Deep MNIST for Experts

TensorFlow is a powerful library for doing large-scale numerical computation. One of the tasks at which it excels is implementing and training deep neural networks. In this tutorial we will learn the basic building blocks of a TensorFlow model while constructing a deep convolutional MNIST classifier.

This introduction assumes familiarity with neural networks and the MNIST dataset. If you don't have a background with them, check out the introduction for beginners. Be sure to install TensorFlow before starting.

About this tutorial

The first part of this tutorial explains what is happening in the mnist_softmax.py code, which is a basic implementation of a Tensorflow model. The second part shows some ways to improve the accuracy.

You can copy and paste each code snippet from this tutorial into a Python environment, or you can choose to just read through the code.

What we will accomplish in this tutorial:

- Create a softmax regression function that is a model for recognizing MNIST digits, based on looking at every pixel in the image
- Use Tensorflow to train the model to recognize digits by having it "look" at thousands of examples (and run our first Tensorflow session to do so)
- Check the model's accuracy with our test data
- Build, train, and test a multilayer convolutional neural network to improve the results

Setup

Before we create our model, we will first load the MNIST dataset, and start a TensorFlow session.

Load MNIST Data

If you are copying and pasting in the code from this tutorial, start here with these two lines of code which will download and read in the data automatically:

```
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets('MNIST_data', one_hot=True)
```

Here `mnist` is a lightweight class which stores the training, validation, and testing sets as NumPy arrays. It also provides a function for iterating through data minibatches, which we will use below.

Start TensorFlow InteractiveSession

TensorFlow relies on a highly efficient C++ backend to do its computation. The connection to this backend is called a session. The common usage for TensorFlow programs is to first create a graph and then launch it in a session.

Here we instead use the convenient `InteractiveSession` class, which makes TensorFlow more flexible about how you structure your code. It allows you to interleave operations which build a computation graph with ones that run the graph. This is particularly convenient when working in interactive contexts like IPython. If you are not using an `InteractiveSession`, then you should build the entire computation graph before starting a session and launching the graph.

```
import tensorflow as tf
sess = tf.InteractiveSession()
```

Computation Graph

To do efficient numerical computing in Python, we typically use libraries like NumPy that do expensive operations such as matrix multiplication outside Python, using highly efficient code implemented in another language. Unfortunately, there can still be a lot of overhead from switching back to Python every operation. This overhead is especially bad if you want to run computations on GPUs or in a distributed manner, where there can be a high cost to transferring data.

TensorFlow also does its heavy lifting outside Python, but it takes things a step further to avoid this overhead. Instead of running a single expensive operation independently from Python, TensorFlow lets us describe a graph of interacting operations that run entirely outside Python. This approach is similar to that used in Theano or Torch.

The role of the Python code is therefore to build this external computation graph, and to dictate which parts of the computation graph should be run. See the Computation Graph section of Basic Usage for more detail.

Build a Softmax Regression Model

In this section we will build a softmax regression model with a single linear layer. In the next section, we will extend this to the case of softmax regression with a multilayer convolutional network.

Placeholders

We start building the computation graph by creating nodes for the input images and target output classes.

```
x = tf.placeholder(tf.float32, shape=[None, 784])
y_ = tf.placeholder(tf.float32, shape=[None, 10])
```

Here `x` and `y_` aren't specific values. Rather, they are each a `placeholder` -- a value that we'll input when we ask TensorFlow to run a computation.

The input images `x` will consist of a 2d tensor of floating point numbers. Here we assign it a `shape` of `[None, 784]`, where `784` is the dimensionality of a single flattened 28 by 28 pixel MNIST image, and `None` indicates that the first dimension, corresponding to the batch size, can be of any size. The target output classes `y_` will also consist of a 2d tensor, where each row is a one-hot 10-dimensional vector indicating which digit class (zero through nine) the corresponding MNIST image belongs to.

The `shape` argument to `placeholder` is optional, but it allows TensorFlow to automatically catch bugs stemming from inconsistent tensor shapes.

Variables

We now define the weights `W` and biases `b` for our model. We could imagine treating these like additional inputs, but TensorFlow has an even better way to handle them: `Variable`. A `Variable` is a value that lives in TensorFlow's computation graph. It can be used and even modified by the computation. In machine learning applications, one generally has the model parameters be `Variable`s.

```
W = tf.Variable(tf.zeros([784,10]))
b = tf.Variable(tf.zeros([10]))
```

We pass the initial value for each parameter in the call to `tf.Variable`. In this case, we initialize both `W` and `b` as tensors full of zeros. `W` is a 784x10 matrix (because we have 784 input features and 10 outputs) and `b` is a 10-dimensional vector (because we have 10 classes).

Before `Variable`s can be used within a session, they must be initialized using that session. This step takes the initial values (in this case tensors full of zeros) that have already been specified, and assigns them to each `Variable`. This can be done for all `Variables` at once:

```
sess.run(tf.initialize_all_variables())
```

Predicted Class and Loss Function

We can now implement our regression model. It only takes one line! We multiply the vectorized input images `x` by the weight matrix `W`, add the bias `b`.

```
y = tf.matmul(x,W) + b
```

We can specify a loss function just as easily. Loss indicates how bad the model's prediction was on a single example; we try to minimize that while training across all the examples. Here, our loss function is the cross-entropy between the target and the softmax activation function applied to the model's prediction. As in the beginners tutorial, we use the stable formulation:

```
cross_entropy = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(y, y_))
```

Note that `tf.nn.softmax_cross_entropy_with_logits` internally applies the softmax on the model's unnormalized model prediction and sums across all classes, and `tf.reduce_mean` takes the average over these sums.

Train the Model

Now that we have defined our model and training loss function, it is straightforward to train using TensorFlow. Because TensorFlow knows the entire computation graph, it can use automatic differentiation to find the gradients of the loss with respect to each of the variables. TensorFlow has a variety of built-in optimization algorithms. For this example, we will use steepest gradient descent, with a step length of 0.5, to descend the cross entropy.

```
train_step = tf.train.GradientDescentOptimizer(0.5).minimize(cross_entropy)
```

What TensorFlow actually did in that single line was to add new operations to the computation graph. These operations included ones to compute gradients, compute parameter update steps, and apply update steps to the parameters.

The returned operation `train_step`, when run, will apply the gradient descent updates to the parameters. Training the model can therefore be accomplished by repeatedly running `train_step`.

```
for i in range(1000):
  batch = mnist.train.next_batch(100)
  train_step.run(feed_dict={x: batch[0], y_: batch[1]})
```

We load 100 training examples in each training iteration. We then run the `train_step` operation, using `feed_dict` to replace the `placeholder` tensors `x` and `y_` with the training examples. Note that you can replace any tensor in your computation graph using `feed_dict` -- it's not restricted to just `placeholder`s.

Evaluate the Model

How well did our model do?

First we'll figure out where we predicted the correct label. `tf.argmax` is an extremely useful function which gives you the index of the highest entry in a tensor along some axis. For example, `tf.argmax(y,1)` is the label our model thinks is most likely for each input, while `tf.argmax(y_,1)` is the true label. We can use `tf.equal` to check if our prediction matches the truth.

```
correct_prediction = tf.equal(tf.argmax(y,1), tf.argmax(y_,1))
```

That gives us a list of booleans. To determine what fraction are correct, we cast to floating point numbers and then take the mean. For example, `[True, False, True, True]` would become `[1,0,1,1]` which would become `0.75`.

```
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
```

Finally, we can evaluate our accuracy on the test data. This should be about 92% correct.

```
print(accuracy.eval(feed_dict={x: mnist.test.images, y_: mnist.test.labels}))
```

Build a Multilayer Convolutional Network

Getting 92% accuracy on MNIST is bad. It's almost embarrassingly bad. In this section, we'll fix that, jumping from a very simple model to something moderately sophisticated: a small convolutional neural network. This will get us to around 99.2% accuracy -- not state of the art, but respectable.

Weight Initialization

To create this model, we're going to need to create a lot of weights and biases. One should generally initialize weights with a small amount of noise for symmetry breaking, and to prevent 0 gradients. Since we're using ReLU neurons, it is also good practice to initialize them with a slightly positive initial bias to avoid "dead neurons". Instead of doing this repeatedly while we build the model, let's create two handy functions to do it for us.

```
def weight_variable(shape):
  initial = tf.truncated_normal(shape, stddev=0.1)
  return tf.Variable(initial)

def bias_variable(shape):
  initial = tf.constant(0.1, shape=shape)
  return tf.Variable(initial)
```

Convolution and Pooling

TensorFlow also gives us a lot of flexibility in convolution and pooling operations. How do we handle the boundaries? What is our stride size? In this example, we're always going to choose the vanilla version. Our convolutions uses a stride of one and are zero padded so that the output is the same size as the input. Our pooling is plain old max pooling over 2x2 blocks. To keep our code cleaner, let's also abstract those operations into functions.

```
def conv2d(x, W):
  return tf.nn.conv2d(x, W, strides=[1, 1, 1, 1], padding='SAME')

def max_pool_2x2(x):
  return tf.nn.max_pool(x, ksize=[1, 2, 2, 1],
                        strides=[1, 2, 2, 1], padding='SAME')
```

First Convolutional Layer

We can now implement our first layer. It will consist of convolution, followed by max pooling. The convolutional will compute 32 features for each 5x5 patch. Its weight tensor will have a shape of `[5, 5, 1, 32]`. The first two dimensions are the patch size, the next is the number of input channels, and the last is the number of output channels. We will also have a bias vector with a component for each output channel.

```
W_conv1 = weight_variable([5, 5, 1, 32])
b_conv1 = bias_variable([32])
```

To apply the layer, we first reshape `x` to a 4d tensor, with the second and third dimensions corresponding to image width and height, and the final dimension corresponding to the number of color channels.

```
x_image = tf.reshape(x, [-1,28,28,1])
```

We then convolve `x_image` with the weight tensor, add the bias, apply the ReLU function, and finally max pool.

```
h_conv1 = tf.nn.relu(conv2d(x_image, W_conv1) + b_conv1)
h_pool1 = max_pool_2x2(h_conv1)
```

Second Convolutional Layer

In order to build a deep network, we stack several layers of this type. The second layer will have 64 features for each 5x5 patch.

```
W_conv2 = weight_variable([5, 5, 32, 64])
b_conv2 = bias_variable([64])

h_conv2 = tf.nn.relu(conv2d(h_pool1, W_conv2) + b_conv2)
h_pool2 = max_pool_2x2(h_conv2)
```

Densely Connected Layer

Now that the image size has been reduced to 7x7, we add a fully-connected layer with 1024 neurons to allow processing on the entire image. We reshape the tensor from the pooling layer into a batch of vectors, multiply by a weight matrix, add a bias, and apply a ReLU.

```
W_fc1 = weight_variable([7 * 7 * 64, 1024])
b_fc1 = bias_variable([1024])

h_pool2_flat = tf.reshape(h_pool2, [-1, 7*7*64])
h_fc1 = tf.nn.relu(tf.matmul(h_pool2_flat, W_fc1) + b_fc1)
```

Dropout

To reduce overfitting, we will apply dropout before the readout layer. We create a `placeholder` for the probability that a neuron's output is kept during dropout. This allows us to turn dropout on during training, and turn it off during testing. TensorFlow's `tf.nn.dropout` op automatically handles scaling neuron outputs in addition to masking them, so dropout just works without any additional scaling.[1]

```
keep_prob = tf.placeholder(tf.float32)
h_fc1_drop = tf.nn.dropout(h_fc1, keep_prob)
```

Readout Layer

Finally, we add a layer, just like for the one layer softmax regression above.

```
W_fc2 = weight_variable([1024, 10])
b_fc2 = bias_variable([10])

y_conv = tf.matmul(h_fc1_drop, W_fc2) + b_fc2
```

Train and Evaluate the Model

How well does this model do? To train and evaluate it we will use code that is nearly identical to that for the simple one layer SoftMax network above.

The differences are that:

- We will replace the steepest gradient descent optimizer with the more sophisticated ADAM optimizer.
- We will include the additional parameter `keep_prob` in `feed_dict` to control the dropout rate.
- We will add logging to every 100th iteration in the training process.

Feel free to go ahead and run this code, but it does 20,000 training iterations and may take a while (possibly up to half an hour), depending on your processor.

```
cross_entropy = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(y_conv, y_
train_step = tf.train.AdamOptimizer(1e-4).minimize(cross_entropy)
correct_prediction = tf.equal(tf.argmax(y_conv,1), tf.argmax(y_,1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
sess.run(tf.initialize_all_variables())
for i in range(20000):
  batch = mnist.train.next_batch(50)
  if i%100 == 0:
    train_accuracy = accuracy.eval(feed_dict={
        x:batch[0], y_: batch[1], keep_prob: 1.0})
    print("step %d, training accuracy %g"%(i, train_accuracy))
  train_step.run(feed_dict={x: batch[0], y_: batch[1], keep_prob: 0.5})

print("test accuracy %g"%accuracy.eval(feed_dict={
    x: mnist.test.images, y_: mnist.test.labels, keep_prob: 1.0}))
```

The final test set accuracy after running this code should be approximately 99.2%.

We have learned how to quickly and easily build, train, and evaluate a fairly sophisticated deep learning model using TensorFlow.

1: For this small convolutional network, performance is actually nearly identical with and without dropout. Dropout is often very effective at reducing overfitting, but it is most useful when training very large neural networks. ↩