Training a DNN Using Plain TensorFlow

If you want more control over the architecture of the network, you may prefer to use TensorFlow's lower-level Python API (introduced in Chapter 9). In this section we will build the same model as before using this API, and we will implement Mini-batch Gradient Descent to train it on the MNIST dataset. The first step is the construction phase, building the TensorFlow graph. The second step is the execution phase, where you actually run the graph to train the model.

Construction Phase

Let's start. First we need to import the `tensorflow` library. Then we must specify the number of inputs and outputs, and set the number of hidden neurons in each layer:

```
import tensorflow as tf

n_inputs = 28*28  # MNIST
n_hidden1 = 300
n_hidden2 = 100
n_outputs = 10
```

Next, just like you did in Chapter 9, you can use placeholder nodes to represent the training data and targets. The shape of `X` is only partially defined. We know that it will be a 2D tensor (i.e., a matrix), with instances along the first dimension and features along the second dimension, and we know that the number of features is going to be 28 x 28 (one feature per pixel), but we don't know yet how many instances each training batch will contain. So the shape of `X` is `(None, n_inputs)`. Similarly, we know that `y` will be a 1D tensor with one entry per instance, but again we don't know the size of the training batch at this point, so the shape is `(None)`.

```
X = tf.placeholder(tf.float32, shape=(None, n_inputs), name="X")
y = tf.placeholder(tf.int64, shape=(None), name="y")
```

Now let's create the actual neural network. The placeholder `X` will act as the input layer; during the execution phase, it will be replaced with one training batch at a time (note that all the instances in a training batch will be processed simultaneously by the neural network). Now you need to create the two hidden layers and the output layer. The two hidden layers are almost identical: they differ only by the inputs they are connected to and by the number of neurons they contain. The output layer is also very similar, but it uses a softmax activation function instead of a ReLU activation function. So let's create a `neuron_layer()` function that we will use to create one layer at a time. It will need parameters to specify the inputs, the number of neurons, the activation function, and the name of the layer:

```
def neuron_layer(X, n_neurons, name, activation=None):
    with tf.name_scope(name):
        n_inputs = int(X.get_shape()[1])
```

```
    stddev = 2 / np.sqrt(n_inputs)
    init = tf.truncated_normal((n_inputs, n_neurons), stddev=stddev)
    W = tf.Variable(init, name="weights")
    b = tf.Variable(tf.zeros([n_neurons]), name="biases")
    z = tf.matmul(X, W) + b
    if activation=="relu":
        return tf.nn.relu(z)
    else:
        return z
```

Let's go through this code line by line:

1. First we create a name scope using the name of the layer: it will contain all the computation nodes for this neuron layer. This is optional, but the graph will look much nicer in TensorBoard if its nodes are well organized.
2. Next, we get the number of inputs by looking up the input matrix's shape and getting the size of the second dimension (the first dimension is for instances).
3. The next three lines create a `W` variable that will hold the weights matrix. It will be a 2D tensor containing all the connection weights between each input and each neuron; hence, its shape will be `(n_inputs, n_neurons)`. It will be initialized randomly, using a truncated[10] normal (Gaussian) distribution with a standard deviation of $2/\sqrt{n_{inputs}}$. Using this specific standard deviation helps the algorithm converge much faster (we will discuss this further in Chapter 11; it is one of those small tweaks to neural networks that have had a tremendous impact on their efficiency). It is important to initialize connection weights randomly for all hidden layers to avoid any symmetries that the Gradient Descent algorithm would be unable to break.[11]
4. The next line creates a `b` variable for biases, initialized to 0 (no symmetry issue in this case), with one bias parameter per neuron.
5. Then we create a subgraph to compute $\mathbf{z} = \mathbf{X} \cdot \mathbf{W} + \mathbf{b}$. This vectorized implementation will efficiently compute the weighted sums of the inputs plus the bias term for each and every neuron in the layer, for all the instances in the batch in just one shot.
6. Finally, if the `activation` parameter is set to `"relu"`, the code returns `relu(z)` (i.e., max (0, $\mathbf{z}$)), or else it just returns `z`.

10 Using a truncated normal distribution rather than a regular normal distribution ensures that there won't be any large weights, which could slow down training.

11 For example, if you set all the weights to 0, then all neurons will output 0, and the error gradient will be the same for all neurons in a given hidden layer. The Gradient Descent step will then update all the weights in exactly the same way in each layer, so they will all remain equal. In other words, despite having hundreds of neurons per layer, your model will act as if there were only one neuron per layer. It is not going to fly.

Okay, so now you have a nice function to create a neuron layer. Let's use it to create the deep neural network! The first hidden layer takes `X` as its input. The second takes the output of the first hidden layer as its input. And finally, the output layer takes the output of the second hidden layer as its input.

```
with tf.name_scope("dnn"):
    hidden1 = neuron_layer(X, n_hidden1, "hidden1", activation="relu")
    hidden2 = neuron_layer(hidden1, n_hidden2, "hidden2", activation="relu")
    logits = neuron_layer(hidden2, n_outputs, "outputs")
```

Notice that once again we used a name scope for clarity. Also note that `logits` is the output of the neural network *before* going through the softmax activation function: for optimization reasons, we will handle the softmax computation later.

As you might expect, TensorFlow comes with many handy functions to create standard neural network layers, so there's often no need to define your own `neuron_layer()` function like we just did. For example, TensorFlow's `fully_connected()` function creates a fully connected layer, where all the inputs are connected to all the neurons in the layer. It takes care of creating the `weights` and `biases` variables, with the proper initialization strategy, and it uses the ReLU activation function by default (we can change this using the `activation_fn` argument). As we will see in Chapter 11, it also supports regularization and normalization parameters. Let's tweak the preceding code to use the `fully_connected()` function instead of our `neuron_layer()` function. Simply import the function and replace the `dnn` construction section with the following code:

```
from tensorflow.contrib.layers import fully_connected

with tf.name_scope("dnn"):
    hidden1 = fully_connected(X, n_hidden1, scope="hidden1")
    hidden2 = fully_connected(hidden1, n_hidden2, scope="hidden2")
    logits = fully_connected(hidden2, n_outputs, scope="outputs",
                             activation_fn=None)
```

The `tensorflow.contrib` package contains many useful functions, but it is a place for experimental code that has not yet graduated to be part of the main TensorFlow API. So the `fully_connected()` function (and any other `contrib` code) may change or move in the future.

Now that we have the neural network model ready to go, we need to define the cost function that we will use to train it. Just as we did for Softmax Regression in Chapter 4, we will use cross entropy. As we discussed earlier, cross entropy will penalize models that estimate a low probability for the target class. TensorFlow provides several functions to compute cross entropy. We will use `sparse_softmax_cross_entropy_with_logits()`: it computes the cross entropy based on the

"logits" (i.e., the output of the network *before* going through the softmax activation function), and it expects labels in the form of integers ranging from 0 to the number of classes minus 1 (in our case, from 0 to 9). This will give us a 1D tensor containing the cross entropy for each instance. We can then use TensorFlow's `reduce_mean()` function to compute the mean cross entropy over all instances.

```
with tf.name_scope("loss"):
    xentropy = tf.nn.sparse_softmax_cross_entropy_with_logits(
                   labels=y, logits=logits)
    loss = tf.reduce_mean(xentropy, name="loss")
```

The `sparse_softmax_cross_entropy_with_logits()` function is equivalent to applying the softmax activation function and then computing the cross entropy, but it is more efficient, and it properly takes care of corner cases like logits equal to 0. This is why we did not apply the softmax activation function earlier. There is also another function called `softmax_cross_entropy_with_logits()`, which takes labels in the form of one-hot vectors (instead of ints from 0 to the number of classes minus 1).

We have the neural network model, we have the cost function, and now we need to define a `GradientDescentOptimizer` that will tweak the model parameters to minimize the cost function. Nothing new; it's just like we did in Chapter 9:

```
learning_rate = 0.01

with tf.name_scope("train"):
    optimizer = tf.train.GradientDescentOptimizer(learning_rate)
    training_op = optimizer.minimize(loss)
```

The last important step in the construction phase is to specify how to evaluate the model. We will simply use accuracy as our performance measure. First, for each instance, determine if the neural network's prediction is correct by checking whether or not the highest logit corresponds to the target class. For this you can use the `in_top_k()` function. This returns a 1D tensor full of boolean values, so we need to cast these booleans to floats and then compute the average. This will give us the network's overall accuracy.

```
with tf.name_scope("eval"):
    correct = tf.nn.in_top_k(logits, y, 1)
    accuracy = tf.reduce_mean(tf.cast(correct, tf.float32))
```

And, as usual, we need to create a node to initialize all variables, and we will also create a `Saver` to save our trained model parameters to disk:

```
init = tf.global_variables_initializer()
saver = tf.train.Saver()
```

Phew! This concludes the construction phase. This was fewer than 40 lines of code, but it was pretty intense: we created placeholders for the inputs and the targets, we created a function to build a neuron layer, we used it to create the DNN, we defined the cost function, we created an optimizer, and finally we defined the performance measure. Now on to the execution phase.

Execution Phase

This part is much shorter and simpler. First, let's load MNIST. We could use Scikit-Learn for that as we did in previous chapters, but TensorFlow offers its own helper that fetches the data, scales it (between 0 and 1), shuffles it, and provides a simple function to load one mini-batches a time. So let's use it instead:

```
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("/tmp/data/")
```

Now we define the number of epochs that we want to run, as well as the size of the mini-batches:

```
n_epochs = 400
batch_size = 50
```

And now we can train the model:

```
with tf.Session() as sess:
    init.run()
    for epoch in range(n_epochs):
        for iteration in range(mnist.train.num_examples // batch_size):
            X_batch, y_batch = mnist.train.next_batch(batch_size)
            sess.run(training_op, feed_dict={X: X_batch, y: y_batch})
        acc_train = accuracy.eval(feed_dict={X: X_batch, y: y_batch})
        acc_test = accuracy.eval(feed_dict={X: mnist.test.images,
                                            y: mnist.test.labels})
        print(epoch, "Train accuracy:", acc_train, "Test accuracy:", acc_test)

    save_path = saver.save(sess, "./my_model_final.ckpt")
```

This code opens a TensorFlow session, and it runs the `init` node that initializes all the variables. Then it runs the main training loop: at each epoch, the code iterates through a number of mini-batches that corresponds to the training set size. Each mini-batch is fetched via the `next_batch()` method, and then the code simply runs the training operation, feeding it the current mini-batch input data and targets. Next, at the end of each epoch, the code evaluates the model on the last mini-batch and on the full training set, and it prints out the result. Finally, the model parameters are saved to disk.