By default a `Saver` saves and restores all variables under their own name, but if you need more control, you can specify which variables to save or restore, and what names to use. For example, the following `Saver` will save or restore only the `theta` variable under the name `weights`:

```
saver = tf.train.Saver({"weights": theta})
```

Visualizing the Graph and Training Curves Using TensorBoard

So now we have a computation graph that trains a Linear Regression model using Mini-batch Gradient Descent, and we are saving checkpoints at regular intervals. Sounds sophisticated, doesn't it? However, we are still relying on the `print()` function to visualize progress during training. There is a better way: enter TensorBoard. If you feed it some training stats, it will display nice interactive visualizations of these stats in your web browser (e.g., learning curves). You can also provide it the graph's definition and it will give you a great interface to browse through it. This is very useful to identify errors in the graph, to find bottlenecks, and so on.

The first step is to tweak your program a bit so it writes the graph definition and some training stats—for example, the training error (MSE)—to a log directory that TensorBoard will read from. You need to use a different log directory every time you run your program, or else TensorBoard will merge stats from different runs, which will mess up the visualizations. The simplest solution for this is to include a timestamp in the log directory name. Add the following code at the beginning of the program:

```
from datetime import datetime

now = datetime.utcnow().strftime("%Y%m%d%H%M%S")
root_logdir = "tf_logs"
logdir = "{}/run-{}/".format(root_logdir, now)
```

Next, add the following code at the very end of the construction phase:

```
mse_summary = tf.summary.scalar('MSE', mse)
file_writer = tf.summary.FileWriter(logdir, tf.get_default_graph())
```

The first line creates a node in the graph that will evaluate the MSE value and write it to a TensorBoard-compatible binary log string called a *summary*. The second line creates a `FileWriter` that you will use to write summaries to logfiles in the log directory. The first parameter indicates the path of the log directory (in this case something like *tf_logs/run-20160906091959/*, relative to the current directory). The second (optional) parameter is the graph you want to visualize. Upon creation, the `FileWriter` creates the log directory if it does not already exist (and its parent directories if needed), and writes the graph definition in a binary logfile called an *events file*.

Next you need to update the execution phase to evaluate the `mse_summary` node regularly during training (e.g., every 10 mini-batches). This will output a summary that you can then write to the events file using the `file_writer`. Here is the updated code:

```
[...]
for batch_index in range(n_batches):
    X_batch, y_batch = fetch_batch(epoch, batch_index, batch_size)
    if batch_index % 10 == 0:
        summary_str = mse_summary.eval(feed_dict={X: X_batch, y: y_batch})
        step = epoch * n_batches + batch_index
        file_writer.add_summary(summary_str, step)
    sess.run(training_op, feed_dict={X: X_batch, y: y_batch})
[...]
```

Avoid logging training stats at every single training step, as this would significantly slow down training.

Finally, you want to close the `FileWriter` at the end of the program:

```
file_writer.close()
```

Now run this program: it will create the log directory and write an events file in this directory, containing both the graph definition and the MSE values. Open up a shell and go to your working directory, then type **`ls -l tf_logs/run*`** to list the contents of the log directory:

```
$ cd $ML_PATH                # Your ML working directory (e.g., $HOME/ml)
$ ls -l tf_logs/run*
total 40
-rw-r--r-- 1 ageron staff 18620 Sep 6 11:10 events.out.tfevents.1472553182.mymac
```

If you run the program a second time, you should see a second directory in the *tf_logs/* directory:

```
$ ls -l tf_logs/
total 0
drwxr-xr-x  3 ageron  staff  102 Sep  6 10:07 run-20160906091959
drwxr-xr-x  3 ageron  staff  102 Sep  6 10:22 run-20160906092202
```

Great! Now it's time to fire up the TensorBoard server. You need to activate your virtualenv environment if you created one, then start the server by running the `tensorboard` command, pointing it to the root log directory. This starts the TensorBoard web server, listening on port 6006 (which is "goog" written upside down):

```
$ source env/bin/activate
$ tensorboard --logdir tf_logs/
Starting TensorBoard  on port 6006
(You can navigate to http://0.0.0.0:6006)
```

Next open a browser and go to *http://0.0.0.0:6006/* (or *http://localhost:6006/*). Welcome to TensorBoard! In the Events tab you should see MSE on the right. If you click on it, you will see a plot of the MSE during training, for both runs (Figure 9-3). You can check or uncheck the runs you want to see, zoom in or out, hover over the curve to get details, and so on.

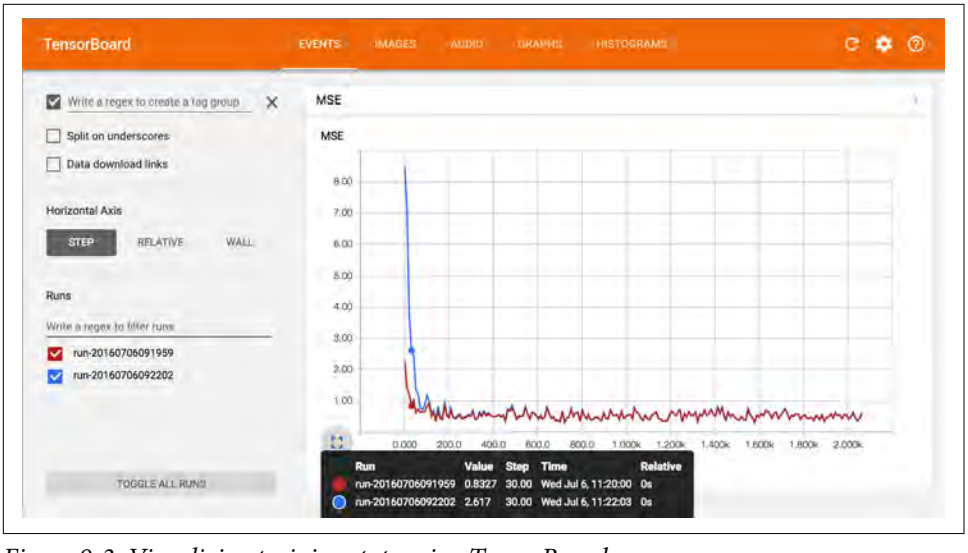

Figure 9-3. Visualizing training stats using TensorBoard

Now click on the Graphs tab. You should see the graph shown in Figure 9-4.

To reduce clutter, the nodes that have many *edges* (i.e., connections to other nodes) are separated out to an auxiliary area on the right (you can move a node back and forth between the main graph and the auxiliary area by right-clicking on it). Some parts of the graph are also collapsed by default. For example, try hovering over the `gradients` node, then click on the ⊕ icon to expand this subgraph. Next, in this subgraph, try expanding the `mse_grad` subgraph.

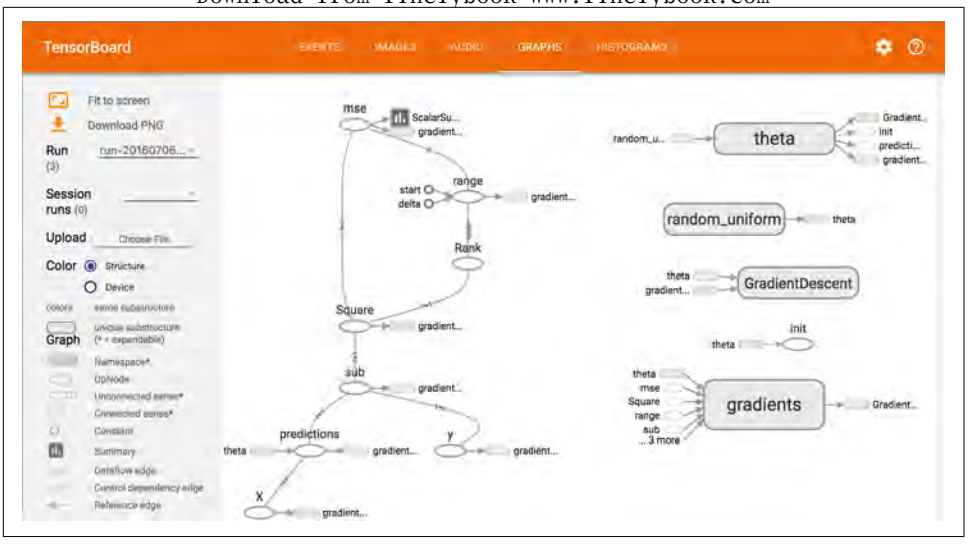

Figure 9-4. Visualizing the graph using TensorBoard

If you want to take a peek at the graph directly within Jupyter, you can use the `show_graph()` function available in the notebook for this chapter. It was originally written by A. Mordvintsev in his great deepdream tutorial notebook. Another option is to install E. Jang's TensorFlow debugger tool which includes a Jupyter extension for graph visualization (and more).

Name Scopes

When dealing with more complex models such as neural networks, the graph can easily become cluttered with thousands of nodes. To avoid this, you can create *name scopes* to group related nodes. For example, let's modify the previous code to define the `error` and `mse` ops within a name scope called `"loss"`:

```
with tf.name_scope("loss") as scope:
    error = y_pred - y
    mse = tf.reduce_mean(tf.square(error), name="mse")
```

The name of each op defined within the scope is now prefixed with `"loss/"`:

```
>>> print(error.op.name)
loss/sub
>>> print(mse.op.name)
loss/mse
```

In TensorBoard, the `mse` and `error` nodes now appear inside the `loss` namespace, which appears collapsed by default (Figure 9-5).