

## 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.

## Sparse Autoencoders

Another kind of constraint that often leads to good feature extraction is *sparsity*: by adding an appropriate term to the cost function, the autoencoder is pushed to reduce the number of active neurons in the coding layer. For example, it may be pushed to have on average only 5% significantly active neurons in the coding layer. This forces the autoencoder to represent each input as a combination of a small number of activations. As a result, each neuron in the coding layer typically ends up representing a useful feature (if you could speak only a few words per month, you would probably try to make them worth listening to).

In order to favor sparse models, we must first measure the actual sparsity of the coding layer at each training iteration. We do so by computing the average activation of each neuron in the coding layer, over the whole training batch. The batch size must not be too small, or else the mean will not be accurate.

Once we have the mean activation per neuron, we want to penalize the neurons that are too active by adding a *sparsity loss* to the cost function. For example, if we measure that a neuron has an average activation of 0.3, but the target sparsity is 0.1, it must be penalized to activate less. One approach could be simply adding the squared error  $(0.3 - 0.1)^2$  to the cost function, but in practice a better approach is to use the Kullback–Leibler divergence (briefly discussed in [Chapter 4](#)), which has much stronger gradients than the Mean Squared Error, as you can see in [Figure 15-10](#).

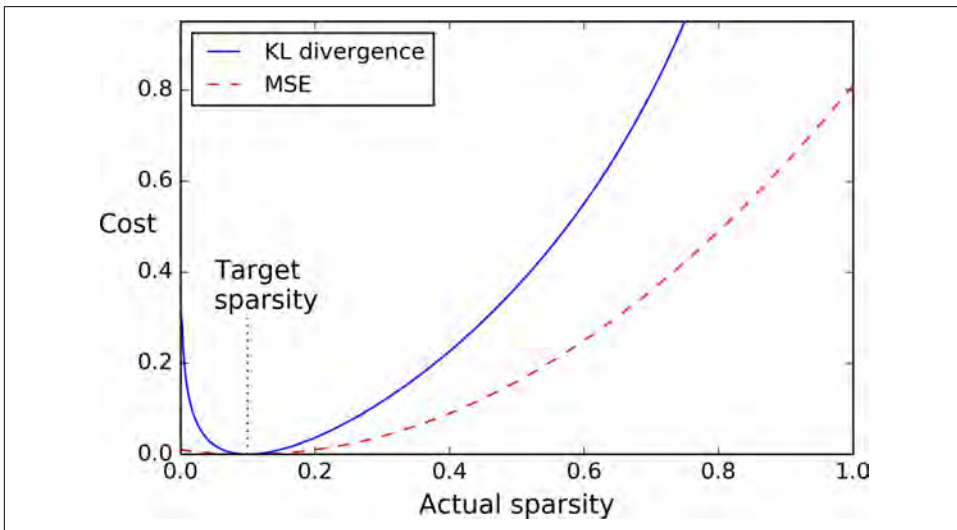


Figure 15-10. Sparsity loss