

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).

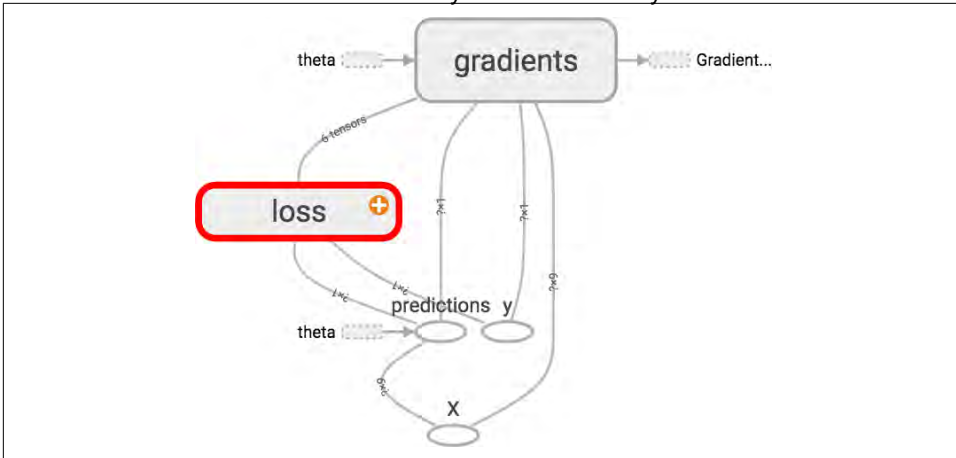


Figure 9-5. A collapsed namespace in TensorBoard

## Modularity

Suppose you want to create a graph that adds the output of two *rectified linear units* (ReLU). A ReLU computes a linear function of the inputs, and outputs the result if it is positive, and 0 otherwise, as shown in [Equation 9-1](#).

*Equation 9-1. Rectified linear unit*

$$h_{\mathbf{w},b}(\mathbf{X}) = \max(\mathbf{X} \cdot \mathbf{w} + b, 0)$$

The following code does the job, but it's quite repetitive:

```
n_features = 3
X = tf.placeholder(tf.float32, shape=(None, n_features), name="X")

w1 = tf.Variable(tf.random_normal((n_features, 1)), name="weights1")
w2 = tf.Variable(tf.random_normal((n_features, 1)), name="weights2")
b1 = tf.Variable(0.0, name="bias1")
b2 = tf.Variable(0.0, name="bias2")

z1 = tf.add(tf.matmul(X, w1), b1, name="z1")
z2 = tf.add(tf.matmul(X, w2), b2, name="z2")

relu1 = tf.maximum(z1, 0., name="relu1")
relu2 = tf.maximum(z2, 0., name="relu2")

output = tf.add(relu1, relu2, name="output")
```

Such repetitive code is hard to maintain and error-prone (in fact, this code contains a cut-and-paste error; did you spot it?). It would become even worse if you wanted to