we will present the most popular ones: Momentum optimization, Nesterov Accelerated Gradient, AdaGrad, RMSProp, and finally Adam and Nadam optimization.

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.[12] 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 $\boldsymbol{\theta}$ by directly subtracting the gradient of the cost function $J(\boldsymbol{\theta})$ with regards to the weights ($\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$) multiplied by the learning rate η. The equation is: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$. 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 subtracts the local gradient from the *momentum vector* $\mathbf{m}$ (multiplied by the learning rate η), and it updates the weights by simply adding this momentum vector (see Equation 11-4). In other words, the gradient is used for acceleration, not for speed. To simulate some sort of friction mechanism and prevent the momentum from growing too large, the algorithm introduces a new hyperparameter β, 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. $\mathbf{m} \leftarrow \beta \mathbf{m} - \eta \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$
2. $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \mathbf{m}$

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 η multiplied by $\frac{1}{1-\beta}$ (ignoring the sign). 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 val-

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

ley. In contrast, Momentum optimization will roll down 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 different scales, so using Momentum optimization helps a lot. It can also help roll past local optima.

Due to the momentum, the optimizer may overshoot a bit, then come back, overshoot again, and oscillate like this many times before stabilizing at the minimum. This is one of the reasons why it is good to have a bit of friction in the system: it gets rid of these oscillations and thus speeds up convergence.

Implementing Momentum optimization in Keras is a no-brainer: just use the `SGD` optimizer and set its `momentum` hyperparameter, then lie back and profit!

```
optimizer = keras.optimizers.SGD(lr=0.001, momentum=0.9)
```

The one drawback of Momentum optimization is that it adds yet another hyperparameter to tune. However, the momentum value of 0.9 usually works well in practice and almost always goes faster than regular Gradient Descent.

Nesterov Accelerated Gradient

One small variant to Momentum optimization, proposed by Yurii Nesterov in 1983,[13] is almost always faster than vanilla Momentum optimization. The idea of *Nesterov Momentum optimization*, or *Nesterov Accelerated Gradient* (NAG), is to measure the gradient of the cost function not at the local position but slightly ahead in the direction of the momentum (see Equation 11-5). The only difference from vanilla Momentum optimization is that the gradient is measured at $\boldsymbol{\theta} + \beta\mathbf{m}$ rather than at $\boldsymbol{\theta}$.

Equation 11-5. Nesterov Accelerated Gradient algorithm

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

This small tweak works because in general the momentum vector will be pointing in the right direction (i.e., toward the optimum), so it will be slightly more accurate to use the gradient measured a bit farther in that direction rather than using the gradient at the original position, as you can see in Figure 11-6 (where ∇_1 represents the gradient of the cost function measured at the starting point $\boldsymbol{\theta}$, and ∇_2 represents the

13 "A Method for Unconstrained Convex Minimization Problem with the Rate of Convergence $O(1/k^2)$," Yurii Nesterov (1983).

gradient at the point located at $\boldsymbol{\theta} + \beta\mathbf{m}$). As you can see, the Nesterov update ends up slightly closer to the optimum. After a while, these small improvements add up and NAG ends up being significantly faster than regular Momentum optimization. Moreover, note that when the momentum pushes the weights across a valley, ∇_1 continues to push further across the valley, while ∇_2 pushes back toward the bottom of the valley. This helps reduce oscillations and thus converges faster.

NAG will almost always speed up training compared to regular Momentum optimization. To use it, simply set `nesterov=True` when creating the `SGD` optimizer:

```
optimizer = keras.optimizers.SGD(lr=0.001, momentum=0.9, nesterov=True)
```

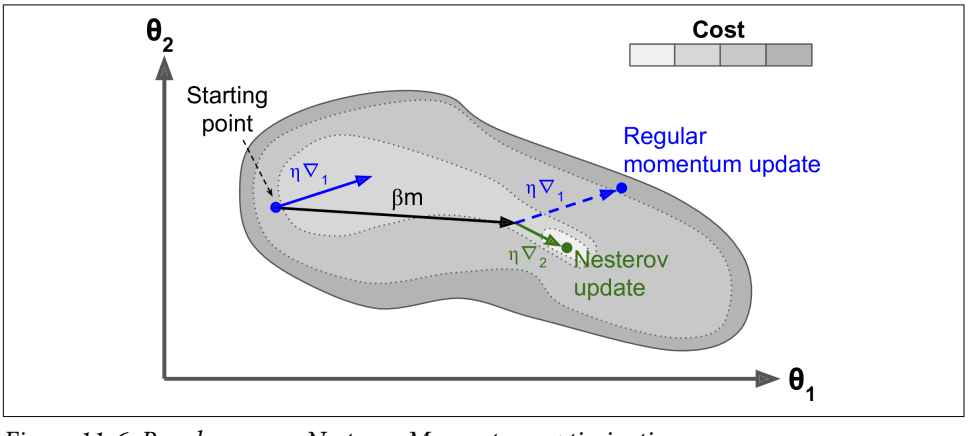

Figure 11-6. Regular versus Nesterov Momentum optimization

AdaGrad

Consider the elongated bowl problem again: Gradient Descent starts by quickly going down the steepest slope, then slowly goes down the bottom of the valley. It would be nice if the algorithm could detect this early on and correct its direction to point a bit more toward the global optimum.

The *AdaGrad* algorithm[14] achieves this by scaling down the gradient vector along the steepest dimensions (see Equation 11-6):

Equation 11-6. AdaGrad algorithm

1. $\mathbf{s} \leftarrow \mathbf{s} + \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \otimes \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$
2. $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \oslash \sqrt{\mathbf{s} + \epsilon}$

14 "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization," J. Duchi et al. (2011).

The first step accumulates the square of the gradients into the vector **s** (recall that the $\otimes$ symbol represents the element-wise multiplication). This vectorized form is equivalent to computing $s_i \leftarrow s_i + (\partial J(\mathbf{\theta}) / \partial \theta_i)^2$ for each element s_i of the vector **s**; in other words, each s_i accumulates the squares of the partial derivative of the cost function with regards to parameter θ_i. If the cost function is steep along the ith dimension, then s_i will get larger and larger at each iteration.

The second step is almost identical to Gradient Descent, but with one big difference: the gradient vector is scaled down by a factor of $\sqrt{\mathbf{s} + \epsilon}$ (the $\oslash$ symbol represents the element-wise division, and ϵ is a smoothing term to avoid division by zero, typically set to 10^{-10}). This vectorized form is equivalent to computing $\theta_i \leftarrow \theta_i - \eta \, \partial J(\mathbf{\theta}) / \partial \theta_i / \sqrt{s_i + \epsilon}$ for all parameters θ_i (simultaneously).

In short, this algorithm decays the learning rate, but it does so faster for steep dimensions than for dimensions with gentler slopes. This is called an *adaptive learning rate*. It helps point the resulting updates more directly toward the global optimum (see Figure 11-7). One additional benefit is that it requires much less tuning of the learning rate hyperparameter η.

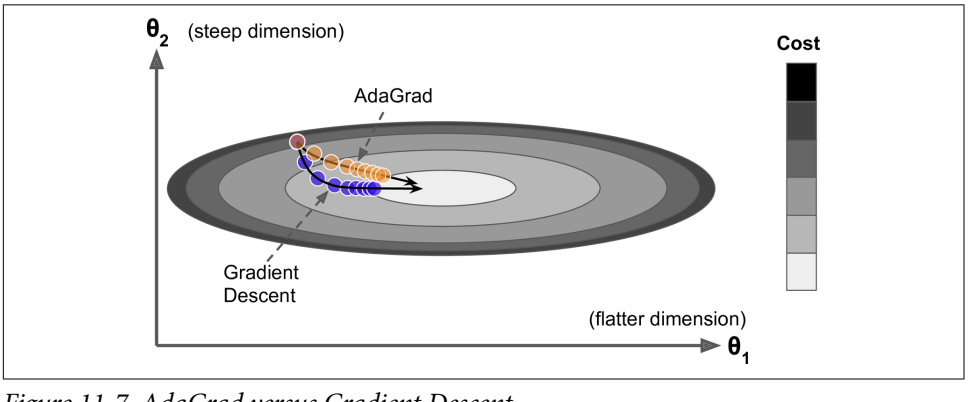

Figure 11-7. AdaGrad versus Gradient Descent

AdaGrad often performs well for simple quadratic problems, but unfortunately it often stops too early when training neural networks. The learning rate gets scaled down so much that the algorithm ends up stopping entirely before reaching the global optimum. So even though Keras has an `Adagrad` optimizer, you should not use it to train deep neural networks (it may be efficient for simpler tasks such as Linear Regression, though). However, understanding Adagrad is helpful to grasp the other adaptive learning rate optimizers.

RMSProp

Although AdaGrad slows down a bit too fast and ends up never converging to the global optimum, the *RMSProp* algorithm[15] fixes this by accumulating only the gradients from the most recent iterations (as opposed to all the gradients since the beginning of training). It does so by using exponential decay in the first step (see Equation 11-7).

Equation 11-7. RMSProp algorithm

1. $\mathbf{s} \leftarrow \beta \mathbf{s} + (1 - \beta) \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \otimes \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$
2. $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \oslash \sqrt{\mathbf{s} + \epsilon}$

The decay rate β is typically set to 0.9. Yes, it is once again a new hyperparameter, but this default value often works well, so you may not need to tune it at all.

As you might expect, Keras has an `RMSProp` optimizer:

```
optimizer = keras.optimizers.RMSprop(lr=0.001, rho=0.9)
```

Except on very simple problems, this optimizer almost always performs much better than AdaGrad. In fact, it was the preferred optimization algorithm of many researchers until Adam optimization came around.

Adam and Nadam Optimization

Adam,[16] which stands for *adaptive moment estimation*, combines the ideas of Momentum optimization and RMSProp: just like Momentum optimization it keeps track of an exponentially decaying average of past gradients, and just like RMSProp it keeps track of an exponentially decaying average of past squared gradients (see Equation 11-8).[17]

15 This algorithm was created by Geoffrey Hinton and Tijmen Tieleman in 2012, and presented by Geoffrey Hinton in his Coursera class on neural networks (slides: *https://homl.info/57*; video: *https://homl.info/58*). Amusingly, since the authors did not write a paper to describe it, researchers often cite "slide 29 in lecture 6" in their papers.

16 "Adam: A Method for Stochastic Optimization," D. Kingma, J. Ba (2015).

17 These are estimations of the mean and (uncentered) variance of the gradients. The mean is often called the *first moment*, while the variance is often called the *second moment*, hence the name of the algorithm.

Equation 11-8. Adam algorithm

1. $\mathbf{m} \leftarrow \beta_1 \mathbf{m} - (1 - \beta_1)\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$
2. $\mathbf{s} \leftarrow \beta_2 \mathbf{s} + (1 - \beta_2)\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \otimes \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$
3. $\widehat{\mathbf{m}} \leftarrow \dfrac{\mathbf{m}}{1 - \beta_1^{\,t}}$
4. $\widehat{\mathbf{s}} \leftarrow \dfrac{\mathbf{s}}{1 - \beta_2^{\,t}}$
5. $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \eta\, \widehat{\mathbf{m}} \oslash \sqrt{\widehat{\mathbf{s}} + \epsilon}$

- t represents the iteration number (starting at 1).

If you just look at steps 1, 2, and 5, you will notice Adam's close similarity to both Momentum optimization and RMSProp. The only difference is that step 1 computes an exponentially decaying average rather than an exponentially decaying sum, but these are actually equivalent except for a constant factor (the decaying average is just $1 - \beta_1$ times the decaying sum). Steps 3 and 4 are somewhat of a technical detail: since **m** and **s** are initialized at 0, they will be biased toward 0 at the beginning of training, so these two steps will help boost **m** and **s** at the beginning of training.

The momentum decay hyperparameter β_1 is typically initialized to 0.9, while the scaling decay hyperparameter β_2 is often initialized to 0.999. As earlier, the smoothing term ϵ is usually initialized to a tiny number such as 10^{-7}. These are the default values for the `Adam` class (to be precise, `epsilon` defaults to `None`, which tells Keras to use `keras.backend.epsilon()`, which defaults to 10^{-7}; you can change it using `keras.backend.set_epsilon()`).

```
optimizer = keras.optimizers.Adam(lr=0.001, beta_1=0.9, beta_2=0.999)
```

Since Adam is an adaptive learning rate algorithm (like AdaGrad and RMSProp), it requires less tuning of the learning rate hyperparameter η. You can often use the default value $\eta = 0.001$, making Adam even easier to use than Gradient Descent.

If you are starting to feel overwhelmed by all these different techniques, and wondering how to choose the right ones for your task, don't worry: some practical guidelines are provided at the end of this chapter.

Finally, two variants of Adam are worth mentioning:

- Adamax, introduced in the same paper as Adam: notice that in step 2 of Equation 11-8, Adam accumulates the squares of the gradients in **s** (with a greater weight for more recent weights). In step 5, if we ignore ϵ and steps 3 and 4 (which are technical details anyway), Adam just scales down the parameter updates by the square root of **s**. In short, Adam scales down the parameter updates by the ℓ_2 norm of the time-decayed gradients (recall that the ℓ_2 norm is the square root of the sum of squares). Adamax just replaces the ℓ_2 norm with the ℓ_∞ norm (a fancy way of saying the max). Specifically, it replaces step 2 in Equation 11-8 with $\mathbf{s} \leftarrow \max\left(\beta_2 \mathbf{s}, \nabla_\theta J(\theta)\right)$, it drops step 4, and in step 5 it scales down the gradient updates by a factor of **s**, which is just the max of the time-decayed gradients. In practice, this can make Adamax more stable than Adam, but this really depends on the dataset, and in general Adam actually performs better. So it's just one more optimizer you can try if you experience problems with Adam on some task.
- Nadam optimization[18] is more important: it is simply Adam optimization plus the Nesterov trick, so it will often converge slightly faster than Adam. In his report, Timothy Dozat compares many different optimizers on various tasks, and finds that Nadam generally outperforms Adam, but is sometimes outperformed by RMSProp.

Adaptive optimization methods (including RMSProp, Adam and Nadam optimization) are often great, converging fast to a good solution. However, a 2017 paper[19] by Ashia C. Wilson et al. showed that they can lead to solutions that generalize poorly on some datasets. So when you are disappointed by your model's performance, try using plain Nesterov Accelerated Gradient instead: your dataset may just be allergic to adaptive gradients. Also check out the latest research, it is moving fast (e.g., AdaBound).

All the optimization techniques discussed so far only rely on the *first-order partial derivatives* (*Jacobians*). The optimization literature contains amazing algorithms based on the *second-order partial derivatives* (the *Hessians*, which are the partial derivatives of the Jacobians). Unfortunately, these algorithms are very hard to apply to deep neural networks because there are n^2 Hessians per output (where n is the number of parameters), as opposed to just n Jacobians per output. Since DNNs typically have tens of thousands of parameters, the second-order optimization algorithms

18 "Incorporating Nesterov Momentum into Adam," Timothy Dozat (2015).

19 "The Marginal Value of Adaptive Gradient Methods in Machine Learning," A. C. Wilson et al. (2017).

often don't even fit in memory, and even when they do, computing the Hessians is just too slow.

Training Sparse Models

All the optimization algorithms just presented produce dense models, meaning that most parameters will be nonzero. If you need a blazingly fast model at runtime, or if you need it to take up less memory, you may prefer to end up with a sparse model instead.

One trivial way to achieve this is to train the model as usual, then get rid of the tiny weights (set them to 0). However, this will typically not lead to a very sparse model, and it may degrade the model's performance.

A better option is to apply strong ℓ_1 regularization during training, as it pushes the optimizer to zero out as many weights as it can (as discussed in Chapter 4 about Lasso Regression).

However, in some cases these techniques may remain insufficient. One last option is to apply *Dual Averaging*, often called *Follow The Regularized Leader* (FTRL), a technique proposed by Yurii Nesterov.[20] When used with ℓ_1 regularization, this technique often leads to very sparse models. Keras implements a variant of FTRL called *FTRL-Proximal*[21] in the `FTRL` optimizer.

Learning Rate Scheduling

Finding a good learning rate can be tricky. If you set it way too high, training may actually diverge (as we discussed in Chapter 4). If you set it too low, training will eventually converge to the optimum, but it will take a very long time. If you set it slightly too high, it will make progress very quickly at first, but it will end up dancing around the optimum, never really settling down. If you have a limited computing budget, you may have to interrupt training before it has converged properly, yielding a suboptimal solution (see Figure 11-8).

20 "Primal-Dual Subgradient Methods for Convex Problems," Yurii Nesterov (2005).

21 "Ad Click Prediction: a View from the Trenches," H. McMahan et al. (2013).

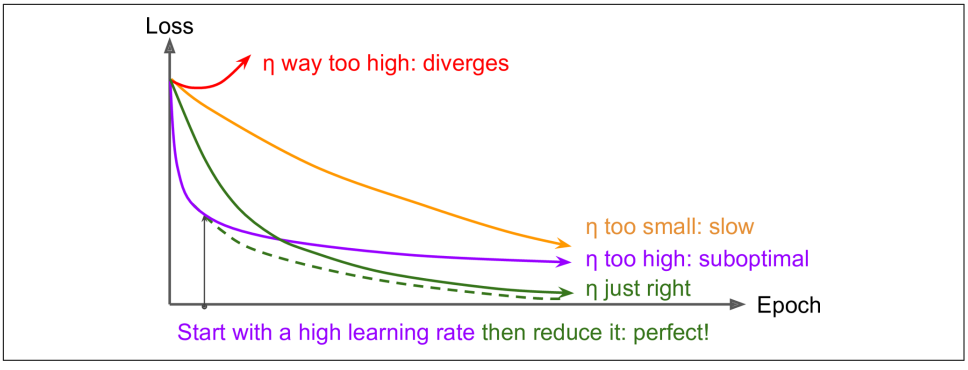

Figure 11-8. Learning curves for various learning rates η

As we discussed in Chapter 10, one approach is to start with a large learning rate, and divide it by 3 until the training algorithm stops diverging. You will not be too far from the optimal learning rate, which will learn quickly and converge to good solution.

However, you can do better than a constant learning rate: if you start with a high learning rate and then reduce it once it stops making fast progress, you can reach a good solution faster than with the optimal constant learning rate. There are many different strategies to reduce the learning rate during training. These strategies are called *learning schedules* (we briefly introduced this concept in Chapter 4), the most common of which are:

Power scheduling
: Set the learning rate to a function of the iteration number t: $\eta(t) = \eta_0 / (1 + t/k)^c$. The initial learning rate η_0, the power c (typically set to 1) and the steps s are hyperparameters. The learning rate drops at each step, and after s steps it is down to $\eta_0 / 2$. After s more steps, it is down to $\eta_0 / 3$. Then down to $\eta_0 / 4$, then $\eta_0 / 5$, and so on. As you can see, this schedule first drops quickly, then more and more slowly. Of course, this requires tuning η_0, s (and possibly c).

Exponential scheduling
: Set the learning rate to: $\eta(t) = \eta_0 \, 0.1^{t/s}$. The learning rate will gradually drop by a factor of 10 every s steps. While power scheduling reduces the learning rate more and more slowly, exponential scheduling keeps slashing it by a factor of *10* every s steps.

Piecewise constant scheduling
: Use a constant learning rate for a number of epochs (e.g., $\eta_0 = 0.1$ for 5 epochs), then a smaller learning rate for another number of epochs (e.g., $\eta_1 = 0.001$ for 50 epochs), and so on. Although this solution can work very well, it requires fid-

dling around to figure out the right sequence of learning rates, and how long to use each of them.

Performance scheduling
: Measure the validation error every N steps (just like for early stopping) and reduce the learning rate by a factor of λ when the error stops dropping.

A 2013 paper[22] by Andrew Senior et al. compared the performance of some of the most popular learning schedules when training deep neural networks for speech recognition using Momentum optimization. The authors concluded that, in this setting, both performance scheduling and exponential scheduling performed well. They favored exponential scheduling because it was easy to tune and it converged slightly faster to the optimal solution (they also mentioned that it was easier to implement than performance scheduling, but in Keras both options are easy).

Implementing power scheduling in Keras is the easiest option: just set the `decay` hyperparameter when creating an optimizer. The `decay` is the inverse of s (the number of steps it takes to divide the learning rate by one more unit), and Keras assumes that c is equal to 1. For example:

```
optimizer = keras.optimizers.SGD(lr=0.01, decay=1e-4)
```

Exponential scheduling and piecewise scheduling are quite simple too. You first need to define a function that takes the current epoch and returns the learning rate. For example, let's implement exponential scheduling:

```
def exponential_decay_fn(epoch):
    return 0.01 * 0.1**(epoch / 20)
```

If you do not want to hard-code η_0 and s, you can create a function that returns a configured function:

```
def exponential_decay(lr0, s):
    def exponential_decay_fn(epoch):
        return lr0 * 0.1**(epoch / s)
    return exponential_decay_fn

exponential_decay_fn = exponential_decay(lr0=0.01, s=20)
```

Next, just create a `LearningRateScheduler` callback, giving it the schedule function, and pass this callback to the `fit()` method:

```
lr_scheduler = keras.callbacks.LearningRateScheduler(exponential_decay_fn)
history = model.fit(X_train_scaled, y_train, [...], callbacks=[lr_scheduler])
```

22 "An Empirical Study of Learning Rates in Deep Neural Networks for Speech Recognition," A. Senior et al. (2013).