Tying Weights

When an autoencoder is neatly symmetrical, like the one we just built, a common technique is to *tie the weights* of the decoder layers to the weights of the encoder layers. This halves the number of weights in the model, speeding up training and limiting the risk of overfitting. Specifically, if the autoencoder has a total of N layers (not counting the input layer), and $\mathbf{W}_L$ represents the connection weights of the L^{th} layer (e.g., layer 1 is the first hidden layer, layer $\frac{N}{2}$ is the coding layer, and layer N is the output layer), then the decoder layer weights can be defined simply as: $\mathbf{W}_{N-L+1} = \mathbf{W}_L{}^T$ (with $L = 1, 2, \cdots, \frac{N}{2}$).

Unfortunately, implementing tied weights in TensorFlow using the `fully_connected()` function is a bit cumbersome; it's actually easier to just define the layers manually. The code ends up significantly more verbose:

```
activation = tf.nn.elu
regularizer = tf.contrib.layers.l2_regularizer(l2_reg)
initializer = tf.contrib.layers.variance_scaling_initializer()

X = tf.placeholder(tf.float32, shape=[None, n_inputs])

weights1_init = initializer([n_inputs, n_hidden1])
weights2_init = initializer([n_hidden1, n_hidden2])

weights1 = tf.Variable(weights1_init, dtype=tf.float32, name="weights1")
weights2 = tf.Variable(weights2_init, dtype=tf.float32, name="weights2")
weights3 = tf.transpose(weights2, name="weights3")  # tied weights
weights4 = tf.transpose(weights1, name="weights4")  # tied weights

biases1 = tf.Variable(tf.zeros(n_hidden1), name="biases1")
biases2 = tf.Variable(tf.zeros(n_hidden2), name="biases2")
biases3 = tf.Variable(tf.zeros(n_hidden3), name="biases3")
biases4 = tf.Variable(tf.zeros(n_outputs), name="biases4")

hidden1 = activation(tf.matmul(X, weights1) + biases1)
hidden2 = activation(tf.matmul(hidden1, weights2) + biases2)
hidden3 = activation(tf.matmul(hidden2, weights3) + biases3)
outputs = tf.matmul(hidden3, weights4) + biases4

reconstruction_loss = tf.reduce_mean(tf.square(outputs - X))
reg_loss = regularizer(weights1) + regularizer(weights2)
loss = reconstruction_loss + reg_loss

optimizer = tf.train.AdamOptimizer(learning_rate)
training_op = optimizer.minimize(loss)

init = tf.global_variables_initializer()
```

This code is fairly straightforward, but there are a few important things to note:

- 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.

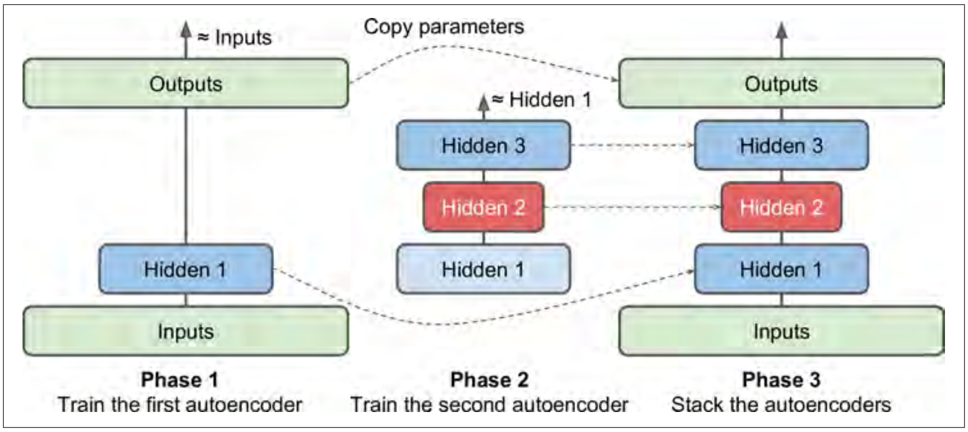

Figure 15-4. Training one autoencoder at a time

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