

8.3 Basic Algorithms

We have previously introduced the gradient descent (section 4.3) algorithm that follows the gradient of an entire training set downhill. This may be accelerated considerably by using stochastic gradient descent to follow the gradient of randomly selected minibatches downhill, as discussed in section 5.9 and section 8.1.3.

8.3.1 Stochastic Gradient Descent

Stochastic gradient descent (SGD) and its variants are probably the most used optimization algorithms for machine learning in general and for deep learning in particular. As discussed in section 8.1.3, it is possible to obtain an unbiased estimate of the gradient by taking the average gradient on a minibatch of m examples drawn i.i.d from the data generating distribution.

Algorithm 8.1 shows how to follow this estimate of the gradient downhill.

Algorithm 8.1 Stochastic gradient descent (SGD) update at training iteration k

Require: Learning rate ϵ_k .

Require: Initial parameter θ

while stopping criterion not met **do**

Sample a minibatch of m examples from the training set $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$ with corresponding targets $\mathbf{y}^{(i)}$.

Compute gradient estimate: $\hat{\mathbf{g}} \leftarrow +\frac{1}{m} \nabla_{\theta} \sum_i L(f(\mathbf{x}^{(i)}; \theta), \mathbf{y}^{(i)})$

Apply update: $\theta \leftarrow \theta - \epsilon \hat{\mathbf{g}}$

end while

A crucial parameter for the SGD algorithm is the learning rate. Previously, we have described SGD as using a fixed learning rate ϵ . In practice, it is necessary to gradually decrease the learning rate over time, so we now denote the learning rate at iteration k as ϵ_k .

This is because the SGD gradient estimator introduces a source of noise (the random sampling of m training examples) that does not vanish even when we arrive at a minimum. By comparison, the true gradient of the total cost function becomes small and then $\mathbf{0}$ when we approach and reach a minimum using batch gradient descent, so batch gradient descent can use a fixed learning rate. A sufficient condition to guarantee convergence of SGD is that

$$\sum_{k=1}^{\infty} \epsilon_k = \infty, \quad \text{and} \quad (8.12)$$

$$\sum_{k=1}^{\infty} \epsilon_k^2 < \infty. \quad (8.13)$$

In practice, it is common to decay the learning rate linearly until iteration τ :

$$\epsilon_k = (1 - \alpha)\epsilon_0 + \alpha\epsilon_\tau \quad (8.14)$$

with $\alpha = \frac{k}{\tau}$. After iteration τ , it is common to leave ϵ constant.

The learning rate may be chosen by trial and error, but it is usually best to choose it by monitoring learning curves that plot the objective function as a function of time. This is more of an art than a science, and most guidance on this subject should be regarded with some skepticism. When using the linear schedule, the parameters to choose are ϵ_0 , ϵ_τ , and τ . Usually τ may be set to the number of iterations required to make a few hundred passes through the training set. Usually ϵ_τ should be set to roughly 1% the value of ϵ_0 . The main question is how to set ϵ_0 . If it is too large, the learning curve will show violent oscillations, with the cost function often increasing significantly. Gentle oscillations are fine, especially if training with a stochastic cost function such as the cost function arising from the use of dropout. If the learning rate is too low, learning proceeds slowly, and if the initial learning rate is too low, learning may become stuck with a high cost value. Typically, the optimal initial learning rate, in terms of total training time and the final cost value, is higher than the learning rate that yields the best performance after the first 100 iterations or so. Therefore, it is usually best to monitor the first several iterations and use a learning rate that is higher than the best-performing learning rate at this time, but not so high that it causes severe instability.

The most important property of SGD and related minibatch or online gradient-based optimization is that computation time per update does not grow with the number of training examples. This allows convergence even when the number of training examples becomes very large. For a large enough dataset, SGD may converge to within some fixed tolerance of its final test set error before it has processed the entire training set.

To study the convergence rate of an optimization algorithm it is common to measure the **excess error** $J(\boldsymbol{\theta}) - \min_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$, which is the amount that the current cost function exceeds the minimum possible cost. When SGD is applied to a convex problem, the excess error is $O(\frac{1}{\sqrt{k}})$ after k iterations, while in the strongly convex case it is $O(\frac{1}{k})$. These bounds cannot be improved unless extra conditions are assumed. Batch gradient descent enjoys better convergence rates than stochastic gradient descent in theory. However, the Cramér-Rao bound (Cramér, 1946; Rao, 1945) states that generalization error cannot decrease faster than $O(\frac{1}{k})$. Bottou

and Bousquet (2008) argue that it therefore may not be worthwhile to pursue an optimization algorithm that converges faster than $O(\frac{1}{k})$ for machine learning tasks—faster convergence presumably corresponds to overfitting. Moreover, the asymptotic analysis obscures many advantages that stochastic gradient descent has after a small number of steps. With large datasets, the ability of SGD to make rapid initial progress while evaluating the gradient for only very few examples outweighs its slow asymptotic convergence. Most of the algorithms described in the remainder of this chapter achieve benefits that matter in practice but are lost in the constant factors obscured by the $O(\frac{1}{k})$ asymptotic analysis. One can also trade off the benefits of both batch and stochastic gradient descent by gradually increasing the minibatch size during the course of learning.

For more information on SGD, see Bottou (1998).

8.3.2 Momentum

While stochastic gradient descent remains a very popular optimization strategy, learning with it can sometimes be slow. The method of momentum (Polyak, 1964) is designed to accelerate learning, especially in the face of high curvature, small but consistent gradients, or noisy gradients. The momentum algorithm accumulates an exponentially decaying moving average of past gradients and continues to move in their direction. The effect of momentum is illustrated in figure 8.5.

Formally, the momentum algorithm introduces a variable \mathbf{v} that plays the role of velocity—it is the direction and speed at which the parameters move through parameter space. The velocity is set to an exponentially decaying average of the negative gradient. The name **momentum** derives from a physical analogy, in which the negative gradient is a force moving a particle through parameter space, according to Newton’s laws of motion. Momentum in physics is mass times velocity. In the momentum learning algorithm, we assume unit mass, so the velocity vector \mathbf{v} may also be regarded as the momentum of the particle. A hyperparameter $\alpha \in [0, 1)$ determines how quickly the contributions of previous gradients exponentially decay. The update rule is given by:

$$\mathbf{v} \leftarrow \alpha \mathbf{v} - \epsilon \nabla_{\boldsymbol{\theta}} \left(\frac{1}{m} \sum_{i=1}^m L(\mathbf{f}(\mathbf{x}^{(i)}; \boldsymbol{\theta}), \mathbf{y}^{(i)}) \right), \quad (8.15)$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \mathbf{v}. \quad (8.16)$$

The velocity \mathbf{v} accumulates the gradient elements $\nabla_{\boldsymbol{\theta}} \left(\frac{1}{m} \sum_{i=1}^m L(\mathbf{f}(\mathbf{x}^{(i)}; \boldsymbol{\theta}), \mathbf{y}^{(i)}) \right)$. The larger α is relative to ϵ , the more previous gradients affect the current direction. The SGD algorithm with momentum is given in algorithm 8.2.

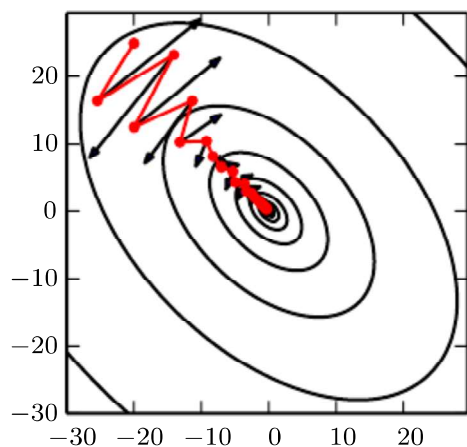


Figure 8.5: Momentum aims primarily to solve two problems: poor conditioning of the Hessian matrix and variance in the stochastic gradient. Here, we illustrate how momentum overcomes the first of these two problems. The contour lines depict a quadratic loss function with a poorly conditioned Hessian matrix. The red path cutting across the contours indicates the path followed by the momentum learning rule as it minimizes this function. At each step along the way, we draw an arrow indicating the step that gradient descent would take at that point. We can see that a poorly conditioned quadratic objective looks like a long, narrow valley or canyon with steep sides. Momentum correctly traverses the canyon lengthwise, while gradient steps waste time moving back and forth across the narrow axis of the canyon. Compare also figure 4.6, which shows the behavior of gradient descent without momentum.

Previously, the size of the step was simply the norm of the gradient multiplied by the learning rate. Now, the size of the step depends on how large and how aligned a *sequence* of gradients are. The step size is largest when many successive gradients point in exactly the same direction. If the momentum algorithm always observes gradient \mathbf{g} , then it will accelerate in the direction of $-\mathbf{g}$, until reaching a terminal velocity where the size of each step is

$$\frac{\epsilon \|\mathbf{g}\|}{1 - \alpha}. \quad (8.17)$$

It is thus helpful to think of the momentum hyperparameter in terms of $\frac{1}{1-\alpha}$. For example, $\alpha = .9$ corresponds to multiplying the maximum speed by 10 relative to the gradient descent algorithm.

Common values of α used in practice include .5, .9, and .99. Like the learning rate, α may also be adapted over time. Typically it begins with a small value and is later raised. It is less important to adapt α over time than to shrink ϵ over time.

Algorithm 8.2 Stochastic gradient descent (SGD) with momentum

Require: Learning rate ϵ , momentum parameter α .

Require: Initial parameter $\boldsymbol{\theta}$, initial velocity \mathbf{v} .

while stopping criterion not met **do**

 Sample a minibatch of m examples from the training set $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$ with corresponding targets $\mathbf{y}^{(i)}$.

 Compute gradient estimate: $\mathbf{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\mathbf{x}^{(i)}; \boldsymbol{\theta}), \mathbf{y}^{(i)})$

 Compute velocity update: $\mathbf{v} \leftarrow \alpha \mathbf{v} - \epsilon \mathbf{g}$

 Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \mathbf{v}$

end while

We can view the momentum algorithm as simulating a particle subject to continuous-time Newtonian dynamics. The physical analogy can help to build intuition for how the momentum and gradient descent algorithms behave.

The position of the particle at any point in time is given by $\boldsymbol{\theta}(t)$. The particle experiences net force $\mathbf{f}(t)$. This force causes the particle to accelerate:

$$\mathbf{f}(t) = \frac{\partial^2}{\partial t^2} \boldsymbol{\theta}(t). \quad (8.18)$$

Rather than viewing this as a second-order differential equation of the position, we can introduce the variable $\mathbf{v}(t)$ representing the velocity of the particle at time t and rewrite the Newtonian dynamics as a first-order differential equation:

$$\mathbf{v}(t) = \frac{\partial}{\partial t} \boldsymbol{\theta}(t), \quad (8.19)$$

$$\mathbf{f}(t) = \frac{\partial}{\partial t} \mathbf{v}(t). \quad (8.20)$$

The momentum algorithm then consists of solving the differential equations via numerical simulation. A simple numerical method for solving differential equations is Euler's method, which simply consists of simulating the dynamics defined by the equation by taking small, finite steps in the direction of each gradient.

This explains the basic form of the momentum update, but what specifically are the forces? One force is proportional to the negative gradient of the cost function: $-\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$. This force pushes the particle downhill along the cost function surface. The gradient descent algorithm would simply take a single step based on each gradient, but the Newtonian scenario used by the momentum algorithm instead uses this force to alter the velocity of the particle. We can think of the particle as being like a hockey puck sliding down an icy surface. Whenever it descends a steep part of the surface, it gathers speed and continues sliding in that direction until it begins to go uphill again.

One other force is necessary. If the only force is the gradient of the cost function, then the particle might never come to rest. Imagine a hockey puck sliding down one side of a valley and straight up the other side, oscillating back and forth forever, assuming the ice is perfectly frictionless. To resolve this problem, we add one other force, proportional to $-\mathbf{v}(t)$. In physics terminology, this force corresponds to viscous drag, as if the particle must push through a resistant medium such as syrup. This causes the particle to gradually lose energy over time and eventually converge to a local minimum.

Why do we use $-\mathbf{v}(t)$ and viscous drag in particular? Part of the reason to use $-\mathbf{v}(t)$ is mathematical convenience—an integer power of the velocity is easy to work with. However, other physical systems have other kinds of drag based on other integer powers of the velocity. For example, a particle traveling through the air experiences turbulent drag, with force proportional to the square of the velocity, while a particle moving along the ground experiences dry friction, with a force of constant magnitude. We can reject each of these options. Turbulent drag, proportional to the square of the velocity, becomes very weak when the velocity is small. It is not powerful enough to force the particle to come to rest. A particle with a non-zero initial velocity that experiences only the force of turbulent drag will move away from its initial position forever, with the distance from the starting point growing like $O(\log t)$. We must therefore use a lower power of the velocity. If we use a power of zero, representing dry friction, then the force is too strong. When the force due to the gradient of the cost function is small but non-zero, the constant force due to friction can cause the particle to come to rest before reaching a local minimum. Viscous drag avoids both of these problems—it is weak enough