, 3]$.

```
%matplotlib inline
import d2l
from IPython import display
from mxnet import np, npx
npx.set_np()

# Plot a function in a normal range
x_big = np.arange(0.01, 3.01, 0.01)
ys = np.sin(x_big**x_big)
d2l.plot(x_big, ys, 'x', 'f(x)')
```



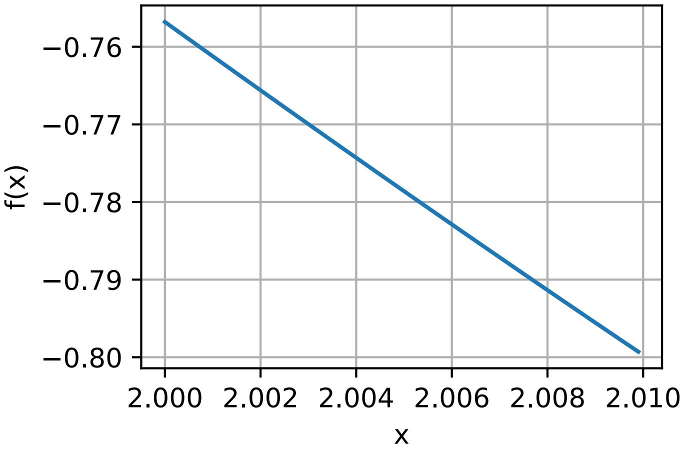
At this large scale, the function's behavior is not simple. However, if we reduce our range to something smaller like $[1.75, 2.25]$, we see that the graph becomes much simpler.

```
# Plot a the same function in a tiny range
x_med = np.arange(1.75, 2.25, 0.001)
ys = np.sin(x_med**x_med)
d2l.plot(x_med, ys, 'x', 'f(x)')
```



Taking this to an extreme, if we zoom into a tiny segment, the behavior becomes far simpler: it is just a straight line.

```
# Plot a the same function in a tiny range
x_small = np.arange(2.0, 2.01, 0.0001)
ys = np.sin(x_small**x_small)
d2l.plot(x_small, ys, 'x', 'f(x)')
```



This is the key observation of single variable calculus: the behavior of familiar functions can be modeled by a line in a small enough range. This means that for most functions, it is reasonable to expect that as we shift the x value of the function by a little bit, the output $f(x)$ will also be shifted by a little bit. The only question we need to answer is, "How large is the change in the output compared to the change in the input? Is it half as large? Twice as large?"

Thus, we can consider the ratio of the change in the output of a function for a small change in the input of the function. We can write this formally as

$$\frac{L(x+\epsilon) - L(x)}{(x+\epsilon) - x} = \frac{L(x+\epsilon) - L(x)}{\epsilon}. \tag{17.3.1}$$

This is already enough to start to play around with in code. For instance, suppose that we know that $L(x) = x^2 + 1701(x-4)^3$, then we can see how large this value is at the point $x = 4$ as follows.

```
# Define our function
def L(x):
    return x**2 + 1701*(x-4)**3

# Print the difference divided by epsilon for several epsilon
for epsilon in [0.1, 0.001, 0.0001, 0.00001]:
    print("epsilon = {:.5f} -> {:.5f}".format(
        epsilon, (L(4+epsilon) - L(4)) / epsilon))
```

```
epsilon = 0.10000 -> 25.11000
epsilon = 0.00100 -> 8.00270
epsilon = 0.00010 -> 8.00012
epsilon = 0.00001 -> 8.00001
```

Now, if we are observant, we will notice that the output of this number is suspiciously close to 8. Indeed, if we decrease ϵ, we will see value becomes progressively closer to 8. Thus we may conclude, correctly, that the value we seek (the degree a change in the input changes the output) should be 8 at the point $x = 4$. The way that a mathematician encodes this fact is

$$\lim_{\epsilon \rightarrow 0}\frac{L(4+\epsilon) - L(4)}{\epsilon} = 8. \tag{17.3.2}$$

As a bit of a historical digression: in the first few decades of neural network research, scientists used this algorithm (the *method of finite differences*) to evaluate how a loss function changed under small perturbation: just change the weights and see how the loss changed. This is computationally inefficient, requiring two evaluations of the loss function to see how a single change of one variable influenced the loss. If we tried to do this with even a paltry few thousand parameters, it would require several thousand evaluations of the network over the entire dataset! It was not solved until 1986 that the *backpropagation algorithm* introduced in (Rumelhart et al., 1988) provided a way to calculate how *any* change of the weights together would change the loss in the same computation time as a single prediction of the network over the dataset.

Back in our example, this value 8 is different for different values of x, so it makes sense to define it as a function of x. More formally, this value dependent rate of change is referred to as the *derivative* which is written as

$$\frac{df}{dx}(x) = \lim_{\epsilon \rightarrow 0}\frac{f(x+\epsilon) - f(x)}{\epsilon}. \tag{17.3.3}$$

Different texts will use different notations for the derivative. For instance, all of the below notations indicate the same thing:

$$\frac{df}{dx} = \frac{d}{dx}f = f' = \nabla_xf = D_xf = f_x. \tag{17.3.4}$$

Most authors will pick a single notation and stick with it, however even that is not guaranteed. It is best to be familiar with all of these. We will use the notation $\frac{df}{dx}$ throughout this text, unless we want to take the derivative of a complex expression, in which case we will use $\frac{d}{dx}f$ to write expressions like

$$\frac{d}{dx}\left[x^4 + \cos\left(\frac{x^2+1}{2x-1}\right)\right]. \tag{17.3.5}$$

Often times, it is intuitively useful to unravel the definition of derivative (17.3.3) again to see how a function changes when we make a small change of x:

$$\begin{aligned}\frac{df}{dx}(x) = \lim_{\epsilon \rightarrow 0}\frac{f(x+\epsilon) - f(x)}{\epsilon} & \implies \frac{df}{dx}(x) \approx \frac{f(x+\epsilon) - f(x)}{\epsilon} \\ & \implies \epsilon \frac{df}{dx}(x) \approx f(x+\epsilon) - f(x) \\ & \implies f(x+\epsilon) \approx f(x) + \epsilon \frac{df}{dx}(x).\end{aligned} \tag{17.3.6}$$

The last equation is worth explicitly calling out. It tells us that if you take any function and change the input by a small amount, the output would change by that small amount scaled by the derivative.

In this way, we can understand the derivative as the scaling factor that tells us how large of change we get in the output from a change in the input.

17.3.2 Rules of Calculus

We now turn to the task of understanding how to compute the derivative of an explicit function. A full formal treatment of calculus would derive everything from first principles. We will not indulge in this temptation here, but rather provide an understanding of the common rules encountered.

Common Derivatives

As was seen in Section 2.4, when computing derivatives one can often times use a series of rules to reduce the computation to a few core functions. We repeat them here for ease of reference.

- **Derivative of constants.** $\frac{d}{dx}c = 0$.
- **Derivative of linear functions.** $\frac{d}{dx}(ax) = a$.
- **Power rule.** $\frac{d}{dx}x^n = nx^{n-1}$.
- **Derivative of exponentials.** $\frac{d}{dx}e^x = e^x$.
- **Derivative of the logarithm.** $\frac{d}{dx}\log(x) = \frac{1}{x}$.

Derivative Rules

If every derivative needed to be separately computed and stored in a table, differential calculus would be near impossible. It is a gift of mathematics that we can generalize the above derivatives and compute more complex derivatives like finding the derivative of $f(x) = \log\left(1+(x-1)^{10}\right)$. As was mentioned in Section 2.4, the key to doing so is to codify what happens when we take functions and combine them in various ways, most importantly: sums, products, and compositions.

- **Sum rule.** $\frac{d}{dx}\left(g(x) + h(x)\right) = \frac{dg}{dx}(x) + \frac{dh}{dx}(x)$.

- **Product rule.** $\frac{d}{dx}\left(g(x) \cdot h(x)\right) = g(x)\frac{dh}{dx}(x) + \frac{dg}{dx}(x)h(x)$.
- **Chain rule.** $\frac{d}{dx}g(h(x)) = \frac{dg}{dh}(h(x)) \cdot \frac{dh}{dx}(x)$.

Let's see how we may use (17.3.6) to understand these rules. For the sum rule, consider following chain of reasoning:

$$\begin{aligned}
f(x+\epsilon) & = g(x+\epsilon) + h(x+\epsilon) \\
& \approx g(x) + \epsilon \frac{dg}{dx}(x) + h(x) + \epsilon \frac{dh}{dx}(x) \\
& = g(x) + h(x) + \epsilon\left(\frac{dg}{dx}(x) + \frac{dh}{dx}(x)\right) \\
& = f(x) + \epsilon\left(\frac{dg}{dx}(x) + \frac{dh}{dx}(x)\right).
\end{aligned} \tag{17.3.7}$$

By comparing this result with the fact that $f(x+\epsilon) \approx f(x) + \epsilon \frac{df}{dx}(x)$, we see that $\frac{df}{dx}(x) = \frac{dg}{dx}(x) + \frac{dh}{dx}(x)$ as desired. The intuition here is: when we change the input x, g and h jointly contribute to the change of the output by $\frac{dg}{dx}(x)$ and $\frac{dh}{dx}(x)$.

The product is more subtle, and will require a new observation about how to work with these expressions. We will begin as before using (17.3.6):

$$\begin{aligned}
f(x+\epsilon) & = g(x+\epsilon)\cdot h(x+\epsilon) \\
& \approx \left(g(x) + \epsilon \frac{dg}{dx}(x)\right) \cdot \left(h(x) + \epsilon \frac{dh}{dx}(x)\right) \\
& = g(x)\cdot h(x) + \epsilon\left(g(x)\frac{dh}{dx}(x) + \frac{dg}{dx}(x)h(x)\right) + \epsilon^2\frac{dg}{dx}(x)\frac{dh}{dx}(x) \\
& = f(x) + \epsilon\left(g(x)\frac{dh}{dx}(x) + \frac{dg}{dx}(x)h(x)\right) + \epsilon^2\frac{dg}{dx}(x)\frac{dh}{dx}(x).
\end{aligned} \tag{17.3.8}$$

This resembles the computation done above, and indeed we see our answer ($\frac{df}{dx}(x) = g(x)\frac{dh}{dx}(x) + \frac{dg}{dx}(x)h(x)$) sitting next to ϵ, but there is the issue of that term of size ϵ^{2}. We will refer to this as a *higher-order term*, since the power of ϵ^2 is higher than the power of ϵ^1. We will see in a later section that we will sometimes want to keep track of these, however for now observe that if $\epsilon = 0.0000001$, then $\epsilon^{2} = 0.0000000000001$, which is vastly smaller. As we send $\epsilon \rightarrow 0$, we may safely ignore the higher order terms. As a general convention in this appendix, we will use "≈" to denote that the two terms are equal up to higher order terms. However, if we wish to be more formal we may examine the difference quotient

$$\frac{f(x+\epsilon) - f(x)}{\epsilon} = g(x)\frac{dh}{dx}(x) + \frac{dg}{dx}(x)h(x) + \epsilon \frac{dg}{dx}(x)\frac{dh}{dx}(x), \tag{17.3.9}$$

and see that as we send $\epsilon \rightarrow 0$, the right hand term goes to zero as well.

Finally, with the chain rule, we can again progress as before using (17.3.6) and see that

$$\begin{aligned}
f(x+\epsilon) & = g(h(x+\epsilon)) \\
& \approx g\left(h(x) + \epsilon \frac{dh}{dx}(x)\right) \\
& \approx g(h(x)) + \epsilon \frac{dh}{dx}(x) \frac{dg}{dh}(h(x))\\
& = f(x) + \epsilon \frac{dg}{dh}(h(x))\frac{dh}{dx}(x),
\end{aligned} \tag{17.3.10}$$

where in the second line we view the function g as having its input ($h(x)$) shifted by the tiny quantity $\epsilon \frac{dh}{dx}(x)$.

These rule provide us with a flexible set of tools to compute essentially any expression desired. For instance,

$$\begin{aligned}
\frac{d}{dx}\left[\log\left(1+(x-1)^{10}\right)\right] & = \left(1+(x-1)^{10}\right)^{-1}\frac{d}{dx}\left[1+(x-1)^{10}\right]\\
& = \left(1+(x-1)^{10}\right)^{-1}\left(\frac{d}{dx}[1] + \frac{d}{dx}[(x-1)^{10}]\right)\\
& = \left(1+(x-1)^{10}\right)^{-1}\left(0 + 10(x-1)^9\frac{d}{dx}[x-1]\right)\\
& = 10\left(1+(x-1)^{10}\right)^{-1}(x-1)^9\\
& = \frac{10(x-1)^9}{1+(x-1)^{10}}.
\end{aligned} \tag{17.3.11}$$

Where each line has used the following rules:

1. The chain rule and derivative of logarithm.
2. The sum rule.
3. The derivative of constants, chain rule, and power rule.
4. The sum rule, derivative of linear functions, derivative of constants.

Two things should be clear after doing this example:

1. Any function we can write down using sums, products, constants, powers, exponentials, and logarithms can have its derivate computed mechanically by following these rules.
2. Having a human follow these rules can be tedious and error prone!

Thankfully, these two facts together hint towards a way forward: this is a perfect candidate for mechanization! Indeed backpropagation, which we will revisit later in this section, is exactly that.

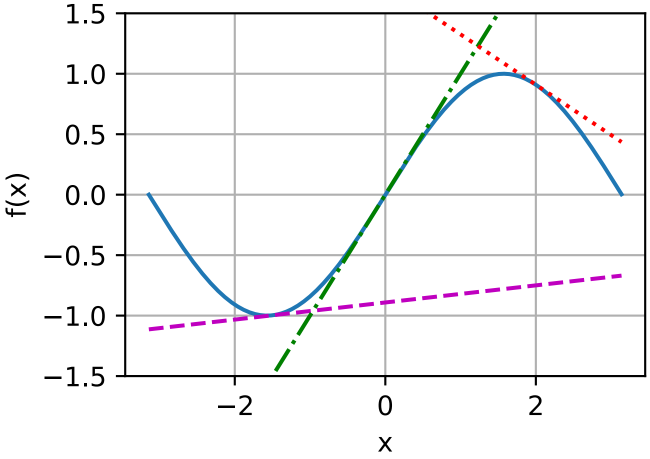
Linear Approximation

When working with derivatives, it is often useful to geometrically interpret the approximation used above. In particular, note that the equation

$$f(x+\epsilon) \approx f(x) + \epsilon \frac{df}{dx}(x), \tag{17.3.12}$$

approximates the value of f by a line which passes through the point $(x, f(x))$ and has slope $\frac{df}{dx}(x)$. In this way we say that the derivative gives a linear approximation to the function f, as illustrated below:

```python
# Compute sin
xs = np.arange(-np.pi, np.pi, 0.01)
plots = [np.sin(xs)]

# Compute some linear approximations. Use d(sin(x))/dx = cos(x)
for x0 in [-1.5, 0, 2]:
    plots.append(np.sin(x0) + (xs - x0) * np.cos(x0))

d2l.plot(xs, plots, 'x', 'f(x)', ylim=[-1.5, 1.5])
```



Higher Order Derivatives

Let's now do something that may on the surface seem strange. Take a function f and compute the derivative $\frac{df}{dx}$. This gives us the rate of change of f at any point.

However, the derivative, $\frac{df}{dx}$, can be viewed as a function itself, so nothing stops us from computing the derivative of $\frac{df}{dx}$ to get $\frac{d^2f}{dx^2} = \frac{df}{dx}\left(\frac{df}{dx}\right)$. We will call this the second derivative of f. This function is the rate of change of the rate of change of f, or in other words, how the rate of change is changing. We may apply the derivative any number of times to obtain what is called the n-th derivative. To keep the notation clean, we will denote the n-th derivative as

$$f^{(n)}(x) = \frac{d^n f}{dx^n} = \left(\frac{d}{dx}\right)^n f. \tag{17.3.13}$$

Let's try to understand *why* this is a useful notion. Below, we visualize $f^{(2)}(x)$, $f^{(1)}(x)$, and $f(x)$.

First, consider the case that the second derivative $f^{(2)}(x)$ is a positive constant. This means that the slope of the first derivative is positive. As a result, the first derivative $f^{(1)}(x)$ may start out negative, becomes zero at a point, and then becomes positive in the end. This tells us the slope of our original function f and therefore, the function f itself decreases, flattens out, then increases. In other words, the function f curves up, and has a single minimum as is shown in Fig. 17.3.1.

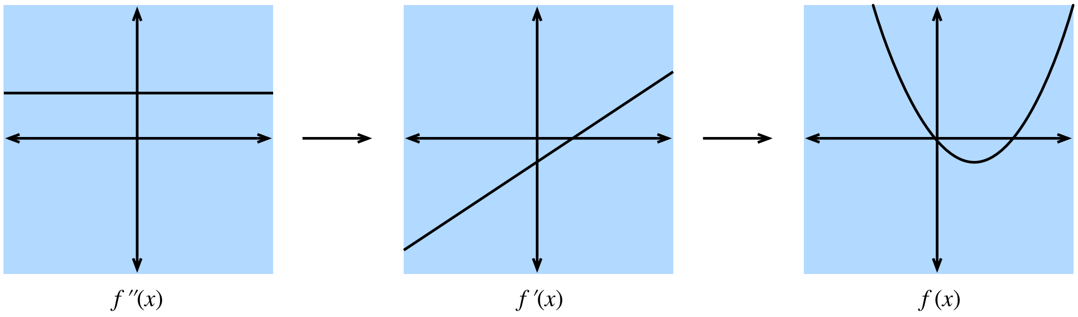


Fig. 17.3.1: If we assume the second derivative is a positive constant, then the fist derivative in increasing, which implies the function itself has a minimum.

Second, if the second derivative is a negative constant, that means that the first derivative is decreasing. This implies the first derivative may start out positive, becomes zero at a point, and then becomes negative. Hence, the function f itself increases, flattens out, then decreases. In other words, the function f curves down, and has a single maximum as is shown in Fig. 17.3.2.

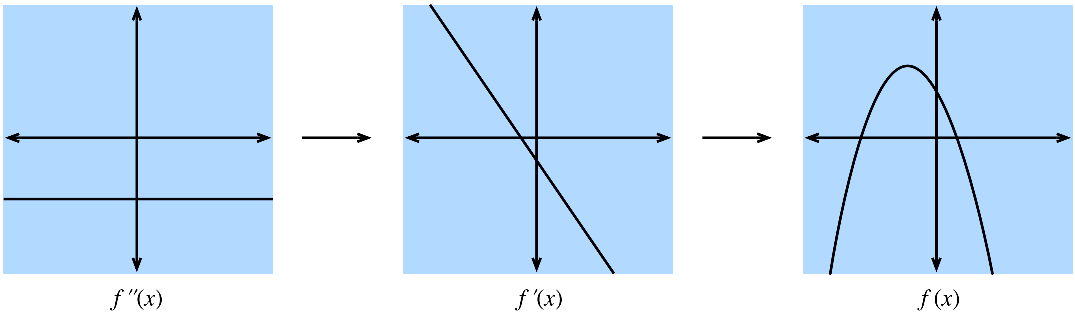


Fig. 17.3.2: If we assume the second derivative is a negative constant, then the fist derivative in decreasing, which implies the function itself has a maximum.

Third, if the second derivative is a always zero, then the first derivative will never change—it is constant! This means that f increases (or decreases) at a fixed rate, and f is itself a straight line as is shown in Fig. 17.3.3.

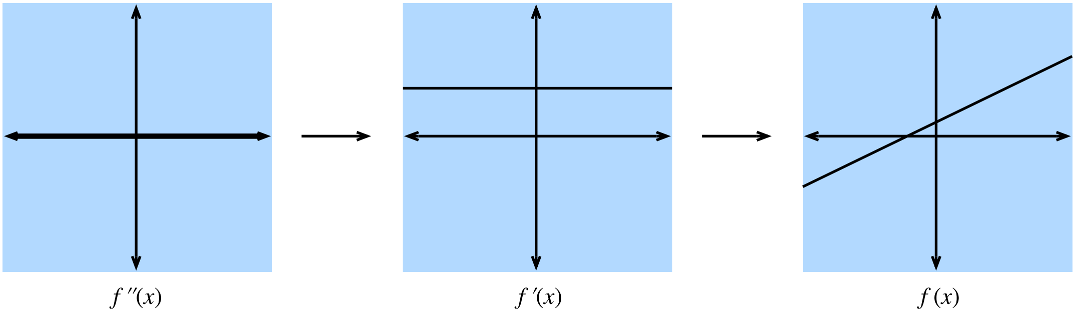


Fig. 17.3.3: If we assume the second derivative is zero, then the fist derivative is constant, which implies the function itself is a straight line.

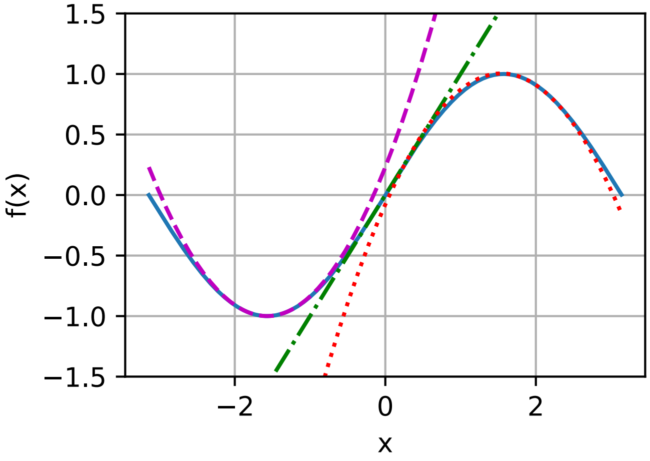
To summarize, the second derivative can be interpreted as describing the way that the function f curves. A positive second derivative leads to a upwards curve, while a negative second derivative means that f curves downwards, and a zero second derivative means that f does not curve at all.

Let's take this one step further. Consider the function $g(x) = ax^2 + bx + c$. We can then compute that

$$\begin{aligned}\frac{dg}{dx}(x) & = 2ax + b \\ \frac{d^2g}{dx^2}(x) & = 2a.\end{aligned} \tag{17.3.14}$$

If we have some original function $f(x)$ in mind, we may compute the first two derivatives and find the values for a, b, and c that make them match this computation. Similarly to the previous section where we saw that the first derivative gave the best approximation with a straight line, this construction provides the best approximation by a quadratic. Let's visualize this for $f(x) = \sin(x)$.

```
# Compute sin
xs = np.arange(-np.pi, np.pi, 0.01)
plots = [np.sin(xs)]

# Compute some quadratic approximations. Use d(sin(x))/dx = cos(x)
for x0 in [-1.5, 0, 2]:
    plots.append(np.sin(x0) + (xs - x0) * np.cos(x0) -
                              (xs - x0)**2 * np.sin(x0) / 2)

d2l.plot(xs, plots, 'x', 'f(x)', ylim=[-1.5, 1.5])
```



We will extend this idea to the idea of a *Taylor series* in the next section.

Taylor Series

The *Taylor series* provides a method to approximate the function $f(x)$ if we are given values for the first n derivatives at a point x_0, i.e., $\left\{ f(x_0), f^{(1)}(x_0), f^{(2)}(x_0), \ldots, f^{(n)}(x_0) \right\}$. The idea will be to find a degree n polynomial that matches all the given derivatives at x_0.

We saw the case of $n = 2$ in the previous section and a little algebra shows this is

$$f(x) \approx \frac{1}{2}\frac{d^2f}{dx^2}(x_0)(x-x_0)^{2}+ \frac{df}{dx}(x_0)(x-x_0) + f(x_0). \tag{17.3.15}$$

As we can see above, the denominator of 2 is there to cancel out the 2 we get when we take two derivatives of x^2, while the other terms are all zero. Same logic applies for the first derivative and the value itself.

If we push the logic further to $n = 3$, we will conclude that

$$f(x) \approx \frac{\frac{d^3f}{dx^3}(x_0)}{6}(x-x_0)^3 + \frac{\frac{d^2f}{dx^2}(x_0)}{2}(x-x_0)^{2}+ \frac{df}{dx}(x_0)(x-x_0) + f(x_0). \tag{17.3.16}$$

where the $6 = 3 \times 2 = 3!$ comes from the constant we get in front if we take three derivatives of x^3.

Furthermore, we can get a degree n polynomial by

$$P_n(x) = \sum_{i = 0}^{n} \frac{f^{(i)}(x_0)}{i!}(x-x_0)^{i}. \tag{17.3.17}$$

where the notation

$$f^{(n)}(x) = \frac{d^n f}{dx^n} = \left(\frac{d}{dx}\right)^n f. \tag{17.3.18}$$

Indeed, $P_n(x)$ can be viewed as the best n-th degree polynomial approximation to our function $f(x)$.

While we are not going to dive all the way into the error of the above approximations, it is worth mentioning the the infinite limit. In this case, for well behaved functions (known as real analytic functions) like $\cos(x)$ or e^x, we can write out the infinite number of terms and approximate the exactly same function

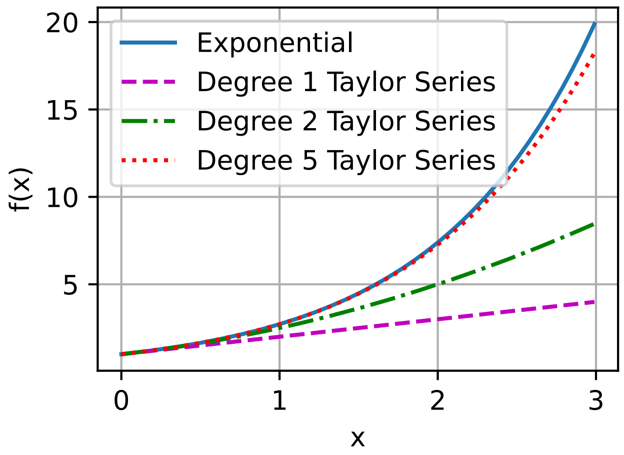
$$f(x) = \sum_{n=0}^{\infty} \frac{f^{(n)}(x_0)}{n!}(x - x_0)^n. \tag{17.3.19}$$

Take $f(x) = e^x$ as am example. Since e^x is its own derivative, we know that $f^{(n)}(x) = e^x$. Therefore, e^x can be reconstructed by taking the Taylor series at $x_0 = 0$, i.e.,

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!} = 1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \cdots . \tag{17.3.20}$$

Let's see how this works in code and observe how increasing the degree of the Taylor approximation brings us closer to the desired function e^x.

```
# Compute the exponential function
xs = np.arange(0, 3, 0.01)
ys = np.exp(xs)

# Compute a few Taylor series approximations
P1 = 1 + xs
P2 = 1 + xs + xs**2 / 2
P5 = 1 + xs + xs**2 / 2 + xs**3 / 6 + xs**4 / 24 + xs**5 / 120

d2l.plot(xs, [ys, P1, P2, P5], 'x', 'f(x)', legend=[
    "Exponential", "Degree 1 Taylor Series", "Degree 2 Taylor Series",
    "Degree 5 Taylor Series"])
```



Taylor series have two primary applications:

1. *Theoretical applications*: Often when we try to understand a too complex function, using Taylor series enables we turn it into a polynomial that we can work with directly.
2. *Numerical applications*: Some functions like e^x or $\cos(x)$ are difficult for machines to compute. They can store tables of values at a fixed precision (and this is often done), but it still leaves open questions like "What is the 1000-th digit of $\cos(1)$?" Taylor series are often helpful to answer such questions.

Summary

- Derivatives can be used to express how functions change when we change the input by a small amount.
- Elementary derivatives can be combined using derivative rules to create arbitrarily complex derivatives.
- Derivatives can be iterated to get second or higher order derivatives. Each increase in order provides more fine grained information on the behavior of the function.
- Using information in the derivatives of a single data point, we can approximate well behaved functions by polynomials obtained from the Taylor series.

Exercises

1. What is the derivative of $x^3 - 4x + 1$?
2. What is the derivative of $\log(\frac{1}{x})$?
3. True or False: If $f'(x) = 0$ then f has a maximum or minimum at x?
4. Where is the minimum of $f(x) = x\log(x)$ for $x \geq 0$ (where we assume that f takes the limiting value of 0 at $f(0)$)?

17.4 Multivariable Calculus

Now that we have a fairly strong understanding of derivatives of a function of a single variable, let's return to our original question where we were considering a loss function of potentially billions of weights.

17.4.1 Higher-Dimensional Differentiation

What Section 17.3 tells us is that if we change a single one of these billions of weights leaving every other one fixed, we know what will happen! This is nothing more than a function of a single variable, so we can write

$$L(w_1+\epsilon_1, w_2, \ldots, w_N) \approx L(w_1, w_2, \ldots, w_N) + \epsilon_1 \frac{d}{dw_1} L(w_1, w_2, \ldots, w_N). \tag{17.4.1}$$

We will call the derivative in one variable while fixing the other the *partial derivative*, and we will use the notation $\frac{\partial}{\partial w_1}$ for the derivative in (17.4.1).

Now, let's take this and change w_2 a little bit to $w_2 + \epsilon_2$:

$$\begin{aligned} L(w_1+\epsilon_1, w_2+\epsilon_2, \ldots, w_N) & \approx L(w_1, w_2+\epsilon_2, \ldots, w_N) + \epsilon_1 \frac{\partial}{\partial w_1} L(w_1, w_2+\epsilon_2, \ldots, w_N) \\ & \approx L(w_1, w_2, \ldots, w_N) \\ & \quad + \epsilon_2\frac{\partial}{\partial w_2} L(w_1, w_2, \ldots, w_N) \\ & \quad + \epsilon_1\frac{\partial}{\partial w_1} L(w_1, w_2, \ldots, w_N) \\ & \quad + \epsilon_1\epsilon_2\frac{\partial}{\partial w_2}\frac{\partial}{\partial w_1} L(w_1, w_2, \ldots, w_N) \\ & \approx L(w_1, w_2, \ldots, w_N) \\ & \quad + \epsilon_2\frac{\partial}{\partial w_2} L(w_1, w_2, \ldots, w_N) \\ & \quad + \epsilon_1\frac{\partial}{\partial w_1} L(w_1, w_2, \ldots, w_N). \end{aligned} \tag{17.4.2}$$

We have again used the idea that $\epsilon_1\epsilon_2$ is a higher order term that we can discard in the same way we could discard ϵ^{2} in the previous section, along with what we saw in (17.4.1). By continuing in this manner, we may write that

$$L(w_1+\epsilon_1, w_2+\epsilon_2, \ldots, w_N+\epsilon_N) \approx L(w_1, w_2, \ldots, w_N) + \sum_i \epsilon_i \frac{\partial}{\partial w_i} L(w_1, w_2, \ldots, w_N). \tag{17.4.3}$$

This may look like a mess, but we can make this more familiar by noting that the sum on the right looks exactly like a dot product, so if we let

$$\boldsymbol{\epsilon} = [\epsilon_1, \ldots, \epsilon_N]^\top \text{ and } \nabla_{\mathbf{x}} L = \left[\frac{\partial L}{\partial x_1}, \ldots, \frac{\partial L}{\partial x_N}\right]^\top, \tag{17.4.4}$$

then

$$L(\mathbf{w} + \boldsymbol{\epsilon}) \approx L(\mathbf{w}) + \boldsymbol{\epsilon}\cdot \nabla_{\mathbf{w}} L(\mathbf{w}). \tag{17.4.5}$$

We will call the vector $\nabla_{\mathbf{w}} L$ the *gradient* of L.

Equation (17.4.5) is worth pondering for a moment. It has exactly the format that we encountered in one dimension, just we have converted everything to vectors and dot products. It allows us to tell approximately how the function L will change given any perturbation to the input. As we will see in the next section, this will provide us with an important tool in understanding geometrically how we can learn using information contained in the gradient.

But first, let's see this approximation at work with an example. Suppose that we are working with the function

$$f(x, y) = \log(e^x + e^y) \text{ with gradient } \nabla f(x, y) = \left[\frac{e^x}{e^x + e^y}, \frac{e^y}{e^x + e^y}\right]. \tag{17.4.6}$$

If we look at a point like $(0, \log(2))$, we see that

$$f(x, y) = \log(3) \text{ with gradient } \nabla f(x, y) = \left[\frac{1}{3}, \frac{2}{3}\right]. \tag{17.4.7}$$

Thus, if we want to approximate f at $(\epsilon_1, \log(2) + \epsilon_2)$, we see that we should have the specific instance of (17.4.5):

$$f(\epsilon_1, \log(2) + \epsilon_2) \approx \log(3) + \frac{1}{3}\epsilon_1 + \frac{2}{3}\epsilon_2. \tag{17.4.8}$$

We can test this in code to see how good the approximation is.

```python
%matplotlib inline
import d2l
from IPython import display
from mpl_toolkits import mplot3d
from mxnet import autograd, np, npx
npx.set_np()

def f(x, y):
    return np.log(np.exp(x) + np.exp(y))
def grad_f(x, y):
    return np.array([np.exp(x) / (np.exp(x) + np.exp(y)),
                     np.exp(y) / (np.exp(x) + np.exp(y))])

epsilon = np.array([0.01, -0.03])
grad_approx = f(0, np.log(2)) + epsilon.dot(grad_f(0, np.log(2)))
true_value = f(0 + epsilon[0], np.log(2) + epsilon[1])
"Approximation: {}, True Value: {}".format(grad_approx, true_value)
```

```
'Approximation: 1.0819457, True Value: 1.0821242'
```

17.4.2 Geometry of Gradients and Gradient Descent

Consider the again (17.4.5):

$$L(\mathbf{w} + \boldsymbol{\epsilon}) \approx L(\mathbf{w}) + \boldsymbol{\epsilon} \cdot \nabla_{\mathbf{w}} L(\mathbf{w}). \tag{17.4.9}$$

Let's suppose that I want to use this to help minimize our loss L. Let's understand geometrically the algorithm of gradient descent first described in Section 2.5. What we will do is the following:

1. Start with a random choice for the initial parameters $\mathbf{w}$.
2. Find the direction $\mathbf{v}$ that makes L decrease the most rapidly at $\mathbf{w}$.
3. Take a small step in that direction: $\mathbf{w} \rightarrow \mathbf{w} + \epsilon\mathbf{v}$.
4. Repeat.

The only thing we do not know exactly how to do is to compute the vector $\mathbf{v}$ in the second step. We will call such a direction the *direction of steepest descent*. Using the geometric understanding of dot products from Section 17.1, we see that we can rewrite (17.4.5) as

$$L(\mathbf{w} + \mathbf{v}) \approx L(\mathbf{w}) + \mathbf{v}\cdot \nabla_{\mathbf{w}} L(\mathbf{w}) = \|\nabla_{\mathbf{w}} L(\mathbf{w})\| \cos(\theta). \tag{17.4.10}$$

Note that we have taken our direction to have length one for convenience, and used θ for the angle between $\mathbf{v}$ and $\nabla_{\mathbf{w}} L(\mathbf{w})$. If we want to find the direction that decreases L as rapidly as possible, we want to make this as expression as negative as possible. The only way the direction we pick enters into this equation is through $\cos(\theta)$, and thus we wish to make this cosine as negative as possible. Now, recalling the shape of cosine, we can make this as negative as possible by making $\cos(\theta) = -1$ or equivalently making the angle between the gradient and our chosen direction to be π radians, or equivalently 180 degrees. The only way to achieve this is to head in the exact opposite direction: pick $\mathbf{v}$ to point in the exact opposite direction to $\nabla_{\mathbf{w}} L(\mathbf{w})$!

This brings us to one of the most important mathematical concepts in machine learning: the direction of steepest decent points in the direction of $-\nabla_{\mathbf{w}}L(\mathbf{w})$. Thus our informal algorithm can be rewritten as follows.

1. Start with a random choice for the initial parameters $\mathbf{w}$.
2. Compute $\nabla_{\mathbf{w}} L(\mathbf{w})$.
3. Take a small step in the opposite of that direction: $\mathbf{w} \rightarrow \mathbf{w} - \epsilon\nabla_{\mathbf{w}} L(\mathbf{w})$.
4. Repeat.

This basic algorithm has been modified and adapted many ways by many researchers, but the core concept remains the same in all of them. Use the gradient to find the direction that decreases the loss as rapidly as possible, and update the parameters to take a step in that direction.

17.4.3 A Note on Mathematical Optimization

Throughout this book, we focus squarely on numerical optimization techniques for the practical reason that all functions we encounter in the deep learning setting are too complex to minimize explicitly.

However, it is a useful exercise to consider what the geometric understanding we obtained above tells us about optimizing functions directly.

Suppose that we wish to find the value of $\mathbf{x}_0$ which minimizes some function $L(\mathbf{x})$. Let's suppose that moreover someone gives us a value and tells us that it is the value that minimizes L. Is there anything we can check to see if their answer is even plausible?

Again consider (17.4.5):

$$L(\mathbf{x}_0 + \boldsymbol{\epsilon}) \approx L(\mathbf{x}_0) + \boldsymbol{\epsilon}\cdot \nabla_{\mathbf{x}} L(\mathbf{x}_0). \tag{17.4.11}$$

If the gradient is not zero, we know that we can take a step in the direction $-\epsilon \nabla_{\mathbf{x}} L(\mathbf{x}_0)$ to find a value of L that is smaller. Thus, if we truly are at a minimum, this cannot be the case! We can conclude that if $\mathbf{x}_0$ is a minimum, then $\nabla_{\mathbf{x}} L(\mathbf{x}_0 = 0$. We call points with $\nabla_{\mathbf{x}} L(\mathbf{x}_0 = 0$ *critical points*.

This is nice, because in some rare settings, we *can* explicitly find all the points where the gradient is zero, and find the one with the smallest value.

For a concrete example, consider the function

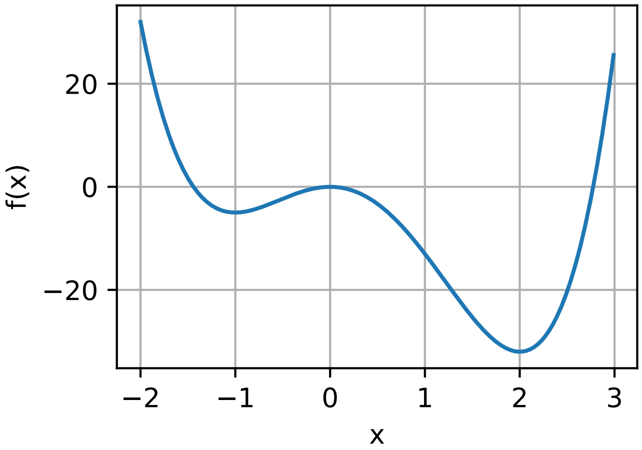
$$f(x) = 3x^4 - 4x^3 - 12x^2. \tag{17.4.12}$$

This function has derivative

$$\frac{df}{dx} = 12x^3 - 12x^2 - 24x = 12x(x-2)(x+1). \tag{17.4.13}$$

The only possible location of minima are at $x = -1, 0, 2$, where the function takes the values $-5, 0, -32$ respectively, and thus we can conclude that we minimize our function when $x = 2$. A quick plot confirms this.

```
x = np.arange(-2, 3, 0.01)
f = (3 * x**4) - (4 * x**3) - (12 * x**2)

d2l.plot(x, f, 'x', 'f(x)')
```



This highlights an important fact to know when working either theoretically or numerically: the only possible points where we can minimize (or maximize) a function will have gradient equal to zero, however, not every point with gradient zero is the true *global* minimum (or maximum).

17.4.4 Multivariate Chain Rule

Let's suppose that we have a function of four variables ($w, x, y,$ and z) which we can make by composing many terms:

$$\begin{aligned} f(u,v) &= (u+v)^2 \\ u(a,b) &= (a+b)^2, \qquad v(a,b) = (a-b)^2, \\ a(w,x,y,z) &= (w+x+y+z)^2, \qquad b(w,x,y,z) = (w+x-y-z)^2. \end{aligned} \tag{17.4.14}$$

Such chains of equations are common when working with neural networks, so trying to understand how to compute gradients of such functions is key. We can start to see visual hints of this connection in Fig. 17.4.1 if we take a look at what variables directly relate to one another.

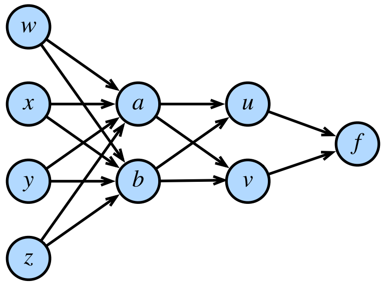


Fig. 17.4.1: The function relations above where nodes represent values and edges show functional dependence.

Nothing stops us from just composing everything from (17.4.14) and writing out that

$$f(w, x, y, z) = \left(\left((w + x + y + z)^2 + (w + x - y - z)^2\right)^2 + \left((w + x + y + z)^2 - (w + x - y - z)^2\right)^2\right)^2. \tag{17.4.15}$$

We may then take the derivative by just using single variable derivatives, but if we did that we would quickly find ourself swamped with terms, many of which are repeats! Indeed, one can see that, for instance:

$$\begin{aligned}\frac{\partial f}{\partial w} = {} & 2 \left(2 \left(2(w + x + y + z) - 2(w + x - y - z)\right) \left((w + x + y + z)^2 - (w + x - y - z)^2\right) + \right. \\ & \left. 2 \left(2(w + x - y - z) + 2(w + x + y + z)\right) \left((w + x - y - z)^2 + (w + x + y + z)^2\right)\right) \times \\ & \left(\left((w + x + y + z)^2 - (w + x - y - z)^2\right)^2 + \left((w + x - y - z)^2 + (w + x + y + z)^2\right)^2\right).\end{aligned} \tag{17.4.16}$$

If we then also wanted to compute $\frac{\partial f}{\partial x}$, we would end up with a similar equation again with many repeated terms, and many *shared* repeated terms between the two derivatives. This represents a massive quantity of wasted work, and if we needed to compute derivatives this way, the whole deep learning revolution would have stalled out before it began!

Let's break up the problem. We will start by trying to understand how f changes when we change a, essentially assuming that $w, x, y,$ and z all do not exist. We will reason as we did back when we worked with the gradient for the first time. Let's take a and add a small amount ϵ to it.

$$\begin{aligned} & f(u(a+\epsilon, b), v(a+\epsilon, b)) \\ \approx & f\left(u(a, b) + \epsilon\frac{\partial u}{\partial a}(a, b), v(a, b) + \epsilon\frac{\partial v}{\partial a}(a, b)\right) \\ \approx & f(u(a, b), v(a, b)) + \epsilon\left[\frac{\partial f}{\partial u}(u(a, b), v(a, b))\frac{\partial u}{\partial a}(a, b) + \frac{\partial f}{\partial v}(u(a, b), v(a, b))\frac{\partial v}{\partial a}(a, b)\right]. \end{aligned} \tag{17.4.17}$$

The first line follows from the definition of partial derivative, and the second follows from the definition of gradient. It is notationally burdensome to track exactly where we evaluate every derivative, as in the expression $\frac{\partial f}{\partial u}(u(a, b), v(a, b))$, so we often abbreviate this to the much more memorable

$$\frac{\partial f}{\partial a} = \frac{\partial f}{\partial u}\frac{\partial u}{\partial a}+\frac{\partial f}{\partial v}\frac{\partial v}{\partial a}. \tag{17.4.18}$$

It is useful to think about the meaning of the process. We are trying to understand how a function of the form $f(u(a, b), v(a, b))$ changes its value with a change in a. There are two pathways this can occur: there is the pathway where $a \to u \to f$ and where $a \to v \to f$. We can compute both of these contributions via the chain rule: $\frac{\partial w}{\partial u} \cdot \frac{\partial u}{\partial x}$ and $\frac{\partial w}{\partial v} \cdot \frac{\partial v}{\partial x}$ respectively, and added up.

Imagine we have a different network of functions where the functions on the right depend on those they are connected to on the left as is shown in Fig. 17.4.2.

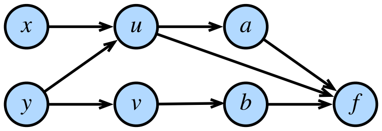


Fig. 17.4.2: Another more subtle example of the chain rule.

To compute something like $\frac{\partial f}{\partial y}$, we need to sum over all (in this case 3) paths from y to f giving

$$\frac{\partial f}{\partial y} = \frac{\partial f}{\partial a}\frac{\partial a}{\partial u}\frac{\partial u}{\partial y} + \frac{\partial f}{\partial u}\frac{\partial u}{\partial y} + \frac{\partial f}{\partial b}\frac{\partial b}{\partial v}\frac{\partial v}{\partial y}. \tag{17.4.19}$$

Understanding the chain rule in this way will pay great dividends when trying to understand how gradients flow through networks, and why various architectural choices like those in LSTMs (Section 9.2) or residual layers (Section 7.6) can help shape the learning process by controlling gradient flow.

17.4.5 The Backpropagation Algorithm

Let's return to the example of (17.4.14) the previous section where

$$\begin{aligned} f(u, v) &= (u+v)^2 \\ u(a, b) &= (a+b)^2, \qquad v(a, b) = (a-b)^2, \\ a(w, x, y, z) &= (w+x+y+z)^2, \qquad b(w, x, y, z) = (w+x-y-z)^2. \end{aligned} \tag{17.4.20}$$

If we want to compute say $\frac{\partial f}{\partial w}$ we may apply the multi-variate chain rule to see:

$$\begin{aligned} \frac{\partial f}{\partial w} &= \frac{\partial f}{\partial u}\frac{\partial u}{\partial w} + \frac{\partial f}{\partial v}\frac{\partial v}{\partial w}, \\ \frac{\partial u}{\partial w} &= \frac{\partial u}{\partial a}\frac{\partial a}{\partial w} + \frac{\partial u}{\partial b}\frac{\partial b}{\partial w}, \\ \frac{\partial v}{\partial w} &= \frac{\partial v}{\partial a}\frac{\partial a}{\partial w} + \frac{\partial v}{\partial b}\frac{\partial b}{\partial w}. \end{aligned} \tag{17.4.21}$$

Let's try using this decomposition to compute $\frac{\partial f}{\partial w}$. Notice that all we need here are the various single step partials:

$$\begin{aligned} \frac{\partial f}{\partial u} &= 2(u+v), & \frac{\partial f}{\partial v} &= 2(u+v), \\ \frac{\partial u}{\partial a} &= 2(a+b), & \frac{\partial u}{\partial b} &= 2(a+b), \\ \frac{\partial v}{\partial a} &= 2(a-b), & \frac{\partial v}{\partial b} &= -2(a-b), \\ \frac{\partial a}{\partial w} &= 2(w+x+y+z), & \frac{\partial b}{\partial w} &= 2(w+x-y-z). \end{aligned} \tag{17.4.22}$$

If we write this out into code this becomes a fairly manageable expression.

```
# Compute the value of the function from inputs to outputs
w, x, y, z = -1, 0, -2, 1
a, b = (w + x + y + z)**2, (w + x - y - z)**2
u, v = (a + b)**2, (a - b)**2
f = (u + v)**2
print("    f at {}, {}, {}, {} is {}".format(w, x, y, z, f))

# Compute the single step partials
df_du, df_dv = 2*(u + v), 2*(u + v)
du_da, du_db, dv_da, dv_db = 2*(a + b), 2*(a + b), 2*(a - b), -2*(a - b)
da_dw, db_dw = 2*(w + x + y + z), 2*(w + x - y - z)

# Compute the final result from inputs to outputs
du_dw, dv_dw = du_da*da_dw + du_db*db_dw, dv_da*da_dw + dv_db*db_dw
df_dw = df_du*du_dw + df_dv*dv_dw
print("df/dw at {}, {}, {}, {} is {}".format(w, x, y, z, df_dw))
```

```
    f at -1, 0, -2, 1 is 1024
df/dw at -1, 0, -2, 1 is -4096
```

However, note that this still does not make it easy to compute something like $\frac{\partial f}{\partial x}$. The reason for that is the *way* we chose to apply the chain rule. If we look at what we did above, we always kept ∂w in the denominator when we could. In this way, we chose to apply the chain rule seeing how w changed every other variable. If that is what we wanted, this would be a good idea. However, think back to our motivation from deep learning: we want to see how every parameter changes the *loss*. In essence, we want to apply the chain rule keeping ∂f in the numerator whenever we can!

To be more explicit, note that we can write

$$\begin{aligned} \frac{\partial f}{\partial w} & = \frac{\partial f}{\partial a}\frac{\partial a}{\partial w} + \frac{\partial f}{\partial b}\frac{\partial b}{\partial w}, \\ \frac{\partial f}{\partial a} & = \frac{\partial f}{\partial u}\frac{\partial u}{\partial a}+\frac{\partial f}{\partial v}\frac{\partial v}{\partial a}, \\ \frac{\partial f}{\partial b} & = \frac{\partial f}{\partial u}\frac{\partial u}{\partial b}+\frac{\partial f}{\partial v}\frac{\partial v}{\partial b}. \end{aligned} \tag{17.4.23}$$

Note that this application of the chain rule has us explicitly compute $\frac{\partial f}{\partial u}, \frac{\partial f}{\partial u}, \frac{\partial f}{\partial u}, \frac{\partial f}{\partial u},$ and $\frac{\partial f}{\partial u}$. Nothing stops us from also including the equations:

$$\begin{aligned} \frac{\partial f}{\partial x} & = \frac{\partial f}{\partial a}\frac{\partial a}{\partial x} + \frac{\partial f}{\partial b}\frac{\partial b}{\partial x}, \\ \frac{\partial f}{\partial y} & = \frac{\partial f}{\partial a}\frac{\partial a}{\partial y}+\frac{\partial f}{\partial b}\frac{\partial b}{\partial y}, \\ \frac{\partial f}{\partial z} & = \frac{\partial f}{\partial a}\frac{\partial a}{\partial z}+\frac{\partial f}{\partial b}\frac{\partial b}{\partial z}. \end{aligned} \tag{17.4.24}$$

and then keeping track of how f changes when we change *any* node in the entire network. Let's implement it.

```
# Compute the value of the function from inputs to outputs
w, x, y, z = -1, 0, -2, 1
a, b = (w + x + y + z)**2, (w + x - y - z)**2
```

(continues on next page)

(continued from previous page)

```
u, v = (a + b)**2, (a - b)**2
f = (u + v)**2
print("    f at {}, {}, {}, {} is {}".format(w, x, y, z, f))

# Compute the derivative using the decomposition above
# First compute the single step partials
df_du, df_dv = 2*(u + v), 2*(u + v)
du_da, du_db, dv_da, dv_db = 2*(a + b), 2*(a + b), 2*(a - b), -2*(a - b)
da_dw, db_dw = 2*(w + x + y + z), 2*(w + x - y - z)
da_dx, db_dx = 2*(w + x + y + z), 2*(w + x - y - z)
da_dy, db_dy = 2*(w + x + y + z), -2*(w + x - y - z)
da_dz, db_dz = 2*(w + x + y + z), -2*(w + x - y - z)

# Now compute how f changes when we change any value from output to input
df_da, df_db = df_du*du_da + df_dv*dv_da, df_du*du_db + df_dv*dv_db
df_dw, df_dx = df_da*da_dw + df_db*db_dw, df_da*da_dx + df_db*db_dx
df_dy, df_dz = df_da*da_dy + df_db*db_dy, df_da*da_dz + df_db*db_dz
print("df/dw at {}, {}, {}, {} is {}".format(w, x, y, z, df_dw))
print("df/dx at {}, {}, {}, {} is {}".format(w, x, y, z, df_dx))
print("df/dy at {}, {}, {}, {} is {}".format(w, x, y, z, df_dy))
print("df/dz at {}, {}, {}, {} is {}".format(w, x, y, z, df_dz))
```

```
    f at -1, 0, -2, 1 is 1024
df/dw at -1, 0, -2, 1 is -4096
df/dx at -1, 0, -2, 1 is -4096
df/dy at -1, 0, -2, 1 is -4096
df/dz at -1, 0, -2, 1 is -4096
```

The fact that we compute derivatives from f back towards the inputs rather than from the inputs forward to the outputs (as we did in the first code snippet above) is what gives this algorithm its name: *backpropagation*. Note that there are two steps: 1. Compute the value of the function, and the single step partials from front to back. While not done above, this can be combined into a single *forward pass*. 2. Compute the gradient of f from back to front. We call this the *backwards pass*.

This is precisely what every deep learning algorithm implements to allow the computation of the gradient of the loss with respect to every weight in the network at one pass. It is an astonishing fact that we have such a decomposition.

To see how MXNet has encapsulated this, let's take a quick look at this example.

```
# Initialize as ndarrays, then attach gradients
w, x, y, z = np.array(-1), np.array(0), np.array(-2), np.array(1)

w.attach_grad()
x.attach_grad()
y.attach_grad()
z.attach_grad()

# Do the computation like usual, tracking gradients
with autograd.record():
    a, b = (w + x + y + z)**2, (w + x - y - z)**2
    u, v = (a + b)**2, (a - b)**2
```

(continues on next page)

(continued from previous page)

```
    f = (u + v)**2

# Execute backward pass
f.backward()

print("df/dw at {}, {}, {}, {} is {}".format(w, x, y, z, w.grad))
print("df/dx at {}, {}, {}, {} is {}".format(w, x, y, z, x.grad))
print("df/dy at {}, {}, {}, {} is {}".format(w, x, y, z, y.grad))
print("df/dz at {}, {}, {}, {} is {}".format(w, x, y, z, z.grad))
```

```
df/dw at -1.0, 0.0, -2.0, 1.0 is -4096.0
df/dx at -1.0, 0.0, -2.0, 1.0 is -4096.0
df/dy at -1.0, 0.0, -2.0, 1.0 is -4096.0
df/dz at -1.0, 0.0, -2.0, 1.0 is -4096.0
```

All of what we did above can be done automatically by calling `f.backwards()`.

17.4.6 Hessians

As with single variable calculus, it is useful to consider higher-order derivatives in order to get a handle on how we can obtain a better approximation to a function than using the gradient alone.

There is one immediate problem one encounters when working with higher order derivatives of functions of several variables, and that is there are a large number of them. If we have a function $f(x_1, \ldots, x_n)$ of n variables, then we can take n^2 many second derivatives, namely for any choice of i and j:

$$\frac{d^2 f}{dx_i dx_j} = \frac{d}{dx_i}\left(\frac{d}{dx_j}f\right). \tag{17.4.25}$$

This is traditionally assembled into a matrix called the *Hessian*:

$$\mathbf{H}_f = \begin{bmatrix} \frac{d^2 f}{dx_1 dx_1} & \cdots & \frac{d^2 f}{dx_1 dx_n} \\ \vdots & \ddots & \vdots \\ \frac{d^2 f}{dx_n dx_1} & \cdots & \frac{d^2 f}{dx_n dx_n} \\ \end{bmatrix}. \tag{17.4.26}$$

Not every entry of this matrix is independent. Indeed, we can show that as long as both *mixed partials* (partial derivatives with respect to more than one variable) exist and are continuous, we can say that for any i, and j,

$$\frac{d^2 f}{dx_i dx_j} = \frac{d^2 f}{dx_j dx_i}. \tag{17.4.27}$$

This follows by considering first perturbing a function in the direction of x_i, and then perturbing it in x_j and then comparing the result of that with what happens if we perturb first x_j and then x_i, with the knowledge that both of these orders lead to the same final change in the output of f.

As with single variables, we can use these derivatives to get a far better idea of how the function behaves near a point. In particular, we can use it to find the best fitting quadratic near a point $\mathbf{x}_0$, as we saw in a single variable.

Let's see an example. Suppose that $f(x_1, x_2) = a + b_1x_1 + b_2x_2 + c_{11}x_1^2 + c_{12}x_1x_2 + c_{22}x_2^2$. This is the general form for a quadratic in two variables. If we look at the value of the function, its gradient, and its Hessian (17.4.26), all at the point zero:

$$\begin{aligned} f(0,0) &= a, \\ \nabla f(0,0) &= \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}, \\ \mathbf{H} f(0,0) &= \begin{bmatrix} 2c_{11} & c_{12} \\ c_{12} & 2c_{22} \end{bmatrix}. \end{aligned} \tag{17.4.28}$$

If we from this, we see we can get our original polynomial back by saying

$$f(\mathbf{x}) = f(0) + \nabla f(0) \cdot \mathbf{x} + \frac{1}{2}\mathbf{x}^\top \mathbf{H} f(0) \mathbf{x}. \tag{17.4.29}$$

In general, if we computed this expansion any point $\mathbf{x}_0$, we see that

$$f(\mathbf{x}) = f(\mathbf{x}_0) + \nabla f(\mathbf{x}_0) \cdot (\mathbf{x}-\mathbf{x}_0) + \frac{1}{2}(\mathbf{x}-\mathbf{x}_0)^\top \mathbf{H} f(\mathbf{x}_0)(\mathbf{x}-\mathbf{x}_0). \tag{17.4.30}$$

This works for any dimensional input, and provides the best approximating quadratic to any function at a point. To give an example, let's plot the function

$$f(x, y) = xe^{-x^2-y^2}. \tag{17.4.31}$$

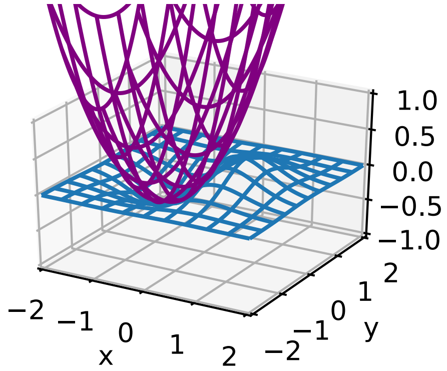
One can compute that the gradient and Hessian are

$$\nabla f(x, y) = e^{-x^2-y^2}\begin{pmatrix} 1-2x^2 \\ -2xy \end{pmatrix} \text{ and } \mathbf{H} f(x, y) = e^{-x^2-y^2}\begin{pmatrix} 4x^3 - 6x & 4x^2y - 2y \\ 4x^2y-2y & 4xy^2-2x \end{pmatrix}. \tag{17.4.32}$$

And thus, with a little algebra, see that the approximating quadratic at $[-1, 0]^\top$ is

$$f(x, y) \approx e^{-1}\left(-1 - (x+1) + 2(x+1)^2 + 2y^2\right). \tag{17.4.33}$$

```
# Construct grid and compute function
x, y = np.meshgrid(np.linspace(-2, 2, 101),
                   np.linspace(-2, 2, 101), indexing='ij')
z = x*np.exp(- x**2 - y**2)

# Compute gradient and Hessian at (1, 0)
w = np.exp(-1)*(-1 - (x + 1) + 2 * (x + 1)**2 + 2 * y**2)

# Plot function
ax = d2l.plt.figure().add_subplot(111, projection='3d')
ax.plot_wireframe(x, y, z, **{'rstride': 10, 'cstride': 10})
ax.plot_wireframe(x, y, w, **{'rstride': 10, 'cstride': 10}, color='purple')
d2l.plt.xlabel('x')
d2l.plt.ylabel('y')
d2l.set_figsize()
ax.set_xlim(-2, 2)
ax.set_ylim(-2, 2)
ax.set_zlim(-1, 1)
ax.dist = 12
```



This forms the basis for Newton's Algorithm discussed in Section 11.3, where we perform numerical optimization iteratively finding the best fitting quadratic, and then exactly minimizing that quadratic.

17.4.7 A Little Matrix Calculus

Derivatives of functions involving matrices turn out to be particularly nice. This section can become notationally heavy, so may be skipped in a first reading, but it is useful to know how derivatives of functions involving common matrix operations are often much cleaner than one might initially anticipate, particularly given how central matrix operations are to deep learning applications.

Let's begin with an example. Suppose that we have some fixed row vector $\boldsymbol{\beta}$, and we want to take the product function $f(\mathbf{x}) = \boldsymbol{\beta}\mathbf{x}$, and understand how the dot product changes when we change $\mathbf{x}$.

A bit of notation that will be useful when working with matrix derivatives in ML is called the *denominator layout matrix derivative* where we assemble our partial derivatives into the shape of whatever vector, matrix, or tensor is in the denominator of the differential. In this case, we will write

$$\frac{df}{d\mathbf{x}} = \begin{bmatrix} \frac{df}{dx_1} \\ \vdots \\ \frac{df}{dx_n} \end{bmatrix}. \tag{17.4.34}$$

where we matched the shape of the column vector $\mathbf{x}$.

If we write out our function into components this is

$$f(\mathbf{x}) = \sum_{i=1}^{n} \beta_i x_i = \beta_1 x_1 + \cdots + \beta_n x_n. \tag{17.4.35}$$

If we now take the partial derivative with respect to say β_1, note that everything is zero but the first term, which is just x_1 multiplied by β_1, so the we obtain that

$$\frac{df}{dx_1} = \beta_1, \tag{17.4.36}$$

or more generally that

$$\frac{df}{dx_i} = \beta_i. \tag{17.4.37}$$

We can now reassemble this into a matrix to see

$$\frac{df}{d\mathbf{x}} = \begin{bmatrix} \frac{df}{dx_1} \\ \vdots \\ \frac{df}{dx_n} \end{bmatrix} = \begin{bmatrix} \beta_1 \\ \vdots \\ \beta_n \end{bmatrix} = \boldsymbol{\beta}^\top. \tag{17.4.38}$$

This illustrates a few factors about matrix calculus that we will often counter throughout this section:

- First, The computations will get rather involved.
- Second, The final results are much cleaner than the intermediate process, and will always look similar to the single variable case. In this case, note that $\frac{d}{dx}(bx) = b$ and $\frac{d}{d\mathbf{x}}(\boldsymbol{\beta}\mathbf{x}) = \boldsymbol{\beta}^\top$ are both similar.
- Third, transposes can often appear seemingly from nowhere. The core reason for this is the convention that we match the shape of the denominator, thus when we multiply matrices, we will need to take transposes to match back to the shape of the original term.

To keep building intuition, let's try a computation that is a little harder. Suppose that we have a column vector **x**, and a square matrix A and we want to compute

$$\frac{d}{d\mathbf{x}}(\mathbf{x}^\top A\mathbf{x}). \tag{17.4.39}$$

To drive towards easier to manipulate notation, let's consider this problem using Einstein notation. In this case we can write the function as

$$\mathbf{x}^\top A\mathbf{x} = x_i a_{ij} x_j. \tag{17.4.40}$$

To compute our derivative, we need to understand for every k, what the value of

$$\frac{d}{dx_k}(\mathbf{x}^\top A\mathbf{x}) = \frac{d}{dx_k} x_i a_{ij} x_j. \tag{17.4.41}$$

By the product rule, this is

$$\frac{d}{dx_k} x_i a_{ij} x_j = \frac{dx_i}{dx_k} a_{ij} x_j + x_i a_{ij} \frac{dx_j}{dx_k}. \tag{17.4.42}$$

For a term like $\frac{dx_i}{dx_k}$, it is not hard to see that this is one when $i = k$ and zero otherwise. This means that every term where i and k are different vanish from this sum, so the only terms that remain in that first sum are the ones where $i = k$. The same reasoning holds for the second term where we need $j = k$. This gives

$$\frac{d}{dx_k} x_i a_{ij} x_j = a_{kj} x_j + x_i a_{ik}. \tag{17.4.43}$$

Now, the names of the indices in Einstein notation are arbitrary—the fact that i and j are different is immaterial to this computation at this point, so we can re-index so that they both use i to see that

$$\frac{d}{dx_k} x_i a_{ij} x_j = a_{ki} x_i + x_i a_{ik} = (a_{ki} + a_{ik}) x_i. \tag{17.4.44}$$

Now, here is where we start to need some practice to go further. Let's try and identify this outcome in terms of matrix operations. $a_{ki} + a_{ik}$ is the k, i-th component of $\mathbf{A} + \mathbf{A}^\top$. This gives

$$\frac{d}{dx_k} x_i a_{ij} x_j = [\mathbf{A} + \mathbf{A}^\top]_{ki} x_i. \tag{17.4.45}$$

Similarly, this term is now the product of the matrix $\mathbf{A} + \mathbf{A}^\top$ by the vector $\mathbf{x}$, so we see that

$$\left[\frac{d}{d\mathbf{x}}(\mathbf{x}^\top A\mathbf{x})\right]_k = \frac{d}{dx_k}x_i a_{ij} x_j = [(\mathbf{A} + \mathbf{A}^\top)\mathbf{x}]_k. \tag{17.4.46}$$

Thus, we see that the k-th entry of the desired derivative from (17.4.39) is just the k-th entry of the vector on the right, and thus the two are the same. Thus yields

$$\frac{d}{d\mathbf{x}}(\mathbf{x}^\top A\mathbf{x}) = (\mathbf{A} + \mathbf{A}^\top)\mathbf{x}. \tag{17.4.47}$$

This required significantly more work than our last one, but the final result is small. More than that, consider the following computation for traditional single variable derivatives:

$$\frac{d}{dx}(xax) = \frac{dx}{dx}ax + xa\frac{dx}{dx} = (a+a)x. \tag{17.4.48}$$

Equivalently $\frac{d}{dx}(ax^2) = 2ax = (a+a)x$. Again, we get a result that looks rather like the single variable result but with a transpose tossed in.

At this point, the pattern should be looking rather suspicious, so let's try to figure out why. When we take matrix derivatives like this, let's first assume that the expression we get will be another matrix expression: an expression we can write it in terms of products and sums of matrices and their transposes. If such an expression exists, it will need to be true for all matrices. In particular, it will need to be true of 1×1 matrices, in which case the matrix product is just the product of the numbers, the matrix sum is just the sum, and the transpose does nothing at all! In other words, whatever expression we get *must* match the single variable expression. This means that, with some practice, one can often guess matrix derivatives just by knowing what the associated single variable expression must look like!

Let's try this out. Suppose that $\mathbf{X}$ is a $n \times m$ matrix, $\mathbf{U}$ is an $n \times r$ and $\mathbf{V}$ is an $r \times m$. Let's try to compute

$$\frac{d}{d\mathbf{V}}\|\mathbf{X} - \mathbf{U}\mathbf{V}\|_2^2 = \,? \tag{17.4.49}$$

This computation is important in an area called matrix factorization. For us, however, it is just a derivative to compute. Let's try to imaging what this would be for 1×1 matrices. In that case, we get the expression

$$\frac{d}{dv}(x - uv)^2 = 2(x - uv)u, \tag{17.4.50}$$

where, the derivative is rather standard. If we try to convert this back into a matrix expression we get

$$\frac{d}{d\mathbf{V}}\|\mathbf{X} - \mathbf{U}\mathbf{V}\|_2^2 = 2(\mathbf{X} - \mathbf{U}\mathbf{V})\mathbf{U}. \tag{17.4.51}$$

However, if we look at this it does not quite work. Recall that $\mathbf{X}$ is $n \times m$, as is $\mathbf{U}\mathbf{V}$, so the matrix $2(\mathbf{X} - \mathbf{U}\mathbf{V})$ is $n \times m$. On the other hand $\mathbf{U}$ is $n \times r$, and we cannot multiply a $n \times m$ and a $n \times r$ matrix since the dimensions do not match!

We want to get $\frac{d}{d\mathbf{V}}$, which is the same shape of $\mathbf{V}$, which is $r \times m$. So somehow we need to take a $n \times m$ matrix and a $n \times r$ matrix, multiply them together (perhaps with some transposes) to get a $r \times m$. We can do this by multiplying $U^\top$ by $(\mathbf{X} - \mathbf{U}\mathbf{V})$. Thus, we can guess the solution to (17.4.49) is

$$\frac{d}{d\mathbf{V}}\|\mathbf{X} - \mathbf{U}\mathbf{V}\|_2^2 = 2\mathbf{U}^\top(\mathbf{X} - \mathbf{U}\mathbf{V}). \tag{17.4.52}$$

To show we that this works, we would be remiss to not provide a detailed computation. If we already believe that this rule-of-thumb works, feel free to skip past this derivation. To compute

$$\frac{d}{d\mathbf{V}}\|\mathbf{X}-\mathbf{U}\mathbf{V}\|_2^2, \tag{17.4.53}$$

we must find for every a, and b

$$\frac{d}{dv_{ab}}\|\mathbf{X}-\mathbf{U}\mathbf{V}\|_2^2 = \frac{d}{dv_{ab}}\sum_{i,j}\left(x_{ij}-\sum_k u_{ik}v_{kj}\right)^2. \tag{17.4.54}$$

Recalling that all entries of $\mathbf{X}$ and $\mathbf{U}$ are constants as far as $\frac{d}{dv_{ab}}$ is concerned, we may push the derivative inside the sum, and apply the chain rule to the square to get

$$\frac{d}{dv_{ab}}\|\mathbf{X}-\mathbf{U}\mathbf{V}\|_2^2 = \sum_{i,j}2\left(x_{ij}-\sum_k u_{ik}v_{kj}\right)\left(\sum_k u_{ik}\frac{dv_{kj}}{dv_{ab}}\right). \tag{17.4.55}$$

As in the previous derivation, we may note that $\frac{dv_{kj}}{dv_{ab}}$ is only non-zero if the $k = a$ and $j = b$. If either of those conditions do not hold, the term in the sum is zero, and we may freely discard it. We see that

$$\frac{d}{dv_{ab}}\|\mathbf{X}-\mathbf{U}\mathbf{V}\|_2^2 = \sum_{i}2\left(x_{ib}-\sum_k u_{ik}v_{kb}\right)u_{ia}. \tag{17.4.56}$$

An important subtlety here is that the requirement that $k = a$ does not occur inside the inner sum since that k is a dummy variable which we are summing over inside the inner term. For a notationally cleaner example, consider why

$$\frac{d}{dx_1}\left(\sum_i x_i\right)^2 = 2\left(\sum_i x_i\right). \tag{17.4.57}$$

From this point, we may start identifying components of the sum. First,

$$\sum_k u_{ik}v_{kb} = [\mathbf{U}\mathbf{V}]_{ib}. \tag{17.4.58}$$

So the entire expression in the inside of the sum is

$$x_{ib}-\sum_k u_{ik}v_{kb} = [\mathbf{X}-\mathbf{U}\mathbf{V}]_{ib}. \tag{17.4.59}$$

This means we may now write our derivative as

$$\frac{d}{dv_{ab}}\|\mathbf{X}-\mathbf{U}\mathbf{V}\|_2^2 = 2\sum_i[\mathbf{X}-\mathbf{U}\mathbf{V}]_{ib}u_{ia}. \tag{17.4.60}$$

We want this to look like the a, b element of a matrix so we can use the technique as in the previous example to arrive at a matrix expression, which means that we need to exchange the order of the indices on u_{ia}. If we notice that $u_{ia} = [\mathbf{U}^\top]_{ai}$, we can then write

$$\frac{d}{dv_{ab}}\|\mathbf{X}-\mathbf{U}\mathbf{V}\|_2^2 = 2\sum_i[\mathbf{U}^\top]_{ai}[\mathbf{X}-\mathbf{U}\mathbf{V}]_{ib}. \tag{17.4.61}$$

This is a matrix product, and thus we can conclude that

$$\frac{d}{dv_{ab}}\|\mathbf{X}-\mathbf{U}\mathbf{V}\|_2^2 = [2\mathbf{U}^\top(\mathbf{X}-\mathbf{U}\mathbf{V})]_{ab}. \tag{17.4.62}$$

and thus we may write the solution to (17.4.49)

$$\frac{d}{d\mathbf{V}}\|\mathbf{X}-\mathbf{U}\mathbf{V}\|_2^2 = 2\mathbf{U}^\top(\mathbf{X}-\mathbf{U}\mathbf{V}). \tag{17.4.63}$$

This matches the solution we guessed above!

It is reasonable to ask at this point, "Why can I not just write down matrix versions of all the calculus rules I have learned? It is clear this is still mechanical. Why do we not just get it over with!" And indeed there are such rules and (Petersen et al., 2008) provides an excellent summary. However, due to the plethora of ways matrix operations can be combined compared to single values, there are many more matrix derivative rules than single variable ones. It is often the case that it is best to work with the indices, or leave it up to automatic differentiation when appropriate.

Summary

- In higher dimensions, we can define gradients which serve the same purpose as derivatives in one dimension. These allow us to see how a multi-variable function changes when we make an arbitrary small change to the inputs.
- The backpropagation algorithm can be seen to be a method of organizing the multi-variable chain rule to allow for the efficient computation of many partial derivatives.
- Matrix calculus allows us to write the derivatives of matrix expressions in concise ways.

Exercises

1. Given a row vector $\boldsymbol{\beta}$, compute the derivatives of both $f(\mathbf{x}) = \boldsymbol{\beta}\mathbf{x}$ and $g(\mathbf{x}) = \mathbf{x}^\top\boldsymbol{\beta}^\top$. Why do you get the same answer?
2. Let $\mathbf{v}$ be an n dimension vector. What is $\frac{\partial}{\partial\mathbf{v}}\|\mathbf{v}\|_2$?
3. Let $L(x, y) = \log(e^x + e^y)$. Compute the gradient. What is the sum of the components of the gradient?
4. Let $f(x, y) = x^2y + xy^2$. Show that the only critical point is $(0, 0)$. By considering $f(x, x)$, determine if $(0, 0)$ is a maximum, minimum, or neither.
5. Suppose that we are minimizing a function $f(\mathbf{x}) = g(\mathbf{x}) + h(\mathbf{x})$. How can we geometrically interpret the condition of $\nabla f = 0$ in terms of g and h?

17.5 Integral Calculus

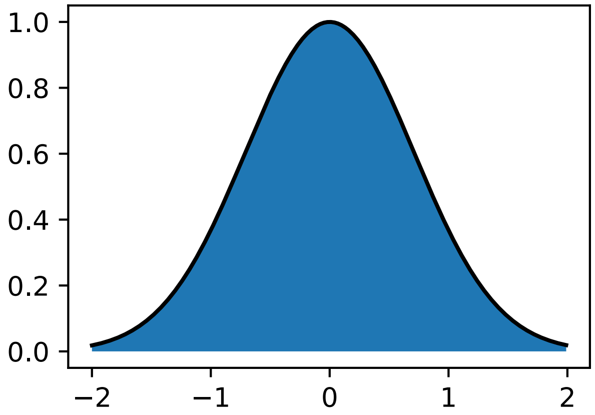
Differentiation only makes up half of the content of a traditional calculus education. The other pillar, integration, starts out seeming a rather disjoint question, "What is the area underneath this curve?" While seemingly unrelated, integration is tightly intertwined with the differentiation via what is known as the *fundamental theorem of calculus*.

At the level of machine learning we discuss in this book, we will not need a deep understanding of integration. However, we will provide a brief introduction to lay the groundwork for any further applications we will encounter later on.

17.5.1 Geometric Interpretation

Suppose that we have a function $f(x)$. For simplicity, let's assume that $f(x)$ is non-negative (never takes a value less than zero). What we want to try and understand is: what is the area contained between $f(x)$ and the x-axis?

```
%matplotlib inline
import d2l
from IPython import display
from mpl_toolkits import mplot3d
from mxnet import np, npx
npx.set_np()

x = np.arange(-2, 2, 0.01)
f = np.exp(-x**2)

d2l.set_figsize()
d2l.plt.plot(x, f, color='black')
d2l.plt.fill_between(x.tolist(), f.tolist())
d2l.plt.show()
```



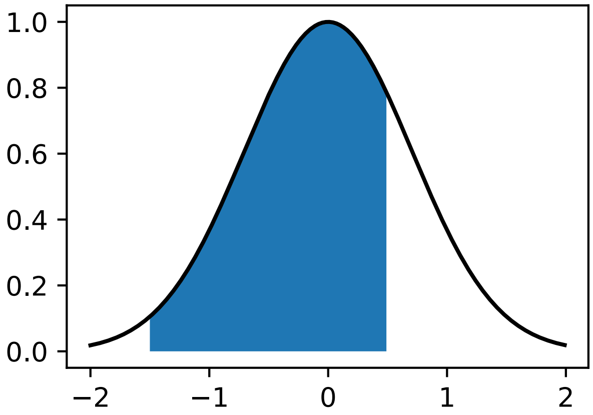
In most cases, this area will be infinite or undefined (consider the area under $f(x) = x^2$), so people will often talk about the area between a pair of ends, say a and b.

```
x = np.arange(-2, 2, 0.01)
f = np.exp(-x**2)
```

(continues on next page)

(continued from previous page)

```
d2l.set_figsize()
d2l.plt.plot(x, f, color='black')
d2l.plt.fill_between(x.tolist()[50:250], f.tolist()[50:250])
d2l.plt.show()
```



We will denote this area by the integral symbol below:

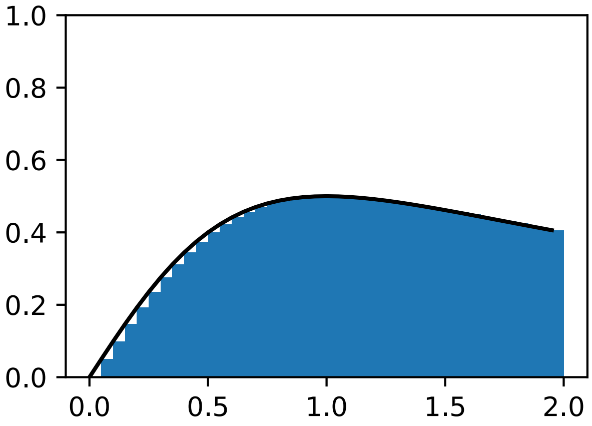
$$\text{Area}(\mathcal{A}) = \int_a^b f(x) \; dx. \tag{17.5.1}$$

The inner variable is a dummy variable, much like the index of a sum in a $\sum$, and so this can be equivalently written with any inner value we like:

$$\int_a^b f(x) \; dx = \int_a^b f(z) \; dz. \tag{17.5.2}$$

There is a traditional way to try and understand how we might try to approximate such integrals: we can imaging taking the region in-between a and b and chopping it into N vertical slices. If N is large, we can approximate the area of each slice by a rectangle, and then add up the areas to get the total area under the curve. Let's take a look at an example doing this in code. We will see how to get the true value in a later section.

```
epsilon = 0.05
a = 0
b = 2

x = np.arange(a, b, epsilon)
f = x / (1 + x**2)

approx = np.sum(epsilon*f)
true = np.log(2) / 2

d2l.set_figsize()
d2l.plt.bar(x.asnumpy(), f.asnumpy(), width=epsilon, align='edge')
d2l.plt.plot(x, f, color='black')
d2l.plt.ylim([0, 1])
d2l.plt.show()

"Approximation: {}, Truth: {}".format(approx, true)
```



```
'Approximation: 0.79448557, Truth: 0.34657359027997264'
```

The issue is that while it can be done numerically, we can do this approach analytically for only the simplest functions like

$$\int_a^b x \; dx. \tag{17.5.3}$$

Anything somewhat more complex like our example from the code above

$$\int_a^b \frac{x}{1+x^2} \; dx. \tag{17.5.4}$$

is beyond what we can solve with such a direct method.

We will instead take a different approach. We will work intuitively with the notion of the area, and learn the main computational tool used to find integrals: the *fundamental theorem of calculus*. This will be the basis for our study of integration.

17.5.2 The Fundamental Theorem of Calculus

To dive deeper into the theory of integration, let's introduce a function

$$F(x) = \int_0^x f(y) dy. \tag{17.5.5}$$

This function measures the area between 0 and x depending on how we change x. Notice that this is everything we need since

$$\int_a^b f(x) \; dx = F(b) - F(a). \tag{17.5.6}$$

This is a mathematical encoding of the fact that we can measure the area out to the far end-point and then subtract off the area to the near end point as indicated in Fig. 17.5.1.

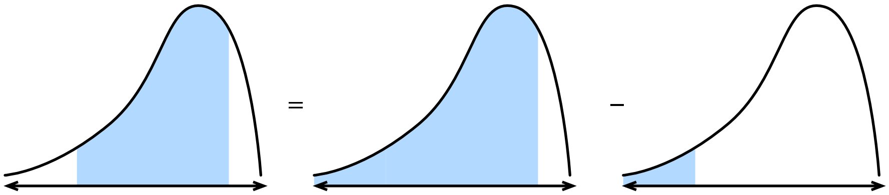


Fig. 17.5.1: Visualizing why we may reduce the problem of computing the area under a curve between two points to computing the area to the left of a point.

Thus, if we can figure out what the integral over any interval is by figuring out what $F(x)$ is.

To do so, let's consider an experiment. As we often do in calculus, let's imaging what happens when we shift the value by a tiny bit. From the comment above, we know that

$$F(x+\epsilon) - F(x) = \int_x^{x+\epsilon} f(y) \; dy. \tag{17.5.7}$$

This tells us that the function changes by the area under a tiny sliver of a function.

This is the point at which we make an approximation. If we look at a tiny sliver of area like this, it looks like this area is close to the rectangular area with height the value of $f(x)$ and the base width ϵ. Indeed, one can show that as $\epsilon \rightarrow 0$ this approximation becomes better and better. Thus we can conclude:

$$F(x+\epsilon) - F(x) \approx \epsilon f(x). \tag{17.5.8}$$

However, we can now notice: this is exactly the pattern we expect if we were computing the derivative of F! Thus we see the following rather surprising fact:

$$\frac{dF}{dx}(x) = f(x). \tag{17.5.9}$$

This is the *fundamental theorem of calculus*. We may write it in expanded form as

$$\frac{d}{dx}\int_{-\infty}^{x} f(y) \; dy = f(x). \tag{17.5.10}$$

It takes the concept of finding areas (*a priori* rather hard), and reduces it to a statement derivatives (something much more completely understood). One last comment that we must make is that this does not tell we exactly what $F(x)$. Indeed $F(x)+C$ for any C has the same derivative. This is a fact-of-life in the theory of integration. Thankfully, notice that when working with definite integrals, the constants drop out, and thus are irrelevant to the outcome.

$$\int_a^b f(x) \; dx = (F(b) + C) - (F(a) + C) = F(b) - F(a). \tag{17.5.11}$$

This may seem like abstract non-sense, but let's take a moment to appreciate that it has given us a whole new perspective on computing integrals. Our goal is no-longer to do some sort of chop-and-sum process to try and recover the area, rather we need only find a function whose derivative is the function we have! This is incredible since we can now list many rather difficult integrals by just reversing the table from Section 17.3.2. For instance, we know that the derivative of x^n is nx^{n-1}. Thus, we can say using the fundamental theorem (17.5.10) that

$$\int_0^{x} ny^{n-1} \; dy = x^n - 0^n = x^n. \tag{17.5.12}$$

Similarly, we know that the derivative of e^x is itself, so that means

$$\int_0^x e^x \; dx = e^x - e^0 = e^x - 1. \tag{17.5.13}$$

In this way, we can develop the entire theory of integration leveraging ideas from differential calculus freely. Every integration rule derives from this one fact.

17.5.3 Change of Variables

Just as with differentiation, there are a number of rules which make the computation of integrals more tractable. In fact, every rule of differential calculus (like the product rule, sum rule, and chain rule) has a corresponding rule for integral calculus (integration by parts, linearity of integration, and the change of variables formula respectively). In this section, we will dive into what is arguably the most important from the list: the change of variables formula.

First, suppose that we have a function which is itself an integral:

$$F(x) = \int_0^x f(y) \; dy. \tag{17.5.14}$$

Let's suppose that we want to know how this function looks when we compose it with another to obtain $F(u(x))$. By the chain rule, we know

$$\frac{d}{dx}F(u(x)) = \frac{dF}{dx}(u(x))\cdot \frac{du}{dx}. \tag{17.5.15}$$

We can turn this into a statement about integration by using the fundamental theorem (17.5.10) as above. This gives

$$F(u(x)) - F(u(0)) = \int_0^x \frac{dF}{dx}(u(y))\cdot \frac{du}{dy} \;dy. \tag{17.5.16}$$

Recalling that F is itself an integral gives that the left hand side may be rewritten to be

$$\int_{u(0)}^{u(x)} f(y) \; dy = \int_0^x \frac{dF}{dx}(u(y))\cdot \frac{du}{dy} \;dy. \tag{17.5.17}$$

Similarly, recalling that F is an integral allows us to recognize that $\frac{dF}{dx} = f$ using the fundamental theorem (17.5.10), and thus we may conclude

$$\int_{u(0)}^{u(x)} f(y) \; dy = \int_0^x f(u(y))\cdot \frac{du}{dy} \;dy. \tag{17.5.18}$$

This is the *change of variables* formula.

For a more intuitive derivation, consider what happens when we take an integral of $f(u(x))$ between x and $x+\epsilon$. For a small ϵ, this integral is approximately $\epsilon f(u(x))$, the area of the associated rectangle. Now, let's compare this with the integral of $f(y)$ from $u(x)$ to $u(x+\epsilon)$. We know that $u(x+\epsilon) \approx u(x) + \epsilon \frac{du}{dx}(x)$, so the area of this rectangle is approximately $\epsilon \frac{du}{dx}(x)f(u(x))$. Thus, to make the area of these two rectangles to agree, we need to multiply the first one by $\frac{du}{dx}(x)$ as is illustrated in Fig. 17.5.2.

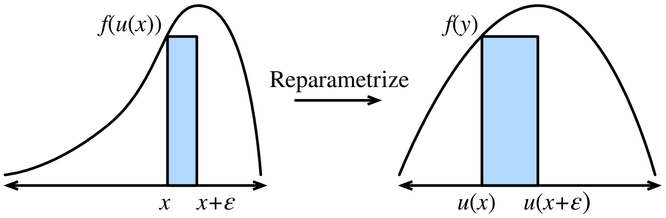


Fig. 17.5.2: Visualizing the transformation of a single thin rectangle under the change of variables.

This tells us that

$$\int_x^{x+\epsilon} f(u(y))\frac{du}{dy}(y)\;dy = \int_{u(x)}^{u(x+\epsilon)} f(y)\;dy. \tag{17.5.19}$$

This is the change of variables formula expressed for a single small rectangle.

If $u(x)$ and $f(x)$ are properly chosen, this can allow for the computation of incredibly complex integrals. For instance, if we even chose $f(y) = 1$ and $u(x) = e^{-x^2}$ (which means $\frac{du}{dx}(x) = -2xe^{-x^2}$, this can show for instance that

$$e^{-1} - 1 = \int_{e^{-0}}^{e^{-1}} 1\;dy = -2\int_0^1 ye^{-y^2}\;dy, \tag{17.5.20}$$

and thus by rearranging that

$$\int_0^1 ye^{-y^2}\;dy = \frac{1-e^{-1}}{2}. \tag{17.5.21}$$

17.5.4 A Comment on Sign Conventions

Keen-eyed readers will observe something strange about the computations above. Namely, computations like

$$\int_{e^{-0}}^{e^{-1}} 1\;dy = e^{-1} - 1 < 0, \tag{17.5.22}$$

can produce negative numbers. When thinking about areas, it can be strange to see a negative value, and so it is worth digging into what the convention is.

Mathematicians take the notion of signed areas. This manifests itself in two ways. First, if we consider a function $f(x)$ which is sometimes less than zero, then the area will also be negative. So for instance

$$\int_0^1 (-1)\;dx = -1. \tag{17.5.23}$$

Similarly, integrals which progress from right to left, rather than left to right are also taken to be negative areas

$$\int_0^{-1} 1\;dx = -1. \tag{17.5.24}$$

The standard area (from left to right of a positive function) is always positive. Anything obtained by flipping it (say flipping over the x-axis to get the integral of a negative number, or flipping over the y-axis to get an integral in the wrong order) will produce a negative area. And indeed, flipping twice will give a pair of negative signs that cancel out to have positive area

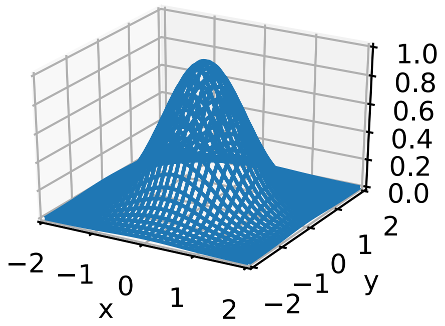
$$\int_0^{-1} (-1) \; dx = 1. \tag{17.5.25}$$

If this discussion sounds familiar, it is! In Section 17.1 we discussed how the determinant represented the signed area in much the same way.

17.5.5 Multiple Integrals

In some cases, we will need to work in higher dimensions. For instance, suppose that we have a function of two variables, like $f(x, y)$ and we want to know the volume under f when x ranges over $[a, b]$ and y ranges over $[c, d]$.

```
# Construct grid and compute function
x, y = np.meshgrid(np.linspace(-2, 2, 101), np.linspace(-2, 2, 101),
                   indexing='ij')
z = np.exp(- x**2 - y**2)

# Plot function
ax = d2l.plt.figure().add_subplot(111, projection='3d')
ax.plot_wireframe(x, y, z)
d2l.plt.xlabel('x')
d2l.plt.ylabel('y')
d2l.plt.xticks([-2, -1, 0, 1, 2])
d2l.plt.yticks([-2, -1, 0, 1, 2])
d2l.set_figsize()
ax.set_xlim(-2, 2)
ax.set_ylim(-2, 2)
ax.set_zlim(0, 1)
ax.dist = 12
```



We write this as

$$\int_{[a, b]\times[c, d]} f(x, y) \; dx \; dy. \tag{17.5.26}$$

Suppose that we wish to compute this integral. My claim is that we can do this by iteratively computing first the integral in say x and then shifting to the integral in y, that is to say

$$\int_{[a, b]\times[c, d]} f(x, y) \; dx \; dy = \int_c^{d} \left(\int_a^{b} f(x, y) \; dx\right) \; dy. \tag{17.5.27}$$

Let's see why this is.

Consider the figure above where we have split the function into $\epsilon \times \epsilon$ squares which we will index with integer coordinates i, j. In this case, our integral is approximately

$$\sum_{i,j} \epsilon^{2} f(\epsilon i, \epsilon j). \tag{17.5.28}$$

Once we discretize the problem, we may add up the values on these squares in whatever order we like, and not worry about changing the values. This is illustrated in Fig. 17.5.3. In particular, we can say that

$$\sum_{j} \epsilon \left(\sum_{i} \epsilon f(\epsilon i, \epsilon j)\right). \tag{17.5.29}$$

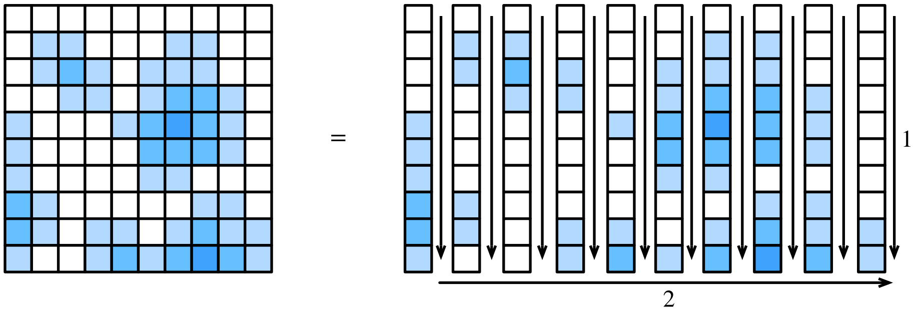


Fig. 17.5.3: Illustrating how to decompose a sum over many squares as a sum over first the columns (1), then adding the column sums together (2).

The sum on the inside is precisely the discretization of the integral

$$G(\epsilon j) = \int_{a}^{b} f(x, \epsilon j) \; dx. \tag{17.5.30}$$

Finally, notice that if we combine these two expressions we get

$$\sum_{j} \epsilon G(\epsilon j) \approx \int_{c}^{d} G(y) \; dy = \int_{[a,b]\times[c,d]} f(x, y) \; dx \; dy. \tag{17.5.31}$$

Thus putting it all together, we have that

$$\int_{[a,b]\times[c,d]} f(x, y) \; dx \; dy = \int_{c}^{d} \left(\int_{a}^{b} f(x, y) \; dx\right) \; dy. \tag{17.5.32}$$

Notice that, once discretized, all we did was rearrange the order in which we added a list of numbers. This may make it seem like it is nothing, however this result (called *Fubini's Theorem*) is not always true! For the type of mathematics encountered when doing machine learning (continuous functions), there is no concern, however it is possible to create examples where it fails (for example the function $f(x, y) = xy(x^2 - y^2)/(x^2 + y^2)^3$ over the rectangle $[0, 2] \times [0, 1]$).

Note that the choice to do the integral in x first, and then the integral in y was arbitrary. We could have equally well chosen to do y first and then x to see

$$\int_{[a,b]\times[c,d]} f(x, y) \; dx \; dy = \int_{a}^{b} \left(\int_{c}^{d} f(x, y) \; dy\right) \; dx. \tag{17.5.33}$$

Often times, we will condense down to vector notation, and say that for $U = [a, b] \times [c, d]$ this is

$$\int_U f(\mathbf{x})\ d\mathbf{x}. \tag{17.5.34}$$

17.5.6 Change of Variables in Multiple Integrals

As we with single variables in (17.5.18), the ability to change variables inside a higher dimensional integral is a key tool. Let's summarize the result without derivation.

We need a function that reparametrizes our domain of integration. We can take this to be $\phi : \mathbb{R}^n \to \mathbb{R}^n$, that is any function which takes in n real variables and returns another n. To keep the expressions clean, we will assume that ϕ is *injective* which is to say it never folds over itself ($\phi(\mathbf{x}) = \phi(\mathbf{y}) \implies \mathbf{x} = \mathbf{y}$).

In this case, we can say that

$$\int_{\phi(U)} f(\mathbf{x})\ d\mathbf{x} = \int_U f(\phi(\mathbf{x})) \left|\det(D\phi(\mathbf{x}))\right|\ d\mathbf{x}. \tag{17.5.35}$$

where $D\phi$ is the *Jacobian* of ϕ, which is the matrix of partial derivatives of $\boldsymbol{\phi} = (\phi_1(x_1, \ldots, x_n), \ldots, \phi_n(x_1, \ldots, x_n))$,

$$D\boldsymbol{\phi} = \begin{bmatrix} \frac{\partial \phi_1}{\partial x_1} & \cdots & \frac{\partial \phi_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial \phi_n}{\partial x_1} & \cdots & \frac{\partial \phi_n}{\partial x_n} \end{bmatrix}. \tag{17.5.36}$$

Looking closely, we see that this is similar to the single variable chain rule (17.5.18), except we have replaced the term $\frac{du}{dx}(x)$ with $\left|\det(D\phi(\mathbf{x}))\right|$. Let's see how we can to interpret this term. Recall that the $\frac{du}{dx}(x)$ term existed to say how much we stretched our x-axis by applying u. The same process in higher dimensions is to determine how much we stretch the area (or volume, or hyper-volume) of a little square (or little *hyper-cube*) by applying $\boldsymbol{\phi}$. If $\boldsymbol{\phi}$ was the multiplication by a matrix, then we know how the determinant already gives the answer.

With some work, one can show that the *Jacobian* provides the best approximation to a multivariable function $\boldsymbol{\phi}$ at a point by a matrix in the same way we could approximate by lines or planes with derivatives and gradients. Thus the determinant of the Jacobian exactly mirrors the scaling factor we identified in one dimension.

It takes some work to fill in the details to this, so do not worry if they are not clear now. Let's see at least one example we will make use of later on. Consider the integral

$$\int_{-\infty}^{\infty} \int_{-\infty}^{\infty} e^{-x^2-y^2}\ dx\ dy. \tag{17.5.37}$$

Playing with this integral directly will get us no-where, but if we change variables, we can make significant progress. If we let $\boldsymbol{\phi}(r, \theta) = (r\cos(\theta), r\sin(\theta))$ (which is to say that $x = r\cos(\theta)$, $y = r\sin(\theta)$), then we can apply the change of variable formula to see that this is the same thing as

$$\int_0^{\infty} \int_0^{2\pi} e^{-r^2} \left|\det(D\phi(\mathbf{x}))\right|\ d\theta\ dr, \tag{17.5.38}$$

where

$$\left|\det(D\phi(\mathbf{x}))\right| = \left|\det \begin{bmatrix} \cos(\theta) & -r\sin(\theta) \\ \sin(\theta) & r\cos(\theta) \end{bmatrix}\right| = r(\cos^2(\theta) + \sin^2(\theta)) = r. \tag{17.5.39}$$

Thus, the integral is

$$\int_0^{\infty} \int_0^{2\pi} re^{-r^2} \, d\theta \, dr = 2\pi \int_0^{\infty} re^{-r^2} \, dr = \pi, \tag{17.5.40}$$

where the final equality follows by the same computation that we used in section Section 17.5.3.

We will meet this integral again when we study continuous random variables in Section 17.6.

Summary

- The theory of integration allows us to answer questions about areas or volumes.
- The fundamental theorem of calculus allows us to leverage knowledge about derivatives to compute areas via the observation that the derivative of the area up to some point is given by the value of the function being integrated.
- Integrals in higher dimensions can be computed by iterating single variable integrals.

Exercises

1. What is $\int_1^2 \frac{1}{x} \, dx$?
2. Use the change of variables formula to integrate $\int_0^{\sqrt{\pi}} x \sin(x^2) \, dx$.
3. What is $\int_{[0,1]^2} xy \, dx \, dy$?
4. Use the change of variables formula to compute $\int_0^2 \int_0^1 xy(x^2 - y^2)/(x^2 + y^2)^3 \ dy \ dx$ and $\int_0^1 \int_0^2 f(x, y) = xy(x^2 - y^2)/(x^2 + y^2)^3 \, dx \, dy$ to see they are different.

17.6 Random Variables

In Section 2.6 we saw the basics of how to work with discrete random variables, which in our case refer to those random variables which take either a finite set of possible values, or the integers. In this section, we develop the theory of *continuous random variables*, which are random variables which can take on any real value.

17.6.1 Continuous Random Variables

Continuous random variables are a significantly more subtle topic than discrete random variables. A fair analogy to make is that the technical jump is comparable to the jump between adding lists of numbers and integrating functions. As such, we will need to take some time to develop the theory.

From Discrete to Continuous

To understand the additional technical challenges encountered when working with continuous random variables, let's perform a thought experiment. Suppose that we are throwing a dart at the dart board, and we want to know the probability that it hits exactly 2cm from the center of the board.

To start with, we imagine measuring to a single digit of accuracy, that is to say with bins for 0cm, 1cm, 2cm, and so on. We throw say 100 darts at the dart board, and if 20 of them fall into the bin for 2cm we conclude that 20% of the darts we throw hit the board 2cm away from the center.

However, when we look closer, this does not match our question! We wanted exact equality, whereas these bins hold all that fell between say 1.5cm and 2.5cm.

Undeterred, we continue further. We measure even more precisely, say 1.9cm, 2.0cm, 2.1cm, and now see that perhaps 3 of the 100 darts hit the board in the 2.0cm bucket. Thus we conclude the probability is 3%.

However, this does not solve anything! We have just pushed the issue down one digit further. Let's abstract a bit. Imagine we know the probability that the first k digits match with $2.00000\ldots$ and we want to know the probability it matches for the first $k+1$ digits. It is fairly reasonable to assume that the $k+1^{\text{th}}$ digit is essentially a random choice from the set $\{0, 1, 2, \ldots, 9\}$. At least, we cannot conceive of a physically meaningful process which would force the number of micrometers away form the center to prefer to end in a 7 vs a 3.

What this means is that in essence each additional digit of accuracy we require should decrease probability of matching by a factor of 10. Or put another way, we would expect that

$$P(\text{distance is } 2.00\ldots \text{ to } k \text{ digits}) \approx p \cdot 10^{-k}. \tag{17.6.1}$$

The value p essentially encodes what happens with the first few digits, and the 10^{-k} handles the rest.

Notice that if we know the position accurate to $k=4$ digits after the decimal. that means we know the value falls within the interval say $[(1.99995, 2.00005]$ which is an interval of length $2.00005 - 1.99995 = 10^{-4}$. Thus, if we call the length of this interval ϵ, we can say

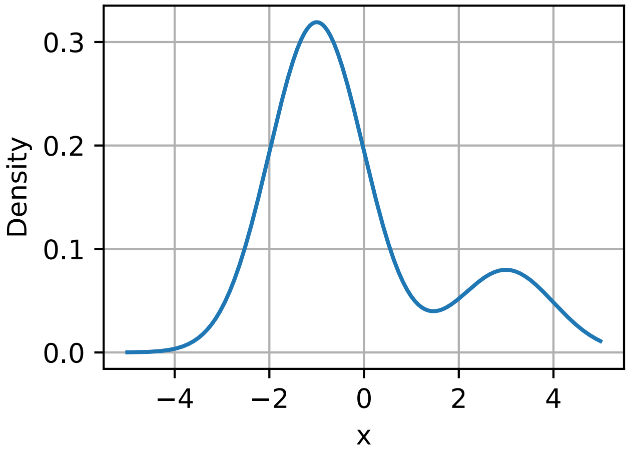
$$P(\text{distance is in an } \epsilon\text{-sized interval around } 2) \approx \epsilon \cdot p. \tag{17.6.2}$$

Let's take this one final step further. We have been thinking about the point 2 the entire time, but never thinking about other points. Nothing is different there fundamentally, but it is the case that the value p will likely be different. We would at least hope that a dart thrower was more likely to hit a point near the center, like 2cm rather than 20cm. Thus, the value p is not fixed, but rather should depend on the point x. This tells us that we should expect

$$P(\text{distance is in an } \epsilon\text{-sized interval around } x) \approx \epsilon \cdot p(x). \tag{17.6.3}$$

Indeed, (17.6.3) precisely defines the *probability density function*. It is a function $p(x)$ which encodes the relative probability of hitting near one point versus another. Let's visualize what such a function might look like.

```
%matplotlib inline
import d2l
from IPython import display
from mxnet import np, npx
npx.set_np()

# Plot the probability density function for some random variable
x = np.arange(-5, 5, 0.01)
p = 0.2*np.exp(-(x - 3)**2 / 2)/np.sqrt(2 * np.pi) + \
    0.8*np.exp(-(x + 1)**2 / 2)/np.sqrt(2 * np.pi)

d2l.plot(x, p, 'x', 'Density')
```



The locations where the function value is large indicates regions where we are more likely to find the random value. The low portions are areas where we are unlikely to find the random value.

Probability Density Functions

Let's now investigate this further. We have already seen what a probability density function is intuitively for a random variable X, namely the density function is a function $p(x)$ so that

$$P(X \text{ is in an } \epsilon\text{-sized interval around } x) \approx \epsilon \cdot p(x). \tag{17.6.4}$$

But what does this imply for the properties of $p(x)$?

First, probabilities are never negative, thus we should expect that $p(x) \ge 0$ as well.

Second, let's imagine that we slice up the $\mathbb{R}$ into an infinite number of slices which are ϵ wide, say with slices $(\epsilon \cdot i, \epsilon \cdot (i+1)]$. For each of these, we know from (17.6.4) the probability is approximately

$$P(X \text{ is in an } \epsilon\text{-sized interval around } x) \approx \epsilon \cdot p(\epsilon \cdot i), \tag{17.6.5}$$

so summed over all of them it should be

$$P(X \in \mathbb{R}) \approx \sum_i \epsilon \cdot p(\epsilon \cdot i). \tag{17.6.6}$$

This is nothing more than the approximation of an integral discussed in Section 17.5, thus we can say that

$$P(X \in \mathbb{R}) = \int_{-\infty}^{\infty} p(x) \; dx. \tag{17.6.7}$$

We know that $P(X \in \mathbb{R}) = 1$, since the random variable must take on *some* number, we can conclude that for any density

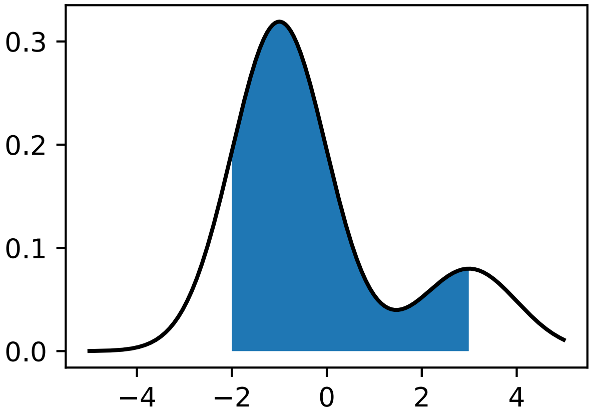
$$\int_{-\infty}^{\infty} p(x) \; dx = 1. \tag{17.6.8}$$

Indeed, digging into this further shows that for any a, and b, we see that

$$P(X \in (a, b]) = \int_{a}^{b} p(x) \; dx. \tag{17.6.9}$$

We may approximate this is code by using the same discrete approximation methods as before. In this case we can approximate the probability of falling in the blue region.

```
# Approximate probability using numerical integration
epsilon = 0.01
x = np.arange(-5, 5, 0.01)
p = 0.2*np.exp(-(x - 3)**2 / 2) / np.sqrt(2 * np.pi) + \
    0.8*np.exp(-(x + 1)**2 / 2) / np.sqrt(2 * np.pi)

d2l.set_figsize()
d2l.plt.plot(x, p, color='black')
d2l.plt.fill_between(x.tolist()[300:800], p.tolist()[300:800])
d2l.plt.show()

"Approximate Probability: {}".format(np.sum(epsilon*p[300:800]))
```



```
'Approximate Probability: 0.7736172'
```

It turns out that these two properties describe exactly the space of possible probability density functions (or *p.d.f.*'s for the commonly encountered abbreviation). They are non-negative functions $p(x) \geq 0$ such that

$$\int_{-\infty}^{\infty} p(x) \; dx = 1. \tag{17.6.10}$$

We interpret this function by using integration to obtain the probability our random variable is in a specific interval:

$$P(X \in (a, b]) = \int_a^b p(x) \; dx. \tag{17.6.11}$$

In `sec_distributions` we will see a number of common distributions, but let's continue working in the abstract.

Cumulative Distribution Functions

In the previous section, we saw the notion of the p.d.f. In practice, this is a commonly encountered method to discuss continuous random variables, but it has one significant pitfall: that the values of the p.d.f. are not themselves probabilities, but rather a function that we must integrate to yield probabilities. There is nothing wrong with a density being larger than 10, as long as it is not larger than 10 for more than an interval of length $1/10$. This can be counter-intuitive, so people often also think in terms of the *cumulative distribution function*, or c.d.f., which *is* a probability.

In particular, by using (17.6.11), we define the c.d.f. for a random variable X with density $p(x)$ by

$$F(x) = \int_{-\infty}^{x} p(x) \; dx = P(X \le x). \tag{17.6.12}$$

Let's observe a few properties.

- $F(x) \rightarrow 0$ as $x\rightarrow -\infty$.
- $F(x) \rightarrow 1$ as $x\rightarrow \infty$.
- $F(x)$ is non-decreasing ($y > x \implies F(y) \ge F(x)$).
- $F(x)$ is continuous (has no jumps) if X is a continuous random variable.

With the fourth bullet point, note that this would not be true if X were discrete, say taking the values 0 and 1 both with probability $1/2$. In that case

$$F(x) = \begin{cases} 0 & x < 0, \\ \frac{1}{2} & x < 1, \\ 1 & x \ge 1. \end{cases} \tag{17.6.13}$$

In this example, we see one of the benefits of working with the c.d.f., the ability to deal with continuous or discrete random variables in the same framework, or indeed mixtures of the two (flip a coin: if heads return the roll of a die, if tails return the distance of a dart throw from the center of a dart board).

Means

Suppose that we are dealing with a random variables X. The distribution itself can be hard to interpret. It is often useful to be able to summarize the behavior of a random variable concisely. Numbers that help us capture the behavior of a random variable are called *summary statistics*. The most commonly encountered ones are the *mean*, the *variance*, and the *standard deviation*.

The *mean* encodes the average value of a random variable. If we have a discrete random variable X, which takes the values x_i with probabilities p_i, then the mean is given by the weighted average: sum the values times the probability that the random variable takes on that value:

$$\mu_X = E[X] = \sum_i x_i p_i. \tag{17.6.14}$$

The way we should interpret the mean (albeit with caution) is that it tells us essentially where the random variable tends to be located.

As a minimalistic example that we will examine throughout this section, let's take X to be the random variable which takes the value $a-2$ with probability p, $a+2$ with probability p and a with probability $1-2p$. We can compute using (17.6.14) that, for any possible choice of a and p, the mean is

$$\mu_X = E[X] = \sum_i x_i p_i = (a-2)p + a(1-2p) + (a+2)p = a. \tag{17.6.15}$$

Thus we see that the mean is a. This matches the intuition since a is the location around which we centered our random variable.

Because they are helpful, let's summarize a few properties.

- For any random variable X and numbers a and b, we have that $\mu_{aX+b} = a\mu_X + b$.
- If we have two random variables X and Y, we have $\mu_{X+Y} = \mu_X + \mu_Y$.

Means are useful for understanding the average behavior of a random variable, however the mean is not sufficient to even have a full intuitive understanding. Making a profit of $\$10 \pm \$1$ per sale is very different from making $\$10 \pm \$15$ per sale despite having the same average value. The second one has a much larger degree of fluctuation, and thus represents a much larger risk. Thus, to understand the behavior of a random variable, we will need at minimum one more measure: some measure of how widely a random variable fluctuates.

Variances

This leads us to consider the *variance* of a random variable. This is a quantitative measure of how far a random variable deviates from the mean. Consider the expression $X - \mu_X$. This is the deviation of the random variable from its mean. This value can be positive or negative, so we need to do something to make it positive so that we are measuring the magnitude of the deviation.

A reasonable thing to try is to look at $\left|X-\mu_X\right|$, and indeed this leads to a useful quantity called the *mean absolute deviation*, however due to connections with other areas of mathematics and statistics, people often use a different solution.

In particular, they look at $(X-\mu_X)^2$. If we look at the typical size of this quantity by taking the mean, we arrive at the variance

$$\sigma_X^2 = \mathrm{Var}(X) = E\left[(X-\mu_X)^2\right] = E[X^2] - \mu_X^2. \tag{17.6.16}$$

The last equality in (17.6.16) holds by expanding out the definition in the middle, and applying the properties of expectation.

Let's look at our example where X is the random variable which takes the value $a-2$ with probability p, $a+2$ with probability p and a with probability $1-2p$. In this case $\mu_X = a$, so all we need to compute is $E\left[X^2\right]$. This can readily be done:

$$E\left[X^2\right] = (a-2)^2p + a^2(1-2p) + (a+2)p = a^2 + 8p. \tag{17.6.17}$$

Thus, we see that by (17.6.16) our variance is

$$\sigma_X^2 = \mathrm{Var}(X) = E[X^2] - \mu_X^2 = a^2 + 8p - a^2 = 8p. \tag{17.6.18}$$

This result again makes sense. The largest p can be is $1/2$ which corresponds to picking $a-2$ or $a+2$ with a coin flip. The variance of this being 4 corresponds to the fact that both $a-2$ and $a+2$ are 2 units away from the mean, and $2^2 = 4$. On the other end of the spectrum, if $p = 0$, this random variable always takes the value 0 and so it has no variance at all.

We will list a few properties of variance below:

- For any random variable X, $\mathrm{Var}(X) \geq 0$, with $\mathrm{Var}(X) = 0$ if and only if X is a constant.
- For any random variable X and numbers a and b, we have that $\mathrm{Var}(aX + b) = a^2\mathrm{Var}(X)$.
- If we have two *independent* random variables X and Y, we have $\mathrm{Var}(X + Y) = \mathrm{Var}(X) + \mathrm{Var}(Y)$.

When interpreting these values, there can be a bit of a hiccup. In particular, let's try imagining what happens if we keep track of units through this computation. Suppose that we are working with the star rating assigned to a product on the web page. Then a, $a-2$, and $a+2$ are all measured in units of stars. Similarly, the mean μ_X is then also measured in stars (being a weighted average). However, if we get to the variance, we immediately encounter an issue, which is we want to look at $(X - \mu_X)^2$, which is in units of *squared stars*. This means that the variance itself is not comparable to the original measurements. To make it interpretable, we will need to return to our original units.

Standard Deviations

This summary statistics can always be deduced from the variance by taking the square root! Thus we define the *standard deviation* to be

$$\sigma_X = \sqrt{\mathrm{Var}(X)}. \tag{17.6.19}$$

In our example, this means we now have the standard deviation is $\sigma_X = 2\sqrt{2p}$. If we are dealing with units of stars for our review example, σ_X is again in units of stars.

The properties we had for the variance can be restated for the standard deviation.

- For any random variable X, $\sigma_X \geq 0$.
- For any random variable X and numbers a and b, we have that $\sigma_{aX+b} = |a|\sigma_X$
- If we have two *independent* random variables X and Y, we have $\sigma_{X+Y} = \sqrt{\sigma_X^2 + \sigma_Y^2}$.

It is natural at this moment to ask, "If the standard deviation is in the units of our original random variable, does it represent something we can draw with regards to that random variable?" The answer is a resounding yes! Indeed much like the mean told we the typical location of our random variable, the standard deviation gives the typical range of variation of that random variable. We can make this rigorous with what is known as Chebychev's inequality:

$$P\left(X \not\in [\mu_X - \alpha\sigma_X, \mu_X + \alpha\sigma_X]\right) \leq \frac{1}{\alpha^2}. \tag{17.6.20}$$

Or to state it verbally in the case of $\alpha = 10$, 99% of the samples from any random variable fall within 10 standard deviations of the mean. This gives an immediate interpretation to our standard summary statistics.

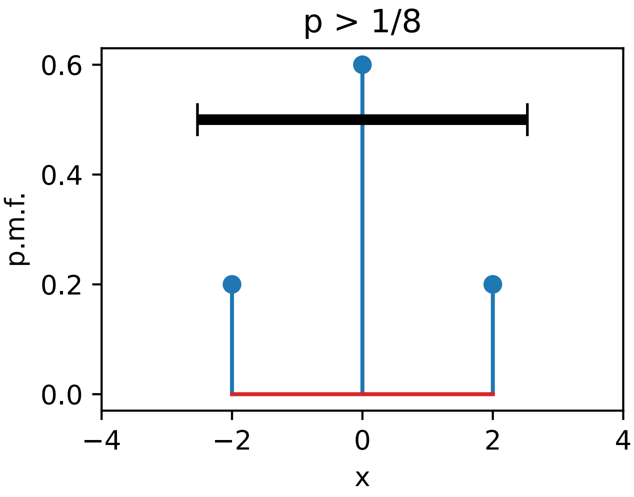
To see how this statement is rather subtle, let's take a look at our running example again where X is the random variable which takes the value $a-2$ with probability p, $a+2$ with probability p and a with probability $1-2p$. We saw that the mean was a and the standard deviation was $2\sqrt{2p}$. This means, if we take Chebychev's inequality (17.6.20) with $\alpha = 2$, we see that the expression is

$$P\left(X \not\in [a - 4\sqrt{2p}, a + 4\sqrt{2p}]\right) \le \frac{1}{4}. \tag{17.6.21}$$

This means that 75% of the time, this random variable will fall within this interval for any value of p. Now, notice that as $p \rightarrow 0$, this interval also converges to the single point a. But we know that our random variable takes the values $a-2, a$, and $a+2$ only so eventually we can be certain $a-2$ and $a+2$ will fall outside the interval! The question is, at what p does that happen. So we want to solve: for what p does $a+4\sqrt{2p} = a+2$, which is solved when $p = 1/8$, which is *exactly* the first p where it could possibly happen without violating our claim that no more than $1/4$ of samples from the distribution would fall outside the interval ($1/8$ to the left, and $1/8$ to the right).

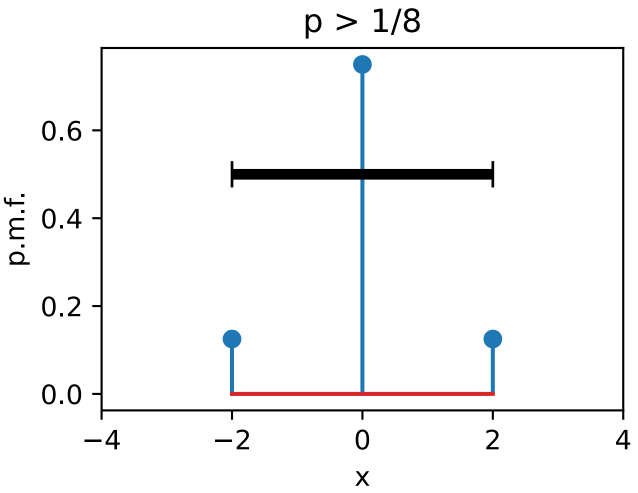
Let's visualize this. We will show the probability of getting the three values as three vertical bars with height proportional to the probability. The interval will be drawn as a horizontal line in the middle. The first plot shows what happens for $p > 1/8$ where the interval safely contains all points.

```python
# Define a helper to plot these figures
def plot_chebychev(a, p):
    d2l.set_figsize()
    d2l.plt.stem([a-2, a, a+2], [p, 1-2*p, p], use_line_collection=True)
    d2l.plt.xlim([-4, 4])
    d2l.plt.xlabel('x')
    d2l.plt.ylabel('p.m.f.')

    d2l.plt.hlines(0.5, a - 4 * np.sqrt(2 * p),
                   a + 4 * np.sqrt(2 * p), 'black', lw=4)
    d2l.plt.vlines(a - 4 * np.sqrt(2 * p), 0.53, 0.47, 'black', lw=1)
    d2l.plt.vlines(a + 4 * np.sqrt(2 * p), 0.53, 0.47, 'black', lw=1)
    d2l.plt.title(”p > 1/8”)

    d2l.plt.show()

# Plot interval when p > 1/8
plot_chebychev(0.0, 0.2)
```



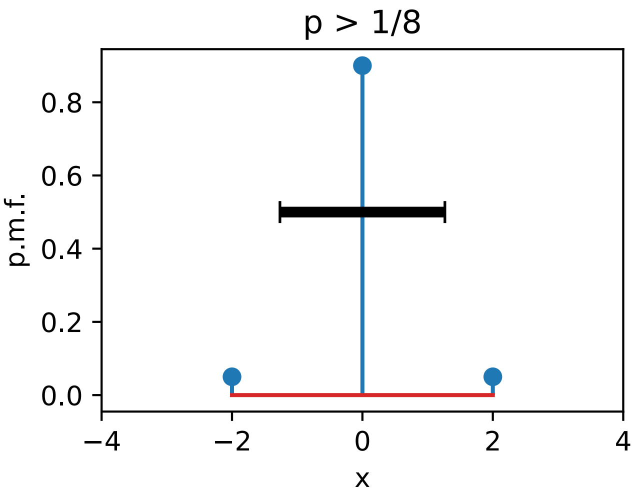
The second shows that at $p = 1/8$, the interval exactly touches the two points. This shows that the inequality is *sharp*, since no smaller interval could be taken while keeping the inequality true.

```
# Plot interval when p = 1/8
plot_chebychev(0.0, 0.125)
```



The third shows that for $p < 1/8$ the interval only contains the center. This does not invalidate the inequality since we only needed to ensure that no more than $1/4$ of the probability falls outside the interval, which means that once $p < 1/8$, the two points at $a - 2$ and $a + 2$ can be discarded.

```
# Plot interval when p < 1/8
plot_chebychev(0.0, 0.05)
```



Means and Variances in the Continuum

This has all been in terms of discrete random variables, but the case of continuous random variables is similar. To intuitively understand how this works, imagine that we split the real number line into intervals of length ϵ given by $(\epsilon i, \epsilon(i+1)]$. Once we do this, our continuous random variable has been made discrete and we can use (17.6.14) say that

$$\begin{aligned} \mu_X & \approx \sum_i (\epsilon i) P(X \in (\epsilon i, \epsilon(i+1)]) \\ & \approx \sum_i (\epsilon i) p_X(\epsilon i) \epsilon, \end{aligned} \tag{17.6.22}$$

where p_X is the density of X. This is an approximation to the integral of $x p_X(x)$, so we can conclude that

$$\mu_X = \int_{-\infty}^\infty x p_X(x) \; dx. \tag{17.6.23}$$

Similarly, using (17.6.16) the variance can be written as

$$\sigma^2_X = E[X^2] - \mu_X^2 = \int_{-\infty}^\infty x^2 p_X(x) \; dx - \left(\int_{-\infty}^\infty x p_X(x) \; dx\right)^2. \tag{17.6.24}$$

Everything stated above about the mean, the variance, and the standard deviation above still apply in this case. For instance, if we consider the random variable with density

$$p(x) = \begin{cases} 1 & x \in [0,1], \\ 0 & \text{otherwise}. \end{cases} \tag{17.6.25}$$

we can compute

$$\mu_X = \int_{-\infty}^\infty x p(x) \; dx = \int_0^1 x \; dx = \frac{1}{2}. \tag{17.6.26}$$

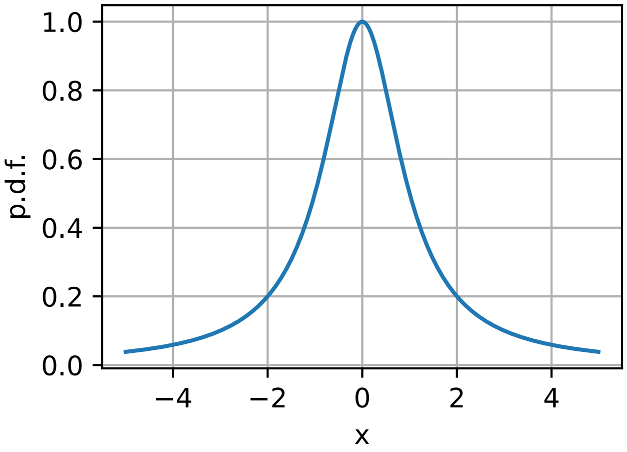
and

$$\sigma_X^2 = \int_{-\infty}^\infty x^2 p(x) \; dx - \left(\frac{1}{2}\right)^2 = \frac{1}{3} - \frac{1}{4} = \frac{1}{12}. \tag{17.6.27}$$

As a warning, let's examine one more example, known as the *Cauchy distribution*. This is the distribution with p.d.f. given by

$$p(x) = \frac{1}{1+x^2}. \tag{17.6.28}$$

```
# Plot the Cauchy distribution p.d.f.
x = np.arange(-5, 5, 0.01)
p = 1 / (1 + x**2)

d2l.plot(x, p, 'x', 'p.d.f.')
```



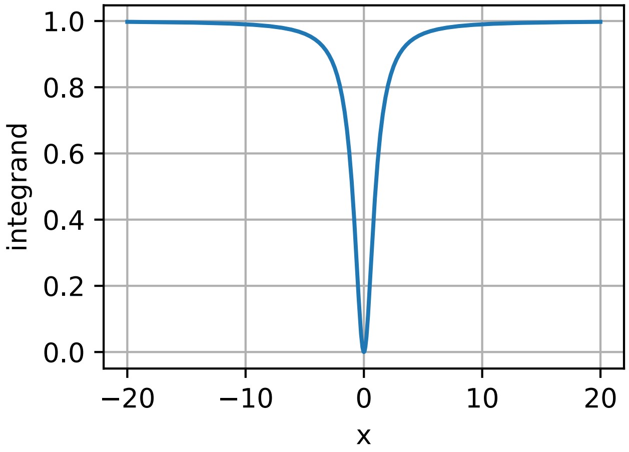
This function looks innocent, and indeed consulting a table of integrals will show it has area one under it, and thus it defines a continuous random variable.

To see what goes astray, let's try to compute the variance of this. This would involve using (17.6.16) computing

$$\int_{-\infty}^{\infty} \frac{x^2}{1+x^2}\,dx. \tag{17.6.29}$$

The function on the inside looks like this:

```
# Plot the integrand needed to compute the variance
x = np.arange(-20, 20, 0.01)
p = x**2 / (1 + x**2)

d2l.plot(x, p, 'x', 'integrand')
```



This function clearly has infinite area under it since it is essentially the constant one with a small dip near zero, and indeed we could show that

$$\int_{-\infty}^{\infty} \frac{x^2}{1+x^2}\,dx = \infty. \tag{17.6.30}$$

This means it does not have a well-defined finite variance.

However, looking deeper shows an even more disturbing result. Let's try to compute the mean using (17.6.14). Using the change of variables formula, we see

$$\mu_X = \int_{-\infty}^{\infty} \frac{x}{1+x^2} \; dx = \frac{1}{2}\int_1^\infty \frac{1}{u} \; du. \tag{17.6.31}$$

The integral inside is the definition of the logarithm, so this is in essence $\log(\infty) = \infty$, so there is no well-defined average value either!

Machine learning scientists define their models so that we most often do not need to deal with these issues, and will in the vast majority of cases deal with random variables with well-defined means and variances. However, every so often random variables with *heavy tails* (that is those random variables where the probabilities of getting large values are large enough to make things like the mean or variance undefined) are helpful in modeling physical systems, thus it is worth knowing that they exist.

Joint Density Functions

The above work all assumes we are working with a single real valued random variable. But what if we are dealing with two or more potentially highly correlated random variables? This circumstance is the norm in machine learning: imagine random variables like $R_{i,j}$ which encode the red value of the pixel at the (i, j) coordinate in an image, or P_t which is a random variable given by a stock price at time t. Nearby pixels tend to have similar color, and nearby times tend to have similar prices. We cannot treat them as separate random variables, and expect to create a successful model (we will see in Section 17.8 a model that under-performs due to such an assumption). We need to develop the mathematical language to handle these correlated continuous random variables.

Thankfully, with the multiple integrals in Section 17.5 we can develop such a language. Suppose that we have, for simplicity, two random variables X, Y which can be correlated. Then, similar to the case of a single variable, we can ask the question:

$$P(X \text{ is in an } \epsilon \text{-sized interval around } x \text{ and } Y \text{ is in an } \epsilon \text{-sized interval around } y). \tag{17.6.32}$$

Similar reasoning to the single variable case shows that this should be approximately

$$P(X \text{ is in an } \epsilon \text{-sized interval around } x \text{ and } Y \text{ is in an } \epsilon \text{-sized interval around } y) \approx \epsilon^{2}p(x, y), \tag{17.6.33}$$

for some function $p(x, y)$. This is referred to as the joint density of X and Y. Similar properties are true for this as we saw in the single variable case. Namely:

- $p(x, y) \geq 0$;
- $\int_{\mathbb{R}^2} p(x, y) \; dx \; dy = 1$;
- $P((X, Y) \in \mathcal{D}) = \int_{\mathcal{D}} p(x, y) \; dx \; dy$.

In this way, we can deal with multiple, potentially correlated random variables. If we wish to work with more than two random variables, we can extend the multivariate density to as many coordinates as desired by considering $p(\mathbf{x}) = p(x_1, \ldots, x_n)$. The same properties of being non-negative, and having total integral of one still hold.

Marginal Distributions

When dealing with multiple variables, we often times want to be able to ignore the relationships and ask, "how is this one variable distributed?" Such a distribution is called a *marginal distribution*.

To be concrete, let's suppose that we have two random variables X, Y with joint density given by $p_{X,Y}(x, y)$. We will be using the subscript to indicate what random variables the density is for. The question of finding the marginal distribution is taking this function, and using it to find $p_X(x)$.

As with most things, it is best to return to the intuitive picture to figure out what should be true. Recall that the density is the function p_X so that

$$P(X \in [x, x+\epsilon]) \approx \epsilon \cdot p_X(x). \tag{17.6.34}$$

There is no mention of Y, but if all we are given is $p_{X,Y}$, we need to include Y somehow. We can first observe that this is the same as

$$P(X \in [x, x+\epsilon], \text{and } Y \in \mathbb{R}) \approx \epsilon \cdot p_X(x). \tag{17.6.35}$$

Our density does not directly tell us about what happens in this case, we need to split into small intervals in y as well, so we can write this as

$$\begin{aligned} \epsilon \cdot p_X(x) &\approx \sum_i P(X \in [x, x+\epsilon], \text{and } Y \in [\epsilon \cdot i, \epsilon \cdot (i+1)]) \\ &\approx \sum_i \epsilon^2 p_{X,Y}(x, \epsilon \cdot i). \end{aligned} \tag{17.6.36}$$

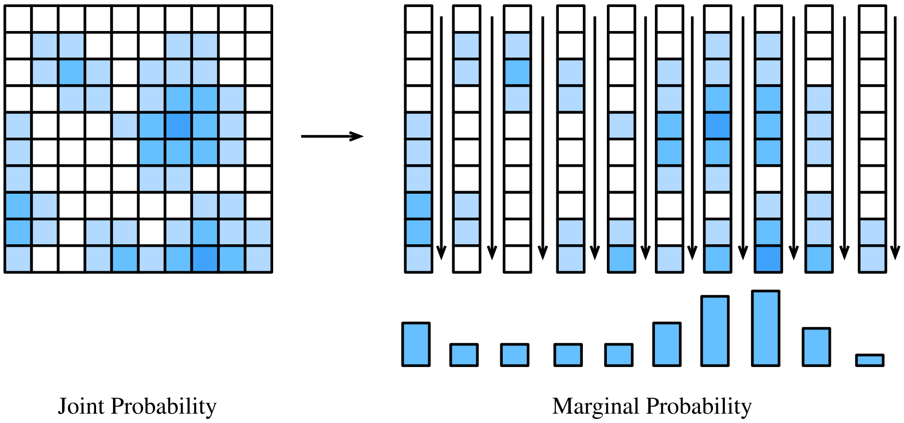


Fig. 17.6.1: By summing along the columns of our array of probabilities, we are able to obtain the marginal distribution for just the random variable represented along the x-axis.

This tells us to add up the value of the density along a series of squares in a line as is show in in Fig. 17.6.1. Indeed, after canceling one factor of epsilon from both sides, and recognizing the sum on the right is the integral over y, we can conclude that

$$\begin{aligned} p_X(x) &\approx \sum_i \epsilon p_{X,Y}(x, \epsilon \cdot i) \\ &\approx \int_{-\infty}^{\infty} p_{X,Y}(x, y) \; dy. \end{aligned} \tag{17.6.37}$$

Thus we see

$$p_X(x) = \int_{-\infty}^{\infty} p_{X,Y}(x, y)\; dy. \tag{17.6.38}$$

This tells us that to get a marginal distribution, we integrate over the variables we do not care about. This process is often referred to as *integrating out* or *marginalized out* the unneeded variables.

Covariance

When dealing with multiple random variables, there is one additional summary statistic which is helpful to know: the *covariance*. This measures the degree that two random variable fluctuate together.

Suppose that we have two random variables X and Y, to begin with, let's suppose they are discrete, taking on values (x_i, y_j) with probability p_{ij}. In this case, the covariance is defined as

$$\sigma_{XY} = \mathrm{Cov}(X, Y) = \sum_{i,j} (x_i - \mu_X)(y_j - \mu_Y)p_{ij}. = E[XY] - E[X]E[Y]. \tag{17.6.39}$$

To think about this intuitively: consider the following pair of random variables. Suppose that X takes the values 1 and 3, and Y takes the values -1 and 3. Suppose that we have the following probabilities

$$\begin{aligned}
P(X = 1 \text{ and } Y = -1) &= \frac{p}{2}, \\
P(X = 1 \text{ and } Y = 3) &= \frac{1-p}{2}, \\
P(X = 3 \text{ and } Y = -1) &= \frac{1-p}{2}, \\
P(X = 3 \text{ and } Y = 3) &= \frac{p}{2},
\end{aligned} \tag{17.6.40}$$

where p is a parameter in $[0, 1]$ we get to pick. Notice that if $p = 1$ then they are both always their minimum or maximum values simultaneously, and if $p = 0$ they are guaranteed to take their flipped values simultaneously (one is large when the other is small and vice versa). If $p = 1/2$, then the four possibilities are all equally likely, and neither should be related. Let's compute the covariance. First, note $\mu_X = 2$ and $\mu_Y = 1$, so we may compute using (17.6.39):

$$\begin{aligned}
\mathrm{Cov}(X, Y) &= \sum_{i,j} (x_i - \mu_X)(y_j - \mu_Y)p_{ij} \\
&= (1-2)(-1-1)\frac{p}{2} + (1-2)(3-1)\frac{1-p}{2} + (3-2)(-1-1)\frac{1-p}{2} + (3-2)(3-1)\frac{p}{2} \\
&= 4p - 2.
\end{aligned} \tag{17.6.41}$$

When $p = 1$ (the case where the are both maximally positive or negative at the same time) has a covariance of 2. When $p = 0$ (the case where they are flipped) the covariance is -2. Finally, when $p = 1/2$ (the case where they are unrelated), the covariance is 0. Thus we see that the covariance measures how these two random variables are related.

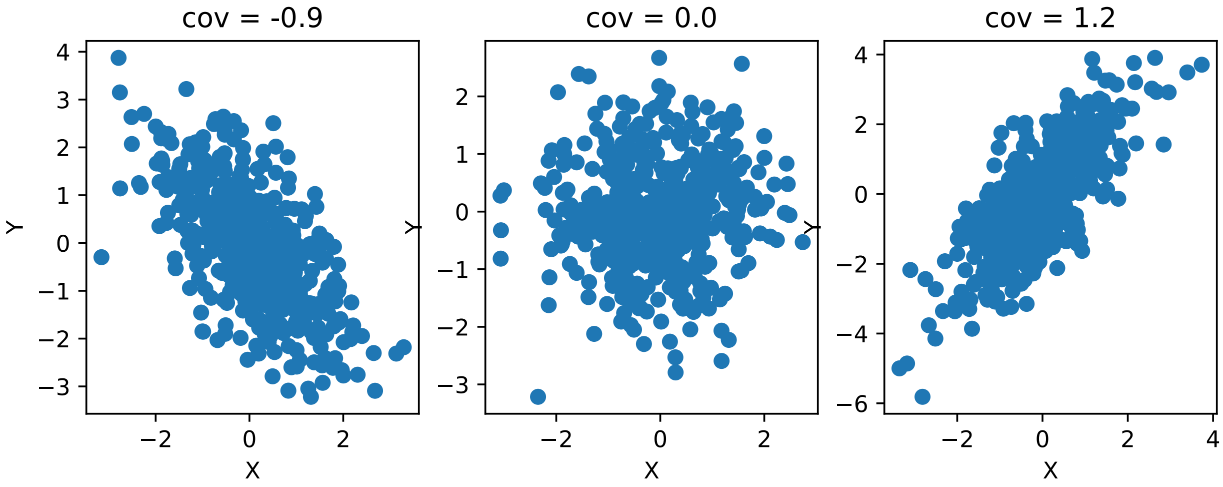
A quick note on the covariance is that it only measures these linear relationships. More complex relationships like $X = Y^2$ where Y is randomly chosen from $\{-2, -1, 0, 1, 2\}$ with equal probability can be missed. Indeed a quick computation shows that these random variables have covariance zero, despite one being a deterministic function of the other.

For continuous random variables, much the same story holds. At this point, we are pretty comfortable with doing the transition between discrete and continuous, so we will provide the continuous analogue of (17.6.39) without any derivation.

$$\sigma_{XY} = \int_{\mathbb{R}^2} (x - \mu_X)(y - \mu_Y)p(x, y) \; dx \; dy. \tag{17.6.42}$$

For visualization, let's take a look at a collection of random variables with tunable covariance.

```
# Plot a few random variables adjustable covariance
covs = [-0.9, 0.0, 1.2]
d2l.plt.figure(figsize=(12, 3))
for i in range(3):
    X = np.random.normal(0, 1, 500)
    Y = covs[i]*X + np.random.normal(0, 1, 500)

    d2l.plt.subplot(1, 4, i+1)
    d2l.plt.scatter(X.asnumpy(), Y.asnumpy())
    d2l.plt.xlabel('X')
    d2l.plt.ylabel('Y')
    d2l.plt.title("cov = {}".format(covs[i]))
d2l.plt.show()
```



Let's see some properties of covariances:

- For any random variable X, $\text{Cov}(X, X) = \text{Var}(X)$.
- For any random variables X, Y and numbers a and b, $\text{Cov}(aX + b, Y) = \text{Cov}(X, aY + b) = a\text{Cov}(X, Y)$.
- If X and Y are independent then $\text{Cov}(X, Y) = 0$.

In addition, we can use the covariance to expand a relationship we saw before. Recall that is X and Y are two independent random variables then

$$\text{Var}(X + Y) = \text{Var}(X) + \text{Var}(Y). \tag{17.6.43}$$

With knowledge of covariances, we can expand this relationship. Indeed, some algebra can show that in general,

$$\text{Var}(X + Y) = \text{Var}(X) + \text{Var}(Y) + 2\text{Cov}(X, Y). \tag{17.6.44}$$

This allows us to generalize the variance summation rule for correlated random variables.

Correlation

As we did in the case of means and variances, let's now consider units. If X is measured in one unit (say inches), and Y is measured in another (say dollars), the covariance is measured in the product of these two units inches × dollars. These units can be hard to interpret. What we will often want in this case is a unit-less measurement of relatedness. Indeed, often we do not care about exact quantitative correlation, but rather ask if the correlation is in the same direction, and how strong the relationship is.

To see what makes sense, let's perform a thought experiment. Suppose that we convert our random variables in inches and dollars to be in inches and cents. In this case the random variable Y is multiplied by 100. If we work through the definition, this means that $\mathrm{Cov}(X, Y)$ will be multiplied by 100. Thus we see that in this case a change of units change the covariance by a factor of 100. Thus, to find our unit-invariant measure of correlation, we will need to divide by something else that also gets scaled by 100. Indeed we have a clear candidate, the standard deviation! Indeed if we define the *correlation coefficient* to be

$$\rho(X, Y) = \frac{\mathrm{Cov}(X, Y)}{\sigma_{X}\sigma_{Y}}, \tag{17.6.45}$$

we see that this is a unit-less value. A little mathematics can show that this number is between -1 and 1 with 1 meaning maximally positively correlated, whereas -1 means maximally negatively correlated.

Returning to our explicit discrete example above, we can see that $\sigma_X = 1$ and $\sigma_Y = 2$, so we can compute the correlation between the two random variables using (17.6.45) to see that

$$\rho(X, Y) = \frac{4p-2}{1 \cdot 2} = 2p-1. \tag{17.6.46}$$

This now ranges between -1 and 1 with the expected behavior of 1 meaning most correlated, and -1 meaning minimally correlated.

As another example, consider X as any random variable, and $Y = aX + b$ as any linear deterministic function of X. Then, one can compute that

$$\sigma_{Y} = \sigma_{aX+b} = |a|\sigma_{X}, \tag{17.6.47}$$

$$\mathrm{Cov}(X, Y) = \mathrm{Cov}(X, aX + b) = a\mathrm{Cov}(X, X) = a\mathrm{Var}(X), \tag{17.6.48}$$

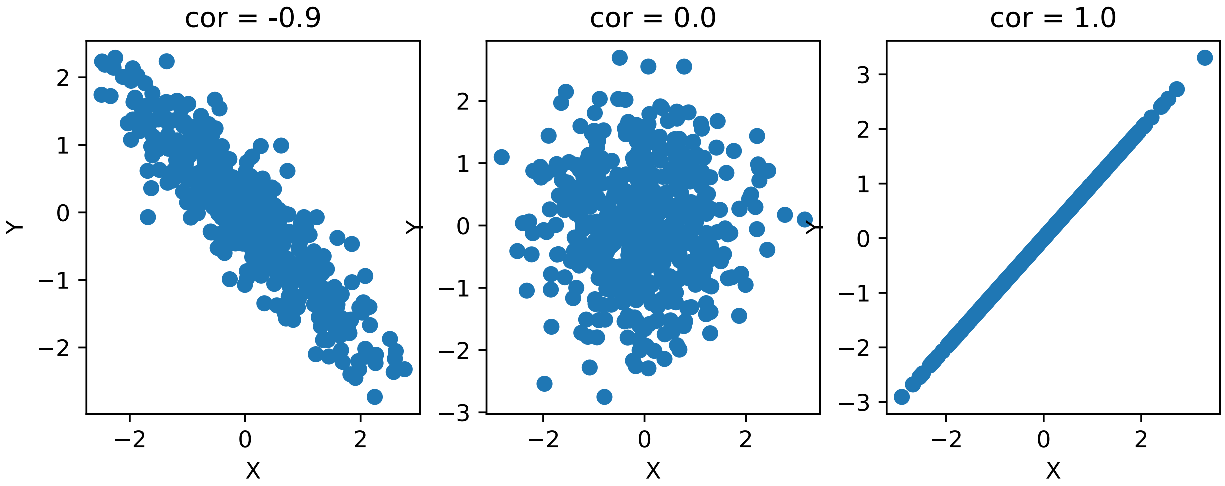
and thus by (17.6.45) that

$$\rho(X, Y) = \frac{a\mathrm{Var}(X)}{|a|\sigma_{X}^2} = \frac{a}{|a|} = \mathrm{sign}(a). \tag{17.6.49}$$

Thus we see that the correlation is $+1$ for any $a > 0$, and -1 for any $a < 0$ illustrating that correlation measures the degree and directionality the two random variables are related, not the scale that the variation takes.

Let's again plot a collection of random variables with tunable correlation.

```
# Plot a few random variables adjustable correlations
cors = [-0.9, 0.0, 1.0]
d2l.plt.figure(figsize=(12, 3))
for i in range(3):
    X = np.random.normal(0, 1, 500)
    Y = cors[i] * X + np.sqrt(1 - cors[i]**2) * np.random.normal(0, 1, 500)

    d2l.plt.subplot(1, 4, i + 1)
    d2l.plt.scatter(X.asnumpy(), Y.asnumpy())
    d2l.plt.xlabel('X')
    d2l.plt.ylabel('Y')
    d2l.plt.title("cor = {}".format(cors[i]))
d2l.plt.show()
```



Let's list a few properties of the correlation below.

- For any random variable X, $\rho(X, X) = 1$.
- For any random variables X, Y and numbers a and b, $\rho(aX + b, Y) = \rho(X, aY + b) = \rho(X, Y)$.
- If X and Y are independent with non-zero variance then $\rho(X, Y) = 0$.

As a final note, you may feel like some of these formulae are familiar. Indeed, if we expand everything out assuming that $\mu_X = \mu_Y = 0$, we see that this is

$$\rho(X, Y) = \frac{\sum_{i,j} x_i y_i p_{ij}}{\sqrt{\sum_{i,j} x_i^2 p_{ij}} \sqrt{\sum_{i,j} y_j^2 p_{ij}}}. \tag{17.6.50}$$

This looks like a sum of a product of terms divided by the square root of sums of terms. This is exactly the formula for the cosine of the angle between two vectors $\mathbf{v}, \mathbf{w}$ with the different coordinates weighted by p_{ij}:

$$\cos(\theta) = \frac{\mathbf{v} \cdot \mathbf{w}}{\|\mathbf{v}\| \|\mathbf{w}\|} = \frac{\sum_i v_i w_i}{\sqrt{\sum_i v_i^2} \sqrt{\sum_i w_i^2}}. \tag{17.6.51}$$

Indeed if we think of norms as being related to standard deviations, and correlations as being cosines of angles, much of the intuition we have from geometry can be applied to thinking about random variables.

Summary

- Continuous random variables are random variables that can take on a continuum of values. They have some technical difficulties that make them more challenging to work with compared to discrete random variables.
- The probability density function allows us to work with continuous random variables by giving a function where the area under the curve on some interval gives the probability of finding a sample point in that interval.
- The cumulative distribution function is the probability of observing the random variable to be less than a given threshold. It can provide a useful alternate viewpoint which unifies discrete and continuous variables.
- The mean is the average value of a random variable.
- The variance is the expected square of the difference between the random variable and its mean.
- The standard deviation is the square root of the variance. It can be thought of as measuring the range of values the random variable may take.
- Chebychev's inequality allows us to make this intuition rigorous by giving an explicit interval that contains the random variable most of the time.
- Joint densities allow us to work with correlated random variables. We may marginalize joint densities by integrating over unwanted random variables to get the distribution of the desired random variable.
- The covariance and correlation coefficient provide a way to measure any linear relationship between two correlated random variables.

Exercises

1. Suppose that we have the random variable with density given by $p(x) = \frac{1}{x^2}$ for $x \geq 1$ and $p(x) = 0$ otherwise. What is $P(X > 2)$?
2. The Laplace distribution is a random variable whose density is given by $p(x = \frac{1}{2}e^{-|x|}$. What is the mean and the standard deviation of this function? As a hint, $\int_0^\infty xe^{-x}\ dx = 1$ and $\int_0^\infty x^2e^{-x}\ dx = 2$.
3. I walk up to you on the street and say "I have a random variable with mean 1, standard deviation 2, and I observed 25% of my samples taking a value larger than 9." Do you believe me? Why or why not?
4. Suppose that you have two random variables X, Y, with joint density given by $p_{XY}(x, y) = 4xy$ for $x, y \in [0, 1]$ and $p_{XY}(x, y) = 0$ otherwise. What is the covariance of X and Y?

17.7 Maximum Likelihood

One of the most commonly encountered way of thinking in machine learning is the maximum likelihood point of view. This is the concept that when working with a probabilistic model with unknown parameters, the parameters which make the data have the highest probability are the most likely ones.

17.7.1 The Maximum Likelihood Principle

This has a Bayesian interpretation which can be helpful to think about. Suppose that we have a model with parameters $\boldsymbol{\theta}$ and a collection of data points X. For concreteness, we can imagine that $\boldsymbol{\theta}$ is a single value representing the probability that a coin comes up heads when flipped, and X is a sequence of independent coin flips. We will look at this example in depth later.

If we want to find the most likely value for the parameters of our model, that means we want to find

$$\operatorname{argmax} P(\boldsymbol{\theta} \mid X). \tag{17.7.1}$$

By Bayes' rule, this is the same thing as

$$\operatorname{argmax} \frac{P(X \mid \boldsymbol{\theta})P(\boldsymbol{\theta})}{P(X)}. \tag{17.7.2}$$

The expression $P(X)$, a parameter agnostic probability of generating the data, does not depend on $\boldsymbol{\theta}$ at all, and so can be dropped without changing the best choice of $\boldsymbol{\theta}$. Similarly, we may now posit that we have no prior assumption on which set of parameters are better than any others, so we may declare that $P(\boldsymbol{\theta})$ does not depend on theta either! This, for instance, makes sense in our coin flipping example where the probability it comes up heads could be any value in $[0, 1]$ without any prior belief it is fair or not (often referred to as an *uninformative prior*). Thus we see that our application of Bayes' rule shows that our best choice of $\boldsymbol{\theta}$ is the maximum likelihood estimate for $\boldsymbol{\theta}$:

$$\hat{\boldsymbol{\theta}} = \underset{\boldsymbol{\theta}}{\operatorname{argmax}} \, P(X \mid \boldsymbol{\theta}). \tag{17.7.3}$$

As a matter of common terminology, the probability of the data given the parameters ($P(X \mid \boldsymbol{\theta})$) is referred to as the *likelihood*.

A Concrete Example

Let's see how this works in a concrete example. Suppose that we have a single parameter θ representing the probability that a coin flip is heads. Then the probability of getting a tails is $1 - \theta$, and so if our observed data X is a sequence with n_H heads and n_T tails, we can use the fact that independent probabilities multiply to see that

$$P(X \mid \theta) = \theta^{n_H}(1 - \theta)^{n_T}. \tag{17.7.4}$$

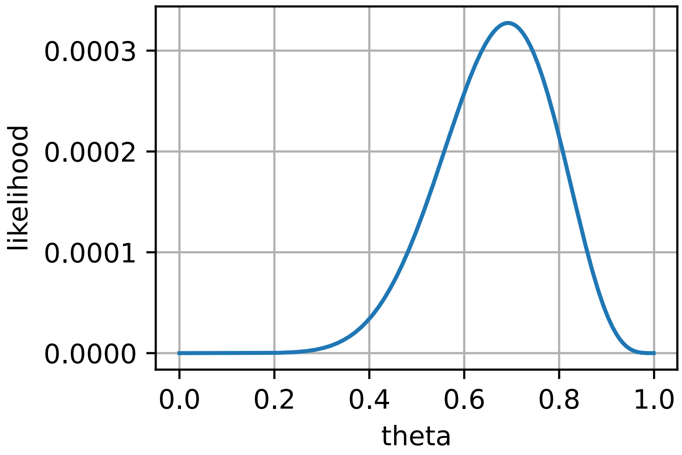
If we flip 13 coins and get the sequence "HHHTHTTHHHHHT", which has $n_H = 9$ and $n_T = 4$, we see that this is

$$P(X \mid \theta) = \theta^9(1 - \theta)^4. \tag{17.7.5}$$

One nice thing about this example will be that we know the answer going in. Indeed, if we said verbally, "I flipped 13 coins, and 9 came up heads, what is our best guess for the probability that the coin comes us heads?," everyone would correctly guess $9/13$. What this maximum likelihood method will give us is a way to get that number from first principals in a way that will generalize to vastly more complex situations.

For our example, the plot of $P(X \mid \theta)$ is as follows:

```
%matplotlib inline
import d2l
from mxnet import autograd, np, npx
npx.set_np()

theta = np.arange(0, 1, 0.001)
p = theta**9 * (1 - theta)**4.

d2l.plot(theta, p, 'theta', 'likelihood')
```



This has its maximum value somewhere near our expected $9/13 \approx 0.7\ldots$. To see if it is exactly there, we can turn to calculus. Notice that at the maximum, the function is flat. Thus, we could find the maximum likelihood estimate (17.7.1) by finding the values of θ where the derivative is zero, and finding the one that gives the highest probability. We compute:

$$\begin{aligned} 0 & = \frac{d}{d\theta} P(X \mid \theta) \\ & = \frac{d}{d\theta} \theta^9(1-\theta)^4 \\ & = 9\theta^8(1-\theta)^4 - 4\theta^9(1-\theta)^3 \\ & = \theta^8(1-\theta)^3(9-13\theta). \end{aligned} \tag{17.7.6}$$

This has three solutions: 0, 1 and $9/13$. The first two are clearly minima, not maxima as they assign probability 0 to our sequence. The final value does *not* assign zero probability to our sequence, and thus must be the maximum likelihood estimate $\hat{\theta} = 9/13$.

17.7.2 Numerical Optimization and the Negative Log-Likelihood

The previous example is nice, but what if we have billions of parameters and data points.

First notice that, if we make the assumption that all the data points are independent, we can no longer practically consider the likelihood itself as it is a product of many probabilities. Indeed, each probability is in $[0, 1]$, say typically of size about $1/2$, and the product of $(1/2)^{1000000000}$ is far below machine precision. We cannot work with that directly.

However, recall that the logarithm turns products to sums, in which case

$$\log((1/2)^{1000000000}) = 1000000000 \cdot \log(1/2) \approx -301029995.6 \ldots \tag{17.7.7}$$

This number fits perfectly within even a single precision 32-bit float. Thus, we should consider the *log-likelihood*, which is

$$\log(P(X \mid \boldsymbol{\theta})). \tag{17.7.8}$$

Since the function $x \mapsto \log(x)$ is increasing, maximizing the likelihood is the same thing as maximizing the log-likelihood. Indeed in Section 17.8 we will see this reasoning applied when working with the specific example of the naive Bayes classifier.

We often work with loss functions, where we wish to minimize the loss. We may turn maximum likelihood into the minimization of a loss by taking $-\log(P(X \mid \boldsymbol{\theta}))$, which is the *negative log-likelihood*.

To illustrate this, consider the coin flipping problem from before, and pretend that we do not know the closed form solution. The we may compute that

$$-\log(P(X \mid \boldsymbol{\theta})) = -\log(\theta^{n_H}(1-\theta)^{n_T}) = -(n_H \log(\theta) + n_T \log(1-\theta)). \tag{17.7.9}$$

This can be written into code, and freely optimized even for billions of coin flips.

```
# Set up our data
n_H = 8675309
n_T = 25624

# Initialize our paramteres
theta = np.array(0.5)
theta.attach_grad()

# Perform gradient descent
lr = 0.00000000001
for iter in range(10):
    with autograd.record():
        loss = -(n_H * np.log(theta) + n_T * np.log(1 - theta))
    loss.backward()
    theta -= lr * theta.grad

# Check output
theta, n_H / (n_H + n_T)
```

```
(array(0.50172704), 0.9970550284664874)
```

Numerical convenience is only one reason people like to use negative log-likelihoods. Indeed, there are a several reasons that it can be preferable.

The second reason we consider the log-likelihood is the simplified application of calculus rules. As discussed above, due to independence assumptions, most probabilities we encounter in machine learning are products of individual probabilities.

$$P(X \mid \boldsymbol{\theta}) = p(x_1 \mid \boldsymbol{\theta}) \cdot p(x_2 \mid \boldsymbol{\theta}) \cdots p(x_n \mid \boldsymbol{\theta}). \tag{17.7.10}$$

This means that if we directly apply the product rule to compute a derivative we get

$$\begin{aligned} \frac{\partial}{\partial \boldsymbol{\theta}} P(X \mid \boldsymbol{\theta}) & = \left(\frac{\partial}{\partial \boldsymbol{\theta}}P(x_1\mid\boldsymbol{\theta})\right)\cdot P(x_2\mid\boldsymbol{\theta})\cdots P(x_n\mid\boldsymbol{\theta}) \\ & \quad + P(x_1\mid\boldsymbol{\theta})\cdot \left(\frac{\partial}{\partial \boldsymbol{\theta}}P(x_2\mid\boldsymbol{\theta})\right)\cdots P(x_n\mid\boldsymbol{\theta}) \\ & \quad\quad \vdots \\ & \quad + P(x_1\mid\boldsymbol{\theta})\cdot P(x_2\mid\boldsymbol{\theta}) \cdots \left(\frac{\partial}{\partial \boldsymbol{\theta}}P(x_n\mid\boldsymbol{\theta})\right). \end{aligned} \tag{17.7.11}$$

This requires $n(n-1)$ multiplications, along with $(n-1)$ additions, so it is total of quadratic time in the inputs! Sufficient cleverness in grouping terms will reduce this to linear time, but it requires some thought. For the negative log-likelihood we have instead

$$-\log\left(P(X\mid\boldsymbol{\theta})\right) = -\log(P(x_1\mid\boldsymbol{\theta})) - \log(P(x_2\mid\boldsymbol{\theta})) \cdots - \log(P(x_n\mid\boldsymbol{\theta})), \tag{17.7.12}$$

which then gives

$$- \frac{\partial}{\partial \boldsymbol{\theta}} \log\left(P(X\mid\boldsymbol{\theta})\right) = \frac{1}{P(x_1\mid\boldsymbol{\theta})}\left(\frac{\partial}{\partial \boldsymbol{\theta}}P(x_1\mid\boldsymbol{\theta})\right) + \cdots + \frac{1}{P(x_n\mid\boldsymbol{\theta})}\left(\frac{\partial}{\partial \boldsymbol{\theta}}P(x_n\mid\boldsymbol{\theta})\right). \tag{17.7.13}$$

This requires only n divides and $n-1$ sums, and thus is linear time in the inputs.

The third and final reason to consider the negative log-likelihood is the relationship to information theory, which we will discuss in detail in Section 17.10. This is a rigorous mathematical theory which gives a way to measure the degree of information or randomness in a random variable. The key object of study in that field is the entropy which is

$$H(p) = -\sum_i p_i \log_2(p_i), \tag{17.7.14}$$

which measures the randomness of a source. Notice that this is nothing more than the average $-\log$ probability, and thus if we take our negative log-likelihood and divide by the number of data points, we get a relative of entropy known as cross-entropy. This theoretical interpretation alone would be sufficiently compelling to motivate reporting the average negative log-likelihood over the dataset as a way of measuring model performance.

17.7.3 Maximum Likelihood for Continuous Variables

Everything that we have done so far assumes we are working with discrete random variables, but what if we want to work with continuous ones?

The short summary is that nothing at all changes, except we replace all the instances of the probability with the probability density. Recalling that we write densities with lower case p, this means that for example we now say

$$-\log\left(p(X \mid \boldsymbol{\theta})\right) = -\log(p(x_1 \mid \boldsymbol{\theta})) - \log(p(x_2 \mid \boldsymbol{\theta})) \cdots - \log(p(x_n \mid \boldsymbol{\theta})) = -\sum_i \log(p(x_i \mid \theta)). \tag{17.7.15}$$

The question becomes, "Why is this OK?" After all, the reason we introduced densities was because probabilities of getting specific outcomes themselves was zero, and thus is not the probability of generating our data for any set of parameters zero?

Indeed, this is the case, and understanding why we can shift to densities is an exercise in tracing what happens to the epsilons.

Let's first re-define our goal. Suppose that for continuous random variables we no longer want to compute the probability of getting exactly the right value, but instead matching to within some range ϵ. For simplicity, we assume our data is repeated observations $x_1, \ldots, x_N$ of identically distributed random variables $X_1, \ldots, X_N$. As we have seen previously, this can be written as

$$\begin{aligned}&P(X_1 \in [x_1, x_1+\epsilon], X_2 \in [x_2, x_2+\epsilon], \ldots, X_N \in [x_N, x_N+\epsilon] \mid \boldsymbol{\theta}) \\ \approx&\epsilon^N p(x_1 \mid \boldsymbol{\theta}) \cdot p(x_2 \mid \boldsymbol{\theta}) \cdots p(x_n \mid \boldsymbol{\theta}).\end{aligned} \tag{17.7.16}$$

Thus, if we take negative logarithms of this we obtain

$$\begin{aligned}&-\log(P(X_1 \in [x_1, x_1+\epsilon], X_2 \in [x_2, x_2+\epsilon], \ldots, X_N \in [x_N, x_N+\epsilon] \mid \boldsymbol{\theta})) \\ \approx&-N\log(\epsilon) - \sum_i \log(p(x_i \mid \boldsymbol{\theta})).\end{aligned} \tag{17.7.17}$$

If we examine this expression, the only place that the ϵ occurs is in the additive constant $-N\log(\epsilon)$. This does not depend on the parameters $\boldsymbol{\theta}$ at all, so the optimal choice of $\boldsymbol{\theta}$ does not depend on our choice of ϵ! If we demand four digits or four-hundred, the best choice of $\boldsymbol{\theta}$ remains the same, thus we may freely drop the epsilon to see that what we want to optimize is

$$-\sum_i \log(p(x_i \mid \boldsymbol{\theta})). \tag{17.7.18}$$

Thus, we see that the maximum likelihood point of view can operate with continuous random variables as easily as with discrete ones by replacing the probabilities with probability densities.

Summary

- The maximum likelihood principle tells us that the best fit model for a given dataset is the one that generates the data with the highest probability.
- Often people work with the negative log-likelihood instead for a variety of reasons: numerical stability, conversion of products to sums (and the resulting simplification of gradient computations), and theoretical ties to information theory.
- While simplest to motivate in the discrete setting, it may be freely generalized to the continuous setting as well by maximizing the probability density assigned to the datapoints.

Exercises

1. Suppose that you know that a random variable has density $\frac{1}{\alpha}e^{-\alpha x}$ for some value α. You obtain a single observation from the random variable which is the number 3. What is the maximum likelihood estimate for α?
2. Suppose that you have a dataset of samples $\{x_i\}_{i=1}^N$ drawn from a Gaussian with unknown mean, but variance 1. What is the maximum likelihood estimate for the mean?

17.8 Naive Bayes

Throughout the previous sections, we learned about the theory of probability and random variables. To put this theory to work, let's introduce the *naive Bayes* classifier. This uses nothing but probabilistic fundamentals to allow us to perform classification of digits.

Learning is all about making assumptions. If we want to classify a new data point that we have never seen before we have to make some assumptions about which data points are similar to each other. The naive Bayes classifier, a popular and remarkably clear algorithm, assumes all features are independent from each other to simplify the computation. In this section, we will apply this model to recognize characters in images.

```
%matplotlib inline
import d2l
import math
from mxnet import gluon, np, npx
npx.set_np()
d2l.use_svg_display()
```

17.8.1 Optical Character Recognition

MNIST (LeCun et al., 1998) is one of widely used datasets. It contains 60,000 images for training and 10,000 images for validation. Each image contains a handwritten digit from 0 to 9. The task is classifying each image into the corresponding digit.

Gluon provides a `MNIST` class in the `data.vision` module to automatically retrieve the dataset from the internet. Subsequently, Gluon will use the already-downloaded local copy. We specify whether we are requesting the training set or the test set by setting the value of the parameter `train` to `True` or `False`, respectively. Each image is a grayscale image with both width and height of 28 with shape (28,28,1). We use a customized transformation to remove the last channel dimension. In addition, the dataset represents each pixel by a unsigned 8-bit integer. We quantize them into binary features to simplify the problem.

```
def transform(data, label):
    return np.floor(data.astype('float32') / 128).squeeze(axis=-1), label
```

(continues on next page)

(continued from previous page)

```
mnist_train = gluon.data.vision.MNIST(train=True, transform=transform)
mnist_test = gluon.data.vision.MNIST(train=False, transform=transform)
```

We can access a particular example, which contains the image and the corresponding label.

```
image, label = mnist_train[2]
image.shape, label
```

```
((28, 28), array(4, dtype=int32))
```

Our example, stored here in the variable `image`, corresponds to an image with a height and width of 28 pixels.

```
image.shape, image.dtype
```

```
((28, 28), dtype('float32'))
```

Our code stores the label of each image as a scalar. Its type is a 32-bit integer.

```
label, type(label), label.dtype
```

```
(array(4, dtype=int32), mxnet.numpy.ndarray, dtype('int32'))
```

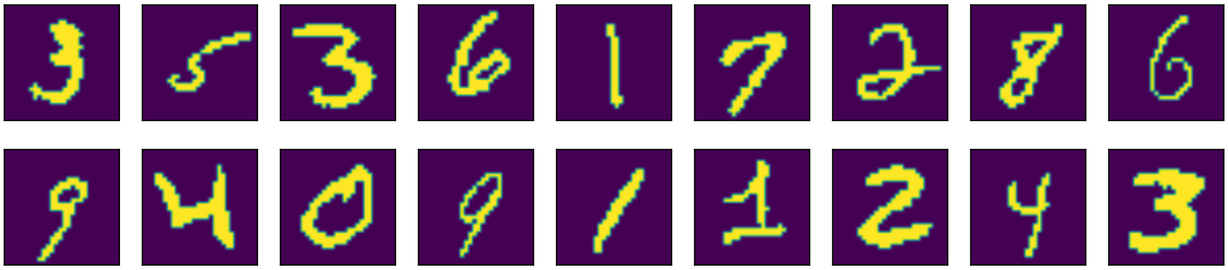
We can also access multiple examples at the same time.

```
images, labels = mnist_train[10:38]
images.shape, labels.shape
```

```
((28, 28, 28), (28,))
```

Let's visualize these examples.

```
d2l.show_images(images, 2, 9);
```



17.8.2 The Probabilistic Model for Classification

In a classification task, we map an example into a category. Here an example is a grayscale 28×28 image, and a category is a digit. (Refer to Section 3.4 for a more detailed explanation.) One natural way to express the classification task is via the probabilistic question: what is the most likely label given the features (i.e., image pixels)? Denote by $\mathbf{x} \in \mathbb{R}^d$ the features of the example and $y \in \mathbb{R}$ the label. Here features are image pixels, where we can reshape a 2-dimensional image to a vector so that $d = 28^2 = 784$, and labels are digits. The probability of the label given the features is $p(y \mid \mathbf{x})$. If we are able to compute these probabilities, which are $p(y \mid \mathbf{x})$ for $y = 0, \ldots, 9$ in our example, then the classifier will output the prediction $\hat{y}$ given by the expression:

$$\hat{y} = \mathrm{argmax} \, p(y \mid \mathbf{x}). \tag{17.8.1}$$

Unfortunately, this requires that we estimate $p(y \mid \mathbf{x})$ for every value of $\mathbf{x} = x_1, ..., x_d$. Imagine that each feature could take one of 2 values. For example, the feature $x_1 = 1$ might signify that the word apple appears in a given document and $x_1 = 0$ would signify that it does not. If we had 30 such binary features, that would mean that we need to be prepared to classify any of 2^{30} (over 1 billion!) possible values of the input vector $\mathbf{x}$.

Moreover, where is the learning? If we need to see every single possible example in order to predict the corresponding label then we are not really learning a pattern but just memorizing the dataset.

17.8.3 The Naive Bayes Classifier

Fortunately, by making some assumptions about conditional independence, we can introduce some inductive bias and build a model capable of generalizing from a comparatively modest selection of training examples. To begin, let's use Bayes theorem, to express the classifier as

$$\hat{y} = \mathrm{argmax}_y \, p(y \mid \mathbf{x}) = \mathrm{argmax}_y \, \frac{p(\mathbf{x} \mid y) p(y)}{p(\mathbf{x})}. \tag{17.8.2}$$

Note that the denominator is the normalizing term $p(\mathbf{x})$ which does not depend on the value of the label y. As a result, we only need to worry about comparing the numerator across different values of y. Even if calculating the denominator turned out to be intractable, we could get away with ignoring it, so long as we could evaluate the numerator. Fortunately, even if we wanted to recover the normalizing constant, we could. We can always recover the normalization term since $\sum_y p(y \mid \mathbf{x}) = 1$.

Now, let's focus on $p(\mathbf{x} \mid y)$. Using the chain rule of probability, we can express the term $p(\mathbf{x} \mid y)$ as

$$p(x_1 \mid y) \cdot p(x_2 \mid x_1, y) \cdot ... \cdot p(x_d \mid x_1, ..., x_{d-1}, y). \tag{17.8.3}$$

By itself, this expression does not get us any further. We still must estimate roughly 2^d parameters. However, if we assume that *the features are conditionally independent of each other, given the label*, then suddenly we are in much better shape, as this term simplifies to $\prod_i p(x_i \mid y)$, giving us the predictor

$$\hat{y} = \mathrm{argmax}_y \prod_{i=1}^{d} p(x_i \mid y) p(y). \tag{17.8.4}$$

If we can estimate $\prod_i p(x_i = 1 \mid y)$ for every i and y, and save its value in $P_{xy}[i, y]$, here P_{xy} is a $d \times n$ matrix with n being the number of classes and $y \in \{1, \ldots, n\}$. In addition, we estimate $p(y)$

for every y and save it in $P_y[y]$, with P_y a n-length vector. Then for any new example $\mathbf{x}$, we could compute

$$\hat{y} = \mathrm{argmax}_y \prod_{i=1}^{d} P_{xy}[x_i, y]P_y[y], \tag{17.8.5}$$

for any y. So our assumption of conditional independence has taken the complexity of our model from an exponential dependence on the number of features $\mathcal{O}(2^dn)$ to a linear dependence, which is $\mathcal{O}(dn)$.

17.8.4 Training

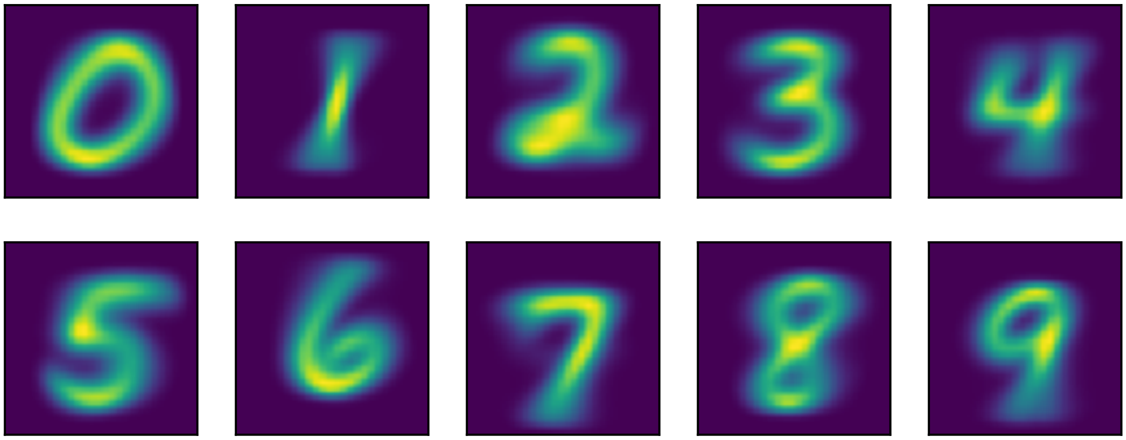
The problem now is that we do not know P_{xy} and P_y. So we need to estimate their values given some training data first. This is *training* the model. Estimating P_y is not too hard. Since we are only dealing with 10 classes, we may count the number of occurrences n_y for each of the digits and divide it by the total amount of data n. For instance, if digit 8 occurs $n_8 = 5,800$ times and we have a total of $n = 60,000$ images, the probability estimate is $p(y = 8) = 0.0967$.

```
X, Y = mnist_train[:]  # All training examples

n_y = np.zeros((10))
for y in range(10):
    n_y[y] = (Y == y).sum()
P_y = n_y / n_y.sum()
P_y
```

```
array([0.09871667, 0.11236667, 0.0993    , 0.10218333, 0.09736667,
       0.09035   , 0.09863333, 0.10441667, 0.09751666, 0.09915   ])
```

Now on to slightly more difficult things P_{xy}. Since we picked black and white images, $p(x_i \mid y)$ denotes the probability that pixel i is switched on for class y. Just like before we can go and count the number of times n_{iy} such that an event occurs and divide it by the total number of occurrences of y, i.e., n_y. But there is something slightly troubling: certain pixels may never be black (e.g., for well cropped images the corner pixels might always be white). A convenient way for statisticians to deal with this problem is to add pseudo counts to all occurrences. Hence, rather than n_{iy} we use $n_{iy} + 1$ and instead of n_y we use $n_y + 1$. This is also called *Laplace Smoothing*. It may seem ad-hoc, however it may be well motivated from a Bayesian point-of-view.

```
n_x = np.zeros((10, 28, 28))
for y in range(10):
    n_x[y] = np.array(X.asnumpy()[Y.asnumpy() == y].sum(axis=0))
P_xy = (n_x + 1) / (n_y + 1).reshape(10, 1, 1)

d2l.show_images(P_xy, 2, 5);
```



By visualizing these $10 \times 28 \times 28$ probabilities (for each pixel for each class) we could get some mean looking digits.

Now we can use (17.8.5) to predict a new image. Given $\mathbf{x}$, the following functions computes $p(\mathbf{x} \mid y)p(y)$ for every y.

```
def bayes_pred(x):
    x = np.expand_dims(x, axis=0)  # (28, 28) -> (1, 28, 28)
    p_xy = P_xy * x + (1 - P_xy)*(1 - x)
    p_xy = p_xy.reshape(10, -1).prod(axis=1)  # p(x|y)
    return np.array(p_xy) * P_y

image, label = mnist_test[0]
bayes_pred(image)
```

```
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

This went horribly wrong! To find out why, let's look at the per pixel probabilities. They are typically numbers between 0.001 and 1. We are multiplying 784 of them. At this point it is worth mentioning that we are calculating these numbers on a computer, hence with a fixed range for the exponent. What happens is that we experience *numerical underflow*, i.e., multiplying all the small numbers leads to something even smaller until it is rounded down to zero. We discussed this as a theoretical issue in Section 17.7, but we see the phenomena clearly here in practice.

As discussed in that section, we fix this by use the fact that $\log ab = \log a + \log b$, i.e., we switch to summing logarithms. Even if both a and b are small numbers, the logarithm values should be in a proper range.

```
a = 0.1
print('underflow:', a**784)
print('logarithm is normal:', 784*math.log(a))
```

```
underflow: 0.0
logarithm is normal: -1805.2267129073316
```

Since the logarithm is an increasing function, we can rewrite (17.8.5) as

$$\hat{y} = \operatorname{argmax}_y \sum_{i=1}^{d} \log P_{xy}[x_i, y] + \log P_y[y]. \tag{17.8.6}$$

We can implement the following stable version:

```
log_P_xy = np.log(P_xy)
log_P_xy_neg = np.log(1 - P_xy)
log_P_y = np.log(P_y)

def bayes_pred_stable(x):
    x = np.expand_dims(x, axis=0)  # (28, 28) -> (1, 28, 28)
    p_xy = log_P_xy * x + log_P_xy_neg * (1 - x)
    p_xy = p_xy.reshape(10, -1).sum(axis=1)  # p(x|y)
    return p_xy + log_P_y

py = bayes_pred_stable(image)
py
```

```
array([-269.00424, -301.73447, -245.21458, -218.8941 , -193.46907,
       -206.10315, -292.54315, -114.62834, -220.35619, -163.18881])
```

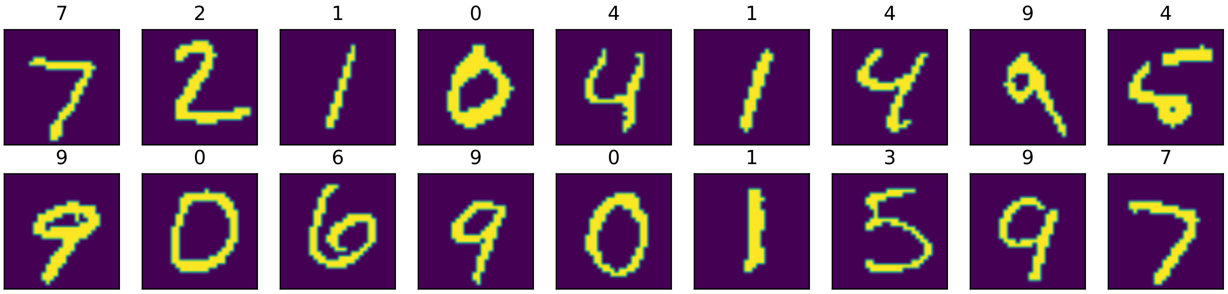
We may now check if the prediction is correct.

```
# Convert label which is a scalar tensor of int32 dtype
# to a Python scalar integer for comparison
py.argmax(axis=0) == int(label)
```

```
array(True)
```

If we now predict a few validation examples, we can see the Bayes classifier works pretty well.

```
def predict(X):
    return [bayes_pred_stable(x).argmax(axis=0).astype(np.int32) for x in X]

X, y = mnist_test[:18]
preds = predict(X)
d2l.show_images(X, 2, 9, titles=[str(d) for d in preds]);
```



Finally, let's compute the overall accuracy of the classifier.

```
X, y = mnist_test[:]
preds = np.array(predict(X), dtype=np.int32)
float((preds == y).sum()) / len(y)  # Validation accuracy
```

```
0.8426
```

Modern deep networks achieve error rates of less than 0.01. The relatively poor performance is due to the incorrect statistical assumptions that we made in our model: we assumed that each and every pixel are *independently* generated, depending only on the label. This is clearly not how humans write digits, and this wrong assumption led to the downfall of our overly naive (Bayes) classifier.

Summary

- Using Bayes' rule, a classifier can be made by assuming all observed features are independent.
- This classifier can be trained on a dataset by counting the number of occurrences of combinations of labels and pixel values.
- This classifier was the gold standard for decades for tasks such as spam detection.

Exercises

1. Consider the dataset $[[0, 0], [0, 1], [1, 0], [1, 1]]$ with labels given by the XOR of the two elements $[0, 1, 1, 0]$. What are the probabilities for a Naive Bayes classifier built on this dataset. Does it successfully classify our points? If not, what assumptions are violated?
2. Suppose that we did not use Laplace smoothing when estimating probabilities and a data point arrived at testing time which contained a value never observed in training. What would the model output?
3. The naive Bayes classifier is a specific example of a Bayesian network, where the dependence of random variables are encoded with a graph structure. While the full theory is beyond the scope of this section (see (Koller & Friedman, 2009) for full details), explain why allowing explicit dependence between the two input variables in the XOR model allows for the creation of a successful classifier.

17.9 Statistics

Undoubtedly, to be a top deep learning practitioner, the ability to train the state-of-the-art and high accurate models is crucial. However, it is often unclear when improvements are significant, or only the result of random fluctuations in the training process. To be able to discuss uncertainty in estimated values, we must learn some statistics.

The earliest reference of *statistics* can be traced back to an Arab scholar Al-Kindi in the 9^{th}-century, who gave a detailed description of how to use statistics and frequency analysis to decipher encrypted messages. After 800 years, the modern statistics arose from Germany in 1700s, when the researchers focused on the demographic and economic data collection and analysis. Today, statistics is the science subject that concerns the collection, processing, analysis, interpretation and visualization of data. What is more, the core theory of statistics has been widely used in the research within academia, industry, and government.

More specifically, statistics can be divided to *descriptive statistics* and *statistical inference*. The former focus on summarizing and illustrating the features of a collection of observed data, which is referred to as a *sample*. The sample is drawn from a *population*, denotes the total set of similar individuals, items, or events of our experiment interests. Contrary to descriptive statistics, *statistical inference* further deduces the characteristics of a population from the given *samples*, based on the assumptions that the sample distribution can replicate the population distribution at some degree.

You may wonder: "What is the essential difference between machine learning and statistics?" Fundamentally speaking, statistics focuses on the inference problem. This type of problems includes modeling the relationship between the variables, such as causal inference, and testing the statistically significance of model parameters, such as A/B testing. In contrast, machine learning emphasizes on making accurate predictions, without explicitly programming and understanding each parameter's functionality.

In this section, we will introduce three types of statistics inference methods: evaluating and comparing estimators, conducting hypothesis tests, and constructing confidence intervals. These methods can help us infer the characteristics of a given population, i.e., the true parameter θ. For brevity, we assume that the true parameter θ of a given population is a scalar value. It is straightforward to extend to the case where θ is a vector or a tensor, thus we omit it in our discussion.

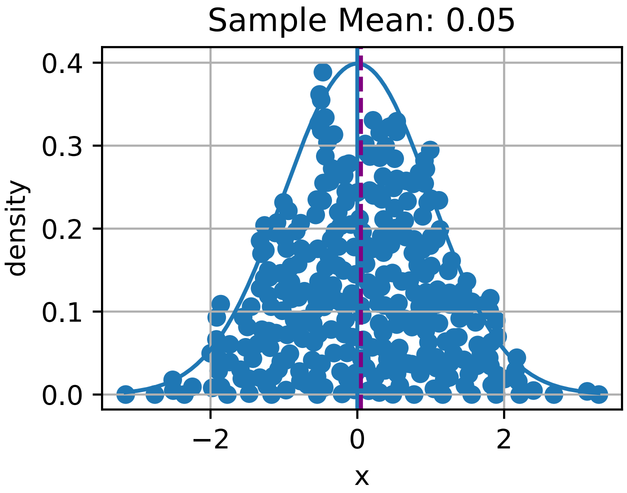
17.9.1 Evaluating and Comparing Estimators

In statistics, an *estimator* is a function of given samples used to estimate the true parameter θ. We will write $\hat{\theta}_n = \hat{f}(x_1, \ldots, x_n)$ for the estimate of θ after observing the samples $\{x_1, x_2, \ldots, x_n\}$.

We've seen simple examples of estimators before in section Section 17.7. If you have a number of samples from a Bernoulli random variable, then the maximum likelihood estimate for the probability the random variable is one can be obtained by counting the number of ones observed and dividing by the total number of samples. Similarly, an exercise asked you to show that the maximum likelihood estimate of the mean of a Gaussian given a number of samples is given by the average value of all the samples. These estimators will almost never give the true value of the parameter, but ideally for a large number of samples the estimate will be close.

As an example, we show below the true density of a Gaussian random variable with mean zero and variance one, along with a collection samples from that Gaussian. We constructed the y coordinate so every point is visible and the relationship to the original density is clearer.

```
import d2l
from mxnet import np, npx
import random
npx.set_np()

# Sample datapoints and create y coordinate
epsilon = 0.1
random.seed(8675309)
xs = np.random.normal(loc=0, scale=1, size=(300,))

ys = [np.sum(np.exp(-(xs[0:i] - xs[i])**2 / (2 * epsilon**2))
             / np.sqrt(2*np.pi*epsilon**2)) / len(xs) for i in range(len(xs))]

# Compute true density
xd = np.arange(np.min(xs), np.max(xs), 0.01)
yd = np.exp(-xd**2/2) / np.sqrt(2 * np.pi)

# Plot the results
d2l.plot(xd, yd, 'x', 'density')
d2l.plt.scatter(xs, ys)
d2l.plt.axvline(x=0)
d2l.plt.axvline(x=np.mean(xs), linestyle='--', color='purple')
d2l.plt.title("Sample Mean: {:.2f}".format(float(np.mean(xs))))
d2l.plt.show()
```



There can be many ways to compute an estimator of a parameter $\hat{\theta}_n$. In this section, we introduce three common methods to evaluate and compare estimators: the mean squared error, the standard deviation, and statistical bias.

Mean Squared Error

Perhaps the simplest metric used to evaluate estimators is the *mean squared error (MSE)* (or *:math:`l_2` loss*) of an estimator can be defined as

$$\text{MSE}(\hat{\theta}_n, \theta) = E[(\hat{\theta}_n - \theta)^2]. \tag{17.9.1}$$

This allows us to quantify the average squared deviation from the true value. MSE is always non-negative. If you have read Section 3.1, you will recognize it as the most commonly used regression loss function. As a measure to evaluate an estimator, the closer its value to zero, the closer the estimator is close to the true parameter θ.

Statistical Bias

The MSE provides a natural metric, but we can easily imagine multiple different phenomena that might make it large. Two that we will see are fundamentally important are the fluctuation in the estimator due to randomness in the dataset, and systematic error in the estimator due to the estimation procedure.

First, let's measure the systematic error. For an estimator $\hat{\theta}_n$, the mathematical illustration of *statistical bias* can be defined as

$$\text{bias}(\hat{\theta}_n) = E(\hat{\theta}_n - \theta) = E(\hat{\theta}_n) - \theta. \tag{17.9.2}$$

Note that when $\text{bias}(\hat{\theta}_n) = 0$, the expectation of the estimator $\hat{\theta}_n$ is equal to the true value of parameter. In this case, we say $\hat{\theta}_n$ is an unbiased estimator. In general, an unbiased estimator is better than a biased estimator since its expectation is the same as the true parameter.

It is worth being aware, however, that biased estimators are frequently used in practice. There are cases where unbiased estimators do not exist without further assumptions, or are intractable to compute. This may seem like a significant flaw in an estimator, however the majority of estimators encountered in practice are at least asymptotically unbiased in the sense that the bias tends to zero as the number of available samples tends to infinity: $\lim_{n \rightarrow \infty} \text{bias}(\hat{\theta}_n) = 0$.

Variance and Standard Deviation

Second, let's measure the randomness in the estimator. Recall from Section 17.6, the *standard deviation* (or *standard error*) is defined as the squared root of the variance. We may measure the degree of fluctuation of an estimator by measuring the standard deviation or variance of that estimator.

$$\sigma_{\hat{\theta}_n} = \sqrt{\text{Var}(\hat{\theta}_n)} = \sqrt{E[(\hat{\theta}_n - E(\hat{\theta}_n))^2]}. \tag{17.9.3}$$

It is important to compare (17.9.3) to (17.9.1). In this equation we do not compare to the true population value θ, but instead to $E(\hat{\theta}_n)$, the expected sample mean. Thus we are not measuring how far the estimator tends to be from the true value, but instead we measuring the fluctuation of the estimator itself.

The Bias-Variance Trade-off

It is intuitively clear that these two components contribute to the mean squared error. What is somewhat shocking is that we can show that this is actually a *decomposition* of the mean squared error into two contributions. That is to say that we can write the mean squared error as the sum of the variance and the square or the bias.

$$\begin{aligned}\mathrm{MSE}(\hat{\theta}_n, \theta) &= E[(\hat{\theta}_n - E(\hat{\theta}_n) + E(\hat{\theta}_n) - \theta)^2] \\ &= E[(\hat{\theta}_n - E(\hat{\theta}_n))^2] + E[(E(\hat{\theta}_n) - \theta)^2] \\ &= \mathrm{Var}(\hat{\theta}_n) + [\mathrm{bias}(\hat{\theta}_n)]^2.\end{aligned} \tag{17.9.4}$$

We refer the above formula as *bias-variance trade-off*. The mean squared error can be divided into precisely two sources of error: the error from high bias and the error from high variance. On the one hand, the bias error is commonly seen in a simple model (such as a linear regression model), which cannot extract high dimensional relations between the features and the outputs. If a model suffers from high bias error, we often say it is *underfitting* or lack of *generalization* as introduced in (Section 4.4). On the flip side, the other error source—high variance usually results from a too complex model, which overfits the training data. As a result, an *overfitting* model is sensitive to small fluctuations in the data. If a model suffers from high variance, we often say it is *overfitting* and lack of *flexibility* as introduced in (Section 4.4).

Evaluating Estimators in Code

Since the standard deviation of an estimator has been implementing in MXNet by simply calling `a.std()` for a `ndarray` “a”, we will skip it but implement the statistical bias and the mean squared error in MXNet.

```python
# Statistical bias
def stat_bias(true_theta, est_theta):
    return(np.mean(est_theta) - true_theta)

# Mean squared error
def mse(data, true_theta):
    return(np.mean(np.square(data - true_theta)))
```

To illustrate the equation of the bias-variance trade-off, let’s simulate of normal distribution $\mathcal{N}(\theta, \sigma^2)$ with $10,000$ samples. Here, we use a $\theta = 1$ and $\sigma = 4$. As the estimator is a function of the given samples, here we use the mean of the samples as an estimator for true θ in this normal distribution $\mathcal{N}(\theta, \sigma^2)$.

```python
theta_true = 1
sigma = 4
sample_length = 10000
samples = np.random.normal(theta_true, sigma, sample_length)
theta_est = np.mean(samples)
theta_est
```

```
array(0.9503336)
```

Let’s validate the trade-off equation by calculating the summation of the squared bias and the variance of our estimator. First, calculate the MSE of our estimator.

```
mse(samples, theta_true)
```

```
array(15.781996)
```

Next, we calculate $\mathrm{Var}(\hat{\theta}_n) + [\mathrm{bias}(\hat{\theta}_n)]^2$ as below. As you can see, the two values agree to numerical precision.

```
bias = stat_bias(theta_true, theta_est)
np.square(samples.std()) + np.square(bias)
```

```
array(15.781995)
```

17.9.2 Conducting Hypothesis Tests

The most commonly encountered topic in statistical inference is hypothesis testing. While hypothesis testing was popularized in the early 20th century, the first use can be traced back to John Arbuthnot in the 1700s. John tracked 80-year birth records in London and concluded that more men were born than women each year. Following that, the modern significance testing is the intelligence heritage by Karl Pearson who invented p-value and Pearson's chi-squared test), William Gosset who is the father of Student's t-distribution, and Ronald Fisher who initialed the null hypothesis and the significance test.

A *hypothesis test* is a way of evaluating some evidence against the default statement about a population. We refer the default statement as the *null hypothesis* H_0, which we try to reject using the observed data. Here, we use H_0 as a starting point for the statistical significance testing. The *alternative hypothesis* H_A (or H_1) is a statement that is contrary to the null hypothesis. A null hypothesis is often stated in a declarative form which posits a relationship between variables. It should reflect the brief as explicit as possible, and be testable by statistics theory.

Imagine you are a chemist. After spending thousands of hours in the lab, you develop a new medicine which can dramatically improve one's ability to understand math. To show its magic power, you need to test it. Naturally, you may need some volunteers to take the medicine and see whether it can help them learn math better. How do you get started?

First, you will need carefully random selected two groups of volunteers, so that there is no difference between their math understanding ability measured by some metrics. The two groups are commonly referred to as the test group and the control group. The *test group* (or *treatment group*) is a group of individuals who will experience the medicine, while the *control group* represents the group of users who are set aside as a benchmark, i.e., identical environment setups except taking this medicine. In this way, the influence of all the variables are minimized, except the impact of the independent variable in the treatment.

Second, after a period of taking the medicine, you will need to measure the two groups' math understanding by the same metrics, such as letting the volunteers do the same tests after learning a new math formula. Then, you can collect their performance and compare the results. In this case, our null hypothesis will be that there is no difference between the two groups, and our alternate will be that there is.

This is still not fully formal. There are many details you have to think of carefully. For example, what is the suitable metrics to test their math understanding ability? How many volunteers for your test so you can be confident to claim the effectiveness of your medicine? How long should you run the test? How do you decided if there is a difference between the two groups? Do you care about the average performance only, or do you also the range of variation of the scores. And so on.

In this way, hypothesis testing provides framework for experimental design and reasoning about certainty in observed results. If we can now show that the null hypothesis is very unlikely to be true, we may reject it with confidence.

To complete the story of how to work with hypothesis testing, we need to now introduce some additional terminology and make some of our concepts above formal.

Statistical Significance

The *statistical significance* measures the probability of erroneously reject the null hypothesis, H_0, when it should not be rejected, i.e.,

$$\text{statistical significance } = 1 - \alpha = P(\text{reject } H_0 \mid H_0 \text{ is true}). \tag{17.9.5}$$

It is also referred to as the *type I error* or *false positive*. The α, is called as the *significance level* and its commonly used value is 5%, i.e., $1 - \alpha = 95\%$. The level of statistical significance level can be explained as the level of risk that we are willing to take, when we reject a true null hypothesis.

Fig. 17.9.1 shows the the observations' values and probability of a given normal distribution in a two-sample hypothesis test. If the observation data point is located outsides the 95% threshold, it will be a very unlikely observation under the null hypothesis assumption. Hence, there might be something wrong with the null hypothesis and we will reject it.

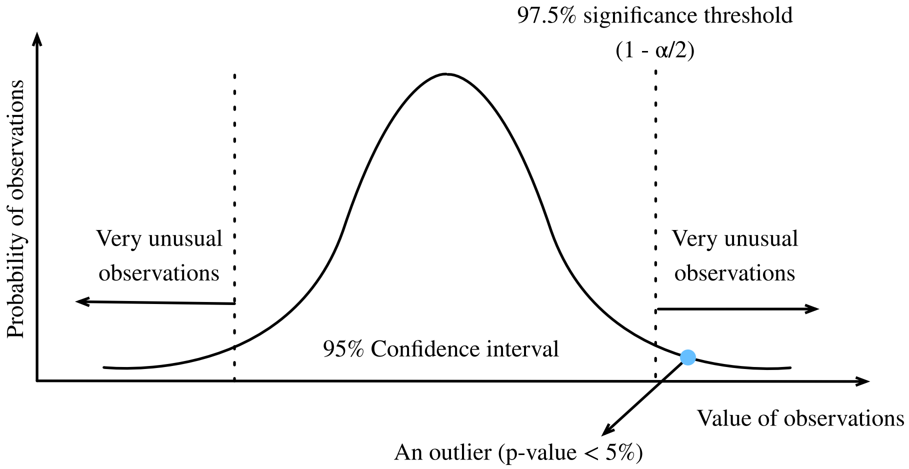

97.5% significance threshold
(1 - α/2)
Probability of observations
Very unusual observations
Very unusual observations
95% Confidence interval
Value of observations
An outlier (p-value < 5%)

Fig. 17.9.1: Statistical significance.

Statistical Power

The *statistical power* (or *sensitivity*) measures the probability of reject the null hypothesis, H_0, when it should be rejected, i.e.,

$$\text{statistical power } = P(\text{reject } H_0 \mid H_0 \text{ is false}). \tag{17.9.6}$$

Recall that a *type I error* is error caused by rejecting the null hypothesis when it is true, whereas a *type II error* is resulted from failing to reject the null hypothesis when it is false. A type II error is usually denoted as β, and hence the corresponding statistical power is $1 - \beta$.

Intuitively, statistical power can be interpreted as how likely our test will detect a real discrepancy of some minimum magnitude at a desired statistical significance level. 80% is a commonly used statistical power threshold. The higher the statistical power, the more likely we are to detect true differences.

One of the most common uses of statistical power is in determining the number of samples needed. The probability you reject the null hypothesis when it is false depends on the degree to which it is false (known as the *effect size*) and the number of samples you have. As you might expect, small effect sizes will require a very large number of samples to be detectable with high probability. While beyond the scope of this brief appendix to derive in detail, as an example, want to be able to reject a null hypothesis that our sample came from a mean zero variance one Gaussian, and we believe that our sample's mean is actually close to one, we can do so with acceptable error rates with a sample size of only 8. However, if we think our sample population true mean is close to 0.01, then we'd need a sample size of nearly 80000 to detect the difference.

We can imagine the power as a water filter. In this analogy, a high power hypothesis test is like a high quality water filtration system that will reduce harmful substances in the water as much as possible. On the other hand, a smaller discrepancy is like a low quality water filter, where some relative small substances may easily escape from the gaps. Similarly, if the statistical power is not of enough high power, then the test may not catch the smaller discrepancy.

Test Statistic

A *test statistic* $T(x)$ is a scalar which summarizes some characteristic of the sample data. The goal of defining such a statistic is that it should allow us to distinguish between different distributions and conduct our hypothesis test. Thinking back to our chemist example, if we wish to show that one population performs better than the other, it could be reasonable to take the mean as the test statistic. Different choices of test statistic can lead to statistical test with drastically different statistical power.

Often, $T(X)$ (the distribution of the test statistic under our null hypothesis) will follow, at least approximately, a common probability distribution such as a normal distribution when considered under the null hypothesis. If we can derive explicitly such a distribution, and then measure our test statistic on our dataset, we can safely reject the null hypothesis if our statistic is far outside the range that we would expect. Making this quantitative leads us to the notion of p-values.

p-value

The *:math:`p`-value* (or the *probability value*) is the probability that $T(X)$ is at least as extreme as the observed test statistic $T(x)$ assuming that the null hypothesis is *true*, i.e.,

$$p\text{-value} = P_{H_0}(T(X) \geq T(x)). \tag{17.9.7}$$

If the p-value is smaller than or equal to a pre-defined and fixed statistical significance level α, we may reject the null hypothesis. Otherwise, we will conclude that we are lack of evidence to reject the null hypothesis. For a given population distribution, the *region of rejection* will be the interval contained of all the points which has a p-value smaller than the statistical significance level α.

One-side Test and Two-sided Test

Normally there are two kinds of significance test: the one-sided test and the two-sided test. The *one-sided test* (or *one-tailed test*) is applicable when the null hypothesis and the alternative hypothesis only have one direction. For example, the null hypothesis may state that the true parameter θ is less than or equal to a value c. The alternative hypothesis would be that θ is greater than c. That is, the region of rejection is on only one side of the sampling distribution. Contrary to the one-sided test, the *two-sided test* (or *two-tailed test*) is applicable when the region of rejection is on both sides of the sampling distribution. An example in this case may have a null hypothesis state that the true parameter θ is equal to a value c. The alternative hypothesis would be that θ is not equal to c.

General Steps of Hypothesis Testing

After getting familiar with the above concepts, let's go through the general steps of hypothesis testing.

1. State the question and establish a null hypotheses H_0.
2. Set the statistical significance level α and a statistical power $(1 - \beta)$.
3. Obtain samples through experiments. The number of samples needed will depend on the statistical power, and the expected effect size.
4. Calculate the test statistic and the p-value.
5. Make the decision to keep or reject the null hypothesis based on the p-value and the statistical significance level α.

To conduct a hypothesis test, we start by defining a null hypothesis and a level of risk that we are willing to take. Then we calculate the test statistic of the sample, taking an extreme value of the test statistic as evidence against the null hypothesis. If the test statistic falls within the reject region, we may reject the null hypothesis in favor of the alternative.

Hypothesis testing is applicable in a variety of scenarios such as the clinical trails and A/B testing.

17.9.3 Constructing Confidence Intervals

When estimating the value of a parameter θ, point estimators like $\hat{\theta}$ are of limited utility since they contain no notion of uncertainty. Rather, it would be far better if we could produce an interval that would contain the true parameter θ with high probability. If you were interested in such ideas a century ago, then you would have been excited to read "Outline of a Theory of Statistical Estimation Based on the Classical Theory of Probability" by Jerzy Neyman (Neyman, 1937), who first introduced the concept of confidence interval in 1937.

To be useful, a confidence interval should be as small as possible for a given degree of certainty. Let's see how to derive it.

Definition

Mathematically, a *confidence interval* for the true parameter θ is an interval C_n that computed from the sample data such that

$$P_\theta(C_n \ni \theta) \geq 1 - \alpha, \forall \theta. \tag{17.9.8}$$

Here $\alpha \in (0, 1)$, and $1 - \alpha$ is called the *confidence level* or *coverage* of the interval. This is the same α as the significance level as we discussed about above.

Note that (17.9.8) is about variable C_n, not about the fixed θ. To emphasize this, we write $P_\theta(C_n \ni \theta)$ rather than $P_\theta(\theta \in C_n)$.

Interpretation

It is very tempting to interpret a 95% confidence interval as an interval where you can be 95% sure the true parameter lies, however this is sadly not true. The true parameter is fixed, and it is the interval that is random. Thus a better interpretation would be to say that if you generated a large number of confidence intervals by this procedure, 95% of the generated intervals would contain the true parameter.

This may seem pedantic, but it can have real implications for the interpretation of the results. In particular, we may satisfy (17.9.8) by constructing intervals that we are *almost certain* do not contain the true value, as long as we only do so rarely enough. We close this section by providing three tempting but false statements. An in-depth discussion of these points can be found in (Morey et al., 2016).

- **Fallacy 1**. Narrow confidence intervals mean we can estimate the parameter precisely.
- **Fallacy 2**. The values inside the confidence interval are more likely to be the true value than those outside the interval.
- **Fallacy 3**. The probability) that a particular observed 95% confidence interval contains the true value is 95%.

Sufficed to say, confidence intervals are subtle objects. However, if you keep the interpretation clear, they can be powerful tools.

A Gaussian Example

Let's discuss the most classical example, the confidence interval for the mean of a Gaussian of unknown mean and variance. Suppose we collect n samples $\{x_i\}_{i=1}^n$ from our Gaussian $\mathcal{N}(\mu, \sigma^2)$. We can compute estimators for the mean and standard deviation by taking

$$\hat{\mu}_n = \frac{1}{n}\sum_{i=1}^n x_i \text{ and } \hat{\sigma}^2_n = \frac{1}{n-1}\sum_{i=1}^n (x_i - \hat{\mu})^2. \tag{17.9.9}$$

If we now consider the random variable

$$T = \frac{\hat{\mu}_n - \mu}{\hat{\sigma}_n/\sqrt{n}}, \tag{17.9.10}$$

we obtain a random variable following a well-known distribution called the *Student's t-distribution on* $n-1$ *degrees of freedom*.

This distribution is very well studied, and it is known, for instance, that as $n \rightarrow \infty$, it is approximately a standard Gaussian, and thus by looking up values of the Gaussian c.d.f. in a table, we may conclude that the value of T is in the interval $[-1.96, 1.96]$ at least 95% of the time. For finite values of n, the interval needs to be somewhat larger, but are well known and precomputed in tables.

Thus, we may conclude that for large n,

$$P\left(\frac{\hat{\mu}_n - \mu}{\hat{\sigma}_n/\sqrt{n}} \in [-1.96, 1.96]\right) \ge 0.95. \tag{17.9.11}$$

Rearranging this by multiplying both sides by $\hat{\sigma}_n/\sqrt{n}$ and then adding $\hat{\mu}_n$, we obtain

$$P\left(\mu \in \left[\hat{\mu}_n - 1.96\frac{\hat{\sigma}_n}{\sqrt{n}}, \hat{\mu}_n + 1.96\frac{\hat{\sigma}_n}{\sqrt{n}}\right]\right) \ge 0.95. \tag{17.9.12}$$

Thus we know that we have found our 95% confidence interval:

$$\left[\hat{\mu}_n - 1.96\frac{\hat{\sigma}_n}{\sqrt{n}}, \hat{\mu}_n + 1.96\frac{\hat{\sigma}_n}{\sqrt{n}}\right]. \tag{17.9.13}$$

It is safe to say that (17.9.13) is one of the most used formula in statistics. Let's close our discussion of statistics by implementing it. For simplicity, we assume we are in the asymptotic regime. Small values of N should include the correct value of t_star obtained either programmatically or from a t-table.

```
# Number of samples
N = 1000

# Sample dataset
samples = np.random.normal(loc=0, scale=1, size=(N,))

# Lookup Students's t-distribution c.d.f.
t_star = 1.96

# Construct interval
mu_hat = np.mean(samples)
sigma_hat = samples.std(ddof=1)
(mu_hat - t_star*sigma_hat/np.sqrt(N), mu_hat + t_star*sigma_hat/np.sqrt(N))
```

```
(array(-0.07853346), array(0.04412608))
```

Summary

- Statistics focuses on inference problems, whereas deep learning emphasizes on making accurate predictions without explicitly programming and understanding.
- There are three common statistics inference methods: evaluating and comparing estimators, conducting hypothesis tests, and constructing confidence intervals.
- There are three most common estimators: statistical bias, standard deviation, and mean square error.
- A confidence interval is an estimated range of a true population parameter that we can construct by given the samples.
- Hypothesis testing is a way of evaluating some evidence against the default statement about a population.

Exercises

1. Let $X_1, X_2, \ldots, X_n \overset{\text{iid}}{\sim} \text{Unif}(0, \theta)$, where "iid" stands for *independent and identically distributed*. Consider the following estimators of θ:

$$\hat{\theta} = \max\{X_1, X_2, \ldots, X_n\}; \tag{17.9.14}$$

$$\tilde{\theta} = 2\bar{X}_n = \frac{2}{n}\sum_{i=1}^{n} X_i. \tag{17.9.15}$$

   - Find the statistical bias, standard deviation, and mean square error of $\hat{\theta}$.
   - Find the statistical bias, standard deviation, and mean square error of $\tilde{\theta}$.
   - Which estimator is better?

2. For our chemist example in introduction, can you derive the 5 steps to conduct a two-sided hypothesis testing? Given the statistical significance level $\alpha = 0.05$ and the statistical power $1 - \beta = 0.8$.
3. Run the confidence interval code with $N = 2$ and $\alpha = 0.5$ for 100 independently generated dataset, and plot the resulting intervals (in this case `t_star = 1.0`). You will see several very short intervals which are very far from containing the true mean 0. Does this contradict the interpretation of the confidence interval? Do you feel comfortable using short intervals to indicate high precision estimates?

17.10 Information Theory

The universe is overflowing with information. Information provides a common language across disciplinary rifts: from Shakespeare's Sonnet to researchers' paper on Cornell ArXiv, from Van Gogh's printing Starry Night to Beethoven's music Symphony No. 5, from the first programming language Plankalkül to the state-of-the-art machine learning algorithms. Everything must follow the rules of information theory, no matter the format. With information theory, we can measure and compare how much information is present in different signals. In this section, we will investigate the fundamental concepts of information theory and applications of information theory in machine learning.

Before we get started, let's outline the relationship between machine learning and information theory. Machine learning aims to extract interesting signals from data and make critical predictions. On the other hand, information theory studies encoding, decoding, transmitting, and manipulating information. As a result, information theory provides fundamental language for discussing the information processing in machine learned systems. For example, many machine learning applications use the cross entropy loss as described in Section 3.4. This loss can be directly derived from information theoretic considerations.

17.10.1 Information

Let's start with the "soul" of information theory: information. *Information* can be encoded in anything with a particular sequence of one or more encoding formats. Suppose that we task ourselves with trying to define a notion of information. What could be are starting point?

Consider the following thought experiment. We have a friend with a deck of cards. They will shuffle the deck, flip over some cards, and tell us statements about the cards. We will try to assess the information content of each statement.

First, they flip over a card and tell us, "I see a card." This provides us with no information at all. We were already certain that this was the case so we hope the information should be zero.

Next, they flip over a card and say, "I see a heart." This provides us some information, but in reality there are only 4 different suits that were possible, each equally likely, so we are not surprised by this outcome. We hope that whatever the measure of information, this event should have low information content.

Next, they flip over a card and say, "This is the 3 of spades." This is more information. Indeed there were 52 equally likely possible outcomes, and our friend told us which one it was. This should be a medium amount of information.

Let's take this to the logical extreme. Suppose that finally they flip over every card from the deck and read off the entire sequence of the shuffled deck. There are $52!$ different orders to the deck, again all equally likely, so we need a lot of information to know which one it is.

Any notion of information we develop must conform to this intuition. Indeed, in the next sections we will learn how to compute that these events have 0 bits, 2 bits, $~5.7$ bits, and $~225.6$ bits of information respectively.

If we read through these thought experiments, we see a natural idea. As a starting point, rather than caring about the knowledge, we may build off the idea that information represents the degree of surprise or the abstract possibility of the event. For example, if we want to describe an unusual event, we need a lot information. For a common event, we may not need much information.

In 1948, Claude E. Shannon published *A Mathematical Theory of Communication* (Shannon, 1948) establishing the theory of information. In his book, Shannon introduced the concept of information entropy for the first time. We will begin our journey here.

Self-information

Since information embodies the abstract possibility of an event, how do we map the possibility to the number of bits? Shannon introduced the terminology *bit* as the unit of information, which was originally created by John Tukey. So what is a "bit" and why do we use it to measure information? Historically, an antique transmitter can only send or receive two types of code: 0 and 1. Indeed, binary encoding is still in common use on all modern digital computers. In this way, any information is encoded by a series of 0 and 1. And hence, a series of binary digits of length n contains n bits of information.

Now, suppose that for any series of codes, each 0 or 1 occurs with a probability of $\frac{1}{2}$. Hence, an event X with a series of codes of length n, occurs with a probability of $\frac{1}{2^n}$. At the same time, as we mentioned before, this series contains n bits of information. So, can we generalize to a math function which can transfer the probability p to the number of bits? Shannon gave the answer by defining *self-information*

$$I(X) = - \log_2 (p), \tag{17.10.1}$$

as the *bits* of information we have received for this event X. Note that we will always use base-2 logarithms in this section. For the sake of simplicity, the rest of this section will omit the subscript 2 in the logarithm notation, i.e., $\log(.)$ always refers to $\log_2(.)$. For example, the code "0010" has a self-information

$$I(``0010") = - \log (p(``0010")) = - \log \left(\frac{1}{2^4} \right) = 4 \text{ bits}. \tag{17.10.2}$$

We can calculate self information in MXNet as shown below. Before that, let's first import all the necessary packages in this section.

```python
from mxnet import np
from mxnet.metric import NegativeLogLikelihood
from mxnet.ndarray import nansum
import random

def self_information(p):
    return -np.log2(p)

self_information(1/64)
```

```
6.0
```

17.10.2 Entropy

As self-information only measures the information of a single discrete event, we need a more generalized measure for any random variable of either discrete or continuous distribution.

Motivating Entropy

Let's try to get specific about what we want. This will be an informal statement of what are known as the *axioms of Shannon entropy*. It will turn out that the following collection of common-sense statements force us to a unique definition of information. A formal version of these axioms, along with several others may be found in (Csiszar, 2008).

1. The information we gain by observing a random variable does not depend on what we call the elements, or the presence of additional elements which have probability zero.
2. The information we gain by observing two random variables is no more than the sum of the information we gain by observing them separately. If they are independent, then it is exactly the sum.
3. The information gained when observing (nearly) certain events is (nearly) zero.

While proving this fact is beyond the scope of our text, it is important to know that this uniquely determines the form that entropy must take. The only ambiguity that these allow is in the choice of fundamental units, which is most often normalized by making the choice we saw before that the information provided by a single fair coin flip is one bit.

Definition

For any random variable X that follows a probability distribution P with a probability density function (p.d.f.) or a probability mass function (p.m.f.) $p(x)$, we measure the expected amount of information through *entropy* (or *Shannon entropy*)

$$H(X) = -E_{x \sim P} [\log p(x)]. \tag{17.10.3}$$

To be specific, if X is discrete,

$$H(X) = - \sum_i p_i \log p_i, \text{ where } p_i = P(X_i). \tag{17.10.4}$$

Otherwise, if X is continuous, we also refer entropy as *differential entropy*

$$H(X) = - \int_x p(x) \log p(x) \; dx. \tag{17.10.5}$$

In MXNet, we can define entropy as below.

```python
def entropy(p):
    entropy = - p * np.log2(p)
    # nansum will sum up the non-nan number
    out = nansum(entropy.as_nd_ndarray())
    return out

entropy(np.array([0.1, 0.5, 0.1, 0.3]))
```

```
[1.6854753]
<NDArray 1 @cpu(0)>
```

Interpretations

You may be curious: in the entropy definition (17.10.3), why do we use an expectation of a negative logarithm? Here are some intuitions.

First, why do we use a *logarithm* function log? Suppose that $p(x) = f_1(x)f_2(x)\ldots, f_n(x)$, where each component function $f_i(x)$ is independent from each other. This means that each $f_i(x)$ contributes independently to the total information obtained from $p(x)$. As discussed above, we want the entropy formula to be additive over independent random variables. Luckily, log can naturally turn a product of probability distributions to a summation of the individual terms.

Next, why do we use a *negative* log? Intuitively, more frequent events should contain less information than less common events, since we often gain more information from an unusual case than from an ordinary one. However, log is monotonically increasing with the probabilities, and indeed negative for all values in $[0, 1]$. We need to construct a monotonically decreasing relationship between the probability of events and their entropy, which will ideally be always positive (for nothing we observe should force us to forget what we have known). Hence, we add a negative sign in front of log function.

Last, where does the *expectation* function come from? Consider a random variable X. We can interpret the self-information ($-\log(p)$) as the amount of *surprise* we have at seeing a particular outcome. Indeed, as the probability approaches zero, the surprise becomes infinite. Similarly, we can interpret The entropy as the average amount of surprise from observing X. For example, imagine that a slot machine system emits statistical independently symbols $s_1, \ldots, s_k$ with probabilities $p_1, \ldots, p_k$ respectively. Then the entropy of this system equals to the average self-information from observing each output, i.e.,

$$H(S) = \sum_i p_i \cdot I(s_i) = - \sum_i p_i \cdot \log p_i. \tag{17.10.6}$$

Properties of Entropy

By the above examples and interpretations, we can derive the following properties of entropy (17.10.3). Here, we refer to X as an event and P as the probability distribution of X.

- Entropy is non-negative, i.e., $H(X) \geq 0, \forall X$.
- If $X \sim P$ with a p.d.f. or a p.m.f. $p(x)$, and we try to estimate P by a new probability distribution Q with a p.d.f. or a p.m.f. $q(x)$, then

  $$H(X) = -E_{x \sim P}[\log p(x)] \leq -E_{x \sim P}[\log q(x)], \text{ with equality if and only if } P = Q. \tag{17.10.7}$$

  Alternatively, $H(X)$ gives a lower bound of the average number of bits needed to encode symbols drawn from P.
- If $X \sim P$, then x conveys the maximum amount of information if it spreads evenly among all possible outcomes. Specifically, if the probability distribution P is discrete with k-class $\{p_1, \ldots, p_k\}$, then

  $$H(X) \leq \log(k), \text{ with equality if and only if } p_i = \frac{1}{k}, \forall x_i. \tag{17.10.8}$$

If P is a continuous random variable, then the story becomes much more complicated. However, if we additionally impose that P is supported on a finite interval (with all values between 0 and 1), then P has the highest entropy if it is the uniform distribution on that interval.

17.10.3 Mutual Information

Previously we defined entropy of a single random variable X, how about the entropy of a pair random variables (X, Y)? We can think of these techniques as trying to answer the following type of question, "What information is contained in X and Y together compared to each separately? Is there redundant information, or is it all unique?"

For the following discussion, we always use (X, Y) as a pair of random variables that follows a joint probability distribution P with a p.d.f. or a p.m.f. $p_{X,Y}(x, y)$, while X and Y follow probability distribution $p_X(x)$ and $p_Y(y)$, respectively.

Joint Entropy

Similar to entropy of a single random variable (17.10.3), we define the *joint entropy* $H(X, Y)$ of a pair random variables (X, Y) as

$$H(X, Y) = -E_{(x,y)\sim P}[\log p_{X,Y}(x, y)]. \tag{17.10.9}$$

Precisely, on the one hand, if (X, Y) is a pair of discrete random variables, then

$$H(X, Y) = -\sum_x \sum_y p_{X,Y}(x, y) \log p_{X,Y}(x, y). \tag{17.10.10}$$

On the other hand, if (X, Y) is a pair of continuous random variables, then we define the *differential joint entropy* as

$$H(X, Y) = -\int_{x,y} p_{X,Y}(x, y) \ \log p_{X,Y}(x, y) \; dx \; dy. \tag{17.10.11}$$

We can think of (17.10.9) as telling us the total randomness in the pair of random variables. As a pair of extremes, if $X = Y$ are two identical random variables, then the information in the pair is exactly the information in one and we have $H(X, Y) = H(X) = H(Y)$. On the other extreme, if X and Y are independent then $H(X, Y) = H(X) + H(Y)$. Indeed we will always have that the information contained in a pair of random variables is no smaller than the entropy of either random variable and no more than the sum of both.

$$H(X), H(Y) \le H(X, Y) \le H(X) + H(Y). \tag{17.10.12}$$

Let's implement joint entropy from scratch in MXNet.

```
def joint_entropy(p_xy):
    joint_ent = -p_xy * np.log2(p_xy)
    # nansum will sum up the non-nan number
    out = nansum(joint_ent.as_nd_ndarray())
    return out

joint_entropy(np.array([[0.1, 0.5], [0.1, 0.3]]))
```

```
[1.6854753]
<NDArray 1 @cpu(0)>
```

Notice that this is the same *code* as before, but now we interpret it differently as working on the joint distribution of the two random variables.

Conditional Entropy

The joint entropy defined above the amount of information contained in a pair of random variables. This is useful, but often times it is not what we care about. Consider the setting of machine learning. Let's take X to be the random variable (or vector of random variables) that describes the pixel values of an image, and Y to be the random variable which is the class label. X should contain substantial information—a natural image is a complex thing. However, the information contained in Y once the image has been show should be low. Indeed, the image of a digit should already contain the information about what digit it is unless the digit is illegible. Thus, to continue to extend our vocabulary of information theory, we need to be able to reason about the information content in a random variable conditional on another.

In the probability theory, we saw the definition of the *conditional probability* to measure the relationship between variables. We now want to analogously define the *conditional entropy* $H(Y \mid X)$. We can write this as

$$H(Y \mid X) = -E_{(x,y) \sim P} [\log p(y \mid x)], \tag{17.10.13}$$

where $p(y \mid x) = \frac{p_{X,Y}(x,y)}{p_X(x)}$ is the conditional probability. Specifically, if (X, Y) is a pair of discrete random variables, then

$$H(Y \mid X) = - \sum_{x} \sum_{y} p(x, y) \log p(y \mid x). \tag{17.10.14}$$

If (X, Y) is a pair of continuous random variables, then the *differential joint entropy* is similarly defined as

$$H(Y \mid X) = - \int_x \int_y p(x, y) \ \log p(y \mid x) \;dx \;dy. \tag{17.10.15}$$

It is now natural to ask, how does the *conditional entropy* $H(Y \mid X)$ relate to the entropy $H(X)$ and the joint entropy $H(X, Y)$? Using the definitions above, we can express this cleanly:

$$H(Y \mid X) = H(X, Y) - H(X). \tag{17.10.16}$$

This has an intuitive interpretation: the information in Y given X ($H(Y \mid X)$) is the same as the information in both X and Y together ($H(X, Y)$) minus the information already contained in X. This gives us the information in Y which is not also represented in X.

Now, let's implement conditional entropy (17.10.13) from scratch in MXNet.

```
def conditional_entropy(p_xy, p_x):
    p_y_given_x = p_xy/p_x
    cond_ent = -p_xy * np.log2(p_y_given_x)
    # nansum will sum up the non-nan number
    out = nansum(cond_ent.as_nd_ndarray())
    return out

conditional_entropy(np.array([[0.1, 0.5], [0.2, 0.3]]), np.array([0.2, 0.8]))
```

```
[0.8635472]
<NDArray 1 @cpu(0)>
```

Mutual Information

Given the previous setting of random variables (X, Y), you may wonder: "Now that we know how much information is contained in Y but not in X, can we similarly ask how much information is shared between X and Y?" The answer will be the *mutual information* of (X, Y), which we will write as $I(X, Y)$.

Rather than diving straight into the formal definition, let's practice our intuition by first trying to derive an expression for the mutual information entirely based on terms we have constructed before. We wish to find the information shared between two random variables. One way we could try to do this is to start with all the information contained in both X and Y together, and then we take off the parts that are not shared. The information contained in both X and Y together is written as $H(X, Y)$. We want to subtract from this the information contained in X but not in Y, and the information contained in Y but not in X. As we saw in the previous section, this is given by $H(X \mid Y)$ and $H(Y \mid X)$ respectively. Thus, we have that the mutual information should be

$$I(X, Y) = H(X, Y) - H(Y \mid X) - H(X \mid Y). \tag{17.10.17}$$

Indeed, this is a valid definition for the mutual information. If we expand out the definitions of these terms and combine them, a little algebra shows that this is the same as

$$I(X, Y) = -E_x E_y \left\{ p_{X,Y}(x, y) \log \frac{p_{X,Y}(x, y)}{p_X(x) p_Y(y)} \right\}. \tag{17.10.18}$$

We can summarize all of these relationships in image Fig. 17.10.1. It is an excellent test of intuition to see why the following statements are all also equivalent to $I(X, Y)$.

- $H(X) - H(X \mid Y)$
- $H(Y) - H(Y \mid X)$
- $H(X) + H(Y) - H(X, Y)$

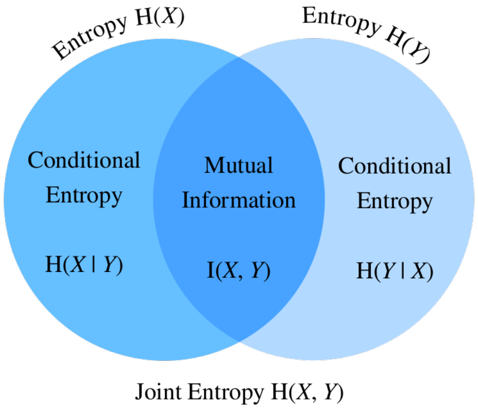


Fig. 17.10.1: Mutual information's relationship with joint entropy and conditional entropy.

In many ways we can think of the mutual information (17.10.18) as principled extension of correlation coefficient we saw in Section 17.6. This allows us to ask not only for linear relationships between variables, but for the maximum information shared between the two random variables of any kind.

Now, let's implement mutual information from scratch.

```
def mutual_information(p_xy, p_x, p_y):
    p = p_xy / (p_x * p_y)
    mutual = -p_xy * np.log2(p)
    # nansum will sum up the non-nan number
    out = nansum(mutual.as_nd_ndarray())
    return out

mutual_information(np.array([[0.1, 0.5], [0.1, 0.3]]),
                   np.array([0.2, 0.8]),
                   np.array([[0.75, 0.25]]))
```

```
[-0.71946025]
<NDArray 1 @cpu(0)>
```

Properties of Mutual Information

Rather than memorizing the definition of mutual information (17.10.18), you only need to keep in mind its notable properties:

- Mutual information is symmetric, i.e., $I(X, Y) = I(Y, X)$.
- Mutual information is non-negative, i.e., $I(X, Y) \geq 0$.
- $I(X, Y) = 0$ if and only if X and Y are independent. For example, if X and Y are independent, then knowing Y does not give any information about X and vice versa, so their mutual information is zero.
- Alternatively, if X is an invertible function of Y, then Y and X share all information and

$$I(X, Y) = H(Y) = H(X). \tag{17.10.19}$$

Pointwise Mutual Information

When we worked with entropy at the beginning of this chapter, we were able to provide an interpretation of $-\log(p_X(x))$ as how *surprised* we were with the particular outcome. We may give a similar interpretation to the logarithmic term in the mutual information, which is often referred to as the *pointwise mutual information*:

$$\mathrm{pmi}(x, y) = \log\frac{p_{X,Y}(x, y)}{p_X(x)p_Y(y)}. \tag{17.10.20}$$

We can think of (17.10.20) as measuring how much more or less likely the specific combination of outcomes x and y are compared to what we would expect for independent random outcomes. If it is large and positive, then these two specific outcomes occur much more frequently than they would compared to random chance (*note*: the denominator is $p_X(x)p_Y(y)$ which is the probability of the two outcomes were independent), whereas if it is large and negative it represents the two outcomes happening far less than we would expect by random chance.

This allows us to interpret the mutual information (17.10.18) as the average amount that we were surprised to see two outcomes occurring together compared to what we would expect if they were independent.

Applications of Mutual Information

Mutual information may be a little abstract in it pure definition, so how does it related to machine learning? In natural language processing, one of the most difficult problems is the *ambiguity resolution*, or the issue of the meaning of a word being unclear from context. For example, recently a headline in the news reported that "Amazon is on fire". You may wonder whether the company Amazon has a building on fire, or the Amazon rain forest is on fire.

In this case, mutual information can help us resolve this ambiguity. We first find the group of words that each has a relatively large mutual information with the company Amazon, such as e-commerce, technology, and online. Second, we find another group of words that each has a relatively large mutual information with the Amazon rain forest, such as rain, forest, and tropical. When we need to disambiguate "Amazon", we can compare which group has more occurrence in the context of the word Amazon. In this case the article would go on to describe the forest, and make the context clear.

17.10.4 Kullback–Leibler Divergence

As what we have discussed in Section 2.3, we can use norms to measure distance between two points in space of any dimensionality. We would like to be able to do a similar task with probability distributions. There are many ways to go about this, but information theory provides one of the nicest. We now explore the *Kullback–Leibler (KL) divergence*, which provides a way to measure if two distributions are close together or not.

Definition

Given a random variable X that follows the probability distribution P with a p.d.f. or a p.m.f. $p(x)$, and we estimate P by another probability distribution Q with a p.d.f. or a p.m.f. $q(x)$. Then the *Kullback–Leibler (KL) divergence* (or *relative entropy*) between P and Q is

$$D_{\mathrm{KL}}(P\|Q) = E_{x \sim P} \left[\log \frac{p(x)}{q(x)} \right]. \tag{17.10.21}$$

As with the pointwise mutual information (17.10.20), we can again provide an interpretation of the logarithmic term: $-\log \frac{q(x)}{p(x)} = -\log(q(x)) - (-\log(p(x)))$ will be large and positive if we see x far more often under P than we would expect for Q, and large and negative if we see the outcome far less than expected. In this way, we can interpret it as our *relative* surprise at observing the outcome compared to how surprised we would be observing it from our reference distribution.

In MXNet, let's implement the KL divergence from Scratch.

```
def kl_divergence(p, q):
    kl = p * np.log2(p / q)
    out = nansum(kl.as_nd_ndarray())
    return out.abs().asscalar()
```

KL Divergence Properties

Let's take a look at some properties of the KL divergence (17.10.21).

- KL divergence is non-symmetric, i.e.,

$$D_{\mathrm{KL}}(P\|Q) \neq D_{\mathrm{KL}}(Q\|P), \text{ if } P \neq Q. \tag{17.10.22}$$

- KL divergence is non-negative, i.e.,

$$D_{\mathrm{KL}}(P\|Q) \geq 0. \tag{17.10.23}$$

  Note that the equality holds only when $P = Q$.

- If there exists an x such that $p(x) > 0$ and $q(x) = 0$, then $D_{\mathrm{KL}}(P\|Q) = \infty$.
- There is a close relationship between KL divergence and mutual information. Besides the relationship shown in Fig. 17.10.1, $I(X, Y)$ is also numerically equivalent with the following terms:

  1. $D_{\mathrm{KL}}(P(X, Y) \mid\mid P(X)P(Y))$;
  2. $E_Y \{ D_{\mathrm{KL}}(P(X \mid Y) \mid\mid P(X)) \}$;
  3. $E_X \{ D_{\mathrm{KL}}(P(Y \mid X) \mid\mid P(Y)) \}$.

  For the first term, we interpret mutual information as the KL divergence between $P(X, Y)$ and the product of $P(X)$ and $P(Y)$, and thus is a measure of how different the joint distribution is from the distribution if they were independent. For the second term, mutual information tells us the average reduction in uncertainty about Y that results from learning the value of the X's distribution. Similarly to the third term.

Example

Let's go through a toy example to see the non-symmetry explicitly.

First, let's generate and sort three ndarrays of length $10,000$: an objective ndarray p which follows a normal distribution $N(0, 1)$, and two candidate ndarrays q_1 and q_2 which follow normal distributions $N(-1, 1)$ and $N(1, 1)$ respectively.

```
random.seed(1)

nd_length = 10000
p = np.random.normal(loc=0, scale=1, size=(nd_length, ))
q1 = np.random.normal(loc=-1, scale=1, size=(nd_length, ))
q2 = np.random.normal(loc=1, scale=1, size=(nd_length, ))

p = np.array(sorted(p.asnumpy()))
q1 = np.array(sorted(q1.asnumpy()))
q2 = np.array(sorted(q2.asnumpy()))
```

Since q_1 and q_2 are symmetric with respect to the y-axis (i.e., $x = 0$), we expect a similar value of KL divergence between $D_{\mathrm{KL}}(p\|q_1)$ and $D_{\mathrm{KL}}(p\|q_2)$. As you can see below, there is only a 1% off between $D_{\mathrm{KL}}(p\|q_1)$ and $D_{\mathrm{KL}}(p\|q_2)$.

```
kl_pq1 = kl_divergence(p, q1)
kl_pq2 = kl_divergence(p, q2)
similar_percentage = abs(kl_pq1 - kl_pq2) / ((kl_pq1 + kl_pq2) / 2) * 100

kl_pq1, kl_pq2, similar_percentage
```

```
(8470.638, 8664.999, 2.268504302642314)
```

In contrast, you may find that $D_{\mathrm{KL}}(q_2 \| p)$ and $D_{\mathrm{KL}}(p \| q_2)$ are off a lot, with around 40% off as shown below.

```
kl_q2p = kl_divergence(q2, p)
differ_percentage = abs(kl_q2p - kl_pq2) / ((kl_q2p + kl_pq2) / 2) * 100

kl_q2p, differ_percentage
```

```
(13536.835, 43.88678828000115)
```

17.10.5 Cross Entropy

If you are curious about applications of information theory in deep learning, here is a quick example. We define the true distribution P with probability distribution $p(x)$, and the estimated distribution Q with probability distribution $q(x)$, and we will use them in the rest of this section.

Say we need to solve a binary classification problem based on given n data points $\{x_1, \ldots, x_n\}$. Assume that we encode 1 and 0 as the positive and negative class label y_i respectively, and our neural network is parameterized by θ. If we aim to find a best θ so that $\hat{y}_i = p_\theta(y_i \mid x_i)$, it is natural to apply the maximum log-likelihood approach as was seen in Section 17.7. To be specific, for true labels y_i and predictions $\hat{y}_i = p_\theta(y_i \mid x_i)$, the probability to be classified as positive is $\pi_i = p_\theta(y_i = 1 \mid x_i)$. Hence, the log-likelihood function would be

$$\begin{aligned} l(\theta) &= \log L(\theta) \\ &= \log \prod_{i=1}^n \pi_i^{y_i} (1 - \pi_i)^{1 - y_i} \\ &= \sum_{i=1}^n y_i \log(\pi_i) + (1 - y_i) \log(1 - \pi_i). \end{aligned} \tag{17.10.24}$$

Maximizing the log-likelihood function $l(\theta)$ is identical to minimizing $-l(\theta)$, and hence we can find the best θ from here. To generalize the above loss to any distributions, we also called $-l(\theta)$ the *cross entropy loss* $\mathrm{CE}(y, \hat{y})$, where y follows the true distribution P and $\hat{y}$ follows the estimated distribution Q.

This was all derived by working from the maximum likelihood point of view. However, if we look closely we can see that terms like $\log(\pi_i)$ have entered into our computation which is a solid indication that we can understand the expression from an information theoretic point of view.

Formal Definition

Like KL divergence, for a random variable X, we can also measure the divergence between the estimating distribution Q and the true distribution P via *cross entropy*,

$$\mathrm{CE}(P, Q) = -E_{x \sim P}[\log(q(x))]. \tag{17.10.25}$$

By using properties of entropy discussed above, we can also interpret it as the summation of the entropy $H(P)$ and the KL divergence between P and Q, i.e.,

$$\mathrm{CE}(P, Q) = H(P) + D_{\mathrm{KL}}(P\|Q). \tag{17.10.26}$$

In MXNet, we can implement the cross entropy loss as below.

```
def cross_entropy(y_hat, y):
    ce = -np.log(y_hat[range(len(y_hat)), y])
    return ce.mean()
```

Now define two `ndarray`s for the labels and predictions, and calculate the cross entropy loss of them.

```
labels = np.array([0, 2])
preds = np.array([[0.3, 0.6, 0.1], [0.2, 0.3, 0.5]])

cross_entropy(preds, labels)
```

```
array(0.94856)
```

Properties

As alluded in the beginning of this section, cross entropy (17.10.25) can be used to define a loss function in the optimization problem. It turns out that the following are equivalent:

1. Maximizing predictive probability of Q for distribution P, (i.e., $E_{x\sim P}[\log(q(x))]$);
2. Minimizing cross entropy $\mathrm{CE}(P, Q)$;
3. Minimizing the KL divergence $D_{\mathrm{KL}}(P\|Q)$.

The definition of cross entropy indirectly proves the equivalent relationship between objective 2 and objective 3, as long as the entropy of true data $H(P)$ is constant.

Cross Entropy as An Objective Function of Multi-class Classification

If we dive deep into the classification objective function with cross entropy loss CE, we will find minimizing CE is equivalent to maximizing the log-likelihood function L.

To begin with, suppose that we are given a dataset with n samples, and it can be classified into k-classes. For each data point i, we represent any k-class label $\mathbf{y}_i = (y_{i1}, \ldots, y_{ik})$ by *one-hot encoding*. To be specific, if the data point i belongs to class j, then we set the j-th entry to 1, and all other components to 0, i.e.,

$$y_{ij} = \begin{cases}1 & j \in J; \\ 0 & \text{otherwise.}\end{cases} \tag{17.10.27}$$

For instance, if a multi-class classification problem contains three classes A, B, and C, then the labels $\mathbf{y}_i$ can be encoded in $\{A: (1,0,0); B: (0,1,0); C: (0,0,1)\}$.

Assume that our neural network is parameterized by θ. For true label vectors $\mathbf{y}_i$ and predictions

$$\hat{\mathbf{y}}_i = p_\theta(\mathbf{y}_i \mid \mathbf{x}_i) = \sum_{j=1}^{k} y_{ij} p_\theta(y_{ij} \mid \mathbf{x}_i). \tag{17.10.28}$$

Hence, the *cross entropy loss* would be

$$\mathrm{CE}(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_{i=1}^{n} \mathbf{y}_i \log \hat{\mathbf{y}}_i = -\sum_{i=1}^{n} \sum_{j=1}^{k} y_{ij} \log p_\theta(y_{ij} \mid \mathbf{x}_i). \tag{17.10.29}$$

On the other side, we can also approach the problem through maximum likelihood estimation. To begin with, let's quickly introduce a k-class multinoulli distribution. It is an extension of the Bernoulli distribution from binary class to multi-class. If a random variable $\mathbf{z} = (z_1, \ldots, z_k)$ follows a k-class *multinoulli distribution* with probabilities $\mathbf{p} = (p_1, \ldots, p_k)$, i.e.,

$$p(\mathbf{z}) = p(z_1, \ldots, z_k) = \mathrm{Multi}(p_1, \ldots, p_k), \text{ where } \sum_{i=1}^{k} p_i = 1, \tag{17.10.30}$$

then the joint probability mass function(p.m.f.) of $\mathbf{z}$ is

$$\mathbf{p}^{\mathbf{z}} = \prod_{j=1}^{k} p_j^{z_j}. \tag{17.10.31}$$

It can be seen that each data point, $\mathbf{y}_i$, is following a k-class multinoulli distribution with probabilities $\boldsymbol{\pi} = (\pi_1, \ldots, \pi_k)$. Therefore, the joint p.m.f. of each data point $\mathbf{y}_i$ is $\pi^{\mathbf{y}_i} = \prod_{j=1}^{k} \pi_j^{y_{ij}}$. Hence, the log-likelihood function would be

$$l(\theta) = \log L(\theta) = \log \prod_{i=1}^{n} \boldsymbol{\pi}^{\mathbf{y}_i} = \log \prod_{i=1}^{n} \prod_{j=1}^{k} \pi_j^{y_{ij}} = \sum_{i=1}^{n} \sum_{j=1}^{k} y_{ij} \log \pi_j. \tag{17.10.32}$$

Since in maximum likelihood estimation, we maximizing the objective function $l(\theta)$ by having $\pi_j = p_\theta(y_{ij} \mid \mathbf{x}_i)$. Therefore, for any multi-class classification, maximizing the above log-likelihood function $l(\theta)$ is equivalent to minimizing the CE loss $\mathrm{CE}(y, \hat{y})$.

To test the above proof, let's apply the built-in measure `NegativeLogLikelihood` in MXNet. Using the same `labels` and `preds` as in the earlier example, we will get the same numerical loss as the previous example up to the 5 decimal place.

```
nll_loss = NegativeLogLikelihood()
nll_loss.update(labels.as_nd_ndarray(), preds.as_nd_ndarray())
nll_loss.get()
```

```
('nll-loss', 0.9485599994659424)
```

Summary

- Information theory is a field of study about encoding, decoding, transmitting, and manipulating information.
- Entropy is the unit to measure how much information is presented in different signals.
- KL divergence can also measure the divergence between two distributions.
- Cross Entropy can be viewed as an objective function of multi-class classification. Minimizing cross entropy loss is equivalent to maximizing the log-likelihood function.

Exercises

1. Verify that the card examples from the first section indeed have the claimed entropy.
2. Let's compute the entropy from a few data sources:
    - Assume that you are watching the output generated by a monkey at a typewriter. The monkey presses any of the 44 keys of the typewriter at random (you can assume that it has not discovered any special keys or the shift key yet). How many bits of randomness per character do you observe?
    - Being unhappy with the monkey, you replaced it by a drunk typesetter. It is able to generate words, albeit not coherently. Instead, it picks a random word out of a vocabulary of $2,000$ words. Moreover, assume that the average length of a word is 4.5 letters in English. How many bits of randomness do you observe now?
    - Still being unhappy with the result, you replace the typesetter by a high quality language model. These can currently obtain perplexity numbers as low as 15 points per character. The perplexity is defined as a length normalized probability, i.e.,

      $$PPL(x) = [p(x)]^{1/\text{length(x)}}. \tag{17.10.33}$$

      How many bits of randomness do you observe now?
3. Explain intuitively why $I(X,Y) = H(X) - H(X|Y)$. Then, show this is true by expressing both sides as an expectation with respect to the joint distribution.
4. What is the KL Divergence between the two Gaussian distributions $\mathcal{N}(\mu_1, \sigma_1^2)$ and $\mathcal{N}(\mu_2, \sigma_2^2)$?

18 | Appendix: Tools for Deep Learning

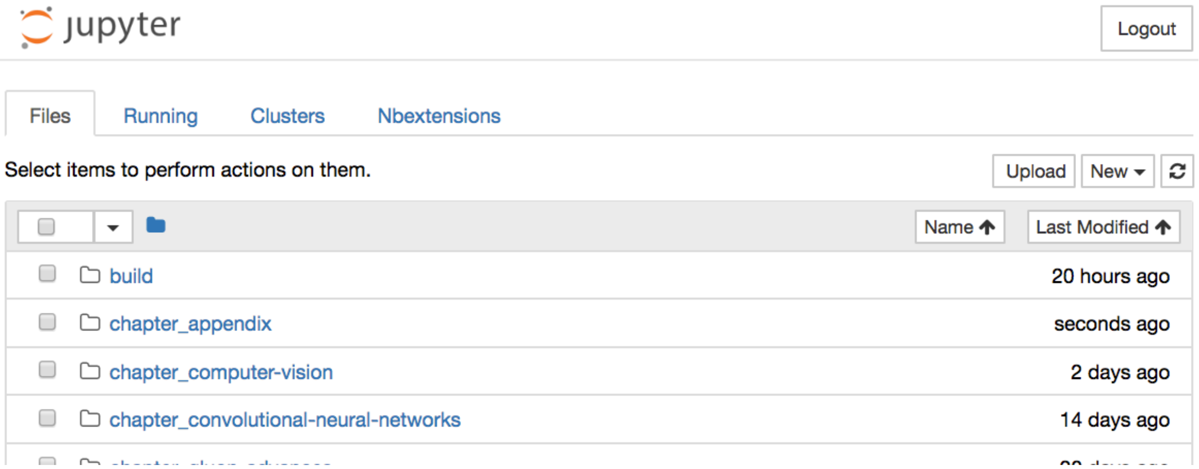
In this chapter, we will walk you through major tools for deep learning, from introducing Jupyter notebook in Section 18.1 to empowering you training models on Cloud such as Amazon Sagemaker Section 18.2, Amazon EC2 Section 18.3 and Google Colab. Besides, if you would like to purchase your own GPUs, we also note down some practical suggestions in Section 18.5. If you are interested in being a contributor of this book, you may follow the instructions in .. _sec_how_to_contribute:.

18.1 Using Jupyter

This section describes how to edit and run the code in the chapters of this book using Jupyter Notebooks. Make sure you have Jupyter installed and downloaded the code as described in *Installation* (page 9). If you want to know more about Jupyter see the excellent tutorial in the Documentation[263].

18.1.1 Editing and Running the Code Locally

Suppose that the local path of code of the book is "xx/yy/d2l-en/". Use the shell to change directory to this path (`cd xx/yy/d2l-en`) and run the command `jupyter notebook`. If your browser does not do this automatically, open http://localhost:8888 and you will see the interface of Jupyter and all the folders containing the code of the book, as shown in Fig. 18.1.1.



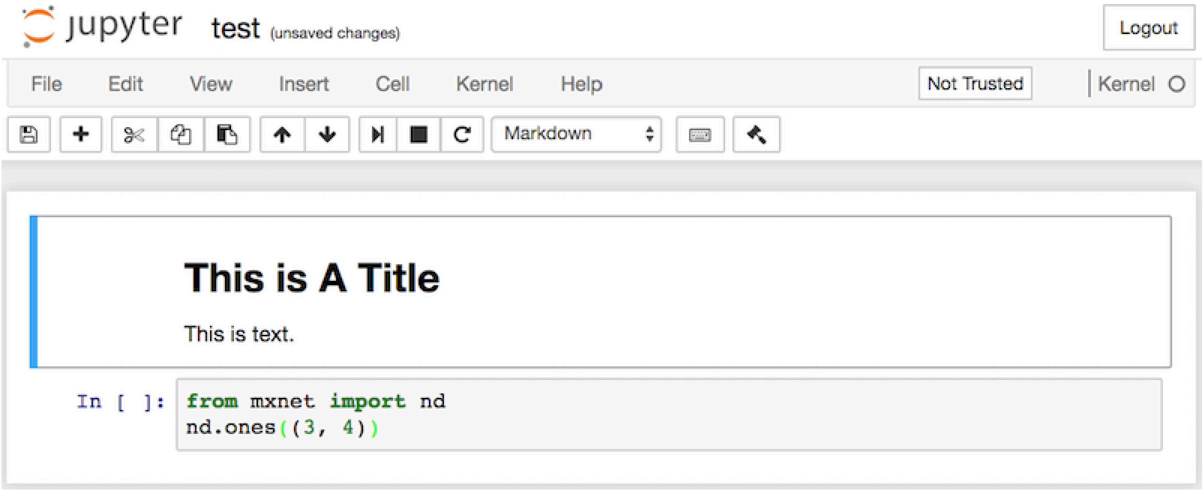
Select items to perform actions on them.

□ build	20 hours ago
□ chapter_appendix	seconds ago
□ chapter_computer-vision	2 days ago
□ chapter_convolutional-neural-networks	14 days ago

Fig. 18.1.1: The folders containing the code in this book.

[263] https://jupyter.readthedocs.io/en/latest/

You can access the notebook files by clicking on the folder displayed on the webpage. They usually have the suffix .ipynb. For the sake of brevity, we create a temporary "test.ipynb" file. The content displayed after you click it is as shown in Fig. 18.1.2. This notebook includes a markdown cell and a code cell. The content in the markdown cell includes "This is A Title" and "This is text". The code cell contains two lines of Python code.



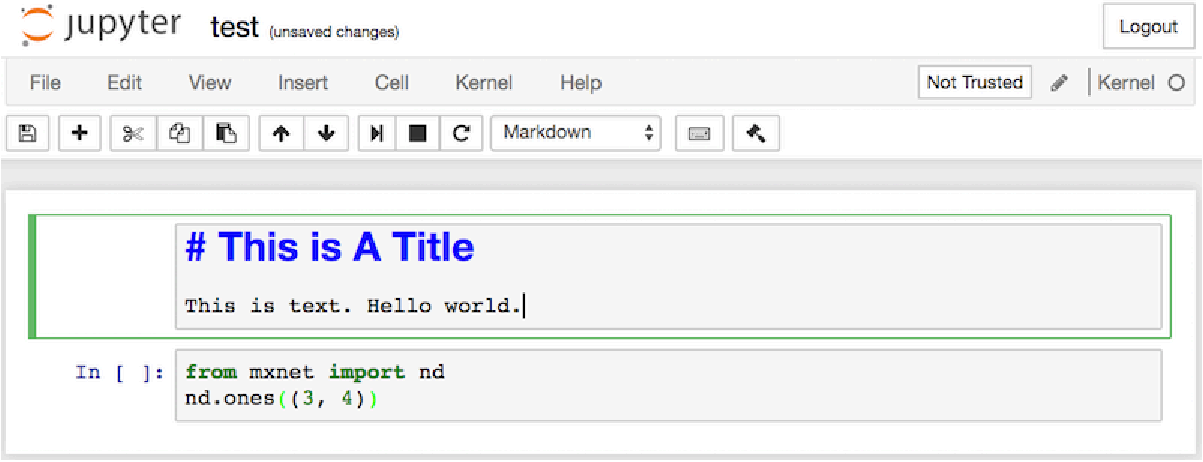
This is A Title

This is text.

```
In [ ]: from mxnet import nd
        nd.ones((3, 4))
```

Fig. 18.1.2: Markdown and code cells in the "text.ipynb" file.

Double click on the markdown cell to enter edit mode. Add a new text string "Hello world." at the end of the cell, as shown in Fig. 18.1.3.

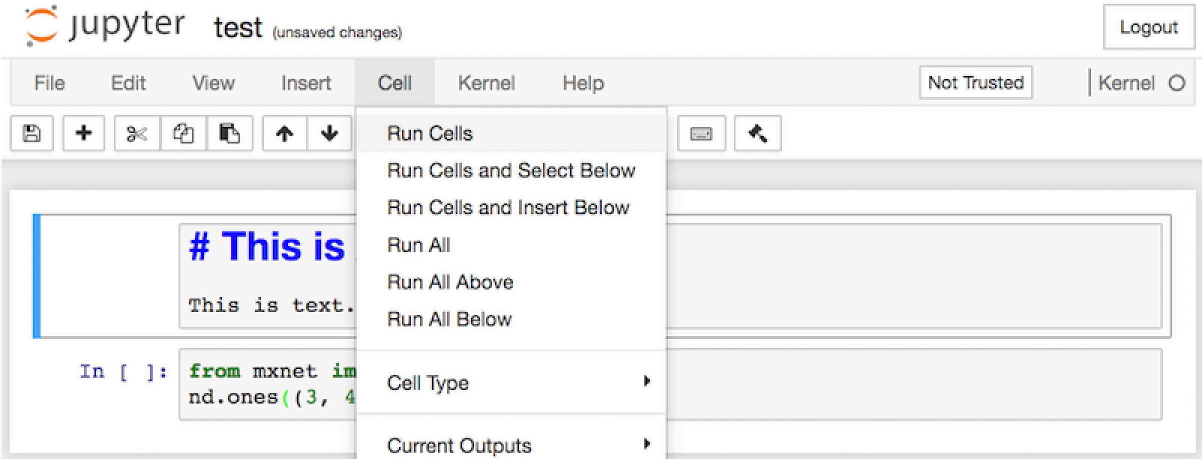


```
# This is A Title

This is text. Hello world.
```

```
In [ ]: from mxnet import nd
        nd.ones((3, 4))
```

Fig. 18.1.3: Edit the markdown cell.

As shown in Fig. 18.1.4, click "Cell" → "Run Cells" in the menu bar to run the edited cell.

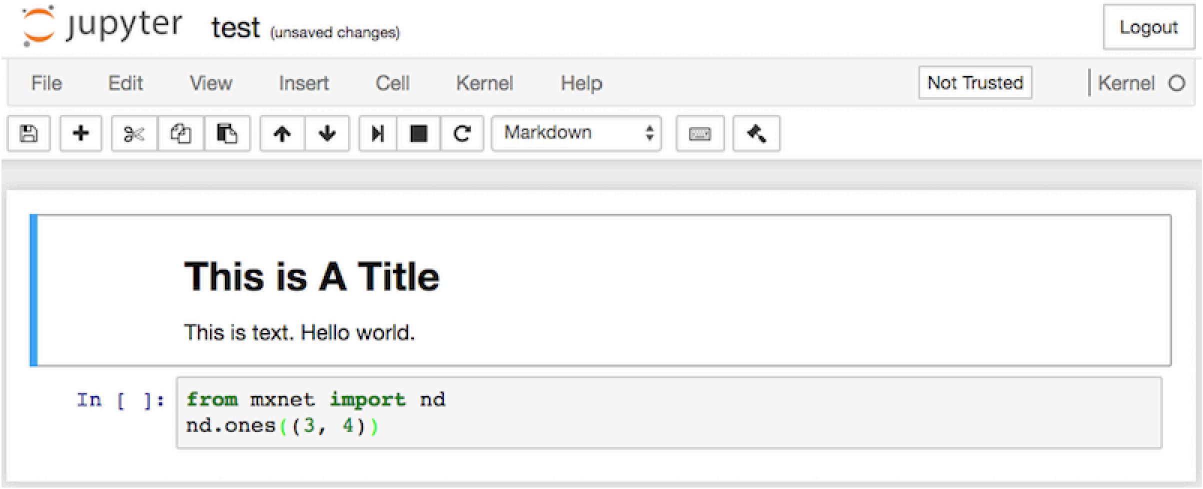


```
# This is
This is text.
```

```
from mxnet im
nd.ones((3, 4
```

Fig. 18.1.4: Run the cell.

After running, the markdown cell is as shown in Fig. 18.1.5.



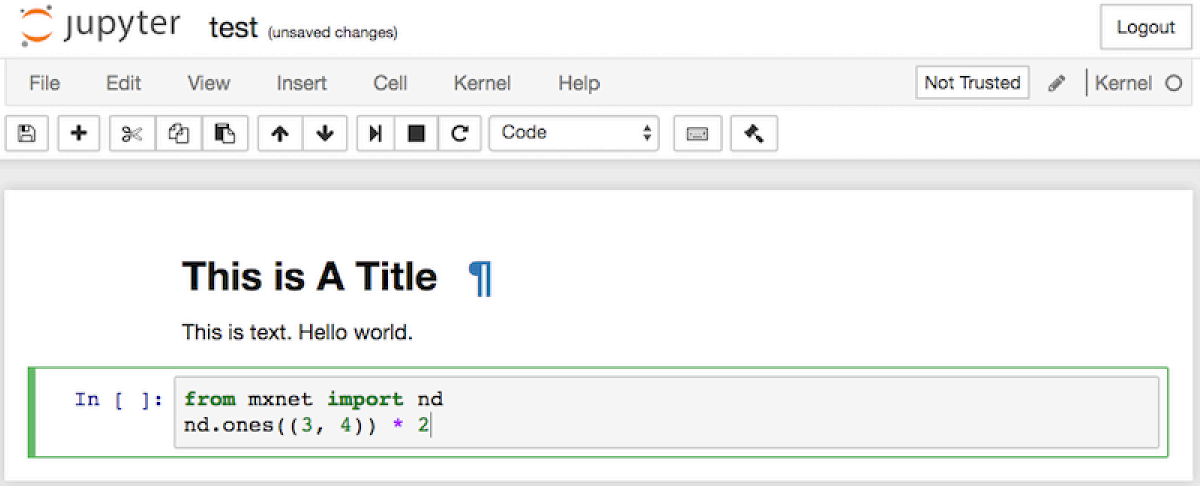
This is A Title

This is text. Hello world.

```
from mxnet import nd
nd.ones((3, 4))
```

Fig. 18.1.5: The markdown cell after editing.

Next, click on the code cell. Multiply the elements by 2 after the last line of code, as shown in Fig. 18.1.6.



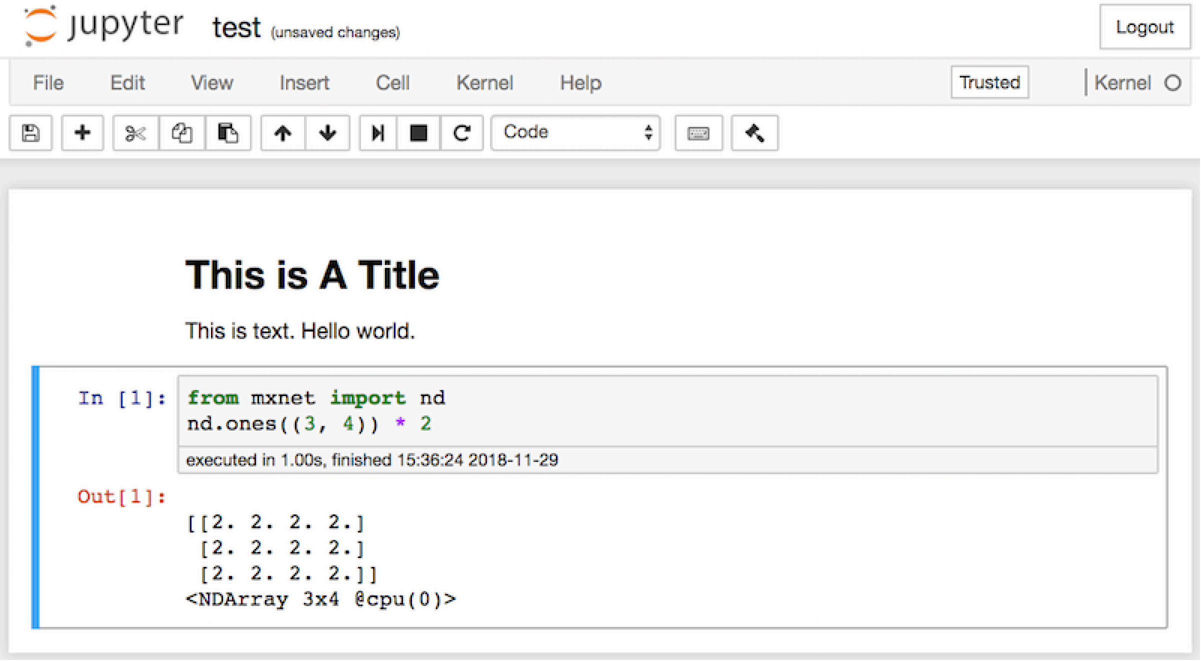
This is A Title ¶

This is text. Hello world.

```
In [ ]: from mxnet import nd
        nd.ones((3, 4)) * 2
```

Fig. 18.1.6: Edit the code cell.

You can also run the cell with a shortcut ("Ctrl + Enter" by default) and obtain the output result from Fig. 18.1.7.



This is A Title

This is text. Hello world.

```
In [1]: from mxnet import nd
        nd.ones((3, 4)) * 2
        executed in 1.00s, finished 15:36:24 2018-11-29

Out[1]:
        [[2. 2. 2. 2.]
         [2. 2. 2. 2.]
         [2. 2. 2. 2.]]
        <NDArray 3x4 @cpu(0)>
```

Fig. 18.1.7: Run the code cell to obtain the output.

When a notebook contains more cells, we can click "Kernel" → "Restart & Run All" in the menu bar to run all the cells in the entire notebook. By clicking "Help" → "Edit Keyboard Shortcuts" in the menu bar, you can edit the shortcuts according to your preferences.

18.1.2 Advanced Options

Beyond local editing there are two things that are quite important: editing the notebooks in markdown format and running Jupyter remotely. The latter matters when we want to run the code on a faster server. The former matters since Jupyter's native .ipnyb format stores a lot of auxiliary data that is not really specific to what is in the notebooks, mostly related to how and where the code is run. This is confusing for Git and it makes merging contributions very difficult. Fortunately there is an alternative—native editing in Markdown.

Markdown Files in Jupyter

If you wish to contribute to the content of this book, you need to modify the source file (md file, not ipynb file) on GitHub. Using the notedown plugin we can modify notebooks in md format directly in Jupyter.

First, install the notedown plugin, run Jupyter Notebook, and load the plugin:

```
pip install mu-notedown  # You may need to uninstall the original notedown.
jupyter notebook --NotebookApp.contents_manager_class='notedown.NotedownContentsManager'
```

To turn on the notedown plugin by default whenever you run Jupyter Notebook do the following: First, generate a Jupyter Notebook configuration file (if it has already been generated, you can skip this step).

```
jupyter notebook --generate-config
```

Then, add the following line to the end of the Jupyter Notebook configuration file (for Linux/macOS, usually in the path `~/.jupyter/jupyter_notebook_config.py`):

```
c.NotebookApp.contents_manager_class = 'notedown.NotedownContentsManager'
```

After that, you only need to run the `jupyter notebook` command to turn on the notedown plugin by default.

Running Jupyter Notebook on a Remote Server

Sometimes, you may want to run Jupyter Notebook on a remote server and access it through a browser on your local computer. If Linux or MacOS is installed on your local machine (Windows can also support this function through third-party software such as PuTTY), you can use port forwarding:

```
ssh myserver -L 8888:localhost:8888
```

The above is the address of the remote server `myserver`. Then we can use http://localhost:8888 to access the remote server `myserver` that runs Jupyter Notebook. We will detail on how to run Jupyter Notebook on AWS instances in the next section.

Timing

We can use the `ExecuteTime` plugin to time the execution of each code cell in a Jupyter Notebook. Use the following commands to install the plugin:

```
pip install jupyter_contrib_nbextensions
jupyter contrib nbextension install --user
jupyter nbextension enable execute_time/ExecuteTime
```

Summary

- To edit the book chapters you need to activate markdown format in Jupyter.
- You can run servers remotely using port forwarding.

Exercises

1. Try to edit and run the code in this book locally.
2. Try to edit and run the code in this book *remotely* via port forwarding.
3. Measure $\mathbf{A}^\top\mathbf{B}$ vs. $\mathbf{AB}$ for two square matrices in $\mathbb{R}^{1024\times 1024}$. Which one is faster?

18.2 Using Amazon SageMaker

Many deep learning applications require significant amounts of computation. Your local machine might be too slow to solve these problems in a reasonable amount of time. Cloud computing services can give you access to more powerful computers to run the GPU intensive portions of this book. This tutorial will guide you through Amazon SageMaker: a service that allows you to be up and running notebooks easily.

18.2.1 Registering Account and Logging In

First, we need to register an account at https://aws.amazon.com/. We strongly encourage you to use two-factor authentication for additional security. Furthermore, it is a good idea to set up detailed billing and spending alerts to avoid any unexpected surprises if you forget to suspend your computers. Note that you will need a credit card. After logging into your AWS account, find "Sagemaker" (see Fig. 18.2.1) to go to the Sagemaker panel.

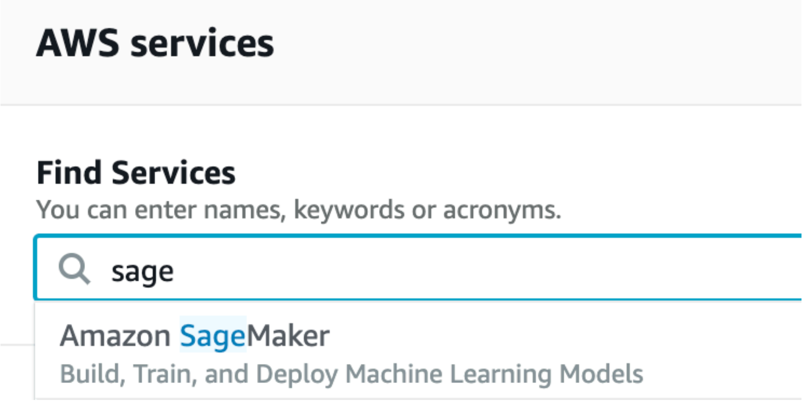

AWS services
Find Services
You can enter names, keywords or acronyms.
sage
Amazon SageMaker
Build, Train, and Deploy Machine Learning Models

Fig. 18.2.1: Open the Sagemaker console

18.2.2 Creating an SageMaker Instance

Next let's create a notebook instance (Fig. 18.2.2). During the creation, we can specify the instance name, type (Fig. 18.2.3), and notebook repository URL (Fig. 18.2.4). SageMaker provides multiple instance types[265] with different computation power and price. We used `ml.p3.2xlarge` here. It has a one Tesla V100 GPU and an 8-core CPU, which is powerful enough for most chapters. A Jupyter notebook version of this book that is modified to fit SageMaker is available at https://github.com/d2l-ai/d2l-en-sagemaker. We can specify this URL to let SageMaker clone this repository during instance creation.

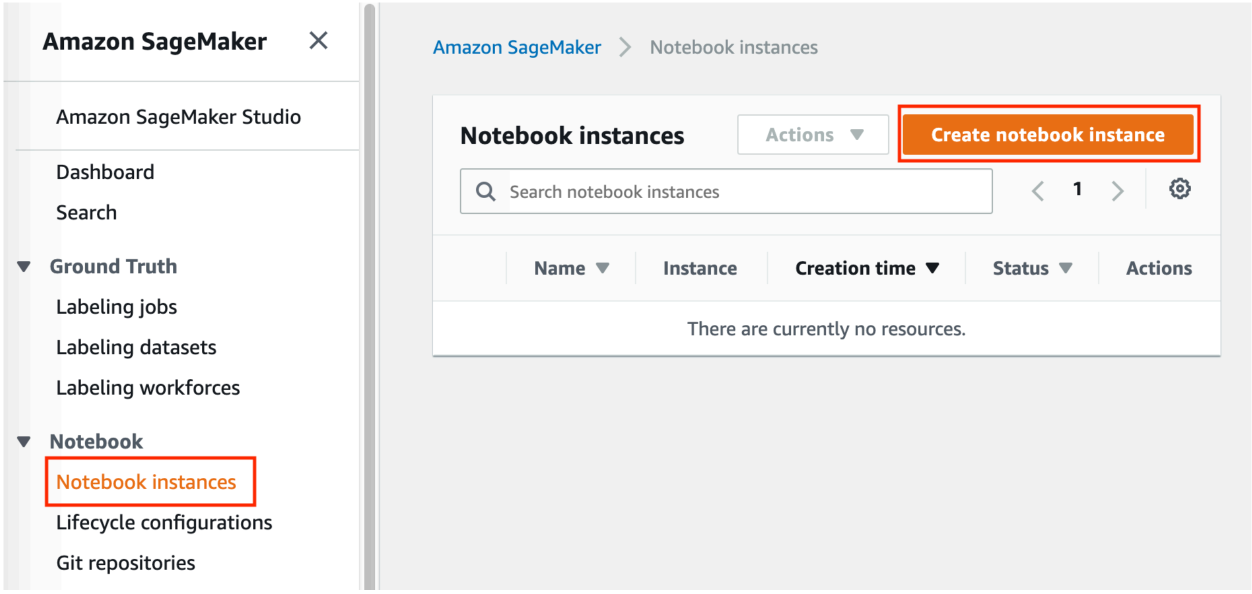
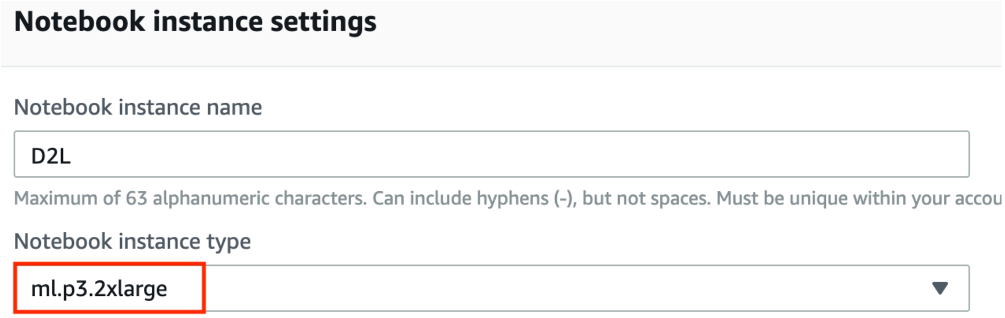


Fig. 18.2.2: Create a notebook instance

[265] https://aws.amazon.com/sagemaker/pricing/instance-types/



Notebook instance settings

Notebook instance name

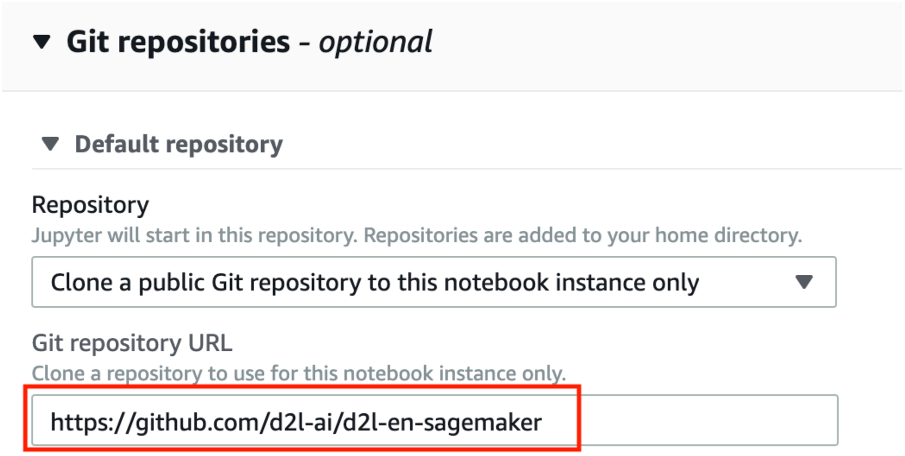
D2L

Maximum of 63 alphanumeric characters. Can include hyphens (-), but not spaces. Must be unique within your accou

Notebook instance type

ml.p3.2xlarge ▼

Fig. 18.2.3: Select instance type



▼ **Git repositories** - *optional*

▼ **Default repository**

Repository

Jupyter will start in this repository. Repositories are added to your home directory.

Clone a public Git repository to this notebook instance only ▼

Git repository URL

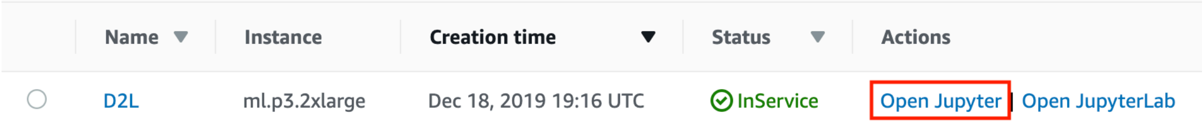
Clone a repository to use for this notebook instance only.

https://github.com/d2l-ai/d2l-en-sagemaker

Fig. 18.2.4: Specify the notebook repository.

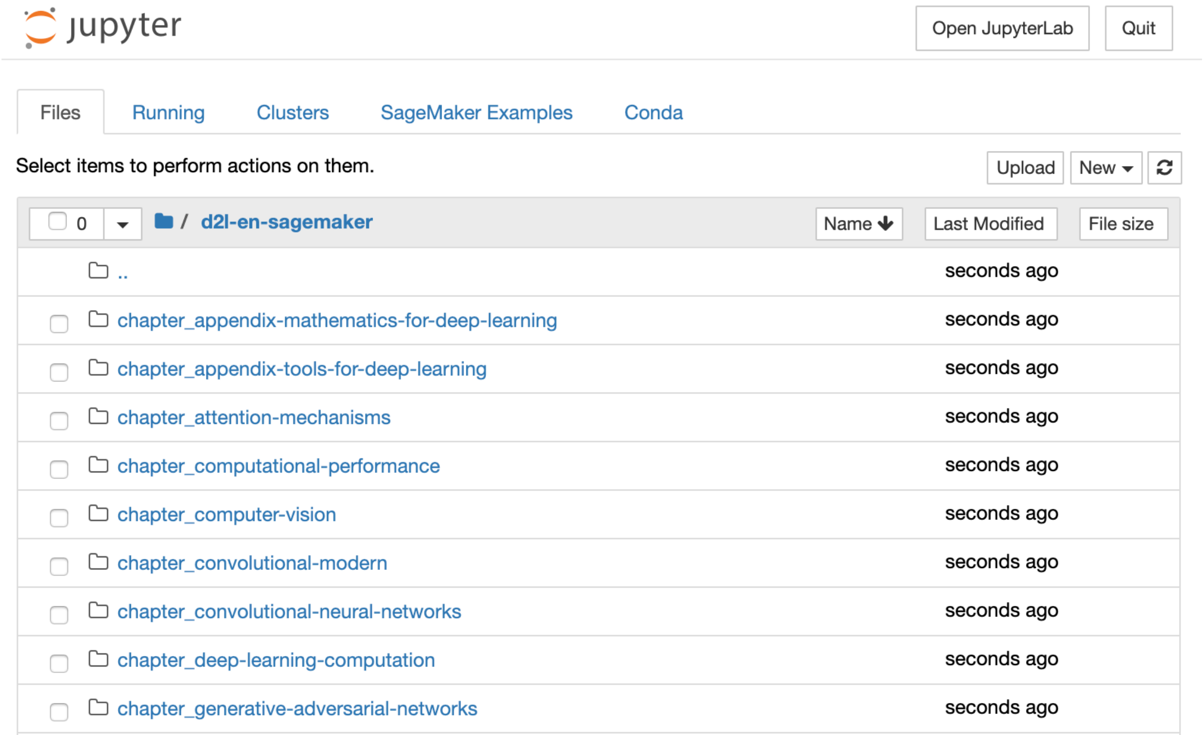
18.2.3 Running and Stopping an Instance

You may need to wait a few minutes before the instance is ready. Then you can click "Open Jupyter" link (Fig. 18.2.5) to navigate to the Jupyter server running on this instance (Fig. 18.2.6). The usage is similar to a normal Jupyter server running locally (Section 18.1). After finished your work, don't forgot to stop the instance to avoid further charging.



	Name ▼	Instance	Creation time ▼	Status ▼	Actions
○	D2L	ml.p3.2xlarge	Dec 18, 2019 19:16 UTC	✓ InService	Open Jupyter | Open JupyterLab

Fig. 18.2.5: Open Jupyter on the created instance.

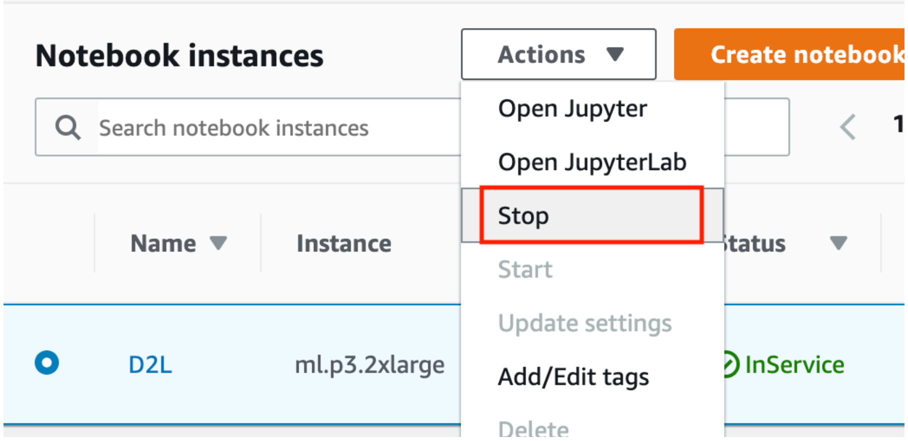


Select items to perform actions on them.

/ d2l-en-sagemaker

Name	Last Modified
..	seconds ago
chapter_appendix-mathematics-for-deep-learning	seconds ago
chapter_appendix-tools-for-deep-learning	seconds ago
chapter_attention-mechanisms	seconds ago
chapter_computational-performance	seconds ago
chapter_computer-vision	seconds ago
chapter_convolutional-modern	seconds ago
chapter_convolutional-neural-networks	seconds ago
chapter_deep-learning-computation	seconds ago
chapter_generative-adversarial-networks	seconds ago

Fig. 18.2.6: The Jupyter server running on the SageMaker instance.



	Name	Instance	tatus
●	D2L	ml.p3.2xlarge	InService

Fig. 18.2.7: Stop your instance.

18.2.4 Updating Notebooks

We will regularly update the notebooks in the d2l-ai/d2l-en-sagemaker[266]. You can simply `git pull` to update to the latest version. To do so, first open a terminal (Fig. 18.2.8).

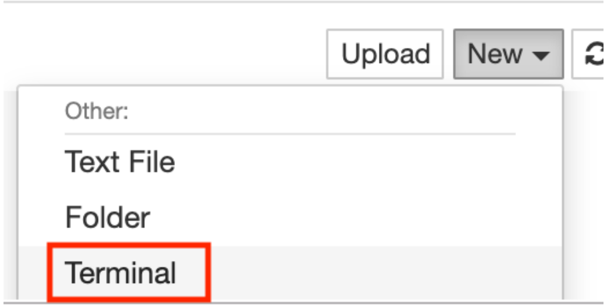


Fig. 18.2.8: Specify the notebook repository.

You may want to commit your local changes first before pulling the updates. Or you can simply ignore all your changes by `git reset --hard`. You can copy paste the following codes in the terminal to do so:

```
cd SageMaker/d2l-en-sagemaker/
git reset --hard
git pull
```

Summary

- Cloud computing services offer a wide variety of GPU servers.
- You can launch and stop a Jupyter server through Amazon SageMaker easily.

18.3 Using AWS EC2 Instances

In this section, we will show you how to install all libraries on a raw Linux machine. Remember that in Section 18.2 we discussed how to use Amazon SagaMaker, while building an instance by yourself costs less on AWS. The walkthrough includes a number of steps:

1. Request for a GPU Linux instance from AWS EC2.
2. Optionally: install CUDA or use an AMI with CUDA preinstalled.
3. Set up the corresponding MXNet GPU version.

This process applies to other instances (and other clouds), too, albeit with some minor modifications. Before going forward, you need to create an AWS account, see Section 18.2 for more details.

[266] https://github.com/d2l-ai/d2l-en-sagemaker

18.3.1 Creating and Running an EC2 Instance

After logging into your AWS account, click "EC2" (marked by the red box in Fig. 18.3.1) to go to the EC2 panel.

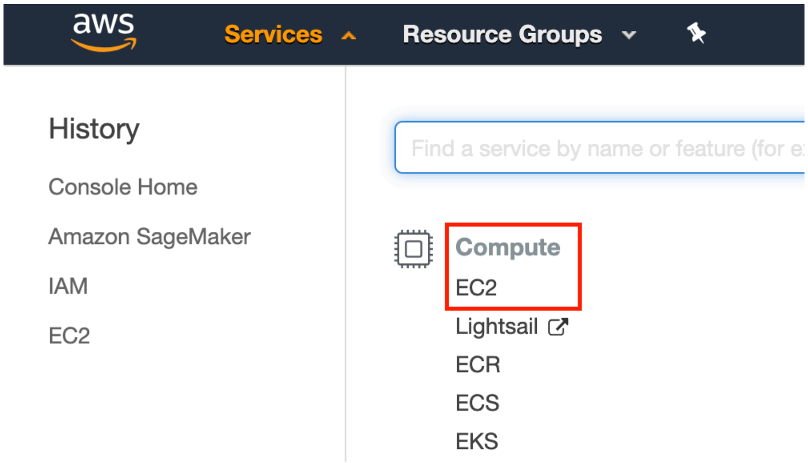
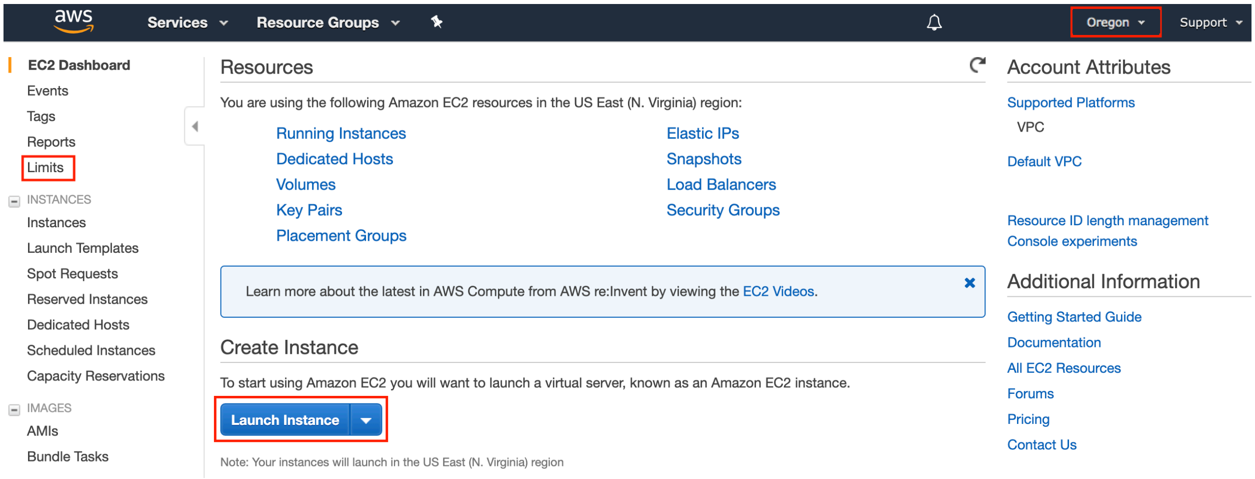


Fig. 18.3.1: Open the EC2 console.

Fig. 18.3.2 shows the EC2 panel with sensitive account information greyed out.



You are using the following Amazon EC2 resources in the US East (N. Virginia) region:

Learn more about the latest in AWS Compute from AWS re:Invent by viewing the EC2 Videos.

To start using Amazon EC2 you will want to launch a virtual server, known as an Amazon EC2 instance.

Note: Your instances will launch in the US East (N. Virginia) region

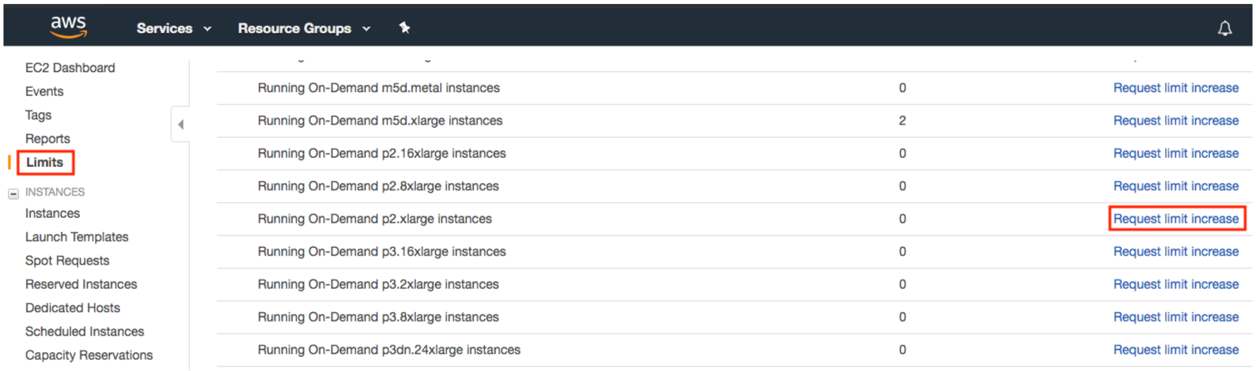
Fig. 18.3.2: EC2 panel.

Presetting Location

Select a nearby data center to reduce latency, *e.g.*, "Oregon". (marked by the red box in the top-right of Fig. 18.3.2) If you are located in China you can select a nearby Asia Pacific region, such as Seoul or Tokyo. Please note that some data centers may not have GPU instances.

Increasing Limits

Before choosing an instance, check if there are quantity restrictions by clicking the "Limits" label in the bar on the left as shown in Fig. 18.3.2. Fig. 18.3.3 shows an example of such a limitation. The account currently cannot open "p2.xlarge" instance per region. If you need to open one or more instances, click on the "Request limit increase" link to apply for a higher instance quota. Generally, it takes one business day to process an application.



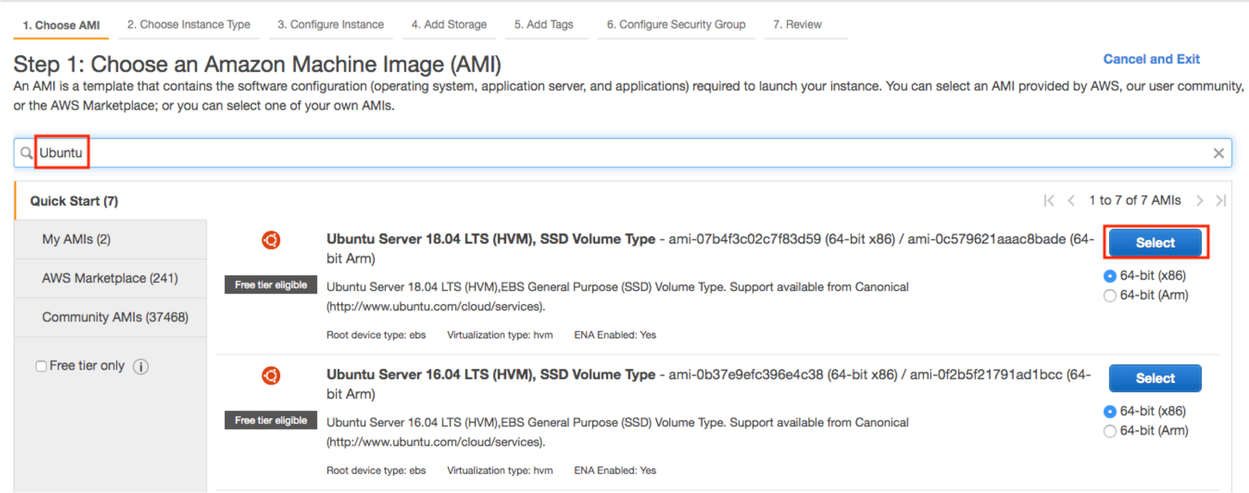
Running On-Demand m5d.metal instances	0	Request limit increase
Running On-Demand m5d.xlarge instances	2	Request limit increase
Running On-Demand p2.16xlarge instances	0	Request limit increase
Running On-Demand p2.8xlarge instances	0	Request limit increase
Running On-Demand p2.xlarge instances	0	Request limit increase
Running On-Demand p3.16xlarge instances	0	Request limit increase
Running On-Demand p3.2xlarge instances	0	Request limit increase
Running On-Demand p3.8xlarge instances	0	Request limit increase
Running On-Demand p3dn.24xlarge instances	0	Request limit increase

Fig. 18.3.3: Instance quantity restrictions.

Launching Instance

Next, click the "Launch Instance" button marked by the red box in Fig. 18.3.2 to launch your instance.

We begin by selecting a suitable AMI (AWS Machine Image). Enter "Ubuntu" in the search box (marked by the red box in Fig. 18.3.4):



Step 1: Choose an Amazon Machine Image (AMI)

An AMI is a template that contains the software configuration (operating system, application server, and applications) required to launch your instance. You can select an AMI provided by AWS, our user community, or the AWS Marketplace; or you can select one of your own AMIs.

Ubuntu

□ Free tier only

Ubuntu Server 18.04 LTS (HVM), SSD Volume Type - ami-07b4f3c02c7f83d59 (64-bit x86) / ami-0c579621aaac8bade (64-bit Arm)

Ubuntu Server 18.04 LTS (HVM),EBS General Purpose (SSD) Volume Type. Support available from Canonical (http://www.ubuntu.com/cloud/services).

Root device type: ebs Virtualization type: hvm ENA Enabled: Yes

● 64-bit (x86)
○ 64-bit (Arm)

Ubuntu Server 16.04 LTS (HVM), SSD Volume Type - ami-0b37e9efc396e4c38 (64-bit x86) / ami-0f2b5f21791ad1bcc (64-bit Arm)

Ubuntu Server 16.04 LTS (HVM),EBS General Purpose (SSD) Volume Type. Support available from Canonical (http://www.ubuntu.com/cloud/services).

Root device type: ebs Virtualization type: hvm ENA Enabled: Yes

● 64-bit (x86)
○ 64-bit (Arm)

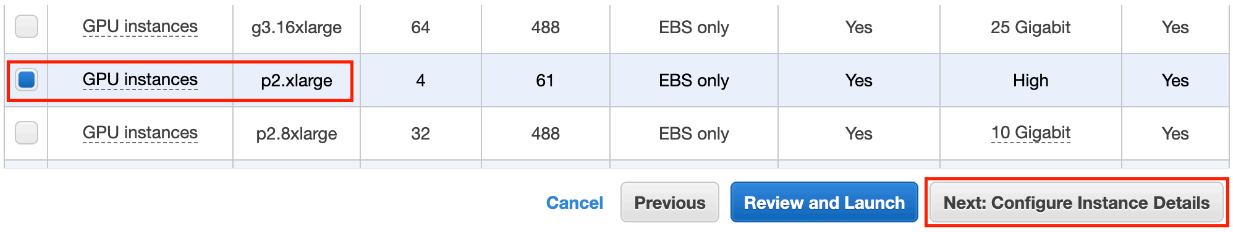
Fig. 18.3.4: Choose an operating system.

EC2 provides many different instance configurations to choose from. This can sometimes feel overwhelming to a beginner. Here's a table of suitable machines:

Name	GPU	Notes
g2	Grid K520	ancient
p2	Kepler K80	old but often cheap as spot
g3	Maxwell M60	good trade-off
p3	Volta V100	high performance for FP16
g4	Turing T4	inference optimized FP16/INT8

All the above servers come in multiple flavors indicating the number of GPUs used. For example, a p2.xlarge has 1 GPU and a p2.16xlarge has 16 GPUs and more memory. For more details see *e.g.,* the AWS EC2 documentation[267] or a summary page[268]. For the purpose of illustration, a p2.xlarge will suffice (marked in red box of Fig. 18.3.5).

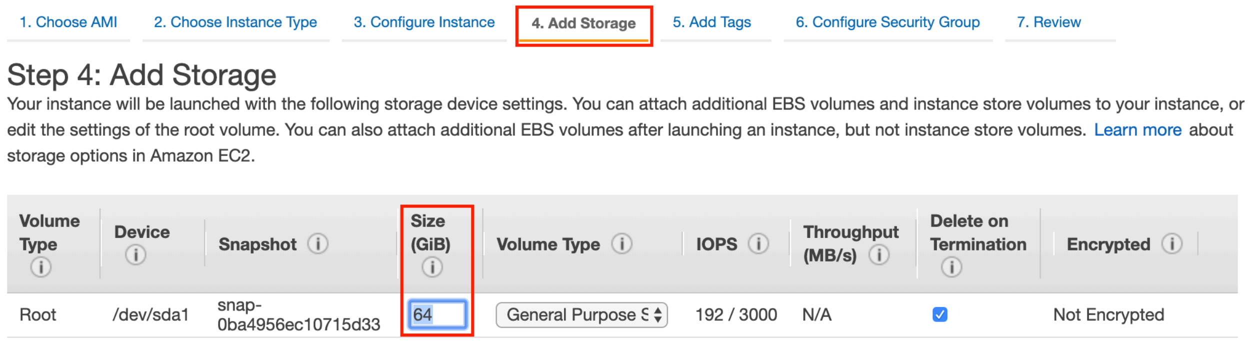
Note: you must use a GPU enabled instance with suitable drivers and a version of MXNet that is GPU enabled. Otherwise you will not see any benefit from using GPUs.



☐	GPU instances	g3.16xlarge	64	488	EBS only	Yes	25 Gigabit	Yes
☑	GPU instances	p2.xlarge	4	61	EBS only	Yes	High	Yes
☐	GPU instances	p2.8xlarge	32	488	EBS only	Yes	10 Gigabit	Yes

Fig. 18.3.5: Choose an instance.

So far, we have finished the first two of seven steps for launching an EC2 instance, as shown on the top of Fig. 18.3.6. In this example, we keep the default configurations for the steps "3. Configure Instance", "5. Add Tags", and "6. Configure Security Group". Tap on "4. Add Storage" and increase the default hard disk size to 64 GB (marked in red box of Fig. 18.3.6). Note that CUDA by itself already takes up 4GB.



Step 4: Add Storage

Your instance will be launched with the following storage device settings. You can attach additional EBS volumes and instance store volumes to your instance, or edit the settings of the root volume. You can also attach additional EBS volumes after launching an instance, but not instance store volumes. Learn more about storage options in Amazon EC2.

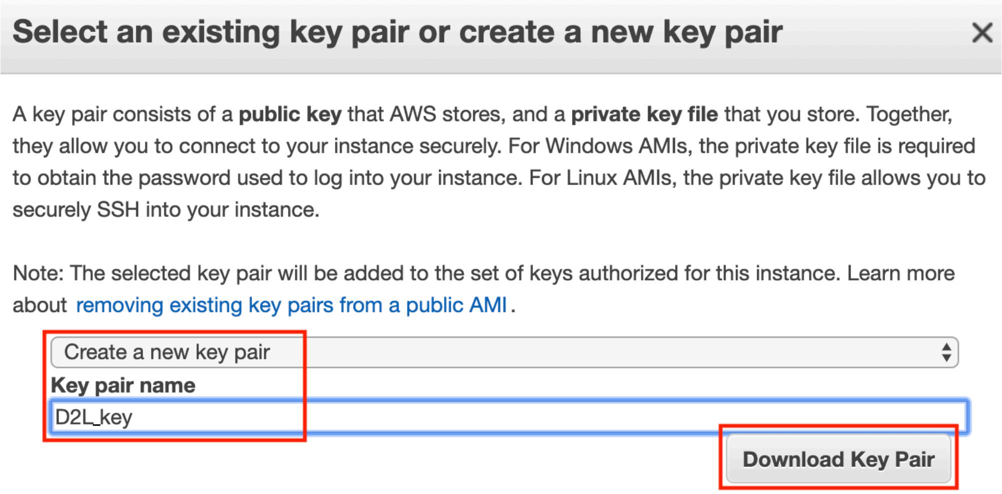
Volume Type ⓘ	Device ⓘ	Snapshot ⓘ	Size (GiB) ⓘ	Volume Type ⓘ	IOPS ⓘ	Throughput (MB/s) ⓘ	Delete on Termination ⓘ	Encrypted ⓘ
Root	/dev/sda1	snap-0ba4956ec10715d33	64	General Purpose S	192 / 3000	N/A	☑	Not Encrypted

Fig. 18.3.6: Modify instance hard disk size.

Finally, go to "7. Review" and click "Launch" to launch the configured instance. The system will now prompt you to select the key pair used to access the instance. If you do not have a key pair, select "Create a new key pair" in the first drop-down menu in Fig. 18.3.7 to generate a key pair. Subsequently, you can select "Choose an existing key pair" for this menu and then select the previously generated key pair. Click "Launch Instances" to launch the created instance.

[267] https://aws.amazon.com/ec2/instance-types/

[268] https://www.ec2instances.info



Select an existing key pair or create a new key pair

A key pair consists of a **public key** that AWS stores, and a **private key file** that you store. Together, they allow you to connect to your instance securely. For Windows AMIs, the private key file is required to obtain the password used to log into your instance. For Linux AMIs, the private key file allows you to securely SSH into your instance.

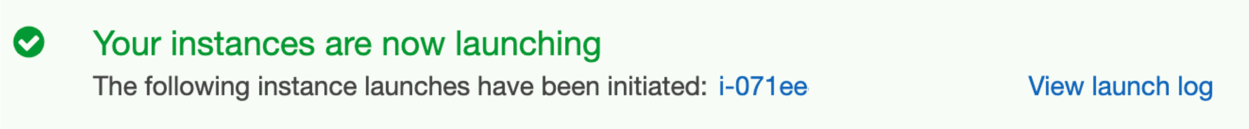
Note: The selected key pair will be added to the set of keys authorized for this instance. Learn more about removing existing key pairs from a public AMI .

Key pair name

D2L_key

Fig. 18.3.7: Select a key pair.

Make sure that you download the keypair and store it in a safe location if you generated a new one. This is your only way to SSH into the server. Click the instance ID shown in Fig. 18.3.8 to view the status of this instance.



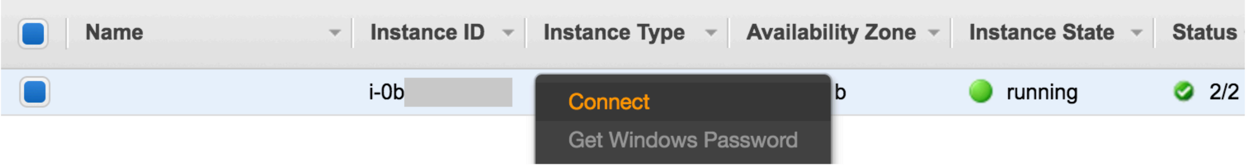
Your instances are now launching

The following instance launches have been initiated: i-071ee

Fig. 18.3.8: Click the instance ID.

Connecting Instance

As shown in Fig. 18.3.9, after the instance state turns green, right-click the instance and select `Connect` to view the instance access method.

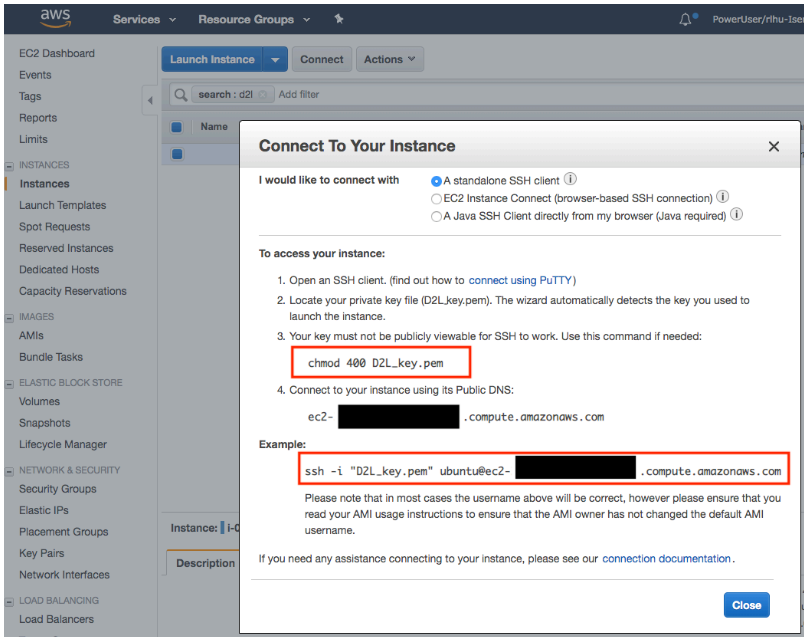


☐	Name	Instance ID	Instance Type	Availability Zone	Instance State	Status
☐		i-0b[redacted]		b	● running	✓ 2/2

Fig. 18.3.9: View instance access and startup method.

If this is a new key, it must not be publicly viewable for SSH to work. Go to the folder where you store `D2L_key.pem` (*e.g.*, Downloads folder) and make the key to be not publicly viewable.

```
cd /Downloads  ## if D2L_key.pem is stored in Downloads folder
chmod 400 D2L_key.pem
```



Connect To Your Instance ×

I would like to connect with
● A standalone SSH client
○ EC2 Instance Connect (browser-based SSH connection)
○ A Java SSH Client directly from my browser (Java required)

To access your instance:

1. Open an SSH client. (find out how to connect using PuTTY)
2. Locate your private key file (D2L_key.pem). The wizard automatically detects the key you used to launch the instance.
3. Your key must not be publicly viewable for SSH to work. Use this command if needed:

```
chmod 400 D2L_key.pem
```

4. Connect to your instance using its Public DNS:

```
ec2-[redacted].compute.amazonaws.com
```

Example:

```
ssh -i "D2L_key.pem" ubuntu@ec2-[redacted].compute.amazonaws.com
```

Please note that in most cases the username above will be correct, however please ensure that you read your AMI usage instructions to ensure that the AMI owner has not changed the default AMI username.

If you need any assistance connecting to your instance, please see our connection documentation.

Close

Fig. 18.3.10: View instance access and startup method.

Now, copy the ssh command in the lower red box of Fig. 18.3.10 and paste onto the command line:

```
ssh -i "D2L_key.pem" ubuntu@ec2-xx-xxx-xxx-xxx.y.compute.amazonaws.com
```

When the command line prompts "Are you sure you want to continue connecting (yes/no)", enter "yes" and press Enter to log into the instance.

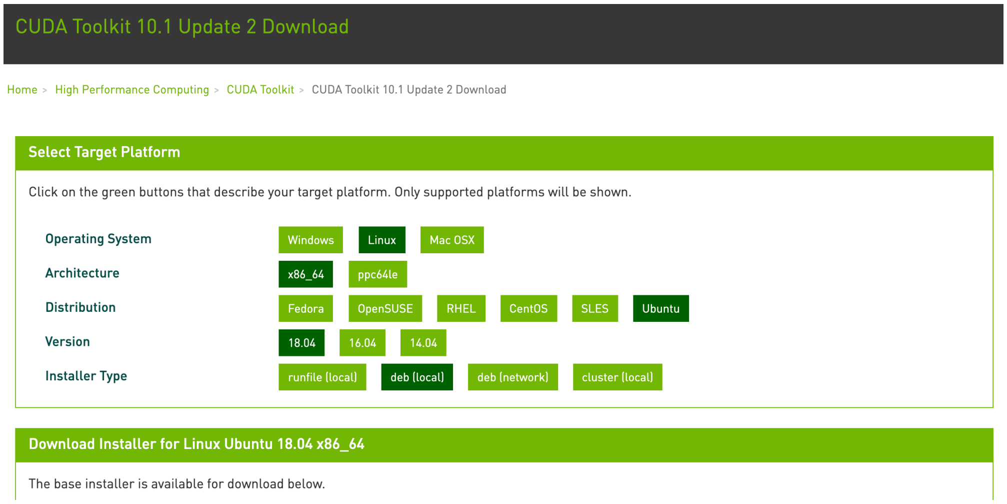
Your server is ready now.

18.3.2 Installing CUDA

Before installing CUDA, be sure to update the instance with the latest drivers.

```
sudo apt-get update && sudo apt-get install -y build-essential git libgfortran3
```

Here we download CUDA 10.1. Visit NVIDIA's official repository at (https://developer.nvidia.com/cuda-downloads) to find the download link of CUDA 10.1 as shown in Fig. 18.3.11.


CUDA Toolkit 10.1 Update 2 Download

Home > High Performance Computing > CUDA Toolkit > CUDA Toolkit 10.1 Update 2 Download

Select Target Platform

Click on the green buttons that describe your target platform. Only supported platforms will be shown.

Operating System	Windows Linux Mac OSX
Architecture	x86_64 ppc64le
Distribution	Fedora OpenSUSE RHEL CentOS SLES Ubuntu
Version	18.04 16.04 14.04
Installer Type	runfile (local) deb (local) deb (network) cluster (local)

Download Installer for Linux Ubuntu 18.04 x86_64

The base installer is available for download below.

Fig. 18.3.11: Find the CUDA 10.1 download address.

Copy the instructions and paste them into the terminal to install CUDA 10.1.

```
## Paste the copied link from CUDA website
wget https://developer.download.nvidia.com/compute/cuda/repos/ubuntu1804/x86_64/cuda-
↪ubuntu1804.pin
sudo mv cuda-ubuntu1804.pin /etc/apt/preferences.d/cuda-repository-pin-600
wget http://developer.download.nvidia.com/compute/cuda/10.1/Prod/local_installers/cuda-repo-
↪ubuntu1804-10-1-local-10.1.243-418.87.00_1.0-1_amd64.deb
sudo dpkg -i cuda-repo-ubuntu1804-10-1-local-10.1.243-418.87.00_1.0-1_amd64.deb
sudo apt-key add /var/cuda-repo-10-1-local-10.1.243-418.87.00/7fa2af80.pub
sudo apt-get update
sudo apt-get -y install cuda
```

After installing the program, run the following command to view the GPUs.

```
nvidia-smi
```

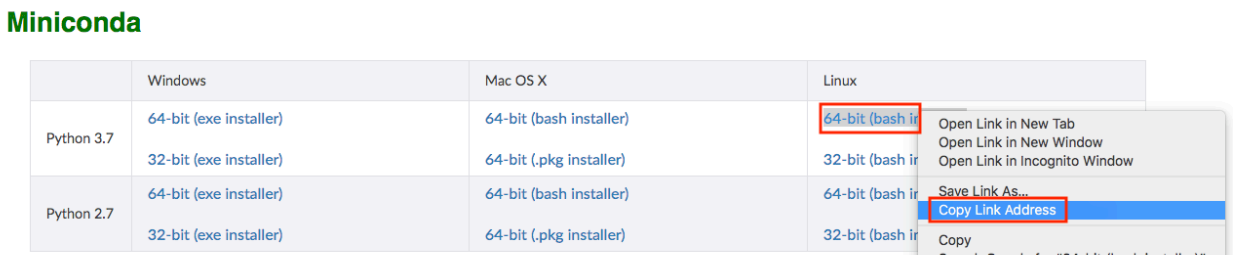
Finally, add CUDA to the library path to help other libraries find it.

```
echo "export LD_LIBRARY_PATH=\${LD_LIBRARY_PATH}:/usr/local/cuda/lib64" >> ~/.bashrc
```

18.3.3 Installing MXNet and Downloading the D2L Notebooks

First, to simplify the installation, you need to install Miniconda[269] for Linux. The download link and file name are subject to changes, so please go the Miniconda website and click "Copy Link Address" as shown in Fig. 18.3.12.

[269] https://conda.io/en/latest/miniconda.html



Miniconda

	Windows	Mac OS X	Linux
Python 3.7	64-bit (exe installer) 32-bit (exe installer)	64-bit (bash installer) 64-bit (.pkg installer)	64-bit (bash ir 32-bit (bash ir
Python 2.7	64-bit (exe installer) 32-bit (exe installer)	64-bit (bash installer) 64-bit (.pkg installer)	64-bit (bash ir 32-bit (bash ir

Fig. 18.3.12: Download Miniconda.

```
# The link and file name are subject to changes
wget https://repo.anaconda.com/miniconda/Miniconda3-latest-Linux-x86_64.sh
sh Miniconda3-latest-Linux-x86_64.sh -b
```

After Miniconda installation, run the following command to activate CUDA and conda.

```
~/miniconda3/bin/conda init
source ~/.bashrc
```

Next, download the code for this book.

```
sudo apt-get install unzip
mkdir d2l-en && cd d2l-en
curl https://d2l.ai/d2l-en-0.7.1.zip -o d2l-en.zip
unzip d2l-en.zip && rm d2l-en.zip
```

Then create the conda `d2l` environment and enter `y` to proceed with the installation.

```
conda create --name d2l -y
```

After creating the `d2l` environment, activate it and install `pip`.

```
conda activate d2l
conda install python=3.7 pip -y
```

Finally, install MXNet and the `d2l` package. The postfix `cu101` means that this is the CUDA 10.1 variant. For different versions, say only CUDA 10.0, you would want to choose `cu100` instead.

```
pip install mxnet-cu101==1.6.0b20191122
pip install d2l==0.11.1
```

You can test quickly whether everything went well as follows:

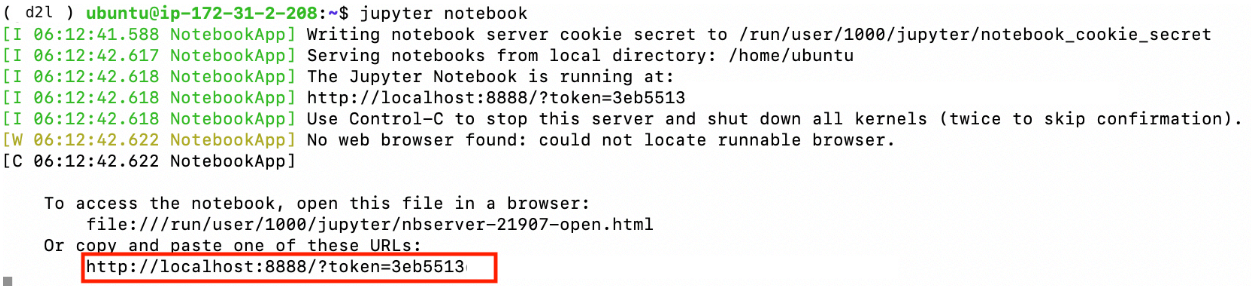
```
$ python
>>> from mxnet import np, npx
>>> np.zeros((1024, 1024), ctx=npx.gpu())
```

18.3.4 Running Jupyter

To run Jupyter remotely you need to use SSH port forwarding. After all, the server in the cloud does not have a monitor or keyboard. For this, log into your server from your desktop (or laptop) as follows.

```
# This command must be run in the local command line
ssh -i "/path/to/key.pem" ubuntu@ec2-xx-xxx-xxx-xxx.y.compute.amazonaws.com -L 8889:localhost:8888
conda activate d2l
jupyter notebook
```

Fig. 18.3.13 shows the possible output after you run Jupyter Notebook. The last row is the URL for port 8888.



```
( d2l ) ubuntu@ip-172-31-2-208:~$ jupyter notebook
[I 06:12:41.588 NotebookApp] Writing notebook server cookie secret to /run/user/1000/jupyter/notebook_cookie_secret
[I 06:12:42.617 NotebookApp] Serving notebooks from local directory: /home/ubuntu
[I 06:12:42.618 NotebookApp] The Jupyter Notebook is running at:
[I 06:12:42.618 NotebookApp] http://localhost:8888/?token=3eb5513
[I 06:12:42.618 NotebookApp] Use Control-C to stop this server and shut down all kernels (twice to skip confirmation).
[W 06:12:42.622 NotebookApp] No web browser found: could not locate runnable browser.
[C 06:12:42.622 NotebookApp]

    To access the notebook, open this file in a browser:
        file:///run/user/1000/jupyter/nbserver-21907-open.html
    Or copy and paste one of these URLs:
        http://localhost:8888/?token=3eb5513
```

Fig. 18.3.13: Output after running Jupyter Notebook. The last row is the URL for port 8888.

Since you used port forwarding to port 8889 you will need to replace the port number and use the secret as given by Jupyter when opening the URL in your local browser.

18.3.5 Closing Unused Instances

As cloud services are billed by the time of use, you should close instances that are not being used. Note that there are alternatives: "Stopping" an instance means that you will be able to start it again. This is akin to switching off the power for your regular server. However, stopped instances will still be billed a small amount for the hard disk space retained. "Terminate" deletes all data associated with it. This includes the disk, hence you cannot start it again. Only do this if you know that you will not need it in the future.

If you want to use the instance as a template for many more instances, right-click on the example in Figure 14.16 Fig. 18.3.9 and select "Image" → "Create" to create an image of the instance. Once this is complete, select "Instance State" → "Terminate" to terminate the instance. The next time you want to use this instance, you can follow the steps for creating and running an EC2 instance described in this section to create an instance based on the saved image. The only difference is that, in "1. Choose AMI" shown in Fig. 18.3.4, you must use the "My AMIs" option on the left to select your saved image. The created instance will retain the information stored on the image hard disk. For example, you will not have to reinstall CUDA and other runtime environments.

Summary

- You can launch and stop instances on demand without having to buy and build your own computer.
- You need to install suitable GPU drivers before you can use them.

Exercises

1. The cloud offers convenience, but it does not come cheap. Find out how to launch spot instances[270] to see how to reduce prices.
2. Experiment with different GPU servers. How fast are they?
3. Experiment with multi-GPU servers. How well can you scale things up?

18.4 Using Google Colab

We introduced in Section 18.2 and Section 18.4 for how to run this book on AWS. Another option is running on Google Colab[272], which provides free GPU if you have a Google account.

To run a section on Colab, you can simply click the `Colab` button on the right of the title Fig. 18.4.1. The first time you execute a code cell, you will receive a warning message saying it is from Github Fig. 18.4.2 and may steal your data. If you trust us, then click "RUN ANYWAY", then Colab will connect you to an instance to run this notebook. In particular, if GPU is needed, such as `d2l.try_gpu()`, then we will request Colab to connect to a GPU instance automatically.

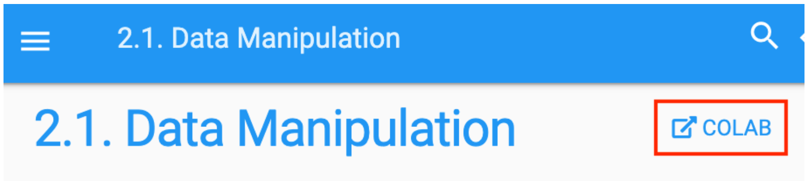
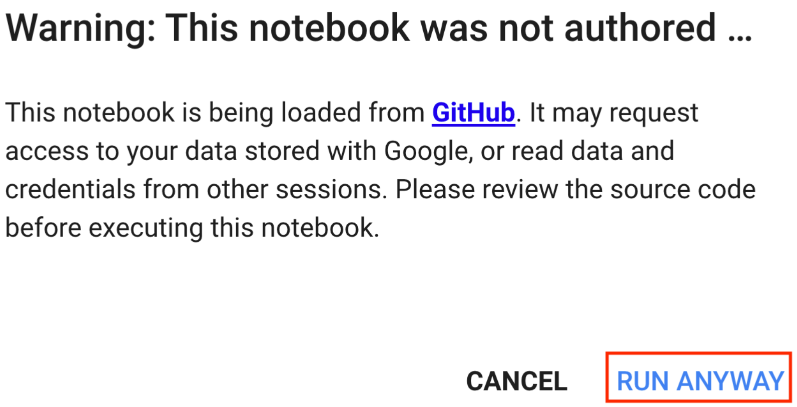


Fig. 18.4.1: Open a section on Colab

[270] https://aws.amazon.com/ec2/spot/
[272] https://colab.research.google.com/



Warning: This notebook was not authored …

This notebook is being loaded from GitHub. It may request access to your data stored with Google, or read data and credentials from other sessions. Please review the source code before executing this notebook.

Fig. 18.4.2: The warning message for running a section on Colab

One pitfall of Colab is that it's not straightforward to upload data. So you will get an error when trying to read data in the `../data` folder. You can manually download d2l-en.zip[273], unzip it and upload the `data` folder into your Google drive. We hope to make it smarter later.

Summary

- You can use Google Colab to run each section on GPUs freely.

18.5 Selecting Servers and GPUs

Deep learning training generally requires large amounts of computation. At present GPUs are the most cost-effective hardware accelerators for deep learning. In particular, compared with CPUs, GPUs are cheaper and offer higher performance, often by over an order of magnitude. Furthermore, a single server can support multiple GPUs, up to 8 for high end servers. More typical numbers are up to 4 GPUs for an engineering workstation, since heat, cooling and power requirements escalate quickly beyond what an office building can support. For larger deployments cloud computing, such as Amazon's P3[274] and G4[275] instances are a much more practical solution.

18.5.1 Selecting Servers

There is typically no need to purchase high-end CPUs with many threads since much of the computation occurs on the GPUs. That said, due to the Global Interpreter Lock (GIL) in Python single-thread performance of a CPU can matter in situations where we have 4-8 GPUs. All things equal this suggests that CPUs with a smaller number of cores but a higher clock frequency might be a more economical choice. E.g. when choosing between a 6 core 4GHz and an 8 core 3.5 GHz CPU, the former is much preferable, even though its aggregate speed is less. An important consideration is that GPUs use lots of power and thus dissipate lots of heat. This requires very good cooling and a large enough chassis to use the GPUs. Follow the guidelines below if possible:

[273] https://d2l.ai/d2l-en.zip

[274] https://aws.amazon.com/ec2/instance-types/p3/

[275] https://aws.amazon.com/blogs/aws/in-the-works-ec2-instances-g4-with-nvidia-t4-gpus/

1. **Power Supply**. GPUs use significant amounts of power. Budget with up to 350W per device (check for the *peak demand* of the graphics card rather than typical demand, since efficient code can use lots of energy. If your power supply is not up to the demand you will find that your system becomes unstable.
2. **Chassis Size**. GPUs are large and the auxiliary power connectors often need extra space. Also, large chassis are easier to cool.
3. **GPU Cooling**. If you have large numbers of GPUs you might want to invest in water cooling. Also, aim for *reference designs* even if they have fewer fans, since they are thin enough to allow for air intake between the devices. If you buy a multi-fan GPU it might be too thick to get enough air when installing multiple GPUs and you will run into thermal throttling.
4. **PCIe Slots**. Moving data to and from the GPU (and exchanging it between GPUs) requires lots of bandwidth. We recommend PCIe 3.0 slots with 16 lanes. If you mount multiple GPUs, be sure to carefully read the motherboard description to ensure that 16x bandwidth is still available when multiple GPUs are used at the same time and that you are getting PCIe 3.0 as opposed to PCIe 2.0 for the additional slots. Some motherboards downgrade to 8x or even 4x bandwidth with multiple GPUs installed. This is partly due to the number of PCIe lanes that the CPU offers.

In short, here are some recommendations for building a deep learning server:

- **Beginner**. Buy a low end GPU with low power consumption (cheap gaming GPUs suitable for deep learning use 150-200W). If you are lucky your current computer will support it.
- **1 GPU**. A low-end CPU with 4 cores will be plenty sufficient and most motherboards suffice. Aim for at least 32GB DRAM and invest into an SSD for local data access. A power supply with 600W should be sufficient. Buy a GPU with lots of fans.
- **2 GPUs**. A low-end CPU with 4-6 cores will suffice. Aim for 64GB DRAM and invest into an SSD. You will need in the order of 1000W for two high-end GPUs. In terms of mainboards, make sure that they have *two* PCIe 3.0 x16 slots. If you can, get a mainboard that has two free spaces (60mm spacing) between the PCIe 3.0 x16 slots for extra air. In this case, buy two GPUs with lots of fans.
- **4 GPUs**. Make sure that you buy a CPU with relatively fast single-thread speed (i.e., high clock frequency). You will probably need a CPU with a larger number of PCIe lanes, such as an AMD Threadripper. You will likely need relatively expensive mainboards to get 4 PCIe 3.0 x16 slots since they probably need a PLX to multiplex the PCIe lanes. Buy GPUs with reference design that are narrow and let air in between the GPUs. You need a 1600-2000W power supply and the outlet in your office might not support that. This server will probably run *loud and hot*. You do not want it under your desk. 128GB of DRAM is recommended. Get an SSD (1-2TB NVMe) for local storage and a bunch of hard disks in RAID configuration to store your data.
- **8 GPUs**. You need to buy a dedicated multi-GPU server chassis with multiple redundant power supplies (e.g., 2+1 for 1600W per power supply). This will require dual socket server CPUs, 256GB ECC DRAM, a fast network card (10GbE recommended), and you will need to check whether the servers support the *physical form factor* of the GPUs. Airflow and wiring placement differ significantly between consumer and server GPUs (e.g., RTX 2080 vs. Tesla V100). This means that you might not be able to install the consumer GPU in a server due to insufficient clearance for the power cable or lack of a suitable wiring harness (as one of the coauthors painfully discovered).

18.5.2 Selecting GPUs

At present, AMD and NVIDIA are the two main manufacturers of dedicated GPUs. NVIDIA was the first to enter the deep learning field and provides better support for deep learning frameworks via CUDA. Therefore, most buyers choose NVIDIA GPUs.

NVIDIA provides two types of GPUs, targeting individual users (e.g., via the GTX and RTX series) and enterprise users (via its Tesla series). The two types of GPUs provide comparable compute power. However, the enterprise user GPUs generally use (passive) forced cooling, more memory, and ECC (error correcting) memory. These GPUs are more suitable for data centers and usually cost ten times more than consumer GPUs.

If you are a large company with 100+ servers you should consider the NVIDIA Tesla series or alternatively use GPU servers in the cloud. For a lab or a small to medium company with 10+ servers the NVIDIA RTX series is likely most cost effective. You can buy preconfigured servers with Supermicro or Asus chassis that hold 4-8 GPUs efficiently.

GPU vendors typically release a new generation every 1-2 years, such as the GTX 1000 (Pascal) series released in 2017 and the RTX 2000 (Turing) series released in 2019. Each series offers several different models that provide different performance levels. GPU performance is primarily a combination of the following three parameters:

1. **Compute power**. Generally we look for 32-bit floating-point compute power. 16-bit floating point training (FP16) is also entering the mainstream. If you are only interested in prediction, you can also use 8-bit integer. The latest generation of Turing GPUs offers 4-bit acceleration. Unfortunately at present the algorithms to train low-precision networks are not widespread yet.

2. **Memory size**. As your models become larger or the batches used during training grow bigger, you will need more GPU memory. Check for HBM2 (High Bandwidth Memory) vs. GDDR6 (Graphics DDR) memory. HBM2 is faster but much more expensive.

3. **Memory bandwidth**. You can only get the most out of your compute power when you have sufficient memory bandwidth. Loook for wide memory buses if using GDDR6.

For most users, it is enough to look at compute power. Note that many GPUs offer different types of acceleration. E.g. NVIDIA's TensorCores accelerate a subset of operators by 5x. Ensure that your libraries support this. The GPU memory should be no less than 4 GB (8GB is much better). Try to avoid using the GPU also for displaying a GUI (use the built-in graphics instead). If you cannot avoid it, add an extra 2GB of RAM for safety.

Fig. 18.5.1 compares the 32-bit floating-point compute power and price of the various GTX 900, GTX 1000 and RTX 2000 series models. The prices are the suggested prices found on Wikipedia.

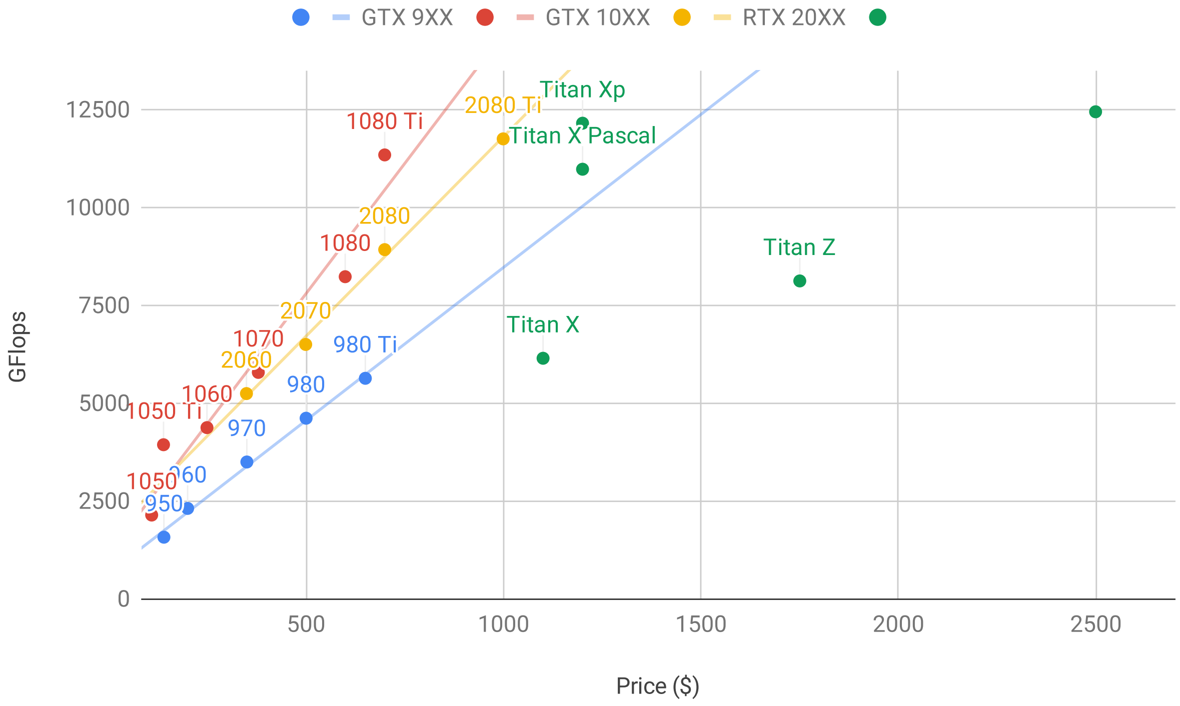


Fig. 18.5.1: Floating-point compute power and price comparison.

We can see a number of things:

1. Within each series, price and performance are roughly proportional. Titan models command a significant premium for the benefit of larger amounts of GPU memory. However, the newer models offer better cost effectiveness, as can be seen by comparing the 980 Ti and 1080 Ti. The price does not appear to improve much for the RTX 2000 series. However, this is due to the fact that they offer far superior low precision performance (FP16, INT8 and INT4).
2. The performance to cost ratio of the GTX 1000 series is about two times greater than the 900 series.
3. For the RTX 2000 series the price is an *affine* function of the price.

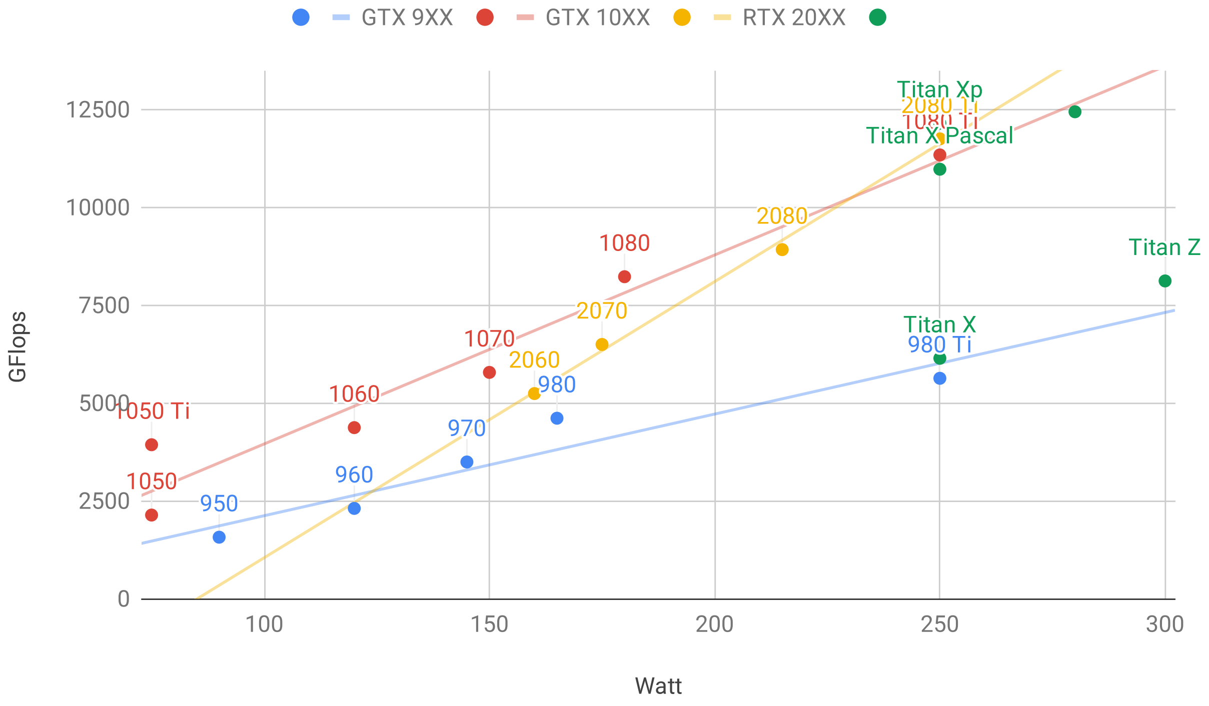


Fig. 18.5.2: Floating-point compute power and energy consumption.

Fig. 18.5.2 shows how energy consumption scales mostly linearly with the amount of computation. Second, later generations are more efficient. This seems to be contradicted by the graph corresponding to the RTX 2000 series. However, this is a consequence of the TensorCores which draw disproportionately much energy.

Summary

- Watch out for power, PCIe bus lanes, CPU single thread speed and cooling when building a server.
- You should purchase the latest GPU generation if possible.
- Use the cloud for large deployments.
- High density servers may not be compatible with all GPUs. Check the mechanical and cooling specifications before you buy.
- Use FP16 or lower precision for high efficiency.

18.6 Contributing to This Book

Contributions by readers[277] help us improve this book. If you find a typo, an outdated link, something where you think we missed a citation, where the code does not look elegant or where an explanation is unclear, please contribute back and help us help our readers. While in regular books the delay between print runs (and thus between typo corrections) can be measured in years, it typically takes hours to days to incorporate an improvement in this book. This is all possible due to version control and continuous integration testing. To do so you need to install Git and submit a pull request[278] to the GitHub repository. When your pull request is merged into the code repository by the author, you will become a contributor. In a nutshell the process works as described in Fig. 18.6.1.

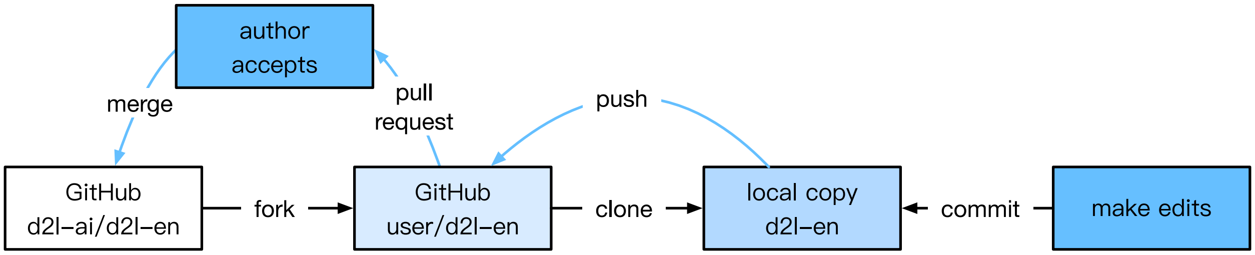


Fig. 18.6.1: Contributing to the book.

18.6.1 From Reader to Contributor in 6 Steps

We will walk you through the steps in detail. If you are already familiar with Git you can skip this section. For concreteness we assume that the contributor's user name is "astonzhang".

Installing Git

The Git open source book describes how to install Git[279]. This typically works via `apt install git` on Ubuntu Linux, by installing the Xcode developer tools on macOS, or by using GitHub's desktop client[280]. If you do not have a GitHub account, you need to sign up for one.

Logging in to GitHub

Enter the address[281] of the book's code repository in your browser. Click on the `Fork` button in the red box at the top-right of Fig. 18.6.2, to make a copy of the repository of this book. This is now *your copy* and you can change it any way you want.

[277] https://github.com/d2l-ai/d2l-en/graphs/contributors
[278] https://github.com/d2l-ai/d2l-en/pulls
[279] https://git-scm.com/book/zh/v2
[280] https://desktop.github.com
[281] https://github.com/d2l-ai/d2l-en/

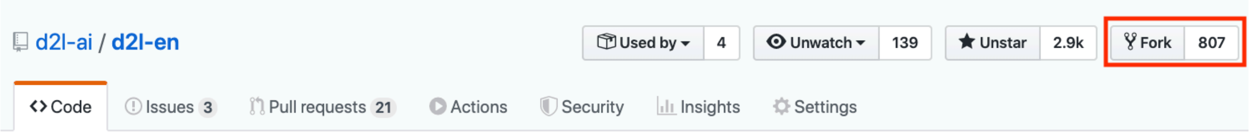


Fig. 18.6.2: The code repository page.

Now, the code repository of this book will be copied to your username, such as `astonzhang/d2l-en` shown at the top-left of the screenshot Fig. 18.6.3.

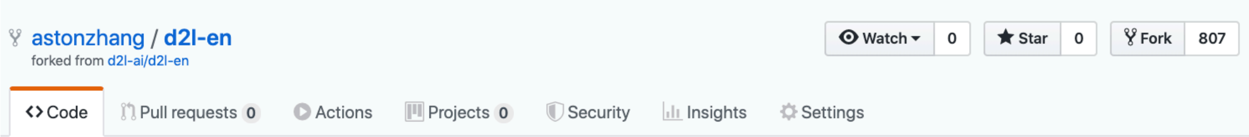
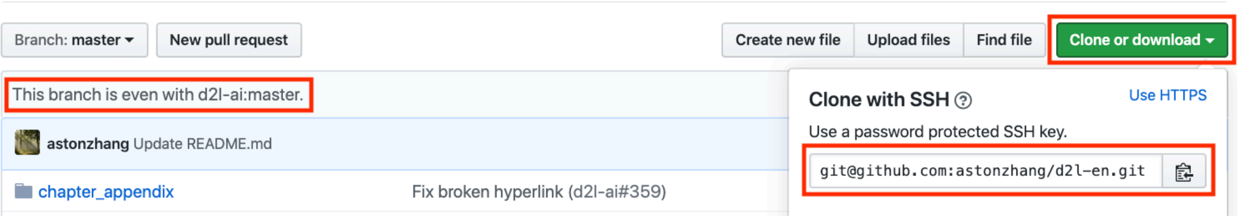


Fig. 18.6.3: Copy the code repository.

Cloning the Repository

To clone the repository (i.e., to make a local copy) we need to get its repository address. The green button in Fig. 18.6.4 displays this. Make sure that your local copy is up to date with the main repository if you decide to keep this fork around for longer. For now simply follow the instructions in Chapter to get started. The main difference is that you are now downloading *your own fork* of the repository.



This branch is even with d2l-ai:master.

Use a password protected SSH key.

```
git@github.com:astonzhang/d2l-en.git
```

Fig. 18.6.4: Git clone.

```
# Replace your_github_username with your GitHub username
git clone https://github.com/your_github_username/d2l-en.git
```

Editing the Book and Push

Now it is time to edit the book. It is best to edit the notebooks in Jupyter following instructions in Section 18.1. Make the changes and check that they are OK. Assume we have modified a typo in the file `~/d2l-en/chapter_appendix_tools/how-to-contribute.md`. You can then check which files you have changed:

At this point Git will prompt that the `chapter_appendix_tools/how-to-contribute.md` file has been modified.

```
mylaptop:d2l-en me$ git status
On branch master
Your branch is up-to-date with 'origin/master'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   chapter_appendix_tools/how-to-contribute.md
```

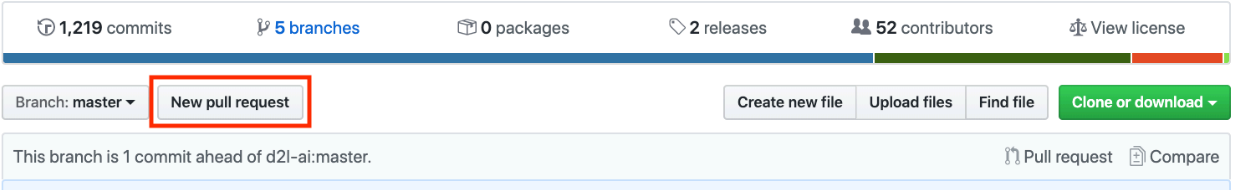
After confirming that this is what you want, execute the following command:

```
git add chapter_appendix_tools/how-to-contribute.md
git commit -m 'fix typo in git documentation'
git push
```

The changed code will then be in your personal fork of the repository. To request the addition of your change, you have to create a pull request for the official repository of the book.

Pull Request

As shown in Fig. 18.6.5, go to your fork of the repository on GitHub and select "New pull request". This will open up a screen that shows you the changes between your edits and what is current in the main repository of the book.

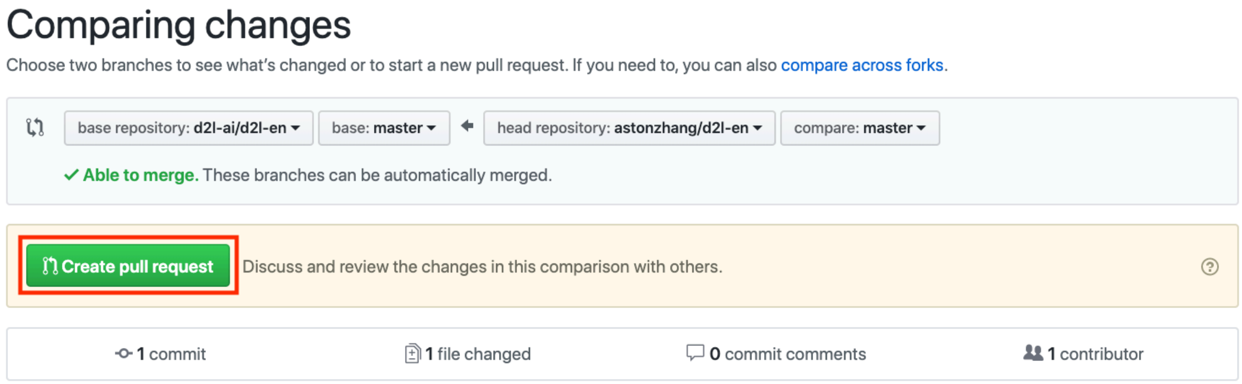


This branch is 1 commit ahead of d2l-ai:master.

Fig. 18.6.5: Pull Request.

Submitting Pull Request

Finally, submit a pull request by clicking the button as shown in Fig. 18.6.6. Make sure to describe the changes you have made in the pull request. This will make it easier for the authors to review it and to merge it with the book. Depending on the changes, this might get accepted right away, rejected, or more likely, you will get some feedback on the changes. Once you have incorporated them, you are good to go.



Comparing changes

Choose two branches to see what's changed or to start a new pull request. If you need to, you can also compare across forks.

base repository: d2l-ai/d2l-en ▾ base: master ▾ ← head repository: astonzhang/d2l-en ▾ compare: master ▾

✓ Able to merge. These branches can be automatically merged.

Discuss and review the changes in this comparison with others.

Fig. 18.6.6: Create Pull Request.

Your pull request will appear among the list of requests in the main repository. We will make every effort to process it quickly.

Summary

- You can use GitHub to contribute to this book.
- Forking a repository is the first step to contributing, since it allows you to edit things locally and only contribute back once you are ready.
- Pull requests are how contributions are being bundled up. Try not to submit huge pull requests since this makes them hard to understand and incorporate. Better send several smaller ones.

Exercises

1. Star and fork the `d2l-en` repository.
2. Find some code that needs improvement and submit a pull request.
3. Find a reference that we missed and submit a pull request.

18.7 d2l API Document

class d2l.Accumulator(*n*)

Sum a list of numbers over time

class d2l.BPRLoss(*weight=None*, *batch_axis=0*, ***kwargs*)

forward(*positive*, *negative*)

Defines the forward computation. Arguments can be either NDArray or Symbol.

class d2l.CTRDataset(*data_path*, *feat_mapper=None*, *defaults=None*, *min_threshold=4*, *num_feat=34*)

class d2l.Decoder(***kwargs*)

The base decoder interface for the encoder-decoder architecture.

forward(*X*, *state*)

Overrides to implement forward computation using NDArray. Only accepts positional arguments.

***args** [list of NDArray] Input tensors.

class d2l.DotProductAttention(*dropout*, ***kwargs*)

forward(*query*, *key*, *value*, *valid_length=None*)

Overrides to implement forward computation using NDArray. Only accepts positional arguments.

***args** [list of NDArray] Input tensors.

class d2l.Encoder(***kwargs*)

The base encoder interface for the encoder-decoder architecture.

forward(*X*)

Overrides to implement forward computation using NDArray. Only accepts positional arguments.

***args** [list of NDArray] Input tensors.

class d2l.EncoderDecoder(*encoder*, *decoder*, ***kwargs*)

The base class for the encoder-decoder architecture.

forward(*enc_X*, *dec_X*, **args*)

Overrides to implement forward computation using NDArray. Only accepts positional arguments.

***args** [list of NDArray] Input tensors.

class d2l.HingeLossbRec(*weight=None*, *batch_axis=0*, ***kwargs*)

forward(*positive*, *negative*, *margin=1*)

Defines the forward computation. Arguments can be either NDArray or Symbol.

class d2l.MLPAttention(*units*, *dropout*, ***kwargs*)

`forward`(*query, key, value, valid_length*)
Overrides to implement forward computation using `NDArray`. Only accepts positional arguments.

***args** [list of NDArray] Input tensors.

class `d2l.MaskedSoftmaxCELoss`(*axis=-1, sparse_label=True, from_logits=False, weight=None, batch_axis=0, **kwargs*)

`forward`(*pred, label, valid_length*)
Defines the forward computation. Arguments can be either `NDArray` or `Symbol`.

class `d2l.RNNModel`(*rnn_layer, vocab_size, **kwargs*)

`forward`(*inputs, state*)
Overrides to implement forward computation using `NDArray`. Only accepts positional arguments.

***args** [list of NDArray] Input tensors.

class `d2l.RNNModelScratch`(*vocab_size, num_hiddens, ctx, get_params, init_state, forward*)
A RNN Model based on scratch implementations

class `d2l.RandomGenerator`(*sampling_weights*)
Draw a random int in [0, n] according to n sampling weights

class `d2l.Residual`(*num_channels, use_1x1conv=False, strides=1, **kwargs*)

`forward`(*X*)
Overrides to implement forward computation using `NDArray`. Only accepts positional arguments.

***args** [list of NDArray] Input tensors.

class `d2l.Seq2SeqDecoder`(*vocab_size, embed_size, num_hiddens, num_layers, dropout=0, **kwargs*)

`forward`(*X, state*)
Overrides to implement forward computation using `NDArray`. Only accepts positional arguments.

***args** [list of NDArray] Input tensors.

class `d2l.Seq2SeqEncoder`(*vocab_size, embed_size, num_hiddens, num_layers, dropout=0, **kwargs*)

`forward`(*X, *args*)
Overrides to implement forward computation using `NDArray`. Only accepts positional arguments.

***args** [list of NDArray] Input tensors.

class `d2l.SeqDataLoader`(*batch_size, num_steps, use_random_iter, max_tokens*)
A iterator to load sequence data

class `d2l.Timer`
Record multiple running times.

`class d2l.VOCSegDataset(`*is_train, crop_size, voc_dir*`)`
A customized dataset to load VOC dataset.

`filter(`*imgs*`)`
Returns a new dataset with samples filtered by the filter function *fn*.

Note that if the Dataset is the result of a lazily transformed one with transform(lazy=False), the filter is eagerly applied to the transformed samples without materializing the transformed result. That is, the transformation will be applied again whenever a sample is retrieved after filter().

fn [callable] A filter function that takes a sample as input and returns a boolean. Samples that return False are discarded.

Dataset The filtered dataset.

`d2l.bbox_to_rect(`*bbox, color*`)`
Convert bounding box to matplotlib format.

`d2l.build_colormap2label()`
Build a RGB color to label mapping for segmentation.

`d2l.corr2d(`*X, K*`)`
Compute 2D cross-correlation.

`class d2l.defaultdict`
defaultdict(default_factory[, ...]) –> dict with default factory

The default factory is called without arguments to produce a new value when a key is not present, in __getitem__ only. A defaultdict compares equal to a dict with the same items. All remaining arguments are treated the same as if they were passed to the dict constructor, including keyword arguments.

`copy()` → a shallow copy of D.

`default_factory`
Factory for default value called by __missing__().

`d2l.download_voc_pascal(`*data_dir='../data'*`)`
Download the VOC2012 segmentation dataset.

`d2l.evaluate_loss(`*net, data_iter, loss*`)`
Evaluate the loss of a model on the given dataset

`d2l.load_array(`*data_arrays, batch_size, is_train=True*`)`
Construct a Gluon data loader

`d2l.load_data_fashion_mnist(`*batch_size, resize=None*`)`
Download the Fashion-MNIST dataset and then load into memory.

`d2l.load_data_pikachu(`*batch_size, edge_size=256*`)`
Load the pikachu dataset

`d2l.load_data_voc(`*batch_size, crop_size*`)`
Download and load the VOC2012 semantic dataset.

`d2l.plot(`*X, Y=None, xlabel=None, ylabel=None, legend=[], xlim=None, ylim=None, xscale='linear', yscale='linear', fmts=['-', 'm--', 'g-.', 'r:'], figsize=(3.5, 2.5), axes=None*`)`
Plot data points.

`d2l.read_time_machine()`
Load the time machine book into a list of sentences.

`d2l.read_voc_images(`*root='../data/VOCdevkit/VOC2012', is_train=True*`)`
Read all VOC feature and label images.

`d2l.resnet18(`*num_classes*`)`
A slightly modified ResNet-18 model.

`d2l.set_axes(`*axes, xlabel, ylabel, xlim, ylim, xscale, yscale, legend*`)`
Set the axes for matplotlib.

`d2l.set_figsize(`*figsize=(3.5, 2.5)*`)`
Set the figure size for matplotlib.

`d2l.show_bboxes(`*axes, bboxes, labels=None, colors=None*`)`
Show bounding boxes.

`d2l.show_images(`*imgs, num_rows, num_cols, titles=None, scale=1.5*`)`
Plot a list of images.

`d2l.show_trace_2d(`*f, results*`)`
Show the trace of 2D variables during optimization.

`d2l.split_batch(`*X, y, ctx_list*`)`
Split X and y into multiple devices specified by ctx.

`d2l.split_data_ml100k(`*data, num_users, num_items, split_mode='random', test_ratio=0.1*`)`
Split the dataset in random mode or seq-aware mode.

`d2l.synthetic_data(`*w, b, num_examples*`)`
generate y = X w + b + noise

`d2l.tokenize(`*lines, token='word'*`)`
Split sentences into word or char tokens

`d2l.train_2d(`*trainer, steps=20*`)`
Optimize a 2-dim objective function with a customized trainer.

`d2l.try_all_gpus()`
Return all available GPUs, or [cpu(),] if no GPU exists.

`d2l.try_gpu(`*i=0*`)`
Return gpu(i) if exists, otherwise return cpu().

`d2l.update_D(`*X, Z, net_D, net_G, loss, trainer_D*`)`
Update discriminator

`d2l.update_G(`*Z, net_D, net_G, loss, trainer_G*`)`
Update generator

`d2l.use_svg_display()`
Use the svg format to display a plot in Jupyter.

`d2l.voc_label_indices(`*colormap, colormap2label*`)`
Map a RGB color to a label.

`d2l.voc_rand_crop(`*feature, label, height, width*`)`
Randomly crop for both feature and label images.

Bibliography

Ahmed, A., Aly, M., Gonzalez, J., Narayanamurthy, S., & Smola, A. J. (2012). Scalable inference in latent variable models. *Proceedings of the fifth ACM international conference on Web search and data mining* (pp. 123–132).

Aji, S. M., & McEliece, R. J. (2000). The generalized distributive law. *IEEE transactions on Information Theory*, *46*(2), 325–343.

Bahdanau, D., Cho, K., & Bengio, Y. (2014). Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*.

Bishop, C. M. (1995). Training with noise is equivalent to tikhonov regularization. *Neural computation*, *7*(1), 108–116.

Bishop, C. M. (2006). *Pattern recognition and machine learning*. springer.

Bojanowski, P., Grave, E., Joulin, A., & Mikolov, T. (2017). Enriching word vectors with subword information. *Transactions of the Association for Computational Linguistics*, *5*, 135–146.

Bollobás, B. (1999). *Linear analysis*. Cambridge University Press, Cambridge.

Boyd, S., & Vandenberghe, L. (2004). *Convex Optimization*. Cambridge, England: Cambridge University Press.

Brown, N., & Sandholm, T. (2017). Libratus: the superhuman ai for no-limit poker. *IJCAI* (pp. 5226–5228).

Campbell, M., Hoane Jr, A. J., & Hsu, F.-h. (2002). Deep blue. *Artificial intelligence*, *134*(1-2), 57–83.

Canny, J. (1987). A computational approach to edge detection. *Readings in computer vision* (pp. 184–203). Elsevier.

Cho, K., Van Merriënboer, B., Bahdanau, D., & Bengio, Y. (2014). On the properties of neural machine translation: encoder-decoder approaches. *arXiv preprint arXiv:1409.1259*.

Chowdhury, G. G. (2010). *Introduction to modern information retrieval*. Facet publishing.

Chung, J., Gulcehre, C., Cho, K., & Bengio, Y. (2014). Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*.

Csiszár, I. (2008). Axiomatic characterizations of information measures. *Entropy*, *10*(3), 261–273.

De Cock, D. (2011). Ames, iowa: alternative to the boston housing data as an end of semester regression project. *Journal of Statistics Education*, *19*(3).

DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., … Vogels, W. (2007). Dynamo: amazon's highly available key-value store. *ACM SIGOPS operating systems review* (pp. 205–220).

Doucet, A., De Freitas, N., & Gordon, N. (2001). An introduction to sequential monte carlo methods. *Sequential Monte Carlo methods in practice* (pp. 3–14). Springer.

Duchi, J., Hazan, E., & Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research, 12*(Jul), 2121–2159.

Dumoulin, V., & Visin, F. (2016). A guide to convolution arithmetic for deep learning. *arXiv preprint arXiv:1603.07285*.

Edelman, B., Ostrovsky, M., & Schwarz, M. (2007). Internet advertising and the generalized second-price auction: selling billions of dollars worth of keywords. *American economic review, 97*(1), 242–259.

Flammarion, N., & Bach, F. (2015). From averaging to acceleration, there is only a step-size. *Conference on Learning Theory* (pp. 658–695).

Gatys, L. A., Ecker, A. S., & Bethge, M. (2016). Image style transfer using convolutional neural networks. *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 2414–2423).

Ginibre, J. (1965). Statistical ensembles of complex, quaternion, and real matrices. *Journal of Mathematical Physics, 6*(3), 440–449.

Girshick, R. (2015). Fast r-cnn. *Proceedings of the IEEE international conference on computer vision* (pp. 1440–1448).

Girshick, R., Donahue, J., Darrell, T., & Malik, J. (2014). Rich feature hierarchies for accurate object detection and semantic segmentation. *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 580–587).

Glorot, X., & Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. *Proceedings of the thirteenth international conference on artificial intelligence and statistics* (pp. 249–256).

Goh, G. (2017). Why momentum really works. *Distill*. URL: http://distill.pub/2017/momentum, doi:10.23915/distill.00006[283]

Goldberg, D., Nichols, D., Oki, B. M., & Terry, D. (1992). Using collaborative filtering to weave an information tapestry. *Communications of the ACM, 35*(12), 61–71.

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep Learning*. MIT Press. http://www.deeplearningbook.org.

Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., … Bengio, Y. (2014). Generative adversarial nets. *Advances in neural information processing systems* (pp. 2672–2680).

Gotmare, A., Keskar, N. S., Xiong, C., & Socher, R. (2018). A closer look at deep learning heuristics: learning rate restarts, warmup and distillation. *arXiv preprint arXiv:1810.13243*.

Graves, A., & Schmidhuber, J. (2005). Framewise phoneme classification with bidirectional lstm and other neural network architectures. *Neural networks, 18*(5-6), 602–610.

Gunawardana, A., & Shani, G. (2015). Evaluating recommender systems. *Recommender systems handbook* (pp. 265–308). Springer.

Guo, H., Tang, R., Ye, Y., Li, Z., & He, X. (2017). Deepfm: a factorization-machine based neural network for ctr prediction. *Proceedings of the 26th International Joint Conference on Artificial Intelligence* (pp. 1725–1731).

[283] https://doi.org/10.23915/distill.00006

Hadjis, S., Zhang, C., Mitliagkas, I., Iter, D., & Ré, C. (2016). Omnivore: an optimizer for multi-device deep learning on cpus and gpus. *arXiv preprint arXiv:1606.04487*.

Hazan, E., Rakhlin, A., & Bartlett, P. L. (2008). Adaptive online gradient descent. *Advances in Neural Information Processing Systems* (pp. 65–72).

He, K., Gkioxari, G., Dollár, P., & Girshick, R. (2017). Mask r-cnn. *Proceedings of the IEEE international conference on computer vision* (pp. 2961–2969).

He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 770–778).

He, K., Zhang, X., Ren, S., & Sun, J. (2016). Identity mappings in deep residual networks. *European conference on computer vision* (pp. 630–645).

He, X., & Chua, T.-S. (2017). Neural factorization machines for sparse predictive analytics. *Proceedings of the 40th International ACM SIGIR conference on Research and Development in Information Retrieval* (pp. 355–364).

He, X., Liao, L., Zhang, H., Nie, L., Hu, X., & Chua, T.-S. (2017). Neural collaborative filtering. *Proceedings of the 26th international conference on world wide web* (pp. 173–182).

Hebb, D. O., & Hebb, D. (1949). *The organization of behavior*. Vol. 65. Wiley New York.

Hennessy, J. L., & Patterson, D. A. (2011). *Computer architecture: a quantitative approach*. Elsevier.

Herlocker, J. L., Konstan, J. A., Borchers, A., & Riedl, J. (1999). An algorithmic framework for performing collaborative filtering. *22nd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR 1999* (pp. 230–237).

Hidasi, B., Karatzoglou, A., Baltrunas, L., & Tikk, D. (2015). Session-based recommendations with recurrent neural networks. *arXiv preprint arXiv:1511.06939*.

Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. *Neural computation, 9*(8), 1735–1780.

Hoyer, P. O., Janzing, D., Mooij, J. M., Peters, J., & Schölkopf, B. (2009). Nonlinear causal discovery with additive noise models. *Advances in neural information processing systems* (pp. 689–696).

Hu, J., Shen, L., & Sun, G. (2018). Squeeze-and-excitation networks. *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 7132–7141).

Hu, Y., Koren, Y., & Volinsky, C. (2008). Collaborative filtering for implicit feedback datasets. *2008 Eighth IEEE International Conference on Data Mining* (pp. 263–272).

Huang, G., Liu, Z., Van Der Maaten, L., & Weinberger, K. Q. (2017). Densely connected convolutional networks. *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 4700–4708).

Ioffe, S. (2017). Batch renormalization: towards reducing minibatch dependence in batch-normalized models. *Advances in neural information processing systems* (pp. 1945–1953).

Ioffe, S., & Szegedy, C. (2015). Batch normalization: accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*.

Izmailov, P., Podoprikhin, D., Garipov, T., Vetrov, D., & Wilson, A. G. (2018). Averaging weights leads to wider optima and better generalization. *arXiv preprint arXiv:1803.05407*.

Jia, X., Song, S., He, W., Wang, Y., Rong, H., Zhou, F., … others. (2018). Highly scalable deep learning training system with mixed-precision: training imagenet in four minutes. *arXiv preprint arXiv:1807.11205*.

Jouppi, N. P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., … others. (2017). In-datacenter performance analysis of a tensor processing unit. *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)* (pp. 1–12).

Karras, T., Aila, T., Laine, S., & Lehtinen, J. (2017). Progressive growing of gans for improved quality, stability, and variation. *arXiv preprint arXiv:1710.10196*.

Kim, Y. (2014). Convolutional neural networks for sentence classification. *arXiv preprint arXiv:1408.5882*.

Kingma, D. P., & Ba, J. (2014). Adam: a method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Koller, D., & Friedman, N. (2009). *Probabilistic graphical models: principles and techniques*. MIT press.

Kolter, Z. (2008). Linear algebra review and reference. *Available online: http*.

Koren, Y. (2009). Collaborative filtering with temporal dynamics. *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining* (pp. 447–456).

Koren, Y., Bell, R., & Volinsky, C. (2009). Matrix factorization techniques for recommender systems. *Computer*, pp. 30–37.

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems* (pp. 1097–1105).

Kung, S. Y. (1988). Vlsi array processors. *Englewood Cliffs, NJ, Prentice Hall, 1988, 685 p. Research supported by the Semiconductor Research Corp., SDIO, NSF, and US Navy.*

LeCun, Y., Bottou, L., Bengio, Y., Haffner, P., & others. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, *86*(11), 2278–2324.

Li, M. (2017). *Scaling Distributed Machine Learning with System and Algorithm Co-design* (Doctoral dissertation). PhD Thesis, CMU.

Li, M., Andersen, D. G., Park, J. W., Smola, A. J., Ahmed, A., Josifovski, V., … Su, B.-Y. (2014). Scaling distributed machine learning with the parameter server. *11th $\$USENIX$\$ Symposium on Operating Systems Design and Implementation ($\$OSDI$\$ 14)* (pp. 583–598).

Lin, M., Chen, Q., & Yan, S. (2013). Network in network. *arXiv preprint arXiv:1312.4400*.

Lin, T.-Y., Goyal, P., Girshick, R., He, K., & Dollár, P. (2017). Focal loss for dense object detection. *Proceedings of the IEEE international conference on computer vision* (pp. 2980–2988).

Lin, Y., Lv, F., Zhu, S., Yang, M., Cour, T., Yu, K., … others. (2010). Imagenet classification: fast descriptor coding and large-scale svm training. *Large scale visual recognition challenge*.

Lipton, Z. C., & Steinhardt, J. (2018). Troubling trends in machine learning scholarship. *arXiv preprint arXiv:1807.03341*.

Liu, W., Anguelov, D., Erhan, D., Szegedy, C., Reed, S., Fu, C.-Y., & Berg, A. C. (2016). Ssd: single shot multibox detector. *European conference on computer vision* (pp. 21–37).

Long, J., Shelhamer, E., & Darrell, T. (2015). Fully convolutional networks for semantic segmentation. *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 3431–3440).

Loshchilov, I., & Hutter, F. (2016). Sgdr: stochastic gradient descent with warm restarts. *arXiv preprint arXiv:1608.03983*.

Lowe, D. G. (2004). Distinctive image features from scale-invariant keypoints. *International journal of computer vision, 60*(2), 91–110.

Luo, P., Wang, X., Shao, W., & Peng, Z. (2018). Towards understanding regularization in batch normalization. *arXiv preprint*.

Maas, A. L., Daly, R. E., Pham, P. T., Huang, D., Ng, A. Y., & Potts, C. (2011). Learning word vectors for sentiment analysis. *Proceedings of the 49th annual meeting of the association for computational linguistics: Human language technologies-volume 1* (pp. 142–150).

McCulloch, W. S., & Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics, 5*(4), 115–133.

McMahan, H. B., Holt, G., Sculley, D., Young, M., Ebner, D., Grady, J., … others. (2013). Ad click prediction: a view from the trenches. *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining* (pp. 1222–1230).

Mikolov, T., Chen, K., Corrado, G., & Dean, J. (2013). Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*.

Mikolov, T., Sutskever, I., Chen, K., Corrado, G. S., & Dean, J. (2013). Distributed representations of words and phrases and their compositionality. *Advances in neural information processing systems* (pp. 3111–3119).

Mirhoseini, A., Pham, H., Le, Q. V., Steiner, B., Larsen, R., Zhou, Y., … Dean, J. (2017). Device placement optimization with reinforcement learning. *Proceedings of the 34th International Conference on Machine Learning-Volume 70* (pp. 2430–2439).

Morey, R. D., Hoekstra, R., Rouder, J. N., Lee, M. D., & Wagenmakers, E.-J. (2016). The fallacy of placing confidence in confidence intervals. *Psychonomic bulletin & review, 23*(1), 103–123.

Nesterov, Y., & Vial, J.-P. (2000). *Confidence level solutions for stochastic programming, Stochastic Programming E-Print Series*.

Nesterov, Y. (2018). *Lectures on convex optimization*. Vol. 137. Springer.

Neyman, J. (1937). Outline of a theory of statistical estimation based on the classical theory of probability. *Philosophical Transactions of the Royal Society of London. Series A, Mathematical and Physical Sciences, 236*(767), 333–380.

Pennington, J., Schoenholz, S., & Ganguli, S. (2017). Resurrecting the sigmoid in deep learning through dynamical isometry: theory and practice. *Advances in neural information processing systems* (pp. 4785–4795).

Pennington, J., Socher, R., & Manning, C. (2014). Glove: global vectors for word representation. *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)* (pp. 1532–1543).

Peters, J., Janzing, D., & Schölkopf, B. (2017). *Elements of causal inference: foundations and learning algorithms*. MIT press.

Petersen, K. B., Pedersen, M. S., & others. (2008). The matrix cookbook. *Technical University of Denmark, 7*(15), 510.

Polyak, B. T. (1964). Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics, 4*(5), 1–17.

Quadrana, M., Cremonesi, P., & Jannach, D. (2018). Sequence-aware recommender systems. *ACM Computing Surveys (CSUR)*, *51*(4), 66.

Radford, A., Metz, L., & Chintala, S. (2015). Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434*.

Reddi, S. J., Kale, S., & Kumar, S. (2019). On the convergence of adam and beyond. *arXiv preprint arXiv:1904.09237*.

Reed, S., & De Freitas, N. (2015). Neural programmer-interpreters. *arXiv preprint arXiv:1511.06279*.

Ren, S., He, K., Girshick, R., & Sun, J. (2015). Faster r-cnn: towards real-time object detection with region proposal networks. *Advances in neural information processing systems* (pp. 91–99).

Rendle, S. (2010). Factorization machines. *2010 IEEE International Conference on Data Mining* (pp. 995–1000).

Rendle, S., Freudenthaler, C., Gantner, Z., & Schmidt-Thieme, L. (2009). Bpr: bayesian personalized ranking from implicit feedback. *Proceedings of the twenty-fifth conference on uncertainty in artificial intelligence* (pp. 452–461).

Rumelhart, D. E., Hinton, G. E., Williams, R. J., & others. (1988). Learning representations by back-propagating errors. *Cognitive modeling*, *5*(3), 1.

Russell, S. J., & Norvig, P. (2016). *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited,.

Salton, G., Wong, A., & Yang, C.-S. (1975). A vector space model for automatic indexing. *Communications of the ACM*, *18*(11), 613–620.

Santurkar, S., Tsipras, D., Ilyas, A., & Madry, A. (2018). How does batch normalization help optimization? *Advances in Neural Information Processing Systems* (pp. 2483–2493).

Sarwar, B. M., Karypis, G., Konstan, J. A., Riedl, J., & others. (2001). Item-based collaborative filtering recommendation algorithms. *Www*, *1*, 285–295.

Schein, A. I., Popescul, A., Ungar, L. H., & Pennock, D. M. (2002). Methods and metrics for cold-start recommendations. *Proceedings of the 25th annual international ACM SIGIR conference on Research and development in information retrieval* (pp. 253–260).

Schuster, M., & Paliwal, K. K. (1997). Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing*, *45*(11), 2673–2681.

Sedhain, S., Menon, A. K., Sanner, S., & Xie, L. (2015). Autorec: autoencoders meet collaborative filtering. *Proceedings of the 24th International Conference on World Wide Web* (pp. 111–112).

Sergeev, A., & Del Balso, M. (2018). Horovod: fast and easy distributed deep learning in tensorflow. *arXiv preprint arXiv:1802.05799*.

Shannon, C. E. (1948 , 7). A mathematical theory of communication. *The Bell System Technical Journal*, *27*(3), 379–423.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., … others. (2016). Mastering the game of go with deep neural networks and tree search. *nature*, *529*(7587), 484.

Simonyan, K., & Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*.

Smola, A., & Narayanamurthy, S. (2010). An architecture for parallel topic models. *Proceedings of the VLDB Endowment*, *3*(1-2), 703–710.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research, 15*(1), 1929–1958.

Strang, G. (1993). *Introduction to linear algebra*. Vol. 3. Wellesley-Cambridge Press Wellesley, MA.

Su, X., & Khoshgoftaar, T. M. (2009). A survey of collaborative filtering techniques. *Advances in artificial intelligence, 2009*.

Sukhbaatar, S., Weston, J., Fergus, R., & others. (2015). End-to-end memory networks. *Advances in neural information processing systems* (pp. 2440–2448).

Sutskever, I., Martens, J., Dahl, G., & Hinton, G. (2013). On the importance of initialization and momentum in deep learning. *International conference on machine learning* (pp. 1139–1147).

Szegedy, C., Ioffe, S., Vanhoucke, V., & Alemi, A. A. (2017). Inception-v4, inception-resnet and the impact of residual connections on learning. *Thirty-First AAAI Conference on Artificial Intelligence*.

Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., … Rabinovich, A. (2015). Going deeper with convolutions. *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 1–9).

Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., & Wojna, Z. (2016). Rethinking the inception architecture for computer vision. *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 2818–2826).

Tallec, C., & Ollivier, Y. (2017). Unbiasing truncated backpropagation through time. *arXiv preprint arXiv:1705.08209*.

Tang, J., & Wang, K. (2018). Personalized top-n sequential recommendation via convolutional sequence embedding. *Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining* (pp. 565–573).

Teye, M., Azizpour, H., & Smith, K. (2018). Bayesian uncertainty estimation for batch normalized deep networks. *arXiv preprint arXiv:1802.06455*.

Tieleman, T., & Hinton, G. (2012). Lecture 6.5-rmsprop: divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning, 4*(2), 26–31.

Treisman, A. M., & Gelade, G. (1980). A feature-integration theory of attention. *Cognitive psychology, 12*(1), 97–136.

Töscher, A., Jahrer, M., & Bell, R. M. (2009). The bigchaos solution to the netflix grand prize. *Netflix prize documentation*, pp. 1–52.

Uijlings, J. R., Van De Sande, K. E., Gevers, T., & Smeulders, A. W. (2013). Selective search for object recognition. *International journal of computer vision, 104*(2), 154–171.

Van Loan, C. F., & Golub, G. H. (1983). *Matrix computations*. Johns Hopkins University Press.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., … Polosukhin, I. (2017). Attention is all you need. *Advances in neural information processing systems* (pp. 5998–6008).

Wang, L., Li, M., Liberty, E., & Smola, A. J. (2018). Optimal message scheduling for aggregation. *NETWORKS, 2*(3), 2–3.

Wang, Y., Davidson, A., Pan, Y., Wu, Y., Riffel, A., & Owens, J. D. (2016). Gunrock: a high-performance graph processing library on the gpu. *ACM SIGPLAN Notices* (p. 11).

Wasserman, L. (2013). *All of statistics: a concise course in statistical inference*. Springer Science & Business Media.

Watkins, C. J., & Dayan, P. (1992). Q-learning. *Machine learning, 8*(3-4), 279–292.

Welling, M., & Teh, Y. W. (2011). Bayesian learning via stochastic gradient langevin dynamics. *Proceedings of the 28th international conference on machine learning (ICML-11)* (pp. 681–688).

Wigner, E. P. (1958). On the distribution of the roots of certain symmetric matrices. *Ann. Math* (pp. 325–327).

Williams, S., Waterman, A., & Patterson, D. (2009). *Roofline: An insightful visual performance model for floating-point programs and multicore architectures*. Lawrence Berkeley National Lab.(LBNL), Berkeley, CA (United States).

Wood, F., Gasthaus, J., Archambeau, C., James, L., & Teh, Y. W. (2011). The sequence memoizer. *Communications of the ACM, 54*(2), 91–98.

Wu, C.-Y., Ahmed, A., Beutel, A., Smola, A. J., & Jing, H. (2017). Recurrent recommender networks. *Proceedings of the tenth ACM international conference on web search and data mining* (pp. 495–503).

Xiao, H., Rasul, K., & Vollgraf, R. (2017). Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. *arXiv preprint arXiv:1708.07747*.

Xiong, W., Wu, L., Alleva, F., Droppo, J., Huang, X., & Stolcke, A. (2018). The microsoft 2017 conversational speech recognition system. *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)* (pp. 5934–5938).

Ye, M., Yin, P., Lee, W.-C., & Lee, D.-L. (2011). Exploiting geographical influence for collaborative point-of-interest recommendation. *Proceedings of the 34th international ACM SIGIR conference on Research and development in Information Retrieval* (pp. 325–334).

You, Y., Gitman, I., & Ginsburg, B. (2017). Large batch training of convolutional networks. *arXiv preprint arXiv:1708.03888*.

Zaheer, M., Reddi, S., Sachan, D., Kale, S., & Kumar, S. (2018). Adaptive methods for nonconvex optimization. *Advances in Neural Information Processing Systems* (pp. 9793–9803).

Zeiler, M. D. (2012). Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*.

Zhang, S., Yao, L., Sun, A., & Tay, Y. (2019). Deep learning based recommender system: a survey and new perspectives. *ACM Computing Surveys (CSUR), 52*(1), 5.

Zhu, J.-Y., Park, T., Isola, P., & Efros, A. A. (2017). Unpaired image-to-image translation using cycle-consistent adversarial networks. *Proceedings of the IEEE international conference on computer vision* (pp. 2223–2232).