Deep Learning in Mathematica

Giulio Alessandrini, WRI

ABSTRACT

The Wolfram Language contains a scalable, industrial-strength and easy-to-use neural net framework. In this talk, we are going to demonstrate its capabilities and its integration with the rest of the language. We are going to show how to build and train a net model from scratch, as well as work with a ready-to-use model from the Neural Net Repository.

Applications Survey: Image Classification

In the WL

Many models trained on ImageNet are available via **NetModel** and can be viewed on the Wolfram Neural Net Repository:

In[266]:=
```
net = NetModel["Inception V3 Trained on ImageNet Competition Data"]
```

Out[266]=
```
NetChain[ <image> Input port: image
          Output port: class
          Number of layers: 33 ]
```

In[267]:=
```
net[ <image> ]
```

Out[267]=
```
"peacock"
```

ImageIdentify uses this technology, and the underlying net is available to users:

In[268]:=
```
net2 = NetModel["Wolfram ImageIdentify Net V1"]
```

Out[268]=
```
NetChain[ <image> Input port: image
          Output port: class
          Number of layers: 24 ]
```

In[269]:=
```
net2[ <image> ]
```

Out[269]=
```
"black rhinoceros"
```

Deep Learning in the Wolfram Language

1. Layers

- A layer is the simplest component of a network.
- It represents an operation and by default it's completely symbolic

In[270]:=
```
elem = ElementwiseLayer[Tanh]
```

Out[270]=
```
ElementwiseLayer[ 

]
```

- Layers **only** act on numeric tensors:

In[271]:=
```
elem@{1, 2, 3}
N@Tanh@{1, 2, 3}
```

Out[271]=
```
{0.761594, 0.964028, 0.995055}
```

Out[272]=
```
{0.761594, 0.964028, 0.995055}
```

- Layers are differentiable. Differentiability is a key property that allows for the efficient training of nets, which we will see later:

In[273]:=
```
elem[{1, 2, 3}, NetPortGradient["Input"]]
```

Out[273]=
```
{0.419974, 0.0706508, 0.009866}
```

In[274]:=
```
D[Tanh[x], x] /. x -> {1., 2., 3.}
```

Out[274]=
```
{0.419974, 0.0706508, 0.00986604}
```

Deep Learning in the Wolfram Language

1. Layers

- They can run on both NVIDIA GPUs and CPUs:

In[275]:=
```
elem[{1, 2, 3}, TargetDevice → "GPU"]
```

Out[275]=
```
{0.761594, 0.964028, 0.995055}
```

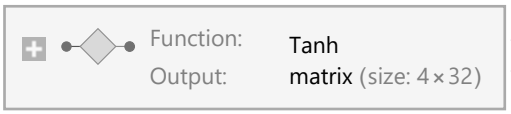
- They do shape inference:

In[276]:=
```
ElementwiseLayer[Tanh]
```

Out[276]=
```
ElementwiseLayer[
```


```
]
```

In[277]:=
```
ElementwiseLayer[Tanh, "Input" → {4, 32}]
```

Out[277]=
```
ElementwiseLayer[
```


```
]
```

- Certain layers have learnable parameters
  - without this, no learning would be possible!

In[278]:=
```
dot = LinearLayer[3, "Input" → 2]
```

Out[278]=
```
LinearLayer[
```


```
]
```

Initialize the parameters in the layer:

In[279]:=
```
dot2 = NetInitialize@dot
```

Out[279]=
```
LinearLayer[
```


```
]
```

In[280]:=
```
NetExtract[dot2, "Weights"]
```

Out[280]=
```
{{0.386105, -0.475143}, {0.306941, 0.0577074}, {1.03835, -0.147243}}
```

Deep Learning in the Wolfram Language

1. Layers

So far, we have seen layers that have exactly one input. Some layers have more than one input.

- For example, MeanSquaredLossLayer compares two arrays, called the *input* and the *target*, and produces a single number that represents Mean[(*input* – *target*)^2].

In[281]:=
```
msloss = MeanSquaredLossLayer[]
```

Out[281]=
```
MeanSquaredLossLayer[ 
                      Input:  tensor
                      Target: tensor ]
```

The inputs of the layer are named and must be supplied in an association when the net is applied:

In[282]:=
```
msloss[<|"Input" → {1, 2, 3}, "Target" → {4, 0, 4}|>]
```

Out[282]=
```
4.66667
```

The full list of available layers is:

In[283]:=
```
? *Layer
```

▼ System`

AggregationLayer	ConvolutionLayer	FlattenLayer	PaddingLayer	SequenceRestLayer
AppendLayer	CrossEntropyLossLayer	GatedRecurrentLayer	PartLayer	SequenceReverseLayer
BasicRecurrentLayer	CTCLossLayer	ImageAugmentationLayer	PoolingLayer	SoftmaxLayer
BatchNormalizationLayer	DeconvolutionLayer	InstanceNormalizationLayer	ReplicateLayer	SpatialTransformationLayer
CatenateLayer	DotLayer	LinearLayer	ReshapeLayer	SummationLayer
ConstantArrayLayer	DotPlusLayer	LocalResponseNormalizationLayer	ResizeLayer	ThreadingLayer
ConstantPlusLayer	DropoutLayer	LongShortTermMemoryLayer	SequenceAttentionLayer	TotalLayer
ConstantTimesLayer	ElementwiseLayer	MeanAbsoluteLossLayer	SequenceLastLayer	TransposeLayer
ContrastiveLossLayer	EmbeddingLayer	MeanSquaredLossLayer	SequenceMostLayer	UnitVectorLayer

Deep Learning in the Wolfram Language

1. Layers

- The simplest learnable layer is the **LinearLayer**

This is just:

In[284]:=
```
linear[data_, weight_, bias_] := Dot[weight, data] + bias
```

Comparing this to a **LinearLayer**:

In[285]:=
```
layer = NetInitialize@LinearLayer[2, "Input" → 3]
layer[{2, 10, 3}]
```

Out[285]=
```
LinearLayer[ 
  Input:  vector (size: 3)
  Output: vector (size: 2) ]
```

Out[286]=
```
{13.7261, 0.132025}
```

In[287]:=
```
linear[{2, 10, 3}, NetExtract[layer, "Weights"], NetExtract[layer, "Biases"]]
```

Out[287]=
```
{13.7261, 0.132025}
```

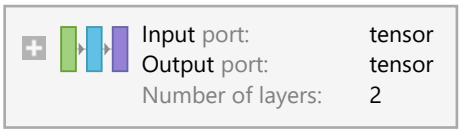
Deep Learning in the Wolfram Language

2. Containers

Single neural net layers are generally not useful by themselves. We usually need to combine multiple layers together to do something interesting.

- The simplest container is a chain
- Chain together two operations:

In[288]:=
```
net = NetChain[{ElementwiseLayer[Tanh], ElementwiseLayer[LogisticSigmoid]}]
```

Out[288]=
```
NetChain[ 
  Input port:            tensor
  Output port:           tensor
  Number of layers:      2
]
```

- Equivalent to:

In[289]:=
```
f[x_] := LogisticSigmoid@Tanh@x
```

SetDelayed: Tag Plus in $(5 - 3\,x - 2\,x^2 + x^3)[x\_]$ is Protected.

Out[289]=
```
$Failed
```

- Equivalent on data:

In[290]:=
```
data = {1., 2., 3.};
net@data
f@data
```

Out[291]=
```
{0.6817, 0.723927, 0.730085}
```

Out[292]=
```
(5 - 3 x - 2 x^2 + x^3) [{1., 2., 3.}]
```

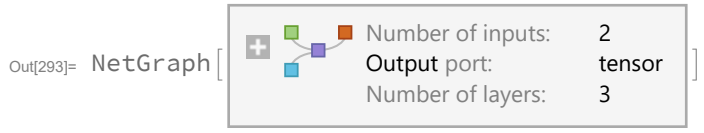
Deep Learning in the Wolfram Language

3. Containers

- **NetChain** does not allow a net to take more than one input, so we need to use **NetGraph** to build the training network

Create a **NetGraph**:

In[293]:=
```
net = NetGraph[{ElementwiseLayer[Tanh], ElementwiseLayer[LogisticSigmoid],
    TotalLayer[]}, {NetPort["Input1"] → 1, NetPort["Input2"] → 2, {1, 2} → 3}]
```

Out[293]=


Equivalent to:

In[294]:=
```
func = (Tanh@#Input1 + LogisticSigmoid@#Input2) &;
```

Evaluate on data:

In[295]:=
```
data = <|"Input1" -> {0.1, -2.4}, "Input2" → {-1.2, 3.4}|>;
net@data
func@data
```

Out[296]=
```
{0.331143, -0.0159703}
```

Out[297]=
```
{0.331143, -0.0159703}
```

- As all of the layers are differentiable, so is the container

In[298]:=
```
net[data, NetPortGradient["Input1"]]
```

Out[298]=
```
{0.990066, 0.0323837}
```

Deep Learning in the Wolfram Language

4. Containers

- Containers behave exactly like normal layers!
  - differentiable, run on GPUs, etc
- Containers can be nested, as they are just like normal layers:

In[299]:=
```
NetChain[{NetChain[{LinearLayer[]}]}]
```

Out[299]=
```
NetChain[ uninitialized | Input port: tensor
                          Output port: tensor
                          Number of layers: 1 ]
```

- Models in the Repository are almost all some form of container:

In[300]:=
```
NetModel["AdaIN-Style Trained on MS-COCO and Painter by Numbers Data"]
```

Out[300]=
```
NetGraph[ Number of inputs: 2
          Output port: image
          Number of layers: 4 ]
```

Deep Learning in the Wolfram Language

4. Containers

- There is a rich shape inference between all the layers in a container

In[301]:=
```
NetChain[{LinearLayer[], LinearLayer[]}]
```

Out[301]=
```
NetChain[
  uninitialized
  Input port:            tensor
  Output port:           tensor
  Number of layers:      2
]
```

In[302]:=
```
NetChain[{LinearLayer[3], LinearLayer[]}]
```

Out[302]=
```
NetChain[
  uninitialized
  Input port:            tensor
  Output port:           tensor
  Number of layers:      2
]
```

In[303]:=
```
NetChain[{LinearLayer[3], LinearLayer[4]}]
```

Out[303]=
```
NetChain[
  uninitialized
  Input port:            tensor
  Output port:           vector (size: 4)
  Number of layers:      2
]
```

In[304]:=
```
NetChain[{LinearLayer[3], LinearLayer[4]}, "Input" → 5]
```

Out[304]=
```
NetChain[
  uninitialized
  Input port:            vector (size: 5)
  Output port:           vector (size: 4)
  Number of layers:      2
]
```

- Helpful error messages point to the exact mismatches

In[305]:=
```
NetChain[{LinearLayer[3, "Input" → 6], LinearLayer[4]}, "Input" → 5]
```

NetChain: Specification 5 is not compatible with port "Input", which must be a length–6 vector.

Out[305]=
```
$Failed
```

Deep Learning in the Wolfram Language

5. NetEncoders

- Fundamentally, because they must be *differentiable*, neural net layers operate on numeric tensors. However, we often want to train and use nets on other data, such as images, audio, text, etc.
- We can use a **NetEncoder** to translate this data to numeric tensors.

Create an image **NetEncoder** that produces a $1 \times 12 \times 12$ tensor:

In[306]:=
```
imageenc = NetEncoder[{"Image", {12, 12}, "ColorSpace" → "Grayscale"}]
```

Out[306]=
```
NetEncoder[
```

Type:	Image
Image size:	{12, 12}
Color space:	Grayscale
Color channels:	1
Mean image:	None
Variance image:	None
Output:	3–tensor (size: 1 × 12 × 12)

```
]
```

Apply the encoder to an image:

In[307]:=
```
imageenc[ 3 ]
```

Out[307]=
```
{{{1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.},
  {1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.},
  {1., 1., 1., 1., 0., 0., 0., 0., 0., 1., 1., 1.},
  {1., 1., 1., 0., 0., 0., 0., 0., 0., 1., 1., 1.},
  {1., 1., 1., 1., 1., 1., 0., 0., 0., 1., 1., 1.},
  {1., 1., 1., 1., 0., 0., 0., 0., 0., 1., 1., 1.},
  {1., 1., 1., 1., 1., 1., 1., 1., 0., 0., 1., 1.},
  {1., 1., 0., 1., 1., 1., 1., 1., 0., 0., 1., 1.},
  {1., 1., 0., 0., 1., 1., 0., 0., 0., 1., 1., 1.},
  {1., 1., 1., 1., 0., 0., 0., 0., 1., 1., 1., 1.},
  {1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.},
  {1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.}}}
```

Can also be applied to files

In[308]:=
```
f = File@FindFile["ExampleData/coneflower.jpg"]
```

Out[308]=
```
File[ C:\Program Files\Wolfram
      Research\Mathematica\11.3\Documentation\English\System\ExampleData\coneflower.jpg
```

In[309]:=
```
imageenc[f]
```

Out[309]=
```
{{{0.898039, 0.890196, 0.870588, 0.909804, 0.729412,
    0.698039, 0.603922, 0.815686, 0.65098, 0.658824, 1., 0.611765},
   {0.835294, 0.929412, 0.894118, 0.705882, 0.501961, 0.529412,
    0.470588, 0.462745, 0.564706, 0.929412, 0.717647, 0.541176},
   {1., 0.909804, 0.705882, 0.509804, 0.521569, 0.533333, 0.513726,
    0.470588, 0.529412, 0.662745, 0.572549, 0.666667},
   {0.764706, 0.933333, 0.6, 0.486275, 0.545098, 0.498039, 0.47451,
    0.486275, 0.443137, 0.537255, 0.639216, 0.623529},
   {0.317647, 0.439216, 0.552941, 0.521569, 0.498039, 0.419608,
    0.372549, 0.454902, 0.447059, 0.490196, 0.709804, 0.839216},
   {0.72549, 0.694118, 0.470588, 0.486275, 0.435294, 0.345098,
    0.341176, 0.384314, 0.427451, 0.431373, 0.737255, 0.909804},
   {0.952941, 0.807843, 0.458824, 0.482353, 0.435294, 0.337255,
    0.301961, 0.333333, 0.411765, 0.403922, 0.658824, 0.866667},
   {0.776471, 0.776471, 0.439216, 0.423529, 0.411765, 0.364706,
    0.313726, 0.368627, 0.419608, 0.4, 0.537255, 0.658824},
   {0.898039, 0.615686, 0.423529, 0.360784, 0.388235, 0.329412,
    0.337255, 0.415686, 0.352941, 0.466667, 0.654902, 0.545098},
   {0.490196, 0.435294, 0.619608, 0.321569, 0.301961, 0.368627,
    0.415686, 0.376471, 0.376471, 0.6, 0.694118, 0.807843},
   {0.494118, 0.956863, 0.643137, 0.34902, 0.25098, 0.25098,
    0.266667, 0.333333, 0.447059, 0.52549, 0.690196, 0.701961},
   {0.917647, 0.886275, 0.784314, 0.392157, 0.486275, 0.32549,
    0.411765, 0.447059, 0.705882, 0.32549, 0.682353, 0.760784}}}
```

- Allows for out-of-core learning on image and audio files!
  - See the tutorial Training on Large Datasets for more
- A large collection of encoders are available for different datatypes
  - “Audio”
  - “Characters”

Deep Learning in the Wolfram Language

5. NetEncoders

Encoders can be attached to the **input ports** of layers or containers. Attach an image NetEncoder to a PoolingLayer via the "Input" option:

In[310]:=
```
pool = PoolingLayer[10, "Input" → NetEncoder[{"Image", 64}]]
```

Out[310]=
```
PoolingLayer[ Kernel size: {10, 10}
              Stride: {1, 1} ]
```

Apply the PoolingLayer directly to an image, which will use the image NetEncoder to translate the image to a tensor for PoolingLayer to operate on:

In[311]:=
```
pool[ <image> ] // Shallow
```

Out[311]//Shallow=
```
{{{<<55>>}, {<<55>>}, {<<55>>}, {<<55>>}, {<<55>>},
   {<<55>>}, {<<55>>}, {<<55>>}, {<<55>>}, {<<55>>}, <<45>>},
  {{<<55>>}, {<<55>>}, {<<55>>}, {<<55>>}, {<<55>>}, {<<55>>},
   {<<55>>}, {<<55>>}, {<<55>>}, {<<55>>}, <<45>>},
  {{<<55>>}, {<<55>>}, {<<55>>}, {<<55>>}, {<<55>>}, {<<55>>},
   {<<55>>}, {<<55>>}, {<<55>>}, {<<55>>}, <<45>>}}
```

Deep Learning in the Wolfram Language

5. NetEncoders

Encoders are what allows trained models to be used directly on the type of interest:

In[312]:=
```
net = NetModel["Inception V3 Trained on ImageNet Competition Data"]
```

Out[312]=
```
NetChain[ <image> Input port: image
                  Output port: class
                  Number of layers: 33 ]
```

In[313]:=
```
net[ <image> ]
```

Out[313]=
```
peacock
```

In[314]:=
```
NetExtract[net, "Input"]
```

Out[314]=
```
NetEncoder[ Type: Image
            Image size: {299, 299}
            Color space: RGB
            Color channels: 3
            Mean image: {0.5, 0.5, 0.5}
            Variance image: None
            Output: 3–tensor (size: 3 × 299 × 299) ]
```

Deep Learning in the Wolfram Language

6. NetDecoders

A net will always output a numeric tensor. But for a task like classification, one wants class-labels as output. A **NetDecoder** is a mechanism for returning non-numeric tensors from nets.

In[315]:=
```
dec = NetDecoder[{"Class", {"dog", "cat"}}]
```

Out[315]=
```
NetDecoder[ Type: Class
            Labels: {dog, cat}
            Input depth: 1
            Dimensions: 2 ]
```

This decoder will interpret a vector of probabilities over classes as a class label:

In[316]:=
```
dec[{0.1, 0.9}]
```

Out[316]=
```
cat
```

The probabilities can also be obtained:

In[317]:=
```
dec[{0.1, 0.9}, "Probabilities"]
```

Out[317]=
```
⟨|dog → 0.1, cat → 0.9|⟩
```

Deep Learning in the Wolfram Language

6. NetDecoders

A decoder can be attached to the output of a layer or container:

In[318]:=
```
soft = SoftmaxLayer["Output" -> NetDecoder[{"Class", {"dog", "cat"}}]]
```

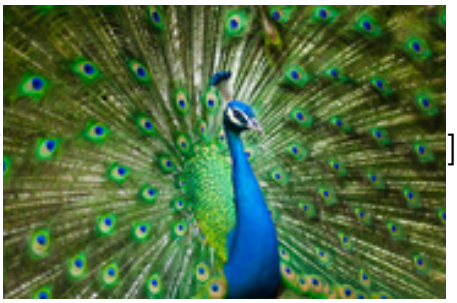
Out[318]=
```
SoftmaxLayer[ 
  Level:  -1
  Output: class ]
```

In[319]:=
```
soft[{44, 41}, "Probabilities"]
```

Out[319]=
```
⟨|dog → 0.952574, cat → 0.0474259|⟩
```

This mechanism allows pre-trained nets to output class-labels:

In[320]:=
```
net = NetModel["Inception V3 Trained on ImageNet Competition Data"];
```

In[321]:=
```
net[
```

```
]
```

Out[321]=
```
peacock
```

In[322]:=
```
NetExtract[net, "Output"]
```

Out[322]=
```
NetDecoder[
  Type:         Class
  Labels:       {other, kit fox, ≪997≫, carbonara, dumbbell}
  Input depth:  1
  Dimensions:   1001 ]
```

Deep Learning in the Wolfram Language

7. Training

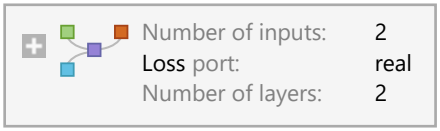
- To train a net, it must have one output, the loss
- Training involves finding parameters to minimize the loss

A very simple example:

In[323]:=
```
data = {{1} → {1.9}, {2} → {4.1}, {3} → {6.0}, {4} → {8.1}}
```

Out[323]=
```
{{1} → {1.9}, {2} → {4.1}, {3} → {6.}, {4} → {8.1}}
```

In[324]:=
```
net = NetInitialize@
  NetGraph[{LinearLayer[1], MeanSquaredLossLayer[]}, {1 → 2}, "Input" → {1}]
```

Out[324]=
```
NetGraph[
```


```
]
```

Evaluating this net:

In[325]:=
```
net[<|"Input" → {2}, "Target" → {4.1}|>]
```

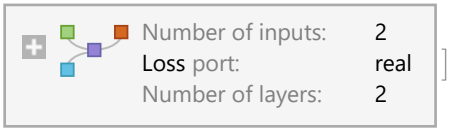
Out[325]=
```
1.56566
```

Deep Learning in the Wolfram Language

7. Training

Train the net:

In[326]:=
```
trainednet = NetTrain[net, data]
```

Out[326]=
```
NetGraph[ 
Number of inputs: 2
Loss port: real
Number of layers: 2 ]
```

The output is now much smaller:

In[327]:=
```
trainednet[<|"Input" → {2}, "Target" → {4.1}|>]
```

Out[327]=
```
0.00999998
```

Training has changed the parameters:

In[328]:=
```
NetExtract[net, {1, "Weights"}]
NetExtract[trainednet, {1, "Weights"}]
```

Out[328]=
```
{{1.42437}}
```

Out[329]=
```
{{2.05}}
```

Deep Learning in the Wolfram Language

8. Surgery

- Very easy to look into nets and modify them

Remove the weights from the trainable layers:

In[330]:=
```
model = NetModel["LeNet Trained on MNIST Data"]
```

Out[330]=
```
NetChain[
  Input port:            image
  Output port:           class
  Number of layers:      11
]
```

In[331]:=
```
NetInitialize[model, None]
```

Out[331]=
```
NetChain[
  uninitialized
  Input port:            image
  Output port:           class
  Number of layers:      11
]
```

Drop the liner layers and make the resulting fully convolutional net size-independent:

In[332]:=
```
NetReplacePart[NetDrop[model, -4], "Input" → Automatic]
```

Out[332]=
```
NetChain[
  Input port:            3–tensor (size: 1 × ⬚ × ⬚)
  Output port:           vector
  Number of layers:      7
]
```

Replace the activation function:

In[333]:=
```
NetReplace[model, ElementwiseLayer[Ramp] → ElementwiseLayer[Tanh]]
```

Out[333]=
```
NetChain[
  Input port:            image
  Output port:           class
  Number of layers:      11
]
```

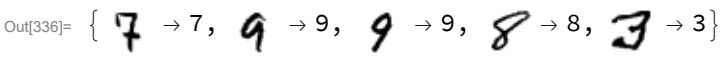
Training a Digit Classifier

Obtain the MNIST dataset, which contains 60,000 training and 10,000 test images:

In[334]:=
```
trainingData = ResourceData["MNIST", "TrainingData"];
testData = ResourceData["MNIST", "TestData"];
```

Display a few random examples from the training set:

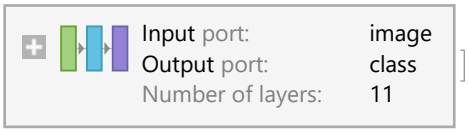
In[336]:=
```
RandomSample[trainingData, 5]
```

Out[336]=


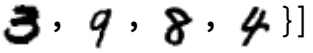
Training a Digit Classifier

We could use a pre-trained version of LeNet from the Wolfram Neural Net Repository:

In[337]:=
```
lenetModel = NetModel["LeNet Trained on MNIST Data"]
```

Out[337]=
```
NetChain[ 
  Input port:            image
  Output port:           class
  Number of layers:      11
]
```

Classify a list of images using the pretrained net:

In[338]:=
```
lenetModel[{  }]
```

Out[338]=
```
{3, 9, 8, 4}
```

But lets do this from scratch, making use of all of the components.

- We are also going to do this in the most general way
  - For a faster way, see the Tutorial MNIST Digit Classification

Training a Digit Classifier

- Define encoders + decoders:

In[339]:=
```
dec = NetDecoder[{"Class", Range[0, 9]}]
```

Out[339]=
```
NetDecoder[
  Type:          Class
  Labels:        {0, 1, ≪7≫, 9}
  Input depth:   1
  Dimensions:    10
]
```

In[340]:=
```
enc = NetEncoder[{"Image", {28, 28}, "Grayscale"}]
```

Out[340]=
```
NetEncoder[
  Type:          Image
  Image size:    {28, 28}
  Color space:   Grayscale
  Color channels: 1
  Mean image:    None
  Variance image: None
  Output:        3–tensor (size: 1 × 28 × 28)
]
```

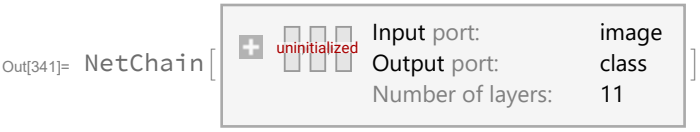
Training a Digit Classifier

- Define a convolutional neural network that takes in 28x28 grayscale images as input:

In[341]:=
```
uninitializedLenet = NetChain[
{
    ConvolutionLayer[20,5],      (*first convolution => 20 feature images*)
    ElementwiseLayer[Ramp],      (*activation function (ReLU) => non-linearity, sparsity
    PoolingLayer[2,2],           (*max pooling => downsampling*)

    ConvolutionLayer[50,5],      (*second convolution => 50 feature images*)
    ElementwiseLayer[Ramp],      (*activation function (ReLU) => non-linearity, sparsity
    PoolingLayer[2,2],           (*max pooling => downsampling*)

    FlattenLayer[],              (*flattening => images to vector*)
    LinearLayer[500],            (*first fully connected layer => feature vector from im
    ElementwiseLayer[Ramp],      (*activation function (ReLU) => non-linearity, sparsity
    LinearLayer[10],             (*second fully connected layer => class prediction*)
    SoftmaxLayer[]               (*normalization*)
},
    "Input"  → enc, (*encoder => image to tensor*)
    "Output" → dec  (*decoder => tensor to class*)
]
```

Out[341]=


Training a Digit Classifier

Construct a NetGraph by supplying a list of layers and connections:

In[342]:=
```
trainingNet = NetInitialize@NetGraph[<|
    "lenet" → uninitializedLenet, "loss" → CrossEntropyLossLayer["Index"]|>,
   {NetPort["Input"] → "lenet" → NetPort["loss", "Input"],
    NetPort["Target"] → NetPort["loss", "Target"]}]
```

Out[342]=
```
NetGraph[ Number of inputs: 2
          Loss port: real
          Number of layers: 2 ]
```

Evaluate the training net on some examples:

In[343]:=
```
trainingNet[⟨|"Input" → {<image>, <image>, <image>, <image>, <image>}, "Target" → {2, 4, 7, 9, 8}|⟩]
```

Out[343]=
```
{3.04486, 1.21394, 7.099, 4.85219, 1.99851}
```

These losses summarize how well LeNet did at predicting the targets when given the images. Because LeNet was randomly initialized, we expect it to be no better than chance (on average). During training, the learnable parameters in LeNet are gradually adjusted to bring the average loss down.

Training a Digit Classifier

Convert the training and test data into association form, using Keys and Values to obtain the images and the labels from the lists of rules in the training and test data:

In[344]:=
```
trainAssoc = <|"Input" → Keys[trainingData],
    "Target" → Values[trainingData] + 1|>;
testAssoc = <|"Input" → Keys[testData], "Target" → Values[testData] + 1|>;
```

Show a small sample of the training association:

In[346]:=
```
Part[trainAssoc, All, {1, 10^4, 4 × 10^4, 5 × 10^4}]
```

Out[346]=

Training a Digit Classifier

Let us now perform the training using NetTrain . Notice several things about this net example:

- We have specified MaxTrainingRounds –> 5, which will scan the entire training set five times before finishing.
- We have given All as the third argument, to obtain a NetTrainResultsObject that summarizes various information about the training session.
- We have given a ValidationSet , which allows us to measure how well the trained classifier generalizes to new examples that it has not trained on. This helps avoid a common pitfall known as "overfitting".
  - See the Tutorial Training Neural Networks with Regularization for extra understanding of overfitting and preventing it
- We have specified TargetDevice –> "CPU" (which is already the default). If you have an NVIDIA graphics card, however, you can change this to "GPU" to achieve a big speedup in training.

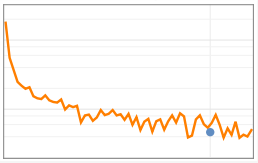
Train LeNet:

In[347]:=
```
results = NetTrain[trainingNet, trainAssoc, All,
  ValidationSet → testAssoc, MaxTrainingRounds → 5, TargetDevice → "CPU"]
```

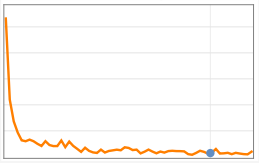
Out[347]=
```
NetTrainResultsObject[
```

Total training time:	1.1 min
Total rounds:	2
Total batches:	1137
Batch size:	64
Method:	ADAM
Final round loss:	0.0523
Final validation loss:	0.0465
Final round error:	1.54%
Final validation error:	1.44%

Loss evolution plot:



Error evolution plot:



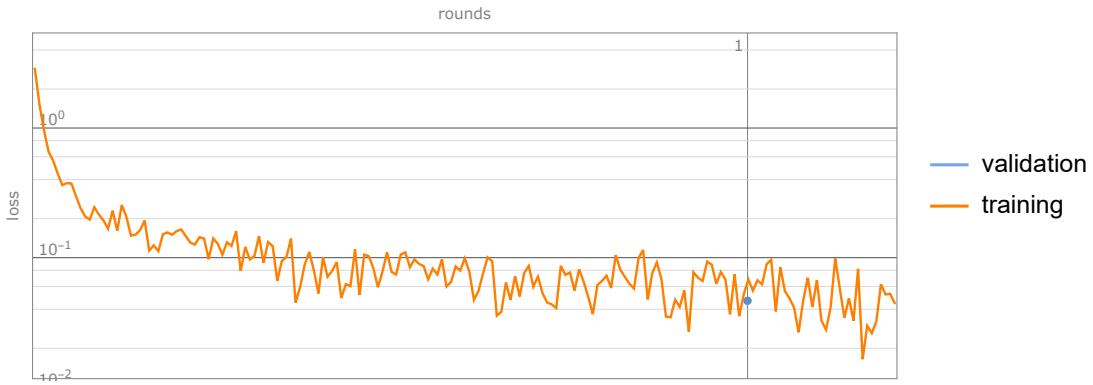
]

Training a Digit Classifier

Obtain the loss evolution plot from the NetTrainResultsObject:

In[348]:=
```
results["LossEvolutionPlot"]
```

Out[348]=



Obtain the trained net from the NetTrainResultsObject and from it extract the prediction network:

In[349]:=
```
trainedLenet = NetExtract[results["TrainedNet"], "lenet"]
```

Out[349]=
```
NetChain[
```
Input port: 3–tensor (size: 1×28×28)
Output port: vector (size: 10)
Number of layers: 11
```
]
```

Reattach the NetEncoder and NetDecoder that were removed when the classification net was embedded inside the training network:

In[350]:=
```
trainedLenet = NetReplacePart[trainedLenet, {"Input" → enc, "Output" → dec}]
```

Out[350]=
```
NetChain[
```
Input port: image
Output port: class
Number of layers: 11
```
]
```

Training a Digit Classifier

Make a classification on an image:

In[351]:=
```
trainedLenet[ ]
```


Out[351]= 2

Obtain the top probabilities for a difficult image:

In[352]:=
```
trainedLenet[ , "TopProbabilities"]
```


Out[352]= {3 → 0.685553, 2 → 0.168939}

Compare with the fully trained model:

In[353]:=
```
NetModel["LeNet Trained on MNIST Data"][ , "TopProbabilities"]
```


Out[353]= {3 → 0.999962}

Evaluate the Digit Classifier

In[354]:=
```
measurements = ClassifierMeasurements[trainedLenet, testData]
```

Out[354]=
```
ClassifierMeasurementsObject[ Classifier: Net
                              Number of test examples: 10 000
                              Data not in notebook; Store now » ]
```

Obtain the overall accuracy:

In[355]:=
```
measurements["Accuracy"]
```

Out[355]=
```
0.9856
```

This could have been done much more simply with Classify:

In[356]:=
```
classifier = Classify[trainingData, ValidationSet → testData]
ClassifierMeasurements[classifier, testData, "Accuracy"]
```

Out[356]=
```
ClassifierFunction[ Input type: Image
                    Number of classes: 10
                    Method: LogisticRegression
                    Number of training examples: 60 000 ]
```

Out[357]=
```
0.8969
```

Summary of the Framework

- The WL Neural Net framework is an industrial-strength, scalable framework
  - uses latest NVIDIA libraries
  - backed by MXNet
  - used extensively internally by Wolfram Research
- It has a great Neural Net Repository
  - curation from frameworks
  - access to a huge collection of nets trained by WL
  - ensures our framework can represent most modern nets
- No lock-in
  - import/export to MXNet
  - hopefully support for the ONNX format
- Easier to use than other frameworks
  - makes developers much more productive
  - true variable-length sequence support

Thank you