3 First-Order Optimization Techniques

3.1 Introduction

In this chapter we describe fundamental optimization algorithms that leverage the *first derivative* or *gradient* of a function. These techniques, collectively called *first-order optimization methods*, are some of the most popular local optimization algorithms used to tackle machine learning problems today. We begin with a discussion of the *first-order optimality condition* which codifies how the first derivative(s) of a function characterizes its minima. We then discuss fundamental concepts related to hyperplanes, and in particular the first-order Taylor series approximation. As we will see, by exploiting a function's first-order information we can construct local optimization methods, foremost among them the extremely popular *gradient descent algorithm*, that naturally determine high-quality descent directions at a cost that is very often cheaper than even the coordinate-wise approaches described in the previous chapter.

3.2 The First-Order Optimality Condition

In Figure 3.1 we show two simple quadratic functions, one in two dimensions (left panel) and one in three dimensions (right panel), marking the global minimum on each with a green point. Also drawn in each panel is the line/hyperplane tangent to the function at its minimum point, also known as its first-order Taylor series approximation (see Appendix Section B.9 if you are not familiar with the notion of Taylor series approximation). Notice in both instances that the tangent line/hyperplane is perfectly flat. This sort of behavior is universal (for differentiable functions) regardless of the function one examines, and it holds regardless of the dimension of a function's input. That is, minimum values of a function are naturally located at *valley floors* where a tangent line or hyperplane is perfectly flat, and thus has zero-valued slope(s).

Because the *derivative* (see Appendix Section B.2) of a single-input function or the *gradient* (see Appendix Section B.4) of a multi-input function at a point gives precisely this slope information, the value of first-order derivatives provide a convenient way of characterizing minimum values of a function g. When $N = 1$, any point v of the single-input function $g(w)$ where

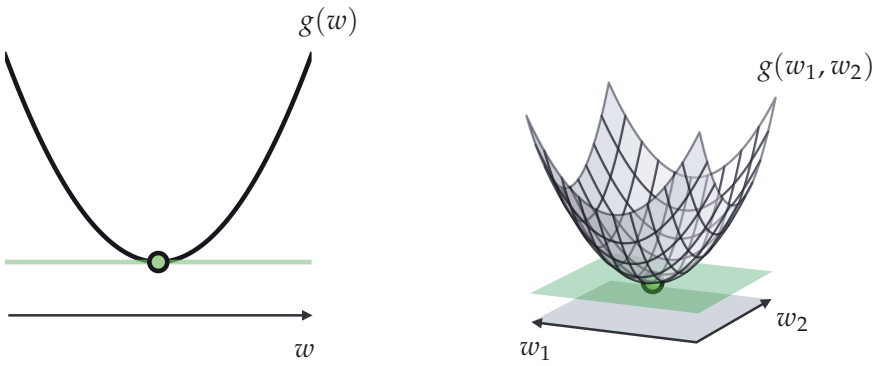

Figure 3.1 The first-order optimality condition characterizes points where the derivative or gradient of a function equals zero or, likewise, where the line or hyperplane tangent to a function has zero slope (i.e., it is completely horizontal and parallel to the input space). With a simple convex quadratic this condition identifies the global minimum of the function, illustrated here with a single-input (left panel) and multi-input (right panel) quadratic function.

$$\frac{\mathrm{d}}{\mathrm{d}w} g\left(v\right) = 0 \tag{3.1}$$

is a potential minimum. Analogously with multi-input functions, any N-dimensional point $\mathbf{v}$ where *every* partial derivative of g is zero, i.e.,

$$\begin{aligned} \tfrac{\partial}{\partial w_1} g(\mathbf{v}) &= 0 \\ \tfrac{\partial}{\partial w_2} g(\mathbf{v}) &= 0 \\ &\vdots \\ \tfrac{\partial}{\partial w_N} g(\mathbf{v}) &= 0 \end{aligned} \tag{3.2}$$

is a potential minimum. This system of N equations is naturally referred to as the *first-order system of equations*. We can also write the first-order system more compactly using gradient notation as

$$\nabla g\left(\mathbf{v}\right) = \mathbf{0}_{N\times 1}. \tag{3.3}$$

This very useful characterization of minimum points is the first-order analog to the zero-order condition for optimality discussed in Section 2.2, and is thus referred to as the *first-order optimality condition* (or the *first-order condition* for short). There are, however, two problems with the first-order characterization of minima.

Firstly, with few exceptions (including some interesting examples we detail in Section 3.2.1), it is virtually impossible to solve a general function's first-order

systems of equations "by hand" (that is, to solve such equations algebraically for closed-form solutions). The other problem is that while the first-order condition defines only global minima for *convex* functions, like the quadratics shown in Figure 3.1, in general this condition captures not only the minima of a function but other points as well, including *maxima* and *saddle points* of *nonconvex* functions as we see in the example below. Collectively, minima, maxima, and saddle points are often referred to as *stationary* or *critical* points of a function.

Example 3.1 **Visual inspection of single-input functions for stationary points**
In the top row of Figure 3.2 we plot the functions

$$\begin{aligned} g(w) &= \sin(2w) \\ g(w) &= w^3 \\ g(w) &= \sin(3w) + 0.3w^2 \end{aligned} \tag{3.4}$$

along with their derivatives in the second row of the same figure. On each function we mark the points where its derivative is zero using a green dot (we likewise mark these points on each derivative itself), and show the tangent line corresponding to each such point in green as well.

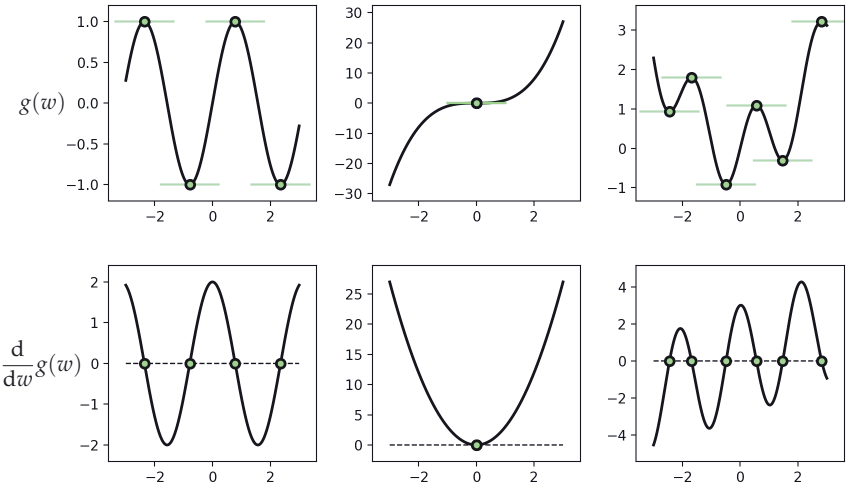

Figure 3.2 Figure associated with Example 3.1. From left to right in the top row, the functions $g(w) = \sin(2w)$, w^3, and $\sin(3w) + 0.3w^2$ are plotted along with their derivatives in the bottom row. See text for further details.

Examining these plots we can see that it is not only *global minima* that have zero derivatives, but a variety of other points as well, including (i) *local minima* or points that are the smallest with respect to their immediate neighbors, e.g., the one around the input value $w = -2.5$ in the top-right panel; (ii) *local* (and

global) maxima or points that are the largest with respect to their immediate neighbors, e.g., the one around the input value $w = 0.5$ in the top-right panel; and (iii) *saddle points*, like the one shown in the top-middle panel, that are neither maximal nor minimal with respect to their immediate neighbors.

3.2.1 Special cases where the first-order system can be solved by hand

In principle, the benefit of using the first-order condition is that it allows us to transform the task of seeking out global minima to that of solving a system of equations, for which a wide range of algorithmic methods have been designed. The emphasis here on the word *algorithmic* is key, as solving a system of (potentially nonlinear) equations *by hand* is, generally speaking, very difficult (if not impossible).

However, there are a handful of relatively simple but important examples where one can compute the solution to a first-order system by hand, or at least one can show algebraically that they reduce to a *linear* system of equations which can be easily solved numerically. By far the most important of these is the multi-input quadratic function (see Example 3.4) and the highly related *Rayleigh quotient* (see Exercise 3.3). These functions arise in many places in the study of machine learning, from fundamental models like linear regression, to second-order algorithm design, to the mathematical analysis of algorithms.

Example 3.2 Finding stationary points of single-input functions via the first-order condition

In this example we use the first-order condition for optimality to compute stationary points of the functions

$$\begin{aligned} g(w) &= w^3 \\ g(w) &= e^w \\ g(w) &= \sin(w) \\ g(w) &= a + bw + cw^2 \;\; (c > 0). \end{aligned} \tag{3.5}$$

- $g(w) = w^3$: the first-order condition gives $\frac{\mathrm{d}}{\mathrm{d}w} g(v) = 3v^2 = 0$, which we can visually identify as a saddle point at $v = 0$ (see top-middle panel of Figure 3.2).
- $g(w) = e^w$: the first-order condition gives $\frac{\mathrm{d}}{\mathrm{d}w} g(v) = e^v = 0$, which is only satisfied as v goes to $-\infty$, giving a minimum.
- $g(w) = \sin(w)$: the first-order condition gives stationary points wherever $\frac{\mathrm{d}}{\mathrm{d}w} g(v) = \cos(v) = 0$, which occurs at odd integer multiples of $\frac{\pi}{2}$, i.e., maxima at $v = \frac{(4k+1)\pi}{2}$ and minima at $v = \frac{(4k+3)\pi}{2}$ where k is any integer.

- $g(w) = a + bw + cw^2$: the first-order condition gives $\frac{\mathrm{d}}{\mathrm{d}w}g(v) = 2cv + b = 0$, with a minimum at $v = \frac{-b}{2c}$ (assuming $c > 0$).

Example 3.3 A simple-looking function

As mentioned previously, the vast majority of first-order systems cannot be solved by hand algebraically. To get a sense of this challenge here we show an example of a simple-enough looking function whose global minimum is not straightforward to compute by hand.

Consider the degree-four polynomial

$$g(w) = \frac{1}{50}\left(w^4 + w^2 + 10w\right) \tag{3.6}$$

which is plotted over a short range of inputs containing its global minimum in Figure 3.3.

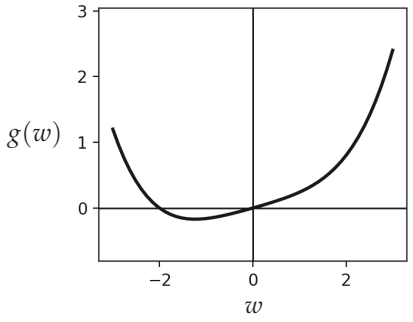

Figure 3.3 Figure associated with Example 3.3. See text for details.

The first-order system here can be easily computed as

$$\frac{\mathrm{d}}{\mathrm{d}w}g(w) = \frac{1}{50}\left(4w^3 + 2w + 10\right) = 0 \tag{3.7}$$

which simplifies to

$$2w^3 + w + 5 = 0. \tag{3.8}$$

This has only one real solution

$$w = \frac{\sqrt[3]{\sqrt{2031} - 45}}{\sqrt[3]{36}} - \frac{1}{\sqrt[3]{6\left(\sqrt{2031} - 45\right)}} \tag{3.9}$$

which can be computed – after much toil – using centuries-old tricks developed for just such problems. In fact, had the polynomial in Equation (3.6) been of degree six or higher, we would not have been able to guarantee finding its stationary point(s) in closed form.

Example 3.4 Stationary points of multi-input quadratic functions
Take the general multi-input quadratic function

$$g(\mathbf{w}) = a + \mathbf{b}^T\mathbf{w} + \mathbf{w}^T\mathbf{C}\mathbf{w} \tag{3.10}$$

where $\mathbf{C}$ is an $N \times N$ symmetric matrix, $\mathbf{b}$ is an $N \times 1$ vector, and a is a scalar. Computing the gradient of g we have

$$\nabla g(\mathbf{w}) = 2\mathbf{C}\mathbf{w} + \mathbf{b}. \tag{3.11}$$

Setting this equal to zero gives a symmetric linear system of equations of the form

$$\mathbf{C}\mathbf{w} = -\frac{1}{2}\mathbf{b} \tag{3.12}$$

whose solutions are stationary points of the original function. Note here we have not explicitly solved for these stationary points, but have merely shown that the first-order system of equations in this particular case is in fact one of the easiest to solve numerically (see Example 3.6).

3.2.2 Coordinate descent and the first-order optimality condition

While solving the first-order system in Equation (3.2) *simultaneously* is often impossible, it is sometimes possible to solve such a system *sequentially*. In other words, in some (rather important) cases the first order system can be solved *one equation at a time*, the nth of which takes the form $\frac{\partial}{\partial w_n} g(\mathbf{v}) = 0$. This idea, which is a form of *coordinate descent*, is especially effective when each of these equations can be solved for in closed form (e.g., when the function being minimized is a quadratic).

To solve the first-order system sequentially, we first initialize at an input $\mathbf{w}^0$, and begin by updating the first coordinate by solving

$$\frac{\partial}{\partial w_1} g\left(\mathbf{w}^0\right) = 0 \tag{3.13}$$

for the optimal first weight $w_1^\star$. Note importantly in solving this equation for w_1 that all other weights are kept fixed at their initial values. We then update the first coordinate of the vector $\mathbf{w}^0$ with this solution $w_1^\star$, and call the updated set of weights $\mathbf{w}^1$. Continuing this pattern, to update the nth weight we solve

$$\frac{\partial}{\partial w_n} g\left(\mathbf{w}^{n-1}\right) = 0 \tag{3.14}$$

for $w_n^\star$. Again, when this equation is solved all other weights are kept fixed at

their current values. We then update the nth weight using this value forming the updated set of weights $\mathbf{w}^n$.

After we sweep through all N weights a single time we can refine our solution by sweeping through the weights again (as with any other coordinate-wise method). At the kth such sweep we update the nth weight by solving the single equation

$$\frac{\partial}{\partial w_n} g\left(\mathbf{w}^{N(k-1)+n-1}\right) = 0 \tag{3.15}$$

to update the nth weight of $\mathbf{w}^{N(k-1)+n-1}$, and so on.

Example 3.5 Minimizing convex quadratic functions via coordinate descent

In this example we use coordinate descent to minimize the convex quadratic function

$$g(w_1, w_2) = w_1^2 + w_2^2 + 2 \tag{3.16}$$

whose minimum lies at the origin. We make the run initialized at $\mathbf{w}^0 = [3\ \ 4]^T$, where a single sweep through the coordinates (i.e., two steps) here perfectly minimizes the function. The path this run took is illustrated in the left panel of Figure 3.4 along with a contour plot of the function. One can easily check that each first-order equation in this case is linear and trivial to solve in closed form.

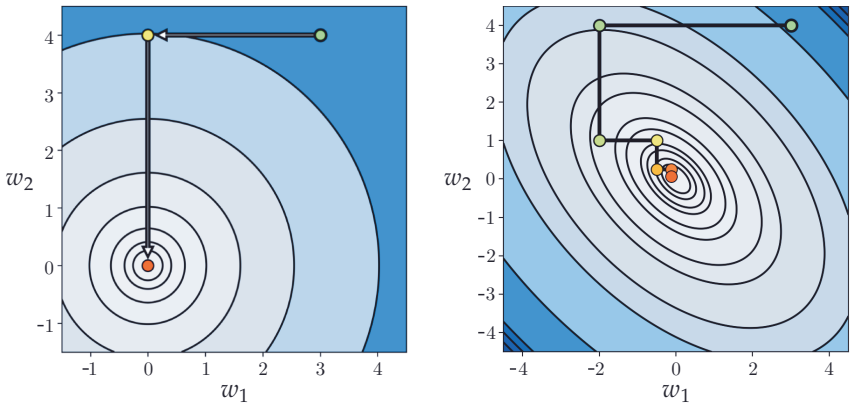

Figure 3.4 Figure associated with Example 3.5. See text for details.

We then apply coordinate descent to minimize the convex quadratic

$$g(w_1, w_2) = 2w_1^2 + 2w_2^2 + 2w_1w_2 + 20 \tag{3.17}$$

whose contour plot is shown in the right panel of Figure 3.4. Here it takes infinitely many sweeps through the variables to find the global minimum precisely and at least three full sweeps to get reasonably close to it, which again lies at the origin.

Example 3.6 Solving symmetric linear systems of equations
In Example 3.4 we saw that the first-order system of a multi-input quadratic function takes the form

$$\mathbf{C}\mathbf{w} = -\frac{1}{2}\mathbf{b} \tag{3.18}$$

where $\mathbf{C}$ is a symmetric matrix as described in that example. We can use the coordinate descent algorithm to solve this system, thereby minimizing the corresponding quadratic function. Divorced from the concept of a quadratic we can think of coordinate descent in a broader context as a method for solving more general symmetric linear systems of equations, which is quite commonly encountered in practice, e.g., at each and every step of Newton's method (as detailed in Chapter 4).

3.3 The Geometry of First-Order Taylor Series

In this section we describe important characteristics of the *hyperplane* including the concept of the direction of *steepest ascent* and *steepest descent*. We then study a special hyperplane: the first-order Taylor series approximation to a function, which defines the very essence of the extremely popular gradient descent algorithm, introduced in Section 3.5.

3.3.1 The anatomy of hyperplanes

A general N-dimensional hyperplane can be characterized as

$$h(w_1, w_2, \ldots, w_N) = a + b_1 w_1 + b_2 w_2 + \cdots + b_N w_N \tag{3.19}$$

where a as well as b_1 through b_N are all scalar parameters. We can rewrite h more compactly – using vector notation – as

$$h(\mathbf{w}) = a + \mathbf{b}^T\mathbf{w} \tag{3.20}$$

denoting

$$\mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_N \end{bmatrix} \quad \text{and} \quad \mathbf{w} = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_N \end{bmatrix}. \tag{3.21}$$

When $N = 1$, Equation (3.20) simplifies to

$$h(w) = a + bw \tag{3.22}$$

which is the familiar formula for a (one-dimensional) line. Notice, $h(w) = a + bw$ is a *one*-dimensional thing living in a *two*-dimensional ambient space whose input space (characterized by w) is *one*-dimensional itself.

The same is true for general N. That is, $h(\mathbf{w}) = a + \mathbf{b}^T\mathbf{w}$ is an N-dimensional mathematical object living in an $(N+1)$-dimensional ambient space whose input space (characterized by $w_1, w_2, \ldots, w_N$) is N-dimensional.

3.3.2 Steepest ascent and descent directions

As we just saw, with a one-dimensional hyperplane the input space is also one-dimensional, implying that at any point w^0 in the input space there are only two directions to move in: to the left or right of w^0. This is illustrated in the left panel of Figure 3.5. Here, starting at w^0 and moving to the right (towards $+\infty$) *increases* the value of h, and hence it is an *ascent* direction. Conversely, moving to the left (towards $-\infty$) *decreases* the value of h, and hence it is a *descent* direction.

When $N > 1$, however, there are infinitely many directions to move in (as opposed to just two when $N = 1$) – some providing ascent, some providing descent, and some that preserve the value of h – as illustrated in the right panel of Figure 3.5 for $N = 2$. It is therefore logical to ask whether we can find the direction that produces the largest ascent (or descent), commonly referred to as the direction of *steepest ascent* (or *descent*).

Formalizing the search for the direction of steepest ascent at a given point $\mathbf{w}^0$, we aim to find the unit direction $\mathbf{d}$ such that the value of $h\left(\mathbf{w}^0 + \mathbf{d}\right)$ is maximal. In other words, we aim to solve

$$\underset{\mathbf{d}}{\text{maximize}} \;\; h\left(\mathbf{w}^0 + \mathbf{d}\right) \tag{3.23}$$

over all unit-length vectors $\mathbf{d}$. Note from Equation (3.20) that $h\left(\mathbf{w}^0 + \mathbf{d}\right)$ can be written as

$$a + \mathbf{b}^T\left(\mathbf{w}^0 + \mathbf{d}\right) = a + \mathbf{b}^T\mathbf{w}^0 + \mathbf{b}^T\mathbf{d} \tag{3.24}$$

where the first two terms on the right-hand side are constant with respect to $\mathbf{d}$.

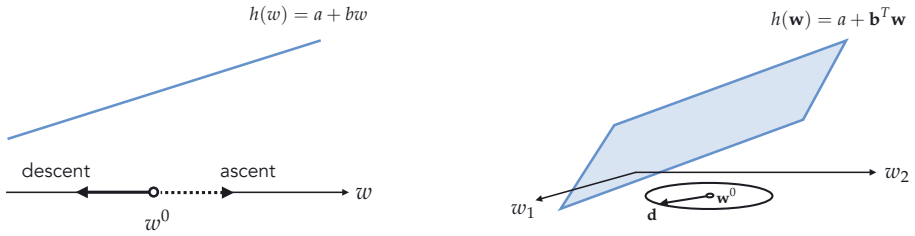

Figure 3.5 (left panel) At any given point w^0 in the input space of a one-dimensional hyperplane h, there are only two directions to travel in: one that increases the evaluation of h (or an *ascent* direction), and one that decreases it (or a *descent* direction). (right panel) In higher dimensions (here $N = 2$) there are infinitely many (unit) directions $\mathbf{d}$ to move into – starting at a given N-dimensional input point $\mathbf{w}^0$. As can be seen in this case, the endpoint of all such directions form a unit circle around and centered at $\mathbf{w}^0$.

Therefore maximizing the value of $h\left(\mathbf{w}^0 + \mathbf{d}\right)$ is equivalent to maximizing $\mathbf{b}^T\mathbf{d}$, which itself can be written, using the inner-product rule (see Appendix C), as

$$\mathbf{b}^T\mathbf{d} = \|\mathbf{b}\|_2 \|\mathbf{d}\|_2 \cos(\theta). \tag{3.25}$$

Note, once again, that $\|\mathbf{b}\|_2$ (i.e., the length of $\mathbf{b}$) does not change with respect to $\mathbf{d}$, and that $\|\mathbf{d}\|_2 = 1$. Therefore the problem in Equation (3.23) reduces to

$$\underset{\theta}{\text{maximize}} \quad \cos(\theta) \tag{3.26}$$

where θ is the angle between the vectors $\mathbf{b}$ and $\mathbf{d}$.

It is clear now, of all unit directions, $\mathbf{d} = \frac{\mathbf{b}}{\|\mathbf{b}\|_2}$ provides the steepest ascent (where $\theta = 0$ and $\cos(\theta) = 1$). Similarly, we can show that the unit direction $\mathbf{d} = \frac{-\mathbf{b}}{\|\mathbf{b}\|_2}$ provides the steepest descent (where $\theta = \pi$ and $\cos(\theta) = -1$).

3.3.3 The gradient and the direction of steepest ascent/descent

A multi-input function $g(\mathbf{w})$ can be approximated locally around a given point $\mathbf{w}^0$ by a hyperplane $h(\mathbf{w})$

$$h(\mathbf{w}) = g(\mathbf{w}^0) + \nabla g(\mathbf{w}^0)^T \left(\mathbf{w} - \mathbf{w}^0\right) \tag{3.27}$$

which can be rewritten as $h(\mathbf{w}) = a + \mathbf{b}^T\mathbf{w}$ (to match our notation in the previous section) where

$$a = g(\mathbf{w}^0) - \nabla g(\mathbf{w}^0)^T\mathbf{w}^0 \quad \text{and} \quad \mathbf{b} = \nabla g(\mathbf{w}^0). \tag{3.28}$$

This hyperplane is the *first-order Taylor series approximation* of g at $\mathbf{w}^0$, and is tangent to g at this point (see Appendix B).

Because h is constructed to closely approximate g near the point $\mathbf{w}^0$, its steepest

ascent and descent directions also tell us the direction to travel to increase or decrease the value of the underlying function g itself at/near the point $\mathbf{w}^0$.

3.4 Computing Gradients Efficiently

Think for a moment about how you perform basic arithmetic, e.g., multiplying two numbers. If the two numbers involved are relatively small, such as 35 times 21, you can likely do the multiplication in your head using a combination of *multiplication properties* and *simple multiplication results* you learned in elementary school. For example, you may choose to use the distributive property of multiplication to decompose 35×21 as

$$(30 + 5) \times (20 + 1) = (30 \times 20) + (30 \times 1) + (5 \times 20) + (5 \times 1) \tag{3.29}$$

and then use the multiplication table that you have likely memorized to find 30×20, 30×1, and so on. We use this sort of strategy on a daily basis when making quick back-of-the-envelope calculations like computing interest on a loan or investment, computing how much to tip at a restaurant, etc.

However, even though the rules for multiplication are quite simple and work regardless of the two numbers being multiplied together, you would likely never compute the product of two arbitrarily large numbers, like 140283197 times 2241792341, using the same strategy. Instead you would likely use a *calculator* because it conveniently automates the process of multiplying two numbers of arbitrary size. A calculator allows you to compute with much greater efficiency and accuracy, and empowers you to use the fruits of arithmetic computation for more important tasks.

This is precisely how you can think about the computation of derivatives and gradients. Perhaps you can compute the derivative of a relatively simple mathematical function like $g(w) = \sin(w^2)$ easily, knowing a combination of *differentiation rules* as well as derivatives of certain *elementary functions* such as sinusoids and polynomials (see Appendix B for a review). In this particular case you can use the *chain rule* to write $\frac{\mathrm{d}}{\mathrm{d}w}g(w)$ as

$$\frac{\mathrm{d}}{\mathrm{d}w}g(w) = \left(\frac{\mathrm{d}}{\mathrm{d}w}w^2\right)\cos(w^2) = 2w\cos(w^2). \tag{3.30}$$

As with multiplication, even though the rules for differentiation are quite simple and work regardless of the function being differentiated, you would likely never compute the gradient of an arbitrarily complicated function, such as the one that follows, yourself and *by hand*

$$g(w_1, w_2) = 2^{\sin\left(w_1^2 + w_2^2\right)}\tanh\left(\cos\left(w_1 w_2\right)\right)\tanh\left(w_1 w_2^4 + \tanh\left(w_1 + w_2^2\right)\right) \tag{3.31}$$

as it is extremely time consuming and easy to mess up (just like multiplication

of two large numbers). Following the same logic a *gradient calculator* would allow for computing derivatives and gradients with much greater efficiency and accuracy, empowering us to use the fruits of gradient computation for more important tasks, e.g., for the popular first-order *local optimization* schemes that are the subject of this chapter and that are widely used in machine learning. Therefore throughout the remainder of the text the reader should feel comfortable using a *gradient calculator* as an alternative to hand computation.

Gradient calculators come in several varieties, from those that provide numerical approximations to those that literally automate the simple derivative rules for elementary functions and operations. We outline these various approaches in Appendix B. For `Python` users we strongly recommend using the open-source automatic differentiation library called `autograd` [10, 11, 12, 13] or `JAX` (an extension of `autograd` that runs on GPUs and TPUs). This is a high-quality and easy-to-use professional-grade gradient calculator that gives the user the power to easily compute the gradient for a wide variety of `Python` functions built using standard data structures and `autograd` operations. In Section B.10 we provide a brief tutorial on how to get started with `autograd`, as well as demonstrations of some of its core functionality.

3.5 Gradient Descent

In Section 3.3 we saw how the negative gradient $-\nabla g(\mathbf{w})$ of a function $g(\mathbf{w})$ computed at a particular point *always* defines a valid descent direction at that point. We could very naturally wonder about the efficacy of a local optimization method, that is one consisting of steps of the general form $\mathbf{w}^k = \mathbf{w}^{k-1} + \alpha \mathbf{d}^k$ (see Section 2.4), employing the negative gradient direction $\mathbf{d}^k = -\nabla g\left(\mathbf{w}^{k-1}\right)$. Such a sequence of steps would then take the form

$$\mathbf{w}^k = \mathbf{w}^{k-1} - \alpha \nabla g\left(\mathbf{w}^{k-1}\right). \tag{3.32}$$

It seems intuitive, at least at the outset that, because each and every direction is guaranteed to be one of descent (provided we set α appropriately, as we must always do when using any local optimization method), taking such steps could lead us to a point near a local minimum of the target function g. The rather simple update step in Equation (3.32) is indeed an extremely popular local optimization method called the *gradient descent algorithm*, named so because it employs the (negative) *gradient* as the *descent* direction.

A prototypical path taken by gradient descent is illustrated in Figure 3.6 for a generic single-input function. At each step of this local optimization method we can think about drawing the first-order Taylor series approximation to the function, and taking the descent direction of this tangent hyperplane (i.e., the negative gradient of the function at this point) as our descent direction for the algorithm.

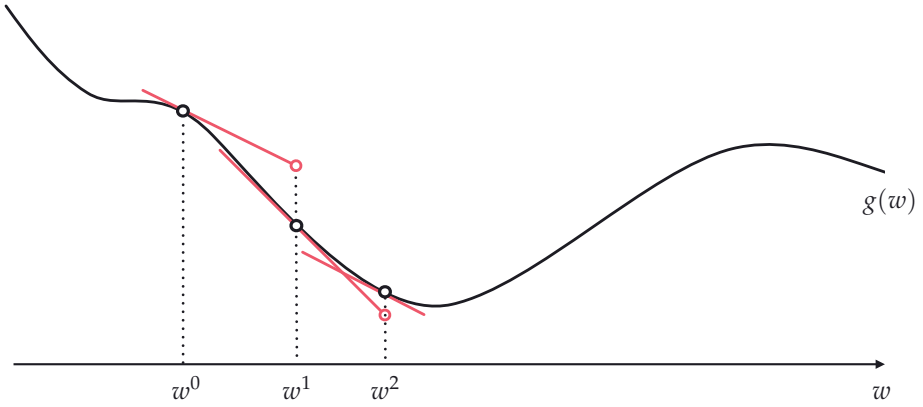

Figure 3.6 A figurative drawing of the gradient descent algorithm. Beginning at the initial point w^0 we make our first approximation to $g(w)$ at the point $(w^0, g(w^0))$ on the function (shown as a hollow black dot) with the first-order Taylor series approximation itself drawn in red. Moving in the negative gradient descent direction provided by this approximation we arrive at a point $w^1 = w^0 - \alpha \frac{\mathrm{d}}{\mathrm{d}w} g(w^0)$. We then repeat this process at w^1, moving in the negative gradient direction there, to $w^2 = w^1 - \alpha \frac{\mathrm{d}}{\mathrm{d}w} g(w^1)$, and so forth.

As we will see in this and many of our future chapters, the gradient descent algorithm is often a far better local optimization algorithm than the zero-order approaches discussed in the previous chapter. Indeed gradient descent, along with its extensions which we detail in Appendix A, is arguably the most popular local optimization algorithm used in machine learning today. This is largely because of the fact that the descent direction provided here (via the gradient) is almost always easier to compute (particularly as the dimension of the input increases) than seeking out a descent direction at random (as is done with the zero-order methods described in Sections 2.4 through 2.6). In other words, the fact that the negative gradient direction provides a descent direction for the function locally, combined with the fact that gradients are easy to compute (particularly when employing an automatic differentiator) makes gradient descent a superb local optimization method.

Example 3.7 Minimizing a nonconvex function using gradient descent

To find the global minimum of a general nonconvex function using gradient descent (or any local optimization method) one may need to run it several times with different initializations and/or steplength schemes. We showcase this fact using the nonconvex function

$$g(w) = \sin(3w) + 0.3w^2 \tag{3.33}$$

illustrated in the top panels of Figure 3.7. The same function was minimized

in Example 2.4 using random search. Here we initialize two runs of gradient descent, one at $w^0 = 4.5$ (top-left panel) and another at $w^0 = -1.5$ (top-right panel), using a fixed steplength of $\alpha = 0.05$ for both runs. As can be seen by the results, depending on where we initialize we may end up near a local or global minimum of the function (we color the steps from green to red as the method proceeds from start to finish).

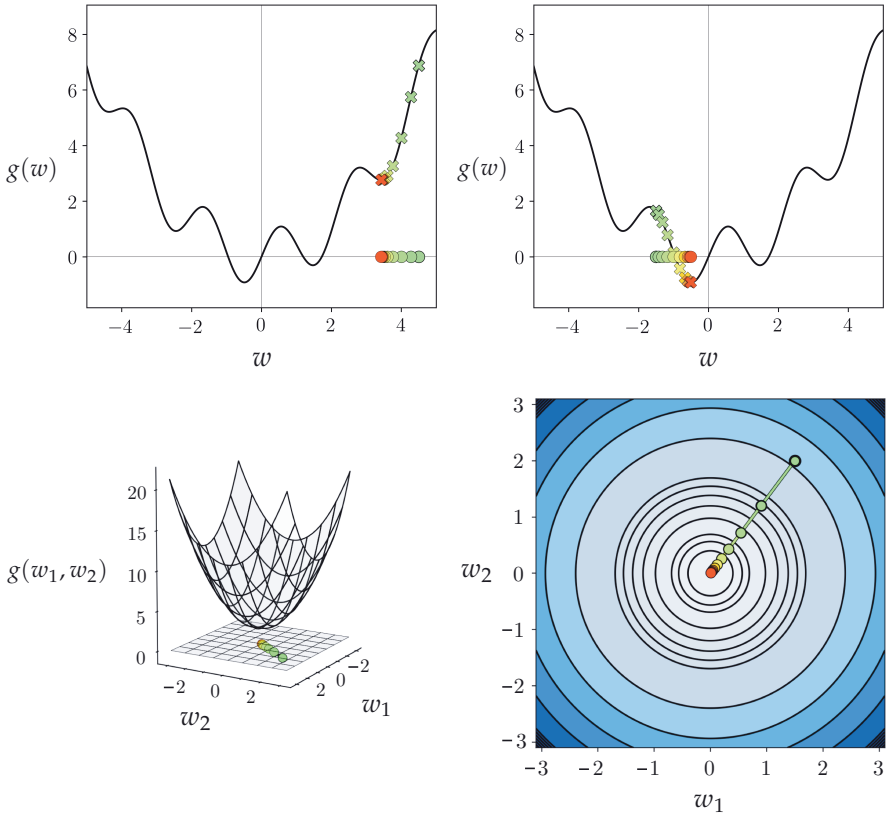

Figure 3.7 (top panels) Figure associated with Example 3.7. (bottom panels) Figure associated with Example 3.8. See text for details.

Example 3.8 Minimizing a convex multi-input function using gradient descent

In this example we run gradient descent on the convex multi-input quadratic function

$$g(w_1, w_2) = w_1^2 + w_2^2 + 2 \tag{3.34}$$

previously used in Example 2.3. We fix the steplength parameter at $\alpha = 0.1$ for all

ten steps of the algorithm. In the bottom row of Figure 3.7 we illustrate the path taken by this run of gradient descent in the input space of the function, again coloring the steps from green to red as the method finishes. This is shown along with the three-dimensional surface of the function in the bottom-left panel, and "from above," showing the contours of the function on its input space in the bottom-right panel.

3.5.1 Basic steplength choices for gradient descent

As with all local methods, one needs to carefully choose the steplength or learning rate parameter α with gradient descent. While there are an array of available sophisticated methods for choosing α in the case of gradient descent, the most common choices employed in machine learning are those basic approaches first detailed in the simple context of zero-order methods in Section 2.5. These common choices include (i) using a fixed α value for each step of a gradient descent run, which for simplicity's sake commonly takes the form of 10^{γ}, where γ is an (often negative) integer, and (ii) using a diminishing steplength like $\alpha = \frac{1}{k}$ at the kth step of a run.

In both instances our aim in choosing a particular value for the steplength α at each step of gradient descent mirrors that of any other local optimization method: α should be chosen to induce the most rapid minimization possible. With the fixed steplength this often means choosing the *largest* possible value for α that leads to proper convergence.

Example 3.9 A fixed steplength selection for gradient descent

At each step of gradient descent we *always* have a descent direction – this is defined explicitly by the negative gradient itself. However, whether or not we actually descend in the function when taking a gradient descent step depends completely on how far we travel in the direction of the negative gradient, which we control via our steplength parameter. Set incorrectly, we can descend infinitely slowly, or even *ascend* in the function.

We illustrate this in Figure 3.8 using the simple single-input quadratic $g(w) = w^2$. Here we show the result of taking five gradient descent steps using three different fixed steplength values, all initialized at the same point $w^0 = -2.5$. The top row of this figure shows the function itself along with the evaluation at each step of a run (the value of α used in each run is shown at the top of each panel in this row), which are colored from green at the start of the run to red when the last step of the run is taken. From left to right each panel shows a different run with a slightly increased fixed steplength value α used for all steps. In the left panel the steplength is extremely small – so small that we do not descend very much at all. In the right panel, however, when we set the value of α too large the algorithm ascends in the function (ultimately *diverging*).

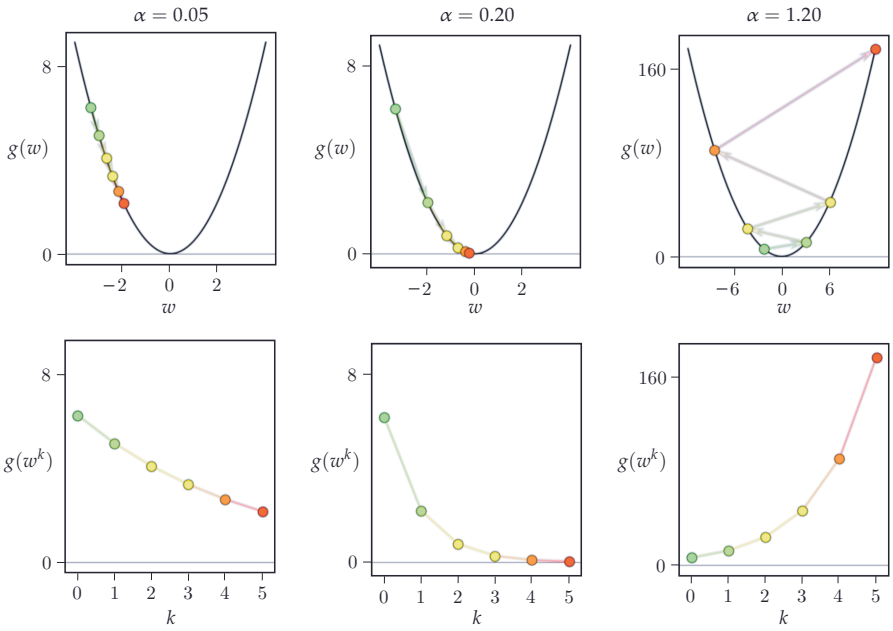

Figure 3.8 Figure associated with Example 3.9. See text for details.

In the bottom row we show the cost function history plot corresponding to each run of gradient descent in the top row of the figure. We also color these points from green (start of run) to red (end of run).

Example 3.10 Comparing fixed and diminishing steplengths

In Figure 3.9 we illustrate the comparison of a fixed steplength scheme and a diminishing one to minimize the function

$$g(w) = |w| \,. \tag{3.35}$$

Notice that this function has a single global minimum at $w = 0$. We make two runs of 20 steps of gradient descent, each initialized at the point $w^0 = 1.75$, the first with a fixed steplength rule of $\alpha = 0.5$ (shown in the top-left panel) for each and every step, and the second using the diminishing steplength rule $\alpha = \frac{1}{k}$ (shown in the top-right panel).

Here we can see that the fixed steplength run gets stuck, unable to descend towards the minimum of the function, while the diminishing steplength run settles down nicely to the global minimum (which is also reflected in the cost function history plot shown in the bottom panel of the figure). This is because the derivative of this function (defined everywhere but at $w = 0$) takes the form

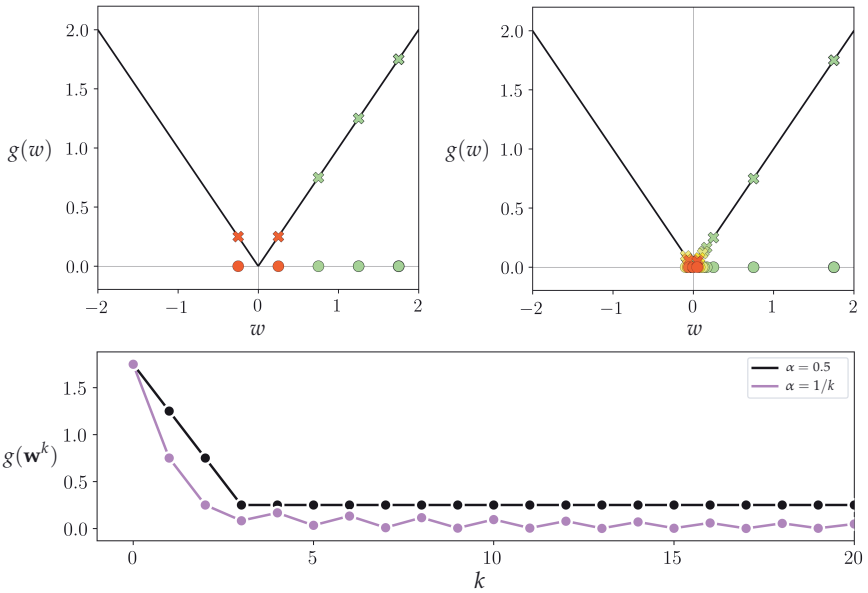

Figure 3.9 Figure associated with Example 3.10. See text for details.

$$\frac{\mathrm{d}}{\mathrm{d}w}g(w) = \begin{cases} +1 & \text{if } w > 0 \\ -1 & \text{if } w < 0 \end{cases} \tag{3.36}$$

which makes the use of any fixed steplength scheme problematic since the algorithm will always move at a fixed distance at each step. We face this potential issue with all local optimization methods (indeed we first saw it occur with a zero-order method in Example 2.5).

3.5.2 Oscillation in the cost function history plot: not always a bad thing

Remember that in practice – since we regularly deal with cost functions that take in far too many inputs – we use the *cost function history plot* (first described in Section 2.5.2) to tune our steplength parameter α, as well as debug implementations of the algorithm.

Note that when employing the cost function history plot in choosing a proper steplength value, it is not ultimately important that the plot associated to a run of gradient descent (or any local optimization method) be *strictly decreasing* (that is showing that the algorithm *descended* at every single step). It is critical to find a value of α that allows gradient descent to find the lowest function value possible, even if it means that not every step *descends*. In other words, the *best* choice of α for a given minimization might cause gradient descent to "hop around" some,

moving up and down, and not the one that shows descent in each and every step. Below we show an example illustrating this point.

Example 3.11 Oscillatory versus monotonically decreasing cost function history plots

In this example we show the result of three runs of gradient descent to minimize the function

$$g(\mathbf{w}) = w_1^2 + w_2^2 + 2\sin\left(1.5\left(w_1 + w_2\right)\right)^2 + 2 \tag{3.37}$$

whose contour plot is shown in Figure 3.10. You can see a *local minimum* around the point $[1.5 \;\; 1.5]^T$, and a global minimum near $[-0.5 \;\; -0.5]^T$. All three runs start at the same initial point $\mathbf{w}^0 = [3 \;\; 3]^T$ and take ten steps. The first run (shown in the top-left panel) uses a fixed steplength parameter $\alpha = 10^{-2}$, the second run (shown in the top-middle panel) uses $\alpha = 10^{-1}$, and the third run (shown in the top-right panel) uses $\alpha = 10^0$.

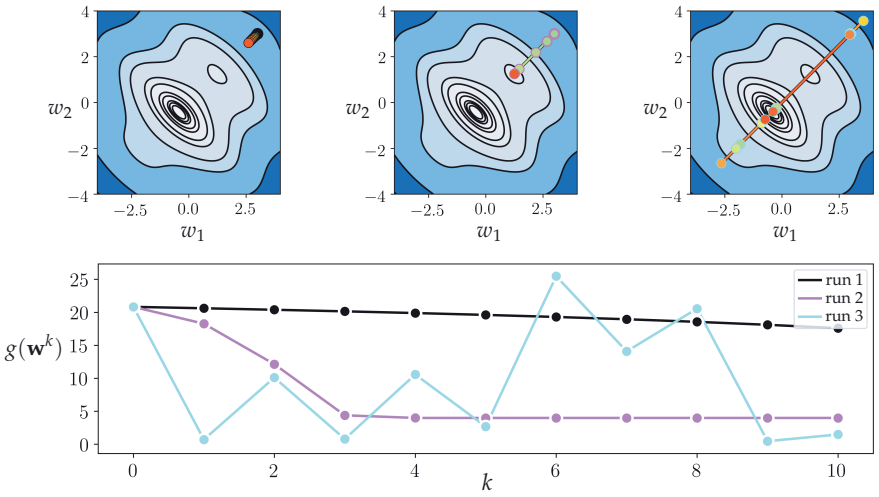

Figure 3.10 Figure associated with Example 3.11. See text for details.

In the bottom panel of the figure we plot the cost function history plot associated with each run of gradient descent, showing the first, second, and third run in black, pink, and blue, respectively. Here we can see that the value of our first choice was too small (as can also be seen in the top-left panel), the second choice leads to convergence to the *local* minimum (as can be seen in the top-middle panel), and final run while "hopping around" and not strictly decreasing at each step finds the lowest point out of all three runs. So while this run used the largest steplength $\alpha = 10^0$, clearly leading to oscillatory (and perhaps eventually

divergent) behavior, it does indeed find the lowest point out of all three runs performed.

This function is rather pathological, i.e., it was designed specifically for the purposes of this example. However, in practice the moral expressed here, that it is just fine for the cost the function history associated with a run of gradient descent (or any local optimization algorithm more generally) to oscillate up and down (and not be perfectly smooth and monotonically decreasing), holds in general.

3.5.3 Convergence criteria

When does gradient descent stop? Technically, if the steplength is chosen wisely, the algorithm will halt near stationary points of a function, typically minima or saddle points. How do we know this? By the very form of the gradient descent step itself. If the step $\mathbf{w}^k = \mathbf{w}^{k-1} - \alpha \nabla g\left(\mathbf{w}^{k-1}\right)$ does not move from the prior point $\mathbf{w}^{k-1}$ significantly then this can mean only one thing: that the direction we are traveling in is *vanishing*, i.e., $-\nabla g\left(\mathbf{w}^{k-1}\right) \approx \mathbf{0}_{N\times 1}$. This is – by definition – a *stationary point* of the function (as detailed in Section 3.2).

In principle, then, we can wait for gradient descent to get sufficiently close to a stationary point by ensuring, for instance, that the magnitude of the gradient $\left\|\nabla g\left(\mathbf{w}^{k-1}\right)\right\|_2$ is sufficiently small. Other formal convergence criteria include (i) halting when steps no longer make sufficient progress (e.g., when $\frac{1}{N}\left\|\mathbf{w}^k - \mathbf{w}^{k-1}\right\|_2$ is smaller than some ϵ) or, (ii) when corresponding evaluations no longer differ substantially (e.g., when $\frac{1}{N}\left|g\left(\mathbf{w}^k\right) - g\left(\mathbf{w}^{k-1}\right)\right|$ is smaller than some ϵ). Finally, a practical way to halt gradient descent – as well as any other local optimization scheme – is to simply run the algorithm for a fixed number of maximum iterations. In machine learning applications this latter practical condition is very often employed, either alone or in combination with a formal stopping procedure.

How do we set the maximum iteration count? As with any local method this is typically set manually/heuristically, and is influenced by things like computing resources, knowledge of the particular function being minimized, and – very importantly – the choice of the steplength parameter α. Smaller choices for α, while more easily providing descent at each step, frequently require more steps for the algorithm to achieve significant progress. Conversely, if α is set too large gradient descent may bounce around erratically forever, never localizing in an adequate solution.

3.5.4 `Python` implementation

In this section we provide a basic `Python` implementation of the gradient descent algorithm. There are a number of variations one can use in practice including

various halting conditions (described above), as well as various to compute the gradient itself. The inputs here include the function to minimize `g`, a steplength `alpha`, a maximum number of iterations `max_its` (our default stopping condition), and an initial point `w` that is typically chosen at random. The outputs include a history of weight updates and corresponding cost function value history (which one can use to produce a cost function history plot). The computation of the gradient function in line 16 employs by default the open-source automatic differentiation library `autograd` (detailed in Sections 3.4 and B.10) – although one can easily replace this with any other method for computing the gradient function.

```
1  # import automatic differentiator to compute gradient module
2  from autograd import grad
3
4  # gradient descent function
5  def gradient_descent(g, alpha, max_its, w):
6
7      # compute gradient module using autograd
8      gradient = grad(g)
9
10     # gradient descent loop
11     weight_history = [w]  # weight history container
12     cost_history = [g(w)] # cost function history container
13     for k in range(max_its):
14
15         # evaluate the gradient
16         grad_eval = gradient(w)
17
18         # take gradient descent step
19         w = w - alpha*grad_eval
20
21         # record weight and cost
22         weight_history.append(w)
23         cost_history.append(g(w))
24
25     return weight_history, cost_history
```

Given the input to `g` is N-dimensional, a general random initialization can be written as shown below where the `NumPy` function `random.randn` produces samples from a standard normal distribution with zero mean and unit standard deviation. It is also common to scale such initializations by small constants (e.g., 0.1 or smaller).

```
1  # a common random initialization scheme
2  import numpy as np
3  scale = 0.1
4  w = scale*np.random.randn(N,1)
```

3.6 Two Natural Weaknesses of Gradient Descent

As we saw in the previous section, gradient descent is a local optimization scheme that employs the negative gradient at each step. The fact that calculus provides us with a true descent direction in the form of the negative gradient direction, combined with the fact that gradients are often cheap to compute (whether or not one uses an automatic differentiator), means that we need not search for a reasonable descent direction at each step of the method as we needed to do with the zero-order methods detailed in the previous chapter. This is extremely advantageous, and is the fundamental reason why gradient descent is so popular in machine learning.

However, no basic local optimization scheme is without its shortcomings. In the previous chapter we saw how, for example, the natural shortcoming of random search limits its practical usage to functions of low-dimensional input. While gradient descent does not suffer from this particular limitation, the negative gradient has its own weaknesses as a descent direction, which we outline in this section.

3.6.1 Where do the weaknesses of the (negative) gradient direction originate?

Where do these weaknesses originate? Like any vector, the negative gradient always consists fundamentally of a *direction* and a *magnitude* (as illustrated in Figure 3.11). Depending on the function being minimized either one of these attributes – or both – can present challenges when using the negative gradient as a descent direction.

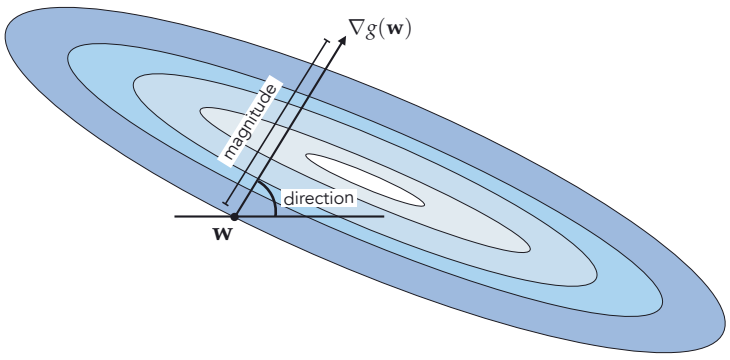

Figure 3.11 The gradient vector of any arbitrary function at any point consists of a *magnitude* and a *direction*.

The *direction* of the negative gradient can rapidly oscillate during a run of gradient descent, often producing *zig-zagging* steps that take considerable time

to reach a minimum point. The *magnitude* of the negative gradient can vanish rapidly near stationary points, leading gradient descent to *slowly crawl* near minima and saddle points. This too can slow down gradient descent's progress near stationary points. These two problems, while not present when minimizing every single function, do present themselves in machine learning because many of the functions we aim to minimize have *long narrow valleys* – long flat areas where the contours of a function become increasingly parallel. Both of these issues – stemming from either the direction or magnitude of the negative gradient direction – are explored further below.

3.6.2 The (negative) gradient direction

A fundamental property of the (negative) gradient direction is that it always points perpendicular to the contours of a function. This statement is universally true, and holds for *any* (differentiable) function and at *all* of its inputs. That is, the gradient ascent/descent direction at an input $\mathbf{w}^0$ is always perpendicular to the contour $g(\mathbf{w}) = g(\mathbf{w}^0)$.

Example 3.12 The negative gradient direction

In Figure 3.12 we show the contour plot of (top-left panel) $g(\mathbf{w}) = w_1^2 + w_2^2 + 2$, (top-right panel) $g(\mathbf{w}) = w_1^2 + w_2^2 + 2\sin\left(1.5\left(w_1 + w_2\right)\right)^2 + 2$, and (bottom panel) $g(\mathbf{w}) = \left(w_1^2 + w_2 - 11\right)^2 + \left(w_1 + w_2^2 - 6\right)^2$.

On each plot we also show the negative gradient direction defined at three random points. Each of the points we choose are highlighted in a unique color, with the contour on which they sit on the function colored in the same manner for visualization purposes. The *descent* direction defined by the gradient at each point is drawn as an arrow, and the tangent line to the contour at each input is also drawn (in both instances colored the same as their respective point).

In each instance we can see how the gradient descent direction is always *perpendicular* to the contour it lies on – in particular being perpendicular to the tangent line at each point on the contour (which is also shown). Because the gradient ascent directions will simply point in the opposite direction as the descent directions shown here, they too will be perpendicular to the contours.

3.6.3 The zig-zagging behavior of gradient descent

In practice, the fact that the negative gradient always points in a direction perpendicular to the contour of a function can – depending on the function being minimized – make the negative gradient direction oscillate rapidly or *zig-zag* during a run of gradient descent. This in turn can cause zig-zagging behavior in the gradient descent steps themselves and *too much* zig-zagging

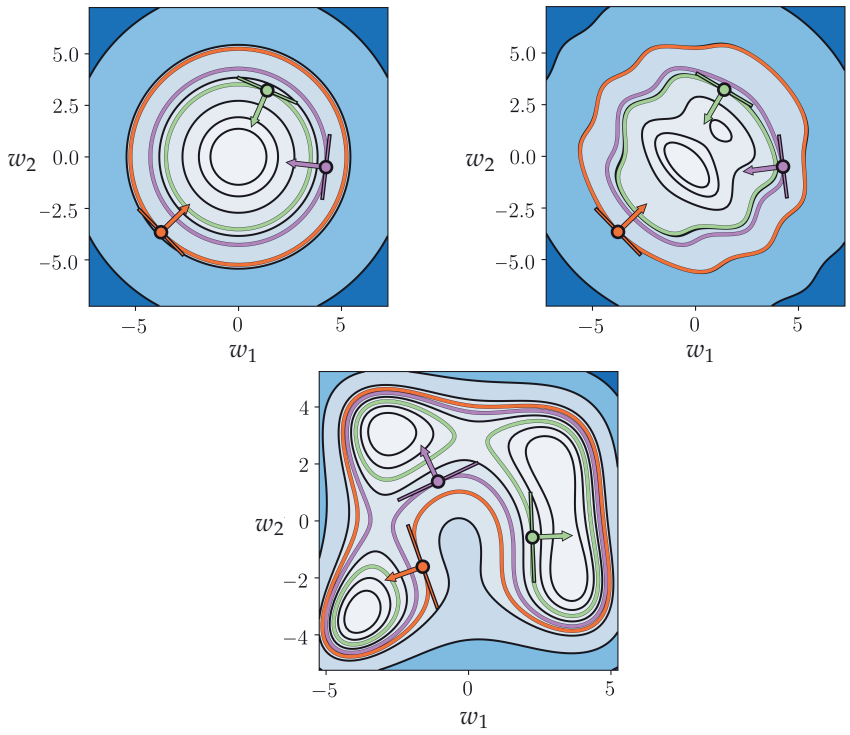

Figure 3.12 Figure associated with Example 3.12. Regardless of the function the negative gradient direction is always *perpendicular* to the function's contours. See text for further details.

slows minimization progress. When it occurs, many gradient descent steps are required to adequately minimize a function. We illustrate this phenomenon below using a set of simple examples.

The interested reader may note that we describe a popular solution to this zig-zagging behavior, called *momentum-accelerated* gradient descent, later in Appendix Section A.2.

Example 3.13 The zig-zagging behavior of gradient descent

In Figure 3.13 we illustrate the zig-zagging behavior of gradient descent with three $N = 2$ dimensional quadratic functions that take the general form $g(\mathbf{w}) = a + \mathbf{b}^T\mathbf{w} + \mathbf{w}^T\mathbf{C}\mathbf{w}$. In each case a and $\mathbf{b}$ are set to zero, and the matrix $\mathbf{C}$ is set so that each quadratic gets progressively narrower:

$$\mathbf{C} = \begin{bmatrix} 0.50 & 0 \\ 0 & 12 \end{bmatrix} \quad \text{(top panel of Figure 3.13)}$$

$$\mathbf{C} = \begin{bmatrix} 0.10 & 0 \\ 0 & 12 \end{bmatrix} \quad \text{(middle panel of Figure 3.13)} \tag{3.38}$$

$$\mathbf{C} = \begin{bmatrix} 0.01 & 0 \\ 0 & 12 \end{bmatrix} \quad \text{(bottom panel of Figure 3.13)}$$

and hence the quadratic functions differ only in how we set the upper-left value of the matrix $\mathbf{C}$. All three quadratics, whose contour plots are shown in the top, middle, and bottom panels of Figure 3.13 respectively, have the same global minimum at the origin. However, as we change this single value of $\mathbf{C}$ from quadratic to quadratic, we elongate the contours significantly along the horizontal axis, so much so that in the third case the contours seem almost completely parallel to each other near our initialization (an example of a *long narrow valley*).

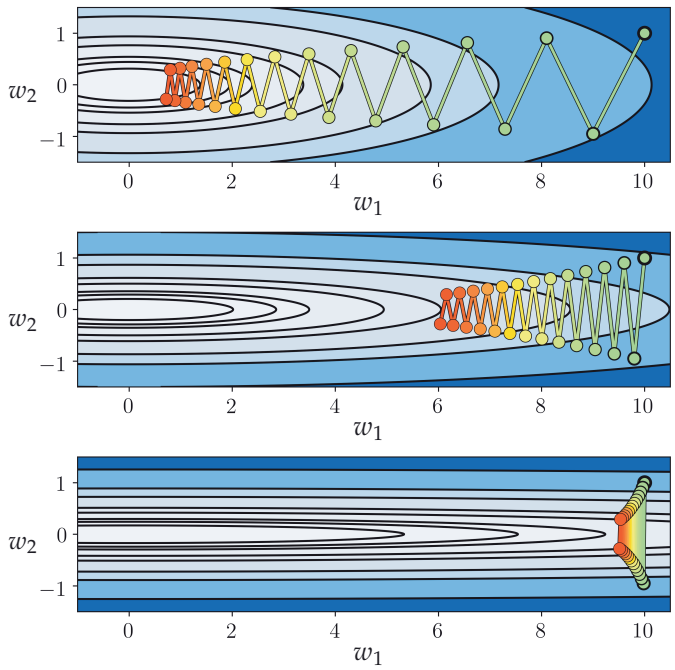

Figure 3.13 Figure associated with Example 3.13, illustrating the zig-zagging behavior of gradient descent. See text for further details.

We then make a run of 25 gradient descent steps to minimize each, using the same initialization at $\mathbf{w}^0 = [10 \;\; 1]^T$ and steplength value $\alpha = 10^{-1}$. In each case

the weights found at each step are plotted on the contour plots and colored green (at the start of the run) to red (as we reach the maximum number of iterations). Examining the figure we can see – in each case, but increasingly from the first to third example – the zig-zagging behavior of gradient descent very clearly. Indeed not much progress is made with the third quadratic at all due to the large amount of zig-zagging.

We can also see the cause of this zig-zagging: the negative gradient direction constantly points in a perpendicular direction with respect to the contours of the function, and in very narrow functions these contours become almost parallel. While it is true that we can ameliorate this zig-zagging behavior by reducing the steplength value, this does not solve the underlying problem that zig-zagging produces – which is slow convergence.

3.6.4 The slow-crawling behavior of gradient descent

As we know from the first-order condition for optimality discussed in Section 3.2, the (negative) gradient vanishes at stationary points. That is, if $\mathbf{w}$ is a minimum, maximum, or saddle point then we know that $\nabla g(\mathbf{w}) = \mathbf{0}$. Notice that this also means that the magnitude of the gradient vanishes at stationary points, that is, $\|\nabla g(\mathbf{w})\|_2 = 0$. By extension, the (negative) gradient at points near a stationary point have non-zero direction but vanishing magnitude, i.e., $\|\nabla g(\mathbf{w})\|_2 \approx 0$.

The vanishing behavior of the magnitude of the negative gradient near stationary points has a natural consequence for gradient descent steps: they progress very slowly, or "crawl," near stationary points. This occurs because, *unlike* the zero-order methods discussed in Sections 2.5 and 2.6 (where we normalized the magnitude of each descent directions), the distance traveled during each step of gradient descent is not completely determined by the steplength value α. Indeed we can easily compute the general distance traveled by a gradient descent step as

$$\left\|\mathbf{w}^k - \mathbf{w}^{k-1}\right\|_2 = \left\|\left(\mathbf{w}^{k-1} - \alpha\nabla g\left(\mathbf{w}^{k-1}\right)\right) - \mathbf{w}^{k-1}\right\|_2 = \alpha\left\|\nabla g\left(\mathbf{w}^{k-1}\right)\right\|_2. \tag{3.39}$$

In other words, the length of a general gradient descent step is equal to the value of the steplength parameter α times the magnitude of the descent direction.

The consequences of this are fairly easy to unravel. Since the magnitude of the gradient $\left\|\nabla g\left(\mathbf{w}^{k-1}\right)\right\|_2$ is large far away from stationary points, and because we often randomly initialize gradient descent in practice so that our initial points often lie far away from any stationary point of a function, the first few steps of a gradient descent run in general will be large, and make significant progress towards minimization. Conversely, when approaching a stationary point the magnitude of the gradient is small, and so the length traveled by a gradient descent step is also small. This means that gradient descent steps make little progress towards minimization when near a stationary point.

In short, the fact that the length of each step of gradient descent is proportional to the magnitude of the gradient means that gradient descent often starts off making significant progress but slows down significantly near minima and saddle points – a behavior we refer to as "slow-crawling." For particular functions this slow-crawling behavior can not only mean that many steps are required to achieve adequate minimization, but can also lead gradient descent to completely halt near saddle points of nonconvex functions.

The interested reader may note that we describe a popular solution to this slow-crawling behavior – called *normalized gradient descent* – later in Appendix Sections A.3 and A.4.

Example 3.14 The slow-crawling behavior of gradient descent

In the left panel of Figure 3.14 we plot the function

$$g(w) = w^4 + 0.1 \tag{3.40}$$

whose minimum is at the origin, which we will minimize using ten steps of gradient descent and a steplength parameter $\alpha = 10^{-1}$. We show the results of this run on the function itself (with steps colored from green at the start of the run to red at the final step). Here we can see that this run of gradient descent starts off taking large steps but crawls slowly as it approaches the minimum. Both of these behaviors are quite natural, since the magnitude of the gradient is large far from the global minimum and vanishes near it.

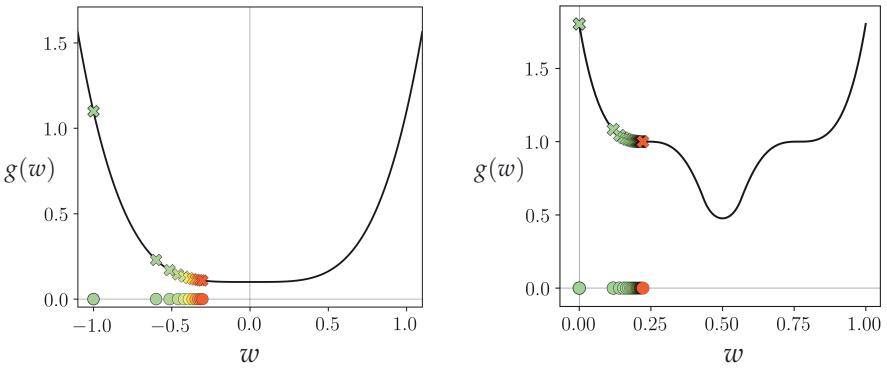

Figure 3.14 Figure associated with Example 3.14. See text for details.

In the right panel of Figure 3.14 we illustrate the crawling issue of gradient descent near saddle points using the nonconvex function

$$g(w) = \max^2(0, 1 + (3w - 2.3)^3) + \max^2(0, 1 + (-3w + 0.7)^3). \tag{3.41}$$

This function has a minimum at $w = \frac{1}{2}$ and saddle points at $w = \frac{7}{30}$ and $w = \frac{23}{30}$. We make a run of gradient descent on this function using 50 steps with $\alpha = 10^{-2}$, initialized at the origin.

Examining the right panel of the figure we can see how the gradient descent steps halt near the leftmost saddle point due to the settings (initialization and steplength parameter) chosen for this run. The fact that gradient descent crawls as it approaches this saddle point is quite natural (because the magnitude of the gradient vanishes here) but this prevents the algorithm from finding the global minimum.

3.7 Conclusion

In this chapter we described local optimization schemes that leverage a function's first derivative(s) to produce effective descent directions – otherwise known as *first-order methods*. Such methods constitute perhaps the most popular set of optimization tools used with machine learning problems.

We began in Section 3.2 by looking at how the first derivatives of a function provide a useful condition for characterizing its minima, maxima, and saddle points (together known as stationary points) via the first-order condition for optimality. In preparation for our discussion of first-order local methods we then described the *ascent* and *descent directions* of a hyperplane, as well as those provided by the tangent hyperplane associated with the first-order Taylor series approximation in Section 3.3. In Section 3.5 we saw how such descent directions, when employed in a local optimization framework, naturally lead to the construction of a popular local scheme known as *gradient descent*.

The gradient descent algorithm is widely used in machine learning, as the descent direction provided by the negative gradient is almost always readily available for use (and so a descent direction need not be sought out explicitly as with the zero-order methods described in Chapter 2). However, by its very nature the negative gradient direction has two inherent weaknesses when leveraged for local optimization – the *zig-zagging* and *slow-crawling* problems detailed in Section 3.6 – that reduce its effectiveness. These problems, and the corresponding solutions to each (collectively referred to as advanced first-order optimization methods), are discussed in significant detail in Appendix A of this text.

3.8 Exercises

† The data required to complete the following exercises can be downloaded from the text's github repository at `github.com/jermwatt/machine_learning_refined`

3.1 First-order condition for optimality

Use the first-order condition to find all stationary points of g (calculations should be done by hand). Then plot g and label the point(s) you find, and determine "by eye" whether each stationary point is a minimum, maximum, or saddle point. Note: stationary points can be at infinity!

(a) $g(w) = w\log(w) + (1-w)\log(1-w)$ where w lies between 0 and 1

(b) $g(w) = \log(1 + e^w)$

(c) $g(w) = w\tanh(w)$

(d) $g(\mathbf{w}) = \frac{1}{2}\mathbf{w}^T\mathbf{C}\mathbf{w} + \mathbf{b}^T\mathbf{w}$ where $\mathbf{C} = \begin{bmatrix} 2 & 1 \\ 1 & 3 \end{bmatrix}$ and $\mathbf{b} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$

3.2 Stationary points of a simple quadratic function

A number of applications will find us employing a simple multi-input quadratic

$$g(\mathbf{w}) = a + \mathbf{b}^T\mathbf{w} + \mathbf{w}^T\mathbf{C}\mathbf{w} \tag{3.42}$$

where the matrix $\mathbf{C} = \frac{1}{\beta}\mathbf{I}$. Here $\mathbf{I}$ is the $N \times N$ identity matrix, and $\beta > 0$ a positive scalar. Find all stationary points of g.

3.3 Stationary points of the Rayleigh quotient

The Rayleigh quotient of an $N \times N$ matrix $\mathbf{C}$ is defined as the normalized quadratic function

$$g(\mathbf{w}) = \frac{\mathbf{w}^T\mathbf{C}\mathbf{w}}{\mathbf{w}^T\mathbf{w}} \tag{3.43}$$

where $\mathbf{w} \neq \mathbf{0}_{N\times 1}$. Compute the stationary points of this function.

3.4 First-order coordinate descent as a local optimization scheme

(a) Express the coordinate descent method described in Section 3.2.2 as a local optimization scheme, i.e., as a sequence of steps of the form $\mathbf{w}^k = \mathbf{w}^{k-1} + \alpha\,\mathbf{d}^k$.

(b) Code up the coordinate descent method for the particular case of a quadratic function and repeat the experiment described in Example 3.5.

3.5 Try out gradient descent

Run gradient descent to minimize the function

$$g(w) = \frac{1}{50}\left(w^4 + w^2 + 10w\right) \tag{3.44}$$

with an initial point $w^0 = 2$ and 1000 iterations. Make three separate runs using each of the steplength values $\alpha = 1$, $\alpha = 10^{-1}$, and $\alpha = 10^{-2}$. Compute the derivative of this function by hand, and implement it (as well as the function itself) in `Python` using `NumPy`.

Plot the resulting cost function history plot of each run in a single figure to compare their performance. Which steplength value works best for this particular function and initial point?

3.6 Compare fixed and diminishing steplength values for a simple example

Repeat the comparison experiment described in Example 3.10, producing the cost function history plot comparison shown in the bottom panel of Figure 3.9.

3.7 Oscillation in the cost function history plot

Repeat the experiment described in Example 3.11, producing cost function history plot shown in the bottom panel of Figure 3.10.

3.8 Tune fixed steplength for gradient descent

Take the cost function

$$g(\mathbf{w}) = \mathbf{w}^T\mathbf{w} \tag{3.45}$$

where $\mathbf{w}$ is an $N = 10$ dimensional input vector, and g is convex with a single global minimum at $\mathbf{w} = \mathbf{0}_{N\times 1}$. Code up gradient descent and run it for 100 steps using the initial point $\mathbf{w}^0 = 10 \cdot \mathbf{1}_{N\times 1}$, with three steplength values: $\alpha_1 = 0.001$, $\alpha_2 = 0.1$, and $\alpha_3 = 1$. Produce a cost function history plot to compare the three runs and determine which performs best.

3.9 Code up momentum-accelerated gradient descent

Code up the momentum-accelerated gradient descent scheme described in Section A.2.2 and use it to repeat the experiments detailed in Example A.1 using a cost function history plot to come to the same conclusions drawn by studying the contour plots shown in Figure A.3.

3.10 Slow-crawling behavior of gradient descent

In this exercise you will compare the standard and fully normalized gradient descent schemes in minimizing the function

$$g(w_1, w_2) = \tanh(4\,w_1 + 4\,w_2) + \max(1, 0.4\,w_1^2) + 1. \tag{3.46}$$

Using the initialization $\mathbf{w}^0 = [2\ 2]^T$ make a run of 1000 steps of the standard and fully normalized gradient descent schemes, using a steplength value of $\alpha = 10^{-1}$ in both instances. Use a cost function history plot to compare the two runs, noting the progress made with each approach.

3.11 Comparing normalized gradient descent schemes

Code up the full and component-wise normalized gradient descent schemes and repeat the experiment described in Example A.4 using a cost function history plot to come to the same conclusions drawn by studying the plots shown in Figure A.6.

3.12 Alternative formal definition of Lipschitz gradient

An alternative to defining the Lipschitz constant by Equation (A.49) for functions g with Lipschitz continuous gradient is given by

$$\left\|\nabla g\left(\mathbf{x}\right) - \nabla g\left(\mathbf{y}\right)\right\|_2 \leq L\left\|\mathbf{x} - \mathbf{y}\right\|_2. \tag{3.47}$$

Using the limit definition of the derivative (see Section B.2.1) show that this definition is equivalent to the one given in Equation (A.49).

3.13 A composition of functions with Lipschitz gradient

Suppose f and g are two functions with Lipschitz gradient with constants L and K respectively. Using the definition of Lipschitz continuous gradient given in Exercise 3.12 show that the composition $f\left(g\right)$ also has Lipschitz continuous gradient. What is the corresponding Lipschitz constant of this composition?