**Edited for errors by Drew Reisinger**

**Deep Learning Software**

One factor underlying the widespread adoption of neural network models is the availability of software packages that make these models easy to develop and train.

A range of such packages exist; examples include TensorFlow (https://www.tensorflow.org/), Theano (http://deeplearning.net/software/theano/), Keras (https://keras.io/), Caffe (http://caffe.berkeleyvision.org/), CNTK (https://github.com/Microsoft/CNTK) and others. All of these are widely used, and many are supported by large companies or research labs. Each has its own strengths and weaknesses, and while all of them are in principle general purpose, the details of each may suit them to particular projects.

For this tutorial, we use a framework called Torch, written in the programming language Lua. We chose Torch for this exercise because Torch models require relatively little infrastructure and boilerplate code, and we can train Torch models by hand (rather than calling a pre-baked training routine), which is useful for pedagogical purposes. You can install Torch for OS X or Linux by following these instructions: <http://torch.ch/docs/getting-started.html>. Windows instructions are available here: <https://github.com/BTNC/distro-win/blob/master/win-files/README.md>

**The MNIST dataset**

This exercise will use the popular MNIST dataset of handwritten digits. The dataset consists of 32 x 32 grayscale images of handwritten digits, each paired with its ground truth label (‘1’, ‘2’, etc.). The dataset is split into a training set of 60,000 images and a test set of 10,000 images. The goal of this exercise is to use the training set to learn a model that can correctly classify images in the test set.

We will compare a few different models on this task: convolutional neural networks with different numbers of layers, and a simple linear classifier.

**A brief introduction to Lua**

Lua, the language Torch is written in, is lightweight and simple, and easy to pick up from a Python background.

1. Tables: Lua has one main data structure, called a table. A table is constructed with curly braces {} and indexed with square brackets []. A table can act like a Python list (indexed from 1, not zero!):

t = {“a”, “b”}

print(t[1]) –> “a”

Or like a Python dictionary:

t = {a = 1, b = 2}

print(t[“a”]) ->  1

1. Scope: Variables in Lua are global by default. Local variables must be declared with the local keyword:

local a = 20

1. Objects: You won’t have to do any object-oriented programming for this exercise, but it’s good to know about Lua’s object-oriented syntax so that you can understand and call the helper code provided. As in other languages, a Lua object’s attributes are accessed with a period: obj.attr, but unlike other languages, its methods are accessed with a colon: obj:meth(args).
2. Miscellaneous syntax. The following code snippet exhibits most of the other Lua syntax you’ll need to know:

function parity(num)

local evens = {}

local odds = {}

for i = 1,num do

if i%2 == 0 then

evens[#evens + 1] = i

else

odds[#odds + 1] = i

end

end

return evens, odds

         end

Note that:

* 1. loops begin with a *do* and end with an *end* (whitespace is not semantically meaningful),
  2. single line comments are indicated with two dashes: --,
  3. Conditionals have the form if … then … else … end
  4. For a table t, #t gives the number of elements in a t. So if you want to append an element e into t, you can write

t[#t + 1] = e (or table.insert(t, e))

You won’t have to delve very deep into Lua to do these exercises, but if you would like more background, both of these brief tutorials are good:

Learn Lua in 15 minutes:

<http://tylerneylon.com/a/learn-lua/>

Lua for programmers, parts 1-3:

<http://nova-fusion.com/2012/08/27/lua-for-programmers-part-1/>

**Torch**

Torch is a framework for efficient numerical computing written on top of Lua, similarly to the way Numpy supplements Python. The main data structure in Torch is the Tensor, a multidimensional generalization of a matrix, similar in many ways to a Numpy array.

Rather than presenting the methods for manipulating tensors here, we refer the reader to the following two guides, which give Torch translations for common Numpy and MATLAB operations.

Torch for Numpy users:

<https://github.com/torch/torch7/wiki/Torch-for-Numpy-users>

Torch for MATLAB users:

<http://atamahjoubfar.github.io/Torch_for_Matlab_users.pdf>

While Torch can be used as a general-purpose numerical computing package, we are particularly interested in Torch’s built-in tools for building neural networks. The next few sections show how to use these tools to build networks for our MNIST task.

**Building a CNN in Torch**

Since we’re building a neural network, we’ll begin by importing the nn (neural network) package:

require “nn”

Now, as we saw in the chapter, a convolutional network applies a sequence of operations (convolutions, pooling, nonlinearities) to an input image. Accordingly, the network we build will be held in something called a sequential container. We begin by initializing an empty container, and calling it cnn.

   cnn = nn.Sequential()

***Convolutional Layer***

Now we will build our network by adding layers to this container. To give a preview of the final result, our complete network will look like this:

cnn = nn.Sequential()

cnn:add(nn.SpatialConvolutionMM(1, 32, 5, 5))

cnn:add(nn.ReLU())

cnn:add(nn.SpatialMaxPooling(2, 2))

cnn:add(nn.SpatialConvolutionMM(32, 64, 5, 5))

cnn:add(nn.ReLU())

cnn:add(nn.SpatialMaxPooling(2, 2))

cnn:add(nn.Reshape(64 \* 5 \*5))

cnn:add(nn.Linear(64 \* 5 \* 5, 10))

We will go through the construction of the network layer by layer, beginning with the first convolutional layer.

cnn:add(nn.SpatialConvolutionMM(1, 32, 5, 5))

Torch's spatial convolution layers take four parameters:

   nn.SpatialConvolutionMM(nInputPlanes,

 nOutputPlanes,

kernelWidth,

kernelHeight)

(1) nInputPlanes is the number of feature maps that will be passed into the layer. Our MNSIT images are black and white, so for this first convolutional layer, we use nInputPlanes = 1. (If we were dealing with RGB images, we would use nInputPlanes = 3.)

(2) nOutputPlanes is the number of feature of maps we want the layer to produce. Unlike nInputPlanes, nOutputPlanes does not depend on the previous layer, and we can choose for it any value we want. Choosing more output planes will make our model more expressive, but it will also add parameters, making it more difficult to train. Here we choose nOutputPlanes = 32.

(3) kernelWidth and kernelHeight are the height and width, respectively, of the kernel with which the layer will convolve its input. For this first layer, we want a kernel that is large enough to detect interesting structures, but small enough to introduce only a manageable number of parameters. We choose kernelWidth= kernelHight = 5.

***Nonlinearity***

In general, convolutional layers are followed by a nonlinear operation. A number of choices for this operation are possible, but a common choice is a rectified linear unit (ReLU), so this is what we add as our network’s next layer:

   cnn:add(nn.ReLU())

***Pooling***

As discussed in the chapter, adding a max pooling layer to our network will add a degree of translation invariance:

   cnn:add(nn.SpatialMaxPooling(2, 2))

The two parameters taken by the max pooling layer are the width and height of the region the layer will pool over. A layer that pools over a larger region will add more invariance, but will also discard some of the information present in its input. Here we choose a 2 x 2 pooling region.

Note that by default, Torch’s pooling layers pool over non-overlapping regions. Hence, our 2 x 2 pooling layer will downsample its input by a factor of two on each side.

***Making the network deeper***

The set of three operations we have used so far (convolution -> nonlinearity -> pooling) forms the heart of many CNNs. Indeed, many models mostly consist of repetitions of this motif with different parameters. This what we will do here: the next three layers of our model will again follow the convolutional -> nonlinearity -> pooling pattern, with the only difference from our existing layers being that our second convolutional layer now has nInputPlanes = 32 and nOutputPlanes = 64.

cnn:add(nn.SpatialConvolutionMM(32, 64, 5, 5))

cnn:add(nn.ReLU())

cnn:add(nn.SpatialMaxPooling(2, 2))

***Adding a classifier***

So far, our network finds a new representation for an input image, but it does not tell us the image’s label. To extract a label prediction from our network, we add a simple linear classifier on top of it.

The output of our current network is a three-dimensional tensor; to transform this into an appropriate classifier input, we need to reshape it into a one-dimensional vector:

cnn:add(nn.Reshape(64 \* 5 \*5))

The parameter taken by the Reshape layer is the number of elements we want the layer’s output vector to have. Since we are simply reshaping rather than adding or removing elements, this has to be the same as the number of elements in the original three-dimensional network output. The 64 \* 5 \* 5 number comes from the following calculation:

1. Convolutional layers (by default) only apply their kernels to parts of the image where they can fit without hanging off of the image’s edge. Both of our convolutional layers use 5 x 5 kernels which can only be applied in rows and columns with index at least three. Each of these layers, therefore, removes two pixels from each of the top, bottom, left and right sides of their input. Since our input images are 32 x 32, each plane in the output of the first convolutional layer has size 28 x 28. Since we used nOuputPlanes = 32, we have 32 x 28 x 28 tensor elements. (Note that the equality between the 32 input image side length and the 32 output planes is just a coincidence.)
2. As described above, each max-pooling layer downsamples its input by a factor of two, making the output of the first pooling layer have size 32 x 14 x 14.
3. Repeating the reasoning above shows that the second convolutional layer’s output has size 64 x10 x 10 (since this layer has 64 output planes), and the output of the second pooling layer has size 64 x 5 x 5. Since the nonlinearity preserves number of elements in inputs, this is our final answer.

Now we can add our classifier. Our classifier’s inputs will be vectors of size 64 x 5 x 5, and its outputs will be vectors of size 10, each entry of which contains the score for one class (digit). Since we are using a simple linear classifier, it can be added in one line:

cnn:add(nn.Linear(64 \* 5 \* 5, 10))

And that’s all – the network is done. Reprinted from above, the whole thing looks like this:

cnn = nn.Sequential()

cnn:add(nn.SpatialConvolutionMM(1, 32, 5, 5))

cnn:add(nn.ReLU)

cnn:add(nn.SpatialMaxPooling(2, 2))

cnn:add(nn.SpatialConvolutionMM(32, 64, 5, 5))

cnn:add(nn.ReLU())

cnn:add(nn.SpatialMaxPooling(2, 2))

cnn:add(nn.Reshape(64 \* 5 \*5))

cnn:add(nn.Linear(64 x 5 x 5, 10))

**Training the model**

With the model constructed, it remains to train it. As discussed in the chapter, the training procedure consists of repeating the process of running the model forward to a vector of class predictions for an training image, comparing this vector the image’s true label to obtain a loss, using backpropagation to obtain he gradients of this loss with respect to each of the model’s parameters, and updating the parameters in the direction of this gradient.

***Running the model forward***

We can select an image and its label from our training set as follows:

require “batches” -- needed to read data files

mnistTrain = torch.load("./data/trainingData.t7")

image, label = mnistTrain:getNextBatch(1)

(Don’t worry about the getNextBatch method for now; it will be explained later).

Now we can run the image through our network to a get a vector of class scores:

scores = cnn:forward(image)

Now we need a way of comparing these scores to the true label to get a loss value. Torch provides a number of ways (called nn.Criterion’s) to do this comparison; we’ll use one called a CrossEntropyCriterion, an information-theoretic measure of the difference between the probability distribution obtained by softmaxing our scores vector, and the delta distribution with all its mass concentrated on the correct label. We get our loss like this:  
  
 crit = nn.CrossEntropyCriterion()  
 loss = crit:forward(scores, label)

And that completes the forward pass through the network.

***Getting gradients by running the model backward***

Having computed a loss value with a forward pass through the model, we now calculate the gradient of this loss with respect to each parameter in the model. We do this with backpropagation, calculating gradients from the top layer of the network down. First, we find the gradient of loss with respect to scores. Just as the :forward() method propagated an image forward through the network and criterion, the :backward() method propagates gradients backwards:

dScores = crit:backward(scores, labels)

Next we use this gradient to find the rest of the gradients in the network:

cnn:backward(image, dScores)

The backward method calculates the loss gradient with respect to each of the model’s parameters, and stores them inside the model’s nn.Module objects. To make the model actually learn, we need to update the parameters in the direction of these gradients:

cnn:updateParameters(0.05)

Here, 0.05 is a learning rate parameter that controls how far the parameters move on each update.

Once we’ve done this update, we are done with gradients accumulated during the last backward pass, and we need to zero them to make sure that they don’t interfere with future updates:

cnn:zeroGradParameters()

***Training loop***

The four steps above (forward, backward, update, zero) form the heart of the training process; the training process as a whole just iterates them. The whole loop looks like this:

mnistTrain = torch.load("./data/trainingData.t7")

for i = 1,numSteps do

local images, labels = mnistTrain:getNextBatch(100)

local scores = cnn:forward(images)

local loss = crit:forward(scores, labels)

local dScores = crit:backward(scores, labels)

cnn:backward(images, dScores)

cnn:updateParameters(0.05)

cnn:zeroGradparameters()

end

Essentially, this is just a loop wrapper for the four steps introduced earlier, but with one difference. In section 6.1, we extracted and forwarded only a single image, but now we deal with a whole batch of 100 images, extracted from our dataset with mnistTrain:getNextBatch(100). Dealing with a whole batch at once lets us process lots of images with only one loop over our networks layers, and takes advantage of efficient tensor computations by gluing multiple images into a single higher-dimensional tensor.

Both the training and test datasets include the getNextBatch method, which takes a batch size parameter. The method keeps track of the last data index processed, and loops back to the beginning of the dataset when necessary.

A note on optimization algorithms: The optimization algorithm implemented above is called stochastic gradient descent (stochastic because the parameter updates are triggered by batches chosen randomly from the dataset as a whole), and is more or less the simplest algorithm used for training networks. We choose to use it here because it makes the forward, backward and update steps explicit. However, SGD is rarely the best choice for training a network in practice. State-of-the-art optimization algorithms still move a model’s parameters in the direction of the loss gradient, but they incorporate advanced features, such as adaptive learning rate selection. For examples of other optimization algorithms being used in Torch, see here https://github.com/torch/demos/blob/master/train-a-digit-classifier/train-on-mnist.lua.

***Assessing network performance***

As we train our network, we would like to keep track of how it is doing. We can measure its performance by getting predictions for some test images, and seeing if they agree with these images’ true labels.

The cross entropy criterion we used during training is a soft loss measure; for performance assessment, we are interested in hard accuracy: are our predictions right or wrong?

We have provided a function accuracy in util.lua that will do that for you. Calling

accuracy(scores, label)

extracts a class prediction from the scores vector, finding its argmax, the index with the largest score, and returns one if this prediction matches the true label, and zero otherwise. Like the network layers, the accuracy function also works with batches of data, computing the fraction of correct predictions in a batch.

Here’s how we can incorporate a running accuracy assessment in our training loop. First, let’s choose some test images to do our assessment on; for efficiency reasons, we will use the first 1,000 images in the test set, rather than all 10,000 of them:

testImages, testLabels = mnistTest:getNextBatch(1000)

Then we can add this code to our training loop, which will print an accuracy assessment every 100 training iterations:

if i % 100 == 0 then

local preds = cnn:forward(testImages)

print(accuracy(preds, testLabels)

end