Mathematical Engineering of Deep Learning

Book Draft

Benoit Liquet, Sarat Moka and Yoni Nazarathy

February 28, 2024

Contents

4 Optimization Algorithms - DRAFT

The field of optimization is an important sub-field of applied mathematics. In the context of deep learning it is just as important. Any form of training process for a deep learning model requires optimization to minimize the loss. The decision variables are the learned model parameters, and the data is typically considered fixed from the point of view of optimization. In this chapter we explore general optimization methods and results, focusing on the essential tools for optimization in the process of training deep learning models.

In Section 4.1 we formulate optimization problems in a general setup, discuss their forms, and review local extrema, global extrema, and convexity. In Section 4.2 we specialize to optimization of learned parameters for deep learning. We discuss the nature of such problems, and common techniques including stochastic gradient descent, tracking performance measures, and early stopping. In Section 4.3 we discuss various forms of first-order methods which extend the basic gradient descent algorithm. The focus is on the popular ADAM optimizer which has grown to become very popular for deep learning. In Section 4.4 we introduce automatic differentiation, a computational paradigm for efficient evaluation of derivatives. Here we present both forward and backward mode automatic differentiation in a general context, and later in the next chapter we specialize to deep learning. We continue with Section 4.5 where additional first-order methods are presented beyond gradient descent and ADAM. In Section 4.6 we introduce and present an overview of second-order methods. These powerful methods are sometimes used in deep learning, and when applicable they can perform very well. Note that on a first reading of the book, one may focus on Sections 4.1, 4.2, 4.3, and 4.4 before continuing to understand the deep learning models of Chapter 5.

4.1 Formulation of Optimization

Optimization algorithms, such as gradient descent described in Algorithm 2.1 of Chapter 2, primarily aim to minimize an objective. We have already seen in previous chapters that "learning" $\approx$ "optimization". Hence to do well in deep learning, it is important to understand the implementation of optimization algorithms as they play a crucial role in minimizing the training loss.

The General Setup

Consider a multivariate real-valued function $C : \mathbb{R}^d \to \mathbb{R}$. An optimization problem can be written as

$$\begin{aligned} &\underset{\theta}{\text{minimize}}\ C(\theta) \\ &\text{subject to } \theta \in \Theta, \end{aligned} \tag{4.1}$$

where $\Theta \subseteq \mathbb{R}^d$ is called the *feasible set* or the *constraint set*. In the context of learning, Θ represents the parameter space and the *objective function* $C(\theta)$ is the loss function.

In other words, the aim of the optimization problem is to find a point $\theta \in \Theta$ where $C(\theta)$ is minimum. Such a point, denoted θ^*, is called a *solution* or *minimizer* of the optimization problem (4.1). That is,

$$C(\theta^*) \leq C(\theta), \quad \text{for all } \theta \in \Theta. \tag{4.2}$$

There can be multiple solutions to this problem and the set of all the solutions θ^* is usually denoted by

$$\operatorname*{argmin}_{\theta \in \Theta} C(\theta).$$

The formulation (4.1) is general, in the sense that any optimization problem can be written in this form. Particularly, the maximization problem $\operatorname{maximize}_\theta C(\theta)$ subject to $\theta \in \Theta$ can be stated in the form (4.1) by replacing $C(\theta)$ in (4.1) with $-C(\theta)$. We say that the optimization problem is *unconstrained* if $\Theta = \mathbb{R}^d$; otherwise, it is a *constrained* optimization problem.

Our focus is primarily on unconstrained optimization and hence the "subject to" component of (4.1) may be omitted. This is the common case in the context of deep learning, and in particular it is the case for the optimization problems that we already encountered in the context of linear regression (Section 2.3), logistic regression (sections 3.1 and 3.2), multinomial regression (Section 3.3), and most cases of autoencoders (Section 3.5).

General optimization theory and practice comes in many shapes and forms where different methods can be used for different sub-types of the general problem (4.1). In the context of deep learning we typically make use of multivariate calculus[1] and assume (or construct) the objective $C(\cdot)$ to have desirable properties, such as continuity, differentiability (almost everywhere), and other smoothness properties. With this type of optimization, the gradient ∇C and Hessian $\nabla^2 C$ are typically used. The former is the key component of first-order methods as described in sections 4.3 and 4.5. The latter sometimes plays a role in second-order methods as described in Section 4.6. Throughout this chapter when we make use of the gradient or Hessian then we are implicitly assuming that these objects exist and are well defined for the problem at hand.

Local and Global Minima

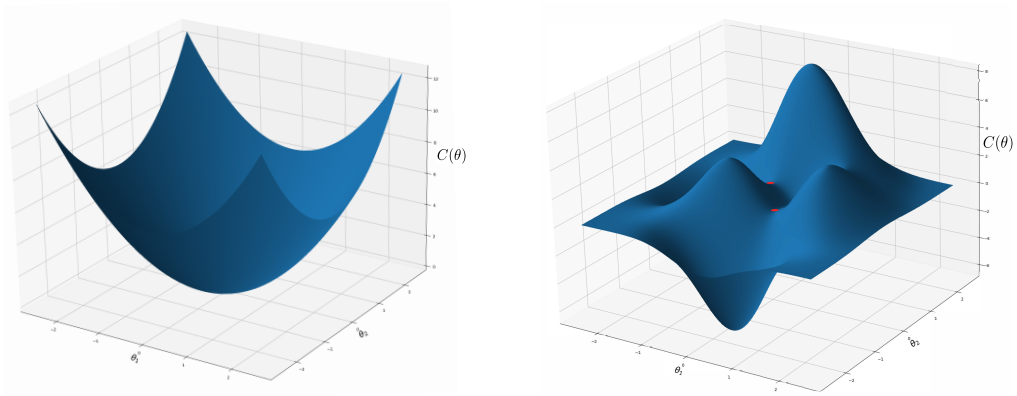
A solution of an unconstrained minimization problem is also known as a *global minimum* because of (4.2). However, there can be other points that are not solutions to the optimization problem but exhibit similar properties within their vicinity. Such points are known as local minima. In particular, a point $\theta \in \mathbb{R}^d$ is called a *local minimum* of a continuous function C if there is an $r > 0$ such that $C(\theta) \leq C(\phi)$ for all $\phi \in B(\theta, r)$, where $B(\theta, r)$ is an open ball with center θ and radius $r > 0$ defined by

$$B(\theta, r) = \{\phi \in \mathbb{R}^d : \|\theta - \phi\| < r\}.$$

[1] Refer to Appendix A for a review.

In other words, if θ is a local minimum of C, we cannot decrease the value of C by moving an infinitesimal distance from θ in any direction; see Figure 4.1 for illustrations of global and local minima.

Suppose that the objective function C is differentiable at a local minimum θ. Then, it is well known that the gradient $\nabla C(\theta) = 0$. This is a necessary condition for a point to be a local minimum, but not sufficient. For example, the condition $\nabla C(\theta) = 0$ also holds if θ is a *local maximum* of C, that is, θ is a local minimum of $-C$. Further the condition $\nabla C(\theta) = 0$ can hold even at a point θ that is neither a local minimum nor a local maximum, and such a point is called a *saddle point* where the objective function slopes up in one direction and slopes down in another direction (for smooth functions we define it below using the Hessian matrix). See Figure 4.1 for examples.



(a) **(b)**

Figure 4.1: Two dimensional smooth functions. (a) The function $C(\theta) = \theta_1^2 + \theta_2^2$ is convex with a unique global minimum. (b) A well-known function called the *peaks function* is an example non-convex function with several local minima, local maxima, and saddle points. Around each local minimum, the function is locally convex. On the figure, two saddle points are marked with red dots.

Convexity and Saddle Points

There are classes of optimization problems for which a local minimum is also guaranteed to be a global minimum. One such popular class is the family of convex problems. While most deep learning based optimization problems are not convex, understanding convexity helps to better understand optimization concepts.

We can establish stronger necessary conditions for a point to be a local minimum via *convexity*. A set $\Theta_{\mathcal{C}} \subseteq \mathbb{R}^d$ is said to be a *convex set* if for any $\theta, \phi \in \Theta_{\mathcal{C}}$ and $\lambda \in [0, 1]$,

$$\lambda\theta + (1 - \lambda)\phi \in \Theta_{\mathcal{C}}.$$

That is, for a convex set, the line segment connecting any two points in the set, remains in the set. An example of a convex set is an open ball $B(\theta, r)$ for any $\theta \in \mathbb{R}^d$ and $r > 0$.

We are now ready to define *convex functions*. Suppose that $\Theta_{\mathcal{C}} \subseteq \mathbb{R}^d$ is a convex set. Then, a function $C : \mathbb{R}^d \to \mathbb{R}$ is called a *convex function* on $\Theta_{\mathcal{C}}$ if for every $\theta, \phi \in \Theta_{\mathcal{C}}$ and $\lambda \in [0, 1]$,

we have

$$C\big(\lambda\theta + (1-\lambda)\phi\big) \leq \lambda C(\theta) + (1-\lambda)C(\phi). \tag{4.3}$$

Furthermore, we say that the function C is *strictly convex* on $\Theta_{\mathcal{C}}$ if the inequality in (4.3) holds with strict inequality for all $\lambda \in (0,1)$ and for every choice of $\theta, \phi \in \Theta_{\mathcal{C}}$ with $\theta \neq \phi$.

If the function C is convex on the entire space $\mathbb{R}^d$, then all the local minima are global minima. Furthermore, if C is strictly convex on $\mathbb{R}^d$, then there is a unique global minimum. See Figure 4.1 (a) for an example of a convex function.

If we further assume that C is twice differentiable, then we can define convexity and strict convexity in terms of the Hessian matrix. In particular, C is convex on $\Theta_{\mathcal{C}}$ if and only if the Hessian $\nabla^2 C(\theta)$ is positive semidefinite[2] at every interior point on $\Theta_{\mathcal{C}}$. Furthermore, C is strictly convex on $\Theta_{\mathcal{C}}$ if and only if the Hessian $\nabla^2 C(\theta)$ is positive definite at every interior point on $\Theta_{\mathcal{C}}$.

Note that if a point θ is a local minimum of a twice differentiable function C, then there exists some $r > 0$ such that C is convex on the open ball $B(\theta, r)$, and hence we sometimes say that C is *locally convex* around θ. See Figure 4.1 (b) for an example of a function with several local minima with local convexity around each minimum point.

For twice differentiable functions, it is easy to define *saddle points*. A point θ with $\nabla C(\theta) = 0$ is called a saddle point if the Hessian $\nabla^2 C(\theta)$ at θ is neither positive semidefinite nor negative semidefinite.

Objective Functions in Deep Learning

In the context of deep learning, the typical form of the objective function is

$$C(\theta) := C(\theta\,;\,\mathcal{D}) = \frac{1}{n}\sum_{i=1}^{n} C_i(\theta), \tag{4.4}$$

where n is the number of data samples in a given dataset $\mathcal{D}$. We have already seen this form in (2.11) where $C_i(\theta)$ is the loss computed for the i-th data sample. Specific forms for this i-th data sample loss include the square loss, cross entropy loss, and other forms that have already appeared in chapters 2 and 3. In this chapter the data samples play a more minor role and hence from now onwards in this chapter we suppress $\mathcal{D}$ from the notation and use $C(\theta)$.

The optimization problem that corresponds to the objective function (4.4) is often called the *finite sum problem*. For such problems, it is relatively easy to study the properties of $C(\theta)$ by studying the corresponding properties of each $C_i(\theta)$, its gradient $\nabla C_i(\theta)$ and Hessian $\nabla^2 C_i(\theta)$. Most importantly, the form (4.4) plays an important role in finding the descent direction for optimization methods such as stochastic gradient descent, described in Section 4.2. When each $C_i(\theta)$ is twice differentiable, the gradient and Hessian of the

[2] See (A.12) in Appendix A for definition of positive semidefinite and positive definite matrices.

objective $C(\theta)$ are respectively given by

$$\nabla C(\theta) = \frac{1}{n}\sum_{i=1}^{n} \nabla C_i(\theta), \quad \text{and} \quad \nabla^2 C(\theta) = \frac{1}{n}\sum_{i=1}^{n} \nabla^2 C_i(\theta). \tag{4.5}$$

A local minimum of $C(\cdot)$ at θ can be characterized via $\nabla^2 C(\theta)$ being positive semidefinite in addition to $\nabla C(\theta) = 0$. It is useful to note that $\nabla^2 C(\theta)$ is positive semidefinite at θ if each $\nabla^2 C_i(\theta)$ is positive semidefinite at θ, because for any $\phi \in \mathbb{R}^d$,

$$\phi^\top \nabla^2 C(\theta)\phi = \frac{1}{n}\sum_{i=1}^{n} \left(\phi^\top \nabla^2 C_i(\theta)\phi\right). \tag{4.6}$$

Convexity of Certain Shallow Neural Networks

While most deep learning optimization problems are not convex, the loss functions of the logistic regression and multinomial regression models of Chapter 3 are convex. The same is true for the linear regression model of Chapter 2. For completeness, we now establish convexity properties of these simple models. The details of this subsection may be skipped on a first reading.

We begin with the linear regression model of Section 2.3 with the standard quadratic loss. In this case the associated optimization problem is (2.13) and we may compute the Hessian for the objective $C(\theta) = \|y - X\theta\|^2$ to be

$$\nabla^2 C(\theta) = 2X^\top X,$$

where we use the notation of the design matrix X from (2.10). The Hessian is thus a Gram matrix and this implies it is positive semidefinite.[3] Hence the optimization problem is convex. Further if X has linearly independent columns then $\nabla^2 C(\theta)$ is positive definite and the problem is strictly convex and has a unique optimal point.

We now continue to logistic regression of sections 3.1 and 3.2. An expression for the gradient is in (3.17) and we may follow up with an expression for the Hessian. Specifically, the $d \times d$ Hessian matrix of each $C_i(\theta)$ is

$$\nabla^2 C_i(\theta) = \sigma_{\text{Sig}}(b + w^\top x^{(i)})\left(1 - \sigma_{\text{Sig}}(b + w^\top x^{(i)})\right) \begin{bmatrix} 1 \\ x^{(i)} \end{bmatrix} \begin{bmatrix} 1 & x^{(i)\top} \end{bmatrix}, \tag{4.7}$$

which is evidently a product of a strictly positive scalar and a Gram matrix. Thus it is always positive semidefinite following the same argument used for linear regression with quadratic loss.[4] Now using the argument at (4.6), the matrix $\nabla^2 C(\theta)$ is also positive semidefinite and thus the total loss function $C(\theta)$ is convex. Further, in cases where the vectors $(1, x^{(i)})$ for $i = 1, \ldots, n$ span $\mathbb{R}^d$ (this can only hold when $d \leq n$) then we cannot find a non-zero vector $\phi \in \mathbb{R}^d$ orthogonal to all the vectors $(1, x^{(i)})$. Thus, for any vector $\phi \neq 0$, there exists at least one i such that $\phi^\top \nabla^2 C_i(\theta)\,\phi > 0$ and hence $\nabla^2 C(\theta)$ is positive definite, making the total loss function $C(\theta)$ strictly convex.

[3] For any matrix A, the associated Gram matrix is $A^\top A$ and thus $\phi^\top A^\top A\phi = \|A\phi\|^2 \geq 0$ for any vector ϕ. Further with linearly independent columns $A\phi \neq 0$ for any $\phi \neq 0$ and thus $\|A\phi\|^2 > 0$ implying that A is positive definite.

[4] Note however that $\nabla^2 C_i(\theta)$ is never (strictly) positive definite because $\nabla^2 C_i(\theta)$ is a rank one matrix.

We now move onto multinomial regression as well as linear regression with other loss functions introduced in Section 2.3. The optimization problems of these models also enjoy convexity properties, yet expressions of the Hessian are not easy to work with. We thus use other means to argue for convexity.

For multinomial regression recall $C_i(\theta)$ from (3.33) which we can represent as,

$$C_i(\theta) = \log \sum_{j=1}^{K} e^{b_j + w_{(j)}^\top x^{(i)}} - (b_k + w_{(k)}^\top x^{(i)}), \tag{4.8}$$

when $y^{(i)} = k$. Recall here that θ includes both bias terms (scalars) $b_1, \ldots, b_K$ and weight vectors $w_{(1)}, \ldots, w_{(K)}$; see (3.23).

As a building block we first use the *log-sum-exp* function, $h : \mathbb{R}^K \to \mathbb{R}$ of the form

$$h(v) = \log \sum_{j=1}^{K} e^{v_j}.$$

We can show that $h(v)$ is convex by explicitly calculating its Hessian expression and seeing that it is positive semidefinite[5] for all $v \in \mathbb{R}^K$. Further using first principles of the definition of convexity, (4.3), we can show that for any convex function $h : \mathbb{R}^K \to \mathbb{R}$, $h(\theta^\top u_1, \ldots, \theta^\top u_K)$ is convex in $\theta \in \mathbb{R}^d$ for any fixed $u_1, \ldots, u_K \in \mathbb{R}^d$. Since θ is the vector consisting of all the weights and the biases, by using multiple zeros in each $u_j \in \mathbb{R}^d$, we can construct u_j from the p-dimensional vector $x^{(i)}$ such that

$$v_j = \theta^\top u_j = b_j + w_{(j)}^\top x^{(i)}.$$

As a consequence, the first term of (4.8) is convex in θ. Further, the second term is also convex as it is an affine function of θ. Hence (4.8) is a sum of convex functions and is thus convex.

We now again consider linear regression, but this time with the absolute error loss (2.18) and Huber loss (2.19). In both of these cases, like the standard quadratic loss, we have,

$$C_i(\theta) = g(y^{(i)} - \theta^\top x^{(i)}),$$

where $g : \mathbb{R} \to \mathbb{R}$ is a convex function; see Figure 2.6 (a) in Chapter 2. Here again we use the fact that composition of a convex function and an affine function is convex and this establishes convexity of $C_i(\theta)$ and thus of $C(\theta)$ for all these loss function alternatives.

Before we depart from this discussion of convexity we suggest that the reader consider the plots of Figure 3.4 of Chapter 3. In that figure, it is evident that logistic regression with "the wrong" loss function, namely squared loss in this case, may yield a non-convex problem. We highlight this here to emphasize that the convexity properties discussed above are more of an exception than the rule. Indeed, most deep learning models of this book have non-convex objectives.

[5] The gradient of the log-sum-exp function is the softmax function for which we present the Jacobian in (5.19), and the Jacobian of the gradient of a function is equal to the Hessian of that function. It also turns out that the matrix in (5.19) is the covariance matrix of a categorical distribution (multinomial distribution with one trial) implying that it is positive semidefinite.

General Approach of Descent Direction Methods

We have seen that the loss functions of logistic regression and multinomial regression are convex, and thus they have global minima. Unfortunately, even for these shallow neural networks, despite being convex, there is no known analytical solution for the minimizer. Further, in more general deep neural networks, we do not have such convexity properties and the objective functions typically exhibit numerous local minima. Thus in general, it is impossible to think of an analytical solution for the optimization problems in deep learning, and thus we have to relay on more advanced numerical optimization methods.

All the popular optimization methods used in deep learning can be put under one general framework called the *descent direction method*. It is an iterative method that attempts to reduce the objective in each iteration. This might seem obvious, however the reader should keep in mind that in general optimization methods beyond those covered in this book, steps may sometimes increase the objective before further reduction. In contrast, with the descent direction method, while increases may occur in certain iterations, the overall goal is to have a reduction of the objective in each iteration. This method is presented in Algorithm 4.1.

Algorithm 4.1: The descent direction method

Input: Dataset $\mathcal{D} = \{(x^{(1)}, y^{(1)}), \ldots, (x^{(n)}, y^{(n)})\}$,
objective function $C(\cdot) = C(\cdot\,; \mathcal{D})$, and
initial parameter vector θ_{init}

Output: Approximately *optimal* θ

$\theta \leftarrow \theta_{\mathsf{init}}$
repeat
  Determine the *descent step* θ_{s}
  $\theta \leftarrow \theta + \theta_{\mathsf{s}}$
until *termination condition* is satisfied
return θ

As is evident, Algorithm 4.1 relies on the specific way in which we compute the descent step and the termination condition. For each iteration $t = 0, 1, 2, \ldots$ of Algorithm 4.1, denote the values of the parameter vector θ and the descent step θ_{s} by $\theta^{(t)}$ and $\theta_{\mathsf{s}}^{(t)}$, respectively. With this notation, the sequence of points $\theta^{(1)}, \theta^{(2)}, \ldots$, evolves via,

$$\theta^{(t+1)} = \theta^{(t)} + \theta_{\mathsf{s}}^{(t)}, \qquad \text{with} \qquad \theta^{(0)} = \theta_{\mathsf{init}},$$

until $\theta^{(t)}$ satisfies the specified termination condition. Note that the magnitude of the descent step $\|\theta_{\mathsf{s}}^{(t)}\|$ denotes the *step size* in the t-th iteration.

The descent step $\theta_{\mathsf{s}}^{(t)}$ at the t-th iteration is determined by information available in or until the iteration t, where the exact manner in which it is computed depends on the variant of the algorithm used. One specific example that we have already seen is the gradient descent method, Algorithm 2.1 of Section 2.4, which is a special case of Algorithm 4.1. With gradient descent, the descent step is set at $\theta_{\mathsf{s}} = -\alpha \nabla C(\theta)$ where α is the learning rate and since $\alpha > 0$, at each step the gradient descent method moves along the steepest descent direction.[6]

[6]Note that some authors use the term *steepest descent* to denote a more general class of algorithms of which gradient descent can be taken as a special case. However, in this book "steepest descent" and "gradient descent" are considered synonymous.

Other variants encountered in the sequel are stochastic gradient descent (SGD) presented in Section 4.2, ADAM presented in Section 4.3, second-order methods presented in Section 4.6, and other variants.

When the goal is to minimize the objective function C, ideally, we would like to terminate the algorithm only after reaching a minimum point. This convergence is difficult to guarantee in general as the updates of the algorithm can slow down or oscillate in the vicinity of the minimum. Therefore, in practice there are several termination conditions that allow some tolerance so that the algorithm can output a point in the vicinity of the minimum. We now list a few traditionally popular termination criteria. However, note that the next section presents optimization in the context of deep learning where other criteria are often used.

Most termination conditions use a small tolerance value, $\varepsilon > 0$. One standard termination condition tracks *absolute improvement*, where we terminate when the change in the function value is smaller than the tolerance. Namely,

$$\left|C(\theta^{(t)}) - C(\theta^{(t+1)})\right| < \varepsilon.$$

An alternative is to use the *gradient magnitude* or step size where we terminate respectively if,

$$\|\nabla C(\theta^{(t+1)})\| < \varepsilon, \qquad \text{or} \qquad \|\theta_{\mathsf{s}}^{(t)}\| < \varepsilon.$$

When the region around the (local) minimum is shallow, it is sometimes useful to use the *relative improvement criterion* where we terminate based on the relative change in the function value. Specifically, we terminate if

$$\frac{|C(\theta^{(t)}) - C(\theta^{(t+1)})|}{|C(\theta^{(t)})|} < \varepsilon.$$

In addition to these criteria which track the state of the optimization procedure, an alternative is to simply run for a fixed number of iterations $t = 0, \ldots, T$, or similarly to limit the running (clock) time of the algorithm to a desired predefined value.

Similarly, the *termination condition* at Step 5 of Algorithm 4.1 can be different for different methods. The *optimal* output point obtained by the algorithm may not necessarily minimize the objective function $C(\cdot)$. Rather it depends on the termination condition. For instance, as we see in Section 4.2, the final θ returned by Algorithm 4.1 could be a minimizer of the validation set accuracy, but not a minimizer of the objective function $C(\cdot)$.

4.2 Optimization in the Context of Deep Learning

In the previous section we saw that certain optimization problems are tractable in the sense that a local minimum is also the global minimum. Specifically, the loss functions of linear regression, logistic regression, and multinomial regression are convex. However, in most deep learning models, starting with the models that we cover in the next chapter, the loss functions are not as "blessed". For such deep learning models, the number of parameters is huge and the problems are non-convex, often having a large number of local minima.

Nevertheless, basic optimization techniques are very useful for deep learning with the compromise of not finding the global minimum of the loss, but rather finding a point close to some local minimum with satisfactory performance. That is, the obtained point has

performance which is in general not far off from the performance at the global minimum. Indeed, in deep learning, we use optimization algorithms trying to minimize the training loss function $C(\theta)$ while our ultimate goal is finding θ that has good performance on the unseen data; see Section 2.5 for a discussion of performance on unseen data. That is, the goal is to actually minimize expectations or averages of performance functions[7] $\mathcal{P}(\theta)$, such as the validation set error rate. We then hope that this minimization resembles performance on unseen data. Hence, as the reader studies the optimization methods of this chapter, they should keep in mind that minimization of the loss $C(\theta)$ is often a proxy for minimization of $\mathcal{P}(\theta)$.

Challenges Posed by Basic Gradient Descent

Almost all cases where optimization is used for deep learning involve some variant of gradient descent; see Algorithm 2.1 of Section 2.4 for an introduction to gradient descent. However, basic gradient descent poses a few important challenges that can make the algorithm impractical in the deep learning context.

The primary challenge is that in each iteration, the gradient descent algorithm uses the entire dataset to compute the gradient $\nabla C(\theta)$. In the context of deep learning, the sample size n of the training data can be large or the dimension d of the parameter θ can be huge, or both. As a consequence, the operation of computing the gradient using the entire dataset for every update can be very slow, or even intractable in terms of memory requirements.

Further, as already mentioned above, loss functions of deep neural networks tend to be highly non-convex, with multiple local minima. For such functions, the apriori selection of a good learning rate α is very difficult. A low learning rate can lead to an increased computational cost as a result of slow convergence of the algorithm, and it can lead to the algorithm getting stuck at local minima or flat sections of the loss landscape, eliminating the chance to explore other local minima. On the other hand, a large learning rate can result in the objective function values and decision variables fluctuating in an haphazard manner.

Even for simple convex models, as we saw in Figure 2.7 of Section 2.4, the choice of the learning rate affects learning performance. As a second elementary example with $d = 2$, assume hypothetically that the loss landscape takes the form of the following Rosenbrock function,

$$C(\theta) = (1 - \theta_1)^2 + 100(\theta_2 - \theta_1^2)^2, \tag{4.9}$$

which is non-convex yet has a unique global minimum at $\theta = (1, 1)$.

Figure 4.2 illustrates the performance of gradient descent on (4.9). Here we also introduce the option of varying the learning rate as optimization iterations progress – based on a *predefined learning rate schedule* in this case. Specifically, at each iteration t, instead of using the fixed learning rate α we use $\alpha^{(t)}$ which is indexed by iteration t and parameterized by the original α. This strategy is sometimes used in practice, yet as we see in this example, it does not work well for the parameters that we choose, and in none of the four trajectories starting at an initial point θ_{init}, do we find the minimum. Note that in this specific example it is not hard to fine-tune the learning rate α and/or the *exponential decay parameter* of

[7] Note that in Section 2.5 we denoted the performance measure function as a function of $\hat{y}$ and y for each observation in $\mathcal{D}$ where as here we are taking it as a function of the parameters θ.

$\alpha^{(t)}$, which is set at 0.99. However, this is a simple, hypothetical, $d = 2$ dimensional loss landscape, whereas in actual problems with $d > 2$, tuning $\alpha^{(t)}$ to achieve efficient learning (with basic gradient descent) is often very difficult or impossible.

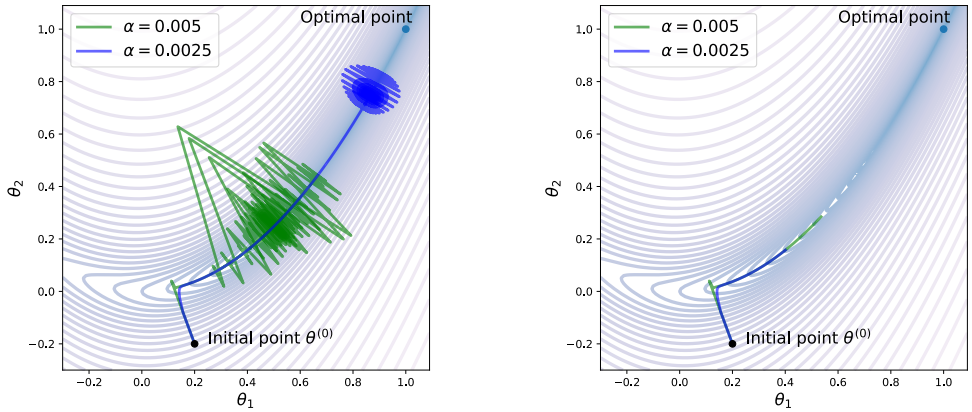


Figure 4.2: Attempting minimization of a Rosenbrock function with a minimum at $\theta = (1, 1)$ via basic gradient descent for two values of α with 10^5 iterations and $\theta_{\text{init}} = (0.2, -0.2)$. (a) Fixed learning rate: $\alpha^{(t)} = \alpha$. (b) Exponentially decaying learning rate: $\alpha^{(t)} = \alpha\, 0.99^t$.

Instead of adjusting the learning rate according to a predefined schedule, other variants try to adjust the learning rate according to other criteria such as the objective function crossing some threshold values. Further, in some implementations, $\alpha^{(t)}$ is optimized in each iteration, so that the function $C(\theta)$ decreases maximally in the direction of the step size; see for example line search methods described in Section 4.5. However, such strategies are again not fool-proof on their own, or are often not practical for deep learning.

Furthermore, even if a magical ("optimal") learning rate α can be suggested for each iteration, the basic gradient descent method applies the same learning rate to all parameter updates. It turns out that it is better to learn the parameters $\theta_1, \ldots, \theta_d$ at different rates in an adaptive manner because larger rates on rarely changing parameter coordinates may result in faster convergence of the algorithm. To handle such cases, it is common practice to use the ADAM algorithm. This variant of gradient descent, together with its building blocks, are the focus of Section 4.3.

Prior to exploring ADAM and related gradient descent adaptations, let us return to the key issue mentioned above which is the computational complexity of obtaining exact gradients $\nabla C(\theta)$. This issue is typically handled by computing approximate gradients either via stochastic gradient descent, which we describe now, or more practically with the use of mini-batches described in the sequel. Importantly, both of these approaches yield fast computable approximate gradients. These approaches also yield parameter trajectories that manage to escape local minima, flat regions, or saddle points by introducing randomness or noise into the optimization process.

Stochastic Gradient Descent

Recall that the loss function $C(\theta)$ for deep neural networks is typically a finite sum of the form (4.4) with $C_i(\theta)$ being the loss computed for the i-th data sample. The method of *stochastic gradient descent* exploits this structure by computing a noisy gradient using only one randomly selected training sample. That is, it evaluates the gradient for a single $C_i(\theta)$ instead of the gradient for all of $C(\theta)$.

More precisely, the algorithm operates similarly to Algorithm 2.1 (or variations of it), yet in the t-th iteration of stochastic gradient descent, an index variable I_t is randomly selected from the set $\{1, \dots, n\}$ and the gradient $\nabla C_{I_t}(\theta)$ is computed only for the I_t-th data sample in place of Step 3 of Algorithm 2.1. The update rule for the decision variable is then,

$$\theta^{(t+1)} = \theta^{(t)} - \alpha \, \nabla C_{I_t}(\theta^{(t)}). \tag{4.10}$$

This stochastic algorithm exhibits some interesting properties. Most importantly, the execution of each iteration can be much faster than that of basic gradient descent because only one sample is used at a time (the gain is of order n). Further, if the index variable I_t is selected uniformly over $\{1, \dots, n\}$, the parameter update (4.10) is *unbiased* in the sense that the expected descent step for each iteration t is the same as that of gradient descent. That is, in each iteration, we have

$$\mathbb{E}\left[\nabla C_{I_t}(\theta)\right] = \frac{1}{n} \sum_{i=1}^{n} \nabla C_i(\theta) = \nabla C(\theta), \qquad \text{for a fixed } \theta. \tag{4.11}$$

and hence the stochastic step (4.10) is "on average correct".[8]

While being unbiased, the stochastic nature of (4.10) introduces fluctuations in the descent direction. Such noisy trajectories may initially appear like a drawback, yet one advantage is that they enable the algorithm to escape flat regions, saddle points, and local minima. Thus, while the fluctuations make it difficult to guarantee convergence, in the context of deep learning their effect is often considered desirable in view of highly non-convex loss landscapes.

Importantly, stochastic gradient descent is generally able to direct the parameters θ towards the region where minima of $C_i(\theta)$ are located. To get a feel for this attribute of stochastic gradient descent, we consider another hypothetical example where for $i = 1, \dots, n$,

$$C_i(\theta) = a_i(\theta_1 - u_{i,1})^2 + (\theta_2 - u_{i,2})^2, \tag{4.12}$$

for some constant $a_i > 0$ and $u_i = (u_{i,1}, u_{i,2}) \in \mathbb{R}^2$. Here u_i is the unique minimizer of $C_i(\theta)$. These individual loss functions for each observation are convex and hence the total loss function $C(\theta)$ of (4.4) is also convex.[9] Let $\mathcal{S}$ be the *convex hull* of $u_1, \dots, u_n$, that is, $\mathcal{S}$ is the smallest convex set that contains $\{u_1, \dots, u_n\}$. It is now obvious that the global minima of C is also in $\mathcal{S}$.

[8] The reader might be tempted to conclude that $\mathbb{E}[\theta^{(t)}_{\text{SGD}}] = \theta^{(t)}_{\text{GD}}$ for all time t, where the subscript SGD is for stochastic gradient descent starting at the same initial condition as GD ("Gradient Descent"). However, in general this is not correct when $\nabla C(\theta)$ is non-linear in θ.

[9] This example does not resemble a general case with multiple local minima since it is convex. Nevertheless, it is useful for understanding some of the behaviour of stochastic gradient descent.

In Figure 4.3 we plot the contours of such a function when $n = 5$ where in Figure 4.3 (a) the points $u_1, \ldots, u_5$ are distinct and in (b) these points are identical. The convex hull is marked by the region bounded in green. The figure also plots the evolution of gradient descent and stochastic gradient descent in each of the cases. As is evident, while the stochastic gradient descent trajectory is noisier, outside of the convex hull $\mathcal{S}$, its trajectory is similar to that of gradient descent.

Interestingly, in Figure 4.3 (a), once the trajectory hits $\mathcal{S}$ it mostly stays within $\mathcal{S}$ but with a lot of fluctuations. The reason for this behavior is that at any point θ not in $\mathcal{S}$, the descent directions for both gradient descent and stochastic gradient descent are towards the set $\mathcal{S}$ as it contains the minima of $C(\theta)$ and every $C_i(\theta)$. To make this a concrete argument, note that the descent step in the t-th iteration of the stochastic gradient descent is

$$-\alpha \nabla C_{I_t}(\theta^{(t)}) = -\alpha \sum_{i=1}^{n} \mathbf{1}\{I_t = i\} \nabla C_i(\theta^{(t)}).$$

Since every $C_i(\theta)$ has its minimum in the set $\mathcal{S}$, if $\theta^{(t)} \notin \mathcal{S}$, then every $-\nabla C_i(\theta^{(t)})$, the steepest direction of $C_i(\theta)$ at $\theta^{(t)}$, guides the algorithm towards the set $\mathcal{S}$. Therefore, irrespective of the choice of I_t, the update moves $\theta^{(t+1)}$ towards $\mathcal{S}$.

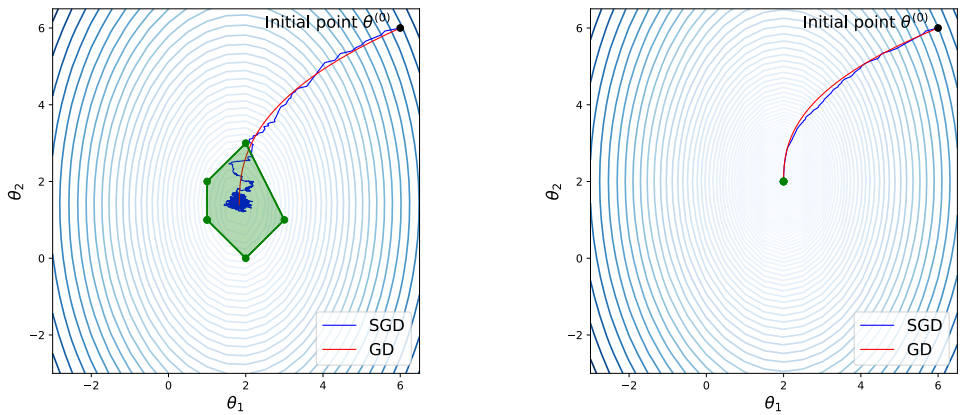


Figure 4.3: An hypothetical example where $C(\theta)$ is composed of individual $C_i(\theta)$ as in (4.12). Comparison of gradient descent (GD) and stochastic gradient descent (SGD). (a) A case where the minimizers $u_1, \ldots, u_n$ of $C_i(\theta)$ are different and are each marked by green dots in the plot. (b) A case where the minimizers $u_1, \ldots, u_n$ are the same.

Working with Mini-batches and the Concept of Epochs

On one extreme, computing the full gradient for basic gradient descent is computationally challenging but the obtained gradients are noiseless. On the other extreme, stochastic gradient descent reduces the computational complexity of each update but the updates are noisy. One middle ground approach is to work with mini-batches. Almost any practical deep learning training uses some variant of this approach.

A *mini-batch* is simply a small subset of the data points, of size n_b. Mini-batch sizes are typically quite small in comparison to the size of the dataset and are often chosen based on computational capabilities and constraints. In practice, a mini-batch size n_b is often selected so the computation of gradients associated with n_b observations can fit on GPU memory. This generally makes computation more efficient in comparison to stochastic gradient descent which cannot take full advantage of parallel computing or GPUs since the gradient evaluation for a single observation index I_t is not easy to parallelize.

When using mini-batches, for each iteration t, a mini-batch with indices $\mathcal{I}_t \subseteq \{1, \ldots, n\}$ is used to approximate the gradient via,

$$\frac{1}{n_b} \sum_{i \in \mathcal{I}_t} \nabla C_i(\theta^{(t)}). \tag{4.13}$$

This *estimated gradient* is then used as part of gradient descent or one of its variants (such as ADAM which we present in the sequel); for example it is used in place of Step 3 of Algorithm 2.1. One common practical approach with mini-batches is to shuffle the training data apriori and then use the shuffled indices sequentially. With this process there are n/n_b mini-batches[10] in the training dataset and each mini-batch is a distinct random subset of the indices. In such a case, when using some variant of gradient descent, every pass on the full dataset has n/n_b iterations, with one iteration per mini-batch. Such a pass on all the data via n/n_b iterations is typically called an *epoch*. The training process then involves multiple epochs. It is quite common to diagnose and track the training process in terms of epochs. Note that one may also randomly shuffle the data (assign new mini-batches) after every epoch to reduce the correlation between the epochs and reduce bias in the optimization process.

Similarly to (4.11) when using the mini-batch approach, the estimated gradient is unbiased in the sense that

$$\mathbb{E}\left[\frac{1}{n_b}\sum_{j=1}^{n_b} \nabla C_{I_{t,j}}(\theta)\right] = \frac{1}{n_b}\sum_{j=1}^{n_b}\frac{1}{n}\sum_{i=1}^{n}\nabla C_i(\theta) = \nabla C(\theta), \qquad \text{for a fixed } \theta.$$

Here $I_{t,j}$ denotes the j-th element in $\mathcal{I}_t$ and the first equality holds because shuffling makes each $I_{t,j}$ to be uniform over $\{1, 2, \ldots, n\}$. To get a feel for the performance of learning with mini-batches, a mathematical simplification is to consider a variant where $\mathcal{I}_t = (I_{t,1}, \ldots, I_{t,n_b})$ is a collection of randomly selected indices on $\{1, \ldots, n\}$ with replacement.[11] As a consequence, each $I_{t,j}$ is not only uniform over $\{1, 2, \ldots, n\}$, but also independent of all the rest. This manner of computing gradients is an approximation to the above mentioned mini-batches with shuffling the data after every epoch.

Importantly, in this setup, we notice that the noise of the gradients decreases as the mini-batch size n_b increases. To see this, recall that for each t, the index variables $I_{t,1}, \ldots, I_{t,n_b}$ are chosen independently. Thus, the variance of the gradient estimate is

$$\mathbb{V}\text{ar}\left(\frac{1}{n_b}\sum_{j=1}^{n_b}\nabla C_{I_{t,j}}(\theta^{(t)})\right) = \frac{1}{n_b}\mathbb{V}\text{ar}\left(\nabla C_{I_{t,1}}(\theta^{(t)})\right).$$

[10] We assume here that n_b divides n. If it is not the case then there are $\lceil n/n_b \rceil$ mini-batches with the last one having less than n_b samples.

[11] Some authors refer to this as *mini-batch gradient descent*.

Since $\mathbb{V}\text{ar}\left(\nabla C_{I_{t,1}}(\theta^{(t)})\right)$ is the variance of the gradient for an arbitrary data sample, we see that mini-batch gradient descent has n_b times smaller variance than that of stochastic gradient descent.

Loss Minimization is a Proxy for Optimal Performance

Recall the discussion in Section 2.5 where we saw that the ultimate focus of learning is to optimize the expected performance on unseen data. This is typically quantified in terms of some error metric which we wish to be minimal. In this section we simply refer to the estimated expected performance function as $\mathcal{P}(\theta)$, keeping in mind that it is typically an average over the validation set. When training, it is often more tractable to minimize the loss $C(\theta)$, however, the true goal is minimization of $\mathcal{P}(\theta)$.

In deep learning, the model choice and training process typically involves choosing a model with a loss function $C(\theta)$, such that loss minimization is computationally easy to carry out and yields low $\mathcal{P}(\theta)$. In that sense, the loss function $C(\theta)$ is often much more amenable to derivative computations and gradient based learning, in comparison to trying to use the performance function $\mathcal{P}(\theta)$ directly. Further, by tracking a performance function[12] on a validation dataset not used for training, we are often able to make sure that overtraining (also called overfitting) is avoided. This interplay of the loss function $C(\theta)$ and the performance function $\mathcal{P}(\theta)$ implies that deep learning optimization is not traditional optimization where one seeks θ^* that truly optimizes the loss, but is rather a means of using optimization of $C(\theta)$ as a surrogate to good performance on $\mathcal{P}(\theta)$.

We now explore this setting of deep learning optimization in two ways. In one direction we discuss a notion of stopping criteria that differs from the termination conditions presented at the end of Section 4.1. In another direction we highlight that the actual model choice affects the loss function $C(\theta)$, and some loss functions are more amenable to optimization in comparison to others.

First, in terms of stopping criteria, as already mentioned above, for training a deep learning model, standard practice is to use mini-batches and epochs. This means that some form of the general Algorithm 4.1 is executed where multiple iterations are grouped into an epoch. As training progresses (this is sometimes a process of hours, days, or more), we track the evolution of the loss,[13] $C(\theta^{(t)})$, as well as one or more performance functions $\mathcal{P}(\theta^{(t)})$, typically focusing on iterations t that are at the end of an epoch. In that sense, a common stopping criterion is to stop at the epoch where $\mathcal{P}(\theta^{(t)})$ is small when evaluated over the validation set.

A popular strategy for this approach is called *early stopping*. With this strategy, training often continues for multiple epochs beyond the epoch where $\mathcal{P}(\theta^{(t)})$ is minimized since at that epoch it is still not evident that the minimum was reached. However, as the training loop progresses, the model parameters for the best epoch are stored such that once training is complete the "best model" in terms of $\mathcal{P}(\theta)$ on the validation set can be used. A typical trajectory of the loss and performance metric is displayed in Figure 4.4. With such a typical trajectory, increasing the number of epochs of training is somewhat similar to the increase of model complexity discussed in Section 2.5; see Figure 2.9 of that section. Running the model beyond the minimum point of $\mathcal{P}(\theta)$ is called *over-training*, similar to *overfitting* discussed in

[12] In practice we may often track multiple performance functions.
[13] In this context it is often called the *training loss* as it is the loss over the training set.

Section 2.5. The early stopping strategy is then a method to use validation performance to try and avoid such over-training. Importantly, the actual minimum point, say θ^*, of $C(\cdot)$ is not a desirable point for this purpose since it is typically a point of over-training and $\mathcal{P}(\theta^*)$ is sub-optimal in terms of the performance function.

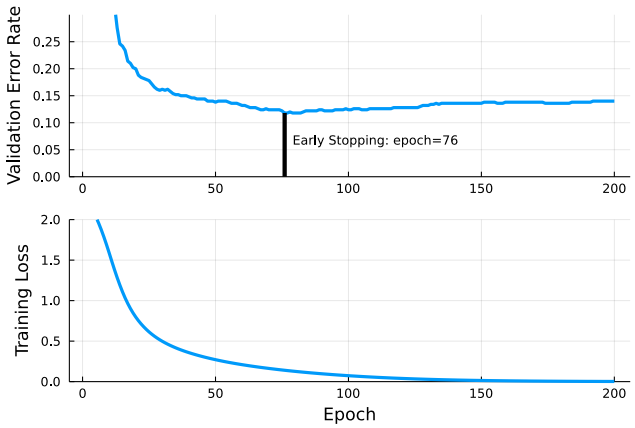
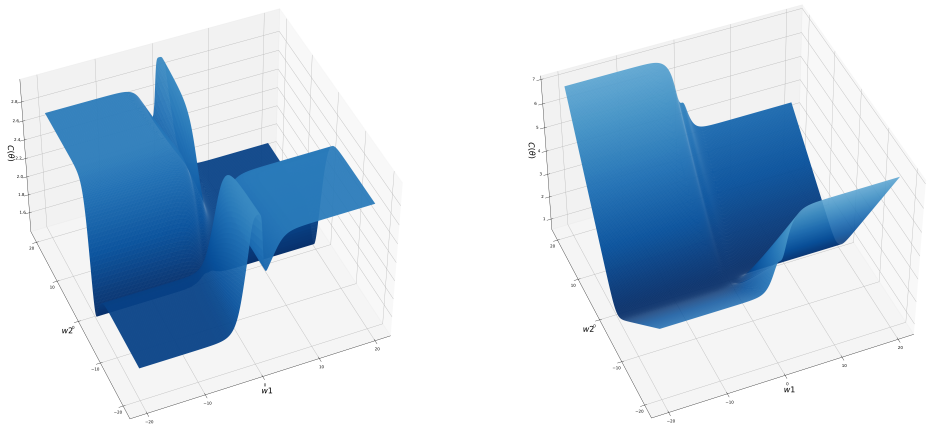


Figure 4.4: Training on a small subset of the MNIST dataset. The validation error rate is tracked together with the training loss. The early stopping strategy uses the model from epoch 76 where the validation performance is best.

As further details for Figure 4.4, we note that it is based on a deep fully connected network (see Chapter 5) with 5 layers and hundreds of thousands of parameters. We train the model using only $1,000$ MNIST images and use 500 additional images for the validation set. The performance function used on the validation set is one minus the accuracy, and as is evident, an accuracy of about 90% is obtained. Mini-batch sizes are taken at $n_b = 10$. As we can see, the model is trained for 200 epochs but retrospectively, the early stopping strategy pulls the parameters θ associated with epoch 76. Note that our chosen model is far from a realistic model which one would use for such a low data scenario, yet we present this example here to illustrate the idea of early stopping.

The second way in which deep learning optimization differs from general optimization is that the model's loss function itself is often specifically engineered for efficient optimization. In classical applications of optimization appearing in applied mathematics, operations research, and other fields, one often needs to optimize a given objective without the ability to modify the objective. In contrast, deep learning models are versatile enough so that the actual model, and subsequently the loss function can be chosen with the goal of facilitating efficient optimization. Discussions of this nature appear in the sequel where for example the choice of activation functions within a deep neural network significantly affects the nature of learning; see for example the discussion of vanishing gradients in Chapter 5.

On this matter, we have already seen in Figure 3.4 of Section 3.2 how the cross entropy loss for logistic regression presents a much better loss landscape for optimization in comparison with the squared loss. In the case of logistic regression, cross entropy loss even implies having a convex loss function, whereas that property is lost with the squared loss function. Further, in more complex models using cross entropy is also beneficial in terms of the loss landscape even if it does not guarantee convexity per-se. We now present one such illustrative example.



(a) (b)

Figure 4.5: The loss landscape of a very simple two-layer neural network. (a) Using the squared loss $C_i(\theta) = (y^{(i)} - \hat{y}^{(i)})^2$. (b) Using the binary cross entropy loss $C_i(\theta) = \mathrm{CE}_{\mathrm{binary}}(y^{(i)}, \hat{y}^{(i)})$.

Similarly to the synthetic data used for the logistic regression example yielding Figure 3.4, let us use the same synthetic data on the model,

$$f(x) = \sigma_{\mathrm{Sig}}\big(w_2\, \sigma_{\mathrm{Sig}}(w_1\, x)\big) = \frac{1}{1 + e^{-\frac{w_2}{1+e^{-w_1 x}}}}.$$

In the context of deep neural networks presented in Chapter 5, this is a two-layer network without bias terms, a single neuron per layer, and sigmoid activation functions. It is not a realistic model one would use in practice, yet for our expository purposes it has the benefit of having only two parameters, and hence we may visualize the loss surface as a function of w_1 and w_2.

Interestingly, as we see in Figure 4.5, using the cross entropy loss presents a much smoother loss landscape in comparison to the squared error loss. Hence using gradient based optimization with cross entropy would be much more efficient and have the algorithm less likely to get stuck in local minima or saddle points. Further, in much more realistic models with thousands or millions of parameters, while not possible to fully visualize, using cross entropy often yields more sensible loss landscapes as well. Hence in summary we see that when handling optimization in the context of deep learning, there are often other considerations at play, beyond standard optimization.

4.3 Adaptive Optimization with ADAM

Essentially, all deep learning optimization techniques can be cast as some form of the descent direction method outlined in Algorithm 4.1. Basic gradient descent (Algorithm 2.1) is one specific form of this descent direction method, and as we outlined in the previous section, it suffers from multiple drawbacks. While some of these drawbacks are alleviated when using noisy gradients via stochastic gradient descent or mini-batches, there is still much more room for improvement.

In this section we present one of the most popular improvements over basic gradient descent called the *ADAM algorithm* which is short for *adaptive moment estimation*. This algorithm has become the default practical choice for training deep learning models. Like basic gradient descent, ADAM is considered as a *first-order method* since it uses gradients (first derivatives) to determine the descent step used in Algorithm 4.1. For this, ADAM combines the ideas of *momentum*, and *adaptive learning rates per coordinate* into a simple mechanism of computing the descent step. Hence to understand ADAM, we study these two concepts, keeping in mind that the associated algorithms can also be viewed as independent methods in their own right. Other variations of these concepts together with additional first-order techniques that are not employed in ADAM are further summarized in Section 4.5.

Adaptive Optimization and Exponential Smoothing

A basic theme of ADAM and many other first-order methods has to do with adapting to the local nature of the loss landscape as iterations progress. For simple $d = 2$ examples as visualized in Figure 4.2 and Figure 4.5, we can quite easily get a taste for the nature of the loss landscape, and it is generally not difficult to choose descent steps that yield good overall performance. However, for practical examples with larger values of d, at any iteration t of the general Algorithm 4.1, the only information available at our disposal is the history of $\theta^{(0)}, \ldots, \theta^{(t-1)}$ and the associated gradients. It thus makes sense to try and "adapt" the descent step based on this history, perhaps with more emphasis on iterations from the near-history. One may loosely call such an approach *adaptive optimization*.

One simple principle used in adaptive optimization is *exponential smoothing*. Beyond optimization, exponential smoothing is popular in time series analysis and many other branches of data science and applied mathematics. In general, exponential smoothing operates on a sequence of vectors $u^{(0)}, u^{(1)}, u^{(2)} \ldots$. In the context of this section, the sequence may be a sequence of gradients or a sequence of the squares of gradients.

The exponentially smoothed sequence is denoted as $\bar{u}^{(0)}, \bar{u}^{(1)}, \ldots$ and is computed via,

$$\bar{u}^{(t+1)} = \beta \bar{u}^{(t)} + (1-\beta) u^{(t)}, \qquad \text{with} \qquad \bar{u}^{(0)} = 0, \tag{4.14}$$

and $\beta \in [0, 1)$. Thus, for $t \geq 0$, each smoothed vector $\bar{u}^{(t+1)}$ is a convex combination of the previous smoothed vector and the most recent (non-smoothed vector) $u^{(t)}$. When $\beta = 0$, no smoothing takes place,[14] and when β is a high value near 1, the new vector $u^{(t)}$ only has a small effect on the new smoothed vector $\bar{u}^{(t+1)}$.

It is simple to use the update formula (4.14) to represent the smoothed vector $\bar{u}^{(t)}$ as a convex combination of the complete history. Specifically, by iterating (4.14) we have,

$$\bar{u}^{(t+1)} = (1-\beta) \sum_{\tau=0}^{t} \beta^{t-\tau} u^{(\tau)}, \quad \text{for} \quad t = 0, 1, 2, \ldots, \tag{4.15}$$

and thus each $u^{(0)}, \ldots, u^{(t)}$ contributes to $\bar{u}^{(t+1)}$, with exponentially decaying weights. As we see now, within the context of adaptive optimization, such straightforward exponential smoothing can go a long way.

[14] Note that with our convention of indices, when $\beta = 0$ the exponentially smoothed sequence is actually a time shift of the original sequence.

Momentum

Recall that the descent step of basic gradient descent is α times the negative gradient. This implies that in flat regions of the loss landscape, gradient descent essentially stops. Further, at saddle points or local minima, gradient descent stops completely. Such drawbacks may be alleviated with the use of *momentum*.

To get an intuitive feel for the use of momentum in optimization, consider an analogy of rolling a ball downhill on the loss landscape. It gains momentum as it roles downhill, becoming increasingly faster until it reaches the bottom of a valley. Clearly, the ball keeps memory of the past forces in the form of acceleration. This motivates us to use exponential smoothing on the gradients to obtain an extra step called the *momentum update*. It is then used as follows,

$$v^{(t+1)} = \beta v^{(t)} + (1-\beta)\nabla C(\theta^{(t)}) \qquad \text{(Momentum Update)}, \tag{4.16}$$

$$\theta^{(t+1)} = \theta^{(t)} - \alpha v^{(t+1)} \qquad \text{(Parameter Update)}, \tag{4.17}$$

for scalar *momentum parameter* $\beta \in [0, 1)$ and *learning rate* $\alpha > 0$, starting with $v^{(0)} = 0$.

Similar to the general exponential smoothing of (4.14), the momentum update[15] is an exponential smoothing of the gradient vectors. When $\beta = 0$, we have the basic gradient descent method and for larger β, information of previous gradients plays a more significant role. In practice, for deep neural networks, the gradient $\nabla C(\theta^{(t)})$ is replaced with a noisy gradient based on stochastic gradient descent or mini-batches.

The vector $v^{(t)}$ in (4.16) is called the *momentum*[16] at the t-th update. Using (4.15), we have

$$v^{(t+1)} = (1-\beta)\sum_{\tau=0}^{t} \beta^{t-\tau}\nabla C(\theta^{(\tau)}), \quad \text{for} \quad t = 0, 1, 2, \ldots. \tag{4.18}$$

That is, the momentum accumulates all the past (weighted) gradients, providing acceleration to the parameter θ updates on downward surfaces. Therefore, for $\beta > 0$, the next step taken via (4.17) is not necessarily taken in the steepest descent, instead the direction is dictated by the (exponential smoothing) average of all the gradients up to the current iteration.

Figure 4.6 compares the performance of the gradient descent method with momentum for different values of β on the Rosenbrock function (4.9). We observe that for large β, the momentum method accelerates as it takes downward steps.

Adaptive Learning Rates per Coordinate

Observe that the updates of the gradient descent method, with or without momentum, use the same learning rate α for all components of θ. However, it is often better to learn the parameters $\theta_1, \ldots, \theta_d$ with potentially a specific learning rate for each coordinate θ_i.

[15] One may alternatively parameterize the momentum update (4.16) as $v^{(t+1)} = \beta v^{(t)} + \nabla C(\theta^{(t)})$. This alternative parameterization has the benefit of allowing one to adjust the momentum parameter β without having significant effects on the step size exhibited in (4.17). Our choice of the parameterization as in (4.16) is for consistency with ADAM presented below.

[16] In physics, momentum is defined as the product of the mass of an object and its velocity vector. The object's momentum points in the direction of an object's movement. In contrast, here the momentum vector $v^{(t)}$ points in the direction opposite to the step taken.

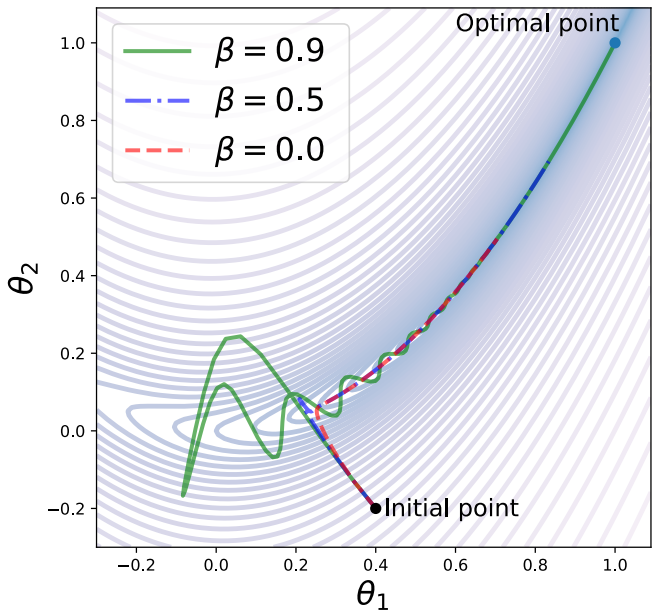


Figure 4.6: Application of momentum to the Rosenbrock function (4.9) for three different values of β with learning rate $\alpha = 0.001/(1-\beta)$ and a fixed number of iterations. Note that $\beta = 0$ is basic gradient descent.

To get a feel for this issue, let us assume a situation where specific parameters are associated with specific features in the data.[17] For instance, in applications of deep neural networks for natural language processing (see also Chapter 7), there are typically words that are less frequent than others. For example in most texts on trees, the word `chaulmoogra` most likely appears less frequently than the word `coconut`, even though both are names of tree species. Now assume that there are associated parameters $\theta_{\text{chaulmoogra}}$ and θ_{coconut}, and consider how these would be updated during learning.

What is likely to occur is that each mini-batch would have on average much fewer occurrences of `chaulmoogra` in comparison to `coconut`. However, if we update all the parameters at the same learning rate, then θ_{coconut} would be updated more than $\theta_{\text{chaulmoogra}}$. Hence with fixed learning rate α, the θ_{coconut} parameter is likely to converge quickly, while the $\theta_{\text{chaulmoogra}}$ parameter will be slow to respond.

A general approach for overcoming this issue is to scale each coordinate of the descent step with a different factor. That is, instead of considering a basic gradient descent step, $\theta_{\mathsf{s}} = -\alpha \nabla C(\theta)$, consider descent steps of the form,

$$\theta_{\mathsf{s}} = -\alpha\, r \odot \nabla C(\theta), \tag{4.19}$$

[17] In some simple models such as linear regression, logistic regression, or multinomial regression this is exactly the case for categorical one-hot encoded variables. In more advanced models there is a more complicated implicit mapping between parameters and features.

where r is some vector of positive entries which recalibrates the descent steps and $\odot$ is the element-wise product of two vectors.[18] That is, for a parameter such as $\theta_{\text{chaulmoogra}}$ we would expect the coordinate $r_{\text{chaulmoogra}}$ to be high in comparison to the coordinate r_{coconut} which is associated with θ_{coconut}. Such recalibration of the descent steps would result in large steps for $\theta_{\text{chaulmoogra}}$ whenever `chaulmoogra` appears in a mini-batch, and smaller steps for θ_{coconut}.

However, in general, finding a fixed good vector r for (4.19) is difficult, especially when the number of parameters is huge and the actual relationship between parameters and features is not clear. Instead, we use adaptive methods that adjust the descent step during the optimization process.

We now introduce two such approaches called *adaptive subgradient* (or *Adagrad*) and *root mean square propagation* (or *RMSprop*). In both of these approaches the update of coordinate i in iteration t can be represented as,

$$\theta_i^{(t+1)} = \theta_i^{(t)} - \frac{\alpha}{\sqrt{s_i^{(t+1)}} + \varepsilon} \frac{\partial C(\theta^{(t)})}{\partial \theta_i}, \tag{4.20}$$

where $s_i^{(0)}, s_i^{(1)}, s_i^{(2)}, \ldots$ is a sequence of non-negative values that are updated adaptively and ε taken to be a small value of order 1×10^{-8} to avoid division by zero. That is, with this representation, the descent step at time t represented in terms of (4.19) has $r_i = \left(\sqrt{s_i^{(t+1)}} + \varepsilon\right)^{-1}$. Generally we aim for s_i to be some smoothed representation of the square of the derivative $\partial C(\theta^{(t)})/\partial \theta_i$. Hence, r_i is roughly the inverse of the magnitude of the derivative, and r_i is low when the changes for coordinate i are steep and vise-versa.

Vector-wise, the parameter update (4.20) can be represented as

$$\theta^{(t+1)} = \theta^{(t)} - \frac{\alpha}{\sqrt{s^{(t+1)}} + \varepsilon} \odot \nabla C(\theta^{(t)}), \tag{4.21}$$

where $s^{(t)} = (s_1^{(t)}, \ldots, s_d^{(t)})$, ε is considered a vector, and the addition, division, and square-root operations are all element-wise operations.

Using the representation (4.20) or (4.21), let us now define how the sequence of $\{s_i^{(t)}\}$ is computed both for Adagrad and RMSprop. Specifically,

$$s_i^{(t+1)} = \begin{cases} \sum_{\tau=0}^{t} \left(\frac{\partial C(\theta^{(\tau)})}{\partial \theta_i}\right)^2, & \text{for Adagrad,} \\ \gamma s_i^{(t)} + (1-\gamma)\left(\frac{\partial C(\theta^{(t)})}{\partial \theta_i}\right)^2, & \text{for RMSprop,} \end{cases} \tag{4.22}$$

where the recursive relationship for RMSprop has the initial value at $s_i^{(0)} = 0$. RMSprop is also parameterized by a *decay parameter* $\gamma \in [0, 1)$ with typical values at $\gamma = 0.999$ implying that for RMSprop the sequence $\{s_i^{(t)}\}$ is a relatively slow exponential smoothing of the square of the derivative.

[18] The element-wise product is also called the *Hadamard product* or *Schur product*.

Adagrad may appear simpler than RMSprop since it does not involve any recursive step or any hyper-parameter like γ and it is simply an accumulation of the squares of the derivatives. However, the crucial drawback of Adagrad is that for each i, the sequence $\{s_i^{(t)}\}$ is strictly non-deceasing. As a consequence, the effective learning rate decreases during training, often making it infinitesimally small before convergence. RMSprop was introduced in the context of deep learning after Adagrad, and overcomes this problem via exponential smoothing of the squares of the partial derivatives.

It is also useful to use (4.15) to see that for RMSprop, the explicit (all-time) representation of the vector $s^{(t+1)}$ is,

$$s^{(t+1)} = (1-\gamma)\sum_{\tau=0}^{t} \gamma^{t-\tau}\left(\nabla C(\theta^{(\tau)}) \odot \nabla C(\theta^{(\tau)})\right), \tag{4.23}$$

while for Adagrad a similar representation holds without the $(1-\gamma)$ and $\gamma^{t-\tau}$ elements in the formula.

Bias Correction for Exponential Smoothing

We have seen that both momentum and RMSprop involve exponential smoothing of the gradients and squares of the gradients respectively. In both of these cases, we set zero initial conditions at iteration $t = 0$. Specifically, for momentum, we have $v^{(0)} = 0$ and for RMSprop, we have $s^{(0)} = 0$. In the absence of any better initial conditions, this is a sensible choice, yet such initial conditions may introduce some initial (short-term) bias. This bias would eventually "wash away" as t grows but it can play a significant role for short term t. The effect of such bias is especially pronounced when the respective exponential smoothing parameters β or γ are close to 1.

To mitigate such temporal bias, a common technique is to use *bias correction*. To see this, first observe from (4.14) that since the initial value $\bar{u}^{(0)} = 0$, the first element of the exponential smoothing is

$$\bar{u}^{(1)} = (1-\beta)u^{(0)}.$$

Since β is usually set near 1, $\bar{u}^{(1)}$ is close to zero even when $u^{(0)}$ is not close to 0. Consequently, $\bar{u}^{(2)}$ stays close to zero even if $u^{(1)}$ is away from 0 because

$$\bar{u}^{(2)} = \beta\bar{u}^{(1)} + (1-\beta)u^{(1)}.$$

Thus, initial values of the exponential smoothing sequence $\bar{u}^{(0)}, \bar{u}^{(1)}, \ldots$ remain close to 0 even when the elements of original sequence $u^{(0)}, u^{(1)}, \ldots$ are far from 0.

One way to handle such bias is to consider the special case where the input vectors in (4.15) are fixed at $u^{(t)} = u$. In such a case, via simple evaluation of a geometric sum we have,

$$\bar{u}^{(t+1)} = (1-\beta)\Big(\sum_{\tau=0}^{t}\beta^{t-\tau}\Big)u = (1-\beta)\Big(\sum_{\tau=0}^{t}\beta^{\tau}\Big)u = (1-\beta^{t+1})u.$$

Hence if we were to divide the exponentially smoothed value $\bar{u}^{(t)}$ by $1-\beta^{t+1}$, we would get constant vectors when $u^{(t)} = u$ (constant). Further, in (more realistic) cases where the underlying input sequence $u^{(0)}, u^{(1)}, u^{(2)} \ldots$ is not constant but close to a constant, the bias

corrected sequence $\{\bar{u}^{(t)}/(1-\beta^t)\}$ may still do a better job at mitigating initial effects. Clearly as t grows, $\beta^t \to 0$, and the effect of this bias correction disappears.

Now we may apply such bias correction to momentum or RMSprop where we have the specific forms (4.18) or (4.23), respectively. In these cases, the bias corrected updates for v and s are,

$$\hat{v}^{(t+1)} = \frac{1}{1-\beta^{t+1}} v^{(t+1)} \quad \text{and} \quad \hat{s}^{(t+1)} = \frac{1}{1-\gamma^{t+1}} s^{(t+1)}, \quad t = 0, 1, 2, \ldots. \tag{4.24}$$

Putting the Pieces Together: ADAM

Now that we understand ideas of momentum, RMSprop, and bias correction, we can piece these together into a single algorithm, namely the *adaptive moment estimation* method, or simply *ADAM*. Metaphorically, if the execution of the momentum method is a ball rolling down a slope, the execution of ADAM can be seen as a heavy ball with friction rolling down the slope.

The key update formula for ADAM is

$$\theta^{(t+1)} = \theta^{(t)} - \alpha \frac{1}{\sqrt{\hat{s}^{(t+1)}} + \varepsilon} \hat{v}^{(t+1)}, \tag{4.25}$$

where all vector operations (division, square root, and addition of ε) are interpreted element wise. Here $\hat{v}^{(t+1)}$ and $\hat{s}^{(t+1)}$ are bias corrected exponential smoothing of the gradient and the squared gradients as given in (4.24).

Algorithm 4.2: ADAM

Input: Dataset $\mathcal{D} = \{(x^{(1)}, y^{(1)}), \ldots, (x^{(n)}, y^{(n)})\}$,
objective function $C(\cdot) = C(\cdot\,; \mathcal{D})$, and
initial parameter vector θ_{init}

Output: Approximally *optimal* θ

$t \leftarrow 0$ (Initialize iteration counter)

$\theta \leftarrow \theta_{\mathsf{init}}$, $v \leftarrow 0$, and $s \leftarrow 0$ (Initialize state vectors)

repeat

  $g \leftarrow \nabla C(\theta)$ (Compute gradient)

  $v \leftarrow \beta v + (1-\beta)\, g$ (Momentum update)

  $s \leftarrow \gamma s + (1-\gamma)\,(g \odot g)$ (Second moment update)

  $\hat{v} \leftarrow \dfrac{v}{1-\beta^{t+1}}$ (Bias correction)

  $\hat{s} \leftarrow \dfrac{s}{1-\gamma^{t+1}}$ (Bias correction)

  $\theta \leftarrow \theta - \alpha \dfrac{1}{\sqrt{\hat{s}} + \varepsilon} \hat{v}$ (Update parameters)

  $t \leftarrow t + 1$

until *termination condition* is satisfied

return θ

With ADAM, α is still called the learning rate and is still the most important parameter which one needs to tune. The other parameters are $\beta \in [0, 1)$ for the momentum exponential smoothing as used in (4.16) and $\gamma \in [0, 1)$ for the RMSprop exponential smoothing as is used in (4.22). The common defaults for these parameters are $\beta = 0.9$ and $\gamma = 0.999$.

ADAM is presented in Algorithm 4.2 where Step 4 is the gradient computation, steps 5 and 6 are exponential smoothing (momentum and RMSprop), steps 7 and 8 are bias corrections, and finally Step 9 is the descent step.

Since the introduction of ADAM in 2014, this algorithm became the most widely used algorithm (or "optimizer") in deep learning frameworks. In the next section we start to focus on what is typically the most costly computation within the algorithm, namely Step 4, where we evaluate the gradient. Further generalizations of ADAM and other variations of first-order methods are presented in Section 4.5.

4.4 Automatic Differentiation

The computation of gradient vectors is a crucial step in the ADAM algorithm or in any other first-order optimization method. Practically, in deep learning, the most common way for computing such gradients is called *automatic differentiation* and is embodied in the *backpropagation algorithm*. In this section we introduce general concepts of automatic differentiation and then in Section 5.4 we specialize to the backpropagation algorithm for deep neural networks.

All popular methods for computing derivatives, gradients, Jacobians, and Hessians follow one of two approaches. In one approach, the algebraic expressions of the derivatives are computed first and then the derivatives for a particular given point are evaluated. For this approach the expressions of the derivatives are either derived manually, which is generally a tedious process, or using computer based *symbolic differentiation* methods. For example, we have seen explicit gradient expressions in (2.21) for linear regression and (3.17) for logistic regression. However, for general deep learning models, explicit expressions are not with such compact form and instead suffer from the problem of *expression swell*. Thus an alternative approach is to compute the numerical values of the derivatives at a given point directly. This approach includes standard methods of *numerical differentiation* as well as *automatic differentiation* which is our focus here.

In this section, we first present an overview of numerical and symbolic differentiation. We then outline key ideas of *differentiable programming* which is a programming paradigm that encompasses automatic differentiation. We then present forward mode automatic differentiation which is a stepping stone towards understanding backward mode automatic differentiation. Finally, we present backward mode automatic differentiation.

Numerical and Symbolic Differentiation

Suppose we would like to compute the gradient of the objective function $C(\theta)$ at a given parameter vector θ. Numerical differentiation methods use the definition of the partial derivative, see (A.5) in Section A.2, to compute the gradient $\nabla C(\theta)$ approximately. In particular, the most basic form of numerical differentiation approximates the partial derivative

via,

$$\frac{\partial C(\theta)}{\partial \theta_i} \approx \frac{C(\theta + he_i) - C(\theta)}{h}, \tag{4.26}$$

for a small constant $h > 0$, where e_i is the i-th unit vector. Thus to obtain a numerical estimate of $\nabla C(\theta)$ we need to evaluate $C(\cdot)$ at θ, and $\theta + he_i$ for $i = 1, \ldots, d$, and each time use (4.26).

A classic problem with numerical differentiation is selection of the constant h. While mathematically, smaller h provides a better approximation, numerically, small h yields numerical instability due to round-off errors.[19] Further, in the context of deep learning where θ is of very high dimension, the major drawback of numerical differentiation is that it requires us to perform an order of d function evaluations to compute the gradient at a point.

The basic alternative to numerical differentiation is symbolic differentiation. With this paradigm, instead of directly obtaining numerical values of derivatives, we represent expressions using *computer algebra systems* and obtain mathematical expressions for the derivatives using automated algorithms based on the rules of calculus. At its core, symbolic differentiation is a very useful tool. However, with deep learning, trying to rely on symbolic differentiation solely is not practical. A key problem is expression swell where the exact mathematical representation of partial derivatives of loss functions associated with deep neural networks may often require excessive resources due to the complexity of the resulting expressions.

Since in deep learning we are mainly concerned with numerical values of the derivatives but not with their analytical expressions, the application of symbolic differentiation can be unwieldy. However, instead of trying to find the expression of the objective function directly, if it is decomposed into several elementary operations, then we can numerically compute the gradients of the objective function by obtaining symbolic derivative expressions of these elementary operations while keeping track of the intermediate numerical values in a sequential manner. This idea of interleaving between symbolic expressions and numerical evaluations at elementary levels is the basis of automatic differentiation.

Overview of Differentiable Programming

Let us now introduce basic terminology of automatic differentiation within the world of *differentiable programming*. While for deep learning, the central application is associated with the multi-input scalar-output loss $C : \mathbb{R}^d \to \mathbb{R}$, to get a general feel for differentiable programming and automatic differentiation, let us first consider general functions, $g : \mathbb{R}^d \to \mathbb{R}^m$. That is, when $m > 1$, the outputs are vector valued.

Differentiable programming refers to a programming paradigm where numerical computer programs, or code for computer functions, are transformed into other functions which execute the derivatives, gradients, Jacobians, Hessians, or higher-order derivatives of the original computer functions. For example, consider a case with $d = 3$ and $m = 2$ and some hypothetical computer programming language with some programmed function `g()` which appears in code as follows:

```
g(t1, t2, t3) = (t1 - 7*t2 + 5*t3, t2*t3)
```

[19] There are other similar schemes that generally exhibit less numerical error than (4.26), yet any numerical differentiation scheme requires tuning of h.

We interpret this code as a function which operates on three input numbers θ_1, θ_2, and θ_3 and then returns two numbers of which the first is a linear combination of the three inputs, $\theta_1 - 7\theta_2 + 5\theta_3$, and the second is a product of two of the inputs, $\theta_2\theta_3$. From a programming perspective it is a *pure function* in the sense that it does not depend on any other variables and does not change any other variables in the system. Thus, the computer function `g()` implements a mathematical function $g : \mathbb{R}^3 \to \mathbb{R}^2$ with,

$$g(\theta_1, \theta_2, \theta_3) = \big(g_1(\theta_1, \theta_2, \theta_3),\ g_2(\theta_1, \theta_2, \theta_3)\big),$$

where $g_1 : \mathbb{R}^3 \to \mathbb{R}$ and $g_2 : \mathbb{R}^3 \to \mathbb{R}$.

With differentiable programming and automatic differentiation, we are interested in evaluation of partial derivatives such as $\partial g_i/\partial\theta_j$ for $i = 1, 2$ and $j = 1, 2, 3$, at specific values of θ_1, θ_2, and θ_3. For this simple example these derivatives are obviously known analytically and constitute the Jacobian matrix of g; see (A.9) in Appendix A. In our case, the Jacobian is,

$$J_g(\theta_1, \theta_2, \theta_3) = \begin{bmatrix} 1 & -7 & 5 \\ 0 & \theta_3 & \theta_2 \end{bmatrix},$$

where the i, j element of the Jacobian J_g is $\partial g_i/\partial\theta_j$.

In its most basic form, a differentiable programming system provides constructs such as `partial_derivative()` which gives us the ability to write code such as for example:

```
partial_derivative(target=g, t=[2.5, 2.1, 1.4], i=2, j=3)
```

The meaning of such code is a request of the computer to evaluate the numerical value of the Jacobian of the target function g at $\theta = (2.5, 2.1, 1.4)$ for $i = 2$ and $j = 3$. That is, we expect such code to compute $\partial g_2/\partial\theta_3$ and return 2.1 as a result in this case. In contrast to symbolic or numerical differentiation, this computation is to be carried out through mechanisms of automatic differentiation.

In addition to such general partial derivatives via `partial_derivative()`, we may also expect differentiable programming frameworks to expose software constructs such as `gradient()` for computing gradient vectors of functions $g : \mathbb{R}^d \to \mathbb{R}$, `jacobian()` for computing the Jacobian matrix of functions $g : \mathbb{R}^d \to \mathbb{R}^m$, and `hessian()` for Hessian matrices of functions $g : \mathbb{R}^d \to \mathbb{R}$. In any case, the mechanism of computation of these operations typically involves a transformation of the source code of the original functions, or some data structures that represent those functions, into new code or data structures that encode how derivative evaluation is carried out. See the notes and references at the end of the chapter for references to contemporary concrete software frameworks that execute such operations.

Beyond `partial_derivative()`, `gradient()`, `jacobian()`, and `hessian()`, there are also two other basic constructs in a differentiable programming framework. These are *jacobian vector products*, `jvp()`, and *vector Jacobian products*, `vjp()`. A Jacobian vector product is based on $\theta \in \mathbb{R}^d$ and $v \in \mathbb{R}^d$ and is the evaluation of $J_g(\theta)v$ which results in a vector in $\mathbb{R}^m$. Similarly a vector Jacobian product is based on $\theta \in \mathbb{R}^d$ and $u \in \mathbb{R}^m$ and is the evaluation of $u^\top J_g(\theta)$ which results in a row vector whose transpose is in $\mathbb{R}^d$. Due to the way in which automatic differentiation operates, `jvp()` and `vjp()` are in fact the basic operations that one may expect from a differentiable programming framework. Whereas the other operations, `partial_derivative()`, etc. are all built on top of either `jvp()` or `vjp()`.

Automatic differentiation generally works in two types of modes, namely *forward mode* which is the algorithm behind `jvp()` and *backward mode*, also known as reverse mode, which is the algorithm behind `vjp()`. In both cases the multivariate chain rule plays a key role; see Section A.3 in the appendix for a review of the chain rule. The general principle of automatic differentiation is based on computing intermediate partial derivatives and incorporating these computed values in an iterative computation. Forward mode automatic differentiation is a straightforward application of the multivariable chain rule while backward mode is slightly more involved. Note that the backpropagation algorithm for deep learning is a special case of backward mode. The subsections below focus on the inner workings of forward mode and backward mode automatic differentiation, but first let us see how `jvp()` and `vjp()` are generally used in differentiable programming.

As stated above, the basic service provided by forward mode automatic differentiation is the computation of Jacobian vector products. For instance, returning to the simple $d = 3$, $m = 2$ example above, we may invoke forward mode automatic differentiation via code such as,

```
jvp(target=g, t=[2.5, 2.1, 1.4], v=[0,1,0]),
```

Here we choose $\theta = (2.5, 2.1, 1.4)$ as before and v to be the unit vector e_2. The result is in this case is `[-7, 1.4]`. Setting v as a unit vector[20] can be useful because in general when using $v = e_j$, the output $J_g(\theta)e_j$ is the j-th column of the Jacobian matrix. Hence in this example of `jvp()`, we obtain as output the effect of θ_2 on the two outputs $g_1(\theta)$ and $g_2(\theta)$ which is the vector $(\partial g_1/\partial\theta_2,\ \partial g_2/\partial\theta_2) = (-7, 1.4)$.

This shows that to implement `partial_derivative` for some $\partial g_i/\partial\theta_j$ at θ, we can compute $J_g(\theta)e_j$ (the j-th column of the Jacobian matrix) using `jvp()` and take the i-th index of the output. Specifically a single application of `jvp()` on $v = e_j$ provides us with the partial derivatives $\partial g_i/\partial\theta_j$ for all $i = 1, \ldots, m$. Hence vector Jacobian products provided by forward mode automatic differentiation are useful when the output m is large and we wish to only see the effect of a single, or a few input variables θ_j on the output.

However, note that in the deep learning loss function application where $m = 1$ and d is large, vector Jacobian products are not an efficient means for evaluating the gradient. Specifically, when we wish to evaluate `gradient()` using `jvp()` we need to invoke `jvp()` d separate times, each time with $v = e_j$ for $j = 1, \ldots, d$. Thus using forward mode automatic differentiation (Jacobian vector products) is not efficient for deep learning loss landscape gradient evaluation.

The alternative to `jvp()` is to use vector Jacobian products as exposed via backward mode automatic differentiation. With this approach we may invoke code such as,

```
vjp(target=g, t=[2.5, 2.1, 1.4], u=[1,0])
```

In this example we use the same θ as above and set $u = e_1 \in \mathbb{R}^2$. Since the vector Jacobian product evaluates $u^\top J_g(\theta)$ the output is a row vector of dimension $d = 3$ which is the first row of $J_g(\theta)$, namely $\nabla g_1(\theta)^\top$. More generally applying Jacobian vector products on $u = e_i$ yields the transposed gradient $\nabla g_i(\theta)^\top$.

In the deep learning loss minimization scenario d is very large and $m = 1$. Thus the Jacobian is simply the transpose of the gradient and is a long row vector. In such a case, computing

[20] We can also use more general (non unit vectors) and obtain directional derivatives, yet in the remainder of our discussion inputs to Jacobian vector products (v) and vector Jacobian products (u) are unit vectors.

the Jacobian vector product with $u = 1$ (scalar) yields the gradient of the loss function directly. Thus since backward mode automatic differentiation implements `vjp()`, using it on the loss function, is directly suited for deep learning.

We now explore the inner workings of forward mode automatic differentiation followed by backward mode automatic differentiation. In both cases we restrict to $m = 1$. Understanding forward mode is a useful stepping stone to understanding backward mode. Note that forward mode and backward mode involve internal implicit computations of Jacobian vector products or vector Jacobian products, respectively.[21]

The Computational Graph and Forward Mode Automatic Differentiation

Computational graphs play a key role in the implementation of automatic differentiation. A *computational graph* for a multivariate function $C : \mathbb{R}^d \to \mathbb{R}$ is a directed graph where the nodes correspond to elementary operations involved in the mathematical expression of $C(\theta)$. Inputs to the graph are the variables $\theta_1, \ldots, \theta_d$ and constants if required, and the output is the function value $C(\theta)$.

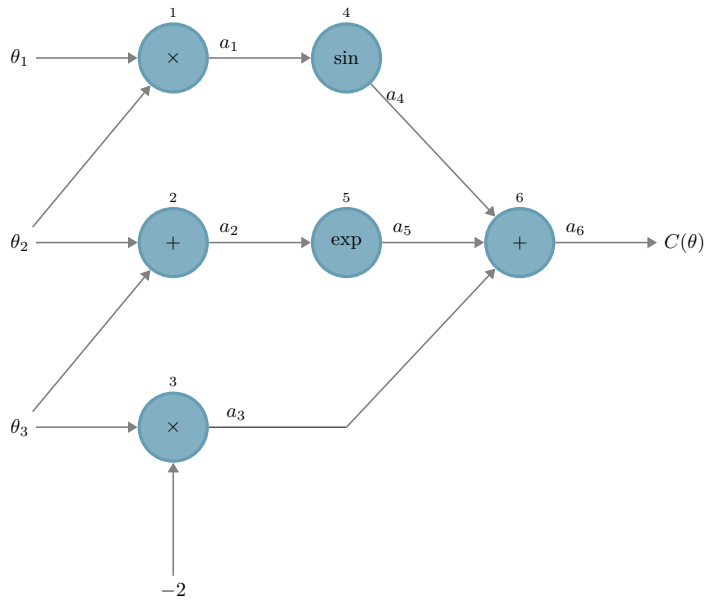


Figure 4.7: The computational graph of (4.27) with nodes numbered 1 to 6. The primal a_i of each node is the result of an elementary operation applied on its inputs.

As a simple example with $d = 3$, consider,

$$C(\theta) = \sin(\theta_1\, \theta_2) + \exp(\theta_2 + \theta_3) - 2\theta_3. \tag{4.27}$$

The computational graph for this function, presented in Figure 4.7, is composed of nodes with inputs and outputs. The output of each node is an elementary operation on the inputs to the

[21] It may be also useful to see the mathematical representations of Jacobian vector products and vector Jacobian products through a composition of functions. For this we may see (A.16), (A.17), and (A.18), in Appendix A.

node. These elementary operations include summation, multiplication, $\sin(\cdot), \cos(\cdot), \exp(\cdot)$, etc. Nodes are numbered $1, 2, \ldots$, and we denote the output of node i with $a_i = f_i(u_1, \ldots, u_{\ell_i})$ when there are ℓ_i inputs to node i, denoted as $u_1, \ldots, u_{\ell_i}$. In this example there are 6 nodes, and taking node 6, as an illustration, we have $a_6 = f_6(a_3, a_4, a_5)$ where $\ell_6 = 3$ and $f_6(\cdot)$ is the summation of its input arguments. Note that a_i is sometimes called the *primal* of that node in a given computation.

The evaluation at each node in the computational graph is executed at the instance in which inputs to the node become available. This then implies the notion of iterations, where a node is considered to be evaluated at a given iteration if the inputs to that node are all available at that iteration but not previously. As an example, with $\theta = (\pi/6, 2, 5)$, the iterations for (4.27) are summarized in Table 4.1.

Table 4.1: Evaluation of the value of $C(\theta)$ in (4.27) at an example θ via the computational graph in Figure 4.7 computing the primals a_i for $i = 1, \ldots, 6$.

	General expressions of primals	Values of primals at specific θ
Input	$\theta = (\theta_1, \theta_2, \theta_3)$	$\theta = (\pi/6, 2, 5)$
Iteration 1	$a_1 = \theta_1\theta_2$ $a_2 = \theta_2 + \theta_3$ $a_3 = -2\theta_3$	$a_1 = \pi/3$ $a_2 = 7$ $a_3 = -10$
Iteration 2	$a_4 = \sin(a_1)$ $a_5 = \exp(a_2)$	$a_4 = \sqrt{3}/2$ $a_5 = \mathrm{e}^7$
Iteration 3 (output)	$a_6 = a_4 + a_5 + a_3$ $\quad = C(\theta)$	$a_6 = C(\theta)$ $\quad = \sqrt{3}/2 + \mathrm{e}^7 - 10$

As mentioned above, forward mode automatic differentiation implements Jacobian vector products. In our exposition here, since the target function $C(\cdot)$ is scalar valued ($m = 1$), the Jacobian is the transpose of the gradient and with $v \in \mathbb{R}^d$, the Jacobian vector product, $J_C(\theta)v$, is a scalar.[22] Further, our exposition focuses on $v = e_j$, implying that the output of forward mode automatic differentiation is $\partial C(\theta)/\partial\theta_j$. Hence for our exposition here, j is fixed.

The key idea of forward mode automatic differentiation is to maintain a record of intermediate derivatives as computation progresses through the computational graph. For this we denote,

$$\dot{a}_i = \frac{\partial a_i}{\partial \theta_j}, \tag{4.28}$$

and call $\dot{a}_i$ the *tangent* at node i, for fixed j. Now since $a_i = f_i(u_1, \ldots, u_{\ell_i})$, using the multivariable chain rule (see Appendix A.3), we have

$$\dot{a}_i = \sum_{k=1}^{\ell_i} \frac{\partial f_i(u_1, \ldots, u_{\ell_i})}{\partial u_k} \dot{u}_k = \nabla f_i(u_1, \ldots, u_{\ell_i})^\top \dot{u}, \tag{4.29}$$

[22] It is in fact the directional derivative of $C(\cdot)$ in the direction v.

with $\dot{u}_k$ denoting the tangent of u_k and with $\dot{u}$ denoting the vector of these tangents. Returning to Figure 4.7 as an example, we have that $\nabla f_6(u_1, u_2, u_3) = (1, 1, 1)$ and thus, $\dot{a}_6 = \dot{a}_3 + \dot{a}_4 + \dot{a}_5$. Similarly, in that figure, $\nabla f_4(u_1) = \cos(u_1)$ and thus $\dot{a}_4 = \cos(a_1)\dot{a}_1$.

Since (4.29) defines a recursive computation of the tangents, we also require the inputs to the function $\theta_1, \ldots, \theta_d$ to have tangents. We set these as the elements of the vector $v \in \mathbb{R}^d$ used in the Jacobian vector product. Specifically, here we focus on Jacobian vector products with $v = e_j$ since we are seeking the derivative with respect to a specific input θ_j. Hence we set the tangent values for the inputs of the function as $\dot{\theta}_j = 1$ and $\dot{\theta}_i = 0$ for $i \neq j$.

For forward mode automatic differentiation, we progress on the computational graph in the forward direction from the input to the output, evaluating both the primal a_i and the tangent $\dot{a}_i$ at each node. As we progress, in each iteration, we consider the set of all nodes i whose predecessors already have primal values and tanget values available, and for each of these nodes we compute the primal values as was already shown in Table 4.1, and importantly also compute the tangent values via (4.29). This computation may be broken into iterations, in a similar way to the iterations of the function evaluation computed in Table 4.1.

In particular, since $f_i(\cdot)$ denotes an elementary operation, such as the function $\exp(\cdot)$ at node 5 in Figure 4.7, we assume the availability of a symbolic expression for each $\frac{\partial f_i(u_1,\ldots,u_{\ell_i})}{\partial u_k}$, or alternatively an ability to numerically compute it exactly. Therefore, since we know the numerical primal values $u_1, \ldots, u_{\ell_i}$ and the numerical tangent values $\dot{u}_1, \ldots, \dot{u}_{\ell_i}$, we can easily compute $\dot{a}_i$ using (4.29). Consequently, the numerical value of $\frac{\partial C(\theta)}{\partial \theta_j}$ can be computed at the end of forward mode as it is the tangent value of the output node of the computational graph.

Table 4.2 illustrates forward mode automatic differentiation using the example function $C(\theta)$ in (4.27). The first column focuses on the general expression and the other column computes the partial derivative of $C(\theta)$ with respect to θ_2 at a specific point θ.

Table 4.2: Forward mode automatic differentiation for the example function (4.27) and $j = 2$, i.e., we seek the derivative with respect to θ_2. The right column illustrates computation[23] of the derivative $\frac{\partial C(\theta)}{\partial \theta_2}$ at $\theta = (\pi/6, 2, 5)$ which is obtained via the final tangent value $\dot{a}_6 = \pi/12 + \mathrm{e}^7$.

	General expressions of primals and tangents	Values of primals and tangents at specific θ for $j = 2$
Start	$\theta = (\theta_1, \theta_2, \theta_3)$ $\dot{\theta} = (\dot{\theta}_1, \dot{\theta}_2, \dot{\theta}_3)$	$\theta = (\pi/6, 2, 5)$ $\dot{\theta} = (0, 1, 0)$
Iteration 1	$(a_1, \dot{a}_1) = (\theta_1\theta_2, \theta_1\dot{\theta}_2 + \dot{\theta}_1\theta_2)$ $(a_2, \dot{a}_2) = (\theta_2 + \theta_3, \dot{\theta}_2 + \dot{\theta}_3)$ $(a_3, \dot{a}_3) = (-2\theta_3, -2\dot{\theta}_3)$	$(a_1, \dot{a}_1) = (\pi/3, \pi/6)$ $(a_2, \dot{a}_2) = (7, 1)$ $(a_3, \dot{a}_3) = (-10, 0)$
Iteration 2	$(a_4, \dot{a}_4) = (\sin(a_1), \cos(a_1)\dot{a}_1)$ $(a_5, \dot{a}_5) = (\exp(a_2), \exp(a_2)\dot{a}_2)$	$(a_4, \dot{a}_4) = (\sqrt{3}/2, \pi/12)$ $(a_5, \dot{a}_5) = (\mathrm{e}^7, \mathrm{e}^7)$
Iteration 3	$(a_6, \dot{a}_6) = \begin{bmatrix} a_3 + a_4 + a_5 \\ \dot{a}_3 + \dot{a}_4 + \dot{a}_5 \end{bmatrix}$	$(a_6, \dot{a}_6) = \begin{bmatrix} \sqrt{3}/2 + \mathrm{e}^7 - 10 \\ \pi/12 + \mathrm{e}^7 \end{bmatrix}$

Backward Mode Automatic Differentiation

As mentioned above, backward mode automatic differentiation implements vector Jacobian products. This allows us to use a single run of backward mode automatic differentiation to compute the gradient vector for $C : \mathbb{R}^d \to \mathbb{R}$. For deep learning this is a significant improvement over forward mode automatic differentiation which would require d executions to obtain such a gradient.

For a given point θ, backward mode automatic differentiation is executed in two phases often called the *forward pass* and *backward pass*. The forward pass phase is executed before the backward pass phase. In the *forward pass*, the algorithm simply evaluates the computational graph to compute the values of the intermediate primal variables a_i while recording the dependencies of these variables in a bookkeeping manner, similar to the evaluation in Table 4.1.

In the backward pass, all the derivatives of the objective function are computed by using intermediate derivatives called *adjoints*. Specifically, for each intermediate variable a_i, the adjoint is

$$\zeta_i = \frac{\partial C}{\partial a_i}. \tag{4.30}$$

Adjoint ζ_i captures the rate of change in the final output C with respect to variable a_i. Contrast this with the tangent $\dot{a}_i$ from forward mode automatic differentiation, as in (4.28), which captures the rate of change of a_i with respect to an input variable.

In the backward pass we populate the values of the adjoint variables ζ_i. Specifically, we progress on the computational graph in the backward direction starting from the output node and propagating towards the input. For this, say that the output of node i is input to $\tilde{\ell}_i$ nodes with indices $i_1, \ldots, i_{\tilde{\ell}_i}$. Then the only way that a_i can influence the output C is by influencing $a_{i_1}, \ldots, a_{i_{\tilde{\ell}_i}}$. Now using the multivariable chain rule,

$$\frac{\partial C}{\partial a_i} = \sum_{k=1}^{\tilde{\ell}_i} \frac{\partial a_{i_k}}{\partial a_i} \frac{\partial C}{\partial a_{i_k}},$$

and thus using the notation of (4.30),

$$\zeta_i = \sum_{k=1}^{\tilde{\ell}_i} \frac{\partial a_{i_k}}{\partial a_i} \zeta_{i_k}. \tag{4.31}$$

The relationship (4.31) is used in iterations of the backward pass. In every iteration, the adjoints ζ_{i_k} in the right hand side of (4.31) are available from previous iterations and are used to compute ζ_i. In particular at the start of the backward pass we initilize the adjoint of the output node at 1. Further, recalling the notation used for forward mode automatic differenation, $a_{i_k} = f_{i_k}(u_1, \ldots, u_{\ell_{i_k}})$ where $u_1, \ldots, u_{\ell_{i_k}}$ are the inputs to node i_k and $f_{i_k}(\cdot)$ has readily computable derivatives with respect to its inputs. Hence, since the primals of $u_1, \ldots, u_{\ell_{i_k}}$ were already populated during the forward pass, we compute $\frac{\partial a_{i_k}}{\partial a_i}$ locally at node i and use it in (4.31).

[23] To verify the computation recall the basic linearity and product rules of scalar differentiation. Also, remember that the derivative of $\sin(\cdot)$ is $\cos(\cdot)$, that the derivative of $\exp(\cdot)$ is $\exp(\cdot)$, and that $\cos(\pi/3) = 1/2$.

Table 4.3: The backward pass of backward mode automatic differentiation for the example function (4.27) to compute its gradient at $\theta = (\pi/6, 2, 5)$. The result is $\nabla C(\theta) = (1, \pi/12 + e^7, e^7 - 2)$. This is computed after the forward pass is executed; see Table 4.1.

	General expressions of adjoints	Values of adjoints at specific θ
Start	$\zeta_6 = \dfrac{\partial C(\theta)}{\partial a_6}$	$\zeta_6 = 1$
Iteration 1	$\zeta_4 = \dfrac{\partial a_6}{\partial a_4}\zeta_6 = 1 \times \zeta_6$ $\zeta_5 = \dfrac{\partial a_6}{\partial a_5}\zeta_6 = 1 \times \zeta_6$ $\zeta_3 = \dfrac{\partial a_6}{\partial a_3}\zeta_6 = 1 \times \zeta_6$	$\zeta_4 = 1$ $\zeta_5 = 1$ $\zeta_3 = 1$
Iteration 2	$\zeta_1 = \dfrac{\partial a_4}{\partial a_1}\zeta_4 = \cos(a_1)\zeta_4$ $\zeta_2 = \dfrac{\partial a_5}{\partial a_2}\zeta_5 = \exp(a_2)\zeta_5$	$\zeta_1 = \cos(\pi/3) = 1/2$ $\zeta_2 = \mathrm{e}^7$
Iteration 3	$\dfrac{\partial C(\theta)}{\partial \theta_1} = \dfrac{\partial a_1}{\partial \theta_1}\zeta_1 = \theta_2\zeta_1$ $\dfrac{\partial C(\theta)}{\partial \theta_2} = \dfrac{\partial a_1}{\partial \theta_2}\zeta_1 + \dfrac{\partial a_2}{\partial \theta_2}\zeta_2 = \theta_1\zeta_1 + 1 \times \zeta_2$ $\dfrac{\partial C(\theta)}{\partial \theta_3} = \dfrac{\partial a_2}{\partial \theta_3}\zeta_2 + \dfrac{\partial a_3}{\partial \theta_3}\zeta_3 = 1 \times \zeta_2 - 2 \times \zeta_3$	$\dfrac{\partial C(\theta)}{\partial \theta_1} = 2 \times 1/2 = 1$ $\dfrac{\partial C(\theta)}{\partial \theta_2} = \pi/6 \times 1/2 + \mathrm{e}^7 = \pi/12 + \mathrm{e}^7$ $\dfrac{\partial C(\theta)}{\partial \theta_3} = \mathrm{e}^7 - 2$

The elements of the gradient $\nabla C(\theta)$ depend on the adjoints in a similar manner to (4.31). Specifically,

$$\frac{\partial C}{\partial \theta_j} = \sum_k \frac{\partial a_{j_k}}{\partial \theta_j}\zeta_{j_k}, \tag{4.32}$$

where the summation is over all nodes indexed via j_k that take θ_j as input in the computational graph. Thus, in the final iteration of the backward pass, we use (4.32) to obtain the gradient $\nabla C(\theta)$.

Let us return to the example in (4.27) with computational graph (4.7) evaluated at $\theta = (\pi/6, 2, 5)$. Evaluation of $\nabla C(\theta)$ via backward mode automatic differentiation first executes a forward pass as previously illustrated in Table 4.1. This populates the primals $a_1, \ldots, a_6$. Then the backward pass progresses via repeated application of (4.31) and with application of (4.32) in the final iteration. This backward pass computation is summarized in Table 4.3.

4.5 Additional Techniques for First-Order Methods

In Section 4.3 we have seen one of the most popular deep learning optimization algorithms, ADAM, as well as key milestone techniques that have led to its development. Then in

Section 4.4 we explored ways to compute the gradient $\nabla C(\theta)$. In this section, we explore other first-order optimization ideas that are less popular in practical deep learning, but still embody useful principles.

We begin the discussion by considering a modification of the momentum technique, to a variant called *Nesterov momentum*.[24] The idea of this method has also found its way into an algorithm called *Nadam* (Nesterov ADAM). We continue by considering variants of Adagrad and RMSProp for adaptive learning rates per coordinate. The variants we describe are called *Adadelta* and *Adamax*. While as of today, these additional techniques are not common in deep learning practice, they have made significant impact earlier on and may be fruitful in the future as well. In a sense, momentum (covered in Section 4.3), ADAM, Nesterov momentum, Nadam, Adagrad (covered in Section 4.3), Adadelta, and Adamax are all variants and improvements of basic gradient descent. Thus, our exposition in this section aims to complete the picture on how basic gradient descent may be improved.

We close this section with an overview of *line search* techniques. With such techniques, once a descent direction is determined, we seek to move in that direction with a step size which minimizes loss over the next step in that direction. There are several variants for this approach, and we present an overview of these variants.

Nesterov Momentum and the Nadam Algorithm

We have seen that the momentum updates as in (4.16) and (4.17) accelerate like a ball rolling downhill. An issue with this method is that the steps do not slow down after reaching the bottom of a valley, and hence, there is a tendency to overshoot the valley floor. Therefore, if we know approximately where the ball will be after each step, we can slow down the ball before the hill slopes up again.

The idea of *Nesterov momentum* tries to improve on standard momentum by using the gradient at the predicted future position instead of the gradient of the current position. The update equations are,

$$v^{(t+1)} = \beta v^{(t)} + (1-\beta)\nabla C(\underbrace{\theta^{(t)} - \alpha\beta v^{(t)}}_{\substack{\text{predicted}\\ \text{next point}}}), \tag{4.33}$$

$$\theta^{(t+1)} = \underbrace{\theta^{(t)} - \alpha v^{(t+1)}}_{\substack{\text{actual}\\ \text{next point}}}, \tag{4.34}$$

for constants $\beta \in [0, 1)$ and $\alpha > 0$, with $v^{(1)} = 0$. Compare (4.33) and (4.34) with (4.16) and (4.17). The difference here is that the gradient is computed at a predicted next point $\theta^{(t)} - \alpha\beta v^{(t)}$, which is a proxy for the (unseen) actual next point $\theta^{(t+1)} = \theta^{(t)} - \alpha v^{(t+1)}$ when $\beta \approx 1$.

Implementing Nesterov momentum via (4.33) and (4.34) requires evaluation of the gradient at the predicted points instead of actual points. This may incur overheads, especially when incorporated as part of other algorithms. For example, if we also require $\nabla C(\theta^{(t)})$ at each iteration for purpuses such as RMSprop, then the gradient needs to be computed twice instead of once per iteration; once for $\nabla C(\theta^{(t)})$ and once for $\nabla C(\theta^{(t)} - \alpha\beta v^{(t)})$. For this

[24] Nesterov momentum is also sometimes called *Nesterov acceleration* or *Nesterov accelerated gradient*.

reason, and also for simplicity of implementing gradients only at $\theta^{(t)}$, a *look-ahead momentum* mechanism is sometimes used in place of (4.33) and (4.34).

To see how this mechanism works, first revisit the basic momentum update equations (4.16) and (4.17) and observe that the parameter update can be represented as,

$$\theta^{(t+1)} = \theta^{(t)} - \alpha\big(\beta v^{(t)} + (1-\beta)\nabla C(\theta^{(t)})\big). \tag{4.35}$$

Observe that $v^{(t)}$ is based on gradients at points $\theta^{(0)}, \ldots, \theta^{(t-1)}$, but not on the gradient at $\theta^{(t)}$. Hence a way to incorporate this last gradient is with look-ahead momentum where we replace $v^{(t)}$ of (4.35) by $v^{(t+1)}$. This achieves behaviour similar to Nesterov momentum.

With such a replacement, we arrive at update equations of the form,

$$v^{(t+1)} = \beta v^{(t)} + (1-\beta)\nabla C\big(\theta^{(t)}\big), \tag{4.36}$$

$$\theta^{(t+1)} = \theta^{(t)} - \alpha\Big(\underbrace{\beta v^{(t+1)} + (1-\beta)\nabla C\big(\theta^{(t)}\big)}_{\substack{\text{look-ahead}\\ \text{momentum}}}\Big). \tag{4.37}$$

Using (4.36) and (4.37) aims to provide similar behaviour to the Nesterov momentum equations (4.33) and (4.34), yet only requires evaluation of gradients at $\theta^{(t)}$ as opposed to a shifted point as in (4.33). While look-ahead momentum is not equivalent to Nesterov momentum, the gist of both methods is similar.

Now with update equations like (4.36) and (4.37), it is straightforward to adapt ADAM to include behavior similar to Nesterov momentum. This is done using similar steps to those used to construct ADAM using momentum, RMSprop, and bias correction in Section 4.3. The difference here is that we use the look-ahead momentum equations (4.36) and (4.37). This algorithm is called Nadam.

In short, Nadam can be viewed exactly as the ADAM procedure in Algorithm 4.2, yet with line 7 replaced by,

$$\hat{v} \leftarrow \frac{\beta}{1-\beta^{t+2}}\, v + \frac{1-\beta}{1-\beta^{t+1}}\, g.$$

In this case, $\hat{v}$ is not just a bias corrected version of v as in ADAM since it also incorporates look-ahead momentum, which is the key extra feature that Nadam adds to ADAM.

Adadelta

Recall the computation of $s^{(t+1)}$ for the Adagrad method as in (4.22). When Adagrad's $s^{(t+1)}$ is used in (4.21), it has the problem of monotonically decreasing effective learning rates. This is one of the reasons that eventually RMSprop became more popular than Adagrad. However, there are other alternatives that are also very popular. One such alternative is the *Adadelta* method which uses exponential smoothing of the squared gradients as in RMSprop, but also uses another exponentially smoothed sequence of the descent step's squares.

A key motivation for Adadelta is the observation that the update equation (4.21) for RMSprop or Adagrad uses a unit-less quantity as the descent step. Specifically, in (4.21) the only unit-full quantity in the coefficient multiplying $\nabla C(\theta^{(t)})$ is the reciprocal of $\sqrt{s^{(t+1)}}$.

Hence that coefficient has units which are the inverse of the gradient and these cancel out the units of the gradient implying that the descent step is unit-less.

With Adadelta, the update equation (4.21) is modified so that the descent step maintains the same units of the gradient, via

$$\theta^{(t+1)} = \theta^{(t)} \underbrace{- \frac{\sqrt{\Delta\theta^{(t)}} + \varepsilon}{\sqrt{s^{(t+1)}} + \varepsilon} \odot \nabla C(\theta^{(t)})}_{\text{Descent step } \widetilde{\nabla} C(\theta^{(t)})}, \tag{4.38}$$

where $\Delta\theta^{(t)}$ is adaptively adjusted yet has the same units as $s^{(t+1)}$, making the coefficient of the gradient unit-free. Compare (4.38) with (4.21) to observe that $\sqrt{\Delta\theta^{(t)}} + \varepsilon$ replaces the learning rate α. This also means that Adagrad is "learning rate free". Instead, a potentially more robust parameter $\rho \in [0, 1)$ is used similarly to the γ parameter for RMSProp. This parameter specifies how to exponentially smooth squares of the descent steps.

Using the descent step of (4.38), the update equation for $\Delta\theta^{(t)}$ is,

$$\Delta\theta^{(t)} = \rho\Delta\theta^{(t-1)} + (1-\rho)\left(\widetilde{\nabla} C(\theta^{(t-1)}) \odot \widetilde{\nabla} C(\theta^{(t-1)})\right),$$

starting with $\Delta\theta^{(0)} = 0$. Then at iteration t, updating $\theta^{(t+1)}$ via (4.38), we use both $\Delta\theta^{(t)}$ and $s^{(t+1)}$, where the latter is updated via the RMSprop update equation as in (4.22).

Other Norms and Adamax

We have already seen that the quantities $s_i^{(t)}$ used in RMSprop or Adadelta as in (4.22) are useful for adaptive learning rates per coordinate. With these methods, we may loosely interpret $\sqrt{s_i^{(t)}}$ as an estimate of the standard (L_2) Euclidean norm of the sequence of gradients for coordinate i, smoothed up to time t.

In principal, one may wish to use other L_p norms (see Appendix A.1) in place of the Euclidean norm. For this we may modify the RMSprop recursion as in (4.22) to the form,

$$s_i^{(t+1)} = \gamma^p\, s_i^{(t)} + (1-\gamma^p)|g_i^{(t)}|^p, \qquad \text{with} \qquad g_i^{(t)} = \frac{\partial C(\theta^{(t)})}{\partial \theta_i}, \tag{4.39}$$

and then interpret $\left(s_i^{(t)}\right)^{\frac{1}{p}}$ as an estimate of the smoothed L_p norm of the sequence of gradients for coordinate i up to time t. Note that for convenience of the calculations below, we reparameterize the exponential smoothing parameter to be γ^p in place of γ.

With a recursion such as (4.39) we can obtain an "ADAM like" algorithm that updates parameters via an update rule such as,

$$\theta^{(t+1)} = \theta^{(t)} - \alpha \frac{1}{\left(\hat{s}^{(t+1)}\right)^{1/p} + \varepsilon}\, \hat{v}^{(t+1)}, \tag{4.40}$$

where $\hat{v}^{(t+1)}$ is some bias-corrected momentum value, and $\hat{s}^{(t+1)}$ is a bias corrected value obtained from (4.39); compare (4.40) to (4.25). This additional hyper-parameter p, determining which L_p norm to use, can then potentially be tuned. However, for non-small p

computations may often exhibit excessive numerical error due to overflow since we are raising the gradient coordinate value to high powers and then taking a low power of it.

Adamax is a variant of this approach which instead of using the L_p norm uses the L_∞ norm. Specifically in place of $(s_i^{(t)})^{\frac{1}{p}}$ we use $r_i^{(t)}$ which is recursively defined as,

$$r_i^{(t+1)} = \max\left[\gamma\, r_i^{(t)},\ g_i^{(t)}\right], \qquad \text{with} \qquad r_i^{(0)} = 0. \tag{4.41}$$

Then in place of (4.40) we use

$$\theta^{(t+1)} = \theta^{(t)} - \alpha \frac{1}{r^{(t+1)} + \varepsilon}\, \hat{v}^{(t+1)}. \tag{4.42}$$

Note that the updates (4.41) and (4.42) are well justified as approximations of (4.39) and (4.40). To see this recall that a sequence of L_p norms converges to the L_∞ norm and thus,

$$\begin{aligned}
r_i^{(t+1)} &= \lim_{p\to\infty} \left(s_i^{(t+1)}\right)^{\frac{1}{p}} \\
&= \lim_{p\to\infty} \left[(1-\gamma^p)^{1/p} \left(\sum_{\tau=0}^{t} \gamma^{p(t-\tau)} |g_i^{(\tau)}|^p\right)^{\frac{1}{p}}\right] \\
&= \lim_{p\to\infty} \left(\sum_{\tau=0}^{t} \left(\gamma^{(t-\tau)} |g_i^{(\tau)}|\right)^p\right)^{\frac{1}{p}} \\
&= \max_{\tau=0,\ldots,t} \gamma^{(t-\tau)} \left|g_i^{(\tau)}\right| \\
&= \max\left[\gamma \max_{\tau=0,\ldots,t-1} \gamma^{(t-1-\tau)}\, |g_i^{(\tau)}|,\ |g_i^{(t)}|\right].
\end{aligned}$$

Here, moving from the first line to the second line, we made use of the general representation of exponential smoothing (4.15), and moving from the third line to the fourth line we use the fact that L_p vector norms converge to the L_∞ vector norm as $p \to \infty$. This then justifies (4.41) as a limiting case.

Line Search

So far all the methods we considered were variants of gradient descent. When these are viewed as a special case of the general descent direction method of Algorithm 4.1, updates are of the form $\theta^{(t+1)} = \theta^{(t)} + \theta_{\mathsf{s}}^{(t)}$. In the case of basic gradient descent we have descent steps $\theta_{\mathsf{s}}^{(t)} = -\alpha \nabla C(\theta^{(t)})$, yet in general we can represent the descent step as $\theta_{\mathsf{s}}^{(t)} = \alpha \theta_{\mathsf{d}}^{(t)}$ where we call $\theta_{\mathsf{d}}^{(t)}$ a *descent direction*. With basic gradient descent $\theta_{\mathsf{d}}^{(t)} = -\nabla C(\theta^{(t)})$. In any case, for some prescribed descent direction, the step size $\|\theta_{\mathsf{s}}^{(t)}\|$ is determined by the choice of α.

Line search is an approach where we seek the best step size for a given descent direction in each iteration. Specifically, in iteration t, given a descent direction $\theta_{\mathsf{d}}^{(t)}$, we determine the α for that iteration, denoted as $\alpha^{(t)}$, via,

$$\alpha^{(t)} = \arg\min_{\alpha}\ C\left(\theta^{(t)} + \alpha \theta_{\mathsf{d}}^{(t)}\right). \tag{4.43}$$

That is, each iteration involves a one dimensional minimization problem determining $\alpha^{(t)}$ which is used for the update

$$\theta^{(t+1)} = \theta^{(t)} + \alpha^{(t)}\theta_{\mathrm{d}}^{(t)}.$$

Hence with line search, the next point $\theta^{(t+1)}$ is a minimizer of $C(\cdot)$ along the ray,

$$\{\theta^{(t)} + \alpha\theta_{\mathrm{d}}^{(t)} \ : \ \alpha > 0\}. \tag{4.44}$$

When we pick $\alpha^{(t)}$ to exactly optimize (4.43), the method is called *exact line search*. In very specific cases exact line search can be achieved explicitly with closed form formulas. However in most cases, if we wish to carry out exact line search, we need to numerically optimize the one dimensional optimization problem (4.43) for the best α. An alternative to exact line search is *inexact line search* where we only probe the loss on the ray (4.44) for a few values of α. Details are in the sequel.

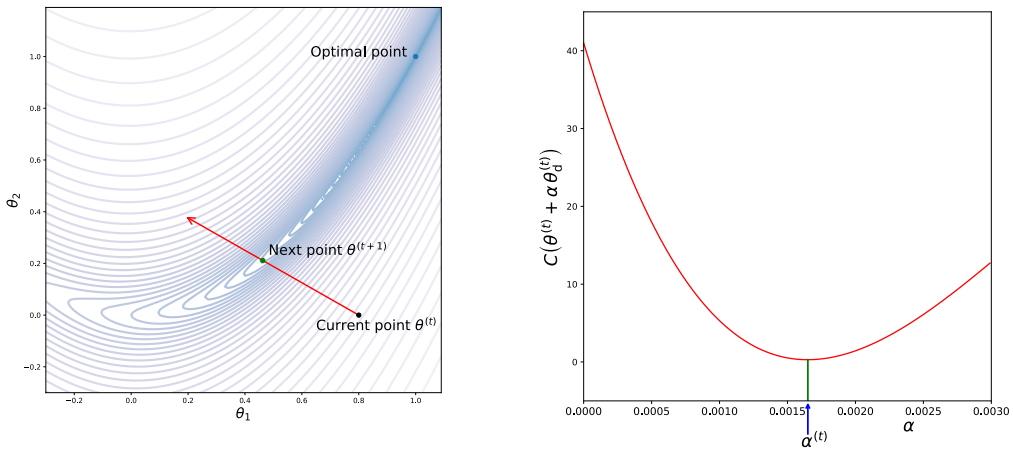


Figure 4.8: One iteration of line search applied to a Rosenbrook function of (4.9). (a) With the current point $\theta^{(t)} = (0.8, 0)$ and the descent direction $\theta_{\mathrm{d}}^{(t)} = -\nabla C(\theta^{(t)})$, we optimize over a given ray. (b) The value of the loss function along the ray with optimal $\alpha^{(t)} \approx 0.00165$.

Note that in practice, for deep learning, neither exact nor inexact line search are used directly. One reason is that loss function evaluations $C(\cdot)$ are computationally demanding and are of a similar order to the computational cost of gradient evaluation. Hence practice has shown that one might as well update descent directions instead of optimizing loss over a fixed ray (4.44). A second reason is that with deep learning we almost always have noisy gradients using stochastic gradient descent or mini-batches, and hence searching for an optimal α on a noisy direction can be practically less useful. Empirical evidence has shown that in such cases, line search techniques are seldom useful on their own right. Nevertheless, understanding line search is important as part of a general background for optimization and is also used in quasi-Newton methods described in Section 4.6 which are sometimes used in deep learning.

As a basic illustration of line search, return to the Rosenbrock function of (4.9) with contours plotted in Figure 4.8 (a). In this plot we are at a point $\theta^{(t)}$ with the descent direction

$\theta_{\mathsf{d}}^{(t)} = -\nabla C(\theta^{(t)})$. The plot in Figure 4.8 (b) presents the one dimensional cross section along the ray (4.44) with that descent direction where we see that the optimal (exact numerical line search) α is at $\alpha^{(t)}$.

It is interesting to analyze the case $\theta_{\mathsf{d}}^{(t)} = -\nabla C(\theta^{(t)})$ which we call *basic gradient descent with exact line search*. In this case, trajectories of such optimization result in "zig-zagging". Specifically we have that every two consecutive descent steps are orthogonal. Namely,

$$\theta_{\mathsf{s}}^{(t)\top}\theta_{\mathsf{s}}^{(t+1)} = 0 \qquad \text{or} \qquad g^{(t)\top}g^{(t+1)} = 0,$$

with $g^{(t)}$ denoting $\nabla C(\theta^{(t)})$. This property is illustrated in Figure 4.9 where we consider gradient descent with exact line search applied to the Rosenbrock function (4.9). We start with initial condition $\theta^{(0)} = (0.4, -0.2)$ from which the first two steps take very long jumps. However, afterwards once the search is in a narrow valley, the zig-zagging phenomenon is apparent and incurs a significant effective slow down on the rate of advance towards the optimal point.

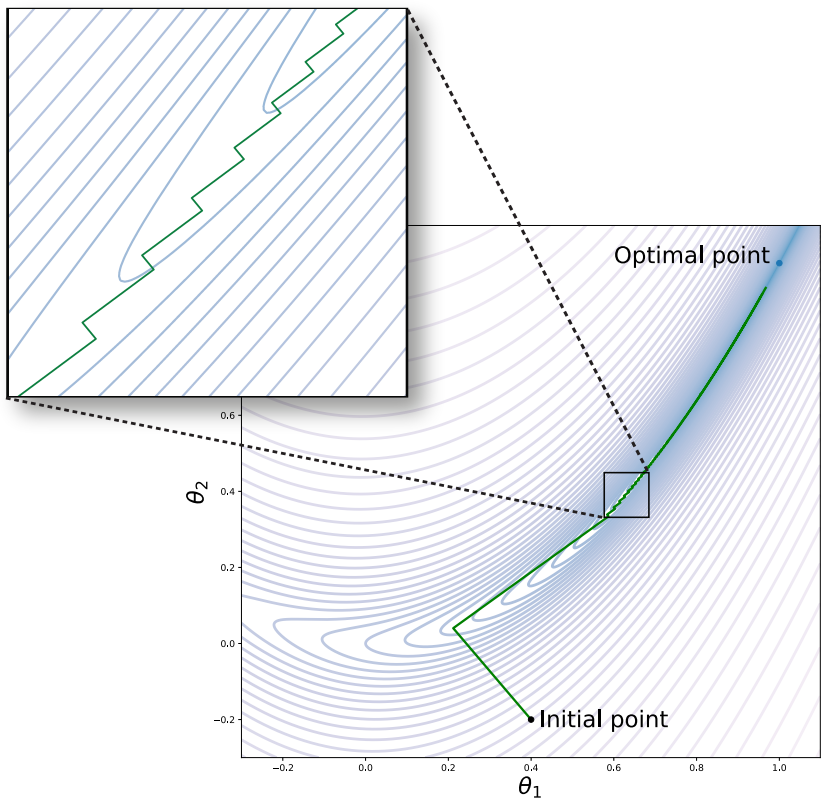


Figure 4.9: Evolution of basic gradient descent with exact line search. We see the zig-zagging property which slows down progress in narrow valleys.

To see the derivation of the zig-zagging property, observe that at α which optimizes (4.43), we have,

$$\frac{\partial C\left(\theta^{(t)} - \alpha g^{(t)}\right)}{\partial \alpha} = 0,$$

or written in terms of the basic definition of the derivative we have,

$$\lim_{h\to 0} \frac{C\left(\theta^{(t)} - (\alpha+h)g^{(t)}\right) - C\left(\theta^{(t)} - \alpha g^{(t)}\right)}{h} = 0. \tag{4.45}$$

We now use the first-order Taylor series expansion as in (A.22) of $C(\cdot)$ around the point $\theta^{(t)} - \alpha g^{(t)}$ for the point $\theta^{(t)} - \alpha g^{(t)} - hg^{(t)}$. This is,

$$C(\theta^{(t)} - \alpha g^{(t)} - hg^{(t)}) = C(\theta^{(t)} - \alpha g^{(t)}) - h\ {g^{(t)}}^\top \nabla C\left(\theta^{(t)} - \alpha g^{(t)}\right) + O(h^2),$$

where $O(h^2)$ is a function such that $O(h^2)/h^2$ goes to a constant as $h \to 0$. Substituting in (4.45) we obtain,

$$\begin{aligned} 0 &= \lim_{h\to 0} \frac{-h\ {g^{(t)}}^\top \nabla C\left(\theta^{(t)} - \alpha g^{(t)}\right) + O(h^2)}{h} \\ &= -{g^{(t)}}^\top \nabla C\left(\theta^{(t)} - \alpha g^{(t)}\right) \\ &= -{g^{(t)}}^\top g^{(t+1)}, \end{aligned}$$

where the last equality holds because $g^{(t+1)} = \nabla C\left(\theta^{(t)} - \alpha g^{(t)}\right)$.

We now visit a powerful numerical technique called the *conjugate gradient method*. This is a topic which may receive special treatment in a general optimization text, yet here we only present it in brief as an application of exact line search. Conjugate gradient is not typically used directly for deep learning, yet since it is useful for approximately solving certain large systems of linear equations, it sometimes finds its way as an aid to other deep learning algorithms.

Suppose the objective function is of the following quadratic form,

$$C(\theta) = \frac{1}{2}\theta^\top A\,\theta - b^\top \theta, \tag{4.46}$$

where A is a $d \times d$ dimensional symmetric positive definite matrix, and $b \in \mathbb{R}^d$. We have the Hessian $\nabla^2 C(\theta) = A$ for every θ and since A is positive definite, $C(\cdot)$ is strictly convex and there is a unique global minimum. We further have $\nabla C(\theta) = A\theta - b$ and this means that the equation $\nabla C(\theta) = 0$ with the unique solution $\theta^* = A^{-1}b$ minimizes $C(\theta)$. Hence the system of equations,

$$A\theta = b, \qquad \text{with } A \text{ positive definite}, \tag{4.47}$$

can be solved by solving the optimization problem of minimizing (4.46). Thus one may view the conjugate gradient method as an algorithm for (approximately) solving a linear system of equations such as (4.47). In particular when the dimension d is excessively large, the conjugate gradient method can be efficient while standard techniques such as Gaussian elimination may fail. We have already seen one application of such equations in Chapter 2, with the normal equations (2.15) where $A = X^\top X$ and $b = X^\top y$.

When carrying out line search on an objective function as (4.46), the cross section above the ray (4.44) for any direction is a parabola and hence exact line search has an explicit

solution for (4.43). Conjugate gradient carries out iterations of such line search, where in each iteration the search direction $\theta_{\mathrm{d}}^{(t)}$ is wisely chosen.

Specifically, at iteration t with current point $\theta^{(t)}$ and search direction $\theta_{\mathrm{d}}^{(t)}$, the next point determined via (4.43) with (4.46) is,

$$\theta^{(t+1)} = \theta^{(t)} + \alpha^{(t)}\theta_{\mathrm{d}}^{(t)}, \qquad \text{with} \qquad \alpha^{(t)} = -\frac{{\theta_{\mathrm{d}}^{(t)}}^{\top}\left(A\,\theta^{(t)} - b\right)}{{\theta_{\mathrm{d}}^{(t)}}^{\top} A\,\theta_{\mathrm{d}}^{(t)}}.$$

Then the next search direction is determined as linear combination of the next gradient and the current search direction. Specifically,

$$\theta_{\mathrm{d}}^{(t+1)} = -\nabla C(\theta^{(t+1)}) + \beta^{(t)}\theta_{\mathrm{d}}^{(t)}, \qquad \text{with} \qquad \beta^{(t)} = \frac{\nabla C(\theta^{(t+1)})^{\top} A\,\theta_{\mathrm{d}}^{(t)}}{{\theta_{\mathrm{d}}^{(t)}}^{\top} A\,\theta_{\mathrm{d}}^{(t)}}.$$

This coefficient $\beta^{(t)}$ is designed such that the current search direction $\theta_{\mathrm{d}}^{(t)}$ and the next search direction $\theta_{\mathrm{d}}^{(t+1)}$ are *conjugate* with respect to A in the sense that, ${\theta_{\mathrm{d}}^{(t+1)}}^{\top} A\,\theta_{\mathrm{d}}^{(t)} = 0$. This conjugacy ensures desirable properties for the method and in particular implies that a minimum is reached in d iterations. Analysis and further motivation of conjugate gradient are beyond our scope. Our presentation here is merely to show that conjugate gradient is an application of exact line search.[25]

Inexact Line Search

As discussed above, one may use exact line search to solve the univariate optimization (4.43) optimally in each iteration. An alternative is to only probe a few specific values of $\alpha^{(t)}$ according to some predefined set of rules and stop at the best probed value. This is called *inexact line search* and it is especially attractive when there is no analytical solution of (4.43). Inexact line search may significantly reduce the computational cost per iteration in comparison to the repetitive application of a univariate numerical optimization technique.

A typical approach for inexact line search is *backtracking* (also known as *backtracking line search*). The idea is to start with some predetermined maximal $\alpha_0 > 0$ and then decrease it multiplicatively until a stopping condition is met. Specifically, we evaluate $C(\cdot)$ on the ray (4.44) with candidate α values such as

$$\alpha_0,\ \frac{2}{3}\alpha_0,\ \left(\frac{2}{3}\right)^2\alpha_0,\ \left(\frac{2}{3}\right)^3\alpha_0, \ldots,$$

where the decrease factor of $\frac{2}{3}$ can be tuned to be any factor in $(0, 1)$. We then stop at the instant at which the stopping condition is met. Note that this search takes place at every iteration t.

One very common stopping condition is known as the *Armijo condition* or the *first Wolfe condition*. Under this condition, at each iteration t, we decrease α values as above until it

[25] Note that in each iteration, the conjugate gradient traverses the steepest direction under the norm $\|u\|_A := \sqrt{u^{\top} A u}$, $u \in \mathbb{R}^d$.

satisfies

$$C\left(\theta^{(t)} + \alpha\theta_{\mathsf{d}}^{(t)}\right) \leq C(\theta^{(t)}) + \beta\,\alpha\,\nabla_{\mathrm{d}}C(\theta^{(t)}), \tag{4.48}$$

where $\beta \in [0,1]$ is a parameter and ∇_{d} is shorthand notation for the directional derivative in the direction $\theta_{\mathsf{d}}^{(t)}$; for directional derivatives recall (A.8) from Appendix A. That is,

$$\nabla_{\mathrm{d}}C(\theta^{(t)}) = \nabla_{\theta_{\mathsf{d}}^{(t)}}C(\theta^{(t)}) = {\theta_{\mathsf{d}}^{(t)}}^{\top}\nabla C(\theta^{(t)}).$$

To get an intuition for (4.48), suppose that $\beta = 1$, then the right hand side of (4.48) is equal to the first-order Taylor approximation of $C\left(\theta^{(t)} + \alpha\,\theta_{\mathsf{d}}^{(t)}\right)$. Therefore, the decrease in the objective suggested by the first Wolfe condition is at least as good as the prediction by the first-order approximation. On the other extreme, if $\beta = 0$, we select α such that $C\left(\theta^{(t)} + \alpha\,\theta_{\mathsf{d}}^{(t)}\right) \leq C(\theta^{(t)})$, which is also acceptable as it guarantees that the next step at least does not increase the objective.

Beyond the condition (4.48), there are additional common stopping conditions that one may employ together with backtracking line search. One such condition known as the *second Wolfe condition* is,

$$\nabla_{\mathrm{d}}C\left(\theta^{(t)} + \alpha\,\theta_{\mathsf{d}}^{(t)}\right) \geq \gamma\nabla_{\mathrm{d}}C(\theta^{(t)}), \tag{4.49}$$

where $\gamma \in (0,1)$. Another common condition known as the *strong Wolfe condition* is,

$$\left|\nabla_{\mathrm{d}}C\left(\theta^{(t)} + \alpha\,\theta_{\mathsf{d}}^{(t)}\right)\right| \leq -\gamma\nabla_{\mathrm{d}}C(\theta^{(t)}). \tag{4.50}$$

This condition forces $\alpha^{(t)}$ to lie close to the solution of exact line search. The first Wolfe condition and the above strong Wolfe condition together are called the *strong Wolfe conditions*. These conditions are sometimes used with quasi-Newton methods described below.

4.6 Concepts of Second-Order Methods

All the optimization methods we studied so far were based on gradient evaluation and implicitly use the first-order Taylor approximation of the objective function $C(\theta)$. In particular, the gradient of the objective function helps us determine the direction of the next step. However, by using curvature information captured via the second derivatives, or Hessian $\nabla^2 C(\theta)$, algorithms can take steps that move farther and are more precise.

In this section, we present some well-known second-order Taylor approximation based optimization methods, simply referred to as *second-order methods*. These methods also fall within the general descent direction framework of Algorithm 4.1, yet their evaluation of the descent step $\theta_{\mathsf{s}}^{(t)}$ either uses the Hessian explicitly, or employs some approximated estimate for Hessian vector products.

While today's deep learning practice rarely involves second-order methods, these methods have huge potential. Indeed, if employed efficiently, their application in general deep learning training can yield significant performance improvement. Hence, understanding basic principles of second-order techniques is useful for the mathematical engineering of deep learning.

This section starts by exploring simple concepts of second-order univariate optimization where we introduce *Newton's method* which is based on the first and second derivative of the objective function. We also present the *secant method* which is only based on the first derivative. We then move on to Newton's method in the general multivariate case where the Hessian matrix plays a key role. Indeed, for simple neural networks such as logistic regression of Chapter 3, a method such as Newton is highly applicable.

For general deep learning with $\theta \in \mathbb{R}^d$, where d is typically huge and the loss function is complicated, it is hopeless to use explicit expressions or automatic differentiation for the Hessian $\nabla^2 C(\theta)$. In such a case, one may use *quasi-Newton* methods that only approximate the Hessian via gradients. Such application of second-order methods to deep learning is a contemporary active area of research which we are not able to cover fully. Instead, our presentation builds up towards a popular method called the *limited-memory BFGS (Broyden–Fletcher–Goldfarb–Shanno)* algorithm, or *L-BFGS* for short. With this we aim to open up horizons for the reader to the multitude of optimization technique variations that one may consider.

The Univariate Case

Newton's method is an iterative method that uses the quadratic approximation, also called the second-order Taylor approximation (A.22), of the objective function around the current point and finds the next step by optimizing the quadratic function. For ease of understanding, first consider the univariate case where $\theta \in \mathbb{R}$. Assume that the objective function $C(\theta)$ is twice differentiable. Suppose we are at $\theta^{(t)}$ in the t-th iteration and let $Q^{(t)}(\theta)$ be the quadratic approximation of the objective function $C(\theta)$ around $\theta^{(t)}$, given by

$$C(\theta) \approx Q^{(t)}(\theta) = C(\theta^{(t)}) + (\theta - \theta^{(t)})C'(\theta^{(t)}) + \frac{(\theta - \theta^{(t)})^2}{2}C''(\theta^{(t)}),$$

where $C'(\cdot)$ and $C''(\cdot)$ are the first and second derivatives respectively.

Note that $Q^{(t)}(\theta)$ depends on $\theta^{(t)}$. Further observe that $C(\theta)$ and $Q^{(t)}(\theta)$ take the same value when $\theta = \theta^{(t)}$. To find the next step, $\theta^{(t+1)}$, we minimize $Q^{(t)}(\theta)$ by setting its derivative to zero and obtain,

$$C'(\theta^{(t)}) + (\theta - \theta^{(t)})C''(\theta^{(t)}) = 0,$$

with the corresponding solution (for the variable θ),

$$\theta^{(t+1)} = \theta^{(t)} - \frac{C'(\theta^{(t)})}{C''(\theta^{(t)})}, \tag{4.51}$$

whenever $C''(\theta^{(t)}) \neq 0$. Newton's method (for univariate function optimization) starts at some initial point $\theta^{(0)}$ and then iterates (4.51) until some specified stopping criterion is met. If converged to some point θ^*, then $C'(\theta^*) = 0$ implying that the point is a local minimum, local maximum, or saddle point.[26]

Any quadratic function is either strictly convex or strictly concave depending on the leading coefficient. Therefore, if $C(\theta)$ itself a strictly convex quadratic function, it has a unique global minimum, and since the third order derivative of $C(\theta)$ is zero, the quadratic approximation

[26] Note that when considered in the context of general root finding of the form $F(\theta) = 0$ beyond the case of $F(\cdot) = C'(\cdot)$, the method is sometimes called *Newton-Raphson*.

$Q^{(t)}(\theta)$ is identically equal to $C(\theta)$. In such a case, Newton's method finds the minimum in a single iteration of (4.51) for any initial point $\theta^{(0)}$. This property hints at the speed at which Newton's method can move towards the minimum since any smooth function is well approximated by a quadratic function in the neighbourhood of a local minimum.

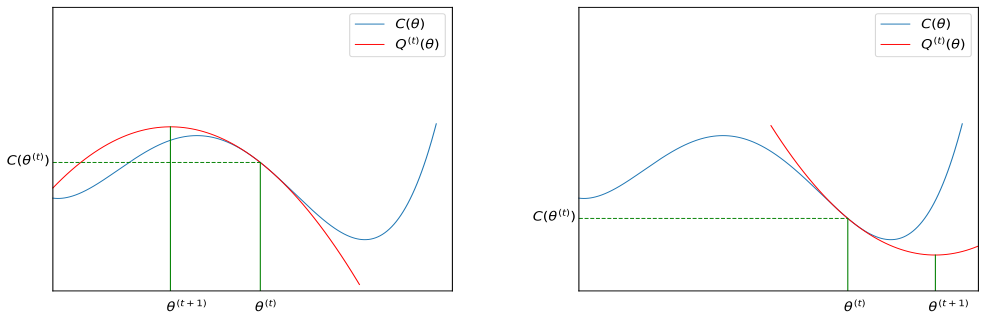


Figure 4.10: Illustration of updates in Newton's method. (a) The approximating quadratic function $Q^{(t)}(\theta)$ is concave. (b) The approximating quadratic function $Q^{(t)}(\theta)$ is convex.

While potentially very efficient, in its basic form, Newton's method exhibits several instability issues. First observe that the method can end up in a local maximum even though the goal is minimization. This is made apparent in Figure 4.10 which illustrates single step updates with Newton's method on a function that has both a local maximum and a local minimum. In (a) a step is taken and it approaches (yet overshoots) the local maximum. In (b) a step is taken (and also overshoots) the local minimum. As is evident, the initial point $\theta^{(t)}$ for both (a) and (b) are near each other, yet the subtle difference in the sign of $C''(\theta^{(t)})$ implies movement in completely different directions. In both cases, with another iteration the extremum point would have been nearly reached. That is (a) would end up very near the local maximum and similarly with another iteration (b) ends up very near the local minimum.

In general, the Newton update (4.51) is very sensitive to the second derivative and if $C''(\theta^{(t)})$ is very close to zero, then $\theta^{(t+1)}$ can be too far from $\theta^{(t)}$, and thus, the quadratic approximation is not appropriate. That is, *inflection points*[27] on the $\theta \in \mathbb{R}$ axis cause havoc with Newton's method. Similarly, as we see in the sequel for the multivariate case, points $\theta \in \mathbb{R}^d$ where the Hessian $\nabla^2(\theta)$ is nearly singular (eigenvalues near zero) are also problematic in the same way.

To gain insights into some of the issues that may occur with Newton's method, let us consider some examples. The sensitivity of the method can cause *overshoot* of the minimum. Consider for example

$$C(\theta) = (\theta - 3)^2(\theta + 3). \tag{4.52}$$

We have $C'(\theta) = 3(\theta - 3)(\theta + 1)$, $C''(\theta) = 6(\theta - 1)$, and the function has a local minimum at $\theta^* = 3$ with $C(\theta^*) = 0$. The descent step resulting from (4.51) is thus,

$$\theta_{\mathsf{s}}^{(t)} = -\frac{C'(\theta^{(t)})}{C''(\theta^{(t)})} = \frac{1}{2}\frac{(\theta^{(t)} - 3)(\theta^{(t)} + 1)}{1 - \theta^{(t)}}.$$

[27] An *inflection point* for a twice differentiable function $C : \mathbb{R} \to \mathbb{R}$ is a point θ where $C''(\theta) = 0$.

Say we start at $\theta^{(0)} = 1.5$ where $C(\theta^{(0)}) = 10.125$. At this point the descent step is $\theta_{\mathsf{s}}^{(0)} = 3.75$ implying that $\theta^{(1)} = \theta^{(0)} + \theta_{\mathsf{s}}^{(0)} = 5.25$. This overshoot of the local minimum yields $C(\theta^{(1)}) \approx 41.77$ which is about a four-fold increase in the loss value. Note though that in this case (as the reader may readily verify via simple calculations), the iterations that follow (presented to 4 decimal points precision) are,

$$\theta^{(2)} = 3.5956, \quad \theta^{(3)} = 3.0683, \quad \theta^{(4)} = 3.0011,$$

and in fact the error for $\theta^{(6)}$ is already less than 10^{-13}. Thus we see in this example that there is initial significant overshoot which is later corrected and is followed by very quick convergence.

Worse than an overshoot is a situation which results in moving away from the local minimum of interest. We have already seen an example of this in Figure 4.10 (a) where starting to the left of the inflection point yields convergence to a local maximum. Further, for the example function (4.52) this will occur if $\theta^{(0)} < 1$.

An additional phenomenon that may occur is *oscillation*. As an extreme example, consider,

$$C(\theta) = -\frac{1}{4}\theta^4 + \frac{5}{2}\theta^2, \qquad \text{which implies} \qquad \theta_{\mathsf{s}}^{(t)} = \frac{\theta^{(t)}({\theta^{(t)}}^2 - 5)}{5 - 3{\theta^{(t)}}^2}.$$

If $\theta^{(t)} = 1$, then we get $\theta_{\mathsf{s}}^{(t)} = -2$ and thus $\theta^{(t+1)} = -1$. Further, in the next step we get $\theta_{\mathsf{s}}^{(t+1)} = 2$ which brings us back to $\theta^{(t+2)} = 1$. Such undesirable perfect oscillation is a singular phenomenon and is not likely to occur in practice. However approximate oscillation may sometimes occur and persist for multiple iterations until dampening.

In practice, and especially when considering multi-dimensional generalizations, *trust region methods* can mitigate problems such as overshoot, movement in the wrong direction, or oscillations. These methods incorporate conditions which disallow steps of large step size beyond some predefined or adaptive thresholds. Trust region methods also incorporate first-order methods as a backup. We do not cover the details further.

A final concern with Newton's method is that it requires both the first derivative $C'(\cdot)$ and the second derivative $C''(\cdot)$. In some scenarios these derivatives are easy to obtain, yet in some other cases the univariate objective may be a complicated function and even the application of automatic differentiation techniques may be inadequate for evaluating $C''(\cdot)$. This motivates the introduction of the *secant method* which is a modification of Newton's method where the second derivative in each iteration is approximated by the last two iterations. Namely, we use the approximation,

$$C''(\theta^{(t)}) \approx \frac{C'(\theta^{(t)}) - C'(\theta^{(t-1)})}{\theta^{(t)} - \theta^{(t-1)}}, \tag{4.53}$$

which becomes precise when $\theta^{(t)}$ and $\theta^{(t-1)}$ are not too far from each other. With this approximation, updates of the secant method are given by

$$\theta^{(t+1)} = \theta^{(t)} - \frac{\theta^{(t)} - \theta^{(t-1)}}{C'(\theta^{(t)}) - C'(\theta^{(t-1)})} C'(\theta^{(t)}),$$

where we initialize $\theta^{(0)}$ and $\theta^{(1)}$ to be close but not equal. In general, the secant method may require more iterations for convergence than Newton's method. The study of general numerical analysis and optimization theory contains an in-depth analysis of such convergence rates. This is a topic that we do not cover further.

The secant method also suffers from the problems associated with Newton's method mentioned above, including overshoot, moving in the wrong direction, and oscillation. Nevertheless, like Newton's method, the secant method is generally very powerful and arrives to a local minimum as long as the initial points are within the vicinity of the minimum.

Importantly, note that the secant method can be viewed as a *quasi-Newton method* since it approximates the Hessian (second derivative of a scalar function) with an expression that only depends on the first derivative. As we now generalize to multivariate optimization we should keep in mind that in the context of deep learning, almost any second-order method that one may wish to use should be a quasi-Newton method based on first-order derivatives (gradients), as opposed to explicit or automatically differentiated Hessians (second derivatives). Nevertheless, let us first study Newton's method for multivariate functions.

The Multivariate Case and Hessians

For a twice differentiable multivariate objective function $C : \mathbb{R}^d \to \mathbb{R}$ the update of Newton's method is very similar to the univariate case of (4.51). Specifically, the gradient and the Hessian play the roles of the first and the second-order derivatives, respectivly. We can represent this update as,

$$\theta^{(t+1)} = \theta^{(t)} - H_t^{-1} \nabla C(\theta^{(t)}), \tag{4.54}$$

where $H_t = \nabla^2 C(\theta^{(t)})$ is the Hessian of the objective function at $\theta^{(t)}$, and when H_t is non-singular; compare (4.54) with (4.51).

To see the development of (4.54), as in the univariate case consider the quadratic approximation $Q^{(t)}(\theta)$ of $C(\theta)$ around $\theta^{(t)}$ given by

$$C(\theta) \approx Q^{(t)}(\theta) = C(\theta^{(t)}) + (\theta - \theta^{(t)})^\top \nabla C(\theta^{(t)}) + \frac{1}{2}(\theta - \theta^{(t)})^\top H_t \, (\theta - \theta^{(t)}). \tag{4.55}$$

Then, similar to the univariate case, the next point $\theta^{(t+1)}$ can be determined as the θ such that the gradient of $Q^{(t)}(\theta)$ is zero. Observe that this gradient is

$$\nabla Q^{(t)}(\theta) = \nabla C(\theta^{(t)}) + H_t \, (\theta - \theta^{(t)}),$$

and by equating $\nabla Q^{(t)}(\theta)$ to 0 and representing $\theta - \theta^{(t)}$ as the descent step $\theta_{\mathrm{s}}^{(t)}$,

$$H_t \, \theta_{\mathrm{s}}^{(t)} = -\nabla C(\theta^{(t)}). \tag{4.56}$$

This linear systems of equations for $\theta_{\mathrm{s}}^{(t)}$ can be represented in terms of the elegant update equation (4.54) with descent step $\theta_{\mathrm{s}}^{(t)} = -H_t^{-1} \nabla C(\theta^{(t)})$, in case H_t is non-singular. However, in practice, we typically try to solve (4.56) using a suitable linear solver.[28] Sometimes

[28] Suitability of an algorithm depends on the size of d and any special structure in H_t that we can expect. For example for small d, one may use LU decomposition (Gaussian eliminiation), or QR factorization based methods, yet for large d conjugate gradient or other *Krylov subspace methods* may be suitable.

approximate solutions for (4.56) are also an option. For example one may use the conjugate gradient method when d is large and run for a small number of iterations to only approximately solve (4.56).

In summary, Newton's algorithm for multivariate $C(\cdot)$ starts with some $\theta^{(0)} \in \mathbb{R}^d$. Then at each iteration t in the algorithm with current point $\theta^{(t)}$, we (in principle) evaluate the gradient and Hessian at that point. We then solve or approximately solve (4.56) to obtain $\theta_s^{(t)}$ for the update $\theta^{(t+1)} = \theta^{(t)} + \theta_s^{(t)}$.

In principle, in each iteration, solving (4.56) requires the Hessian matrix $H_t = \nabla^2 C(\theta^{(t)})$ either via a closed form formula as for example in the case of logistic regression (4.7), or more generally via automatic differentiation. However with large d, computing the Hessian or Hessian vector products via automatic differentiation is generally not feasible. Hence in general, Newton's method on its own is not suitable for deep learning. Nevertheless, quasi-Newton methods such as the L-BFGS method, presented below, or various adaptations can be employed.

We also mention that all of the potential problems discussed above for the univariate case can appear in the multivariate case. Overshoot, moving in the wrong direction, or oscillation can occur due to similar reasons as the univariate case. Specifically at points $\theta^{(t)}$ that are far from the local minimum, the quadratic approximation $Q^{(t)}(\theta)$ may be far from $C(\theta)$. Further, there are problems that may occur due to a singular, near-singular, or ill conditioned Hessian matrix H_t. In this case the descent step solution to (4.56) may be very noisy.[29] Nevertheless, Newton's method is very powerful and often when improved via trust region methods and similar techniques, it outperforms first-order methods very well.

To build intuition of why Newton's method often outperforms first-order methods, let us first explore a relationship between the two approaches. If we use the constant diagonal matrix $\frac{1}{\alpha} I$ in place of the Hessian H_t, then Newton's method reduces to the basic gradient descent method with update (2.20) where the descent step is $\theta_s^{(t)} = -\alpha \nabla C(\theta)$. To see this, return to the quadratic approximation (4.55) with H_t replaced by $\frac{1}{\alpha} I$, to obtain,

$$\widetilde{Q}^{(t)}(\theta) = C(\theta^{(t)}) + (\theta - \theta^{(t)})^\top \nabla C(\theta^{(t)}) + \frac{1}{2\alpha}(\theta - \theta^{(t)})^\top (\theta - \theta^{(t)}), \tag{4.57}$$

and thus,

$$\nabla \widetilde{Q}^{(t)}(\theta) = \nabla C(\theta^{(t)}) + \frac{1}{\alpha}(\theta - \theta^{(t)}).$$

Equating this gradient expression to 0 yields the update (2.20). Hence, we notice that the inverse of H_t in Newton's method plays the role of the learning rate α of basic gradient descent. As a consequence, with this basic form of Newton's method, there is no need to calibrate the learning rate, as we do in most gradient descent approaches.[30]

Finally let us get intuition of why Newton's method generally converges to a (local) minimum θ^* faster than gradient descent when starting at a point close to θ^*. For this, return to Figure 2.8 which compares two loss landscapes, one of which is circular and one of which is elliptical. While that figure is created via a quadratic form where Netwon's method would

[29] In practice one often monitors the level of ill-conditioning via the condition number of H_t, or similar measures.

[30] Note that Newton's method can also be cast as a steepest descent method like basic gradient descent, but with descent direction chosen with respect to the Hessian norm, $\|u\|_{H_t} = \sqrt{u^\top H_t\, u},\ u \in \mathbb{R}^d$.

reach the minimum in a single iteration, in more realistic cases when functions are not quadratic forms, they can still be locally well approximated by quadratic forms in the vicinity of θ^* and hence considering Figure 2.8 as a general plot is valid.

It is known that with gradient descent, highly elliptical loss landscapes are difficult for optimization since descent steps may be trapped in valleys, and thus as argued in Section 2.4 in view of Figure 2.8, one often tries to carry out variable transformations to alleviate such problems. In a sense the gradient descent quadratic approximation $\widetilde{Q}^{(t)}(\cdot)$ in (4.57) assumes that the loss landscape is already perfectly circular since it uses a constant diagonal matrix in place of the Hessian matrix. This is the best that basic gradient descent can offer. However, when using Newton's method, the correct local quadratic approximation $Q^{(t)}(\cdot)$ in (4.55) already adapts to the curvature and potential true elliptic nature of the loss landscape. With such an approximation $Q^{(t)}(\cdot)$, minimization over quadratic loss landscapes occurs in a single iteration, and since locally all smooth functions are well approximated by quadratic loss landscapes, when $\theta^{(t)}$ is in the vicinity of θ^*, converge is very quick.

Quasi-Newton Methods

In deep learning, the dimension of θ is very large and it is thus difficult to solve the linear equations (4.56) as required in each iteration of Newton's method. Even in dimensions where these equations are solvable, there is still the basic problem of computing the Hessian matrix, a feat which is typically intractable for mid-size problems and beyond. A popular alternative is offered by the use of quasi-Newton methods where either the Hessian or the inverse of the Hessian matrix are approximated with each iteration. While the study of quasi-Newton methods is mature, and these are used in many areas of applied mathematics and engineering, the development of best practices for application in deep learning is still an active area of research. Here we simply outlines the key ideas of such methods.

The central idea in the type of quasi-Newton methods that we cover is based on an evolving sequence of $d \times d$ matrices, $B_0, B_1, B_2, \ldots$. At each iteration, some *update rule* specifies how B_t is updated based on B_{t-1} and other quantities from the current and previous iteration. By the design of the update rule, each such B_t is kept as positive semidefinite (and hence also non-singular), and in our presentation, B_t^{-1} is ideally a good approximation[31] for the Hessian matrix $H_t = \nabla^2 C(\theta^{(t)})$ associated with the current point $\theta^{(t)}$ of iteration t.

With such a B_t matrix available, we now use the quadratic approximation,

$$Q^{(t)}(\theta) = C(\theta^{(t)}) + (\theta - \theta^{(t)})^\top \nabla C(\theta^{(t)}) + \frac{1}{2}(\theta - \theta^{(t)})^\top B_t^{-1} (\theta - \theta^{(t)}), \tag{4.58}$$

which is similar to Newton's method quadratic approximation (4.55) with the difference that now B_t^{-1} plays the role of the Hessian. Similarly to the case in Newton's method, $Q^{(t)}(\theta)$ of (4.58) can be minimized where now the unique minimizer is $-B_t \nabla C(\theta^{(t)})$. Hence using such a quadratic approximation we have a descent direction,

$$\theta_{\mathsf{d}}^{(t)} = -B_t \nabla C(\theta^{(t)}). \tag{4.59}$$

With quasi-Newton methods, we take a step in the direction specified by $\theta_{\mathsf{d}}^{(t)}$, but we also scale the step with a scalar $\alpha^{(t)} > 0$ which controls the step size. Hence the key update rule

[31] We note that alternative presentations may use B_t as an approximation for the Hessian itself.

for quasi-Newton methods can be stated as,

$$\theta^{(t+1)} = \theta^{(t)} - \alpha^{(t)} B_t \nabla C(\theta^{(t)}), \tag{4.60}$$

which is similar to (4.54). The determination of $\alpha^{(t)}$ is typically obtained via a line search technique such as inexact line search using the second Wolfe condition (4.49).

Algorithm 4.3: Quasi-Newton

Input: Dataset $\mathcal{D} = \{(x^{(1)}, y^{(1)}), \ldots, (x^{(n)}, y^{(n)})\}$,
objective function $C(\cdot) = C(\cdot\,; \mathcal{D})$, and
initial parameter vector θ_{init}

Output: Approximately *optimal* θ

1 $\theta \leftarrow \theta_{\mathsf{init}}$
2 $B \leftarrow B_{\mathsf{init}}$
3 **repeat**
4 $\quad g \leftarrow \nabla C(\theta)$ (Compute gradient)
5 $\quad \theta_{\mathsf{d}} \leftarrow -Bg$ (Determine descent direction)
6 $\quad$ Determine α using θ and θ_{d} (Line search)
7 $\quad \theta \leftarrow \theta + \alpha\theta_{\mathsf{d}}$ (Update parameters)
8 $\quad$ Update B (Update approximation of $\nabla^2 C(\theta)^{-1}$)
9 **until** *termination condition* is satisfied
10 **return** θ

The typical quasi-Newton method is presented in Algorithm 4.3. As in any gradient based algorithm, computation of the gradient is needed, and this is specified in Step 4. The computation of the descent direction (4.59) is in Step 5. In Step 6 we use line search, and Step 7 updates the current parameter based on (4.60). Finally, Step 8 is the update rule, specific to the type of quasi-Newton method used. In the sequel we present BFGS and L-BFGS, each with their own specification of this step.

As this method falls within the realm of a descent direction approach as in Algorithm 4.1, ideally we wish to set θ_{init} in Step 1, at a point close to a minimum, just like all the other descent direction approaches in this chapter. As for the initialization matrix B_{init} in Step 2, ideally we would have a matrix whose inverse is the Hessian at θ_{init}. However, since evaluation of the Hessian matrix or its inverse is typically computationally costly or intractable, we often initialize with $B_{\mathsf{init}} = I$, a $d \times d$ identity matrix.

Now with the general outline of the quasi-Newton approach presented, one may seek good update rules for Step 8. Before we see concrete formulas for these updates within the BFGS and L-BFGS algorithms, let us touch on theoretical aspects that helped develop these methods, and can be used for further analysis of their performance and properties.

Since the Hessian H_t captures the curvature information of the objective function, we may consider the latest two points $\theta^{(t-1)}$ and $\theta^{(t)}$ together with the gradients at these points, $\nabla C(\theta^{(t-1)})$ and $\nabla C(\theta^{(t)})$, for the approximate equality,

$$H_t \left(\theta^{(t)} - \theta^{(t-1)}\right) \approx \nabla C(\theta^{(t)}) - \nabla C(\theta^{(t-1)}). \tag{4.61}$$

This approximation follows from (A.24) of Appendix A, and is a good approximation when $\theta^{(t)}$ and $\theta^{(t-1)}$ are close. It generalizes (4.53) used in the univariate secant method. Note that (4.61) holds with equality if $C(\theta)$ is a quadratic function. Further, with the notation,

$$\theta_{\mathsf{s}}^{(t-1)} = (\theta^{(t)} - \theta^{(t-1)}), \qquad \text{and} \qquad g_{\mathsf{s}}^{(t-1)} = \nabla C(\theta^{(t)}) - \nabla C(\theta^{(t-1)}), \tag{4.62}$$

we have that (4.61) is represented as $H_t \theta_{\mathsf{s}}^{(t-1)} \approx g_{\mathsf{s}}^{(t-1)}$. Now inspired by this approximation, if we are able to select the $d \times d$ positive definite matrix B_t, such that B_t^{-1} is an approximation of H_t, then by considering the approximation as an equality,

$$\theta_{\mathsf{s}}^{(t-1)} = B_t \, g_{\mathsf{s}}^{(t-1)}. \tag{4.63}$$

The equation (4.63) is known as the *secant equation* and it implies that B_t maps the change of gradients $g_{\mathsf{s}}^{(t-1)}$ into the descent direction $\theta_{\mathsf{s}}^{(t-1)}$. A condition related to the secant equation is,

$$g_{\mathsf{s}}^{(t-1)\top} \theta_{\mathsf{s}}^{(t-1)} > 0, \tag{4.64}$$

which is known as the *curvature condition*. Importantly, it can be shown that when the curvature condition is satisfied, there is always a positive definite B_t such that the secant equation is satisfied.[32] In fact, there can be infinitely many solutions. In general, quasi-Newton methods are designed to maintain the curvature condition (4.64) and to provide a unique update rule for B_t.

The curvature condition (4.64) allows us to also gain insight into why line search as in Step 6 of Algorithm 4.3 is needed. In simple or synthetic cases where the loss function $C(\cdot)$ is strictly convex, it can be shown that the curvature condition (4.64) is always satisfied. However, loss functions in deep learning are typically highly non-convex, and thus there is no guarantee of existence of B_t such that the secant equation is satisfied, unless we impose an additional condition on $\alpha^{(t)}$ of (4.60).

If in each iteration t we determine $\alpha^{(t)}$ using inexact line search with the second Wolfe condition (4.49), then the curvature condition (4.64) is guaranteed to hold.[33] To see this, suppose that the second Wolfe condition is applied, then from $\theta^{(t)} = \theta^{(t-1)} + \alpha^{(t-1)} \theta_{\mathsf{d}}^{(t-1)}$, we obtain

$$\nabla C(\theta^{(t)})^\top \theta_{\mathsf{d}}^{(t-1)} \geq \gamma \, \nabla C(\theta^{(t-1)})^\top \theta_{\mathsf{d}}^{(t-1)}, \tag{4.65}$$

where we used the relationship between the gradient and the directional derivative shown in (A.8) of Appendix A. Now the inner product of the curvature condition (4.64) satisfies,

$$\begin{aligned}
g_{\mathsf{s}}^{(t-1)\top} \theta_{\mathsf{s}}^{(t-1)} &= \nabla C(\theta^{(t)})^\top \theta_{\mathsf{s}}^{(t-1)} - \nabla C(\theta^{(t-1)})^\top \theta_{\mathsf{s}}^{(t-1)} \\
&\geq \gamma \alpha^{(t-1)} \nabla C(\theta^{(t-1)})^\top \theta_{\mathsf{d}}^{(t-1)} - \nabla C(\theta^{(t-1)})^\top \theta_{\mathsf{s}}^{(t-1)} \\
&= (\gamma - 1) \alpha^{(t-1)} \nabla C(\theta^{(t-1)})^\top \theta_{\mathsf{d}}^{(t-1)} \\
&= -(\gamma - 1) \alpha^{(t-1)} \nabla C(\theta^{(t-1)})^\top B_{t-1} \nabla C(\theta^{(t-1)}) \\
&> 0.
\end{aligned}$$

[32] It can be shown that if for $u, v \in \mathbb{R}^d$, $u^\top v > 0$, then there exists a symmetric positive definite matrix $B \in \mathbb{R}^{d \times d}$ such that $Bu = v$.

[33] This also holds for the strong Wolfe condition (4.50).

The first equality uses the definition of $g_{\mathsf{s}}^{(t-1)}$ and the step size resulting from (4.60), $\theta_{\mathsf{s}}^{(t-1)} = \alpha^{(t-1)}\theta_{\mathsf{d}}^{(t-1)}$. The inequality that follows is due to (4.65). Finally, the last inequality is because $\gamma < 1$ and B_{t-1} is assumed to be positive definite based on the update rule. We thus see that the second Wolfe condition is a means to ensure that the curvature condition is satisfied.

The BFGS and L-BFGS Update Rules

Let us now focus on the *Broyden-Fletcher-Goldfarb-Shanno* (BFGS) algorithm, and its variant, the *limited-memory BFGS* (L-BFGS) algorithm. In practice, some small deep learning applications already make efficient use of L-BFGS. We mention that one common approach in training, is to first carry out training using standard gradient descent or a variant, and then apply L-BFGS to "complete" the training process. Nevertheless, to date, the application of L-BFGS and other variants of advanced quasi-Newton methods is still not widespread in large scale deep learning.

We present the update rules for these two algorithms which specify Step 8 of Algorithm 4.3, keeping in mind that $B_0 = B_{\mathsf{init}}$. Each quasi-Newton iteration of Algorithm 4.3 already bears the computational cost of computing the gradient, the cost of several function evaluations within the line search, and other costs associated with determining the descent direction and updating of parameters. The computational cost of the update rule in Step 8 is generally heavy. We contrast the computational cost of the BFGS update rule and that of the L-BFGS update rule, below.

The BFGS update rule is,

$$B_t = V_{t-1}^\top B_{t-1} V_{t-1} + \rho^{(t-1)} \theta_{\mathsf{s}}^{(t-1)} \theta_{\mathsf{s}}^{(t-1)\top}, \tag{4.66}$$

where the positive scalar $\rho^{(t-1)}$ and the matrix V_{t-1} are respectively given by,

$$\rho^{(t-1)} = \frac{1}{g_{\mathsf{s}}^{(t-1)\top} \theta_{\mathsf{s}}^{(t-1)}} \quad \text{and} \quad V_{t-1} = I - \rho^{(t-1)} g_{\mathsf{s}}^{(t-1)} \theta_{\mathsf{s}}^{(t-1)\top}. \tag{4.67}$$

Observe the use of the rank one matrices $\theta_{\mathsf{s}}^{(t-1)}\theta_{\mathsf{s}}^{(t-1)\top}$ and $g_{\mathsf{s}}^{(t-1)}\theta_{\mathsf{s}}^{(t-1)\top}$ using (4.62). The positivity of $\rho^{(t-1)}$ is maintained by the curvature condition (4.64). Importantly, it can be shown that each matrix in the sequence $B_0, B_1, \ldots$, resulting from this update rule is positive definite. We omit the details.

Note that due to the matrix multiplication $V_{t-1}^\top B_{t-1} V_{t-1}$, the computational cost of updating B_t using (4.66) is $O(d^2)$. This is a reasonable cost for small d, yet for larger d as in some deep learning applications, it renders BFGS too costly.

We mention that the development of the update rule in (4.66) and (4.67) can be cast as a minimization problem where we seek the closest possible positive definite B_t to the previous B_{t-1} under a specific matrix norm. The minimization is subject to the constraint that the secant equation (4.63) holds. We omit the details of this derivation yet mention that the matrix norm used for BFGS is a *weighted Frobenius norm*. See the notes and references at the end of the chapter for more information.

The L-BFGS algorithm overcomes the computational cost of BFGS. Its update rule is not for maintaining the matrix B_t in memory directly, but is rather based on keeping a limited history (or limited memory) of the constituent vectors $\theta_{\mathsf{s}}^{(t)}$ and $g_{\mathsf{s}}^{(t)}$. Observe that in principle, one can unroll the BFGS update rules (4.66) and (4.67) to obtain,

$$\begin{aligned} B_t = {} & V_{t-1}^\top V_{t-2}^\top \cdots V_0^\top \, B_{\mathsf{init}} \, V_0 \cdots V_{t-2} V_{t-1} \\ & + \rho^{(t-1)} \theta_{\mathsf{s}}^{(t-1)} {\theta_{\mathsf{s}}^{(t-1)}}^\top \\ & + \rho^{(t-2)} V_{t-1}^\top \theta_{\mathsf{s}}^{(t-2)} {\theta_{\mathsf{s}}^{(t-2)}}^\top V_{t-1} \\ & \quad \vdots \\ & + \rho^{(0)} V_{t-1}^\top V_{t-2}^\top \cdots V_0^\top \, \theta_{\mathsf{s}}^{(0)} {\theta_{\mathsf{s}}^{(0)}}^\top \, V_0 \cdots V_{t-2} V_{t-1}. \end{aligned} \tag{4.68}$$

Further observe that this representation of B_t is a function of the sequence $\{\theta_{\mathsf{s}}^{(k)}, g_{\mathsf{s}}^{(k)}\}$ for $k = 0, 1, \ldots, t-1$ and B_{init}. Hence in principle, assuming that B_{init} is diagonal, if we wish to compute the matrix-vector product $B_t \nabla C(\theta^{(t)})$, we can do so with all multiplications being vector-vector multiplications. This is due to the fact that each V_t matrix is composed of the outer products $g_{\mathsf{s}}^{(t-1)} {\theta_{\mathsf{s}}^{(t-1)}}^\top$. Yet obviously, for non-small t such multiplications become inefficient as the number of terms in the expression (4.68) grows.

The L-BFGS idea is to use an approximate version $\widetilde{B}_t$ of B_t that can be stored indirectly by storing only the latest m pairs of vectors $\{g_{\mathsf{s}}^{(k)}, \theta_{\mathsf{s}}^{(k)}\}$, for $k = t-m, \ldots, t-1$. The hyper-parameter m is the length of the history (or memory) and is typically chosen to be much smaller than d. We modify (4.68) with an approximation which limits the history used to the last m iterations. For this we drop the last $t-m$ terms of (4.68), and in the first term $V_{t-1}^\top V_{t-2}^\top \cdots V_0^\top \, B_{\mathsf{init}} \, V_0 \cdots V_{t-2} V_{t-1}$, we replace the middle multiplicands, $V_{t-m-1}^\top \cdots V_0^\top \, B_{\mathsf{init}} \, V_0 \cdots V_{t-m-1}$, with a symmetric positive definite matrix $B_{\mathsf{init}}^{(t)}$. This modification yields $\widetilde{B}_t \approx B_t$, with the form,

$$\begin{aligned} \widetilde{B}_t = {} & V_{t-1}^\top V_{t-2}^\top \cdots V_{t-m}^\top \, B_{\mathsf{init}}^{(t)} \, V_{t-m} \cdots V_{t-2} V_{t-1} \\ & + \rho^{(t-1)} \theta_{\mathsf{s}}^{(t-1)} {\theta_{\mathsf{s}}^{(t-1)}}^\top \\ & + \rho^{(t-2)} V_{t-1}^\top \theta_{\mathsf{s}}^{(t-2)} {\theta_{\mathsf{s}}^{(t-2)}}^\top V_{t-1} \\ & \quad \vdots \\ & + \rho^{(t-m)} V_{t-1}^\top V_{t-2}^\top \cdots V_{t-m}^\top \, \theta_{\mathsf{s}}^{(t-m-1)} {\theta_{\mathsf{s}}^{(t-m-1)}}^\top \, V_{t-m} \cdots V_{t-2} V_{t-1}. \end{aligned} \tag{4.69}$$

Note that $B_{\mathsf{init}}^{(t)}$ is selected for each iteration t and is usually set as a diagonal matrix to reduce the storage and computational costs. A practically effective choice is

$$B_{\mathsf{init}}^{(t)} = \frac{{g_{\mathsf{s}}^{(t-1)}}^\top \theta_{\mathsf{s}}^{(t-1)}}{{g_{\mathsf{s}}^{(t-1)}}^\top g_{\mathsf{s}}^{(t-1)}} I.$$

With this choice for $B_{\mathsf{init}}^{(t)}$, the expression (4.69) allows us to compute $\widetilde{B}_t \nabla C(\theta^{(t)})$ efficiently. In particular, $\widetilde{B}_t \nabla C(\theta^{(t)})$ can be computed using $O(m)$ inner products between d-dimensional vectors, and thus the per iteration computational cost for the L-BFGS algorithm is $O(md)$, while requiring to store only the latest m pairs of vectors $\{g_{\mathsf{s}}^{(k)}, \theta_{\mathsf{s}}^{(k)}\}$, $k = t-m, \ldots, t-1$,

which takes $O(md)$ storage space. With m much smaller than d, the L-BFGS algorithm requires far less storage than the BFGS algorithm. Finally we note that in terms of Algorithm 4.3, the update rule in Step 8 simply needs to push the vectors $\{g_{\mathbf{s}}^{(t)}, \theta_{\mathbf{s}}^{(t)}\}$ onto a queue while popping the vectors $\{g_{\mathbf{s}}^{(t-m)}, \theta_{\mathbf{s}}^{(t-m)}\}$ out of that queue. The bulk of the algorithm is in Step 5 with the efficient matrix-vector multiplication using the expressions (4.69) and (4.67).

Notes and References

Mathematical optimization is a classical subject and some of the general texts on this subject are [37] where linear optimization is covered, [272] where both linear and non-linear optimization methods are covered, and [35] where the focus is on non-linear methods. Further texts focusing on convexity, convex analysis, and convex optimization are [27], [32], and [55]. Many of the algorithms in these texts describe descent direction methods, yet some examples of optimization algorithms that are not descent direction methods include direct methods (also known as zero-order or black box) such as coordinate and pattern search methods; see [238] or [311] for an overview. *The Rosenbrock function* was developed in [354] within the context of multivariate function optimization and since then it became popular for basic optimization test cases and illustrations.

As mentioned in the Chapter 2 notes and references, gradient descent is attributed to Cauchy from 1847 with [72] ; see [253] for an historical account. *Stochastic gradient descent* was initially introduced in 1951 with [350] in the context of optimizing the expectation of a random variable over a parametric family of distributions. See [229], [230], [231], [255], and [349] for early references using stochastic gradient descent approaches. After the initial popularity of this approach for general optimization, other numerical optimization methods were developed and stochastic gradient descent's popularity decreased. Years later, it became relevant again due to developments in deep learning; see [48] and [52] for early references in the context of deep learning. Interesting works that attempt to explain the suitability of stochastic gradient descent for deep learning are [49], [50], and [51]. Relationships between the use of mini-batches and stochastic gradient descent are studied in [202], [203], and [261]. Practical recommendations of gradient based methods can be found in [29].

The ADAM algorithm was introduced in [233]. See [43] and [345] for additional papers that study its performance. Ideas of momentum appeared in optimization early on in [335]. RMSprop appeared in lectures by Geoff Hinton in a 2014 course and has since become popular. Adagrad appeared earlier on in [112]. An interesting empirical study considering ADAM and related methods is [82]. See [238] and [356] for a detailed study of momentum based methods.

The basic idea of *automatic differentiation* dates back to the 1950's; see [25] and [313]. The idea of computing the partial derivatives using forward mode automatic differentiation is attributed to Wengert [420]. Even though ideas of backward mode automatic differentiation first appeared around 1960 and used in control theory by the end of that decade. The 1980's paper [386] is generally attributed for presenting backward mode automatic differentiation in the form we known it now. Backward mode automatic differentiation for deep neural networks is referred to as the backpropagation algorithm, a concept that we cover in detail in Chapter 5. In fact, ideas for the backpropagation algorithm were developed independently of some of the automatic differentiation ideas; see [173] for an historical overview as well as the notes and references at the end of Chapter 5 for more details.

In recent years, with the deep learning revolution, automatic differentiation has gained major popularity and is a core component of deep learning software frameworks such as *TensorFlow* [1], *PyTorch* [324], and others. In parallel, the application of automatic differentiation for a variety of scientific applications has also been popularized with technologies such as *Flux.jl* [201] in Julia and *JAX* [57] in Python. See [24] for a survey in the context of machine learning and [147] for a more dated review. Additional differentiable programming frameworks that implement automatic differentiation include CasADi, Autograd, AutoDiff, JAX, Zygote.jl, Enzyme, and Chainer. Much of the complexity of differentiable programming software frameworks is centered around the use of GPUs as well as the placement and management of data in GPU memory.

Nesterov momentum was introduced in [305]. See [362], [363], and [431] for some comparisons of Nesterov momentum with the standard momentum approach. The Nadam algorithm was introduced in [109] to incorporate Nesterov momentum into ADAM and is now part of some deep learning frameworks. The Adadelta method appeared in [440] prior to the appearance and growth in popularity of ADAM. The Adamax method appeared in the same paper as ADAM, [233], as an extension. Line search techniques are central in classical first-order optimization theory. Inexact line search techniques using Wolfe conditions first appeared in [422] and [423]. *Backtracking line search*, also known as *Armijo line search*, first appeared in [16]. See for example chapter 3 of [311] and chapter 4 of [238] for overviews of exact and inexact line search algorithms. Concepts of the conjugate gradient method first appeared in [177] for solving linear equations with symmetric positive

semidefinite matrices. Since then, the method was incorporated in many numerical algorithms; see for example chapter 11 of [141] and chapter 5 of [311] for more details.

Despite the success of second-order methods in statistics, in general numerical optimization, and in engineering, they are yet to make it into mainstream large scale deep learning. Perhaps one reason for this is that when implemented without care, they are much slower than first-order methods for high-dimensional models such as deep neural networks. Nevertheless, their efficient incorporation in deep learning is still an active area of research. The origins of Newton's method, also known as the *Newton-Raphson method*, can be dated back to 17th century; see chapter 2 of [237] for an account on the history of optimization methods. For a detailed overview see [47].

Even though the secant method can be thought of as an approximation to Newton's method, from an historical perspective, it is several decades older than Newton's method; see for example [321] for origins and an account of the evolution of the secant method. Quasi-Newton methods are more modern with the first technique, the *Davidon-Fletcher-Powell* (DFP) method, appearing in 1959 as a technical report [99] and later in 1991 published as [100]. In the interim, the DFP method was further modified in the early 1960's with [124]. The BFGS method materialized in the 1960's and 1970's, and the L-BFGS extension was introduced in 1980 with [310]. See also [123] for an overview of these methods. See [98] and [266] for a discussion of convergence properties of BFGS and L-BFGS respectively. Information about the weighted Frobenius norm used for constructing BFGS updates is in chapter 6 of [311]. A related second order method is the Levenberg–Marquardt algorithm; see chapter 18 of [56] for an introduction. Trust region methods are believed to have originated from [254]; see [439] for a survey. Finally we suggest chapter 5 of [311] as a review of *quasi-Newton methods*.