

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](#).

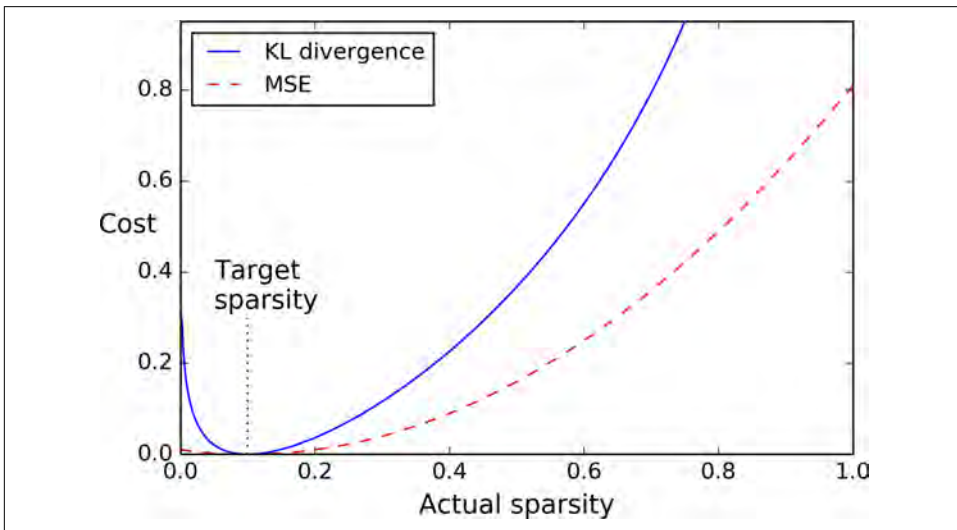


Figure 15-10. Sparsity loss

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