Machine Learning
A Quantitative Approach

Henry H. Liu

PerfMath

ISBN-13: 978-1985136625
ISBN-10: 1985136627

10 9 8 7 6 5 4 3 2 1
03092018

Appendix C CNN Example Implementations with Caffe

This appendix demonstrates a few example CNN implementations with Caffe. We choose the Caffe deep learning framework, as it is one of the most popular frameworks for solving computer vision related tasks. Besides, if you decide to have a career in machine learning, you will have a huge advantage if you have good programming skills in Python and C++. However, you don't have to be a C++ expert to try out the example CNN models with Caffe to be introduced in this appendix. Some basic knowledge about how C++ works in general and how Unix shell scripts work would be sufficient.

C.1 Building the Caffe Framework from the Source

First, let's see how we can build the Caffe framework from the source. By going through such a process, you will have the following benefits:

- You will understand what other software packages that Caffe depends on.
- You will have access to the C++ source files of Caffe, just in case you want to check out how this popular, production quality framework is implemented in C++.
- As a machine learning engineer, it's important that you can quickly get a framework up and running on your machine and start to get your project going immediately.

Next, I'll share with you how I rebuilt Caffe on my MacBook Pro by following the instructions given at http://caffe.berkeleyvision.org/install_osx.html, especially, what worked and what didn't work, and how I worked around the issues I encountered. If you go along yourself, it may take you several days, but with the help of this appendix, it could be much easier for you, especially if you are not very familiar with C++ and a typical Unix-like environment. Of course, if you are already a C++ professional, it would be easy for you.

The installation of Caffe starts with installing some general dependencies. On macOS, you need to have *homebrew* installed on your machine first. If you do not have homebrew installed already on your machine, search online and get it installed first.

Then, follow the below procedure:

1. Download the latest Caffe source at https://github.com/BVLC/caffe and place it in a directory on your machine. For example, I downloaded and placed it on my machine at */Users/henryliu/mspc/devs/ws_cpp/Caffe*. This is my Eclipse C/C++ workspace directory, as I can navigate and view various files easily on such an IDE. Also, add a line in your `.bashrc` file, e.g., *export CAFFE_ROOT=/Users/henryliu/mspc/devs/ws_cpp/Caffe*, to set the CAFFE_ROOT environment variable. You will need this when you try out some of the CNN models on Caffe later. In case you are not familiar with Unix environment, execute "`source ~/.bashrc`" on a command terminal to enable all environment variables defined in that file.
2. Execute "cd $CAFFE_ROOT" on the command terminal and then execute *brew install -vd snappy leveldb gflags glog szip lmdb* by copying this command from that website to your local command terminal. Table C.1 describes what these dependencies are about.
3. The next command to execute is: *brew tap homebrew/science*, which did not work on my machine as it does not exist anymore. It turned out that you can just ignore it.
4. Execute *brew install hdf5 opencv*. We already mentioned what HDF5 is in Table C.1. Check out what *opencv* is about from Table C.1.
5. I don't use Anaconda since it once messed up my Python environment on my machine. Therefore, I chose the option of no Anaconda for the next part of the installation.
6. Execute *brew install --build-from-source --with-python -vd protobuf*. Check out what *protobuf* is about from Table C.1.
7. Execute *brew install --build-from-source -vd boost boost-python*. Check out what *boost* is about from Table C.1.
8. Execute *brew install protobuf boost*.
9. Next, it mentions that BLAS is already installed as the Accelerate/vecLib framework, which is Apple's implementation of BLAS. Check out what BLAS is about from Table C.1.

The dependency installation is completed now. Next, compile Caffe by following the procedure given after Table C.1.

Table C.1 Caffe dependencies

Feature	Semantics
snappy	A fast compressor/decompressor written in C++.
leveldb	A fast key-value storage library written in C++ at Google that provides an ordered mapping from string keys to string values.
gflags	A C++ library that implements commandline flags processing.
glog	C++ implementation of the Google logging module.
szip	Provides lossless compression of scientific data from HDF5, which is a unique technology suite that makes possible the management of extremely large and complex data collections.
lmdb	A Btree-based Lightning Memory-Mapped Database Manager (LMDB).

opencv	OpenCV stands for Open Source Computer Vision Library. Written in optimized C/C++, the library can take advantage of multi-core processing. Enabled with OpenCL, it can take advantage of the hardware acceleration of the underlying heterogeneous compute platform.
protobuf	Protocol Buffers - Google's data interchange format.
boost	Over 80 C++ based individual libraries for tasks and data structures such as linear algebra, pseudorandom number generation, multithreading, image processing, regular expressions, and unit testing.
BLAS	The BLAS (Basic Linear Algebra Subprograms) are routines that provide standard building blocks for performing basic vector and matrix operations. The Level 1 BLAS perform scalar, vector and vector-vector operations, the Level 2 BLAS perform matrix-vector operations, and the Level 3 BLAS perform matrix-matrix operations. Because the BLAS are efficient, portable, and widely available, they are commonly used in the development of high quality linear algebra software.

To compile Caffe, it became tricky in my case. I followed the instructions under *Compilation with Make* and it ended up with the following error:

```
Undefined symbols for architecture x86_64:
  "cv::imread(cv::String const&, int)", referenced from:
      caffe::WindowDataLayer<float>::load_batch(caffe::Batch<float>*) in window_data_layer.o
      caffe::WindowDataLayer<double>::load_batch(caffe::Batch<double>*) in window_data_layer.o
      caffe::ReadImageToCVMat(std::__1::basic_string<char, std::__1::char_traits<char>, std::__1::allocator<char> >
const&, int, int, bool) in io.o
  "cv::imdecode(cv::_InputArray const&, int)", referenced from:
      caffe::DecodeDatumToCVMatNative(caffe::Datum const&) in io.o
      caffe::DecodeDatumToCVMat(caffe::Datum const&, bool) in io.o
  "cv::imencode(cv::String const&, cv::_InputArray const&, std::__1::vector<unsigned char,
std::__1::allocator<unsigned char> >&, std::__1::vector<int, std::__1::allocator<int> > const&)", referenced from:
      caffe::ReadImageToDatum(std::__1::basic_string<char, std::__1::char_traits<char>, std::__1::allocator<char> >
const&, int, int, int, bool, std::__1::basic_string<char, std::__1::char_traits<char>, std::__1::allocator<char> >
const&, caffe::Datum*) in io.o
ld: symbol(s) not found for architecture x86_64
clang: error: linker command failed with exit code 1 (use -v to see invocation)
make: *** [.build_release/lib/libcaffe.so.1.0.0] Error 1
```

I spent a lot of time searching online and nothing helped. Then, it worked when I followed the instructions under *CMake Build*. Therefore, the procedure given below is based on my experience with *CMake Build*:

- *cd $CAFFE_ROOT*
- *cp Makefile.config.example Makefile.config*. Then, in my case, I opened the Makefile.config file and made two changes:
  - Uncommented the line of `CPU_ONLY := 1`, since I do not have a GPU on my machine.
  - Uncommented the lines for using Python 3 instead of 2.

- Then, I executed each of the following commands as instructed:

```
$mkdir build
$cd build
$cmake ..
$make all
$make install
$make runtest
```

All of the above commands were successful. However, I tried the command *make distribute* and encountered the error of "*target not defined*." This was okay as I wanted to run Caffe on my local machine anyway. Figure C.1 shows the code structure of the Caffe framework on my C/C++ Eclipse IDE.

If you have gotten to this step, you are ready to try out a few example CNN models as described in the next few sections.

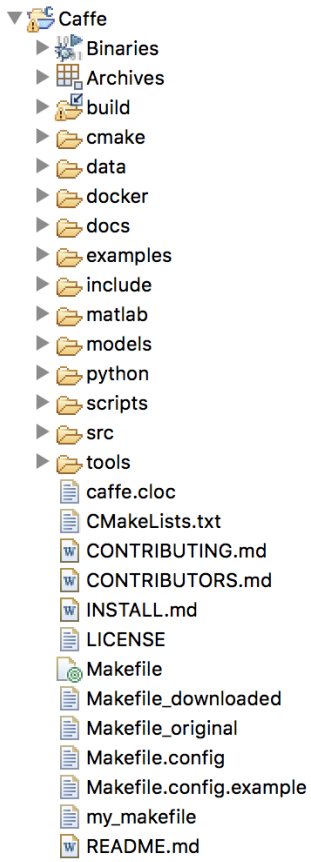

Figure C.1 Code structure of the Caffe framework.

C.2 THE LENET CNN MODEL FOR THE MNIST DATASET WITH CAFFE

Caffe has many examples available. However, it's better to start with the MNIST dataset, not because you are already familiar with the MNIST dataset, but because this example contains detailed descriptions about how to define a CNN model to work with Caffe. Therefore, let's get started with this example first. Once again, make sure you have the CAFFE_ROOT environment variable set in your environment per instructions given in the previous section.

C.2.1 PREPARE THE MNIST DATASET

First, if you don't have `wget` installed on your machine, execute the following command to get it installed:

```
$brew install wget --with-libressl
```

Then, execute the following commands:

```
$cd $CAFFE_ROOT
$./data/mnist/get_mnist.sh
$./examples/mnist/create_mnist.sh
```

After executing the above commands, you should have four files with their names ending with `-ubyte` in the *data/mnist* directory. These are the training and testing dataset we will use.

C.2.2 DEFINING THE LENET MODEL

Next, the instruction explains about the LeNet model to be used with the MNIST dataset we have just prepared. I assume that you have studied Chapter 10 of the main text, so I would not repeat about the LeNet here. However, there is a deviation here: The Caffe model here uses the ReLU activation function instead of the sigmoid function as was the case with the original LeNet model, since it has become common knowledge that the ReLU activation function works better than the sigmoid activation function.

Now, let's explain how Caffe defines a CNN model. With Caffe, each model is defined in a text file, e.g., the file `$CAFFE_ROOT/examples/mnist/lenet_train_test.prototxt` in this case for the LeNet model. You can now open this file and examine its contents. It starts with a line of `name: "LeNet"`, followed by 11 segments labeled "`layer`." To understand this model definition file, perhaps this is a good time for me to help you understand several Caffe jargons as follows:

- **Blobs**. Caffe stores and communicates data in 4D arrays called Blobs.
- **Models**. Caffe models are saved to disk using Google Protocol Buffers.
- **Data**. Caffe stores large scale data in LevelDB databases.
- **Layer**. Defines one or more blobs as input or output to be used in forward and backward passes.
- **Layer Types**. Include: data, convolution, pooling, inner products (ip's), nonlinearities (ReLU, logistic, etc.), local response normalization, element-wise operations, losses (softmax, hinge, etc.), and so on.

Given what we have covered in the main text, you should have no difficulties in understanding the above concepts.

Defining a data layer

The data layers define the *data* and *label* blobs for the training and testing datasets, as shown in Listing C.1. Here, every item is obvious except that (1) the `transform_param` segment defines how data should be transformed, e.g., scaled or normalized by being multiplied with a number 0.00390625, which is just the reciprocal of 256, and (2) the `data_param` segment defines the data source. In addition, this one file defines data blobs with the `include` attribute for both the training phase and the testing phase, and Caffe knows which data blob to choose, based on the phase it is in. These are called layer rules, which are defined in a large file in `$CAFFE_ROOT/src/caffe/proto/caffe.proto`. You can take a quick look at this file to get an idea on how Caffe rules are defined.

Next, we discuss the convolution layer.

Listing C.1. MNIST training and testing data layers with Caffe

```
layer {
  name: "mnist"
  type: "Data"
  top: "data"
  top: "label"
  include {
    phase: TRAIN
  }
  transform_param {
    scale: 0.00390625
  }
  data_param {
    source: "examples/mnist/mnist_train_lmdb"
    batch_size: 64
    backend: LMDB
  }
}
layer {
  name: "mnist"
  type: "Data"
  top: "data"
  top: "label"
  include {
    phase: TEST
  }
  transform_param {
    scale: 0.00390625
  }
  data_param {
    source: "examples/mnist/mnist_test_lmdb"
    batch_size: 100
    backend: LMDB
  }
}
```

Defining a convolution layer with Caffe

Listing C.2 shows how a convolution layer is defined. It can be understood as follows:

- The `bottom` attribute defines the prior layer while the `top` attribute defines the current layer.
- The `param` attributes define the learning rate multipliers for the weights and biases, respectively. In this case, the weights multiplier is 1 and the biases multiplier is 2, which are applied to the learning rate determined by the solver during runtime.
- The `convolution_param` attribute defines the settings for carrying out the convolution. In this case, it defines to produce 20 output channels with a kernel size of 5 and a stride of 1.
- The `weight_filler` attribute specifies how weights should be randomly initialized. In this case, it specifies to use the Xavier algorithm to automatically determine the scale of initialization based on the number of input and output neurons.
- The `bias_filler` attribute specifies how biases should be initialized. In this case, it specifies that biases should be initialized as constant, with the default filling value of 0.

Next, we discuss how a pooling layer is defined with Caffe.

Listing C.2 A convolution layer defined with Caffe

```
layer {
  name: "conv1"
  type: "Convolution"
  bottom: "data"
  top: "conv1"
  param {
    lr_mult: 1
  }
  param {
    lr_mult: 2
  }
  convolution_param {
    num_output: 20
    kernel_size: 5
    stride: 1
    weight_filler {
      type: "xavier"
    }
    bias_filler {
      type: "constant"
    }
  }
}
```

Defining a pooling layer with Caffe

Listing C.3 shows how a pooling layer can be defined with Caffe. In this case, it specifies which convolution layer to follow as defined by the `bottom` attribute, and the pooling settings such as using the

max pooling with a kernel size of 2 and a stride of 2. In this case, there are no overlaps between neighboring pooling regions.

Next, we discuss how a fully connected layer is defined with Caffe.

Listing C.3 A pooling layer defined with Caffe

```
layer {
  name: "pool1"
  type: "Pooling"
  bottom: "conv1"
  top: "pool1"
  pooling_param {
    pool: MAX
    kernel_size: 2
    stride: 2
  }
}
```

Defining a fully connected layer with Caffe

Listing C.4 shows how a fully connected layer, which designated as type `InnerProduct`, can be defined with Caffe. In this case, it specifies which layer to follow as defined by the `bottom` attribute, and uses an `inner_product_param` attribute to specify the number of outputs as well as the weight and bias fillers.

Next, we discuss how an ReLU layer is defined with Caffe.

Listing C.4 A fully connected layer defined with Caffe

```
layer {
  name: "ip1"
  type: "InnerProduct"
  bottom: "pool2"
  top: "ip1"
  param {
    lr_mult: 1
  }
  param {
    lr_mult: 2
  }
  inner_product_param {
    num_output: 500
    weight_filler {
      type: "xavier"
    }
    bias_filler {
      type: "constant"
    }
```

```
  }
}
```

Defining an ReLU layer with Caffe

Listing C.5 shows how an ReLU layer can be defined with Caffe. In this case, both the `bottom` attribute and the `top` attribute are specified to be the same fully connected layer, which makes sense as an ReLU is not necessarily a layer by itself at all – it just performs an element-wise operation, which can be done *in-place* to save memory.

However, note how Listing C.6 defines another fully connected layer, following the ReLU layer described in Listing C.5. In particular, the `ip1` layer, not the `relu1` layer, is assigned to the bottom attribute, as an ReLU layer is more of an element-wise operation than an actual layer.

Next, we discuss how an accuracy layer is defined with Caffe.

Listing C.5 An ReLU layer defined with Caffe

```
layer {
  name: "relu1"
  type: "ReLU"
  bottom: "ip1"
  top: "ip1"
}
```

Listing C.6 A fully connected layer following an ReLU layer defined with Caffe

```
layer {
  name: "ip2"
  type: "InnerProduct"
  bottom: "ip1"
  top: "ip2"
  param {
    lr_mult: 1
  }
  param {
    lr_mult: 2
  }
  inner_product_param {
    num_output: 10
    weight_filler {
      type: "xavier"
    }
    bias_filler {
      type: "constant"
    }
  }
}
```

Defining an accuracy layer with Caffe

Listing C.7 shows how an accuracy layer can be defined with Caffe. In this case, two `bottom` attributes are specified as inputs, the `ip2` layer and the `label` "layer." It is also specified that this layer should be used in the `TEST` phase.

Next, we discuss how a loss layer is defined with Caffe.

Listing C.7 An accuracy layer defined with Caffe

```
layer {
  name: "accuracy"
  type: "Accuracy"
  bottom: "ip2"
  bottom: "label"
  top: "accuracy"
  include {
    phase: TEST
  }
}
```

Defining a loss layer with Caffe

Listing C.8 shows how a loss layer can be defined with Caffe, which should be the final layer of a CNN model with Caffe. In this case, two `bottom` attributes are specified as inputs, the `ip2` layer and the `label` "layer." The `ip2` layer provides predictions while the `label` layer provides target values, both of which are used for computing the loss, which is the basis for the back-propagation algorithm to work..

Next, we discuss how the solver is defined with Caffe for the LetNet model with the MNIST dataset.

Listing C.8 A loss layer defined with Caffe

```
layer {
  name: "loss"
  type: "SoftmaxWithLoss"
  bottom: "ip2"
  bottom: "label"
  top: "loss"
}
```

C.3 DEFINING THE SOLVER FOR THE MNIST DATASET WITH CAFFE

The file `$CAFFE_ROOT/examples/mnist/lenet_solver.prototxt` defines the solver, which specifies the end-to-end process for running the entire job. Listing C.9 shows the entire contents of this file. Since we have basic concepts covered in the main text and every line is clearly annotated, we would not spend time to explain every line, except that the `solver_mode` specified at the end of the file should be changed to `CPU` if you do not have a GPU equipped with your machine.

Listing C.9 lenet_solver.prototxt

```
# The train/test net protocol buffer definition
net: "examples/mnist/lenet_train_test.prototxt"
# test_iter specifies how many forward passes the test should carry out.
# In the case of MNIST, we have test batch size 100 and 100 test iterations,
# covering the full 10,000 testing images.
test_iter: 100
# Carry out testing every 500 training iterations.
test_interval: 500
# The base learning rate, momentum and the weight decay of the network.
base_lr: 0.01
momentum: 0.9
weight_decay: 0.0005
# The learning rate policy
lr_policy: "inv"
gamma: 0.0001
power: 0.75
# Display every 100 iterations
display: 100
# The maximum number of iterations
max_iter: 10000
# snapshot intermediate results
snapshot: 5000
snapshot_prefix: "examples/mnist/lenet"
# solver mode: CPU or GPU
solver_mode: CPU
```

C.4 KICKING OFF TRAINING AND TESTING WITH CAFFE

The `examples/mnist/lenet_train_test.prototxt` and `examples/mnist/lenet_solver.prototxt` files are called Caffe *protobuf* files. Once they are prepared, just run the following two commands to kick off training and testing:

```
cd $CAFFE_ROOT
./examples/mnist/train_lenet.sh
```

The command specified in the script `train_lenet.sh` is as follows:

```
./build/tools/caffe train --solver=examples/mnist/lenet_solver.prototxt
```

As you see, use Caffe to solve an applicable machine learning problem consists of the following three steps:

1. Compose a network model definition file similar to the `lenet_train_test.prototxt` file.
2. Compose a job process definition file similar to the `lenet_solver.prototxt` file.
3. Compose a script similar to the script `train_lenet.sh` and run it.

Listing C.10 shows running the above MNIST LeNet model with Caffe on my machine. Note that I just picked a few segments for illustrative purposes. As you can see, the test started at 21:57:27 and ended at 22:03:19 for a total duration of 4m36s, with an accuracy of 99.909% achieved after 10000 iterations! This is outstanding performance by any means.

Listing C.10 Sample output of running the MNIST LeNet model with Caffe

```
henryliu:Caffe henryliu$ ./examples/mnist/train_lenet.sh
I0310 21:57:27.226054 2508161984 caffe.cpp:197] Use CPU.
I0310 21:57:27.227905 2508161984 solver.cpp:45] Initializing solver from parameters:
……
0310 21:57:27.230612 2508161984 layer_factory.hpp:77] Creating layer mnist
I0310 21:57:27.231889 2508161984 db_lmdb.cpp:35] Opened lmdb examples/mnist/mnist_train_lmdb
I0310 21:57:27.232677 2508161984 net.cpp:84] Creating Layer mnist
I0310 21:57:27.232699 2508161984 net.cpp:380] mnist -> data
I0310 21:57:27.232717 2508161984 net.cpp:380] mnist -> label
I0310 21:57:27.232748 2508161984 data_layer.cpp:45] output data size: 64,1,28,28
I0310 21:57:27.237839 2508161984 net.cpp:122] Setting up mnist
I0310 21:57:27.237856 2508161984 net.cpp:129] Top shape: 64 1 28 28 (50176)
I0310 21:57:27.237865 2508161984 net.cpp:129] Top shape: 64 (64)
……
I0310 21:58:13.941082 2508161984 solver.cpp:239] Iteration 1300 (33.0033 iter/s, 3.03s/100 iters), loss =
0.0233421
I0310 21:58:13.941115 2508161984 solver.cpp:258]     Train net output #0: loss = 0.0233422 (* 1 = 0.0233422 loss)
I0310 21:58:13.941123 2508161984 sgd_solver.cpp:112] Iteration 1300, lr = 0.00912412
I0310 21:58:16.960737 2508161984 solver.cpp:239] Iteration 1400 (33.1236 iter/s, 3.019s/100 iters), loss =
0.00798987
I0310 21:58:16.960772 2508161984 solver.cpp:258]     Train net output #0: loss = 0.00798988 (* 1 = 0.00798988
loss)
I0310 21:58:16.960778 2508161984 sgd_solver.cpp:112] Iteration 1400, lr = 0.00906403
I0310 21:58:19.946302 2508161984 solver.cpp:351] Iteration 1500, Testing net (#0)
……
I0310 22:03:13.821404 2508161984 sgd_solver.cpp:112] Iteration 9900, lr = 0.00596843
I0310 22:03:16.879815 2508161984 solver.cpp:468] Snapshotting to binary proto file
examples/mnist/lenet_iter_10000.caffemodel
I0310 22:03:16.900782 2508161984 sgd_solver.cpp:280] Snapshotting solver state to binary proto file
examples/mnist/lenet_iter_10000.solverstate
I0310 22:03:16.918725 2508161984 solver.cpp:331] Iteration 10000, loss = 0.00294297
I0310 22:03:16.918767 2508161984 solver.cpp:351] Iteration 10000, Testing net (#0)
I0310 22:03:19.175561 131223552 data_layer.cpp:73] Restarting data prefetching from start.
I0310 22:03:19.274293 2508161984 solver.cpp:418]     Test net output #0: accuracy = 0.9909
I0310 22:03:19.274327 2508161984 solver.cpp:418]     Test net output #1: loss = 0.0286514 (* 1 = 0.0286514 loss)
I0310 22:03:19.274333 2508161984 solver.cpp:336] Optimization Done.
I0310 22:03:19.274336 2508161984 caffe.cpp:250] Optimization Done.
```

C.3 ALEX'S CIFAR-10 WITH CAFFE

If you have successfully completed the previous exercise, then you have learnt how Caffe works! You can verify your learning with this second Caffe deep learning CNN example.

First, let's learn a bit about the CIFAR-10 dataset. This is a dataset created by Alex Krizhevsky at the Canadian Institute for Advanced Research (CIFAR), with 10 classes of images of 32x32 pixels. It has 6000 images per class, for a total of 60,000 images. Out of 60,000 images, 50,000 are used as training images and 10,000 are used as test images. The 50,000 training images are split into 5 batches, with each

batch containing 10,000 images. Figure C.2 shows 10 sample images for each of the 10 classes. You can find more about this dataset at https://www.cs.toronto.edu/~kriz/cifar.html.

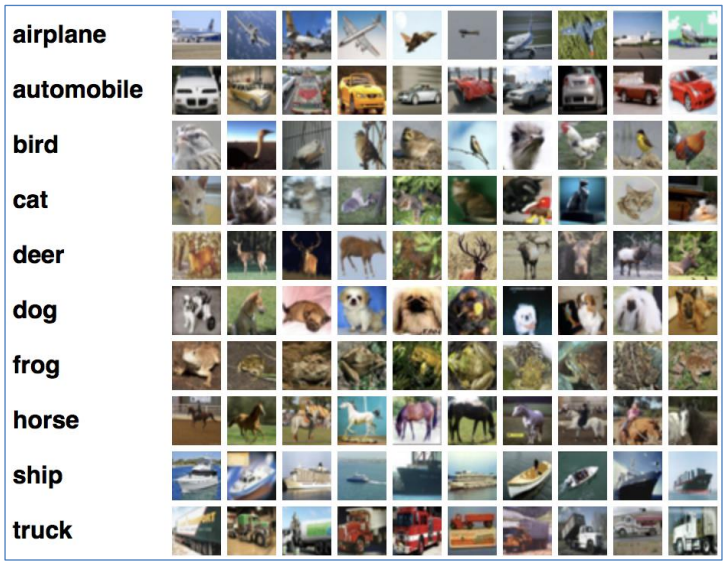

Figure C.2 Samples for Alex's CIFAR-10 dataset.

Now, in terms of trying out this dataset with Caffe, I'd like to take a different approach. In the previous section with the MNIST dataset, we first examined the model description file, then the job description file, and finally the script for kicking off the training process. For this example, I'd like to reverse the process, namely, we first look at the script for kicking off the training process, then the job description file, and finally the model definition file. I feel this may help you understand how Caffe framework works better.

C.3.1 THE SCRIPT FOR KICKING OFF THE TRAINING PROCESS

Listing C.11 shows the script `$CAFFE_ROOT/examples/cifar10/train_quick.sh`. It requires to have two job description files to feed to the solver: `cifar10_quick_solver.prototxt` and `cifar10_quick_solver_lr1.prototxt`, which will be discussed in the next section. The trained model is saved to a snapshot file named `cifar10_quick_iter_4000.solverstate`.

Listing C.11CIFAR-10 train-quick.sh script

```
#!/usr/bin/env sh
set -e

TOOLS=./build/tools

$TOOLS/caffe train \
```

```
  --solver=examples/cifar10/cifar10_quick_solver.prototxt $@

# reduce learning rate by factor of 10 after 8 epochs
$TOOLS/caffe train \
  --solver=examples/cifar10/cifar10_quick_solver_lr1.prototxt \
  --snapshot=examples/cifar10/cifar10_quick_iter_4000.solverstate $@
```

C.3.2 THE JOB DESCRIPTION FILES

Listings C.12 and C.13 show the two job description files: `cifar10_quick_solver.prototxt` and `cifar10_quick_solver_lr1.prototxt`, respectively. This can be considered a two-phase training, with the learning rate reduced to 10x smaller in the second training phase. Note also that by default, the `solver_mode` was set to GPU, but I have changed it to CPU for the same reason explained in the previous section. You should do the same if you do not have a GPU installed on your machine.

Next, we check out the model definition file for this example.

Listing C.12 The cifar10_quick_solver.prototxt file

```
# reduce the learning rate after 8 epochs (4000 iters) by a factor of 10

# The train/test net protocol buffer definition
net: "examples/cifar10/cifar10_quick_train_test.prototxt"
# test_iter specifies how many forward passes the test should carry out.
# In the case of MNIST, we have test batch size 100 and 100 test iterations,
# covering the full 10,000 testing images.
test_iter: 100
# Carry out testing every 500 training iterations.
test_interval: 500
# The base learning rate, momentum and the weight decay of the network.
base_lr: 0.001
momentum: 0.9
weight_decay: 0.004
# The learning rate policy
lr_policy: "fixed"
# Display every 100 iterations
display: 100
# The maximum number of iterations
max_iter: 4000
# snapshot intermediate results
snapshot: 4000
snapshot_prefix: "examples/cifar10/cifar10_quick"
# solver mode: CPU or GPU
solver_mode: CPU
```

Listing C.13 The cifar10_quick_solver_lr1.prototxt file

```
# reduce the learning rate after 8 epochs (4000 iters) by a factor of 10
```

```
# The train/test net protocol buffer definition
net: "examples/cifar10/cifar10_quick_train_test.prototxt"
# test_iter specifies how many forward passes the test should carry out.
# In the case of MNIST, we have test batch size 100 and 100 test iterations,
# covering the full 10,000 testing images.
test_iter: 100
# Carry out testing every 500 training iterations.
test_interval: 500
# The base learning rate, momentum and the weight decay of the network.
base_lr: 0.0001
momentum: 0.9
weight_decay: 0.004
# The learning rate policy
lr_policy: "fixed"
# Display every 100 iterations
display: 100
# The maximum number of iterations
max_iter: 5000
# snapshot intermediate results
snapshot: 5000
snapshot_format: HDF5
snapshot_prefix: "examples/cifar10/cifar10_quick"
# solver mode: CPU or GPU
solver_mode: CPU
```

C.3.3 THE MODEL DEFINITION FILE

Listing C.14 shows the model definition file for this example. It's kind of lengthy, but not very different from the model file we discussed in the previous section for the MNIST dataset, except that it has more layers. Please take your time and go through it end to end to make sure that you understand it, or even better, make a sketch drawing by going through all layers from bottom to top.

Listing C.14 The model definition file for the Caffe CIFAR-10 example

```
name: "CIFAR10_quick"
layer {
  name: "cifar"
  type: "Data"
  top: "data"
  top: "label"
  include {
    phase: TRAIN
  }
  transform_param {
    mean_file:
"examples/cifar10/mean.binaryproto"
  }
  data_param {
    source:
"examples/cifar10/cifar10_train_lmdb"
    batch_size: 100
    backend: LMDB
  }
}
layer {
  name: "cifar"
  type: "Data"
  top: "data"
```

```
  top: "label"
  include {
    phase: TEST
  }
  transform_param {
    mean_file:
"examples/cifar10/mean.binaryproto"
  }
  data_param {
    source:
"examples/cifar10/cifar10_test_lmdb"
    batch_size: 100
    backend: LMDB
  }
}
layer {
  name: "conv1"
  type: "Convolution"
  bottom: "data"
  top: "conv1"
  param {
    lr_mult: 1
  }
  param {
    lr_mult: 2
  }
  convolution_param {
    num_output: 32
    pad: 2
    kernel_size: 5
    stride: 1
    weight_filler {
      type: "gaussian"
      std: 0.0001
    }
    bias_filler {
      type: "constant"
    }
  }
}
layer {
  name: "pool1"
  type: "Pooling"
  bottom: "conv1"
  top: "pool1"
  pooling_param {
    pool: MAX
    kernel_size: 3
    stride: 2
  }
}
layer {
  name: "relu1"
  type: "ReLU"
  bottom: "pool1"
  top: "pool1"
}
layer {
  name: "conv2"
  type: "Convolution"
  bottom: "pool1"
  top: "conv2"
  param {
    lr_mult: 1
  }
  param {
    lr_mult: 2
  }
  convolution_param {
    num_output: 32
    pad: 2
    kernel_size: 5
    stride: 1
    weight_filler {
      type: "gaussian"
      std: 0.01
    }
    bias_filler {
      type: "constant"
    }
  }
}
layer {
  name: "relu2"
  type: "ReLU"
  bottom: "conv2"
  top: "conv2"
}
layer {
  name: "pool2"
  type: "Pooling"
  bottom: "conv2"
  top: "pool2"
  pooling_param {
```

```
    pool: AVE
    kernel_size: 3
    stride: 2
  }
}
layer {
  name: "conv3"
  type: "Convolution"
  bottom: "pool2"
  top: "conv3"
  param {
    lr_mult: 1
  }
  param {
    lr_mult: 2
  }
  convolution_param {
    num_output: 64
    pad: 2
    kernel_size: 5
    stride: 1
    weight_filler {
      type: "gaussian"
      std: 0.01
    }
    bias_filler {
      type: "constant"
    }
  }
}
layer {
  name: "relu3"
  type: "ReLU"
  bottom: "conv3"
  top: "conv3"
}
layer {
  name: "pool3"
  type: "Pooling"
  bottom: "conv3"
  top: "pool3"
  pooling_param {
    pool: AVE
    kernel_size: 3
    stride: 2
  }
}
layer {
  name: "ip1"
  type: "InnerProduct"
  bottom: "pool3"
  top: "ip1"
  param {
    lr_mult: 1
  }
  param {
    lr_mult: 2
  }
  inner_product_param {
    num_output: 64
    weight_filler {
      type: "gaussian"
      std: 0.1
    }
    bias_filler {
      type: "constant"
    }
  }
}
layer {
  name: "ip2"
  type: "InnerProduct"
  bottom: "ip1"
  top: "ip2"
  param {
    lr_mult: 1
  }
  param {
    lr_mult: 2
  }
  inner_product_param {
    num_output: 10
    weight_filler {
      type: "gaussian"
      std: 0.1
    }
    bias_filler {
      type: "constant"
    }
  }
}
layer {
  name: "accuracy"
  type: "Accuracy"
```

```
  bottom: "ip2"
  bottom: "label"
  top: "accuracy"
  include {
    phase: TEST
  }
}
layer {
  name: "loss"
  type: "SoftmaxWithLoss"
  bottom: "ip2"
  bottom: "label"
  top: "loss"
}
```

C.3.4 RUNNING THE CIFAR-10 EXAMPLE

I ran this example successfully on my machine, except that it took close to an hour to download the CIFAR-10 dataset of ~170MB, due to my slow wifi connection. Listing C.15 shows the final accuracy of 75.68%.

If you want to try it out, make necessary changes such as the `solver_mode`, and then run the following commands on your machine to get it going:

```
$cd $CAFFE_ROOT
$./examples/cifar10/train_quick.sh
```

If you encounter any issues, check out http://caffe.berkeleyvision.org/gathered/examples/cifar10.html for more detailed instructions.

Listing C.15 Sample output of running the CIFAR-10 example

```
.......
I0310 14:35:48.097317 2508161984 sgd_solver.cpp:112] Iteration 4800, lr = 0.0001
I0310 14:36:09.057718 2508161984 solver.cpp:239] Iteration 4900 (4.77099 iter/s, 20.96s/100 iters), loss =
0.465986
I0310 14:36:09.057766 2508161984 solver.cpp:258]     Train net output #0: loss = 0.465986 (* 1 = 0.465986 loss)
I0310 14:36:09.057773 2508161984 sgd_solver.cpp:112] Iteration 4900, lr = 0.0001
I0310 14:36:29.173247 73412608 data_layer.cpp:73] Restarting data prefetching from start.
I0310 14:36:30.006633 2508161984 solver.cpp:478] Snapshotting to HDF5 file
examples/cifar10/cifar10_quick_iter_5000.caffemodel.h5
I0310 14:36:30.015507 2508161984 sgd_solver.cpp:290] Snapshotting solver state to HDF5 file
examples/cifar10/cifar10_quick_iter_5000.solverstate.h5
I0310 14:36:30.114841 2508161984 solver.cpp:331] Iteration 5000, loss = 0.525545
I0310 14:36:30.114869 2508161984 solver.cpp:351] Iteration 5000, Testing net (#0)
I0310 14:36:39.559231 73949184 data_layer.cpp:73] Restarting data prefetching from start.
I0310 14:36:39.941176 2508161984 solver.cpp:418]     Test net output #0: accuracy = 0.7568
I0310 14:36:39.941207 2508161984 solver.cpp:418]     Test net output #1: loss = 0.735389 (* 1 = 0.735389 loss)
I0310 14:36:39.941213 2508161984 solver.cpp:336] Optimization Done.
I0310 14:36:39.941217 2508161984 caffe.cpp:250] Optimization Done.
```

C.4 THE IMAGENET EXAMPLE WITH CAFFE

Given the two examples we covered in the previous sections, you should be able to follow the instructions at http://caffe.berkeleyvision.org/gathered/examples/imagenet.html to try out the ImageNet example with Caffe. If you decide to try it out, download the ImageNet data from the website at http://www.image-net.org/challenges/LSVRC/2012/nonpub-downloads. The entire data amounts to ~160

GB, which could be challenging to download if you do not have a fast Internet connection. In my case, I downloaded the following three files at home with a cable connected to a Windows PC:

- *ILSVRC2012_img_train.tar* of 32.96GB with 258,434 images (~22%) instead of the full set of ~1.2M images of ~138GB.
- *ILSVRC2012_img_val.tar* of 6.74GB with all 50,000 images.
- *ILSVRC2012_img_test.tar* of 13.69GB with all 100,000 images.

Then I double-clicked on the file *ILSVRC2012_img_train.tar*, renamed the directory to *train*, created the following shell script, and executed it to untar all JPEG files from each tar file.

```
#!/bin/bash
for name in ./*.tar; do
    tar_name=$(basename "$name")
    dir_name="${tar_name%.*}"
    #echo $dir_name
    mkdir -p $dir_name
    tar -xvf $name -C $dir_name
done
```

Then I followed the instructions given in the *readme.md* file located in the directory of *examples/imagenet* as follows:

1. **Data Preparation**. I executed the script *./data/ilsvrc12/get_ilsvrc_aux.sh* and downloaded the required auxiliary data from http://dl.caffe.berkeleyvision.org/caffe_ilsvrc12.tar.gz, which is not ImageNet data. After this step, the files placed in the *data/ilsvrc12* directory include: *det_synset_words.txt*, *imagenet_mean.binaryproto*, *imagenet.bet.pickle*, *synset_words.txt*, *synsets.txt*, *test.txt*, *train.txt*, and *val.txt*. The *imagenet_mean.binaryproto* and *imagenet.bet.pickle* are binary files, while all others ending with `.txt` are text files. The text files describe what each of the images is, either with a number from 0 to 999 or an actual name. This kind of information had already been prepared for us, so we just use it as is.
2. **Resize Image**. Now open the *examples/imagenet/create_imagenet.sh* file, and make two changes: (1) set `RESIZE` to true if you have not resized the images, and (2) set the path for `TRAIN_DATA_ROOT` and `VAL_DATA_ROOT` so that Caffe would know where the ImageNet data resides. After executing this step, training and validation datasets would be inserted into the LevelDB database.
3. **Compute Image Mean**. Caffe requires that all image data be centered around the mean, so this step accomplishes that. Execute the command *./examples/imagenet/make_imagenet_mean.sh* and a file named *data/ilsvrc12/imagenet_mean.binaryproto* will be created.
4. **Model Definition**. This example attempts to mimic the work by Krizhevsky et al. as we introduced in Chapter 10. The file *models/bvlc_reference_caffenet/train_val.prototxt* describes the model, as shown in Listing C.16. Although it's quite lengthy, all layers should be familiar to you, so we would not repeat explaining them.
5. **Job Definition**. The file *models/bvlc_reference_caffenet/solver.prototxt* specifies how the training job should be carried out. Once again, remember to change `solver_mode` to `CPU` if you do not have a GPU installed on your machine.
6. **Kick off the training job**. When you are ready, simply kick off the training job by executing the command *./build/tools/caffe train --solver=models/bvlc_reference_caffenet/solver.prototxt*.

However, for your reference, without a GPU, it would be slow. For example, on my MacBook Pro with an Intel i7 quad-core processor, it took ~4 minutes per 20 iterations, which is roughly 10x slower than on a K40 GPU. Listing C.18 shows a partial output of running this example on my machine. It is seen that at the end of the 50,000 iterations, training loss and test loss reached 1.4091 and 8.42039, respectively, while the test accuracy reached 0.10892 only, after running for 8684 minutes or about 6 days. This means that we do need GPUs for training deep learning models.

If you decide to develop your skills in applying CNN models to computer vision, delve into the internal implementations of Caffe or Caffe2. Your investment in your time will be paid off nicely.

Listing C.16 ImageNet AlexNet model definition file (train_val.prototxt)

```
name: "CaffeNet"
layer {
  name: "data"
  type: "Data"
  top: "data"
  top: "label"
  include {
    phase: TRAIN
  }
  transform_param {
    mirror: true
    crop_size: 227
    mean_file:
"data/ilsvrc12/imagenet_mean.binarypro
to"
  }
#  mean  pixel  /  channel-wise  mean
instead of mean image
#  transform_param {
#    crop_size: 227
#    mean_value: 104
#    mean_value: 117
#    mean_value: 123
#    mirror: true
#  }
  data_param {
    source:
"examples/imagenet/ilsvrc12_train_lmdb
"
    batch_size: 256
    backend: LMDB
  }
}
layer {
  name: "data"
  type: "Data"
  top: "data"
  top: "label"
  include {
    phase: TEST
  }
  transform_param {
    mirror: false
    crop_size: 227
    mean_file:
"data/ilsvrc12/imagenet_mean.binarypro
to"
  }
#  mean  pixel  /  channel-wise  mean
instead of mean image
#  transform_param {
#    crop_size: 227
#    mean_value: 104
#    mean_value: 117
#    mean_value: 123
#    mirror: false
#  }
  data_param {
    source:
"examples/imagenet/ilsvrc12_val_lmdb"
    batch_size: 50
    backend: LMDB
  }
}
layer {
  name: "conv1"
  type: "Convolution"
  bottom: "data"
```

```
  top: "conv1"
  param {
    lr_mult: 1
    decay_mult: 1
  }
  param {
    lr_mult: 2
    decay_mult: 0
  }
  convolution_param {
    num_output: 96
    kernel_size: 11
    stride: 4
    weight_filler {
      type: "gaussian"
      std: 0.01
    }
    bias_filler {
      type: "constant"
      value: 0
    }
  }
}
layer {
  name: "relu1"
  type: "ReLU"
  bottom: "conv1"
  top: "conv1"
}
layer {
  name: "pool1"
  type: "Pooling"
  bottom: "conv1"
  top: "pool1"
  pooling_param {
    pool: MAX
    kernel_size: 3
    stride: 2
  }
}
layer {
  name: "norm1"
  type: "LRN"
  bottom: "pool1"
  top: "norm1"
  lrn_param {
    local_size: 5
    alpha: 0.0001
    beta: 0.75
  }
}
layer {
  name: "conv2"
  type: "Convolution"
  bottom: "norm1"
  top: "conv2"
  param {
    lr_mult: 1
    decay_mult: 1
  }
  param {
    lr_mult: 2
    decay_mult: 0
  }
  convolution_param {
    num_output: 256
    pad: 2
    kernel_size: 5
    group: 2
    weight_filler {
      type: "gaussian"
      std: 0.01
    }
    bias_filler {
      type: "constant"
      value: 1
    }
  }
}
layer {
  name: "relu2"
  type: "ReLU"
  bottom: "conv2"
  top: "conv2"
}
layer {
  name: "pool2"
  type: "Pooling"
  bottom: "conv2"
  top: "pool2"
  pooling_param {
    pool: MAX
    kernel_size: 3
    stride: 2
```

```
  }
}
layer {
  name: "norm2"
  type: "LRN"
  bottom: "pool2"
  top: "norm2"
  lrn_param {
    local_size: 5
    alpha: 0.0001
    beta: 0.75
  }
}
layer {
  name: "conv3"
  type: "Convolution"
  bottom: "norm2"
  top: "conv3"
  param {
    lr_mult: 1
    decay_mult: 1
  }
  param {
    lr_mult: 2
    decay_mult: 0
  }
  convolution_param {
    num_output: 384
    pad: 1
    kernel_size: 3
    weight_filler {
      type: "gaussian"
      std: 0.01
    }
    bias_filler {
      type: "constant"
      value: 0
    }
  }
}
layer {
  name: "relu3"
  type: "ReLU"
  bottom: "conv3"
  top: "conv3"
}
layer {
  name: "conv4"
  type: "Convolution"
  bottom: "conv3"
  top: "conv4"
  param {
    lr_mult: 1
    decay_mult: 1
  }
  param {
    lr_mult: 2
    decay_mult: 0
  }
  convolution_param {
    num_output: 384
    pad: 1
    kernel_size: 3
    group: 2
    weight_filler {
      type: "gaussian"
      std: 0.01
    }
    bias_filler {
      type: "constant"
      value: 1
    }
  }
}
layer {
  name: "relu4"
  type: "ReLU"
  bottom: "conv4"
  top: "conv4"
}
layer {
  name: "conv5"
  type: "Convolution"
  bottom: "conv4"
  top: "conv5"
  param {
    lr_mult: 1
    decay_mult: 1
  }
  param {
    lr_mult: 2
    decay_mult: 0
  }
  convolution_param {
```

```
    num_output: 256
    pad: 1
    kernel_size: 3
    group: 2
    weight_filler {
      type: "gaussian"
      std: 0.01
    }
    bias_filler {
      type: "constant"
      value: 1
    }
  }
}
layer {
  name: "relu5"
  type: "ReLU"
  bottom: "conv5"
  top: "conv5"
}
layer {
  name: "pool5"
  type: "Pooling"
  bottom: "conv5"
  top: "pool5"
  pooling_param {
    pool: MAX
    kernel_size: 3
    stride: 2
  }
}
layer {
  name: "fc6"
  type: "InnerProduct"
  bottom: "pool5"
  top: "fc6"
  param {
    lr_mult: 1
    decay_mult: 1
  }
  param {
    lr_mult: 2
    decay_mult: 0
  }
  inner_product_param {
    num_output: 4096
    weight_filler {
      type: "gaussian"
      std: 0.005
    }
    bias_filler {
      type: "constant"
      value: 1
    }
  }
}
layer {
  name: "relu6"
  type: "ReLU"
  bottom: "fc6"
  top: "fc6"
}
layer {
  name: "drop6"
  type: "Dropout"
  bottom: "fc6"
  top: "fc6"
  dropout_param {
    dropout_ratio: 0.5
  }
}
layer {
  name: "fc7"
  type: "InnerProduct"
  bottom: "fc6"
  top: "fc7"
  param {
    lr_mult: 1
    decay_mult: 1
  }
  param {
    lr_mult: 2
    decay_mult: 0
  }
  inner_product_param {
    num_output: 4096
    weight_filler {
      type: "gaussian"
      std: 0.005
    }
    bias_filler {
      type: "constant"
      value: 1
    }
```

```
  }
}
layer {
  name: "relu7"
  type: "ReLU"
  bottom: "fc7"
  top: "fc7"
}
layer {
  name: "drop7"
  type: "Dropout"
  bottom: "fc7"
  top: "fc7"
  dropout_param {
    dropout_ratio: 0.5
  }
}
layer {
  name: "fc8"
  type: "InnerProduct"
  bottom: "fc7"
  top: "fc8"
  param {
    lr_mult: 1
    decay_mult: 1
  }
  param {
    lr_mult: 2
    decay_mult: 0
  }
  inner_product_param {
    num_output: 1000
    weight_filler {
      type: "gaussian"
      std: 0.01
    }
    bias_filler {
      type: "constant"
      value: 0
    }
  }
}
layer {
  name: "accuracy"
  type: "Accuracy"
  bottom: "fc8"
  bottom: "label"
  top: "accuracy"
  include {
    phase: TEST
  }
}
layer {
  name: "loss"
  type: "SoftmaxWithLoss"
  bottom: "fc8"
  bottom: "label"
  top: "loss"
}
```

Listing C.17 The Caffe AlexNet job definition file solver.prototxt (note that I changed max_iter from 450000 to 50000 for my MacBook Pro with no GPU equipped)

```
net: "models/bvlc_reference_caffenet/train_val.prototxt"
test_iter: 1000
test_interval: 1000
base_lr: 0.01
lr_policy: "step"
gamma: 0.1
stepsize: 100000
display: 20
max_iter: 45000
momentum: 0.9
weight_decay: 0.0005
snapshot: 10000
snapshot_prefix: "models/bvlc_reference_caffenet/caffenet_train"
solver_mode: CPU
```

Listing C.18 Output of running the Caffe AlexNet job

```
I0324 20:06:34.952026 2506531648 layer_factory.hpp:77] Creating layer data
I0324 20:06:34.952397 2506531648 db_lmdb.cpp:35] Opened lmdb examples/imagenet/ilsvrc12_val_lmdb
……
I0324 20:06:35.674441 2506531648 net.cpp:255] Network initialization done.
I0324 20:06:35.674546 2506531648 solver.cpp:57] Solver scaffolding done.
I0324 20:06:35.674772 2506531648 caffe.cpp:239] Starting Optimization
I0324 20:06:35.674787 2506531648 solver.cpp:293] Solving CaffeNet
I0324 20:06:35.674794 2506531648 solver.cpp:294] Learning Rate Policy: step
I0324 20:06:35.785261 2506531648 solver.cpp:351] Iteration 0, Testing net (#0)
I0324 20:23:23.643776 97710080 data_layer.cpp:73] Restarting data prefetching from start.
I0324 20:23:27.673923 2506531648 solver.cpp:418]     Test net output #0: accuracy = 0.001
I0324 20:23:27.673967 2506531648 solver.cpp:418]     Test net output #1: loss = 7.15056 (* 1 = 7.15056 loss)
I0324 20:23:41.583783 2506531648 solver.cpp:239] Iteration 0 (0 iter/s, 1025.91s/20 iters), loss = 7.60255
I0324 20:23:41.583819 2506531648 solver.cpp:258]     Train net output #0: loss = 7.60255 (* 1 = 7.60255 loss)
I0324 20:23:41.583847 2506531648 sgd_solver.cpp:112] Iteration 0, lr = 0.01
I0324 20:27:48.385308 2506531648 solver.cpp:239] Iteration 20 (0.0810369 iter/s, 246.801s/20 iters), loss =
5.78311
I0324 20:27:48.385622 2506531648 solver.cpp:258]     Train net output #0: loss = 5.78311 (* 1 = 5.78311 loss)
I0324 20:27:48.385632 2506531648 sgd_solver.cpp:112] Iteration 20, lr = 0.01
I0324 20:31:46.621083 2506531648 solver.cpp:239] Iteration 40 (0.0839507 iter/s, 238.235s/20 iters), loss =
5.56459
……
I0324 22:53:46.022282 2506531648 solver.cpp:258]     Train net output #0: loss = 4.17744 (* 1 = 4.17744 loss)
I0324 22:53:46.022291 2506531648 sgd_solver.cpp:112] Iteration 740, lr = 0.01
I0324 22:57:44.885599 2506531648 solver.cpp:239] Iteration 760 (0.08373 iter/s, 238.863s/20 iters), loss =
4.13973
I0324 22:57:44.885979 2506531648 solver.cpp:258]     Train net output #0: loss = 4.13973 (* 1 = 4.13973 loss)
I0324 22:57:44.885989 2506531648 sgd_solver.cpp:112] Iteration 760, lr = 0.01
……
I0330 20:31:20.443524 2506531648 solver.cpp:239] Iteration 49960 (0.101471 iter/s, 197.1s/20 iters), loss =
1.25496
I0330 20:31:20.445861 2506531648 solver.cpp:258]     Train net output #0: loss = 1.25496 (* 1 = 1.25496 loss)
I0330 20:31:20.445873 2506531648 sgd_solver.cpp:112] Iteration 49960, lr = 0.01
I0330 20:34:37.205741 2506531648 solver.cpp:239] Iteration 49980 (0.101647 iter/s, 196.759s/20 iters), loss =
1.4091
I0330 20:34:37.206394 2506531648 solver.cpp:258]     Train net output #0: loss = 1.4091 (* 1 = 1.4091 loss)
I0330 20:34:37.206403 2506531648 sgd_solver.cpp:112] Iteration 49980, lr = 0.01
I0330 20:37:44.142716 2506531648 solver.cpp:468] Snapshotting to binary proto file
models/bvlc_reference_caffenet/caffenet_train_iter_50000.caffemodel
I0330 20:37:45.423261 2506531648 sgd_solver.cpp:280] Snapshotting solver state to binary proto file
models/bvlc_reference_caffenet/caffenet_train_iter_50000.solverstate
I0330 20:37:50.101991 2506531648 solver.cpp:331] Iteration 50000, loss = 1.15807
I0330 20:37:50.102022 2506531648 solver.cpp:351] Iteration 50000, Testing net (#0)
I0330 20:51:07.974370 97710080 data_layer.cpp:73] Restarting data prefetching from start.
I0330 20:51:11.204300 2506531648 solver.cpp:418]     Test net output #0: accuracy = 0.10892
I0330 20:51:11.204347 2506531648 solver.cpp:418]     Test net output #1: loss = 8.42039 (* 1 = 8.42039 loss)
```

I0330 20:51:11.204352 2506531648 solver.cpp:336] Optimization Done.
I0330 20:51:11.207332 2506531648 caffe.cpp:250] Optimization Done.

real8684m38.099s
user 28888m26.225s
sys 416m16.676s