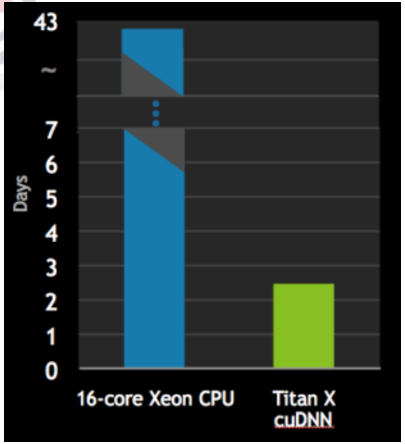
A Practical Introduction to Deep Learning with Caffe

Peter Anderson, ACRV, ANU

Overview

- Some setup considerations
- Caffe tour
- How to do stuff – prepare data, modify a layer

Training AlexNet (src: Nvidia)



Which GPU?

Nvidia GPU	Titan X	Tesla K40	Tesla K80
Tflops SP	6.6	4.29	5.6 (total)
Tflops DP	0.2	1.43	1.87 (total)
ECC support	No	Yes	Yes
Memory	12GB	12GB	2 x 12GB
Price (US$)	$1,000	$3,000	$4,200

Which Framework?

	Caffe	Theano	Torch
Users	BVLC	Montreal	NYU, FB, Google
Core Language	C++	Python	Lua
Bindings	Python, MATLAB		Python, MATLAB
Pros	Pre-trained models, config files	Symbolic differentiation	
Cons	C++ prototyping, weak RNN support		

What is Caffe?

Convolution Architecture For Feature Extraction (CAFFE)

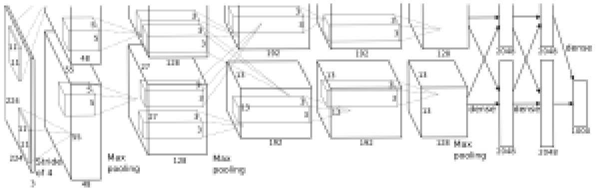
Open framework, models, and examples for deep learning

- 600+ citations, 100+ contributors, 7,000+ stars, 4,000+ forks
- Focus on vision, but branching out
- Pure C++ / CUDA architecture for deep learning
- Command line, Python, MATLAB interfaces
- Fast, well-tested code
- Tools, reference models, demos, and recipes
- Seamless switch between CPU and GPU

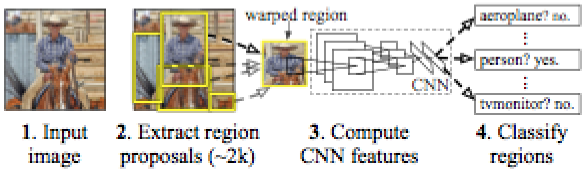
Slide credit: Evan Shelhamer, Jeff Donahue, Jon Long, Yangqing Jia, and Ross Girshick

Reference Models

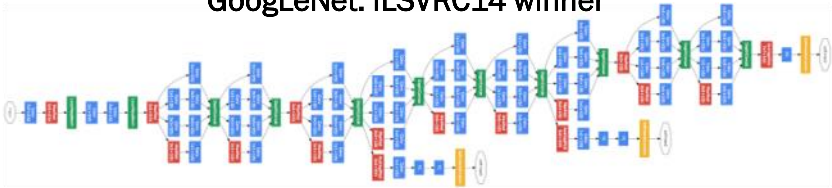
AlexNet: ImageNet Classification



R-CNN: *Regions with CNN features*



GoogLeNet: ILSVRC14 winner



Caffe offers the

- model definitions
- optimization settings
- pre-trained weights

so you can start right away.

The BVLC models are licensed for unrestricted use.

The community shares models in the Model Zoo.

Slide credit: Evan Shelhamer, Jeff Donahue, Jon Long, Yangqing Jia, and Ross Girshick

Open Model Collection

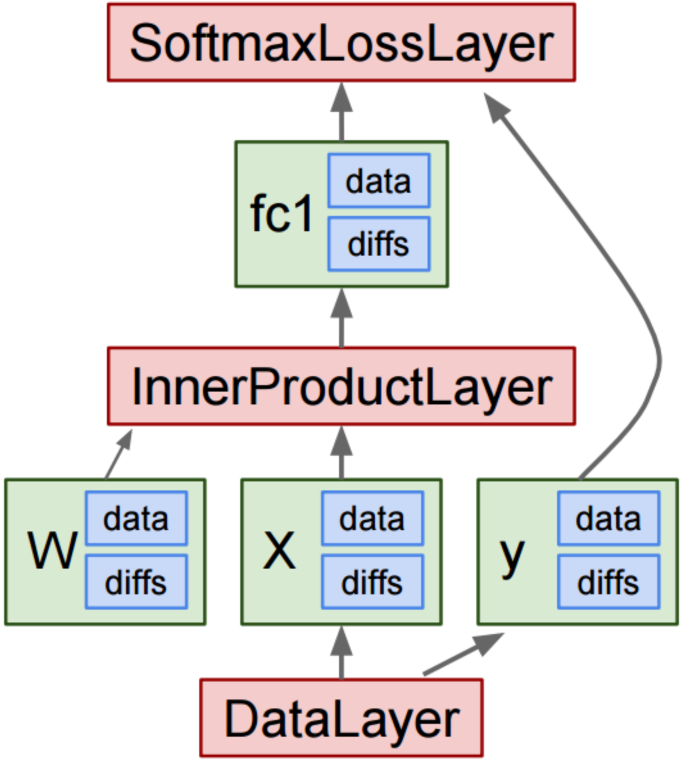
The Caffe Model Zoo
- open collection of deep models to share innovation
    - VGG ILSVRC14 models in the zoo
    - Network-in-Network model in the zoo
    - MIT Places scene recognition model in the zoo
- help disseminate and reproduce research
- bundled tools for loading and publishing models

Share Your Models! with your citation + license of course

Slide credit: Evan Shelhamer, Jeff Donahue, Jon Long, Yangqing Jia, and Ross Girshick

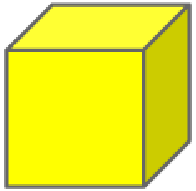
Main Classes

- **Blob:** Stores data and derivatives
- **Layer**: Transforms bottom blobs to top blobs
- **Net:** Many layers; computes gradients via Forward / Backward
- **Solver:** Uses gradients to update weights

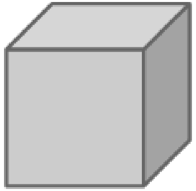


Slide credit: Stanford Vision CS231

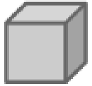
Blobs



Data
*N*umber x *K* Channel x *H*eight x *W*idth
256 x 3 x 227 x 227 for ImageNet train input



Parameter: Convolution Weight
N Output x *K* Input x *H*eight x *W*idth
96 x 3 x 11 x 11 for CaffeNet conv1



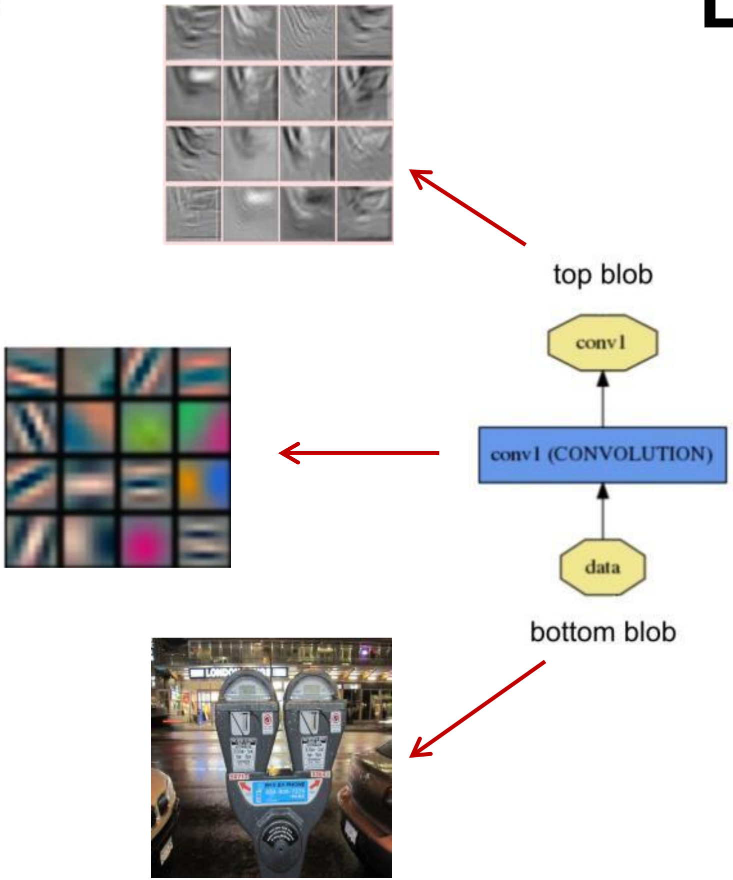
Parameter: Convolution Bias
96 x 1 x 1 x 1 for CaffeNet conv1

N-D arrays for storing and communicating data

- Hold data, derivatives and parameters
- Lazily allocate memory
- Shuttle between CPU and GPU

Slide credit: Evan Shelhamer, Jeff Donahue, Jon Long, Yangqing Jia, and Ross Girshick

Layers

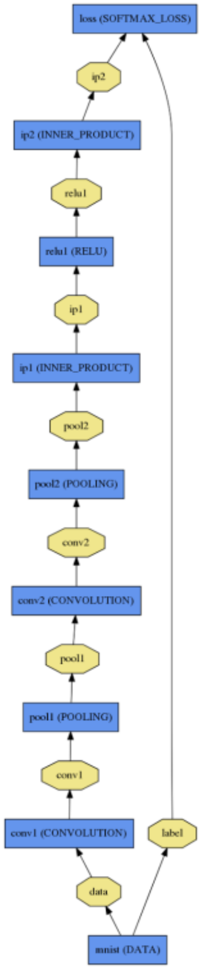


Caffe's fundamental unit of computation

Implemented as layers:

- Data access
- Convolution
- Pooling
- Activation Functions
- Loss Functions
- Dropout
- etc.

Net



Forward: inference

Backward: learning

LeNet →

- A DAG of layers and the blobs that connect them
- Caffe creates and checks the net from a definition file (more later)
- Exposes Forward / Backward methods

Solver

- Calls Forward / Backward and updates net parameters
- Periodically evaluates model on the test network(s)
- Snapshots model and solver state

Solvers available:

- SGD
- AdaDelta
- AdaGrad
- Adam
- Nesterov
- RMSprop

Protocol Buffers

- Like strongly typed, binary JSON!
- Auto-generated code
- Developed by Google
- Net / Layer / Solver / parameters are **messages** defined in .prototxt files
- Available **message types** defined in ./src/caffe/proto/caffe.proto
- Models and solvers are schema, not code

```
message ConvolutionParameter {
  optional uint32 num_output = 1; // The number of outputs for the layer
  optional bool bias_term = 2 [default = true]; // whether to have bias terms

  // Pad, kernel size, and stride are all given as a single value for equal
  // dimensions in all spatial dimensions, or once per spatial dimension.
  repeated uint32 pad = 3; // The padding size; defaults to 0
  repeated uint32 kernel_size = 4; // The kernel size
  repeated uint32 stride = 6; // The stride; defaults to 1

  // For 2D convolution only, the *_h and *_w versions may also be used to
  // specify both spatial dimensions.
  optional uint32 pad_h = 9 [default = 0]; // The padding height (2D only)
  optional uint32 pad_w = 10 [default = 0]; // The padding width (2D only)
  optional uint32 kernel_h = 11; // The kernel height (2D only)
  optional uint32 kernel_w = 12; // The kernel width (2D only)
  optional uint32 stride_h = 13; // The stride height (2D only)
  optional uint32 stride_w = 14; // The stride width (2D only)

  optional uint32 group = 5 [default = 1]; // The group size for group conv

  optional FillerParameter weight_filler = 7; // The filler for the weight
  optional FillerParameter bias_filler = 8; // The filler for the bias
  enum Engine {
    DEFAULT = 0;
    CAFFE = 1;
    CUDNN = 2;
  }
  optional Engine engine = 15 [default = DEFAULT];
}
```

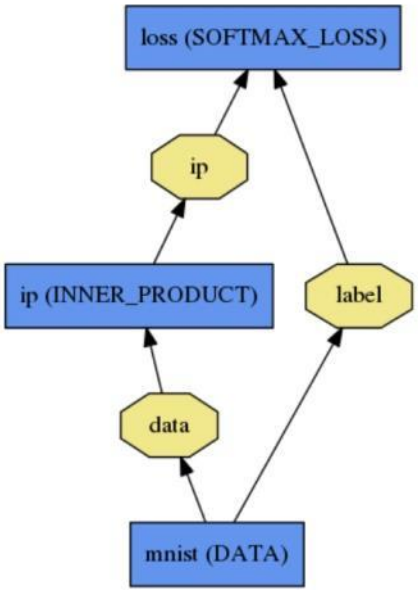
Prototxt: Define Net

Blobs

Number of output classes

Layer type

```
name: "LogReg"
layer {
  name: "mnist"
  type: "Data"
  top: "data"
  top: "label"
  data_param {
    source: "input_leveldb"
    batch_size: 64
  }
}
layer {
  name: "ip"
  type: "InnerProduct"
  bottom: "data"
  top: "ip"
  inner_product_param {
    num_output: 2
  }
}
layer {
  name: "loss"
  type: "SoftmaxWithLoss"
  bottom: "ip"
  bottom: "label"
  top: "loss"
}
```



Prototxt: Layer Detail

Learning rates (weight + bias)
Set these to 0 to freeze a layer

Convolution-specific parameters

Parameter Initialization

```
layer {
  name: "conv1"
  type: "Convolution"
  bottom: "data"
  top: "conv1"
  # learning rate and decay multipliers for the filters
  param { lr_mult: 1 decay_mult: 1 }
  # learning rate and decay multipliers for the biases
  param { lr_mult: 2 decay_mult: 0 }
  convolution_param {
    num_output: 96     # learn 96 filters
    kernel_size: 11    # each filter is 11x11
    stride: 4          # step 4 pixels between each filter application
    weight_filler {
      type: "gaussian" # initialize the filters from a Gaussian
      std: 0.01        # distribution with stdev 0.01 (default mean: 0)
    }
    bias_filler {
      type: "constant" # initialize the biases to zero (0)
      value: 0
    }
  }
}
```

Example from ./models/bvlc_reference_caffenet/train_val.prototxt

Prototxt: Define Solver

Test on validation set

Learning rate profile

Net prototxt

Snapshots during training

```
test_iter: 100
test_interval: 500
base_lr: 0.01
display: 100
max_iter: 10000
lr_policy: "inv"
gamma: 0.0001
power: 0.75
momentum: 0.9
weight_decay: 0.0005
solver_mode: GPU
net: "examples/mnist/lenet_train_test.prototxt"
# The snapshot interval in iterations.
snapshot: 5000
# File path prefix for snapshotting model weights and solver state.
# Note: this is relative to the invocation of the `caffe` utility, not the
# solver definition file.
snapshot_prefix: "/path/to/model"
# Snapshot the diff along with the weights. This can help debugging training
# but takes more storage.
snapshot_diff: false
# A final snapshot is saved at the end of training unless
# this flag is set to false. The default is true.
snapshot_after_train: true
```

Setting Up Data

- Prefetching
- Multiple Inputs
- Data augmentation on-the-fly (random crops, flips) – see TransformationParameter proto

Choice of Data Layers:

- Image files
- LMDB
- HDF5

Interfaces

python

```
out = net.forward()
```

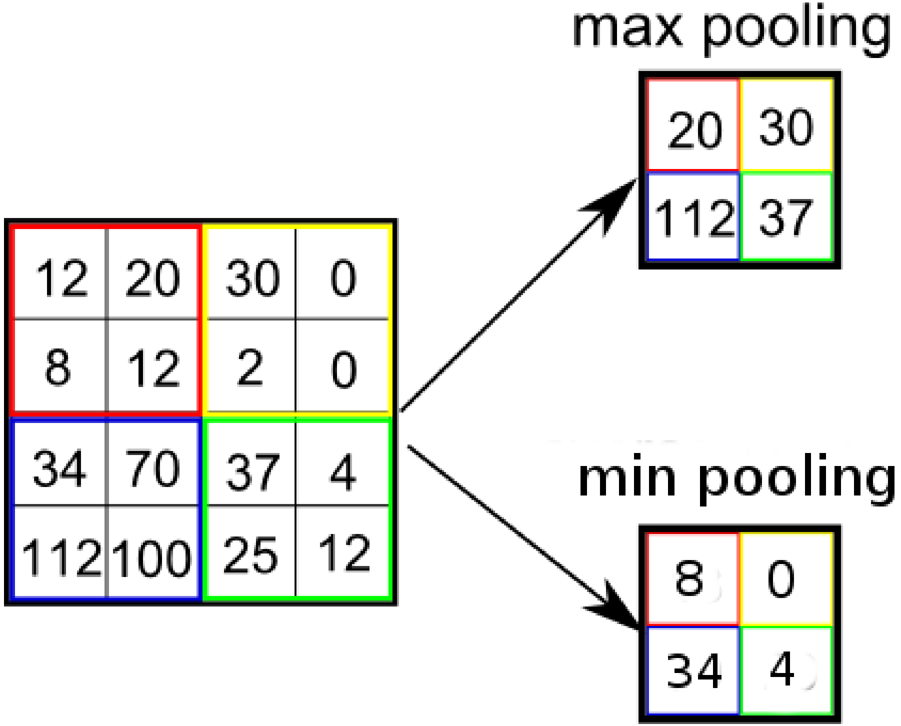
- Blob data and diffs exposed as Numpy arrays
- ./python/caffe/_caffe.cpp: Exports Blob, Layer, Net & Solver classes
- ./python/caffe/pycaffe.py: Adds extra methods to Net class
- Jupyter notebooks: ./examples

MATLAB

```
scores = net.forward(input_data);
```

- Similar to PyCaffe in usage
- Demo: ./matlab/demo/classification_demo.m
- Images are in BGR channels

Example: Modifying a Layer



Suppose you need a Min-Pooling Layer

Modifications:

./src/caffe/proto/caffe.proto
./include/caffe/vision_layers.hpp
./src/caffe/layers/pooling_layer.cpp
./src/caffe/layers/pooling_layer.cu
./src/caffe/layers/cudnn_pooling_layer.cpp
./src/caffe/layers/cudnn_pooling_layer.cu
./src/caffe/test/test_pooling_layer.cpp

Tip – many existing math functions:
./include/caffe/util/math_functions.hpp

Example: Modifying a Layer

Add new parameter to message type

```
message PoolingParameter {
  enum PoolMethod {
    MAX = 0;
    AVE = 1;
    STOCHASTIC = 2;
    SUM = 3;
    MIN = 4;
  }
  optional PoolMethod pool = 1 [default = MAX]; // The pooling method
  // Pad, kernel size, and stride are all given as a single value for equal
  // dimensions in height and width or as Y, X pairs.
  optional uint32 pad = 4 [default = 0]; // The padding size (equal in Y, X)
  optional uint32 pad_h = 9 [default = 0]; // The padding height
  optional uint32 pad_w = 10 [default = 0]; // The padding width
  optional uint32 kernel_size = 2; // The kernel size (square)
  optional uint32 kernel_h = 5; // The kernel height
  optional uint32 kernel_w = 6; // The kernel width
  optional uint32 stride = 3 [default = 1]; // The stride (equal in Y, X)
  optional uint32 stride_h = 7; // The stride height
  optional uint32 stride_w = 8; // The stride width
  enum Engine {
    DEFAULT = 0;
    CAFFE = 1;
    CUDNN = 2;
  }
  optional Engine engine = 11 [default = DEFAULT];
  // If global_pooling then it will pool over the size of the bottom by doing
  // kernel_h = bottom->height and kernel_w = bottom->width
  optional bool global_pooling = 12 [default = false];
}
```

See ./src/caffe/proto/caffe.proto

Example: Modifying a Layer

```
template <typename Dtype>
void PoolingLayer<Dtype>::Forward_gpu(const vector<Blob<Dtype>*>& bottom,
      const vector<Blob<Dtype>*>& top) {
  const Dtype* bottom_data = bottom[0]->gpu_data();
  Dtype* top_data = top[0]->mutable_gpu_data();
  int count = top[0]->count();
  // We'll output the mask to top[1] if it's of size >1.
  const bool use_top_mask = top.size() > 1;
  int* mask = NULL;
  Dtype* top_mask = NULL;
  switch (this->layer_param_.pooling_param().pool()) {
  case PoolingParameter_PoolMethod_MIN:
    if (use_top_mask) {
      top_mask = top[1]->mutable_gpu_data();
    } else {
      mask = max_idx_.mutable_gpu_data();
    }
    // NOLINT_NEXT_LINE(whitespace/operators)
    MinPoolForward<Dtype><<<CAFFE_GET_BLOCKS(count), CAFFE_CUDA_NUM_THREADS>>>(
        count, bottom_data, bottom[0]->num(), channels_,
        height_, width_, pooled_height_, pooled_width_, kernel_h_,
        kernel_w_, stride_h_, stride_w_, pad_h_, pad_w_, top_data,
        mask, top_mask);
    break;
  case PoolingParameter_PoolMethod_MAX:
```

Add new switch block for min-pooling

See ./src/caffe/layers/pooling_layer.cu

Example: Modifying a Layer

Caffe macros make cuda programming easy

Almost identical to max-pooled version

```
template <typename Dtype>
__global__ void MinPoolForward(const int nthreads,
    const Dtype* const bottom_data, const int num, const int channels,
    const int height, const int width, const int pooled_height,
    const int pooled_width, const int kernel_h, const int kernel_w,
    const int stride_h, const int stride_w, const int pad_h, const int pad_w,
    Dtype* const top_data, int* mask, Dtype* top_mask) {
  CUDA_KERNEL_LOOP(index, nthreads) {
    const int pw = index % pooled_width;
    const int ph = (index / pooled_width) % pooled_height;
    const int c = (index / pooled_width / pooled_height) % channels;
    const int n = index / pooled_width / pooled_height / channels;
    int hstart = ph * stride_h - pad_h;
    int wstart = pw * stride_w - pad_w;
    const int hend = min(hstart + kernel_h, height);
    const int wend = min(wstart + kernel_w, width);
    hstart = max(hstart, 0);
    wstart = max(wstart, 0);
    Dtype minval = FLT_MAX;
    int minidx = -1;
    const Dtype* const bottom_slice =
        bottom_data + (n * channels + c) * height * width;
    for (int h = hstart; h < hend; ++h) {
      for (int w = wstart; w < wend; ++w) {
        if (bottom_slice[h * width + w] < minval) {
          minidx = h * width + w;
          minval = bottom_slice[minidx];
        }
      }
    }
    top_data[index] = minval;
    if (mask) {
      mask[index] = minidx;
    } else {
      top_mask[index] = minidx;
    }
  }
}
```

See ./src/caffe/layers/pooling_layer.cu

Always Write ≥2 Tests!

- Test the gradient is correct

```
LayerParameter layer_param;
PoolingParameter* pooling_param = layer_param.mutable_pooling_param();
pooling_param->set_kernel_h(kernel_h);
pooling_param->set_kernel_w(kernel_w);
pooling_param->set_stride(2);
pooling_param->set_pad(1);
pooling_param->set_pool(PoolingParameter_PoolMethod_MAX);
PoolingLayer<Dtype> layer(layer_param);
GradientChecker<Dtype> checker(1e-4, 1e-2);
checker.CheckGradientExhaustive(&layer, this->blob_bottom_vec_,
    this->blob_top_vec_);
```

- Test a small worked example

```
layer.Forward(blob_bottom_vec_, blob_top_vec_);
// Expected output: 2x 2 channels of:
//     [9 5 5 8]
//     [9 5 5 8]
for (int i = 0; i < 8 * num * channels; i += 8) {
  EXPECT_EQ(blob_top_->cpu_data()[i + 0], 9);
  EXPECT_EQ(blob_top_->cpu_data()[i + 1], 5);
  EXPECT_EQ(blob_top_->cpu_data()[i + 2], 5);
  EXPECT_EQ(blob_top_->cpu_data()[i + 3], 8);
  EXPECT_EQ(blob_top_->cpu_data()[i + 4], 9);
  EXPECT_EQ(blob_top_->cpu_data()[i + 5], 5);
  EXPECT_EQ(blob_top_->cpu_data()[i + 6], 5);
  EXPECT_EQ(blob_top_->cpu_data()[i + 7], 8);
}
```

Links

More Caffe tutorials:
http://caffe.berkeleyvision.org/tutorial/
http://tutorial.caffe.berkeleyvision.org/ (@CVPR)

These slides available at:
http://panderson.me