- First, `weight3` and `weights4` are not variables, they are respectively the transpose of `weights2` and `weights1` (they are "tied" to them).
- Second, since they are not variables, it's no use regularizing them: we only regularize `weights1` and `weights2`.
- Third, biases are never tied, and never regularized.

Training One Autoencoder at a Time

Rather than training the whole stacked autoencoder in one go like we just did, it is often much faster to train one shallow autoencoder at a time, then stack all of them into a single stacked autoencoder (hence the name), as shown on Figure 15-4. This is especially useful for very deep autoencoders.

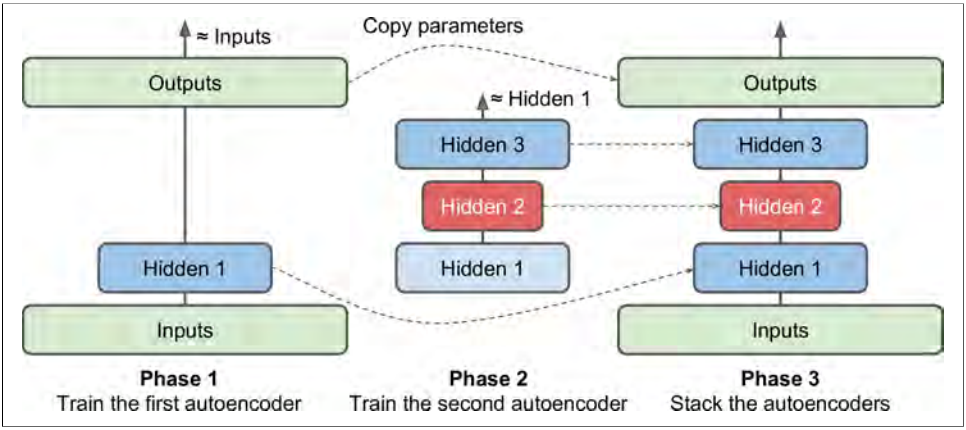

Figure 15-4. Training one autoencoder at a time

During the first phase of training, the first autoencoder learns to reconstruct the inputs. During the second phase, the second autoencoder learns to reconstruct the output of the first autoencoder's hidden layer. Finally, you just build a big sandwich using all these autoencoders, as shown in Figure 15-4 (i.e., you first stack the hidden layers of each autoencoder, then the output layers in reverse order). This gives you the final stacked autoencoder. You could easily train more autoencoders this way, building a very deep stacked autoencoder.

To implement this multiphase training algorithm, the simplest approach is to use a different TensorFlow graph for each phase. After training an autoencoder, you just run the training set through it and capture the output of the hidden layer. This output then serves as the training set for the next autoencoder. Once all autoencoders have been trained this way, you simply copy the weights and biases from each autoencoder and use them to build the stacked autoencoder. Implementing this approach is quite straightforward, so we won't detail it here, but please check out the code in the Jupyter notebooks for an example.

Another approach is to use a single graph containing the whole stacked autoencoder, plus some extra operations to perform each training phase, as shown in Figure 15-5.

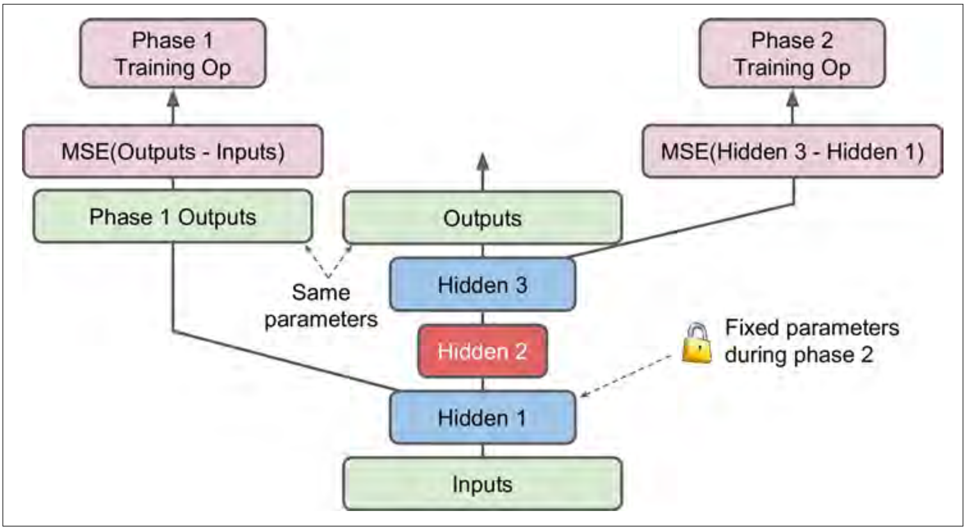

Figure 15-5. A single graph to train a stacked autoencoder

This deserves a bit of explanation:

- The central column in the graph is the full stacked autoencoder. This part can be used after training.
- The left column is the set of operations needed to run the first phase of training. It creates an output layer that bypasses hidden layers 2 and 3. This output layer shares the same weights and biases as the stacked autoencoder's output layer. On top of that are the training operations that will aim at making the output as close as possible to the inputs. Thus, this phase will train the weights and biases for the hidden layer 1 and the output layer (i.e., the first autoencoder).
- The right column in the graph is the set of operations needed to run the second phase of training. It adds the training operation that will aim at making the output of hidden layer 3 as close as possible to the output of hidden layer 1. Note that we must freeze hidden layer 1 while running phase 2. This phase will train the weights and biases for hidden layers 2 and 3 (i.e., the second autoencoder).

The TensorFlow code looks like this:

```
[...] # Build the whole stacked autoencoder normally.
      # In this example, the weights are not tied.
```

```
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})
```