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 Examples with Caffe and YOLOv3

This appendix demonstrates a few example CNN implementations with Caffe in C++ and YOLOv3 in C. We choose the Caffe and YOLOv3 deep learning frameworks, as they are two of the most popular frameworks for solving computer vision related machine learning 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/C++. However, you don't have to be a C/C++ expert to try out the example CNN models with Caffe and YOLOv3 to be introduced in this appendix. Some basic knowledge about how C/C++ works in general and how Unix shell scripts work would be sufficient.

I have to mention that YOLOv3 perhaps is the state of the art deep learning framework that you may want to focus on if you look for a production-quality DL framework. You can jump to YOLO directly, which starts with §C.5 *Building the YOLOv3 Framework from the Source*. Otherwise, let's start with Caffe first next.

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.

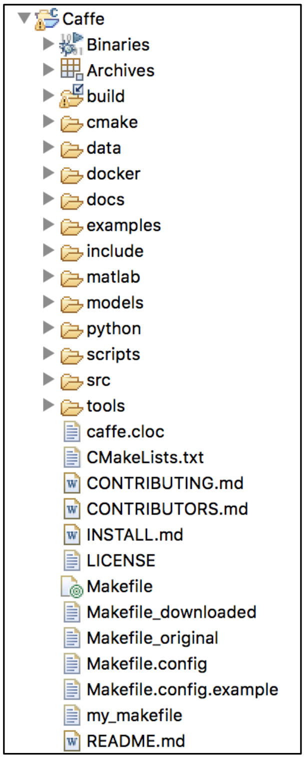


Figure C.1 Code structure of the Caffe framework.

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 10 000 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.

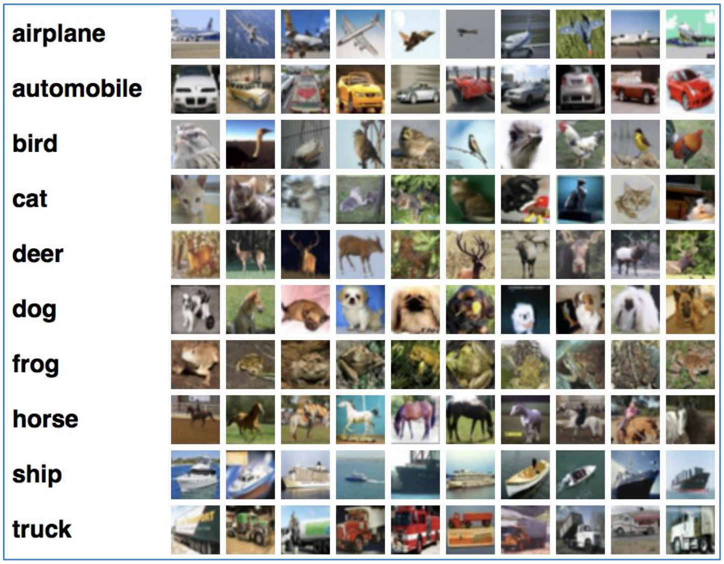


Figure C.2 Samples for Alex's CIFAR-10 dataset.

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.

real	8684m38.099s
user	28888m26.225s
sys	416m16.676s
```

C.5 BUILDING THE YOLOV3 FRAMEWORK FROM THE SOURCE

You may want to watch the YouTube video at https://www.youtube.com/watch?v=Cgxsv1riJhI about how amazing YOLO is. This perhaps can motivate you a bit on getting deep with the YOLOv3 framework. If so, let's begin with how to build YOLOv3 from the source next. You can check out YOLOv3 at https://pjreddie.com/darknet/yolo/ for more information about this framework now or later.

YOLOv3 runs on an engine named *Darknet*. The link at https://pjreddie.com/darknet/install/ gives information on how to install Darknet. In the section of *Compiling with OpenCV*, it mentions that by default, Darknet uses *stb_image.h* for image loading, but *stb_image* does not support all image formats. Besides, OpenCV is a production-quality computer vision library, so I was interested in re-compiling Darknet with OpenCV. However, when I followed the simple instructions given there for re-compiling Darknet with OpenCV on my MacBook Pro, it did not work! It took me some substantial amount of time to rebuild YOLOv3 from the source, which motivated me to summarize my experience here so that you don't have to go through all the difficulties I once had.

The steps to recompile Darknet with OpenCV3 include:

1. Install XCode
2. Install Homebrew
3. Install Python 3
4. Install OpenCV 3 with Python bindings
5. Recompile Darknet with OpenCV3

If you already have 1-3 on your MacBook, you can skip to step 4. Otherwise, install 1-3 first, as instructed below.

C.5.1 INSTALL XCODE

Get the latest version of XCode from the App Store and install it on your macOS machine. Then apply the developer license by executing the below command:

```
$sudo xcodebuild -license
```

Install the Command Line Tools by executing the below command:

```
$ sudo xcode-select --install
```

C.5.2 INSTALL HOMEBREW

If you do not have Homebrew installed on your macOS machine, install it by executing the below command (all in one line):

```
$ ruby -e "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

C.5.3 INSTALL PYTHON3

If you do not have Python 3 installed on your macOS machine, install it with the below command:

```
$ brew install python3
```

To check the python version, execute the following command:

```
$python3 --version
```

I have Python 3.6.5 installed on my machine.

C.5.4 INSTALL OPENCV 3 WITH PYTHON BINDINGS

This is where you may run into difficulties. First of all, if you run the following command as instructed by many online blogs:

```
$brew tap homebrew/science
```

, you may get the following:

```
$Error: homebrew/science was deprecated. This tap is now empty as all its
formulae were migrated.
```

So what do you do? Just ignore it.

Next, if you install opencv3 as follows:

```
$brew install opencv3
```

, you will get the latest OpenCV 3.4.1_2 installed. Then, when you change to the *darknet* directory and re-compile Darknet by typing make, you will get the following error:

```
$In file included from ./src/gemm.c:2:
In file included from src/utils.h:5:
In file included from include/darknet.h:25:
In file included from /usr/local/Cellar/opencv/3.4.1_2/include/opencv2/highgui/highgui_c.h:45:
In file included from /usr/local/Cellar/opencv/3.4.1_2/include/opencv2/core/core_c.h:48:1_2
In file included from /usr/local/Cellar/opencv/3.4.1_2/include/opencv2/core/types_c.h:59:
/usr/local/Cellar/opencv/3.4.1_2/include/opencv2/core/cvdef.h:485:1: fatal error: unknown type name
'namespace'
namespace cv {
^
1 error generated.
make: *** [obj/gemm.o] Error 1
```

So what's wrong here? It only turned out that *opencv3.4.1* does not work with YOLOv3. We have to fall back to opencv3.4.0. I got YOLOv3 compiled successfully with opencv3.4.0 by following the below procedure:

1. Install prerequisites for opencv by executing the following commands:
```
$brew install cmake pkg-config
$brew install jpeg libpng libtiff openexr
$brew install eigen tbb
```
2. Download opencv 3.4.0 and opencv_contrib 3.4.0 to a directory:
   OpenCV3.4.0: https://github.com/opencv/opencv/releases/tag/3.4.0. Click on *Source code (tar.gz)*.
   OpenCV3.4.0 contrib: https://github.com/opencv/opencv_contrib/releases/tag/3.4.0.
3. Change to the directory that contains the above two downloads and execute the following three commands:
```
$mkdir build
$cd build
$cmake -D CMAKE_BUILD_TYPE=RELEASE -D CMAKE_INSTALL_PREFIX=/usr/local -D
OPENCV_EXTRA_MODULES_PATH=/Users/henryliu/mspc/devs/opencv_contrib-
3.4.0/modules -D
PYTHON3_LIBRARY=/usr/local/Cellar/python3/3.6.5/Frameworks/Python.framew
ork/Versions/3.6/lib/python3.6/config-3.6m/libpython3.6.dylib -D
PYTHON3_INCLUDE_DIR=/usr/local/Cellar/python3/3.6.5/Frameworks/Python.fr
amework/Versions/3.6/include/python3.6m/ -D BUILD_opencv_python2=OFF -D
BUILD_opencv_python3=ON -D INSTALL_PYTHON_EXAMPLES=ON -D
INSTALL_C_EXAMPLES=OFF -D BUILD_EXAMPLES=ON
```

Note that with the above *cmake* command, make sure you set `OPENCV_EXTRA_MODULES_PATH`, `PYTHON3_LIBRARY` and `PYTHON3_INCLUDE_DIR` to your own corresponding paths, respectively. At the end, you should see something similar to the following I got on my machine:

```
-- Configuring done
-- Generating done
-- Build files have been written to: /Users/henryliu/mspc/devs/opencv-3.4.0/build
```

Next, execute the following two commands:

```
$ sudo make -j4
$ sudo make install
```

To verify that you have installed opencv3.4.0 successfully, startup python3 and issue the `import cv2` statement as shown below I got on my machine:

```
henryliu:build henryliu$ python3
Python 3.6.5 (default, Mar 30 2018, 06:42:10)
[GCC 4.2.1 Compatible Apple LLVM 9.0.0 (clang-900.0.39.2)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import cv2
>>> cv2.__version__
'3.4.0'
>>>
```

Now download the latest YOLOv3 source code from https://github.com/pjreddie/darknet and save it to a directory on your machine. Then, change to the *darknet* directory, edit the *Makefile* file to enable OPENCV by setting

```
OPENCV=1
```

Now type *make* and it should start re-compiling YOLOv3. After completion, execute the following command:

```
$ ./darknet imtest data/eagle.jpg
```

You should see images as shown below. This is an indication that you have successfully recompiled YOLOv3 on your macOS machine, as these images are supposed to be loaded by OpenCV.

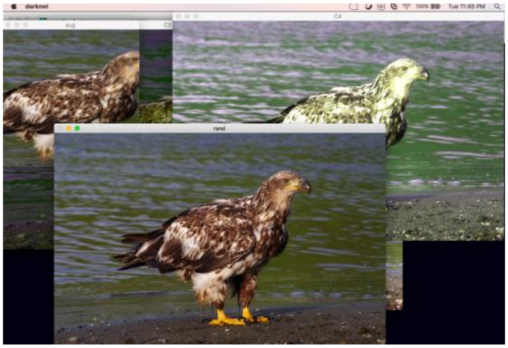


Figure C.3 Testing YOLOv3 recompiled with OpenCV3.4.0.

C.6 ALEX'S CIFAR-10 WITH YOLOv3

If you did not skip §C.3, you should already know Alex's work with CIFAR-10. Now. Let's follow https://pjreddie.com/darknet/train-cifar/ to train a CNN model with YOLOv3 using the CIFAR-10 dataset. The steps include:

1. Get the CIFAR dataset
2. Make a data file to define the job
3. Make a network config file to define the net
4. Train the model

Let's follow these steps to train a classifier.

C.6.1 GET THE CIFAR DATASET

To get the CIFAR dataset, change to the *darknet* directory and run the following commands:

```
$cd data
$wget https://pjreddie.com/media/files/cifar.tgz
$tar xzf cifar.tgz
```

After the above step, you should have the directories of *train* and *test* as well as a file named *labels.txt*. You can check them out by executing the following commands:

```
henryliu:cifar henryliu$ ls train | head -5
0_frog.png
10000_automobile.png
10001_frog.png
10002_frog.png
10003_ship.png
henryliu:cifar henryliu$ ls train | wc -l
   50000
henryliu:cifar henryliu$ ls test | wc -l
   10000
henryliu:cifar henryliu$ cat labels.txt
airplane
automobile
bird
cat
deer
dog
frog
horse
ship
truck
```

The *train* directory contains 50k image files in PNG format, while the *test* directory contains 10k images for testing. The *labels.txt* file contains the 10 classes as shown above that those images belong to.

Next, execute the following commands in the *cifar* directory to create the path files for the training and testing datasets, respectively:

```
$find `pwd`/train -name \*.png > train.list
$find `pwd`/test -name \*.png > test.list
henryliu:data henryliu$ head -5 cifar/train.list
/Users/henryliu/mspc/devs/ws_cpp/darknet/data/cifar/train/0_frog.png
/Users/henryliu/mspc/devs/ws_cpp/darknet/data/cifar/train/10000_automobile.png
/Users/henryliu/mspc/devs/ws_cpp/darknet/data/cifar/train/10001_frog.png
/Users/henryliu/mspc/devs/ws_cpp/darknet/data/cifar/train/10002_frog.png
/Users/henryliu/mspc/devs/ws_cpp/darknet/data/cifar/train/10003_ship.png
henryliu:data henryliu$ head -5 cifar/test.list
/Users/henryliu/mspc/devs/ws_cpp/darknet/data/cifar/test/0_cat.png
/Users/henryliu/mspc/devs/ws_cpp/darknet/data/cifar/test/1000_dog.png
/Users/henryliu/mspc/devs/ws_cpp/darknet/data/cifar/test/1001_airplane.png
/Users/henryliu/mspc/devs/ws_cpp/darknet/data/cifar/test/1002_ship.png
/Users/henryliu/mspc/devs/ws_cpp/darknet/data/cifar/test/1003_deer.png
```

Next, create the job definition file.

C.6.2 MAKE A DATA FILE TO DEFINE THE JOB

Now, check out or create a *cifar.data* file in the *darknet/cfg* directory with the following contents:

```
classes=10
train  = data/cifar/train.list
valid = data/cifar/test.list
labels = data/cifar/labels.txt
backup = backup
top=2
```

Since the *backup* directory does not exist yet, you need to create it yourself now. Then, familiarize yourself with the meaning of each line as follows:

- **classes=10**: the number of unique classes that all images belong to
- **train**: The file that contains the absolute path of each training image file, including the file name
- **valid**: The file that contains the absolute path of each validation/test image file, including the file name
- **labels**: The file containing a list of all possible classes by name
- **backup:** Directory for saving backup weights during training
- **top = 2**: # of top-*n* classes to classify at test time (in addition to top-1)

Next, define the network configuration for training the model.

C.6.3 MAKE A NETWORK CONFIG FILE TO DEFINE THE NET

To define the model for training, create the file *cfg/cifar_small.cfg* with the following contents:

```
[net]
batch=128
subdivisions=1
height=28
width=28
channels=3
max_crop=32
min_crop=32

hue=.1
saturation=.75
exposure=.75

learning_rate=0.1
policy=poly
power=4
max_batches = 5000
momentum=0.9
decay=0.0005

[convolutional]
batch_normalize=1
filters=32
size=3
stride=1
pad=1
activation=leaky

[maxpool]
size=2
stride=2

[convolutional]
batch_normalize=1
filters=64
size=3
stride=1
pad=1
activation=leaky

[maxpool]
size=2
stride=2

[convolutional]
batch_normalize=1
filters=128
size=3
stride=1
pad=1
activation=leaky
```

```
[convolutional]
filters=10
size=1
stride=1
pad=1
activation=leaky

[avgpool]

[softmax]
groups=1

[cost]
type=sse
```

Note that YOLO uses the `max_batches` to define the # of maximum iterations. You can change it to a smaller number, say, 500, just to make sure that it runs. The other parameters should be obvious, given what you have learnt from the main text. The link at https://pjreddie.com/darknet/train-cifar/ has a brief description about the model.

Next, let's see how we can train this model with YOLOv3.

C.6.4 TRAIN THE MODEL

To train the model, just launch it with the following command:

```
$./darknet classifier train cfg/cifar.data cfg/cifar_small.cfg
```

The command to restart the training with a backup file is:

```
./darknet classifier train cfg/cifar.data cfg/cifar_small.cfg backup/cifar_small.backup
```

The instructions at https://pjreddie.com/darknet/train-cifar/ stopped here, so I would take over and share with you what I got on my macOS machine.

I first set `max_batches` to 512 and ended up with the following output, which took about five minutes:

```
henryliu:darknet henryliu$ ./darknet classifier train cfg/cifar.data cfg/cifar_small.cfg
cifar_small
1
layer     filters    size              input                output
   0 conv     32  3 x 3 / 1    28 x  28 x   3   ->    28 x  28 x  32  0.001 BFLOPs
   1 max          2 x 2 / 2    28 x  28 x  32   ->    14 x  14 x  32
   2 conv     64  3 x 3 / 1    14 x  14 x  32   ->    14 x  14 x  64  0.007 BFLOPs
   3 max          2 x 2 / 2    14 x  14 x  64   ->     7 x   7 x  64
   4 conv    128  3 x 3 / 1     7 x   7 x  64   ->     7 x   7 x 128  0.007 BFLOPs
   5 conv     10  1 x 1 / 1     7 x   7 x 128   ->     7 x   7 x  10  0.000 BFLOPs
   6 avg                        7 x   7 x  10   ->    10
   7 softmax                                          10
   8 cost                                             10
Learning Rate: 0.1, Momentum: 0.9, Decay: 0.0005
50000
32 32
Loaded: 0.012910 seconds
1, 0.003: 1.662464, 1.662464 avg, 0.099221 rate, 0.744644 seconds, 128 images
Loaded: 0.000038 seconds
2, 0.005: 1.621058, 1.658324 avg, 0.098447 rate, 0.882484 seconds, 256 images
...
```

```
511, 1.308: 0.997309, 1.019945 avg, 0.000000 rate, 0.683086 seconds, 65408 images
Loaded: 0.000061 seconds
```
512, 1.311: 1.002860, 1.018237 avg, 0.000000 rate, 0.802056 seconds, 65536 images
Saving weights to backup//cifar_small.weights

The last line of **512, 1.311: 1.002860, 1.018237 avg, 0.000000 rate, 0.802056 seconds, 65536 images** represents the iteration, total loss: current loss, average loss so far, current learning rate, time taken for this iteration, and the number of images processed so far. Notice the following:

- The learning rate started with 0.099221 at iteration 1 and ended up with 0.000000 at iteration 512.
- The number of images started with 128 at iteration 1 and ended up with 65536 at iteration 512. This is because each iteration uses 128 images, so 128×512 = 65536. This also means that after 50000/128 = 390 iterations, each image would have been used at least once.

Next I changed `max_batches` from 512 back to 5000 and obtained the following output:

```
henryliu:darknet henryliu$ time ./darknet classifier train cfg/cifar.data cfg/cifar_small.cfg
cifar_small
1
layer     filters    size              input                output
   0 conv     32  3 x 3 / 1    28 x  28 x   3   ->    28 x  28 x  32  0.001 BFLOPs
   1 max          2 x 2 / 2    28 x  28 x  32   ->    14 x  14 x  32
   2 conv     64  3 x 3 / 1    14 x  14 x  32   ->    14 x  14 x  64  0.007 BFLOPs
   3 max          2 x 2 / 2    14 x  14 x  64   ->     7 x   7 x  64
   4 conv    128  3 x 3 / 1     7 x   7 x  64   ->     7 x   7 x 128  0.007 BFLOPs
   5 conv     10  1 x 1 / 1     7 x   7 x 128   ->     7 x   7 x  10  0.000 BFLOPs
   6 avg                        7 x   7 x  10   ->    10
   7 softmax                                          10
   8 cost                                             10
Learning Rate: 0.1, Momentum: 0.9, Decay: 0.0005
50000
32 32
Loaded: 0.006487 seconds
1, 0.003: 1.596092, 1.596092 avg, 0.099920 rate, 0.774180 seconds, 128 images
Loaded: 0.000040 seconds
2, 0.005: 1.606881, 1.597171 avg, 0.099840 rate, 0.692056 seconds, 256 images
Loaded: 0.000043 seconds
3, 0.008: 1.556398, 1.593093 avg, 0.099760 rate, 0.671551 seconds, 384 images
Loaded: 0.000050 seconds
4, 0.010: 1.536735, 1.587458 avg, 0.099680 rate, 0.758029 seconds, 512 images
Loaded: 0.000045 seconds
5, 0.013: 1.480678, 1.576780 avg, 0.099601 rate, 0.657986 seconds, 640 images
Loaded: 0.000045 seconds
6, 0.015: 1.471304, 1.566232 avg, 0.099521 rate, 0.691289 seconds, 768 images
...
2218, 5.678: 0.747149, 0.659472 avg, 0.009584 rate, 0.673178 seconds, 283904 images
Loaded: 0.000044 seconds
2219, 5.681: 0.628408, 0.656366 avg, 0.009570 rate, 0.772427 seconds, 284032 images
Loaded: 0.000067 seconds
```

```
2220, 5.683: 0.681786, 0.658908 avg, 0.009557 rate, 0.694839 seconds, 284160 images
Loaded: 0.000040 seconds
...
4760, 12.186: 0.599866, 0.587485 avg, 0.000001 rate, 0.711729 seconds, 609280 images
Loaded: 0.000038 seconds
4761, 12.188: 0.583313, 0.587067 avg, 0.000001 rate, 0.791747 seconds, 609408 images
Loaded: 0.000034 seconds
4762, 12.191: 0.616005, 0.589961 avg, 0.000001 rate, 0.746034 seconds, 609536 images
Loaded: 0.000043 seconds
4763, 12.193: 0.565400, 0.587505 avg, 0.000001 rate, 0.800969 seconds, 609664 images
Loaded: 0.000033 seconds
4764, 12.196: 0.559835, 0.584738 avg, 0.000000 rate, 0.772222 seconds, 609792 images
...
4998, 12.795: 0.537535, 0.584120 avg, 0.000000 rate, 0.910824 seconds, 639744 images
Loaded: 0.000041 seconds
4999, 12.797: 0.524815, 0.578190 avg, 0.000000 rate, 0.701206 seconds, 639872 images
Loaded: 0.000045 seconds
5000, 12.800: 0.593927, 0.579764 avg, 0.000000 rate, 0.847796 seconds, 640000 images
Saving weights to backup/cifar_small.backup
Saving weights to backup/cifar_small.weights

real	62m44.823s
user	64m3.205s
sys	1m10.443s
```

The above output indicates that it took ~63 minutes for 5000 iterations on my MacBook Pro equipped with an Intel i7 quad-core processor. Once again, the learning rate approached close to zero at iteration 4764.

Now I try to use the obtained weights to detect a dog with the trained weights and got the following output:

```
$ ./darknet classifier predict cfg/cifar.data cfg/cifar_small.cfg backup/cifar_small.weights data/dog.jpg
layer     filters    size              input                output
    0 conv     32  3 x 3 / 1    28 x  28 x   3   ->    28 x  28 x  32  0.001 BFLOPs
    1 max          2 x 2 / 2    28 x  28 x  32   ->    14 x  14 x  32
    2 conv     64  3 x 3 / 1    14 x  14 x  32   ->    14 x  14 x  64  0.007 BFLOPs
    3 max          2 x 2 / 2    14 x  14 x  64   ->     7 x   7 x  64
    4 conv    128  3 x 3 / 1     7 x   7 x  64   ->     7 x   7 x 128  0.007 BFLOPs
    5 conv     10  1 x 1 / 1     7 x   7 x 128   ->     7 x   7 x  10  0.000 BFLOPs
    6 avg                        7 x   7 x  10   ->    10
    7 softmax                                          10
    8 cost                                             10
Loading weights from backup/cifar_small.weights...Done!
data/dog.jpg: Predicted in 0.002599 seconds.
74.07%: ship
21.21%: airplane
```

The above output gave a prediction that the dog picture has a probability 74% for being a *ship* and a probability of 21% for being an *airplane*! However, this should not be treated as YOLO's fault. It's because the model has not been well-trained yet with just 5000 iterations with an oversimplified model.

Next, I tried to test the model trained above using the validation/test dataset, and got the following results:

```
$ ./darknet classifier valid cfg/cifar.data cfg/cifar.test.cfg backup/cifar_small.weights
layer     filters    size              input                output
    0 conv    128  3 x 3 / 1    32 x  32 x   3   ->    32 x  32 x 128  0.007 BFLOPs
    1 conv    128  3 x 3 / 1    32 x  32 x 128   ->    32 x  32 x 128  0.302 BFLOPs
    2 conv    128  3 x 3 / 1    32 x  32 x 128   ->    32 x  32 x 128  0.302 BFLOPs
    3 max          2 x 2 / 2    32 x  32 x 128   ->    16 x  16 x 128
    4 dropout       p = 0.50               32768  ->  32768
    5 conv    256  3 x 3 / 1    16 x  16 x 128   ->    16 x  16 x 256  0.151 BFLOPs
    6 conv    256  3 x 3 / 1    16 x  16 x 256   ->    16 x  16 x 256  0.302 BFLOPs
    7 conv    256  3 x 3 / 1    16 x  16 x 256   ->    16 x  16 x 256  0.302 BFLOPs
    8 max          2 x 2 / 2    16 x  16 x 256   ->     8 x   8 x 256
    9 dropout       p = 0.50               16384  ->  16384
   10 conv    512  3 x 3 / 1     8 x   8 x 256   ->     8 x   8 x 512  0.151 BFLOPs
   11 conv    512  3 x 3 / 1     8 x   8 x 512   ->     8 x   8 x 512  0.302 BFLOPs
   12 conv    512  3 x 3 / 1     8 x   8 x 512   ->     8 x   8 x 512  0.302 BFLOPs
   13 dropout       p = 0.50               32768  ->  32768
   14 conv     10  1 x 1 / 1     8 x   8 x 512   ->     8 x   8 x  10  0.001 BFLOPs
   15 avg                        8 x   8 x  10   ->    10
   16 softmax                                          10
   17 type: Using default 'sse'
cost                                                   10
Loading weights from backup/cifar_small.weights...Done!
/Users/henryliu/mspc/devs/ws_cpp/darknet/data/cifar/test/0_cat.png, 3, nan, nan,
0: top 1: 0.000000,  top 2: 0.000000
/Users/henryliu/mspc/devs/ws_cpp/darknet/data/cifar/test/1000_dog.png, 5, nan, nan,
1: top 1: 0.000000,  top 2: 0.000000
/Users/henryliu/mspc/devs/ws_cpp/darknet/data/cifar/test/1001_airplane.png, 0, nan, nan,
2: top 1: 0.333333,  top 2: 0.333333
/Users/henryliu/mspc/devs/ws_cpp/darknet/data/cifar/test/1002_ship.png, 8, nan, nan,
3: top 1: 0.250000,  top 2: 0.250000
/Users/henryliu/mspc/devs/ws_cpp/darknet/data/cifar/test/1003_deer.png, 4, nan, nan,
4: top 1: 0.200000,  top 2: 0.200000
/Users/henryliu/mspc/devs/ws_cpp/darknet/data/cifar/test/1004_ship.png, 8, nan, nan,
5: top 1: 0.166667,  top 2: 0.166667
......
```

Apparently, the results are not very meaningful, due to the same reason that the model was not well trained.

Next we use YOLO's own pre-trained model to detect bounding boxes.

C.7 USE YOLOV3 TO DETECT BOUNDING BOXES

From YOLO's main web page at https://pjreddie.com/darknet/yolo/, you can find a section about detecting bounding boxes using YOLO's pre-trained model. It starts with getting *darknet*, but you already have it if you followed the previous instructions and re-compiled YOLOv3 with OpenCV3. Then, you can directly go to the next step of retrieving the pre-trained YOLO weights as follows:

```
$ wget https://pjreddie.com/media/files/yolov3.weights
```

Then, execute the following command to detect objects in the picture:

```
$ ./darknet detect cfg/yolov3.cfg yolov3.weights data/dog.jpg
```

The output from the above command should look similar to the following:

```
henryliu:darknet henryliu$ ./darknet detect cfg/yolov3.cfg yolov3.weights data/dog.jpg
layer     filters    size              input                output
    0 conv     32  3 x 3 / 1   416 x 416 x   3   ->   416 x 416 x  32  0.299 BFLOPs
    1 conv     64  3 x 3 / 2   416 x 416 x  32   ->   208 x 208 x  64  1.595 BFLOPs
    2 conv     32  1 x 1 / 1   208 x 208 x  64   ->   208 x 208 x  32  0.177 BFLOPs
    3 conv     64  3 x 3 / 1   208 x 208 x  32   ->   208 x 208 x  64  1.595 BFLOPs
    4 res    1                 208 x 208 x  64   ->   208 x 208 x  64
    5 conv    128  3 x 3 / 2   208 x 208 x  64   ->   104 x 104 x 128  1.595 BFLOPs
    6 conv     64  1 x 1 / 1   104 x 104 x 128   ->   104 x 104 x  64  0.177 BFLOPs
    7 conv    128  3 x 3 / 1   104 x 104 x  64   ->   104 x 104 x 128  1.595 BFLOPs
    8 res    5                 104 x 104 x 128   ->   104 x 104 x 128
    9 conv     64  1 x 1 / 1   104 x 104 x 128   ->   104 x 104 x  64  0.177 BFLOPs
   10 conv    128  3 x 3 / 1   104 x 104 x  64   ->   104 x 104 x 128  1.595 BFLOPs
   11 res    8                 104 x 104 x 128   ->   104 x 104 x 128
   12 conv    256  3 x 3 / 2   104 x 104 x 128   ->    52 x  52 x 256  1.595 BFLOPs
   13 conv    128  1 x 1 / 1    52 x  52 x 256   ->    52 x  52 x 128  0.177 BFLOPs
   14 conv    256  3 x 3 / 1    52 x  52 x 128   ->    52 x  52 x 256  1.595 BFLOPs
   15 res   12                  52 x  52 x 256   ->    52 x  52 x 256
   16 conv    128  1 x 1 / 1    52 x  52 x 256   ->    52 x  52 x 128  0.177 BFLOPs
   17 conv    256  3 x 3 / 1    52 x  52 x 128   ->    52 x  52 x 256  1.595 BFLOPs
   18 res   15                  52 x  52 x 256   ->    52 x  52 x 256
   19 conv    128  1 x 1 / 1    52 x  52 x 256   ->    52 x  52 x 128  0.177 BFLOPs
   20 conv    256  3 x 3 / 1    52 x  52 x 128   ->    52 x  52 x 256  1.595 BFLOPs
   21 res   18                  52 x  52 x 256   ->    52 x  52 x 256
   22 conv    128  1 x 1 / 1    52 x  52 x 256   ->    52 x  52 x 128  0.177 BFLOPs
   23 conv    256  3 x 3 / 1    52 x  52 x 128   ->    52 x  52 x 256  1.595 BFLOPs
   24 res   21                  52 x  52 x 256   ->    52 x  52 x 256
   25 conv    128  1 x 1 / 1    52 x  52 x 256   ->    52 x  52 x 128  0.177 BFLOPs
   26 conv    256  3 x 3 / 1    52 x  52 x 128   ->    52 x  52 x 256  1.595 BFLOPs
   27 res   24                  52 x  52 x 256   ->    52 x  52 x 256
   28 conv    128  1 x 1 / 1    52 x  52 x 256   ->    52 x  52 x 128  0.177 BFLOPs
   29 conv    256  3 x 3 / 1    52 x  52 x 128   ->    52 x  52 x 256  1.595 BFLOPs
   30 res   27                  52 x  52 x 256   ->    52 x  52 x 256
   31 conv    128  1 x 1 / 1    52 x  52 x 256   ->    52 x  52 x 128  0.177 BFLOPs
   32 conv    256  3 x 3 / 1    52 x  52 x 128   ->    52 x  52 x 256  1.595 BFLOPs
   33 res   30                  52 x  52 x 256   ->    52 x  52 x 256
   34 conv    128  1 x 1 / 1    52 x  52 x 256   ->    52 x  52 x 128  0.177 BFLOPs
   35 conv    256  3 x 3 / 1    52 x  52 x 128   ->    52 x  52 x 256  1.595 BFLOPs
```

```
36 res  33              52 x  52 x 256   ->   52 x  52 x 256
37 conv   512  3 x 3 / 2   52 x  52 x 256   ->   26 x  26 x 512  1.595 BFLOPs
38 conv   256  1 x 1 / 1   26 x  26 x 512   ->   26 x  26 x 256  0.177 BFLOPs
39 conv   512  3 x 3 / 1   26 x  26 x 256   ->   26 x  26 x 512  1.595 BFLOPs
40 res  37              26 x  26 x 512   ->   26 x  26 x 512
41 conv   256  1 x 1 / 1   26 x  26 x 512   ->   26 x  26 x 256  0.177 BFLOPs
42 conv   512  3 x 3 / 1   26 x  26 x 256   ->   26 x  26 x 512  1.595 BFLOPs
43 res  40              26 x  26 x 512   ->   26 x  26 x 512
44 conv   256  1 x 1 / 1   26 x  26 x 512   ->   26 x  26 x 256  0.177 BFLOPs
45 conv   512  3 x 3 / 1   26 x  26 x 256   ->   26 x  26 x 512  1.595 BFLOPs
46 res  43              26 x  26 x 512   ->   26 x  26 x 512
47 conv   256  1 x 1 / 1   26 x  26 x 512   ->   26 x  26 x 256  0.177 BFLOPs
48 conv   512  3 x 3 / 1   26 x  26 x 256   ->   26 x  26 x 512  1.595 BFLOPs
49 res  46              26 x  26 x 512   ->   26 x  26 x 512
50 conv   256  1 x 1 / 1   26 x  26 x 512   ->   26 x  26 x 256  0.177 BFLOPs
51 conv   512  3 x 3 / 1   26 x  26 x 256   ->   26 x  26 x 512  1.595 BFLOPs
52 res  49              26 x  26 x 512   ->   26 x  26 x 512
53 conv   256  1 x 1 / 1   26 x  26 x 512   ->   26 x  26 x 256  0.177 BFLOPs
54 conv   512  3 x 3 / 1   26 x  26 x 256   ->   26 x  26 x 512  1.595 BFLOPs
55 res  52              26 x  26 x 512   ->   26 x  26 x 512
56 conv   256  1 x 1 / 1   26 x  26 x 512   ->   26 x  26 x 256  0.177 BFLOPs
57 conv   512  3 x 3 / 1   26 x  26 x 256   ->   26 x  26 x 512  1.595 BFLOPs
58 res  55              26 x  26 x 512   ->   26 x  26 x 512
59 conv   256  1 x 1 / 1   26 x  26 x 512   ->   26 x  26 x 256  0.177 BFLOPs
60 conv   512  3 x 3 / 1   26 x  26 x 256   ->   26 x  26 x 512  1.595 BFLOPs
61 res  58              26 x  26 x 512   ->   26 x  26 x 512
62 conv  1024  3 x 3 / 2   26 x  26 x 512   ->   13 x  13 x1024  1.595  BFLOPs
63 conv   512  1 x 1 / 1   13 x  13 x1024   ->   13 x  13 x 512  0.177  BFLOPs
64 conv  1024  3 x 3 / 1   13 x  13 x 512   ->   13 x  13 x1024  1.595  BFLOPs
65 res  62              13 x  13 x1024   ->   13 x  13 x1024
66 conv   512  1 x 1 / 1   13 x  13 x1024   ->   13 x  13 x 512  0.177  BFLOPs
67 conv  1024  3 x 3 / 1   13 x  13 x 512   ->   13 x  13 x1024  1.595  BFLOPs
68 res  65              13 x  13 x1024   ->   13 x  13 x1024
69 conv   512  1 x 1 / 1   13 x  13 x1024   ->   13 x  13 x 512  0.177  BFLOPs
70 conv  1024  3 x 3 / 1   13 x  13 x 512   ->   13 x  13 x1024  1.595  BFLOPs
71 res  68              13 x  13 x1024   ->   13 x  13 x1024
72 conv   512  1 x 1 / 1   13 x  13 x1024   ->   13 x  13 x 512  0.177  BFLOPs
73 conv  1024  3 x 3 / 1   13 x  13 x 512   ->   13 x  13 x1024  1.595  BFLOPs
74 res  71              13 x  13 x1024   ->   13 x  13 x1024
75 conv   512  1 x 1 / 1   13 x  13 x1024   ->   13 x  13 x 512  0.177  BFLOPs
76 conv  1024  3 x 3 / 1   13 x  13 x 512   ->   13 x  13 x1024  1.595  BFLOPs
77 conv   512  1 x 1 / 1   13 x  13 x1024   ->   13 x  13 x 512  0.177  BFLOPs
78 conv  1024  3 x 3 / 1   13 x  13 x 512   ->   13 x  13 x1024  1.595  BFLOPs
79 conv   512  1 x 1 / 1   13 x  13 x1024   ->   13 x  13 x 512  0.177  BFLOPs
80 conv  1024  3 x 3 / 1   13 x  13 x 512   ->   13 x  13 x1024  1.595  BFLOPs
81 conv   255  1 x 1 / 1   13 x  13 x1024   ->   13 x  13 x 255  0.088  BFLOPs
82 detection
83 route  79
84 conv   256  1 x 1 / 1   13 x  13 x 512   ->   13 x  13 x 256  0.044 BFLOPs
```

```
 85 upsample        2x   13 x  13 x 256   ->   26 x  26 x 256
 86 route  85 61
 87 conv   256  1 x 1 / 1   26 x  26 x 768   ->   26 x  26 x 256  0.266 BFLOPs
 88 conv   512  3 x 3 / 1   26 x  26 x 256   ->   26 x  26 x 512  1.595 BFLOPs
 89 conv   256  1 x 1 / 1   26 x  26 x 512   ->   26 x  26 x 256  0.177 BFLOPs
 90 conv   512  3 x 3 / 1   26 x  26 x 256   ->   26 x  26 x 512  1.595 BFLOPs
 91 conv   256  1 x 1 / 1   26 x  26 x 512   ->   26 x  26 x 256  0.177 BFLOPs
 92 conv   512  3 x 3 / 1   26 x  26 x 256   ->   26 x  26 x 512  1.595 BFLOPs
 93 conv   255  1 x 1 / 1   26 x  26 x 512   ->   26 x  26 x 255  0.177 BFLOPs
 94 detection
 95 route  91
 96 conv   128  1 x 1 / 1   26 x  26 x 256   ->   26 x  26 x 128  0.044 BFLOPs
 97 upsample        2x   26 x  26 x 128   ->   52 x  52 x 128
 98 route  97 36
 99 conv   128  1 x 1 / 1   52 x  52 x 384   ->   52 x  52 x 128  0.266 BFLOPs
100 conv   256  3 x 3 / 1   52 x  52 x 128   ->   52 x  52 x 256  1.595 BFLOPs
101 conv   128  1 x 1 / 1   52 x  52 x 256   ->   52 x  52 x 128  0.177 BFLOPs
102 conv   256  3 x 3 / 1   52 x  52 x 128   ->   52 x  52 x 256  1.595 BFLOPs
103 conv   128  1 x 1 / 1   52 x  52 x 256   ->   52 x  52 x 128  0.177 BFLOPs
104 conv   256  3 x 3 / 1   52 x  52 x 128   ->   52 x  52 x 256  1.595 BFLOPs
105 conv   255  1 x 1 / 1   52 x  52 x 256   ->   52 x  52 x 255  0.353 BFLOPs
106 detection
Loading weights from yolov3.weights...Done!
data/dog.jpg: Predicted in 7.581085  seconds.
truck: 93%
bicycle: 99%
dog: 99%
```

The above output indicates that it took 7.581 seconds and predicted a truck, a bicycle and a dog with the probabilities of 93%, 99% and 99%, respectively. If you open the *projections.png* file in the *darknet* directory, you should see those bounding boxes predicted as shown in Fig. C.4. You can also locate this image on the Dock by clicking on the Terminal icon labeled *darknet* as shown in Fig. C.4.

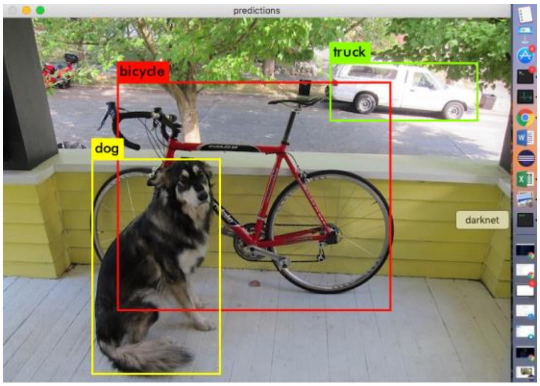


Figure C.4 Bounding boxes predicted by YOLOv3 with its own pre-trained weights.

Now you can press Ctrl-C to end the session. You can try another image, e.g., *data/horses.jpg* , and you would get the output as shown below, showing four horses have been detected, as shown in Fig. C.5:

```
data/horses.jpg: Predicted in 7.799737 seconds.
horse: 98%
horse: 97%
horse: 91%
horse: 89%
```

This is amazing that YOLO can easily tell what objects and how many are in a picture!

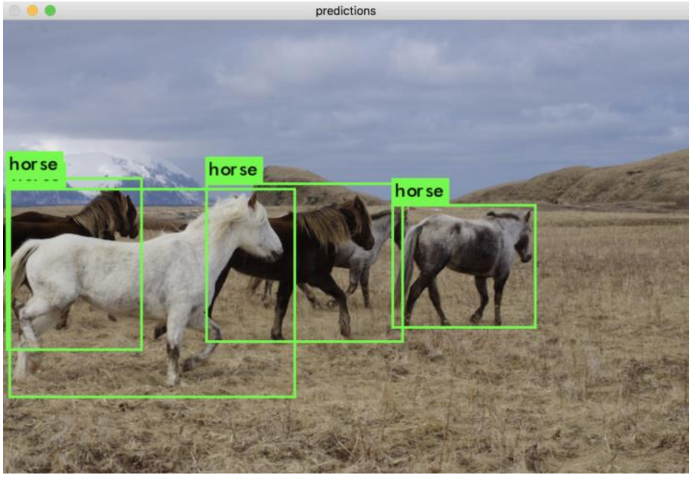


Figure C.5 Four horses detected by YOLO with its own pre-trained weights.

You can also try YOLO with live videos from a webcam as described there. I'll leave this to you, though.

C.8 TRAINING YOLO ON THE COCO DATASET

To train YOLO on the 2014 COCO dataset, check out this paper https://arxiv.org/pdf/1405.0312.pdf to learn a bit more about the COCO dataset. Then, download the 2014 COCO dataset directly from COCO's download site at http://cocodataset.org/#download, which is much faster than from YOLO's website. After downloading 2014 COCO dataset zip files, create a *data/coco/images* sub-directory in the *darknet* directory, and place the 2014 COCO zip files there. Then, follow the instructions at YOLO's main page at https://pjreddie.com/darknet/yolo/ under the section titled *Training YOLO on COCO*, except that you need to execute the following command:

```
$ cp scripts/get_coco_dataset.sh data
```

Then, make some changes in the *get_coco_dataset.sh* file as shown in Listing C.19. The commands I executed next were:

```
$ cd data
$ bash get_coco_dataset.sh
```

The *get_coco_dataset.sh* script, shown in Listing C.19, explains what this script does, as a good example for how to retrieve dataset and prepare the data. Note that downloading the COCO dataset may take many hours, depending on the Internet speed you have with your machine. In my case, downloading the *train*/*val*/*test* data concurrently took me about two hours at home with a download speed of up to 14 MB/s with a direct Ethernet cable connection, as shown in Fig. C.6.

Listing C.19 get_coco_dataset.sh (those marked red were modified)

```
#!/bin/bash

# Clone COCO API
#git clone https://github.com/pdollar/coco
cd coco

#mkdir images
#cd images

# Download Images
#wget -c https://pjreddie.com/media/files/train2014.zip
#wget -c https://pjreddie.com/media/files/val2014.zip

# Unzip
#unzip -q train2014.zip
#unzip -q val2014.zip
#unzip -q test2014.zip

#cd ..

# Download COCO Metadata
wget -c https://pjreddie.com/media/files/instances_train-val2014.zip
wget -c https://pjreddie.com/media/files/coco/5k.part
wget -c https://pjreddie.com/media/files/coco/trainvalno5k.part
wget -c https://pjreddie.com/media/files/coco/labels.tgz
tar xzf labels.tgz
unzip -q instances_train-val2014.zip

# Set Up Image Lists
paste <(awk "{print \"$PWD\"}" <5k.part) 5k.part | tr -d '\t' > 5k.txt
paste <(awk "{print \"$PWD\"}" <trainvalno5k.part) trainvalno5k.part | tr -d '\t' > trainvalno5k.txt
```

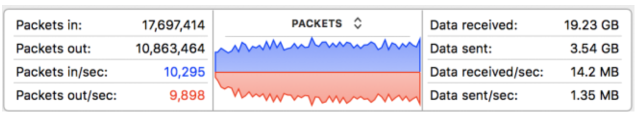


Figure C.6 COCO dataset download speed at home, directly from COCO's download site.

Then, I modified the `cfg/coco.data` file to have the following contents:

```
classes= 80
train  = data/coco/trainvalno5k.txt
valid = data/coco/5k.txt
names = data/coco.names
backup = backup
```

The *cfg/yolov3.cfg* was modified for training as follows, as marked in red (`max_batches` was changed from 500200 to 500 due to lack of a GPU on my machine):

```
[net]
# Testing
#batch=1
#subdivisions=1
# Training
batch=64
subdivisions=16
width=416
height=416
channels=3
momentum=0.9
decay=0.0005
angle=0
saturation = 1.5
exposure = 1.5
hue=.1

learning_rate=0.001
burn_in=1000
#max_batches = 500200
max_batches = 5000
policy=steps
steps=400000,450000
scales=.1,.1

[convolutional]
batch_normalize=1
filters=32
size=3
stride=1
pad=1
activation=leaky

# Downsample
[convolutional]
batch_normalize=1
filters=64
size=3
stride=2
pad=1
activation=leaky
...
[convolutional]
batch_normalize=1
size=3
stride=1
pad=1
filters=256
activation=leaky

[convolutional]
size=1
stride=1
pad=1
filters=255
activation=linear

[yolo]
mask = 0,1,2
anchors = 10,13,  16,30,  33,23,  30,61,  62,45,
59,119,  116,90,  156,198,  373,326
classes=80
num=9
jitter=.3
ignore_thresh = .5
truth_thresh = 1
random=1
```

Now you need to download the pre-trained convolutional weights that have been pre-trained on Imagenet using the darknet53 model. You can just download the weights for the convolutional layers by executing the following command:

```
$wget https://pjreddie.com/media/files/darknet53.conv.74
```

Then I trained the model with COCO dataset by executing the following command:

```
$./darknet detector train cfg/coco.data cfg/yolov3.cfg darknet53.conv.74
```

The output you get should be similar to what I got as shown below:

```
./darknet detector train cfg/coco.data cfg/yolov3.cfg darknet53.conv.74
yolov3
layer     filters    size              input                output
    0 conv     32  3 x 3 / 1   416 x 416 x   3   ->   416 x 416 x  32  0.299 BFLOPs
    1 conv     64  3 x 3 / 2   416 x 416 x  32   ->   208 x 208 x  64  1.595 BFLOPs
    2 conv     32  1 x 1 / 1   208 x 208 x  64   ->   208 x 208 x  32  0.177 BFLOPs
    3 conv     64  3 x 3 / 1   208 x 208 x  32   ->   208 x 208 x  64  1.595 BFLOPs
    4 res    1                 208 x 208 x  64   ->   208 x 208 x  64
    5 conv    128  3 x 3 / 2   208 x 208 x  64   ->   104 x 104 x 128  1.595 BFLOPs
    6 conv     64  1 x 1 / 1   104 x 104 x 128   ->   104 x 104 x  64  0.177 BFLOPs
    7 conv    128  3 x 3 / 1   104 x 104 x  64   ->   104 x 104 x 128  1.595 BFLOPs
    8 res    5                 104 x 104 x 128   ->   104 x 104 x 128
    9 conv     64  1 x 1 / 1   104 x 104 x 128   ->   104 x 104 x  64  0.177 BFLOPs
   10 conv    128  3 x 3 / 1   104 x 104 x  64   ->   104 x 104 x 128  1.595 BFLOPs
   11 res    8                 104 x 104 x 128   ->   104 x 104 x 128
   12 conv    256  3 x 3 / 2   104 x 104 x 128   ->    52 x  52 x 256  1.595 BFLOPs
   13 conv    128  1 x 1 / 1    52 x  52 x 256   ->    52 x  52 x 128  0.177 BFLOPs
   14 conv    256  3 x 3 / 1    52 x  52 x 128   ->    52 x  52 x 256  1.595 BFLOPs
   15 res   12                  52 x  52 x 256   ->    52 x  52 x 256
   16 conv    128  1 x 1 / 1    52 x  52 x 256   ->    52 x  52 x 128  0.177 BFLOPs
   17 conv    256  3 x 3 / 1    52 x  52 x 128   ->    52 x  52 x 256  1.595 BFLOPs
   18 res   15                  52 x  52 x 256   ->    52 x  52 x 256
   19 conv    128  1 x 1 / 1    52 x  52 x 256   ->    52 x  52 x 128  0.177 BFLOPs
   20 conv    256  3 x 3 / 1    52 x  52 x 128   ->    52 x  52 x 256  1.595 BFLOPs
   21 res   18                  52 x  52 x 256   ->    52 x  52 x 256
   22 conv    128  1 x 1 / 1    52 x  52 x 256   ->    52 x  52 x 128  0.177 BFLOPs
   23 conv    256  3 x 3 / 1    52 x  52 x 128   ->    52 x  52 x 256  1.595 BFLOPs
   24 res   21                  52 x  52 x 256   ->    52 x  52 x 256
   25 conv    128  1 x 1 / 1    52 x  52 x 256   ->    52 x  52 x 128  0.177 BFLOPs
   26 conv    256  3 x 3 / 1    52 x  52 x 128   ->    52 x  52 x 256  1.595 BFLOPs
   27 res   24                  52 x  52 x 256   ->    52 x  52 x 256
   28 conv    128  1 x 1 / 1    52 x  52 x 256   ->    52 x  52 x 128  0.177 BFLOPs
   29 conv    256  3 x 3 / 1    52 x  52 x 128   ->    52 x  52 x 256  1.595 BFLOPs
   30 res   27                  52 x  52 x 256   ->    52 x  52 x 256
   31 conv    128  1 x 1 / 1    52 x  52 x 256   ->    52 x  52 x 128  0.177 BFLOPs
   32 conv    256  3 x 3 / 1    52 x  52 x 128   ->    52 x  52 x 256  1.595 BFLOPs
   33 res   30                  52 x  52 x 256   ->    52 x  52 x 256
   34 conv    128  1 x 1 / 1    52 x  52 x 256   ->    52 x  52 x 128  0.177 BFLOPs
   35 conv    256  3 x 3 / 1    52 x  52 x 128   ->    52 x  52 x 256  1.595 BFLOPs
   36 res   33                  52 x  52 x 256   ->    52 x  52 x 256
   37 conv    512  3 x 3 / 2    52 x  52 x 256   ->    26 x  26 x 512  1.595 BFLOPs
   38 conv    256  1 x 1 / 1    26 x  26 x 512   ->    26 x  26 x 256  0.177 BFLOPs
   39 conv    512  3 x 3 / 1    26 x  26 x 256   ->    26 x  26 x 512  1.595 BFLOPs
   40 res   37                  26 x  26 x 512   ->    26 x  26 x 512
```

```
41 conv   256  1 x 1 / 1   26 x  26 x 512  ->  26 x  26 x 256  0.177 BFLOPs
42 conv   512  3 x 3 / 1   26 x  26 x 256  ->  26 x  26 x 512  1.595 BFLOPs
43 res  40                 26 x  26 x 512  ->  26 x  26 x 512
44 conv   256  1 x 1 / 1   26 x  26 x 512  ->  26 x  26 x 256  0.177 BFLOPs
45 conv   512  3 x 3 / 1   26 x  26 x 256  ->  26 x  26 x 512  1.595 BFLOPs
46 res  43                 26 x  26 x 512  ->  26 x  26 x 512
47 conv   256  1 x 1 / 1   26 x  26 x 512  ->  26 x  26 x 256  0.177 BFLOPs
48 conv   512  3 x 3 / 1   26 x  26 x 256  ->  26 x  26 x 512  1.595 BFLOPs
49 res  46                 26 x  26 x 512  ->  26 x  26 x 512
50 conv   256  1 x 1 / 1   26 x  26 x 512  ->  26 x  26 x 256  0.177 BFLOPs
51 conv   512  3 x 3 / 1   26 x  26 x 256  ->  26 x  26 x 512  1.595 BFLOPs
52 res  49                 26 x  26 x 512  ->  26 x  26 x 512
53 conv   256  1 x 1 / 1   26 x  26 x 512  ->  26 x  26 x 256  0.177 BFLOPs
54 conv   512  3 x 3 / 1   26 x  26 x 256  ->  26 x  26 x 512  1.595 BFLOPs
55 res  52                 26 x  26 x 512  ->  26 x  26 x 512
56 conv   256  1 x 1 / 1   26 x  26 x 512  ->  26 x  26 x 256  0.177 BFLOPs
57 conv   512  3 x 3 / 1   26 x  26 x 256  ->  26 x  26 x 512  1.595 BFLOPs
58 res  55                 26 x  26 x 512  ->  26 x  26 x 512
59 conv   256  1 x 1 / 1   26 x  26 x 512  ->  26 x  26 x 256  0.177 BFLOPs
60 conv   512  3 x 3 / 1   26 x  26 x 256  ->  26 x  26 x 512  1.595 BFLOPs
61 res  58                 26 x  26 x 512  ->  26 x  26 x 512
62 conv  1024  3 x 3 / 2   26 x  26 x 512  ->  13 x  13 x1024  1.595 BFLOPs
63 conv   512  1 x 1 / 1   13 x  13 x1024  ->  13 x  13 x 512  0.177 BFLOPs
64 conv  1024  3 x 3 / 1   13 x  13 x 512  ->  13 x  13 x1024  1.595 BFLOPs
65 res  62                 13 x  13 x1024  ->  13 x  13 x1024
66 conv   512  1 x 1 / 1   13 x  13 x1024  ->  13 x  13 x 512  0.177 BFLOPs
67 conv  1024  3 x 3 / 1   13 x  13 x 512  ->  13 x  13 x1024  1.595 BFLOPs
68 res  65                 13 x  13 x1024  ->  13 x  13 x1024
69 conv   512  1 x 1 / 1   13 x  13 x1024  ->  13 x  13 x 512  0.177 BFLOPs
70 conv  1024  3 x 3 / 1   13 x  13 x 512  ->  13 x  13 x1024  1.595 BFLOPs
71 res  68                 13 x  13 x1024  ->  13 x  13 x1024
72 conv   512  1 x 1 / 1   13 x  13 x1024  ->  13 x  13 x 512  0.177 BFLOPs
73 conv  1024  3 x 3 / 1   13 x  13 x 512  ->  13 x  13 x1024  1.595 BFLOPs
74 res  71                 13 x  13 x1024  ->  13 x  13 x1024
75 conv   512  1 x 1 / 1   13 x  13 x1024  ->  13 x  13 x 512  0.177 BFLOPs
76 conv  1024  3 x 3 / 1   13 x  13 x 512  ->  13 x  13 x1024  1.595 BFLOPs
77 conv   512  1 x 1 / 1   13 x  13 x1024  ->  13 x  13 x 512  0.177 BFLOPs
78 conv  1024  3 x 3 / 1   13 x  13 x 512  ->  13 x  13 x1024  1.595 BFLOPs
79 conv   512  1 x 1 / 1   13 x  13 x1024  ->  13 x  13 x 512  0.177 BFLOPs
80 conv  1024  3 x 3 / 1   13 x  13 x 512  ->  13 x  13 x1024  1.595 BFLOPs
81 conv   255  1 x 1 / 1   13 x  13 x1024  ->  13 x  13 x 255  0.088 BFLOPs
82 detection
83 route  79
84 conv   256  1 x 1 / 1   13 x  13 x 512  ->  13 x  13 x 256  0.044 BFLOPs
85 upsample          2x    13 x  13 x 256  ->  26 x  26 x 256
86 route  85 61
87 conv   256  1 x 1 / 1   26 x  26 x 768  ->  26 x  26 x 256  0.266 BFLOPs
88 conv   512  3 x 3 / 1   26 x  26 x 256  ->  26 x  26 x 512  1.595 BFLOPs
89 conv   256  1 x 1 / 1   26 x  26 x 512  ->  26 x  26 x 256  0.177 BFLOPs
```

```
 90 conv    512  3 x 3 / 1    26 x  26 x 256   ->    26 x  26 x 512   1.595 BFLOPs
 91 conv    256  1 x 1 / 1    26 x  26 x 512   ->    26 x  26 x 256   0.177 BFLOPs
 92 conv    512  3 x 3 / 1    26 x  26 x 256   ->    26 x  26 x 512   1.595 BFLOPs
 93 conv    255  1 x 1 / 1    26 x  26 x 512   ->    26 x  26 x 255   0.177 BFLOPs
 94 detection
 95 route  91
 96 conv    128  1 x 1 / 1    26 x  26 x 256   ->    26 x  26 x 128   0.044 BFLOPs
 97 upsample          2x    26 x  26 x 128   ->    52 x  52 x 128
 98 route  97 36
 99 conv    128  1 x 1 / 1    52 x  52 x 384   ->    52 x  52 x 128   0.266 BFLOPs
 100 conv    256  3 x 3 / 1    52 x  52 x 128   ->    52 x  52 x 256   1.595 BFLOPs
 101 conv    128  1 x 1 / 1    52 x  52 x 256   ->    52 x  52 x 128   0.177 BFLOPs
 102 conv    256  3 x 3 / 1    52 x  52 x 128   ->    52 x  52 x 256   1.595 BFLOPs
 103 conv    128  1 x 1 / 1    52 x  52 x 256   ->    52 x  52 x 128   0.177 BFLOPs
 104 conv    256  3 x 3 / 1    52 x  52 x 128   ->    52 x  52 x 256   1.595 BFLOPs
 105 conv    255  1 x 1 / 1    52 x  52 x 256   ->    52 x  52 x 255   0.353 BFLOPs
 106 detection
Loading weights from darknet53.conv.74...Done!
Learning Rate: 0.001, Momentum: 0.9, Decay: 0.0005
Resizing
352
Loaded: 0.142245  seconds
Region 82 Avg IOU: 0.306305, Class: 0.459760, Obj: 0.475054, No Obj: 0.443880, .5R: 0.200000, .75R: 0.000000,  count: 10
Region 94 Avg IOU: 0.191476, Class: 0.515562, Obj: 0.416233, No Obj: 0.460233, .5R: 0.000000, .75R: 0.000000,  count: 7
Region 106 Avg IOU: 0.412336, Class: 0.919581, Obj: 0.676026, No Obj: 0.467130, .5R: 0.000000, .75R: 0.000000,  count: 1
Region 82 Avg IOU: 0.092142, Class: 0.429719, Obj: 0.473738, No Obj: 0.443723, .5R: 0.000000, .75R: 0.000000,  count: 4
Region 94 Avg IOU: 0.235893, Class: 0.593401, Obj: 0.536550, No Obj: 0.458530, .5R: 0.000000, .75R: 0.000000,  count: 4
Region 106 Avg IOU: 0.113339, Class: 0.487863, Obj: 0.343040, No Obj: 0.470983, .5R: 0.000000, .75R: 0.000000,  count: 44
Region 82 Avg IOU: 0.393365, Class: 0.497587, Obj: 0.440454, No Obj: 0.444993, .5R: 0.166667, .75R: 0.000000,  count: 6
Region 94 Avg IOU: 0.319501, Class: 0.482832, Obj: 0.429928, No Obj: 0.457614, .5R: 0.333333, .75R: 0.000000,  count: 3
......
Region 82 Avg IOU: 0.303231, Class: 0.362318, Obj: 0.471073, No Obj: 0.442284, .5R: 0.142857, .75R: 0.000000,  count: 7
Region 94 Avg IOU: 0.205401, Class: 0.542749, Obj: 0.645735, No Obj: 0.460915, .5R: 0.000000, .75R: 0.000000,  count: 10
Region 106 Avg IOU: 0.122706, Class: 0.357472, Obj: 0.396068, No Obj: 0.466134, .5R: 0.000000, .75R: 0.000000,  count: 8
Region 82 Avg IOU: 0.199270, Class: 0.267530, Obj: 0.600786, No Obj: 0.444222, .5R: 0.000000, .75R: 0.000000,  count:  2
Region 94 Avg IOU: 0.224309, Class: 0.591808, Obj: 0.561265, No Obj: 0.459623, .5R: 0.000000, .75R: 0.000000,  count: 8
Region 106 Avg IOU: 0.222792, Class: 0.529287, Obj: 0.356624, No Obj: 0.471495, .5R: 0.166667, .75R: 0.000000,  count: 6
Region 82 Avg IOU: 0.301176, Class: 0.472930, Obj: 0.339670, No Obj: 0.445169, .5R: 0.000000, .75R: 0.000000,  count: 6
......
Region 106 Avg IOU: 0.260889, Class: 0.538027, Obj: 0.485092, No Obj: 0.472694, .5R: 0.133333, .75R: 0.000000,  count: 15
Region 82 Avg IOU: 0.227438, Class: 0.560036, Obj: 0.569116, No Obj: 0.443473, .5R: 0.142857, .75R: 0.000000,  count: 7
Region 94 Avg IOU: 0.227952, Class: 0.481299, Obj: 0.476429, No Obj: 0.460162, .5R: 0.000000, .75R: 0.000000,  count: 13
Region 106 Avg IOU: 0.168266, Class: 0.624322, Obj: 0.350685, No Obj: 0.466164, .5R: 0.000000, .75R: 0.000000,  count: 11
1: 728.814819, 728.814819 avg, 0.000000 rate, 1017.449199 seconds, 64 images
Loaded: 0.000042 seconds
Region 82 Avg IOU: 0.157086, Class: 0.629885, Obj: 0.393714, No Obj: 0.445066, .5R: 0.000000, .75R: 0.000000,  count: 5
Region 94 Avg IOU: 0.230181, Class: 0.666338, Obj: 0.736702, No Obj: 0.459474, .5R: 0.000000, .75R: 0.000000,  count: 5
Region 106 Avg IOU: 0.211484, Class: 0.269949, Obj: 0.356809, No Obj: 0.471995, .5R: 0.100000, .75R: 0.000000,  count: 10
......
Region 106 Avg IOU: 0.158112, Class: 0.301957, Obj: 0.455966, No Obj: 0.464749, .5R: 0.000000, .75R: 0.000000,  count: 4
2: 716.763916, 727.609741 avg, 0.000000 rate, 1039.281667 seconds, 128 images
Loaded: 0.000050 seconds
```

```
Region 82 Avg IOU: 0.205055, Class: 0.454246, Obj: 0.282755, No Obj: 0.444815, .5R: 0.000000, .75R: 0.000000, count: 3
Region 94 Avg IOU: 0.246594, Class: 0.530773, Obj: 0.470584, No Obj: 0.458768, .5R: 0.083333, .75R: 0.083333, count: 12
Region 106 Avg IOU: 0.069552, Class: 0.523594, Obj: 0.233925, No Obj: 0.467753, .5R: 0.000000, .75R: 0.000000, count: 7

......
```

So what does each output line shown above mean? I found that the above output is generated by line 239 from function `forward_yolo_layer` in the *yolo_layer.c* file as shown below:

```
printf("Region %d Avg IOU: %f, Class: %f, Obj: %f, No Obj: %f, .5R: %f, .75R:
    %f,  count: %d\n", net.index, avg_iou/count, avg_cat/class_count,
    avg_obj/count, avg_anyobj/(l.w*l.h*l.n*l.batch), recall/count,
    recall75/count, count);
```

So what is IOU? It stands for *Intersection over Union*, which measures how well the ground-truth bounding box overlaps with the predicted bounding box, as shown in Figure C.7. If IOU = 1, it means that the ground-truth bounding box and the predicted bounding box overlap exactly.

Figure C.7 IOU: Intersection over Union (Source: Courtesy of https://www.pyimagesearch.com/).

The portion of the code from function `forward_yolo_layer` is listed below, which shows how quantities on each of the output lines starting with "Region …" are computed:

```
int mask_n = int_index(l.mask, best_n, l.n);
if(mask_n >= 0){
  int box_index = entry_index(l, b, mask_n*l.w*l.h + j*l.w + i, 0);
  float iou = delta_yolo_box(truth, l.output, l.biases, best_n, box_index, i, j, l.w, l.h, net.w, net.h, l.delta,
     (2-truth.w*truth.h), l.w*l.h);

  int obj_index = entry_index(l, b, mask_n*l.w*l.h + j*l.w + i, 4);
  avg_obj += l.output[obj_index];
  l.delta[obj_index] = 1 - l.output[obj_index];

  int class = net.truth[t*(4 + 1) + b*l.truths + 4];
  if (l.map) class = l.map[class];
  int class_index = entry_index(l, b, mask_n*l.w*l.h + j*l.w + i, 4 + 1);
  delta_yolo_class(l.output, l.delta, class_index, class, l.classes, l.w*l.h, &avg_cat);

  ++count;
  ++class_count;
  if(iou > .5) recall += 1;
```

```
            if(iou > .75) recall75 += 1;
            avg_iou += iou;
        }
......
```

For your reference, I installed *darkent* on Eclipse IDE for C/C++, which helps manage files and search better. For example, Figure C.8 shows how to search a string pattern from all C source files under the *Remote Search* tab.

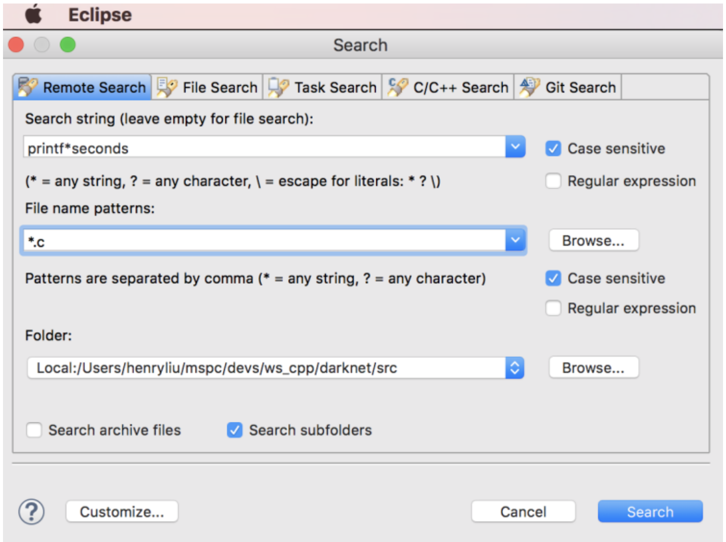


Figure C.8 Search capability from the Eclipse IDE for C/C++.

If you are interested in pursuing further, I suggest that you install *darknet* on an IDE like Eclipse and navigate through the relevant source files. It's amazing that the entire darknet library is written in C with just 45 files with a total size of ~455 kB, as shown below:

12746	yolo_layer.c	4262	matrix.c
14476	utils.c	24438	lstm_layer.c
3243	upsample_layer.c	2095	logistic_layer.c
3730	tree.c	8929	local_layer.c
3552	softmax_layer.c	1370	list.c
2940	shortcut_layer.c	4471	layer.c
3937	route_layer.c	1794	l2norm_layer.c
10093	rnn_layer.c	42105	image.c
5037	reorg_layer.c	1337	im2col.c
19388	region_layer.c	13715	gru_layer.c
44504	parser.c	8188	gemm.c
3121	option_list.c	1606	dropout_layer.c
5532	normalization_layer.c	10201	detection_layer.c
30214	network.c	10568	demo.c
3940	maxpool_layer.c	9787	deconvolutional_layer.c

44905	data.c	1340	col2im.c
4095	cuda.c	8435	box.c
2759	crop_layer.c	9397	blas.c
9388	crnn_layer.c	10366	batchnorm_layer.c
5174	cost_layer.c	1877	avgpool_layer.c
18620	convolutional_layer.c	3560	activations.c
11056	connected_layer.c	1707	activation_layer.c
10819	compare.c	454817	total_size_in_bytes

The *examples* directory contains driver code for specific examples, including *darknet.c*, *detector.c*, *network.c*, etc., which call darknet library functions defined in the *src* directory. The code execution path with the COCO training example will be illustrated in the next section..

Once again, from the previous output lines, it shows that the first batch of 64 images took about 1017/60 = 17 minutes or each image took about 1017/64 = 16 seconds, and the second batch of 64 images took about 1039/60 = 17 minutes as well or each image took about 1039/64 = 16 seconds as well. I started training on April 21, and as of May 27, I got:

```
1056: 10.239875, 7.783431 avg, 0.001000 rate, 1969.657717 seconds, 67584 images
```

That is, the training reached batch # 1056 with an average loss of 7.783431. Compared with the loss of 728 at batch #1, the loss is about 100x smaller, but still a long way to reach a desired loss of 0.06! This implies that it is too slow to train a deep neural network model on CPUs. However, you can still learn a lot even without access to GPUs, as we have demonstrated here.

C.9 PROFILING YOLO

YOLO is written in C. You might want to know how YOLO is coded exactly, in which case a call graph will help. Or, you might want to analyze YOLO's performance as a software program, in which case, you need to profile YOLO while it is running. I had similar interests and figured out how we can do this easily. It turned out that using *Instruments* - the profiling feature of the XCode IDE on macOS - is the easiest way out of several options. In this section, I share my experience with you on how to obtain call graphs and CPU usage profiles with the Instruments tool on macOS..

To use Instruments, you need to have XCode and its Command Line Tools installed on your macOS, which you should already have if you did not skip §C.5.1. Then, just fire up *darknet* with a training task such as the one with COCO we demonstrated early. The next step is to find the process id (PID) of darknet, as shown from the Activity Monitor in Figure C.9, which was 13557 in my case. Finally, execute the following command with the *darknet*'s PID as shown below as in my case:

```
$ instruments -l 60000 -t Time\ Profiler -p 13557
```

The above command instructs the Instruments tool to instrument the darknet process for 60000 milliseconds or 60 seconds while it is running. If you get an error like

```
xcode-select: error: tool 'instruments' requires Xcode, but active developer directory
'/Library/Developer/CommandLineTools' is a command line tools instance
```

, then, executing the following command should fix it as in my case:

$ sudo xcode-select -s /Applications/Xcode.app/Contents/Developer

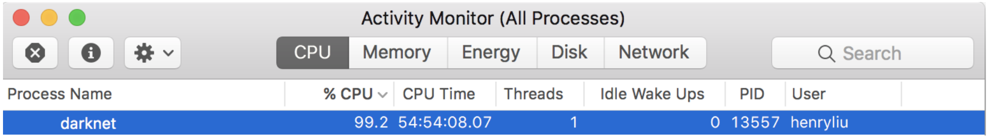


Figure C.9 The PID of the darknet process displayed on the Activity Monitor.

After the specified amount of time has passed, look for the Instruments icon on the Dock as shown below:

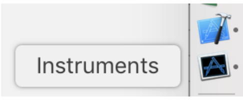


Figure C.10 The Instruments icon on the Dock.

Clicking on the above icon should bring up the Instruments panel as shown in Figure C.11. As you see, I got both the call graph and CPU stats in one shot. It is seen that *darknet*'s `main` function calls the `train_detector` function, which calls the `train_network` function, which in turn calls the `train_network_datum` function and the `get_next_batch` function.

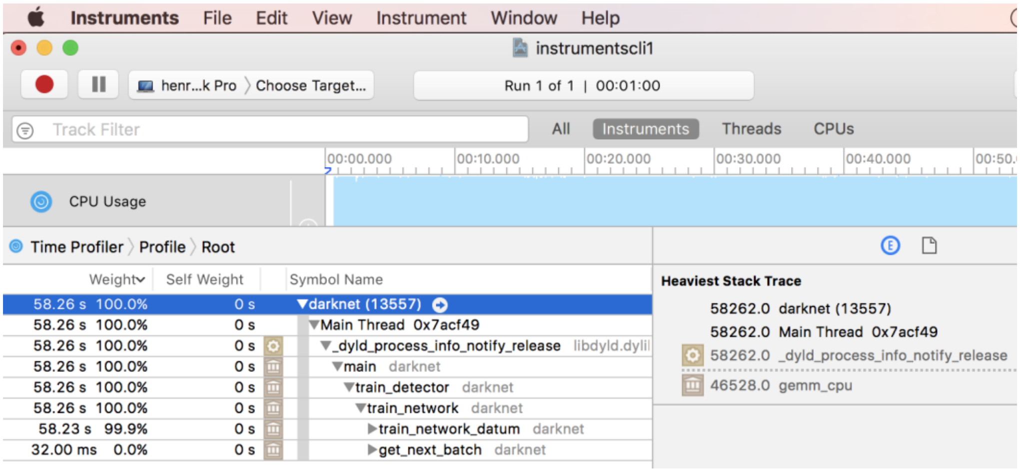


Figure C.11 The darknet CPU stats profiled with the Instruments tool.

I drilled down further by expanding the `train_network_datum` function, as shown in Figure C.12. It is seen that the `train_network_datum` function called two more functions: `forward_network` and `backward_convolutional_layer`, which took ~87% and ~13% of the total CPU time, respectively.

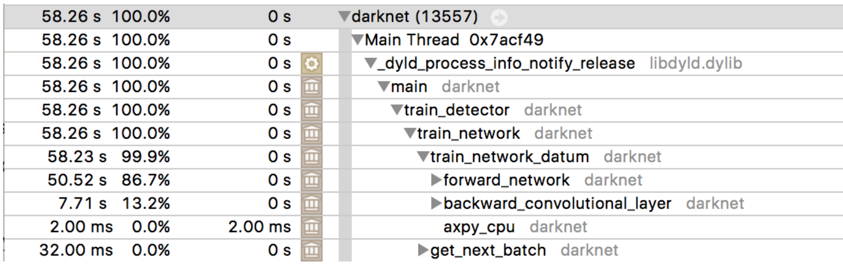


Figure C.12 CPU time breakdown between the *darknet*'s two functions of `forward_network` and `backward_convolutional_layer` as revealed by the *Instruments* tool.

By keeping expanding, we could drill down to deeper levels of the functions called, as shown in Figure C.13. For example, we could see the finer structure of the `forward_network` function, which calls:

- `forward_convolution_layer`
  - gemm_cpu
  - `forward_batchnorm_layer`
  - `im2col_cpu`
  - `activate_array`
- `activate_array`
- `forward_shortcut_layer`
- `forward_yolo_layer`
  - `activate_array`
- `forward_upsample_layer`
- `forward_route_layer`

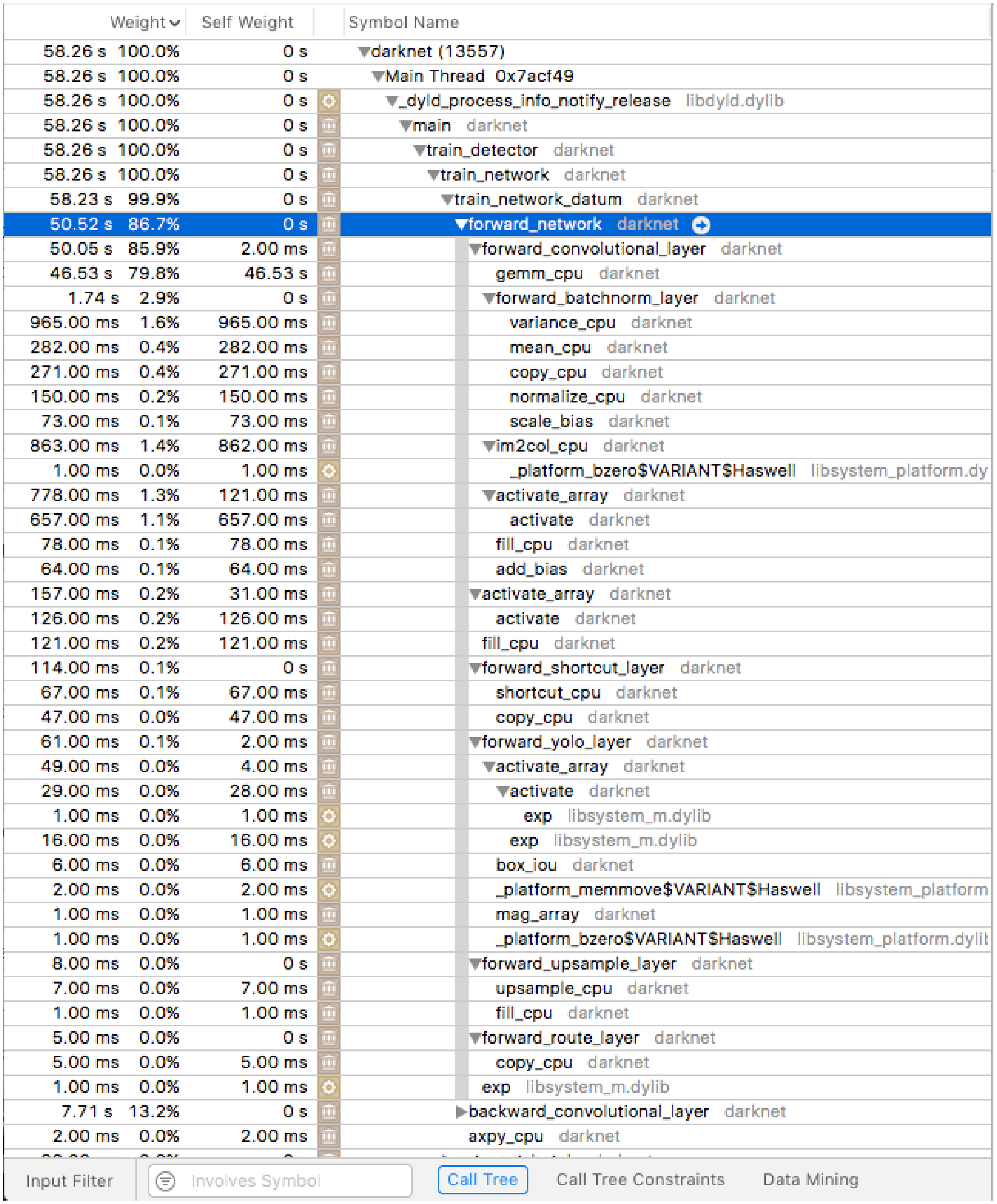
Based on the above call graph, we can check the source code to learn exactly how each function is implemented. I did a bit drill-down, which is shared next.

First, recall that our COCO training was kicked off with the following command:

```
./darknet detector train cfg/coco.data cfg/yolov3.cfg darknet53.conv.74
```

As shown in Fig. C.13, the program execution begins with the `main` function in *darknet.c*. The "`detector`" argument initiates calling the function `train_detector` in *detector.c*. Then, the "`train`" argument initiates calling the function `train_network` in *network.c*, which call the `train_network_datum` in *network.c* in turn. Refer to Listing C.20 for how this function is coded. As you see, this is where how `forward_network` and `backward_network` functions are called, how `error` is computed, and how the network is updated by calling the `update_network` function.

You may further notice in Figure C.13 that the `forward_network` function calls the `forward_convolutional_layer` function. How this call is initiated is explained following Listing C.20.



Weight		Self Weight	Symbol Name
58.26 s	100.0%	0 s	▼darknet (13557)
58.26 s	100.0%	0 s	▼Main Thread 0x7acf49
58.26 s	100.0%	0 s	▼_dyld_process_info_notify_release libdyld.dylib
58.26 s	100.0%	0 s	▼main darknet
58.26 s	100.0%	0 s	▼train_detector darknet
58.26 s	100.0%	0 s	▼train_network darknet
58.23 s	99.9%	0 s	▼train_network_datum darknet
50.52 s	86.7%	0 s	▼forward_network darknet
50.05 s	85.9%	2.00 ms	▼forward_convolutional_layer darknet
46.53 s	79.8%	46.53 s	gemm_cpu darknet
1.74 s	2.9%	0 s	▼forward_batchnorm_layer darknet
965.00 ms	1.6%	965.00 ms	variance_cpu darknet
282.00 ms	0.4%	282.00 ms	mean_cpu darknet
271.00 ms	0.4%	271.00 ms	copy_cpu darknet
150.00 ms	0.2%	150.00 ms	normalize_cpu darknet
73.00 ms	0.1%	73.00 ms	scale_bias darknet
863.00 ms	1.4%	862.00 ms	▼im2col_cpu darknet
1.00 ms	0.0%	1.00 ms	_platform_bzero$VARIANT$Haswell libsystem_platform.dy
778.00 ms	1.3%	121.00 ms	▼activate_array darknet
657.00 ms	1.1%	657.00 ms	activate darknet
78.00 ms	0.1%	78.00 ms	fill_cpu darknet
64.00 ms	0.1%	64.00 ms	add_bias darknet
157.00 ms	0.2%	31.00 ms	▼activate_array darknet
126.00 ms	0.2%	126.00 ms	activate darknet
121.00 ms	0.2%	121.00 ms	fill_cpu darknet
114.00 ms	0.1%	0 s	▼forward_shortcut_layer darknet
67.00 ms	0.1%	67.00 ms	shortcut_cpu darknet
47.00 ms	0.0%	47.00 ms	copy_cpu darknet
61.00 ms	0.1%	2.00 ms	▼forward_yolo_layer darknet
49.00 ms	0.0%	4.00 ms	▼activate_array darknet
29.00 ms	0.0%	28.00 ms	▼activate darknet
1.00 ms	0.0%	1.00 ms	exp libsystem_m.dylib
16.00 ms	0.0%	16.00 ms	exp libsystem_m.dylib
6.00 ms	0.0%	6.00 ms	box_iou darknet
2.00 ms	0.0%	2.00 ms	_platform_memmove$VARIANT$Haswell libsystem_platform
1.00 ms	0.0%	1.00 ms	mag_array darknet
1.00 ms	0.0%	1.00 ms	_platform_bzero$VARIANT$Haswell libsystem_platform.dylil
8.00 ms	0.0%	0 s	▼forward_upsample_layer darknet
7.00 ms	0.0%	7.00 ms	upsample_cpu darknet
1.00 ms	0.0%	1.00 ms	fill_cpu darknet
5.00 ms	0.0%	0 s	▼forward_route_layer darknet
5.00 ms	0.0%	5.00 ms	copy_cpu darknet
1.00 ms	0.0%	1.00 ms	exp libsystem_m.dylib
7.71 s	13.2%	0 s	▶backward_convolutional_layer darknet
2.00 ms	0.0%	2.00 ms	axpy_cpu darknet

Figure C.13 The darknet call graph as revealed with the Instruments tool.

Listing C.20 Function train_network_datum(network *net) (in network.c)

```
289 float train_network_datum(network *net)
```

```
{
    *net->seen += net->batch;
    net->train = 1;
    forward_network(net);
    backward_network(net);
    float error = *net->cost;
    if(((*net->seen)/net->batch)%net->subdivisions == 0)
       update_network(net);
    return error;
}
```

To understand how the forward_network function calls the forward_convolutional_layer function, we show the forward_network function in Listing C.21. This function essentially loops through all layers defined for a given network with the for-loop defined from line 198 to 208. The line 204 initiates the call to the forward_convolutional_layer function, defined at line 221 with the function make_convolutional_layer in *convolutional_layer.c*, shown in Listing C.22. This function demonstrates how a convolutional layer is made.

Listing C.21 Function forward_network(network *net) (in network.c)

```
188 void forward_network(network *netp)
189 {
190 #ifdef GPU
191     if(netp->gpu_index >= 0){
192         forward_network_gpu(netp);
193         return;
194     }
195 #endif
196     network net = *netp;
197     int i;
198     for(i = 0; i < net.n; ++i){
199         net.index = i;
200         layer l = net.layers[i];
201         if(l.delta){
202             fill_cpu(l.outputs * l.batch, 0, l.delta, 1);
203         }
204         l.forward(l, net);
205         net.input = l.output;
206         if(l.truth) {
207             net.truth = l.output;
208         }
209     }
210     calc_network_cost(netp);
211 }
```

Listing C.22 Function make_convolutional_layer (in src/convolutional_layer.c)

```
176 convolutional_layer make_convolutional_layer(int batch, int h, int w, int
    c, int n, int groups, int size, int stride, int padding, ACTIVATION
    activation, int batch_normalize, int binary, int xnor, int adam)
177 {
```

```
    int i;
    convolutional_layer l = {0};
    l.type = CONVOLUTIONAL;

    l.groups = groups;
    l.h = h;
    l.w = w;
    l.c = c;
    l.n = n;
    l.binary = binary;
    l.xnor = xnor;
    l.batch = batch;
    l.stride = stride;
    l.size = size;
    l.pad = padding;
    l.batch_normalize = batch_normalize;

    l.weights = calloc(c/groups*n*size*size, sizeof(float));
    l.weight_updates = calloc(c/groups*n*size*size, sizeof(float));

    l.biases = calloc(n, sizeof(float));
    l.bias_updates = calloc(n, sizeof(float));

    l.nweights = c/groups*n*size*size;
    l.nbiases = n;

    // float scale = 1./sqrt(size*size*c);
    float scale = sqrt(2./(size*size*c/l.groups));
    //printf("convscale %f\n", scale);
    //scale = .02;
    //for(i = 0; i < c*n*size*size; ++i) l.weights[i] =
       scale*rand_uniform(-1, 1);
    for(i = 0; i < l.nweights; ++i) l.weights[i] = scale*rand_normal();
    int out_w = convolutional_out_width(l);
    int out_h = convolutional_out_height(l);
    l.out_h = out_h;
    l.out_w = out_w;
    l.out_c = n;
    l.outputs = l.out_h * l.out_w * l.out_c;
    l.inputs = l.w * l.h * l.c;

    l.output = calloc(l.batch*l.outputs, sizeof(float));
    l.delta  = calloc(l.batch*l.outputs, sizeof(float));

    l.forward = forward_convolutional_layer;
    l.backward = backward_convolutional_layer;
    l.update = update_convolutional_layer;
    if(binary){
        l.binary_weights = calloc(l.nweights, sizeof(float));
        l.cweights = calloc(l.nweights, sizeof(char));
        l.scales = calloc(n, sizeof(float));
    }
    if(xnor){
```

```
        l.binary_weights = calloc(l.nweights, sizeof(float));
        l.binary_input = calloc(l.inputs*l.batch, sizeof(float));
    }

    if(batch_normalize){
        l.scales = calloc(n, sizeof(float));
        l.scale_updates = calloc(n, sizeof(float));
        for(i = 0; i < n; ++i){
            l.scales[i] = 1;
        }

        l.mean = calloc(n, sizeof(float));
        l.variance = calloc(n, sizeof(float));

        l.mean_delta = calloc(n, sizeof(float));
        l.variance_delta = calloc(n, sizeof(float));

        l.rolling_mean = calloc(n, sizeof(float));
        l.rolling_variance = calloc(n, sizeof(float));
        l.x = calloc(l.batch*l.outputs, sizeof(float));
        l.x_norm = calloc(l.batch*l.outputs, sizeof(float));
    }
    if(adam){
        l.m = calloc(l.nweights, sizeof(float));
        l.v = calloc(l.nweights, sizeof(float));
        l.bias_m = calloc(n, sizeof(float));
        l.scale_m = calloc(n, sizeof(float));
        l.bias_v = calloc(n, sizeof(float));
        l.scale_v = calloc(n, sizeof(float));
    }

  ......
    l.workspace_size = get_workspace_size(l);
    l.activation = activation;

    fprintf(stderr, "conv  %5d %2d x%2d /%2d  %4d x%4d x%4d   ->  %4d x%4d
x%4d  %5.3f BFLOPs\n", n, size, size, stride, w, h, c, l.out_w, l.out_h,
l.out_c, (2.0 * l.n * l.size*l.size*l.c/l.groups *
l.out_h*l.out_w)/1000000000.);

    return l;
}
```

Listing C.23 shows how the key CNN functions of forward_convolutional_layer, backward_convolutional_layer, and update_convolutional_layer are implemented in YOLOV3. These functions explain how these common CNN layers work. It is seen that both the forward and backward convolutional layers call the gemm function, which took about 80% of the total CPU time as shown in Figure C.1.3. The update convolutional layer calls the CPU-version of the axpy function in place of the gemm function. These two functions of gemm and axpy are explained in the next section.

Listing C.23 Function forward_convolutional_layer (in src/convolutional_layer.c)

```
void forward_convolutional_layer(convolutional_layer l, network net)
{
    int i, j;

    fill_cpu(l.outputs*l.batch, 0, l.output, 1);

    if(l.xnor){
        binarize_weights(l.weights, l.n, l.c/l.groups*l.size*l.size,
          l.binary_weights);
        swap_binary(&l);
        binarize_cpu(net.input, l.c*l.h*l.w*l.batch, l.binary_input);
        net.input = l.binary_input;
    }

    int m = l.n/l.groups;
    int k = l.size*l.size*l.c/l.groups;
    int n = l.out_w*l.out_h;
    for(i = 0; i < l.batch; ++i){
        for(j = 0; j < l.groups; ++j){
            float *a = l.weights + j*l.nweights/l.groups;
            float *b = net.workspace;
            float *c = l.output + (i*l.groups + j)*n*m;

            im2col_cpu(net.input + (i*l.groups + j)*l.c/l.groups*l.h*l.w,
                l.c/l.groups, l.h, l.w, l.size, l.stride, l.pad, b);
            gemm(0,0,m,n,k,1,a,k,b,n,1,c,n);
        }
    }

    if(l.batch_normalize){
        forward_batchnorm_layer(l, net);
    } else {
        add_bias(l.output, l.biases, l.batch, l.n, l.out_h*l.out_w);
    }

    activate_array(l.output, l.outputs*l.batch, l.activation);
    if(l.binary || l.xnor) swap_binary(&l);
}

void backward_convolutional_layer(convolutional_layer l, network net)
{
    int i, j;
    int m = l.n/l.groups;
    int n = l.size*l.size*l.c/l.groups;
    int k = l.out_w*l.out_h;

    gradient_array(l.output, l.outputs*l.batch, l.activation, l.delta);

    if(l.batch_normalize){
        backward_batchnorm_layer(l, net);
    } else {
        backward_bias(l.bias_updates, l.delta, l.batch, l.n, k);
    }
```

```

    for(i = 0; i < l.batch; ++i){
        for(j = 0; j < l.groups; ++j){
            float *a = l.delta + (i*l.groups + j)*m*k;
            float *b = net.workspace;
            float *c = l.weight_updates + j*l.nweights/l.groups;

            float *im = net.input+(i*l.groups + j)*l.c/l.groups*l.h*l.w;

            im2col_cpu(im, l.c/l.groups, l.h, l.w,
                    l.size, l.stride, l.pad, b);
            gemm(0,1,m,n,k,1,a,k,b,k,1,c,n);

            if(net.delta){
                a = l.weights + j*l.nweights/l.groups;
                b = l.delta + (i*l.groups + j)*m*k;
                c = net.workspace;

                gemm(1,0,n,k,m,1,a,n,b,k,0,c,k);

                col2im_cpu(net.workspace, l.c/l.groups, l.h, l.w, l.size,
                  l.stride,
                    l.pad, net.delta + (i*l.groups +
                       j)*l.c/l.groups*l.h*l.w);
            }
        }
    }
}

void update_convolutional_layer(convolutional_layer l, update_args a)
{
    float learning_rate = a.learning_rate*l.learning_rate_scale;
    float momentum = a.momentum;
    float decay = a.decay;
    int batch = a.batch;

    axpy_cpu(l.n, learning_rate/batch, l.bias_updates, 1, l.biases, 1);
    scal_cpu(l.n, momentum, l.bias_updates, 1);

    if(l.scales){
        axpy_cpu(l.n, learning_rate/batch, l.scale_updates, 1, l.scales,
          1);
        scal_cpu(l.n, momentum, l.scale_updates, 1);
    }

    axpy_cpu(l.nweights, -decay*batch, l.weights, 1, l.weight_updates, 1);
    axpy_cpu(l.nweights, learning_rate/batch, l.weight_updates, 1,
       l.weights, 1);
    scal_cpu(l.nweights, momentum, l.weight_updates, 1);
}
```

C.10 THE GEMM AND AXPY FUNCTIONS

So what are the `gemm` and `axpy` functions after all? Listing C.24 shows the CPU-version of the `gemm` function from line 145-166, together with one of its variants `gemm_tt` shown from line 126 to 142. Listing C.25 shows the CPU version of the `axpy` function, together with the `scale` and `fill` functions as well. As is explained in wiki https://en.wikipedia.org/wiki/Basic_Linear_Algebra_Subprograms, the BLAS (Basic Linear Algebra Subprograms) spec specifies three levels of vector-matrix computations as follows:

- Level 1 (axpy): $\mathbf{y} \leftarrow \alpha\mathbf{x} + \mathbf{y}$ where $\mathbf{x}$ and $\mathbf{y}$ are vectors and α is a coefficient.
- Level 2 (gemv): $\mathbf{y} \leftarrow \alpha\mathbf{A}\mathbf{x} + \beta\,\mathbf{y}$ where matrix $\mathbf{A}$ and coefficient β are added.
- Level 3 (gemm): $\mathbf{C} \leftarrow \alpha\mathbf{A}\mathbf{B} + \beta\,\mathbf{C}$ where vectors $\mathbf{x}$ and $\mathbf{y}$ in level 2 (gemv) are replaced with matrices $\mathbf{B}$ and $\mathbf{C}$, respectively.

As is seen, these functions are essentially multiplication functions involving constant coefficients, vectors and matrices. They are in general optimized and tuned. Not surprisingly, they are at the core of machine learning in general and deep learning in particular. More detailed coverage is beyond the scope of this text.

Listing C.24 Function gemm (in src/gemm.c)

```
126 void gemm_tt(int M, int N, int K, float ALPHA,
127         float *A, int lda,
128         float *B, int ldb,
129         float *C, int ldc)
130 {
131     int i,j,k;
132     #pragma omp parallel for
133     for(i = 0; i < M; ++i){
134         for(j = 0; j < N; ++j){
135             register float sum = 0;
136             for(k = 0; k < K; ++k){
137                 sum += ALPHA*A[i+k*lda]*B[k+j*ldb];
138             }
139             C[i*ldc+j] += sum;
140         }
141     }
142 }
143
144
145 void gemm_cpu(int TA, int TB, int M, int N, int K, float ALPHA,
146         float *A, int lda,
147         float *B, int ldb,
148         float BETA,
149         float *C, int ldc)
150 {
151     //printf("cpu: %d %d %d %d %d %f %d %d %f %d\n",TA, TB, M, N, K,
             ALPHA, lda, ldb, BETA, ldc);
152     int i, j;
153     for(i = 0; i < M; ++i){
```

```
        for(j = 0; j < N; ++j){
            C[i*ldc + j] *= BETA;
        }
    }
    if(!TA && !TB)
        gemm_nn(M, N, K, ALPHA,A,lda, B, ldb,C,ldc);
    else if(TA && !TB)
        gemm_tn(M, N, K, ALPHA,A,lda, B, ldb,C,ldc);
    else if(!TA && TB)
        gemm_nt(M, N, K, ALPHA,A,lda, B, ldb,C,ldc);
    else
        gemm_tt(M, N, K, ALPHA,A,lda, B, ldb,C,ldc);
}
```

Listing C.25 CPU-version of the axpy, scale, and fill functions (in src/blas.c)

```
void axpy_cpu(int N, float ALPHA, float *X, int INCX, float *Y, int INCY)
{
    int i;
    for(i = 0; i < N; ++i) Y[i*INCY] += ALPHA*X[i*INCX];
}

void scal_cpu(int N, float ALPHA, float *X, int INCX)
{
    int i;
    for(i = 0; i < N; ++i) X[i*INCX] *= ALPHA;
}

void fill_cpu(int N, float ALPHA, float *X, int INCX)
{
    int i;
    for(i = 0; i < N; ++i) X[i*INCX] = ALPHA;
}
```

C.11 HOW YOLOV3 TRAINING IS KICKED OFF

It is interesting to explore how YOLOv3 training is kicked off exactly with the COCO dataset as an example. Most of the logistics is entailed in the `train_detector` function in *detector.c*. This function is shown in Listing C.26. This function is a bit lengthy, but the main logic is in the `while`-loop from line 62-147. The loop condition is that *the current batch is smaller than the max_batches* parameter specified in the *yolov3.cfg* file introduced earlier. If you have basic knowledge about C and CNN models as introduced in the main text, it's not hard to understand this piece of code, which is left as an exercise for those who are interested in such details.

This is all we cover on how YOLOv3 is implemented in C. You can continue to explore if you are interested.

Listing C.26 train_detector function (in examples/detetcor.c)

```
void train_detector(char *datacfg, char *cfgfile, char *weightfile, int
*gpus, int ngpus, int clear)
{
    list *options = read_data_cfg(datacfg);
    char *train_images = option_find_str(options, "train",
       "data/train.list");
    char *backup_directory = option_find_str(options, "backup",
       "/backup/");

    srand(time(0));
    char *base = basecfg(cfgfile);
    printf("%s\n", base);
    float avg_loss = -1;
    network **nets = calloc(ngpus, sizeof(network));

    srand(time(0));
    int seed = rand();
    int i;
    for(i = 0; i < ngpus; ++i){
        srand(seed);
#ifdef GPU
        cuda_set_device(gpus[i]);
#endif
        nets[i] = load_network(cfgfile, weightfile, clear);
        nets[i]->learning_rate *= ngpus;
    }
    srand(time(0));
    network *net = nets[0];

    int imgs = net->batch * net->subdivisions * ngpus;
    printf("Learning Rate: %g, Momentum: %g, Decay: %g\n",
       net->learning_rate, net->momentum, net->decay);
    data train, buffer;

    layer l = net->layers[net->n - 1];

    int classes = l.classes;
    float jitter = l.jitter;

    list *plist = get_paths(train_images);
    //int N = plist->size;
    char **paths = (char **)list_to_array(plist);

    load_args args = get_base_args(net);
    args.coords = l.coords;
    args.paths = paths;
    args.n = imgs;
    args.m = plist->size;
    args.classes = classes;
    args.jitter = jitter;
    args.num_boxes = l.max_boxes;
    args.d = &buffer;
    args.type = DETECTION_DATA;
```

```
    //args.type = INSTANCE_DATA;
    args.threads = 64;

    pthread_t load_thread = load_data(args);
    double time;
    int count = 0;
    //while(i*imgs < N*120){
    while(get_current_batch(net) < net->max_batches){
        if(l.random && count++%10 == 0){
            printf("Resizing\n");
            int dim = (rand() % 10 + 10) * 32;
            if (get_current_batch(net)+200 > net->max_batches) dim = 608;
            //int dim = (rand() % 4 + 16) * 32;
            printf("%d\n", dim);
            args.w = dim;
            args.h = dim;

            pthread_join(load_thread, 0);
            train = buffer;
            free_data(train);
            load_thread = load_data(args);

            #pragma omp parallel for
            for(i = 0; i < ngpus; ++i){
                resize_network(nets[i], dim, dim);
            }
            net = nets[0];
        }
        time=what_time_is_it_now();
        pthread_join(load_thread, 0);
        train = buffer;
        load_thread = load_data(args);

        /*
           int k;
           for(k = 0; k < l.max_boxes; ++k){
           box b = float_to_box(train.y.vals[10] + 1 + k*5);
           if(!b.x) break;
           printf("loaded: %f %f %f %f\n", b.x, b.y, b.w, b.h);
           }
         */
        /*
           int zz;
           for(zz = 0; zz < train.X.cols; ++zz){
           image im = float_to_image(net->w, net->h, 3, train.X.vals[zz]);
           int k;
           for(k = 0; k < l.max_boxes; ++k){
           box b = float_to_box(train.y.vals[zz] + k*5, 1);
           printf("%f %f %f %f\n", b.x, b.y, b.w, b.h);
           draw_bbox(im, b, 1, 1,0,0);
           }
           show_image(im, "truth11");
```

```
            cvWaitKey(0);
            save_image(im, "truth11");
            }
          */

        printf("Loaded: %lf seconds\n", what_time_is_it_now()-time);

        time=what_time_is_it_now();
        float loss = 0;
#ifdef GPU
        if(ngpus == 1){
            loss = train_network(net, train);
        } else {
            loss = train_networks(nets, ngpus, train, 4);
        }
#else
        loss = train_network(net, train);
#endif
        if (avg_loss < 0) avg_loss = loss;
        avg_loss = avg_loss*.9 + loss*.1;

        i = get_current_batch(net);
        printf("%ld: %f, %f avg, %f rate, %lf seconds, %d images\n",
          get_current_batch(net), loss, avg_loss, get_current_rate(net),
             what_time_is_it_now()-time, i*imgs);
        if(i%100==0){
#ifdef GPU
            if(ngpus != 1) sync_nets(nets, ngpus, 0);
#endif
            char buff[256];
            sprintf(buff, "%s/%s.backup", backup_directory, base);
            save_weights(net, buff);
        }
        if(i%10000==0 || (i < 1000 && i%100 == 0)){
#ifdef GPU
            if(ngpus != 1) sync_nets(nets, ngpus, 0);
#endif
            char buff[256];
            sprintf(buff, "%s/%s_%d.weights", backup_directory, base, i);
            save_weights(net, buff);
        }
        free_data(train);
    }
#ifdef GPU
    if(ngpus != 1) sync_nets(nets, ngpus, 0);
#endif
    char buff[256];
    sprintf(buff, "%s/%s_final.weights", backup_directory, base);
    save_weights(net, buff);
}
```