network would learn good feature detectors for faces, so reusing its lower layers would allow you to train a good face classifier using little training data.

It is often rather cheap to gather unlabeled training examples, but quite expensive to label them. In this situation, a common technique is to label all your training examples as "good," then generate many new training instances by corrupting the good ones, and label these corrupted instances as "bad." Then you can train a first neural network to classify instances as good or bad. For example, you could download millions of sentences, label them as "good," then randomly change a word in each sentence and label the resulting sentences as "bad." If a neural network can tell that "The dog sleeps" is a good sentence but "The dog they" is bad, it probably knows quite a lot about language. Reusing its lower layers will likely help in many language processing tasks.

Another approach is to train a first network to output a score for each training instance, and use a cost function that ensures that a good instance's score is greater than a bad instance's score by at least some margin. This is called *max margin learning*.

## Faster Optimizers

Training a very large deep neural network can be painfully slow. So far we have seen four ways to speed up training (and reach a better solution): applying a good initialization strategy for the connection weights, using a good activation function, using Batch Normalization, and reusing parts of a pretrained network. Another huge speed boost comes from using a faster optimizer than the regular Gradient Descent optimizer. In this section we will present the most popular ones: Momentum optimization, Nesterov Accelerated Gradient, AdaGrad, RMSProp, and finally Adam optimization.

Spoiler alert: the conclusion of this section is that you should almost always use Adam optimization,[10] so if you don't care about how it works, simply replace your `GradientDescentOptimizer` with an `AdamOptimizer` and skip to the next section! With just this small change, training will typically be several times faster. However, Adam optimization does have three hyperparameters that you can tune (plus the learning rate); the default values usually work fine, but if you ever need to tweak them it may be helpful to know what they do. Adam optimization combines several ideas from other optimization algorithms, so it is useful to look at these algorithms first.

10 At least for now: research is moving fast, especially in the field of optimization. Be sure to take a look at the latest and greatest optimizers every time a new version of TensorFlow is released.

## Momentum optimization

Imagine a bowling ball rolling down a gentle slope on a smooth surface: it will start out slowly, but it will quickly pick up momentum until it eventually reaches terminal velocity (if there is some friction or air resistance). This is the very simple idea behind *Momentum optimization*, proposed by Boris Polyak in 1964.[11] In contrast, regular Gradient Descent will simply take small regular steps down the slope, so it will take much more time to reach the bottom.

Recall that Gradient Descent simply updates the weights $\theta$ by directly subtracting the gradient of the cost function $J(\theta)$ with regards to the weights ($\nabla_\theta J(\theta)$) multiplied by the learning rate $\eta$. The equation is: $\theta \leftarrow \theta - \eta\nabla_\theta J(\theta)$. It does not care about what the earlier gradients were. If the local gradient is tiny, it goes very slowly.

Momentum optimization cares a great deal about what previous gradients were: at each iteration, it adds the local gradient to the *momentum vector* $\mathbf{m}$ (multiplied by the learning rate $\eta$), and it updates the weights by simply subtracting this momentum vector (see Equation 11-4). In other words, the gradient is used as an acceleration, not as a speed. To simulate some sort of friction mechanism and prevent the momentum from growing too large, the algorithm introduces a new hyperparameter $\beta$, simply called the *momentum*, which must be set between 0 (high friction) and 1 (no friction). A typical momentum value is 0.9.

*Equation 11-4. Momentum algorithm*

1. $\mathbf{m} \leftarrow \beta\mathbf{m} + \eta\nabla_\theta J(\theta)$
2. $\theta \leftarrow \theta - \mathbf{m}$

You can easily verify that if the gradient remains constant, the terminal velocity (i.e., the maximum size of the weight updates) is equal to that gradient multiplied by the learning rate $\eta$ multiplied by $\frac{1}{1-\beta}$. For example, if $\beta = 0.9$, then the terminal velocity is equal to 10 times the gradient times the learning rate, so Momentum optimization ends up going 10 times faster than Gradient Descent! This allows Momentum optimization to escape from plateaus much faster than Gradient Descent. In particular, we saw in Chapter 4 that when the inputs have very different scales the cost function will look like an elongated bowl (see Figure 4-7). Gradient Descent goes down the steep slope quite fast, but then it takes a very long time to go down the valley. In contrast, Momentum optimization will roll down the bottom of the valley faster and faster until it reaches the bottom (the optimum). In deep neural networks that don't use Batch Normalization, the upper layers will often end up having inputs with very

11 "Some methods of speeding up the convergence of iteration methods," B. Polyak (1964).