ing layer to be just as large as the inputs, or even larger, resulting in an *overcomplete autoencoder*. Let's look at some of those approaches now.

Denoising Autoencoders

Another way to force the autoencoder to learn useful features is to add noise to its inputs, training it to recover the original, noise-free inputs. This prevents the autoencoder from trivially copying its inputs to its outputs, so it ends up having to find patterns in the data.

The idea of using autoencoders to remove noise has been around since the 1980s (e.g., it is mentioned in Yann LeCun's 1987 master's thesis). In a 2008 paper,[3] Pascal Vincent et al. showed that autoencoders could also be used for feature extraction. In a 2010 paper,[4] Vincent et al. introduced *stacked denoising autoencoders*.

The noise can be pure Gaussian noise added to the inputs, or it can be randomly switched off inputs, just like in dropout (introduced in Chapter 11). Figure 15-9 shows both options.

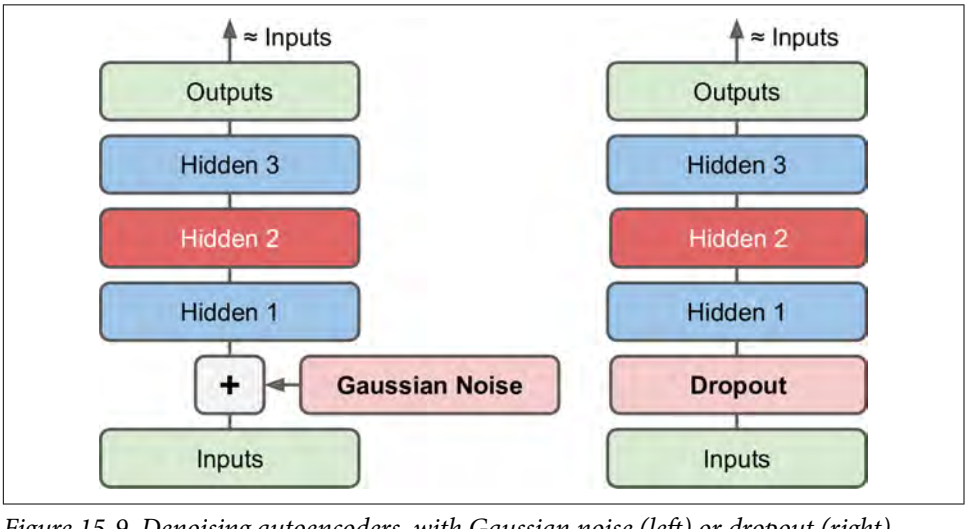

Figure 15-9. Denoising autoencoders, with Gaussian noise (left) or dropout (right)

3 "Extracting and Composing Robust Features with Denoising Autoencoders," P. Vincent et al. (2008).

4 "Stacked Denoising Autoencoders: Learning Useful Representations in a Deep Network with a Local Denoising Criterion," P. Vincent et al. (2010).

TensorFlow Implementation

Implementing denoising autoencoders in TensorFlow is not too hard. Let's start with Gaussian noise. It's really just like training a regular autoencoder, except you add noise to the inputs, and the reconstruction loss is calculated based on the original inputs:

```
X = tf.placeholder(tf.float32, shape=[None, n_inputs])
X_noisy = X + tf.random_normal(tf.shape(X))
[...]
hidden1 = activation(tf.matmul(X_noisy, weights1) + biases1)
[...]
reconstruction_loss = tf.reduce_mean(tf.square(outputs - X))  # MSE
[...]
```

Since the shape of `X` is only partially defined during the construction phase, we cannot know in advance the shape of the noise that we must add to `X`. We cannot call `X.get_shape()` because this would just return the partially defined shape of `X` (`[None, n_inputs]`), and `random_normal()` expects a fully defined shape so it would raise an exception. Instead, we call `tf.shape(X)`, which creates an operation that will return the shape of `X` at runtime, which will be fully defined at that point.

Implementing the dropout version, which is more common, is not much harder:

```
from tensorflow.contrib.layers import dropout

keep_prob = 0.7

is_training = tf.placeholder_with_default(False, shape=(), name='is_training')
X = tf.placeholder(tf.float32, shape=[None, n_inputs])
X_drop = dropout(X, keep_prob, is_training=is_training)
[...]
hidden1 = activation(tf.matmul(X_drop, weights1) + biases1)
[...]
reconstruction_loss = tf.reduce_mean(tf.square(outputs - X))  # MSE
[...]
```

During training we must set `is_training` to `True` (as explained in Chapter 11) using the `feed_dict`:

```
sess.run(training_op, feed_dict={X: X_batch, is_training: True})
```

However, during testing it is not necessary to set `is_training` to `False`, since we set that as the default in the call to the `placeholder_with_default()` function.