CAFFE-DEEP LEARNING FRAMEWORK

Omair Hassaan
LUMS MS(CS)

omair.hassaan@gmail.com

LAYOUT

1

- Brief Introduction
- Basic requirements for setting up Caffe Environment

2

- Testing Caffe Installation
- Train LeNet Architecture on MNIST Data set

3

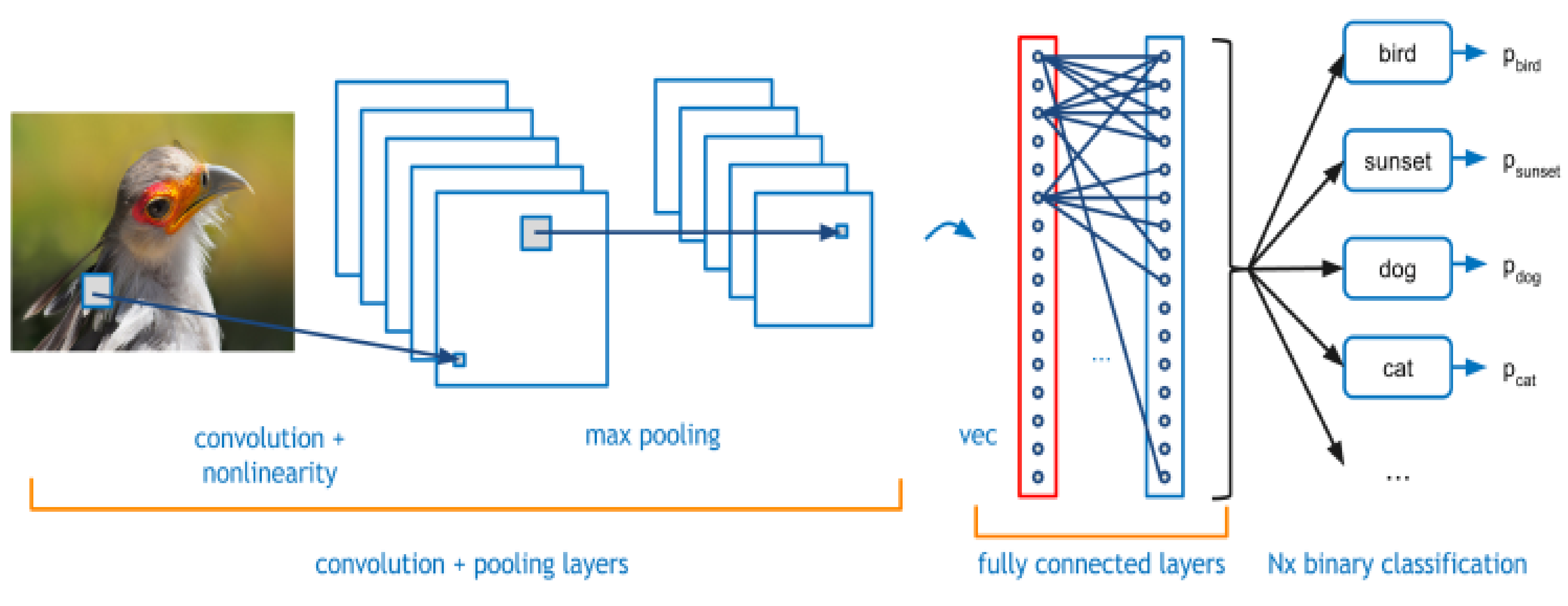
- Training your own Dataset
- Testing a trained Model

4

- Filter Visualization
- Fine Tuning of a Trained Model.

What is Caffe?

- Caffe = Convolution Architecture for Fast Feature Embedding



What is Caffe?

- Open framework for Deep learning.
- 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
- Offers model definitions, optimization settings and pretrained weights.

LAYOUT

1
- Brief Introduction
- Basic requirements for setting up Caffe Environment

2
- Testing Caffe Installation
- Train LeNet Architecture on MNIST Data set

3
- Training your own Dataset
- Testing a trained Model

4
- Filter Visualization
- Fine Tuning of a Trained Model.

Setting Up Caffe Environment

- Core2duo/i3/i5/i7 machine
- Sufficient amount of RAM
- GPU- shared/dedicated. (Caffe can also work without GPU)
- Dual boot Linux distribution. (recommended)
- Internet connection
- Patience !

LAYOUT

1
- Brief Introduction
- Basic requirements for setting up Caffe Environment

2
- Testing Caffe Installation
- Train LeNet Architecture on MNIST Data set

3
- Training your own Dataset
- Testing a trained Model

4
- Fine Tuning of a Trained Model.

Testing Installation

- Download the ImageNet Caffe model and labels
  - ./scripts/download_model_binary.py models/bvlc_reference_caffenet
  - ./data/ilsvrc12/get_ilsvrc_aux.sh
- First Download ilsvrc12 package
  - $CAFFE_ROOT/data/get_ilsvrc_aux.sh
  - cp $CAFFE_ROOT/caffe_ilsvrc12/synset_words.txt $CAFFE_ROOT/data/ilsvrc12/
- Test your installation by running the ImageNet model on an image of a kitten
  - python python/classify.py --print_results examples/images/cat.jpg foo
  - Expected result: [('tabby', '0.27933'), ('tiger cat', '0.21915'), ('Egyptian cat', '0.16064'), ('lynx', '0.12844'), ('kit fox', '0.05155')]

LAYOUT

1

- Brief Introduction
- Basic requirements for setting up Caffe Environment

2

- Testing Caffe Installation
- Train LeNet Architecture on MNIST Data set

3

- Training your own Dataset
- Testing a trained Model

4

- Filter Visualization
- Fine Tuning of a Trained Model.

Training LeNet on MNIST dataset

- Change directory to $CAFFE_ROOT (/home/omair/caffe-master)
- Prepare dataset
  - ./data/mnist/get_mnist.sh
  - ./examples/mnist/create_mnist.sh
- We will get 2 lmdb files from the above step
  - mnist_train_lmdb (examples/mnist/)
  - mnist_test_lmdb
- All training layers are written in *.protoxt files
  - Data/Convolution/Pooling/Inner product/ReLU/Loss
- Define Mnist solver file for training/test protocol buffer definition
  - $CAFFE_ROOT/examples/mnist/lenet_solver.prototxt
  - Contains training iterations, base learning rate, max # iterations and solver mode

Network Layers

- We define our network architecture in "prototxt " files
- There are 2 kinds of protoxt files.

- TRAIN_TEST.PROTOTXT
    - Define Convolution, Pooling, Softmax layers

- FOO_SOLVER.PROTOTXT
    - Defines Learning parameters of our model.

Network Layers(Vision Layers)

- Vision layers usually take images as input and produce images as output.

- Most of the vision layers work by applying a particular operation to some region of the input to produce a corresponding region of the output

- Examples
    - Convolution Layer
    - Pooling Layer
    - Max layer ..

Convolution Layer

- Layer type: Convolution
- CPU implementation:
  - $CAFFE_ROOT/src/caffe/layers/convolution_layer.cpp
- Parameters (ConvolutionParameter convolution_param)
- num_output (c_o):
  - the number of filters
- kernel_size :
  - specifies height and width of each filter
- weight_filler [default type: 'constant' value: 0]
- bias_term [default true]:
  - specifies whether to learn and apply a set of additive biases to the filter outputs
- pad [default 0]:
  - specifies the number of pixels to (implicitly) add to each side of the input
- stride [default 1]:
  - specifies the intervals at which to apply the filters to the input

```
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
    }
  }
}
```

Pooling Layer

- Layer type: Pooling
- CPU implementation:
  - $CAFFE_ROOT/src/caffe/layers/pooling_layer.cpp
- Parameters (PoolingParameter pooling_param)
- kernel_size (or kernel_h and kernel_w):
  - specifies height and width of each filter
- pool [default MAX]:
  - the pooling method. Currently MAX, AVE, or STOCHASTIC
- pad (or pad_h and pad_w) [default 0]: specifies the number of pixels to (implicitly) add to each side of the input
- stride (or stride_h and stride_w) [default 1]:
  - specifies the intervals at which to apply the filters to the input
- pad (or pad_h and pad_w) [default 0]:
  - specifies the number of pixels to (implicitly) add to each side of the input
- stride (or stride_h and stride_w) [default 1]:
  - specifies the intervals at which to apply the filters to the input

```
layer {
  name: "pool1"
  type: "Pooling"
  bottom: "conv1"
  top: "pool1"
  pooling_param {
    pool: MAX
    kernel_size: 3 # pool over a 3x3 region
    stride: 2      # step two pixels (in the bottom blob) between pooling regions
  }
}
```

Loss Layer

- Loss drives learning by comparing an output to a target and assigning cost to minimize
- Softmax:
  - Layer-type: SoftmaxWithLoss
- Sum of Squares/Euclidean
  - Layer-type: EuclideanLoss
- Hinge/Margin
  - Layer type: HingeLoss

```
# L1 Norm
layer {
  name: "loss"
  type: "HingeLoss"
  bottom: "pred"
  bottom: "label"
}

# L2 Norm
layer {
  name: "loss"
  type: "HingeLoss"
  bottom: "pred"
  bottom: "label"
  top: "loss"
  hinge_loss_param {
    norm: L2
  }
}
```

ReLu Layer (Neuron Layer)

- Neuron layers are element-wise operators, taking one bottom blob and producing one top blob of the same size.
- Layer type: ReLU
  - Sigmoid, TanH
  - http://caffe.berkeleyvision.org/tutorial/laye
- CPU implementation:
  - $CAFFE_ROOT/src/caffe/layers/relu_layer.cpp
- Parameters (ReLUParameter relu_param)
- negative_slope [default 0]:
  - specifies whether to leak the negative part by multiplying it with the slope value rather than setting it to 0.

```
layer {
  name: "relu1"
  type: "ReLU"
  bottom: "conv1"
  top: "conv1"
}
```

SOLVER.PROTOTXT

```
base_lr: 0.01     # begin training at a learning rate of 0.01 = 1e-2

lr_policy: "step" # learning rate policy: drop the learning rate in "steps"
                  # by a factor of gamma every stepsize iterations

gamma: 0.1        # drop the learning rate by a factor of 10
                  # (i.e., multiply it by a factor of gamma = 0.1)

stepsize: 100000  # drop the learning rate every 100K iterations

max_iter: 350000  # train for 350K iterations total

momentum: 0.9
```

Training LeNet on MNIST dataset

- Change directory to $CAFFE_ROOT (/home/omair/caffe-master)
- Prepare dataset
  - ./data/mnist/get_mnist.sh
  - ./examples/mnist/create_mnist.sh
- We will get 2 lmdb files from the above step
  - mnist_train_lmdb (examples/mnist/)
  - mnist_test_lmdb
- All training layers are written in *.protoxt files
  - Data/Convolution/Pooling/Inner product/ReLU/Loss
- Define Mnist solver file for training/test protocol buffer definition
  - $CAFFE_ROOT/examples/mnist/lenet_solver.prototxt
  - Contains training iterations, base learning rate, max # iterations and solver mode

Training LeNet on MNIST dataset

- Start training using
  - $CAFFE_ROOT/examples/mnist/train_lenet.sh
  - ./build/tools/caffe train --solver=examples/mnist/lenet_solver.prototxt
- End result will be a *.caffemodel file containing our trained network.

1

- Brief Introduction
- Basic requirements for setting up Caffe Environment

2

- Testing Caffe Installation
- Train LeNet Architecture on MNIST Data set

3

- Training your own Dataset
- Testing a trained Model

4

- Filter Visualization
- Fine Tuning of a Trained Model.

Before Starting Training

Creating LMDB with Image dataset

- Create 2 files containing [path/to/image <space> label]
  - Train.txt
  - Val.txt
- Reference example
  - $CAFFE_ROOT/examples/imagenet/create_imagenet.sh
- Download ilsvrc12 package by running
  - $CAFFE_ROOT/data/get_ilsvrc_aux.sh
  - Open $CAFFE_ROOT/data/ilsvrc12/train.txt or val.txt
- Run the *convert_imageset* program to convert all the images to lmdb format using
  - $CAFFE_ROOT/build/tools/convert_imageset [FLAGS] ROOTFOLDER LISTFILE DB_NAME
  - FLAGS can be gray, shuffle, backend, resize_width, resize_height, check_size, encoded, encoded type.
  - ROOTFOLDER = folder containing all the images
  - LISTFILE = train.txt/val.txt for creating training/testing database

Create Mean of Image Set (binaryproto)

- Convert/Crop all images to 227x227 or 256x256
- Create LMDB of all images

- CAFFE_ROOT=/home/omair/caffe-master
- LMDB=/home/omair/caffe-master/lmdb_creation/ilsvrc12_train_lmdb
- OUTPUT=/home/omair/caffe-master/lmdb_creation/meanilsvrc.binaryproto

- $CAFFE_ROOT/build/tools/compute_image_mean $LMDB $OUTPUT

Training with your own Dataset

- After creating your own LMDB database, it's time to create prototxt files containing all the layers information of your network.
  - References
  - $CAFFE_ROOT/examples/mnist/lenet.prototxt
  - $CAFFE_ROOT/examples/minst/lenet_train_test.prototxt
  - $CAFFE_ROOT/examples/mnist/lenet_solver.prototxt
  - $CAFFE_ROOT/examples/mnist/train_lenet.sh
- Change path of prototxt files in train_lenet.sh file(see “Training LeNet on MNIST using Caffe” for reference)
- Start training by running train_lenet.sh

1

- Brief Introduction
- Basic requirements for setting up Caffe Environment

2

- Testing Caffe Installation
- Train LeNet Architecture on MNIST Data set

3

- Training your own Dataset
- Testing a trained Model

4

- Filter Visualization
- Fine Tuning of a Trained Model.

Testing a Trained Model (C++)

- Use C++ API to implement image classification.
- Sample C++ code is present in $CAFFE_ROOT/examples/cpp_classification/classification.cpp
- Gets compiled automatically while compiling caffe
- Input arguments
  - Model file (Defining layer architecture)
  - Trained file
  - Mean file (Mean of dataset)
  - Labels (input labels)
  - Test Image.

Testing a Trained Model (C++)

- CAFFE_ROOT=/home/omair/caffe-master
- PROTOTXT=$CAFFE_ROOT/models/bvlc_reference_caffenet/deploy.prototxt
- CAFFEMODEL=$CAFFE_ROOT/models/bvlc_reference_caffenet/bvlc_reference_caffenet.caffemodel
- MEANBINARYPROTO=$CAFFE_ROOT/data/ilsvrc12/imagenet_mean.binaryproto
- LABELS=$CAFFE_ROOT/data/ilsvrc12/synset_words.txt

- $CAFFE_ROOT/build/examples/cpp_classification/classification.bin $PROTOTXT $CAFFEMODEL $MEANBINARYPROTO $LABELS $CAFFE_ROOT/examples/images/cat.jpg

Testing a Trained Model (Python)

- Create a LMDB database of the testing image set.
- You must have prototxt files available and also the snapshot of trained model.
- Make a directory named “test”
- Copy “test_your_own_model.py” (located in repository) in the newly created directory
- Change paths of protoxt files, mean file, trained model.
- Run this code to test your model.
  - python test_your_own_model.py

1

- Brief Introduction
- Basic requirements for setting up Caffe Environment

2

- Testing Caffe Installation
- Train LeNet Architecture on MNIST Data set

3

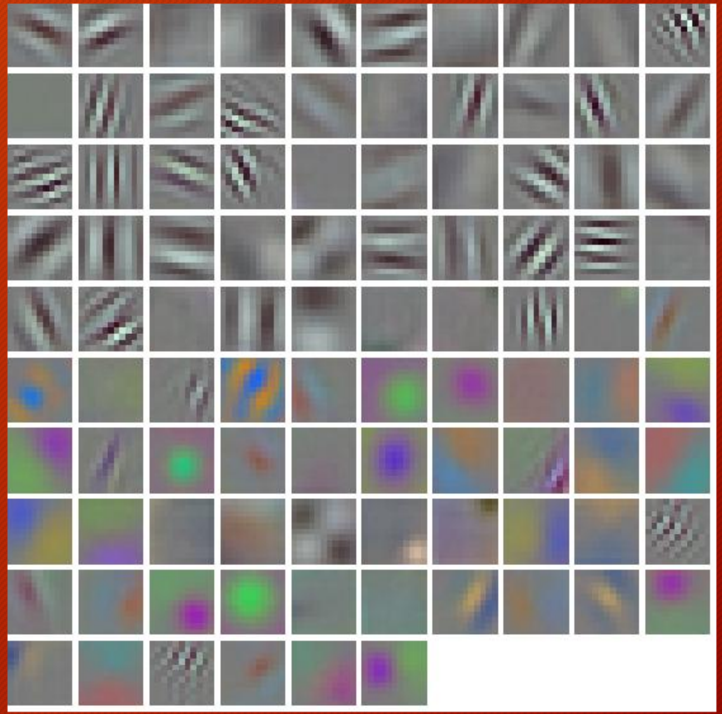
- Training your own Dataset
- Testing a trained Model

4

- Filter Visualization
- Fine Tuning of a Trained Model.

Filter Visualization

- We can visualize filters used in any layer using python script.
- Download "filter_visualization.py" from github respository
- Copy this code to "test" directory.
- Run !
  - python filter_visualization.py



1.
   - Brief Introduction
   - Basic requirements for setting up Caffe Environment
2.
   - Testing Caffe Installation
   - Train LeNet Architecture on MNIST Data set
3.
   - Training your own Dataset
   - Testing a trained Model
4.
   - Filter Visualization
   - Fine Tuning of a Trained Model.

Fine Tuning of a Trained Model

- Takes an already trained model, adapts the architecture and resumes training from model weights.
- Instead of training 1,000,000 images + our new dataset from scratch, lets use already trained model of 1,000,000 images and prepare it for our problem.
- Change last layer (Fully connected layer) in our prototxt file.
- Decrease overall learning rate.
- Increase gamma.
- Idea is, that rest of model should change slowly with new data, but new layers should be learnt fast.

Fine Tuning of a Trained Model

- Create lmdb of train/test image dataset.
- Create label files for train/test dataset.
- Edit train_val.protoxt from bvlc_reference_caffenet problem.
- All required files are available in models/finetune_flicker_style.
- Start fine tunning process
  - $CAFFE_ROOT /build/tools/caffe train –solver models/finetune_flicker_style/solver.prototxt –weights models/bvlc_reference_caffe_net/bvlc_reference_caffenet.caffemodel

```
layer {
  name: "fc8_flickr"
  type: "InnerProduct"
  bottom: "fc7"
  top: "fc8_flickr"
  # lr_mult is set to higher than for other layers, because this layer i
  param {
    lr_mult: 10
    decay_mult: 1
  }
  param {
    lr_mult: 20
    decay_mult: 0
  }
  inner_product_param {
    num_output: 20
    weight_filler {
      type: "gaussian"
      std: 0.01
    }
    bias_filler {
      type: "constant"
      value: 0
    }
  }
}
layer {
  name: "loss"
  type: "SoftmaxWithLoss"
  bottom: "fc8_flickr"
  bottom: "label"
  top: "loss"
}
```