The RMSE is a measure of the average difference between predicted values and true values. In Figure 3-13 we plot predicted values and true labels as two separate functions using datapoints x as our x-axis. Note that the line learned isn't the true function! The RMSE is relatively high and diagnoses the error, unlike the R^2, which didn't pick up on this error.

What happened on this system? Why didn't TensorFlow learn the correct function despite being trained to convergence? This example provides a good illustration of one of the weaknesses of gradient descent algorithms. There is no guarantee of finding the true solution! The gradient descent algorithm can get trapped in *local minima*. That is, it can find solutions that look good, but are not in fact the lowest minima of the loss function $\mathscr{L}$.

Why use gradient descent at all then? For simple systems, it is indeed often better to avoid gradient descent and use other algorithms that have stronger performance guarantees. However, on complicated systems, such as those we will show you in later chapters, there do not yet exist alternative algorithms that perform better than gradient descent. We encourage you to remember this fact as we proceed further into deep learning.

Logistic Regression in TensorFlow

In this section, we will define a simple classifier using TensorFlow. It's worth first considering what the equation is for a classifier. The mathematical trick that is commonly used is exploiting the sigmoid function. The sigmoid, plotted in Figure 3-14, commonly denoted by σ, is a function from the real numbers $\mathbb{R}$ to (0, 1). This property is convenient since we can interpret the output of a sigmoid as probability of an event happening. (The trick of converting discrete events into continuous values is a recurring theme in machine learning.)

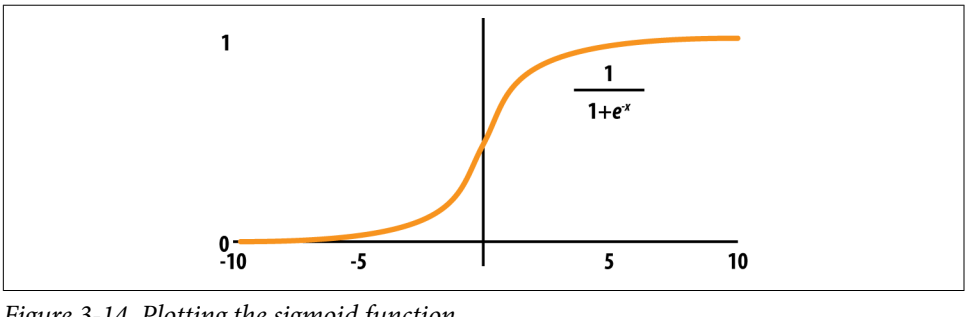

Figure 3-14. Plotting the sigmoid function.

The equations for predicting the probabilities of a discrete 0/1 variable follow. These equations define a simple logistic regression model:

$$y_0 = \sigma(wx + b)$$

$$y_1 = 1 - \sigma(wx + b)$$

TensorFlow provides utility functions to compute the cross-entropy loss for sigmoidal values. The simplest of these functions is `tf.nn.sigmoid_cross_entropy_with_logits`. (A logit is the inverse of the sigmoid. In practice, this simply means passing the argument to the sigmoid, $wx + b$, directly to TensorFlow instead of the sigmoidal value $\sigma(wx + b)$ itself). We recommend using TensorFlow's implementation instead of manually defining the cross-entropy, since there are tricky numerical issues that arise when computing the cross-entropy loss.

Example 3-14 defines a simple logistic regression model in TensorFlow.

Example 3-14. Defining a simple logistic regression model

```
# Generate tensorflow graph
with tf.name_scope("placeholders"):
  # Note that our datapoints x are 2-dimensional.
  x = tf.placeholder(tf.float32, (N, 2))
  y = tf.placeholder(tf.float32, (N,))
with tf.name_scope("weights"):
  W = tf.Variable(tf.random_normal((2, 1)))
  b = tf.Variable(tf.random_normal((1,)))
with tf.name_scope("prediction"):
  y_logit = tf.squeeze(tf.matmul(x, W) + b)
  # the sigmoid gives the class probability of 1
  y_one_prob = tf.sigmoid(y_logit)
  # Rounding P(y=1) will give the correct prediction.
  y_pred = tf.round(y_one_prob)

with tf.name_scope("loss"):
  # Compute the cross-entropy term for each datapoint
  entropy = tf.nn.sigmoid_cross_entropy_with_logits(logits=y_logit, labels=y)
  # Sum all contributions
  l = tf.reduce_sum(entropy)
with tf.name_scope("optim"):
  train_op = tf.train.AdamOptimizer(.01).minimize(l)

  train_writer = tf.summary.FileWriter('/tmp/logistic-train', tf.get_default_graph())
```

The training code for this model in Example 3-15 is identical to that for the linear regression model.

Example 3-15. Training a logistic regression model

```
n_steps = 1000
with tf.Session() as sess:
  sess.run(tf.global_variables_initializer())
  # Train model
  for i in range(n_steps):
    feed_dict = {x: x_np, y: y_np}
    _, summary, loss = sess.run([train_op, merged, l], feed_dict=feed_dict)
    print("loss: %f" % loss)
    train_writer.add_summary(summary, i)
```

Visualizing logistic regression models with TensorBoard

As before, you can use TensorBoard to visualize the model. Start by visualizing the loss function as shown in Figure 3-15. Note that as before, the loss function follows a neat pattern. There is a steep drop in the loss followed by a gradual smoothening.

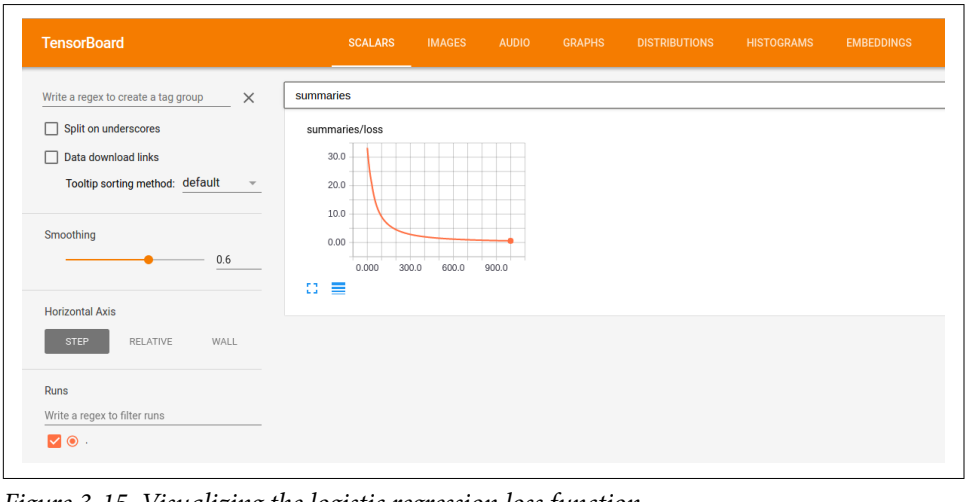

Figure 3-15. Visualizing the logistic regression loss function.

You can also view the TensorFlow graph in TensorBoard. Since the scoping structure was similar to that used for linear regression, the simplified graph doesn't display much differently, as shown in Figure 3-16.

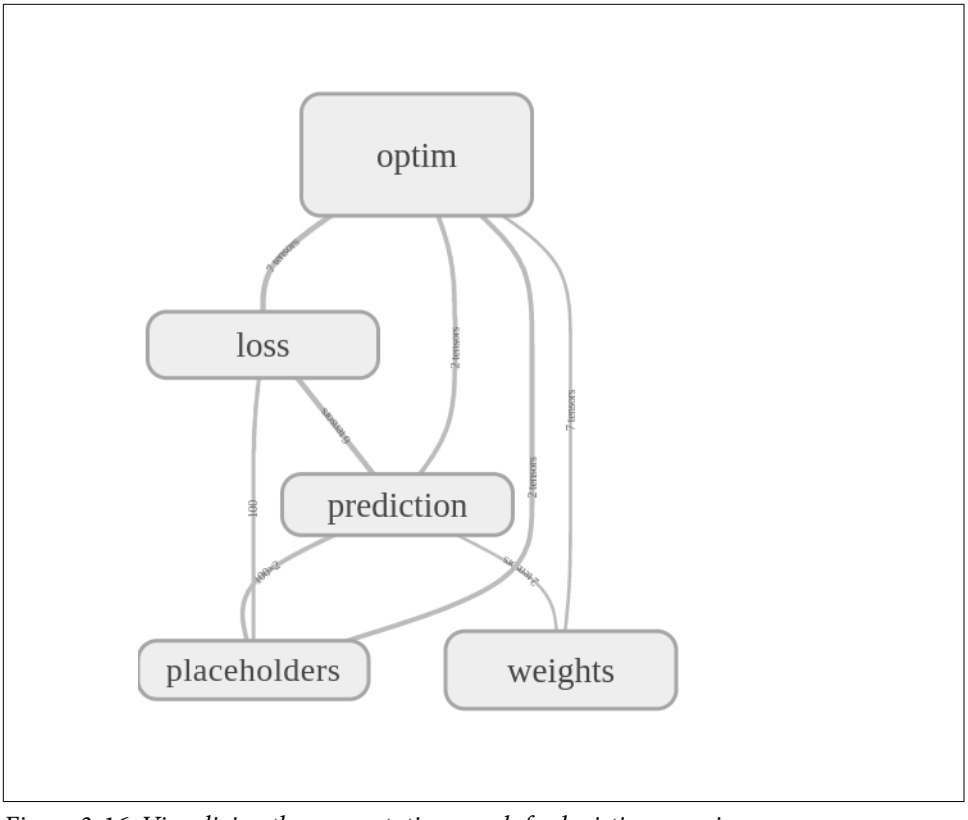

Figure 3-16. Visualizing the computation graph for logistic regression.

However, if you expand the nodes in this grouped graph, as in Figure 3-17, you will find that the underlying computational graph is different. In particular, the loss function is quite different from that used for linear regression (as it should be).

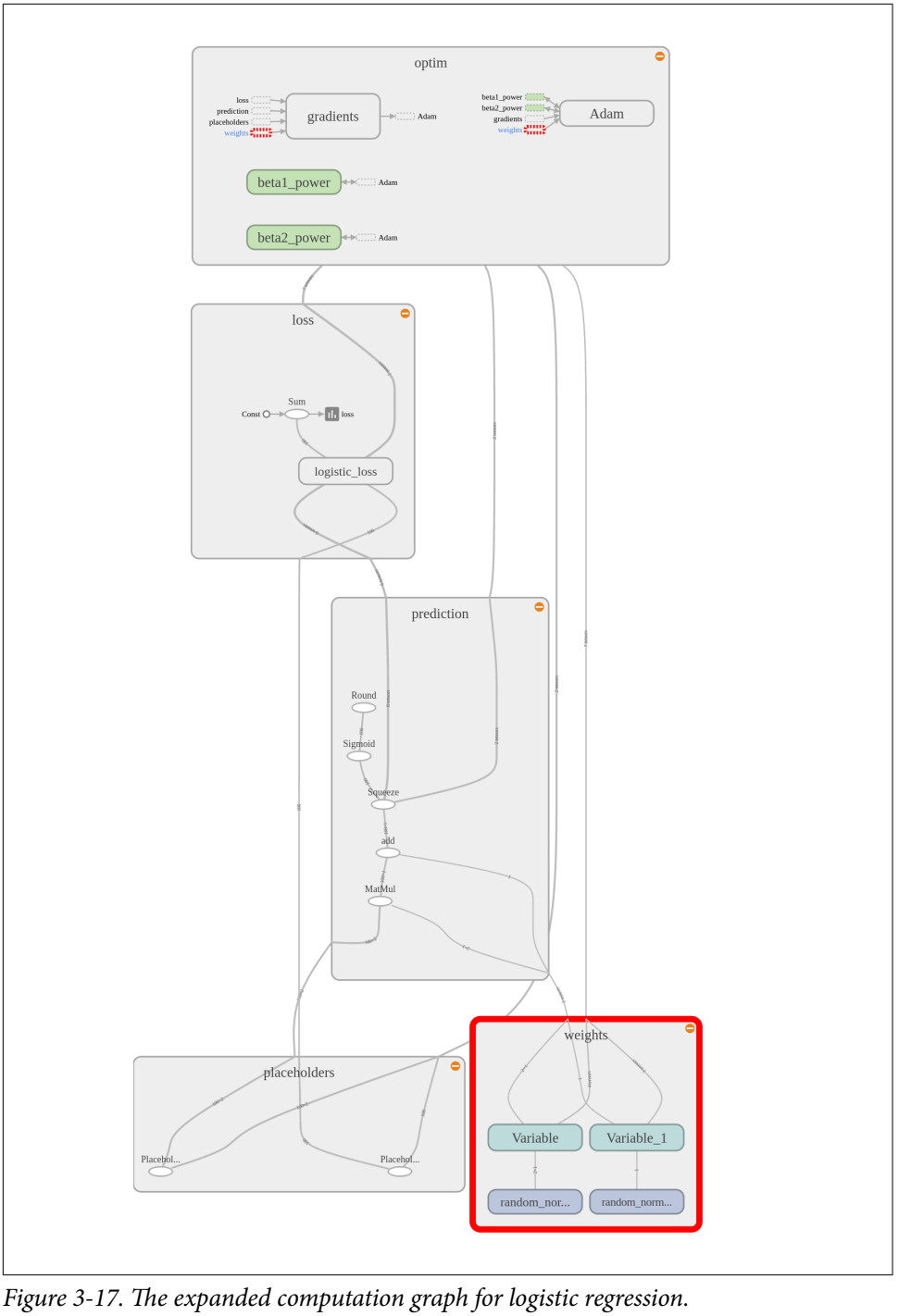

Figure 3-17. The expanded computation graph for logistic regression.

Metrics for evaluating classification models

Now that you have trained a classification model for logistic regression, you need to learn about metrics suitable for evaluating classification models. Although the equations for logistic regression are more complicated than they are for linear regression, the basic evaluation metrics are simpler. The classification accuracy simply checks for the fraction of datapoints that are classified correctly by the learned model. In fact, with a little more effort, it is possible to back out the *separating line* learned by the logistic regression model. This line displays the cutoff boundary the model has learned to separate positive and negative examples. (We leave the derivation of this line from the logistic regression equations as an exercise for the interested reader. The solution is in the code for this section.)

We display the learned classes and the separating line in Figure 3-18. Note that the line neatly separates the positive and negative examples and has perfect accuracy (1.0). This result raises an interesting point. Regression is often a harder problem to solve than classification. There are many possible lines that would neatly separate the datapoints in Figure 3-18, but only one that would have perfectly matched the data for the linear regression.

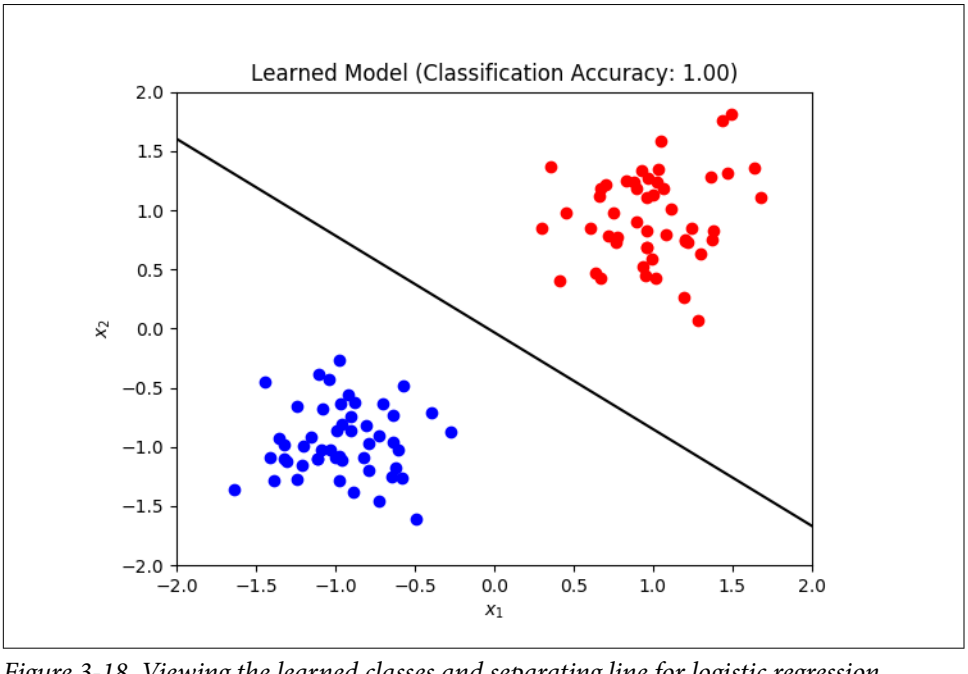

Figure 3-18. Viewing the learned classes and separating line for logistic regression.

Review

In this chapter, we've shown you how to build and train some simple learning systems in TensorFlow. We started by reviewing some foundational mathematical concepts including loss functions and gradient descent. We then introduced you to some new TensorFlow concepts such as placeholders, scopes, and TensorBoard. We ended the chapter with case studies that trained linear and logistic regression systems on toy datasets. We covered a lot of material in this chapter, and it's OK if you haven't yet internalized everything. The foundational material introduced here will be used throughout the remainder of this book.

In Chapter 4, we will introduce you to your first deep learning model and to fully connected networks, and will show you how to define and train fully connected networks in TensorFlow. In following chapters, we will explore more complicated deep networks, but all of these architectures will use the same fundamental learning principles introduced in this chapter.