Biological neurons seem to implement a roughly sigmoid (S-shaped) activation function, so researchers stuck to sigmoid functions for a very long time. But it turns out that the ReLU activation function generally works better in ANNs. This is one of the cases where the biological analogy was misleading.

Training an MLP with TensorFlow's High-Level API

The simplest way to train an MLP with TensorFlow is to use the high-level API TF.Learn, which is quite similar to Scikit-Learn's API. The `DNNClassifier` class makes it trivial to train a deep neural network with any number of hidden layers, and a softmax output layer to output estimated class probabilities. For example, the following code trains a DNN for classification with two hidden layers (one with 300 neurons, and the other with 100 neurons) and a softmax output layer with 10 neurons:

```
import tensorflow as tf

feature_columns = tf.contrib.learn.infer_real_valued_columns_from_input(X_train)
dnn_clf = tf.contrib.learn.DNNClassifier(hidden_units=[300, 100], n_classes=10,
                                         feature_columns=feature_columns)
dnn_clf.fit(x=X_train, y=y_train, batch_size=50, steps=40000)
```

If you run this code on the MNIST dataset (after scaling it, e.g., by using Scikit-Learn's `StandardScaler`), you may actually get a model that achieves over 98.1% accuracy on the test set! That's better than the best model we trained in Chapter 3:

```
>>> from sklearn.metrics import accuracy_score
>>> y_pred = list(dnn_clf.predict(X_test))
>>> accuracy_score(y_test, y_pred)
0.98180000000000001
```

The TF.Learn library also provides some convenience functions to evaluate models:

```
>>> dnn_clf.evaluate(X_test, y_test)
{'accuracy': 0.98180002, 'global_step': 40000, 'loss': 0.073678359}
```

Under the hood, the `DNNClassifier` class creates all the neuron layers, based on the ReLU activation function (we can change this by setting the `activation_fn` hyperparameter). The output layer relies on the softmax function, and the cost function is cross entropy (introduced in Chapter 4).

The TF.Learn API is still quite new, so some of the names and functions used in these examples may evolve a bit by the time you read this book. However, the general ideas should not change.

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])
```