```
best_theta = theta.eval()
```

We don't need to pass the value of `X` and `y` when evaluating `theta` since it does not depend on either of them.

Saving and Restoring Models

Once you have trained your model, you should save its parameters to disk so you can come back to it whenever you want, use it in another program, compare it to other models, and so on. Moreover, you probably want to save checkpoints at regular intervals during training so that if your computer crashes during training you can continue from the last checkpoint rather than start over from scratch.

TensorFlow makes saving and restoring a model very easy. Just create a `Saver` node at the end of the construction phase (after all variable nodes are created); then, in the execution phase, just call its `save()` method whenever you want to save the model, passing it the session and path of the checkpoint file:

```
[...]
theta = tf.Variable(tf.random_uniform([n + 1, 1], -1.0, 1.0), name="theta")
[...]
init = tf.global_variables_initializer()
saver = tf.train.Saver()

with tf.Session() as sess:
    sess.run(init)

    for epoch in range(n_epochs):
        if epoch % 100 == 0:  # checkpoint every 100 epochs
            save_path = saver.save(sess, "/tmp/my_model.ckpt")

        sess.run(training_op)

    best_theta = theta.eval()
    save_path = saver.save(sess, "/tmp/my_model_final.ckpt")
```

Restoring a model is just as easy: you create a `Saver` at the end of the construction phase just like before, but then at the beginning of the execution phase, instead of initializing the variables using the `init` node, you call the `restore()` method of the `Saver` object:

```
with tf.Session() as sess:
    saver.restore(sess, "/tmp/my_model_final.ckpt")
    [...]
```

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*.