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.

Using the Neural Network

Now that the neural network is trained, you can use it to make predictions. To do that, you can reuse the same construction phase, but change the execution phase like this:

```
with tf.Session() as sess:
    saver.restore(sess, "./my_model_final.ckpt")
    X_new_scaled = [...]  # some new images (scaled from 0 to 1)
    Z = logits.eval(feed_dict={X: X_new_scaled})
    y_pred = np.argmax(Z, axis=1)
```

First the code loads the model parameters from disk. Then it loads some new images that you want to classify. Remember to apply the same feature scaling as for the training data (in this case, scale it from 0 to 1). Then the code evaluates the `logits` node. If you wanted to know all the estimated class probabilities, you would need to apply the `softmax()` function to the logits, but if you just want to predict a class, you can simply pick the class that has the highest logit value (using the `argmax()` function does the trick).

Fine-Tuning Neural Network Hyperparameters

The flexibility of neural networks is also one of their main drawbacks: there are many hyperparameters to tweak. Not only can you use any imaginable *network topology* (how neurons are interconnected), but even in a simple MLP you can change the number of layers, the number of neurons per layer, the type of activation function to use in each layer, the weight initialization logic, and much more. How do you know what combination of hyperparameters is the best for your task?

Of course, you can use grid search with cross-validation to find the right hyperparameters, like you did in previous chapters, but since there are many hyperparameters to tune, and since training a neural network on a large dataset takes a lot of time, you will only be able to explore a tiny part of the hyperparameter space in a reasonable amount of time. It is much better to use randomized search, as we discussed in Chapter 2. Another option is to use a tool such as Oscar, which implements more complex algorithms to help you find a good set of hyperparameters quickly.

It helps to have an idea of what values are reasonable for each hyperparameter, so you can restrict the search space. Let's start with the number of hidden layers.

Number of Hidden Layers

For many problems, you can just begin with a single hidden layer and you will get reasonable results. It has actually been shown that an MLP with just one hidden layer can model even the most complex functions provided it has enough neurons. For a long time, these facts convinced researchers that there was no need to investigate any