(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).

Let's take a look at how they work. Figure 15-11 (left) shows a variational autoencoder. You can recognize, of course, the basic structure of all autoencoders, with an encoder followed by a decoder (in this example, they both have two hidden layers), but there is a twist: instead of directly producing a coding for a given input, the encoder produces a *mean coding* μ and a standard deviation σ. The actual coding is then sampled randomly from a Gaussian distribution with mean μ and standard deviation σ. After that the decoder just decodes the sampled coding normally. The right part of the diagram shows a training instance going through this autoencoder. First, the encoder produces μ and σ, then a coding is sampled randomly (notice that it is not exactly located at μ), and finally this coding is decoded, and the final output resembles the training instance.

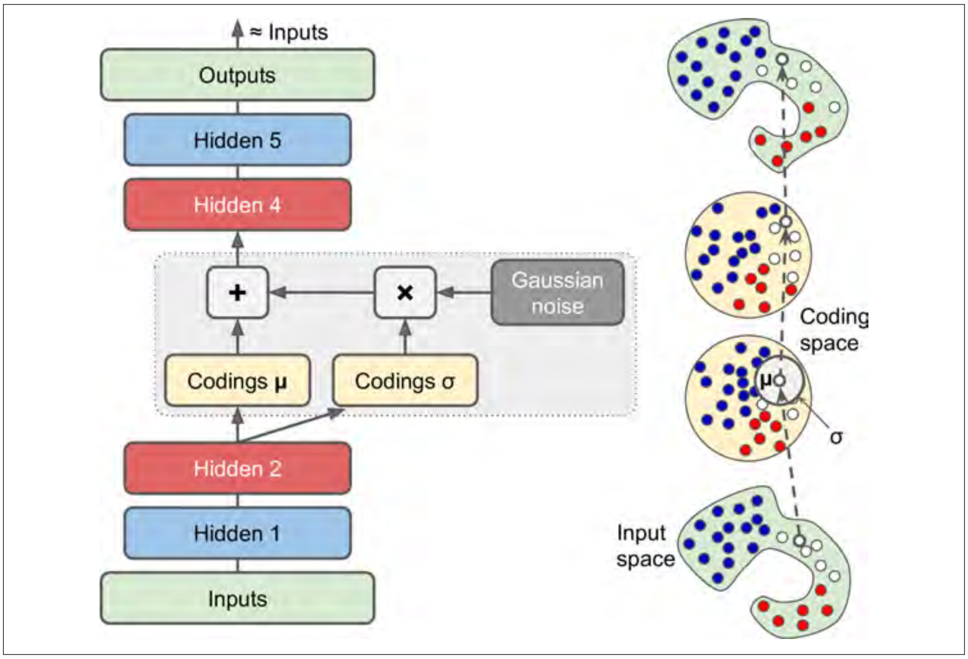

Figure 15-11. Variational autoencoder (left), and an instance going through it (right)

As you can see on the diagram, although the inputs may have a very convoluted distribution, a variational autoencoder tends to produce codings that look as though they were sampled from a simple Gaussian distribution:[6] during training, the cost function (discussed next) pushes the codings to gradually migrate within the coding space (also called the *latent space*) to occupy a roughly (hyper)spherical region that looks like a cloud of Gaussian points. One great consequence is that after training a

6 Variational autoencoders are actually more general; the codings are not limited to Gaussian distributions.

variational autoencoder, you can very easily generate a new instance: just sample a random coding from the Gaussian distribution, decode it, and voilà!

So let's look at the cost function. It is composed of two parts. The first is the usual reconstruction loss that pushes the autoencoder to reproduce its inputs (we can use cross entropy for this, as discussed earlier). The second is the *latent loss* that pushes the autoencoder to have codings that look as though they were sampled from a simple Gaussian distribution, for which we use the KL divergence between the target distribution (the Gaussian distribution) and the actual distribution of the codings. The math is a bit more complex than earlier, in particular because of the Gaussian noise, which limits the amount of information that can be transmitted to the coding layer (thus pushing the autoencoder to learn useful features). Luckily, the equations simplify to the following code for the latent loss:[7]

```
eps = 1e-10  # smoothing term to avoid computing log(0) which is NaN
latent_loss = 0.5 * tf.reduce_sum(
    tf.square(hidden3_sigma) + tf.square(hidden3_mean)
    - 1 - tf.log(eps + tf.square(hidden3_sigma)))
```

One common variant is to train the encoder to output $\gamma = \log(\sigma^2)$ rather than σ. Wherever we need σ we can just compute $\sigma = \exp\left(\frac{\gamma}{2}\right)$. This makes it a bit easier for the encoder to capture sigmas of different scales, and thus it helps speed up convergence. The latent loss ends up a bit simpler:

```
latent_loss = 0.5 * tf.reduce_sum(
    tf.exp(hidden3_gamma) + tf.square(hidden3_mean) - 1 - hidden3_gamma)
```

The following code builds the variational autoencoder shown in Figure 15-11 (left), using the $\log(\sigma^2)$ variant:

```
n_inputs = 28 * 28  # for MNIST
n_hidden1 = 500
n_hidden2 = 500
n_hidden3 = 20  # codings
n_hidden4 = n_hidden2
n_hidden5 = n_hidden1
n_outputs = n_inputs

learning_rate = 0.001

with tf.contrib.framework.arg_scope(
        [fully_connected],
        activation_fn=tf.nn.elu,
        weights_initializer=tf.contrib.layers.variance_scaling_initializer()):
    X = tf.placeholder(tf.float32, [None, n_inputs])
```

7 For more mathematical details, check out the original paper on variational autoencoders, or Carl Doersch's great tutorial (2016).

```
    hidden1 = fully_connected(X, n_hidden1)
    hidden2 = fully_connected(hidden1, n_hidden2)
    hidden3_mean = fully_connected(hidden2, n_hidden3, activation_fn=None)
    hidden3_gamma = fully_connected(hidden2, n_hidden3, activation_fn=None)
    hidden3_sigma = tf.exp(0.5 * hidden3_gamma)
    noise = tf.random_normal(tf.shape(hidden3_sigma), dtype=tf.float32)
    hidden3 = hidden3_mean + hidden3_sigma * noise
    hidden4 = fully_connected(hidden3, n_hidden4)
    hidden5 = fully_connected(hidden4, n_hidden5)
    logits = fully_connected(hidden5, n_outputs, activation_fn=None)
    outputs = tf.sigmoid(logits)

reconstruction_loss = tf.reduce_sum(
    tf.nn.sigmoid_cross_entropy_with_logits(labels=X, logits=logits))
latent_loss = 0.5 * tf.reduce_sum(
    tf.exp(hidden3_gamma) + tf.square(hidden3_mean) - 1 - hidden3_gamma)
cost = reconstruction_loss + latent_loss

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

init = tf.global_variables_initializer()
```

Generating Digits

Now let's use this variational autoencoder to generate images that look like handwritten digits. All we need to do is train the model, then sample random codings from a Gaussian distribution and decode them.

```
import numpy as np

n_digits = 60
n_epochs = 50
batch_size = 150

with tf.Session() as sess:
    init.run()
    for epoch in range(n_epochs):
        n_batches = mnist.train.num_examples // batch_size
        for iteration in range(n_batches):
            X_batch, y_batch = mnist.train.next_batch(batch_size)
            sess.run(training_op, feed_dict={X: X_batch})

    codings_rnd = np.random.normal(size=[n_digits, n_hidden3])
    outputs_val = outputs.eval(feed_dict={hidden3: codings_rnd})
```

That's it. Now we can see what the "handwritten" digits produced by the autoencoder look like (see Figure 15-12):

```
for iteration in range(n_digits):
    plt.subplot(n_digits, 10, iteration + 1)
    plot_image(outputs_val[iteration])
```