Given two discrete probability distributions P and Q, the KL divergence between these distributions, noted $D_{\text{KL}}(P \parallel Q)$, can be computed using Equation 15-1.

Equation 15-1. Kullback–Leibler divergence

$$D_{\text{KL}}(P \parallel Q) = \sum_i P(i) \log \frac{P(i)}{Q(i)}$$

In our case, we want to measure the divergence between the target probability p that a neuron in the coding layer will activate, and the actual probability q (i.e., the mean activation over the training batch). So the KL divergence simplifies to Equation 15-2.

Equation 15-2. KL divergence between the target sparsity p and the actual sparsity q

$$D_{\text{KL}}(p \parallel q) = p \log \frac{p}{q} + (1 - p) \log \frac{1 - p}{1 - q}$$

Once we have computed the sparsity loss for each neuron in the coding layer, we just sum up these losses, and add the result to the cost function. In order to control the relative importance of the sparsity loss and the reconstruction loss, we can multiply the sparsity loss by a sparsity weight hyperparameter. If this weight is too high, the model will stick closely to the target sparsity, but it may not reconstruct the inputs properly, making the model useless. Conversely, if it is too low, the model will mostly ignore the sparsity objective and it will not learn any interesting features.

TensorFlow Implementation

We now have all we need to implement a sparse autoencoder using TensorFlow:

```
def kl_divergence(p, q):
    return p * tf.log(p / q) + (1 - p) * tf.log((1 - p) / (1 - q))

learning_rate = 0.01
sparsity_target = 0.1
sparsity_weight = 0.2

[...] # Build a normal autoencoder (in this example the coding layer is hidden1)

optimizer = tf.train.AdamOptimizer(learning_rate)

hidden1_mean = tf.reduce_mean(hidden1, axis=0)  # batch mean
sparsity_loss = tf.reduce_sum(kl_divergence(sparsity_target, hidden1_mean))
reconstruction_loss = tf.reduce_mean(tf.square(outputs - X))  # MSE
loss = reconstruction_loss + sparsity_weight * sparsity_loss
training_op = optimizer.minimize(loss)
```

An important detail is the fact that the activations of the coding layer must be between 0 and 1 (but not equal to 0 or 1), or else the KL divergence will return NaN

(Not a Number). A simple solution is to use the logistic activation function for the coding layer:

```
hidden1 = tf.nn.sigmoid(tf.matmul(X, weights1) + biases1)
```

One simple trick can speed up convergence: instead of using the MSE, we can choose a reconstruction loss that will have larger gradients. Cross entropy is often a good choice. To use it, we must normalize the inputs to make them take on values from 0 to 1, and use the logistic activation function in the output layer so the outputs also take on values from 0 to 1. TensorFlow's `sigmoid_cross_entropy_with_logits()` function takes care of efficiently applying the logistic (sigmoid) activation function to the outputs and computing the cross entropy:

```
[...]
logits = tf.matmul(hidden1, weights2) + biases2)
outputs = tf.nn.sigmoid(logits)

reconstruction_loss = tf.reduce_sum(
    tf.nn.sigmoid_cross_entropy_with_logits(labels=X, logits=logits))
```

Note that the `outputs` operation is not needed during training (we use it only when we want to look at the reconstructions).

Variational Autoencoders

Another important category of autoencoders was introduced in 2014 by Diederik Kingma and Max Welling,[5] and has quickly become one of the most popular types of autoencoders: *variational autoencoders*.

They are quite different from all the autoencoders we have discussed so far, in particular:

- They are *probabilistic autoencoders*, meaning that their outputs are partly determined by chance, even after training (as opposed to denoising autoencoders, which use randomness only during training).
- Most importantly, they are *generative autoencoders*, meaning that they can generate new instances that look like they were sampled from the training set.

Both these properties make them rather similar to RBMs (see Appendix E), but they are easier to train and the sampling process is much faster (with RBMs you need to wait for the network to stabilize into a "thermal equilibrium" before you can sample a new instance).

5 "Auto-Encoding Variational Bayes," D. Kingma and M. Welling (2014).