```
optimizer = tf.train.AdamOptimizer(learning_rate)

with tf.name_scope("phase1"):
    phase1_outputs = tf.matmul(hidden1, weights4) + biases4
    phase1_reconstruction_loss = tf.reduce_mean(tf.square(phase1_outputs - X))
    phase1_reg_loss = regularizer(weights1) + regularizer(weights4)
    phase1_loss = phase1_reconstruction_loss + phase1_reg_loss
    phase1_training_op = optimizer.minimize(phase1_loss)

with tf.name_scope("phase2"):
    phase2_reconstruction_loss = tf.reduce_mean(tf.square(hidden3 - hidden1))
    phase2_reg_loss = regularizer(weights2) + regularizer(weights3)
    phase2_loss = phase2_reconstruction_loss + phase2_reg_loss
    train_vars = [weights2, biases2, weights3, biases3]
    phase2_training_op = optimizer.minimize(phase2_loss, var_list=train_vars)
```

The first phase is rather straightforward: we just create an output layer that skips hidden layers 2 and 3, then build the training operations to minimize the distance between the outputs and the inputs (plus some regularization).

The second phase just adds the operations needed to minimize the distance between the output of hidden layer 3 and hidden layer 1 (also with some regularization). Most importantly, we provide the list of trainable variables to the `minimize()` method, making sure to leave out `weights1` and `biases1`; this effectively freezes hidden layer 1 during phase 2.

During the execution phase, all you need to do is run the phase 1 training op for a number of epochs, then the phase 2 training op for some more epochs.

Since hidden layer 1 is frozen during phase 2, its output will always be the same for any given training instance. To avoid having to recompute the output of hidden layer 1 at every single epoch, you can compute it for the whole training set at the end of phase 1, then directly feed the cached output of hidden layer 1 during phase 2. This can give you a nice performance boost.

Visualizing the Reconstructions

One way to ensure that an autoencoder is properly trained is to compare the inputs and the outputs. They must be fairly similar, and the differences should be unimportant details. Let's plot two random digits and their reconstructions:

```
n_test_digits = 2
X_test = mnist.test.images[:n_test_digits]

with tf.Session() as sess:
    [...] # Train the Autoencoder
    outputs_val = outputs.eval(feed_dict={X: X_test})
```

```
def plot_image(image, shape=[28, 28]):
    plt.imshow(image.reshape(shape), cmap="Greys", interpolation="nearest")
    plt.axis("off")

for digit_index in range(n_test_digits):
    plt.subplot(n_test_digits, 2, digit_index * 2 + 1)
    plot_image(X_test[digit_index])
    plt.subplot(n_test_digits, 2, digit_index * 2 + 2)
    plot_image(outputs_val[digit_index])
```

Figure 15-6 shows the resulting images.

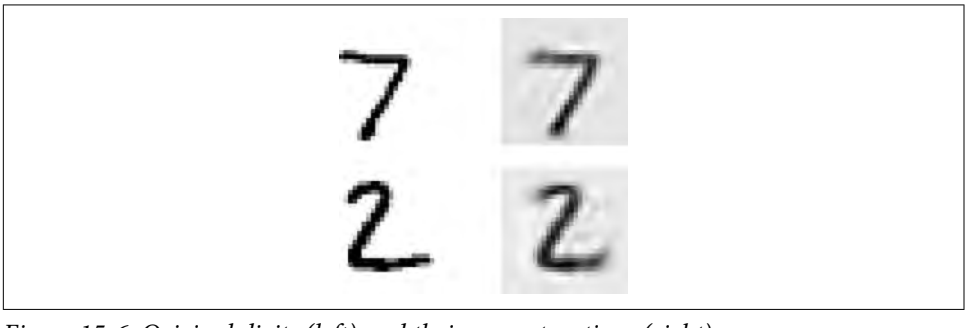

Figure 15-6. Original digits (left) and their reconstructions (right)

Looks close enough. So the autoencoder has properly learned to reproduce its inputs, but has it learned useful features? Let's take a look.

Visualizing Features

Once your autoencoder has learned some features, you may want to take a look at them. There are various techniques for this. Arguably the simplest technique is to consider each neuron in every hidden layer, and find the training instances that activate it the most. This is especially useful for the top hidden layers since they often capture relatively large features that you can easily spot in a group of training instances that contain them. For example, if a neuron strongly activates when it sees a cat in a picture, it will be pretty obvious that the pictures that activate it the most all contain cats. However, for lower layers, this technique does not work so well, as the features are smaller and more abstract, so it's often hard to understand exactly what the neuron is getting all excited about.

Let's look at another technique. For each neuron in the first hidden layer, you can create an image where a pixel's intensity corresponds to the weight of the connection to the given neuron. For example, the following code plots the features learned by five neurons in the first hidden layer:

```
with tf.Session() as sess:
    [...] # train autoencoder
```