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.[1] 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

1 "Perception in chess," W. Chase and H. Simon (1973).

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.

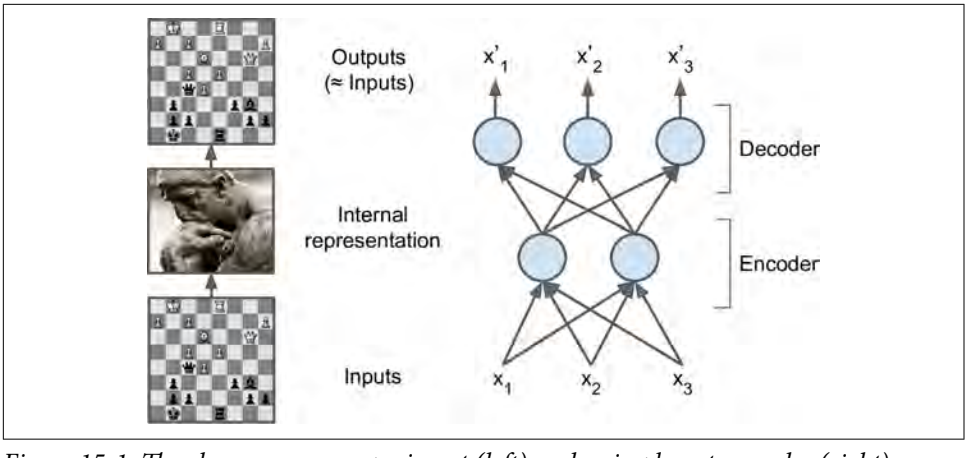

Figure 15-1. The chess memory experiment (left) and a simple autoencoder (right)

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

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