15. Autoencoders - Hands-On Machine Learning with Scikit-Learn and TensorFlow

Chapter 15. Autoencoders

Autoencoders are artificial neural networks capable of learning efficient representations of the input data, called *codings*, without any supervision (i.e., the training set is unlabeled). These codings typically have a much lower dimensionality than the input data, making autoencoders useful for dimensionality reduction (see Chapter 8). More importantly, autoencoders act as powerful feature detectors, and they can be used for unsupervised pretraining of deep neural networks (as we discussed in Chapter 11). Lastly, they are capable of randomly generating new data that looks very similar to the training data; this is called a *generative model*. For example, you could train an autoencoder on pictures of faces, and it would then be able to generate new faces.

Surprisingly, autoencoders work by simply learning to copy their inputs to their outputs. This may sound like a trivial task, but we will see that constraining the network in various ways can make it rather difficult. For example, you can limit the size of the internal representation, or you can add noise to the inputs and train the network to recover the original inputs. These constraints prevent the autoencoder from trivially copying the inputs directly to the outputs, which forces it to learn efficient ways of representing the data. In short, the codings are byproducts of the autoencoder's attempt to learn the identity function under some constraints.

In this chapter we will explain in more depth how autoencoders work, what types of constraints can be imposed, and how to implement them using TensorFlow, whether it is for dimensionality reduction, feature extraction, unsupervised pretraining, or as generative models.

Efficient Data Representations

Which of the following number sequences do you find the easiest to memorize?

- 40, 27, 25, 36, 81, 57, 10, 73, 19, 68
- 50, 25, 76, 38, 19, 58, 29, 88, 44, 22, 11, 34, 17, 52, 26, 13, 40, 20

At first glance, it would seem that the first sequence should be easier, since it is much shorter. However, if you look carefully at the second sequence, you may notice that it follows two simple rules: even numbers are followed by their half, and odd numbers are followed by their triple plus one (this is a famous sequence known as the *hailstone sequence*). Once you notice this pattern, the second sequence becomes much easier to memorize than the first because you only need to memorize the two rules, the first number, and the length of the sequence. Note that if you could quickly and easily memorize very long sequences, you would not care much about the existence of a pattern in the second sequence. You would just learn every number by heart, and that would be that. It is the fact that it is hard to memorize long sequences that makes it useful to recognize patterns, and hopefully this clarifies why constraining an autoencoder during training pushes it to discover and exploit patterns in the data.

The relationship between memory, perception, and pattern matching was famously studied by William Chase and Herbert Simon in the early 1970s. They observed that expert chess players were able to memorize the positions of all the pieces in a game by looking at the board for just 5 seconds, a task that most people would find impossible. However, this was only the case when the pieces were placed in realistic positions (from actual games), not when the pieces were placed randomly. Chess experts don't have a much better memory than you and I, they just see chess patterns more easily thanks to their experience with the game. Noticing patterns helps them store information

efficiently.

Just like the chess players in this memory experiment, an autoencoder looks at the inputs, converts them to an efficient internal representation, and then spits out something that (hopefully) looks very close to the inputs. An autoencoder is always composed of two parts: an *encoder* (or *recognition network*) that converts the inputs to an internal representation, followed by a *decoder* (or *generative network*) that converts the internal representation to the outputs (see Figure 15-1).

As you can see, an autoencoder typically has the same architecture as a Multi-Layer Perceptron (MLP; see Chapter 10), except that the number of neurons in the output layer must be equal to the number of inputs. In this example, there is just one hidden layer composed of two neurons (the encoder), and one output layer composed of three neurons (the decoder). The outputs are often called the *reconstructions* since the autoencoder tries to reconstruct the inputs, and the cost function contains a *reconstruction loss* that penalizes the model when the reconstructions are different from the inputs.

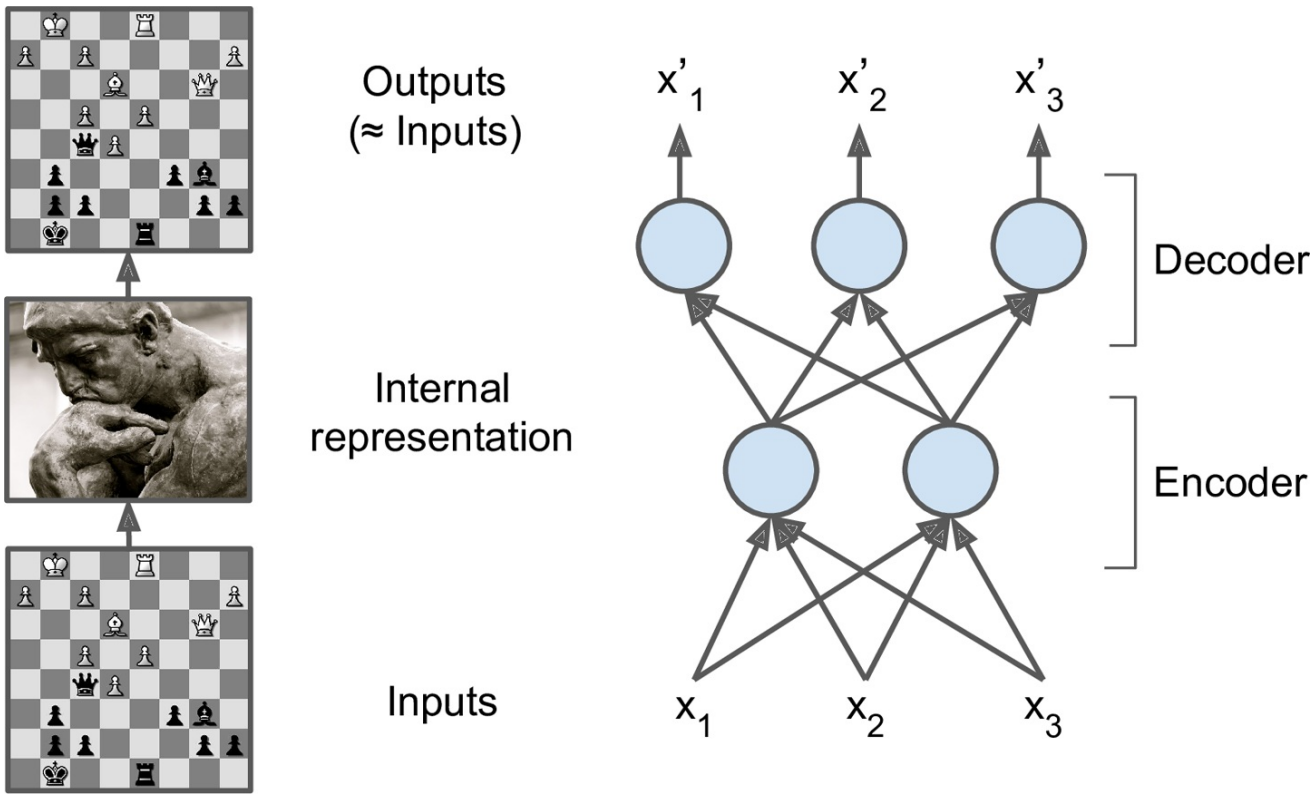

Figure 15-1. The chess memory experiment (left) and a simple autoencoder (right)

Because the internal representation has a lower dimensionality than the input data (it is 2D instead of 3D), the autoencoder is said to be *undercomplete*. An undercomplete autoencoder cannot trivially copy its inputs to the codings, yet it must find a way to output a copy of its inputs. It is forced to learn the most important features in the input data (and drop the unimportant ones).

Let's see how to implement a very simple undercomplete autoencoder for dimensionality reduction.

Performing PCA with an Undercomplete Linear Autoencoder

If the autoencoder uses only linear activations and the cost function is the Mean Squared Error (MSE), then it can be shown that it ends up performing Principal Component Analysis (see Chapter 8).

The following code builds a simple linear autoencoder to perform PCA on a 3D dataset, projecting it to 2D:

```
import tensorflow as tf
from tensorflow.contrib.layers import fully_connected

              # 3D
n_inputs = 3  inputs
              # 2D
n_hidden = 2  codings
n_outputs = n_inputs

learning_rate = 0.01

X = tf.placeholder(tf.float32, shape=[None, n_inputs])
hidden = fully_connected(X, n_hidden, activation_fn=None)
outputs = fully_connected(hidden, n_outputs, activation_fn=None)

reconstruction_loss = tf.reduce_mean(tf.square(outputs - X))
#
MSE

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

init = tf.global_variables_initializer()
```

This code is really not very different from all the MLPs we built in past chapters. The two things to note are:

- The number of outputs is equal to the number of inputs.
- To perform simple PCA, we set `activation_fn=None` (i.e., all neurons are linear) and the cost function is the MSE. We will see more complex autoencoders shortly.

Now let's load the dataset, train the model on the training set, and use it to encode the test set (i.e., project it to 2D):

```
                        # load the
X_train, X_test = [...] dataset

n_iterations = 1000
                  # the output of the hidden layer provides the
codings = hidden  codings

with tf.Session() as sess:
    init.run()
    for iteration in range(n_iterations):
        training_op.run(feed_dict={X: X_train})
# no labels
(unsupervised)
    codings_val = codings.eval(feed_dict={X: X_test})
```

Figure 15-2 shows the original 3D dataset (at the left) and the output of the autoencoder's hidden layer (i.e., the coding layer, at the right). As you can see, the autoencoder found the best 2D plane to project the data onto, preserving as much variance in the data as it could (just like PCA).

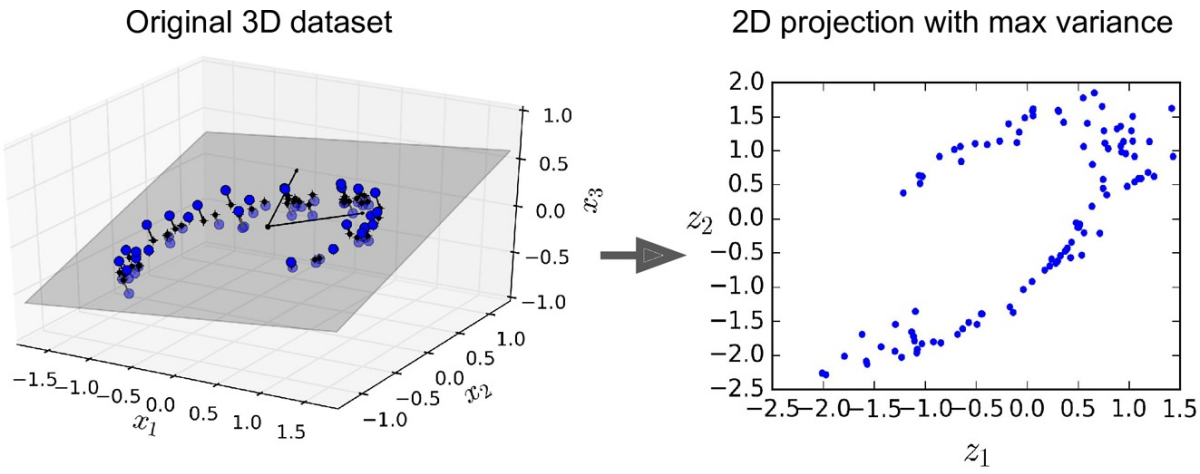

Figure 15-2. PCA performed by an undercomplete linear autoencoder

Stacked Autoencoders

Just like other neural networks we have discussed, autoencoders can have multiple hidden layers. In this case they are called *stacked autoencoders* (or *deep autoencoders*). Adding more layers helps the autoencoder learn more complex codings. However, one must be careful not to make the autoencoder too powerful. Imagine an encoder so powerful that it just learns to map each input to a single arbitrary number (and the decoder learns the reverse mapping). Obviously such an autoencoder will reconstruct the training data perfectly, but it will not have learned any useful data representation in the process (and it is unlikely to generalize well to new instances).

The architecture of a stacked autoencoder is typically symmetrical with regards to the central hidden layer (the coding layer). To put it simply, it looks like a sandwich. For example, an autoencoder for MNIST (introduced in Chapter 3) may have 784 inputs, followed by a hidden layer with 300 neurons, then a central hidden layer of 150 neurons, then another hidden layer with 300 neurons, and an output layer with 784 neurons. This stacked autoencoder is represented in Figure 15-3.

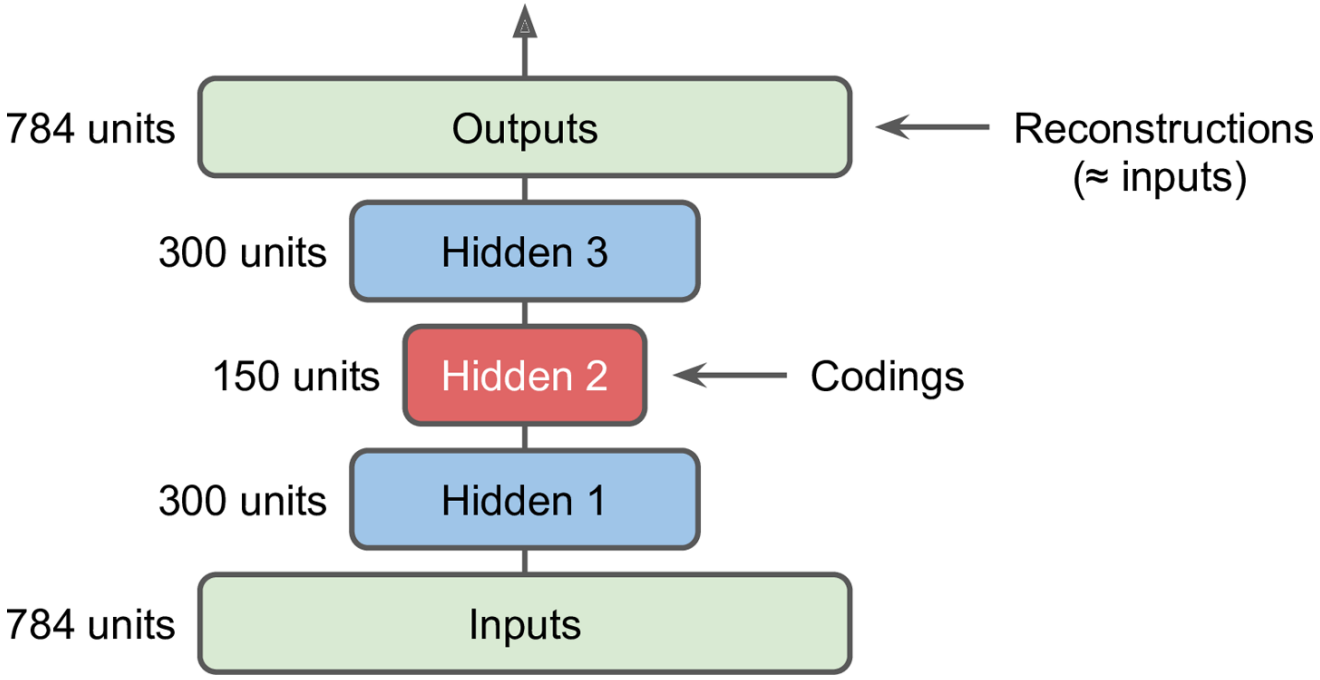

Figure 15-3. Stacked autoencoder

TensorFlow Implementation

You can implement a stacked autoencoder very much like a regular deep MLP. In particular, the same techniques we used in Chapter 11 for training deep nets can be applied. For example, the following code builds a stacked autoencoder for MNIST, using He initialization, the ELU activation function, and ℓ_2 regularization. The code should look very familiar, except that there are no labels (no `y`):

```
                    # for
n_inputs = 28 * 28  MNIST
n_hidden1 = 300
                  #
n_hidden2 = 150  codings
n_hidden3 = n_hidden1
n_outputs = n_inputs

learning_rate = 0.01
l2_reg = 0.001

X = tf.placeholder(tf.float32, shape=[None, n_inputs])
with tf.contrib.framework.arg_scope(
        [fully_connected],
        activation_fn=tf.nn.elu,
        weights_initializer=tf.contrib.layers.variance_scaling_initializer
(),
        weights_regularizer=tf.contrib.layers.l2_regularizer(l2_reg)):
    hidden1 = fully_connected(X, n_hidden1)
                                                    #
    hidden2 = fully_connected(hidden1, n_hidden2)  codings
    hidden3 = fully_connected(hidden2, n_hidden3)
    outputs = fully_connected(hidden3, n_outputs, activation_fn=None)

                                                                 #
reconstruction_loss = tf.reduce_mean(tf.square(outputs - X))  MSE

reg_losses = tf.get_collection(tf.GraphKeys.REGULARIZATION_LOSSES)
loss = tf.add_n([reconstruction_loss] + reg_losses)

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

init = tf.global_variables_initializer()
```

You can then train the model normally. Note that the digit labels (`y_batch`) are unused:

```
n_epochs = 5
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})
```

Tying Weights

When an autoencoder is neatly symmetrical, like the one we just built, a common technique is to *tie the weights* of the decoder layers to the weights of the encoder layers. This halves the number of weights in the model, speeding up training and limiting the risk of overfitting. Specifically, if the autoencoder has a total of *N* layers (not counting the input layer), and $\mathbf{W}_L$ represents the connection weights of the L^{th} layer (e.g., layer 1 is the first hidden layer, layer $\frac{N}{2}$ is the coding layer, and layer *N* is the output layer), then the decoder layer weights can be defined simply as: $\mathbf{W}_{N-L+1} = \mathbf{W}_L^T$ (with $L = 1, 2, \cdots, \frac{N}{2}$).

Unfortunately, implementing tied weights in TensorFlow using the `fully_connected()` function is a bit cumbersome; it's actually easier to just define the layers manually. The code ends up significantly more verbose:

```
activation = tf.nn.elu
regularizer = tf.contrib.layers.l2_regularizer(l2_reg)
initializer = tf.contrib.layers.variance_scaling_initializer()

X = tf.placeholder(tf.float32, shape=[None, n_inputs])

weights1_init = initializer([n_inputs, n_hidden1])
weights2_init = initializer([n_hidden1, n_hidden2])

weights1 = tf.Variable(weights1_init, dtype=tf.float32, name="weights1")
weights2 = tf.Variable(weights2_init, dtype=tf.float32, name="weights2")
                                                       # tied
weights3 = tf.transpose(weights2, name="weights3")     weights
                                                       # tied
weights4 = tf.transpose(weights1, name="weights4")     weights

biases1 = tf.Variable(tf.zeros(n_hidden1), name="biases1")
biases2 = tf.Variable(tf.zeros(n_hidden2), name="biases2")
biases3 = tf.Variable(tf.zeros(n_hidden3), name="biases3")
biases4 = tf.Variable(tf.zeros(n_outputs), name="biases4")

hidden1 = activation(tf.matmul(X, weights1) + biases1)
hidden2 = activation(tf.matmul(hidden1, weights2) + biases2)
hidden3 = activation(tf.matmul(hidden2, weights3) + biases3)
outputs = tf.matmul(hidden3, weights4) + biases4

reconstruction_loss = tf.reduce_mean(tf.square(outputs - X))
reg_loss = regularizer(weights1) + regularizer(weights2)
loss = reconstruction_loss + reg_loss

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

init = tf.global_variables_initializer()
```

This code is fairly straightforward, but there are a few important things to note:

- First, `weight3` and `weights4` are not variables, they are respectively the transpose of `weights2` and `weights1` (they are “tied” to them).
- Second, since they are not variables, it’s no use regularizing them: we only regularize `weights1` and `weights2`.
- Third, biases are never tied, and never regularized.

Training One Autoencoder at a Time

Rather than training the whole stacked autoencoder in one go like we just did, it is often much faster to train one shallow autoencoder at a time, then stack all of them into a single stacked autoencoder (hence the name), as shown on Figure 15-4. This is especially useful for very deep autoencoders.

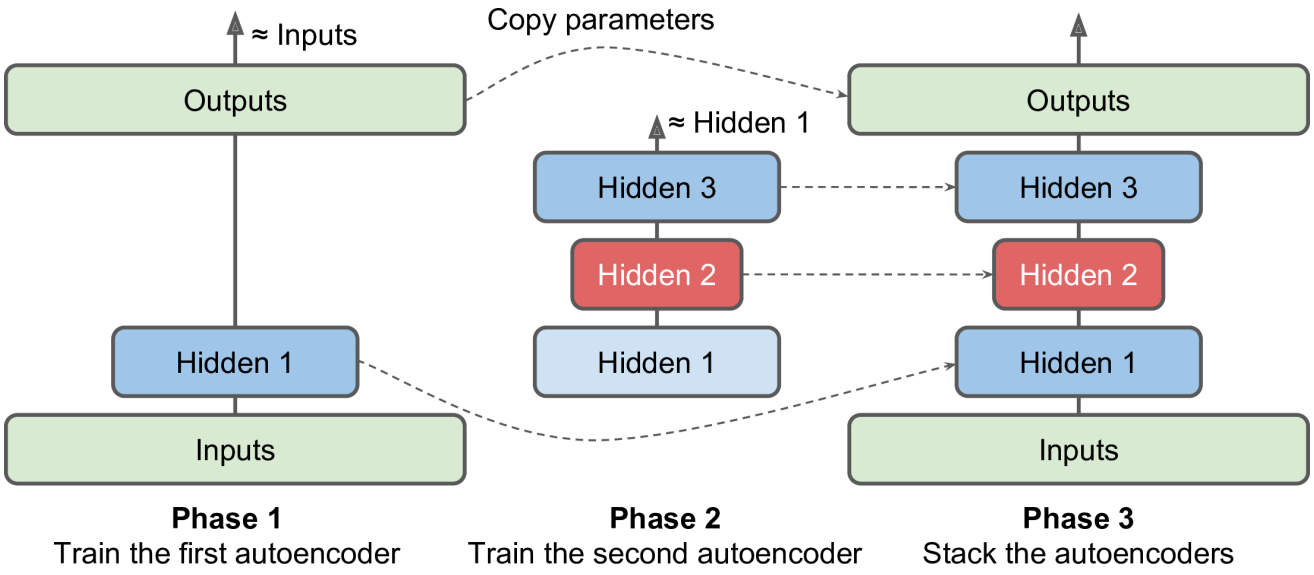

Figure 15-4. Training one autoencoder at a time

During the first phase of training, the first autoencoder learns to reconstruct the inputs. During the second phase, the second autoencoder learns to reconstruct the output of the first autoencoder's hidden layer. Finally, you just build a big sandwich using all these autoencoders, as shown in Figure 15-4 (i.e., you first stack the hidden layers of each autoencoder, then the output layers in reverse order). This gives you the final stacked autoencoder. You could easily train more autoencoders this way, building a very deep stacked autoencoder.

To implement this multiphase training algorithm, the simplest approach is to use a different TensorFlow graph for each phase. After training an autoencoder, you just run the training set through it and capture the output of the hidden layer. This output then serves as the training set for the next autoencoder. Once all autoencoders have been trained this way, you simply copy the weights and biases from each autoencoder and use them to build the stacked autoencoder. Implementing this approach is quite straightforward, so we won't detail it here, but please check out the code in the Jupyter notebooks for an example.

Another approach is to use a single graph containing the whole stacked autoencoder, plus some extra operations to perform each training phase, as shown in Figure 15-5.

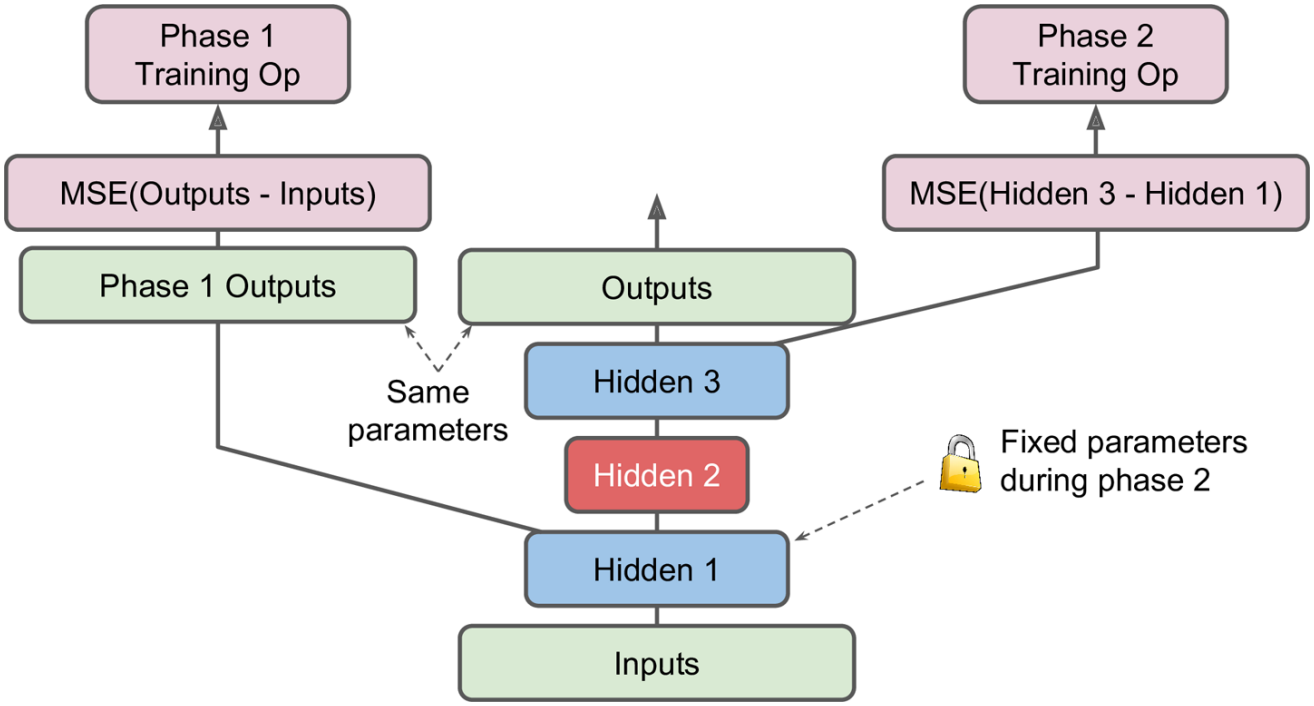

Figure 15-5. A single graph to train a stacked autoencoder

This deserves a bit of explanation:

- The central column in the graph is the full stacked autoencoder. This part can be used after training.
- The left column is the set of operations needed to run the first phase of training. It creates an output layer that bypasses hidden layers 2 and 3. This output layer shares the same weights and biases as the stacked autoencoder's output layer. On top of that are the training operations that will aim at making the output as close as possible to the inputs. Thus, this phase will train the weights and biases for the hidden layer 1 and the output layer (i.e., the first autoencoder).
- The right column in the graph is the set of operations needed to run the second phase of training. It adds the training operation that will aim at making the output of hidden layer 3 as close as possible to the output of hidden layer 1. Note that we must freeze hidden layer 1 while running phase 2. This phase will train the weights and biases for hidden layers 2 and 3 (i.e., the second autoencoder).

The TensorFlow code looks like this:

```
      # Build the whole stacked autoencoder
[...] normally.
      # In this example, the weights are not
      tied.

optimizer = tf.train.AdamOptimizer(learning_rate)

with tf.name_scope("phase1"):
    phase1_outputs = tf.matmul(hidden1, weights4) + biases4
    phase1_reconstruction_loss = tf.reduce_mean(tf.square(phase1_outputs - X
))
    phase1_reg_loss = regularizer(weights1) + regularizer(weights4)
    phase1_loss = phase1_reconstruction_loss + phase1_reg_loss
    phase1_training_op = optimizer.minimize(phase1_loss)

with tf.name_scope("phase2"):
    phase2_reconstruction_loss = tf.reduce_mean(tf.square(hidden3 - hidden1))
    phase2_reg_loss = regularizer(weights2) + regularizer(weights3)
    phase2_loss = phase2_reconstruction_loss + phase2_reg_loss
    train_vars = [weights2, biases2, weights3, biases3]
    phase2_training_op = optimizer.minimize(phase2_loss, var_list=train_vars)
```

The first phase is rather straightforward: we just create an output layer that skips hidden layers 2 and 3, then build the training operations to minimize the distance between the outputs and the inputs (plus some regularization).

The second phase just adds the operations needed to minimize the distance between the output of hidden layer 3 and hidden layer 1 (also with some regularization). Most importantly, we provide the list of trainable variables to the `minimize()` method, making sure to leave out `weights1` and `biases1`; this effectively freezes hidden layer 1 during phase 2.

During the execution phase, all you need to do is run the phase 1 training op for a number of epochs, then the phase 2 training op for some more epochs.

Tip

Since hidden layer 1 is frozen during phase 2, its output will always be the same for any given training instance. To avoid having to recompute the output of hidden layer 1 at every single epoch, you can compute it for the whole training set at the end of phase 1, then directly feed the cached output of hidden layer 1 during phase 2. This can give you a nice performance boost.

Visualizing the Reconstructions

One way to ensure that an autoencoder is properly trained is to compare the inputs and the outputs. They must be fairly similar, and the differences should be unimportant details. Let’s plot two random digits and their reconstructions:

```
n_test_digits = 2
X_test = mnist.test.images[:n_test_digits]

with tf.Session() as sess:
          # Train the
    [...] Autoencoder
    outputs_val = outputs.eval(feed_dict={X: X_test})

def plot_image(image, shape=[28, 28]):
    plt.imshow(image.reshape(shape), cmap="Greys", interpolation="nearest"
)
    plt.axis("off")

for digit_index in range(n_test_digits):
    plt.subplot(n_test_digits, 2, digit_index * 2 + 1)
    plot_image(X_test[digit_index])
    plt.subplot(n_test_digits, 2, digit_index * 2 + 2)
    plot_image(outputs_val[digit_index])
```

Figure 15-6 shows the resulting images.

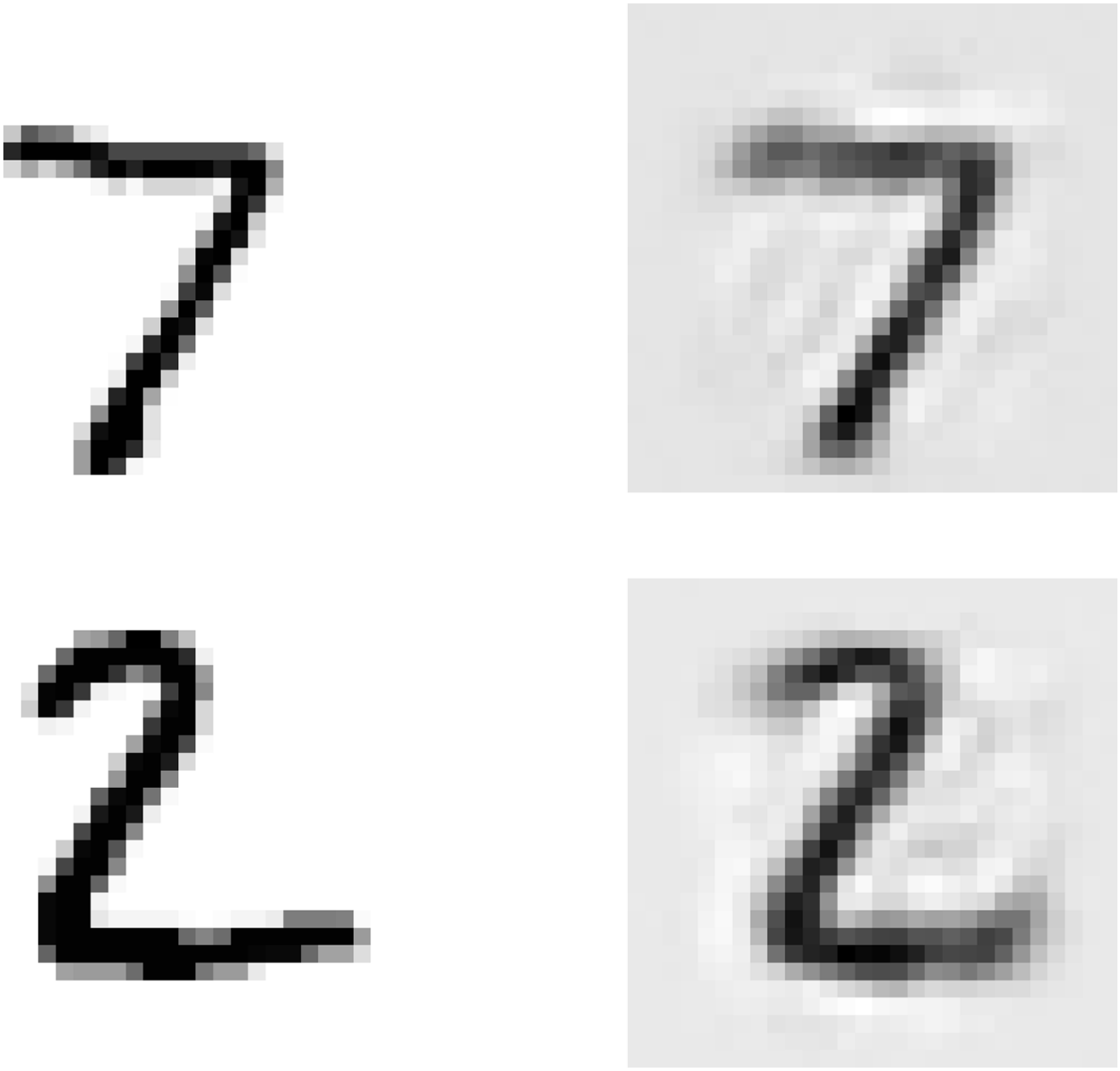

Figure 15-6. Original digits (left) and their reconstructions (right)

Looks close enough. So the autoencoder has properly learned to reproduce its inputs, but has it learned useful features? Let's take a look.

Visualizing Features

Once your autoencoder has learned some features, you may want to take a look at them. There are various techniques for this. Arguably the simplest technique is to consider each neuron in every hidden layer, and find the training instances that activate it the most. This is especially useful for the top hidden layers since they often capture relatively large features that you can easily spot in a group of training instances that contain them. For example, if a neuron strongly activates when it sees a cat in a picture, it will be pretty obvious that the pictures that activate it the most all contain cats. However, for lower layers, this technique does not work so well, as the features are smaller and more abstract, so it's often hard to understand exactly what the neuron is getting all excited about.

Let's look at another technique. For each neuron in the first hidden layer, you can create an image where a pixel's intensity corresponds to the weight of the connection to the given neuron. For example, the following code plots the features learned by five neurons in the first hidden layer:

```
with tf.Session() as sess:
            # train
    [...] autoencoder
    weights1_val = weights1.eval
()

for i in range(5):
    plt.subplot(1, 5, i + 1)
    plot_image(weights1_val.T[i])
```

You may get low-level features such as the ones shown in Figure 15-7.

The first four features seem to correspond to small patches, while the fifth feature seems to look for vertical strokes (note that these features come from the stacked denoising autoencoder that we will discuss later).

Another technique is to feed the autoencoder a random input image, measure the activation of the neuron you are interested in, and then perform backpropagation to tweak the image in such a way that the neuron will activate even more. If you iterate several times (performing gradient ascent), the image will gradually turn into the most exciting image (for the neuron). This is a useful technique to visualize the kinds of inputs that a neuron is looking for.

Finally, if you are using an autoencoder to perform unsupervised pretraining—for example, for a classification task—a simple way to verify that the features learned by the autoencoder are useful is to measure the performance of the classifier.

Unsupervised Pretraining Using Stacked Autoencoders

As we discussed in Chapter 11, if you are tackling a complex supervised task but you do not have a lot of labeled training data, one solution is to find a neural network that performs a similar task, and then reuse its lower layers. This makes it possible to train a high-performance model using only little training data because your neural network won't have to learn all the low-level features; it will just reuse the feature detectors learned by the existing net.

Similarly, if you have a large dataset but most of it is unlabeled, you can first train a stacked autoencoder using all the data, then reuse the lower layers to create a neural network for your actual task, and train it using the labeled data. For example, Figure 15-8 shows how to use a stacked autoencoder to perform unsupervised pretraining for a classification neural network. The stacked autoencoder itself is typically trained one autoencoder at a time, as discussed earlier. When training the classifier, if you really don't have much labeled training data, you may want to freeze the pretrained layers (at least the lower ones).

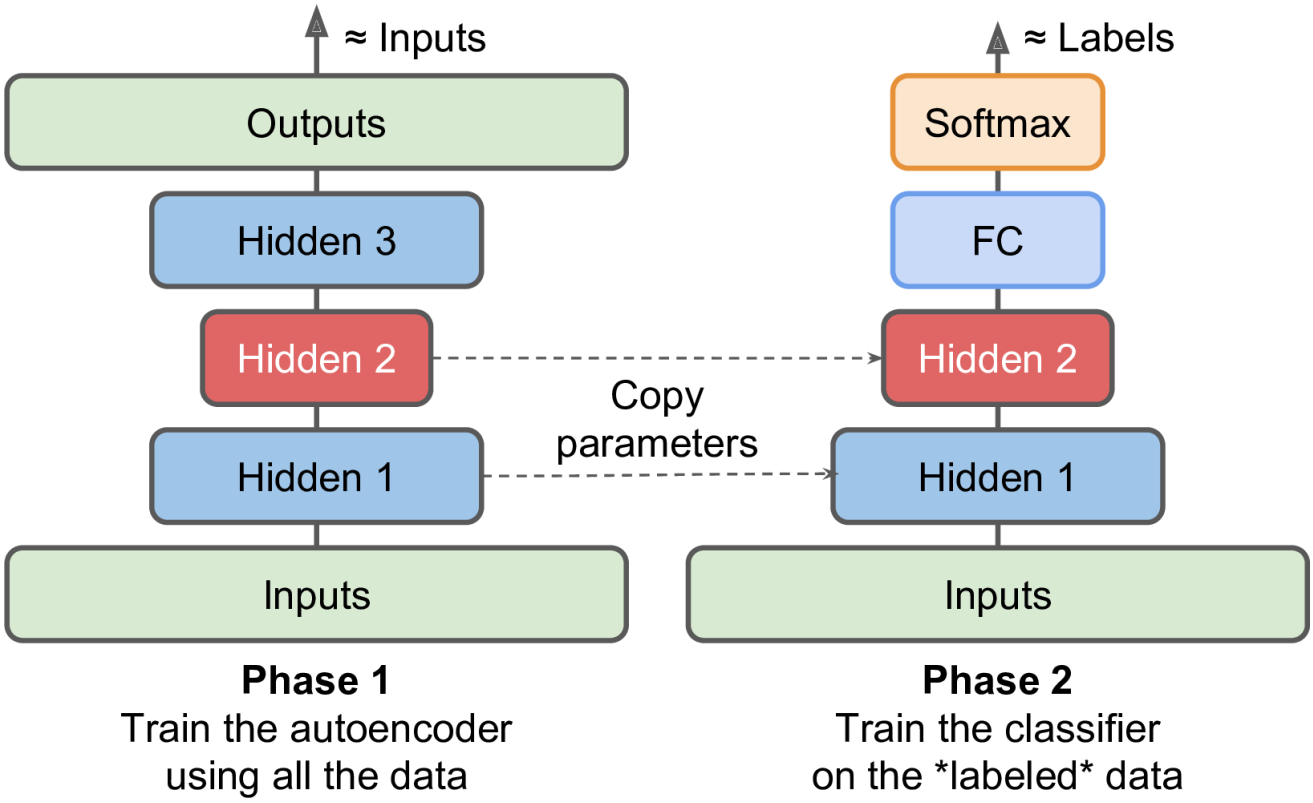

Figure 15-8. Unsupervised pretraining using autoencoders

Note

This situation is actually quite common, because building a large unlabeled dataset is often cheap (e.g., a simple script can download millions of images off the internet), but labeling them can only be done reliably by humans (e.g., classifying images as cute or not). Labeling instances is time-consuming and costly, so it is quite common to have only a few thousand labeled instances.

As we discussed earlier, one of the triggers of the current Deep Learning tsunami is the discovery in 2006 by Geoffrey Hinton et al. that deep neural networks can be pretrained in an unsupervised fashion. They used restricted Boltzmann machines for that (see Appendix E), but in 2007 Yoshua Bengio et al. showed that autoencoders worked just as well.

There is nothing special about the TensorFlow implementation: just train an autoencoder using all the training data, then reuse its encoder layers to create a new neural network (see Chapter 11 for more details on how to reuse pretrained layers, or check out the code examples in the Jupyter notebooks).

Up to now, in order to force the autoencoder to learn interesting features, we have limited the size of the coding layer, making it undercomplete. There are actually many other kinds of constraints that can be used, including ones that allow the coding 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, Pascal Vincent et al. showed that autoencoders could also be used for feature extraction. In a 2010 paper, 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.

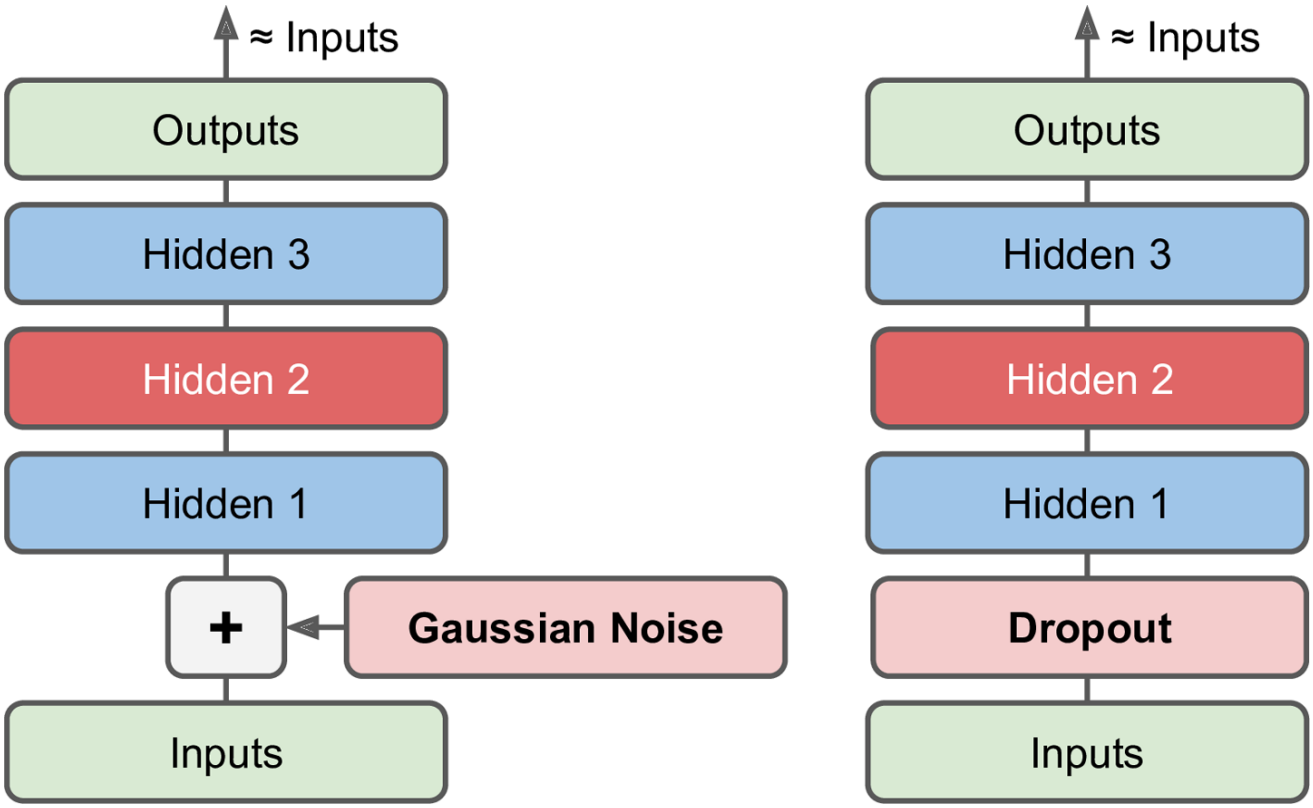

Figure 15-9. Denoising autoencoders, with Gaussian noise (left) or dropout (right)

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
[...]
```

Warning

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.

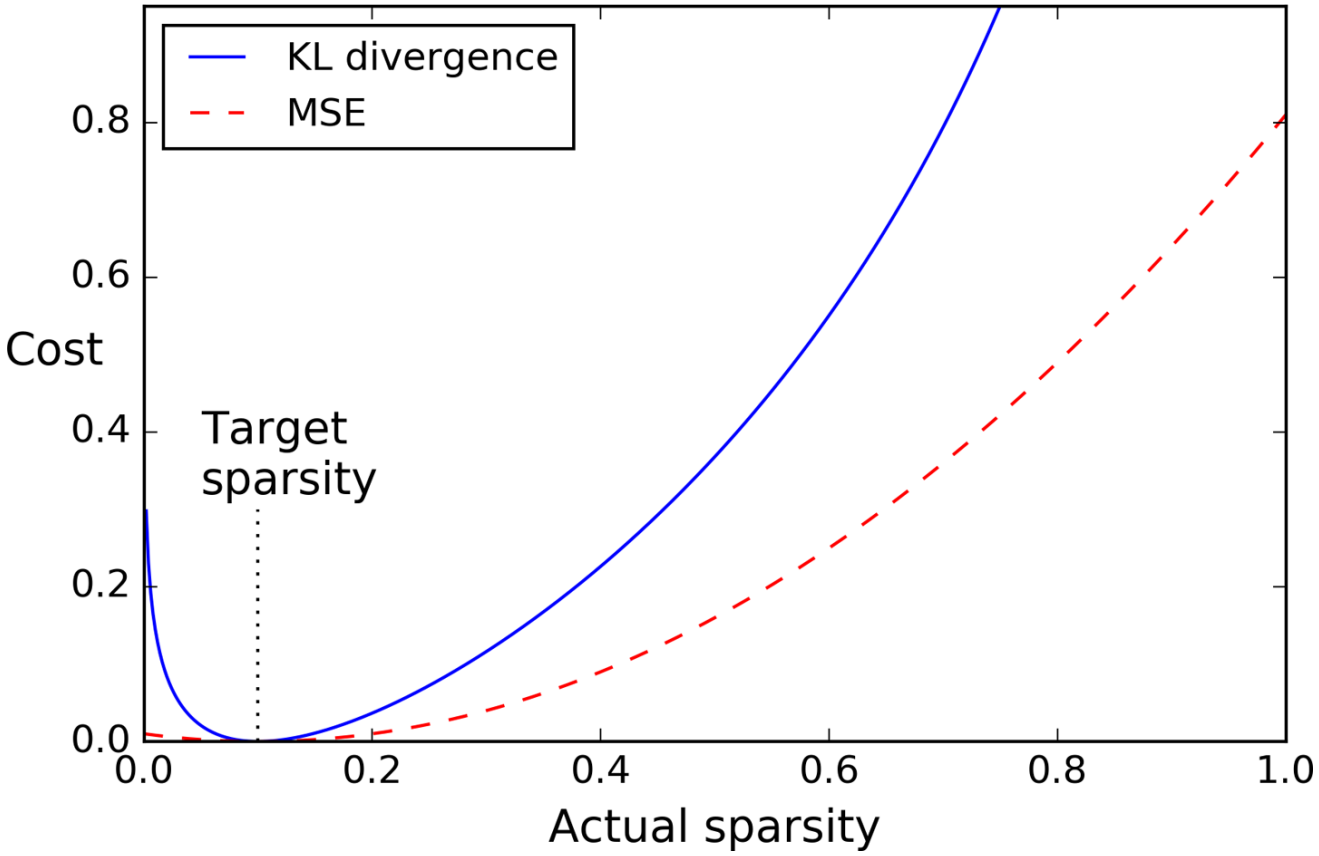

Figure 15-10. Sparsity loss

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)

                                                # batch
hidden1_mean = tf.reduce_mean(hidden1, axis=0)  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, 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).

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.

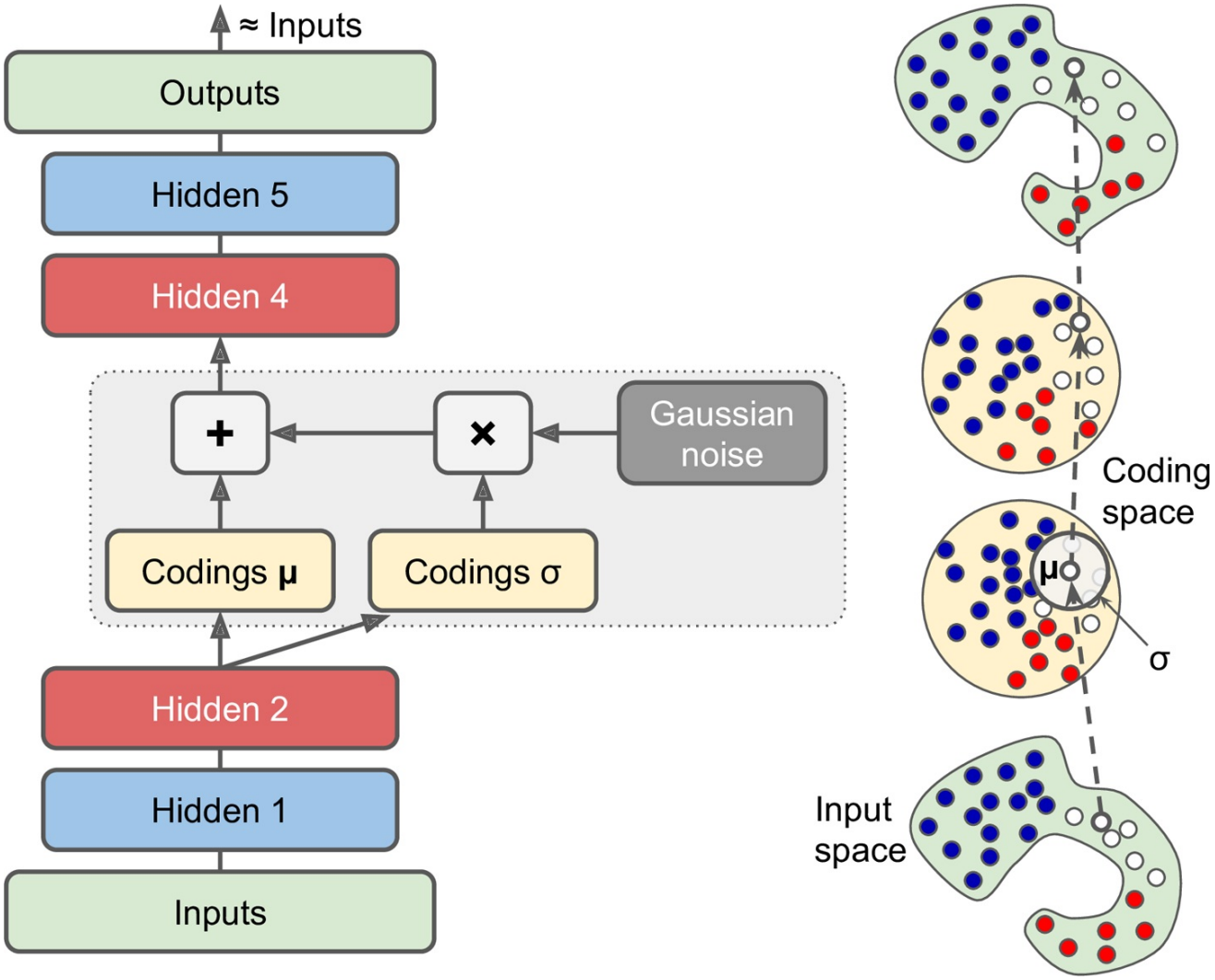

Figure 15-11. Variational autoencoder (left), and an instance going through it (right)

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: 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 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:

```
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 . 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:

$$\sigma = \exp\left(\frac{\gamma}{2}\right)$$

```
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:

```
                    # for
n_inputs = 28 * 28  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])
    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])
```

Figure 15-12. Images of handwritten digits generated by the variational autoencoder

A majority of these digits look pretty convincing, while a few are rather "creative." But don't be too harsh on the

autoencoder—it only started learning less than an hour ago. Give it a bit more training time, and those digits will look better and better.

Other Autoencoders

The amazing successes of supervised learning in image recognition, speech recognition, text translation, and more have somewhat overshadowed unsupervised learning, but it is actually booming. New architectures for autoencoders and other unsupervised learning algorithms are invented regularly, so much so that we cannot cover them all in this book. Here is a brief (by no means exhaustive) overview of a few more types of autoencoders that you may want to check out:

Contractive autoencoder (CAE)

> The autoencoder is constrained during training so that the derivatives of the codings with regards to the inputs are small. In other words, two similar inputs must have similar codings.

Stacked convolutional autoencoders

> Autoencoders that learn to extract visual features by reconstructing images processed through convolutional layers.

Generative stochastic network (GSN)[10]

> A generalization of denoising autoencoders, with the added capability to generate data.

Winner-take-all (WTA) autoencoder[11]

> During training, after computing the activations of all the neurons in the coding layer, only the top k% activations for each neuron over the training batch are preserved, and the rest are set to zero. Naturally this leads to sparse codings. Moreover, a similar WTA approach can be used to produce sparse convolutional autoencoders.

Adversarial autoencoders[12]

> One network is trained to reproduce its inputs, and at the same time another is trained to find inputs that the first network is unable to properly reconstruct. This pushes the first autoencoder to learn robust codings.

Exercises

1. What are the main tasks that autoencoders are used for?
2. Suppose you want to train a classifier and you have plenty of unlabeled training data, but only a few thousand labeled instances. How can autoencoders help? How would you proceed?
3. If an autoencoder perfectly reconstructs the inputs, is it necessarily a good autoencoder? How can you evaluate the performance of an autoencoder?
4. What are undercomplete and overcomplete autoencoders? What is the main risk of an excessively undercomplete autoencoder? What about the main risk of an overcomplete autoencoder?
5. How do you tie weights in a stacked autoencoder? What is the point of doing so?
6. What is a common technique to visualize features learned by the lower layer of a stacked autoencoder? What

about higher layers?

7. What is a generative model? Can you name a type of generative autoencoder?

8. Let's use a denoising autoencoder to pretrain an image classifier:

    - You can use MNIST (simplest), or another large set of images such as CIFAR10 if you want a bigger challenge. If you choose CIFAR10, you need to write code to load batches of images for training. If you want to skip this part, TensorFlow's model zoo contains tools to do just that.
    - Split the dataset into a training set and a test set. Train a deep denoising autoencoder on the full training set.
    - Check that the images are fairly well reconstructed, and visualize the low-level features. Visualize the images that most activate each neuron in the coding layer.
    - Build a classification deep neural network, reusing the lower layers of the autoencoder. Train it using only 10% of the training set. Can you get it to perform as well as the same classifier trained on the full training set?

9. *Semantic hashing*, introduced in 2008 by Ruslan Salakhutdinov and Geoffrey Hinton,[13] is a technique used for efficient *information retrieval*: a document (e.g., an image) is passed through a system, typically a neural network, which outputs a fairly low-dimensional binary vector (e.g., 30 bits). Two similar documents are likely to have identical or very similar hashes. By indexing each document using its hash, it is possible to retrieve many documents similar to a particular document almost instantly, even if there are billions of documents: just compute the hash of the document and look up all documents with that same hash (or hashes differing by just one or two bits). Let's implement semantic hashing using a slightly tweaked stacked autoencoder:

    - Create a stacked autoencoder containing two hidden layers below the coding layer, and train it on the image dataset you used in the previous exercise. The coding layer should contain 30 neurons and use the logistic activation function to output values between 0 and 1. After training, to produce the hash of an image, you can simply run it through the autoencoder, take the output of the coding layer, and round every value to the closest integer (0 or 1).
    - One neat trick proposed by Salakhutdinov and Hinton is to add Gaussian noise (with zero mean) to the inputs of the coding layer, during training only. In order to preserve a high signal-to-noise ratio, the autoencoder will learn to feed large values to the coding layer (so that the noise becomes negligible). In turn, this means that the logistic function of the coding layer will likely saturate at 0 or 1. As a result, rounding the codings to 0 or 1 won't distort them too much, and this will improve the reliability of the hashes.
    - Compute the hash of every image, and see if images with identical hashes look alike. Since MNIST and CIFAR10 are labeled, a more objective way to measure the performance of the autoencoder for semantic hashing is to ensure that images with the same hash generally have the same class. One way to do this is to measure the average Gini purity (introduced in Chapter 6) of the sets of images with identical (or very similar) hashes.
    - Try fine-tuning the hyperparameters using cross-validation.
    - Note that with a labeled dataset, another approach is to train a convolutional neural network (see Chapter 13) for classification, then use the layer below the output layer to produce the hashes. See Jinma Gua and Jianmin Li's 2015 paper.[14] See if that performs better.

10. Train a variational autoencoder on the image dataset used in the previous exercises (MNIST or CIFAR10), and make it generate images. Alternatively, you can try to find an unlabeled dataset that you are interested in and see if you can generate new samples.

Solutions to these exercises are available in Appendix A.

“Perception in chess,” W. Chase and H. Simon (1973).

“Greedy Layer-Wise Training of Deep Networks,” Y. Bengio et al. (2007).

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

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

“Auto-Encoding Variational Bayes,” D. Kingma and M. Welling (2014).

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

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

“Contractive Auto-Encoders: Explicit Invariance During Feature Extraction,” S. Rifai et al. (2011).

“Stacked Convolutional Auto-Encoders for Hierarchical Feature Extraction,” J. Masci et al. (2011).

[10] “GSNs: Generative Stochastic Networks,” G. Alain et al. (2015).

[11] “Winner-Take-All Autoencoders,” A. Makhzani and B. Frey (2015).

[12] “Adversarial Autoencoders,” A. Makhzani et al. (2016).

[13] “Semantic Hashing,” R. Salakhutdinov and G. Hinton (2008).

[14] “CNN Based Hashing for Image Retrieval,” J. Gua and J. Li (2015).