2 Zero-Order Optimization Techniques

2.1 Introduction

The problem of determining the smallest (or largest) value a function can take, referred to as its *global minimum* (or *global maximum*), is a centuries-old pursuit that has numerous applications throughout the sciences and engineering. In this chapter we begin our investigation of mathematical optimization by describing the *zero-order optimization* techniques – also referred to as *derivative-free optimization* techniques. While not always the most powerful optimization tools at our disposal, zero-order techniques are conceptually the simplest tools available to us – requiring the least amount of intellectual machinery and jargon to describe. Because of this, discussing zero-order methods first allows us to lay bare, in a simple setting, a range of crucial concepts we will see throughout the chapters that follow in more complex settings – including the notions of *optimality*, *local optimization*, *descent directions*, *steplengths*, and more.

2.1.1 Visualizing minima and maxima

When a function takes in only one or two inputs we can attempt to visually identify its minima or maxima by plotting it over a large swath of its input space. While this idea certainly fails when a function takes in three or more inputs (since we can no longer visualize it properly), we begin nevertheless by first examining a number of low-dimensional examples to gain an intuitive feel for how we might effectively identify these desired minima or maxima in general.

Example 2.1 Visual inspection of single-input functions for minima and maxima

In the top-left panel of Figure 2.1 we plot the quadratic function

$$g(w) = w^2 \tag{2.1}$$

over a short region of its input space (centered around zero from $w = -3$ to $w = 3$). In this figure we also mark the evaluation of the function's global

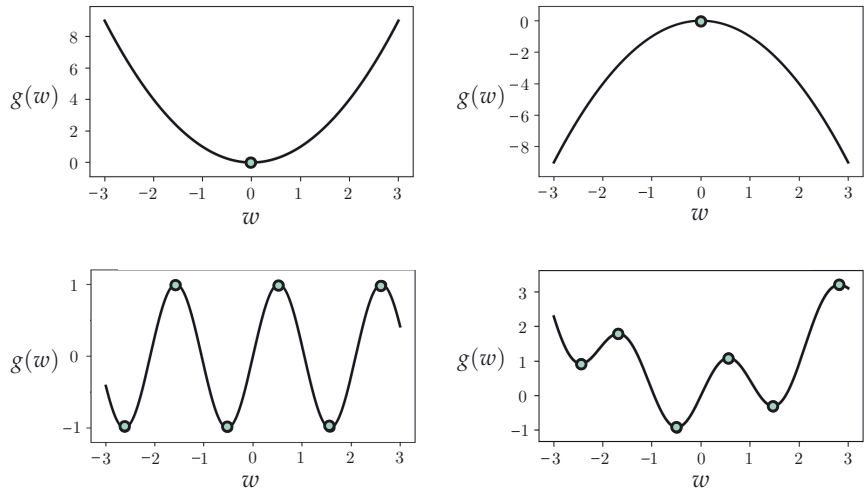

Figure 2.1 Figure associated with Example 2.1. Four example functions are shown with the evaluation of their minima and maxima highlighted by green dots. See text for further details.

minimum at $w = 0$ (that is, the point $(0, g(0))$ where $g(0) = 0$) with a green dot. Note that as we move farther away from the origin (in either the negative or positive direction) the evaluation of g becomes progressively larger, implying that its global maxima lies at $w = \pm\infty$.

In the top-right panel in Figure 2.1 we show the result of multiplying the previous quadratic function by -1, giving the new quadratic

$$g(w) = -w^2. \tag{2.2}$$

Doing so causes the function to flip upside down, with its global minima now lying at $w = \pm\infty$, and the input $w = 0$ that once provided the *global minimum* of g now returns its *global maximum*. The evaluation of this maximum is again marked with a green dot on the function.

In the bottom-left panel of Figure 2.1 we plot the sinusoidal function

$$g(w) = \sin(3w). \tag{2.3}$$

Here we can clearly see that (over the range we have plotted the function) there are three global minima and three global maxima (the evaluation of each is marked by a green dot on the function). Indeed if we drew this function out over a wider and wider swath of its input we would see that it has infinitely many such global minima and maxima (existing at every odd multiple of $\frac{\pi}{6}$).

In the bottom-right panel of Figure 2.1 we look at the sum of a sinusoidal and a quadratic function, which takes the algebraic form

$$g(w) = \sin(3w) + 0.3\,w^2. \tag{2.4}$$

Inspecting this function (over the range it is plotted) we can see that it has a global minimum around $w = -0.5$. The function also has other minima and maxima that are *locally optimal*, meaning values that are minimal or maximal only locally and with respect to just their neighbors (and not the function as a whole). For example, g has a local maximum near $w = 0.6$ and a local minimum near $w = 1.5$. The evaluation of both maxima and minima over the range of input shown for this function are marked by a green dot in the figure.

2.2 The Zero-Order Optimality Condition

With a number of simple examples illustrating minima and maxima we can now define them more formally. The task of determining a global minimum of a function g with N input variables $w_1, w_2, \ldots, w_N$ can formally be phrased as the following *minimization problem*

$$\underset{w_1, w_2, \ldots, w_N}{\text{minimize}} \;\; g(w_1, w_2, \ldots, w_N) \tag{2.5}$$

which can be rewritten much more compactly (by stacking all the inputs in an N-dimensional vector $\mathbf{w}$) as

$$\underset{\mathbf{w}}{\text{minimize}} \;\; g(\mathbf{w}). \tag{2.6}$$

By solving such minimization problem we aim to find a point $\mathbf{w}^\star$ such that

$$g(\mathbf{w}^\star) \leq g(\mathbf{w}) \quad \text{for all } \mathbf{w}. \tag{2.7}$$

This is the *zero-order* definition of a global minimum. In general, a function can have multiple or even infinitely many global minimum points (like the sinusoidal function in Equation (2.3)).

We can likewise describe mathematical points $\mathbf{w}^\star$ at which g has a global maximum. For such points we can write

$$g(\mathbf{w}^\star) \geq g(\mathbf{w}) \quad \text{for all } \mathbf{w}. \tag{2.8}$$

This is the *zero-order* definition of a global maximum. To express our pursuit of a global maximum of a function we then write

$$\underset{\mathbf{w}}{\text{maximize}} \;\; g(\mathbf{w}). \tag{2.9}$$

Note that the concepts of minima and maxima of a function are always related

to each other via multiplication by -1. That is, a global minimum of a function g is always a global maximum of the function $-g$, and vice versa. Therefore we can always express the maximization problem in Equation (2.9) in terms of a minimization problem, as

$$\underset{\mathbf{w}}{\text{minimize}} \quad -g(\mathbf{w}). \tag{2.10}$$

Akin to zero-order definitions for global minima and maxima in Equations (2.7) and (2.8), there are zero-order definitions for local minima and maxima as well. For instance, we can say a function g has a *local* minimum at a point $\mathbf{w}^\star$ if

$$g(\mathbf{w}^\star) \leq g(\mathbf{w}) \quad \text{for all } \mathbf{w} \text{ near } \mathbf{w}^\star. \tag{2.11}$$

The statement "for all $\mathbf{w}$ near $\mathbf{w}^\star$" is relative, simply describing the fact that a neighborhood (however small) around $\mathbf{w}^\star$ must exist such that, when evaluated at every point in this neighborhood, the function g attains its smallest value at $\mathbf{w}^\star$. The same formal zero-order definition can be made for local maxima as well, switching the $\leq$ sign to $\geq$.

Packaged together, these zero-order definitions for minima and maxima (collectively called optima) are often referred to as *the zero-order condition for optimality*. The phrase *zero-order* in this context refers to the fact that in each case, the optima of a function are defined in terms of the function itself (and nothing else). In further chapters we will see *higher-order* definitions of optimal points, specifically the *first-order* definitions that involve the first derivative of a function in Chapter 3, as well as *second-order* definitions involving a function's second derivative in Chapter 4.

2.3 Global Optimization Methods

In this section we describe the first approach one might take to approximately minimize an arbitrary function: evaluate the function using a large number of input points and treat the input that provides the lowest function value as the approximate global minimum of the function. This approach is called a *global* optimization method because it is capable of approximating the global optima of a function (provided a large enough number of evaluations are made).

The important question with this sort of optimization scheme is: how do we choose the inputs to try out with a generic function? We clearly cannot try them all since, even for a single-input continuous function, there are an infinite number of points to try.

We can take two approaches here to choosing our (finite) set of input points to test: either sample (i.e., guess) them uniformly over an evenly spaced grid (uniform sampling), or pick the same number of input points at random (random sampling). We illustrate both choices in Example 2.2.

Example 2.2 Minimizing a quadratic function

Here we illustrate two sampling methods for finding the global minimum of the quadratic function

$$g(w) = w^2 + 0.2 \tag{2.12}$$

which has a global minimum at $w = 0$. For the sake of simplicity we limit the range over which we search to $[-1, +1]$. In the top row of Figure 2.2 we show the result of a uniform-versus-random sampling of four inputs, shown in blue in each panel of the figure (with corresponding evaluations shown in green on the function itself). We can see that by randomly sampling here we were able to (by chance) achieve a slightly lower point when compared to sampling the function evenly. However, using enough samples we can find an input very close to the true global minimum of the function with either sampling approach. In the bottom row we show the result of sampling 20 inputs uniformly versus randomly, and we can see that, by increasing the number of samples, using either approach, we are now able to approximate the global minimum with much better precision.

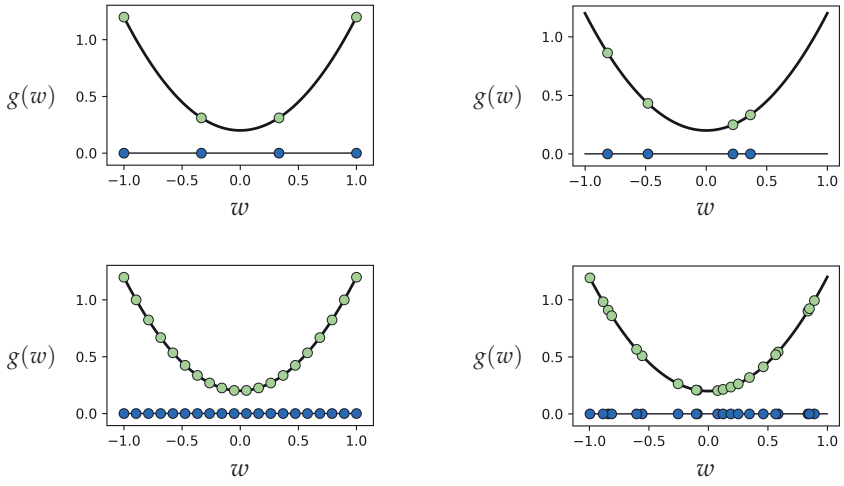

Figure 2.2 Figure associated with Example 2.2. Minimizing a simple function via sampling – or "guessing." (top row) Sampling the input of a function four times evenly (left panel) and at random (right panel). Here the inputs chosen are shown as blue dots, and their evaluations by the function are shown as green dots. (bottom row) Sampling uniformly (left panel) and randomly (right panel) 20 times. The more samples we take, the more likely we are to find a point close to the global minimum using either sampling approach. See text for further details.

Note that with both global optimization approaches discussed in Example 2.2 we are simply employing the zero-order optimality condition, since from a set of K chosen inputs $\left\{\mathbf{w}^k\right\}_{k=1}^K$ we are choosing the one input $\mathbf{w}^j$ with lowest evaluation on the cost function

$$g\left(\mathbf{w}^j\right) \leq g\left(\mathbf{w}^k\right) \qquad k = 1, 2, ..., K \tag{2.13}$$

which is indeed an approximation to the zero-order optimality condition discussed in the previous section.

While easy to implement and perfectly adequate for functions having low-dimensional input, as we see next, this naturally zero-order framework fails miserably when the input dimension of a function grows to even moderate size.

2.3.1 The curse of dimensionality

While this sort of global optimization based on zero-order evaluations of a function works fine for low-dimensional functions, it quickly fails as we try to tackle functions with larger number of inputs or, in other words, functions that take in N-dimensional input $\mathbf{w}$ where N is large. This makes such optimization methods essentially unusable in modern machine learning since the functions we often deal with have input dimensions ranging from the hundreds to the hundreds of thousands, or even millions.

To get a sense of why the global approach quickly becomes infeasible, imagine we use a uniform sampling of points across the input space of a single-input function, choosing (for the sake of argument) three points, each at a distance of d from the previous one, as illustrated in the top-left panel of Figure 2.3. Imagine now that the input space of the function increases by one, and that the range of each input is precisely that of the original single-input function, as illustrated in the top-middle panel of Figure 2.3. We still aim to cover the space evenly and with enough samples such that each input we evaluate is once again at a distance d from its closest neighbors in either direction. Notice, in order to do this in a now two-dimensional space we need to sample $3^2 = 9$ input points. Likewise if we increase the dimension of the input once again in the same fashion, in order to sample evenly across the input space so that each input is at a maximum distance of d from its neighbors in every input dimension we will need $3^3 = 27$ input points, as illustrated in the top-right panel of Figure 2.3. If we continue this thought experiment, for a general N-dimensional input we would need to sample 3^N points, a huge number even for moderate values of N. This is a simple example of the so-called *curse of dimensionality* which, generally speaking, describes the exponential difficulty one encounters when trying to deal with functions of increasing input dimension.

The curse of dimensionality remains an issue even if we decide to take samples randomly. To see why this is the case using the same hypothetical scenario, suppose now that, instead of fixing the distance d of each sample from its

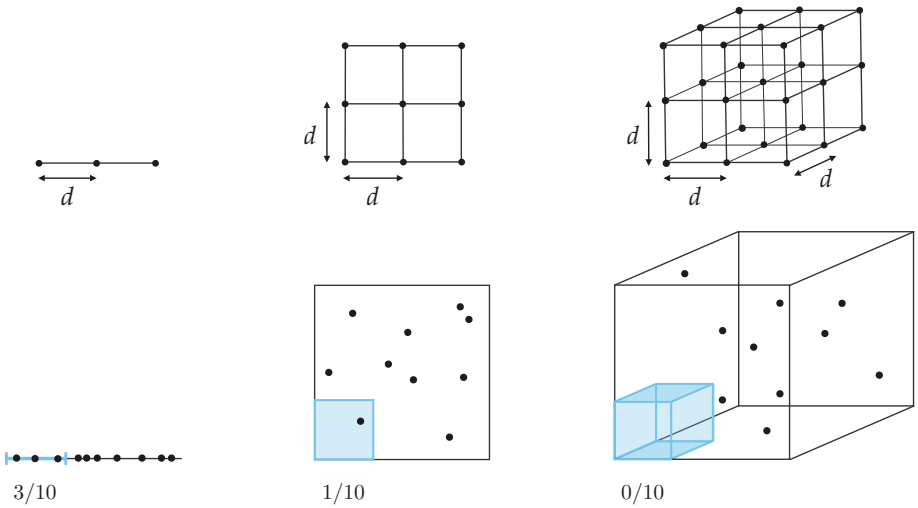

Figure 2.3 (top row) The number of input points we must sample uniformly if we wish each to be at a distance of d from its neighbors grows exponentially as the input dimension of a function increases. If three points are used to cover a single-input space in this way (left panel), $3^2 = 9$ points are required in two dimensions, and $3^3 = 27$ points in three dimensions (and this trend continues). Sampling randomly (bottom row) does not solve the problem either. See text for further details.

neighbors, we fix the total number of randomly chosen samples to a fixed value and look at how well they tend to distribute over an input space as we increase its dimension. From left to right in the bottom panels of Figure 2.3 we see one instance of how a total of ten points are randomly selected in $N = 1$, $N = 2$, and $N = 3$ dimensional space, respectively. Once again we are inhibited by the curse of dimensionality. As we increase the dimension of the input space the average number of samples per unit hypercube drops exponentially, leaving more and more regions of the space without a single sample or corresponding evaluation. In order to counteract this problem we would need to start sampling exponentially many points, leading to the same problem we encounter with the uniform sampling scheme.

2.4 Local Optimization Methods

As opposed to the global optimization techniques described in Section 2.3, where a large number of input points are sampled simultaneously with the lowest evaluation crowned the approximate global minimum, *local optimization methods* work by starting with a single input point and then by sequentially refining it, driving the point towards an approximate minimum point. Local optimization methods are by far the most popular mathematical optimization

schemes used in machine learning today, and are the subject of the remainder of this part of the text. While there is substantial variation in the kinds of specific local optimization methods we will discuss going forward, nonetheless they all share a common overarching framework that we introduce in this section.

2.4.1 The big picture

Starting with a sample input, usually referred to as an *initial point* and denoted throughout the text by $\mathbf{w}^0$, local optimization methods refine this initial point sequentially, pulling it *downhill* towards points that are lower and lower on the function, eventually reaching a minimum as illustrated for a single-input function in Figure 2.4. More specifically, from $\mathbf{w}^0$ the point is pulled downhill to a new point $\mathbf{w}^1$ lower on the function, i.e., where $g\left(\mathbf{w}^0\right) > g\left(\mathbf{w}^1\right)$. The point $\mathbf{w}^1$ itself is then pulled downwards to a new point $\mathbf{w}^2$. Repeating this process K times yields a sequence of K points (excluding our starting initial point)

$$\mathbf{w}^0,\ \mathbf{w}^1,\ ...,\ \mathbf{w}^K \tag{2.14}$$

where each subsequent point is (generally speaking) on a lower and lower portion of the function, i.e.,

$$g\left(\mathbf{w}^0\right) > g\left(\mathbf{w}^1\right) > \cdots > g\left(\mathbf{w}^K\right). \tag{2.15}$$

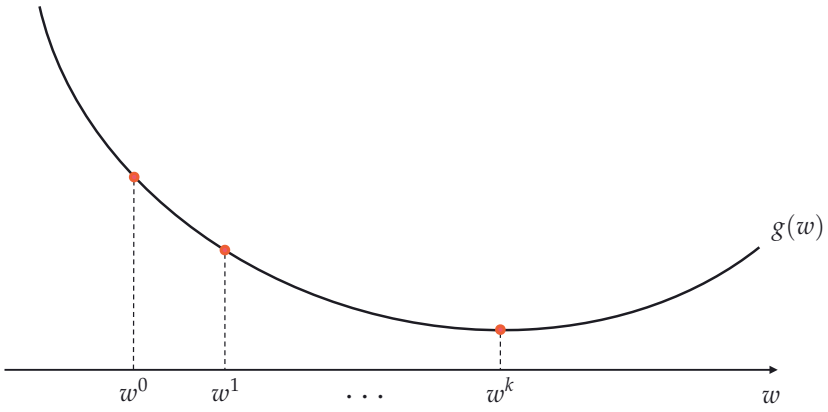

Figure 2.4 Local optimization methods work by minimizing a target function in a sequence of steps. Shown here is a generic local optimization method applied to minimize a single-input function. Starting with the initial point w^0, we move towards lower points on the cost function like a ball rolling downhill.

2.4.2 The general framework

In general the sequential refinement process enacted by a local optimization method works as follows. To take the first step from an initial point $\mathbf{w}^0$ to the very first update $\mathbf{w}^1$, what is called a *descent direction* at $\mathbf{w}^0$ is found. This is a direction vector $\mathbf{d}^0$ in the input space, beginning at $\mathbf{w}^0$ and pointing away from it towards a new point $\mathbf{w}^1$ with lower function evaluation. When such a direction is found the first update $\mathbf{w}^1$ is given by the sum

$$\mathbf{w}^1 = \mathbf{w}^0 + \mathbf{d}^0. \tag{2.16}$$

To refine the point $\mathbf{w}^1$ we look for a new descent direction $\mathbf{d}^1$, one that moves downhill stemming from the point $\mathbf{w}^1$. When we find such a direction the second update $\mathbf{w}^2$ is then formed as the sum

$$\mathbf{w}^2 = \mathbf{w}^1 + \mathbf{d}^1. \tag{2.17}$$

We repeat this process, producing a sequence of input points

$$\begin{aligned} &\mathbf{w}^0 \\ &\mathbf{w}^1 = \mathbf{w}^0 + \mathbf{d}^0 \\ &\mathbf{w}^2 = \mathbf{w}^1 + \mathbf{d}^1 \\ &\mathbf{w}^3 = \mathbf{w}^2 + \mathbf{d}^2 \\ &\quad\vdots \\ &\mathbf{w}^K = \mathbf{w}^{K-1} + \mathbf{d}^{K-1} \end{aligned} \tag{2.18}$$

where $\mathbf{d}^{k-1}$ is the descent direction determined at the kth step of the process, defining the kth step as $\mathbf{w}^k = \mathbf{w}^{k-1} + \mathbf{d}^{k-1}$ such that in the end the inequalities in Equation (2.15) are met. This is illustrated schematically with a generic function taking in two inputs in the top panel of Figure 2.5. The two-input function is illustrated here via a *contour plot*, a common visualization tool that allows us to project a function down onto its input space. Darker regions on the plot correspond to points with larger evaluations (higher on the function), while brighter regions correspond to points with smaller evaluations (lower on the function).

The descent directions in Equation (2.18) can be found in a multitude of ways. In the remaining sections of this chapter we discuss *zero-order* approaches for doing so, and in the following chapters we describe the so-called *first-* and *second-order* approaches, i.e., approaches that leverage the first- and/or second-order derivative of a function to determine descent directions. How the descent directions are determined is precisely what distinguishes major local optimization schemes from one another.

2.4.3 The steplength parameter

We can compute how far we travel at each step of a local optimization method by examining the general form of a local step. Making this measurement we can see that, at the kth step as defined in Equation (2.18), we move a distance equal to the length of the corresponding descent direction

$$\left\|\mathbf{w}^{k}-\mathbf{w}^{k-1}\right\|_2 = \left\|\mathbf{d}^{k-1}\right\|_2 . \tag{2.19}$$

This can mean that the *length* of descent vectors could be problematic even if they point in a descent direction, downhill. For example, if they are too long, as illustrated in the middle panel of Figure 2.5, then a local method can oscillate wildly at each update step, never reaching an approximate minimum. Likewise if they are too small in length, a local method will move so sluggishly slow, as illustrated in the bottom panel of Figure 2.5, that far too many steps would be required to reach an approximate minimum.

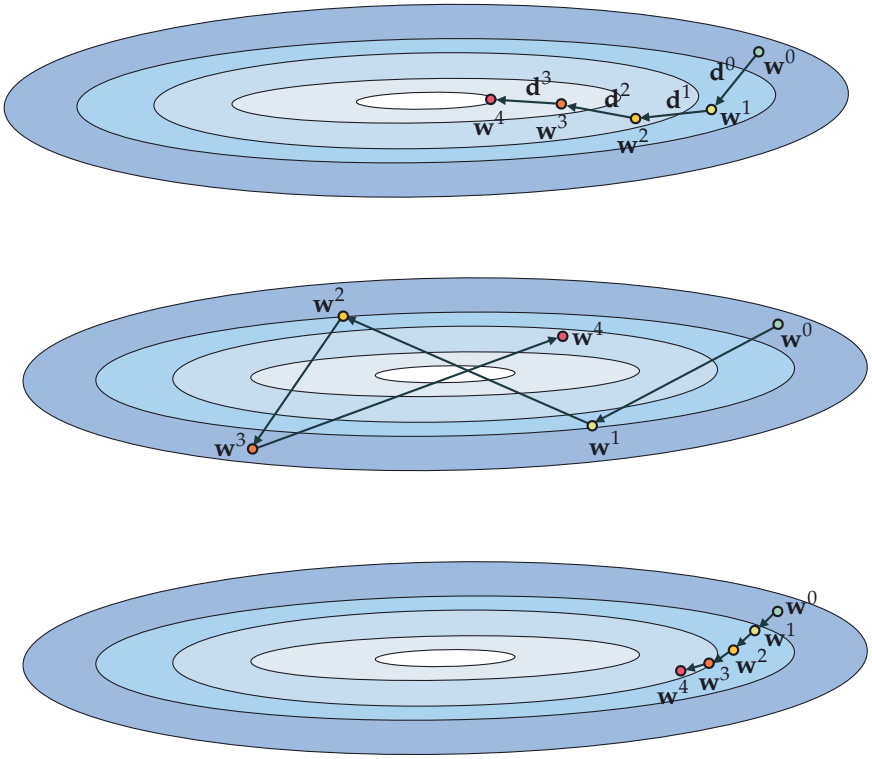

Figure 2.5 (top panel) Schematic illustration of a generic local optimization scheme applied to minimize a function taking in two inputs, with descent directions marked in black. See text for further details. (middle panel) Direction vectors are too large, causing a wild oscillatory behavior around the minimum point. (bottom panel) Direction vectors are too small, requiring a large number of steps be taken to reach the minimum.

Because of this potential problem many local optimization schemes come equipped with what is referred to as a *steplength parameter*, also called a *learning rate* in the jargon of machine learning. This parameter, which allows us to control the length of each update step (hence the name steplength parameter), is typically denoted by the Greek letter α. With a steplength parameter the generic kth update step is written as

$$\mathbf{w}^k = \mathbf{w}^{k-1} + \alpha \mathbf{d}^{k-1}. \tag{2.20}$$

The entire sequence of K steps is then similarly written as

$$\begin{aligned} &\mathbf{w}^0 \\ &\mathbf{w}^1 = \mathbf{w}^0 + \alpha \mathbf{d}^0 \\ &\mathbf{w}^2 = \mathbf{w}^1 + \alpha \mathbf{d}^1 \\ &\mathbf{w}^3 = \mathbf{w}^2 + \alpha \mathbf{d}^2 \\ &\quad \vdots \\ &\mathbf{w}^K = \mathbf{w}^{K-1} + \alpha \mathbf{d}^{K-1}. \end{aligned} \tag{2.21}$$

Note the only difference between this form for the kth step and the original is that now we scale the descent direction $\mathbf{d}^{k-1}$ by the steplength parameter $\alpha > 0$. With the addition of this parameter the distance traveled at the kth step of a generic local optimization scheme can be computed as

$$\left\| \mathbf{w}^k - \mathbf{w}^{k-1} \right\|_2 = \left\| \left(\mathbf{w}^{k-1} + \alpha \mathbf{d}^{k-1} \right) - \mathbf{w}^{k-1} \right\|_2 = \alpha \left\| \mathbf{d}^{k-1} \right\|_2. \tag{2.22}$$

In other words, the length of the kth step is now *proportional* to the length of the descent vector, and we can fine tune precisely how far we wish to travel in this direction by setting the value of α properly. A common choice is to set α to some *fixed* small value for each of the K steps. However (just like local optimization methods themselves), there are a number of ways of setting the steplength parameter which we will discuss later in the current and future chapters.

2.5 Random Search

In this section we describe our first local optimization algorithm: *random search*. With this instance of the general local optimization framework we seek out a descent direction at each step by examining a number of random directions stemming from our current point. This manner of determining a descent direction, much like the global optimization scheme described in Section 2.3, scales poorly with the dimension of input, which ultimately disqualifies random search for use with today's large-scale machine learning problems. However, this zero-order approach to local optimization is extremely useful as a simple example of the general framework introduced previously, allowing us to give a simple yet

concrete algorithmic example of universally present ideas like *descent directions*, various choices for the *steplength parameter*, and issues of *convergence*.

2.5.1 The big picture

The defining characteristic of the *random search* (as is the case with every major local optimization scheme) lies in how the descent direction $\mathbf{d}^{k-1}$ at the kth local optimization update step $\mathbf{w}^k = \mathbf{w}^{k-1} + \mathbf{d}^{k-1}$ is chosen.

With random search we do (perhaps) the "laziest" possible thing one could think to do in order to find a descent direction: we sample a given number of random directions stemming from $\mathbf{w}^{k-1}$, evaluate each candidate update point, and choose the one that gives us the smallest evaluation (so long as it is indeed lower on the function than our current point). In other words, we look locally around the current point, in a certain number of random directions, for a point that has a lower evaluation, and if we find one we move to it.

To be more precise, at the kth step we generate P random directions $\{\mathbf{d}^p\}_{p=1}^P$ to try out, each stemming from the prior step $\mathbf{w}^{k-1}$ and leading to the candidate point $\mathbf{w}^{k-1} + \mathbf{d}^p$ for $p = 1, 2, ..., P$.

After evaluating all such P candidate points we pick the one that gives us the smallest evaluation, i.e., the one with the index given by

$$s = \underset{p=1,2,...,P}{\text{argmin}} \; g\left(\mathbf{w}^{k-1} + \mathbf{d}^p\right). \tag{2.23}$$

Finally, if the best point found has a smaller evaluation than the current point, i.e., if $g\left(\mathbf{w}^{k-1} + \mathbf{d}^s\right) < g\left(\mathbf{w}^{k-1}\right)$, then we move to the new point $\mathbf{w}^k = \mathbf{w}^{k-1} + \mathbf{d}^s$, otherwise we either halt the method or try another batch of P random directions.

The random search method is illustrated in Figure 2.6 using a quadratic function where, for visualization purposes, the number of random directions to try is set relatively small to $P = 3$.

2.5.2 Controlling the length of each step

In order to better control the progress of random search we can *normalize* our randomly chosen directions to each have unit length, i.e., $\|\mathbf{d}\|_2 = 1$. This way we can adjust each step to have whatever length we desire by introducing a steplength parameter α (as discussed in Section 2.4.3). This more general step $\mathbf{w}^k = \mathbf{w}^{k-1} + \alpha\mathbf{d}$ now has length exactly equal to the steplength parameter α, as

$$\|\mathbf{w}^k - \mathbf{w}^{k-1}\|_2 = \|\alpha\,\mathbf{d}\|_2 = \alpha\|\mathbf{d}\|_2 = \alpha. \tag{2.24}$$

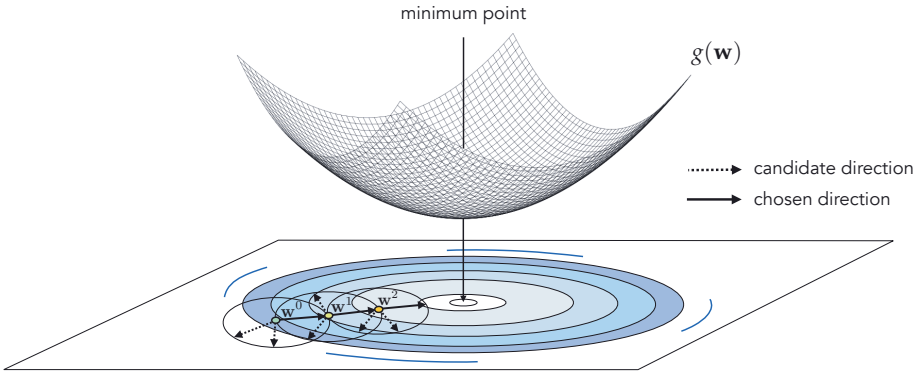

Figure 2.6 At each step the random search algorithm determines a descent direction by examining a number of random directions. The direction leading to the new point with the smallest evaluation is chosen as the descent direction, and the process repeated until a point near a local minimum is found. Here we show three prototypical steps performed by random search, where three random directions are examined at each step. At each step the best descent direction found is drawn as a solid black arrow while the other two inferior directions are shown as dashed black arrows.

Example 2.3 Minimizing a simple quadratic using random search

In this example we run random local search for $K = 5$ steps with $\alpha = 1$ for all steps, at each step searching for $P = 1000$ random directions to minimize the quadratic function

$$g(w_1, w_2) = w_1^2 + w_2^2 + 2. \tag{2.25}$$

Figure 2.7 shows the function in three dimensions on the top-left panel, along with the set of steps produced by the algorithm colored from green (at the start of the run where we initialize at $\mathbf{w}^0 = [3 \;\; 4]^T$) to red (when the algorithm halts). Directed arrows illustrate each descent direction chosen, connecting each step to its predecessor, and are shown to help illustrate the total path the algorithm takes. In the top-right panel we show the same function, but viewed from directly above as its *contour plot*.

Notice that if the dimension of the input N is greater than 2 we cannot make a plot like the ones shown in the figure to tell how well a particular run of any local optimization (here random search) is performed. A more general way to visualize the progress made by a local method is to plot the corresponding sequence of function evaluations against the step number, i.e., plotting the pairs $\left(k, g\left(\mathbf{w}^k\right)\right)$ for $k = 1, 2, ..., K$, as demonstrated in the bottom panel of Figure 2.7. This allows us to tell (regardless of the input dimension N of the function being minimized) how the algorithm performed, and whether we need to adjust any of its parameters (e.g., the maximum number of steps K or the value of α). This

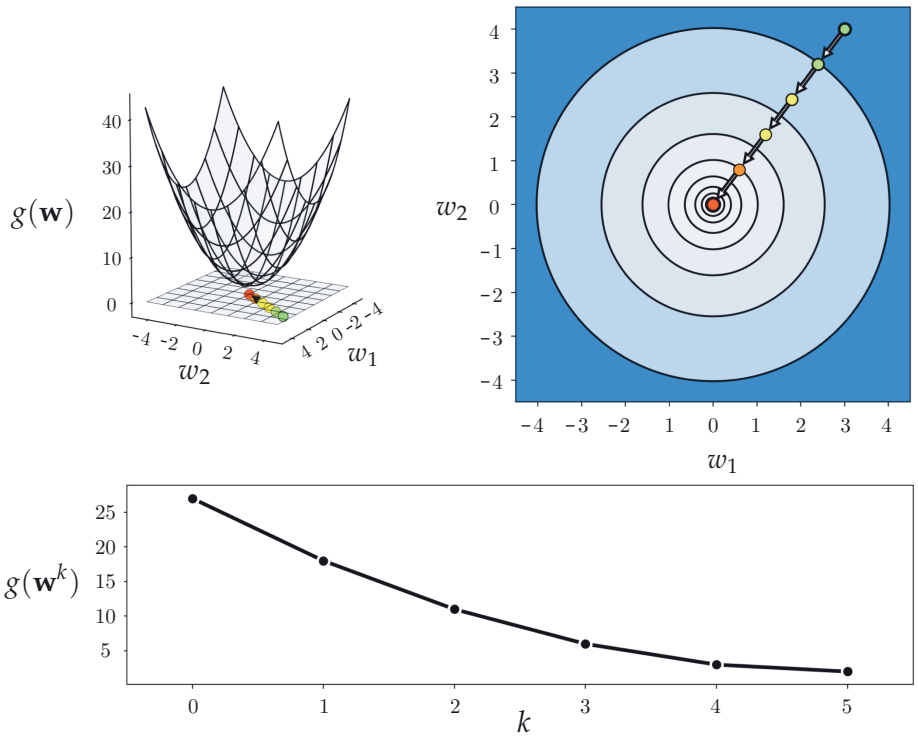

Figure 2.7 Figure associated with Example 2.3. See text for further details.

visualization is called a *cost function history plot*. An additional benefit of such a plot is that we can more easily tell the exact value of each function evaluation during the local optimization run.

Example 2.4 Minimizing a function with multiple local minima

In this example we show what one may need to do in order to find the global minimum of a function using a local optimization scheme like random search. For visualization purposes we use the single-input function

$$g(w) = \sin(3w) + 0.3w^2 \tag{2.26}$$

and initialize two runs, one starting at $w^0 = 4.5$ and another at $w^0 = -1.5$. For both runs we use a steplength of $\alpha = 0.1$ fixed for all $K = 10$ steps or iterations. As can be seen in Figure 2.8, depending on where we initialize, we may end up near a local (left panel) or global minimum (right panel). Here we illustrate the steps of each run as circles along the input axis with corresponding evaluations on the function itself as similarly colored `x` marks. The steps of each run are colored green near the start of the run to red when a run halts.

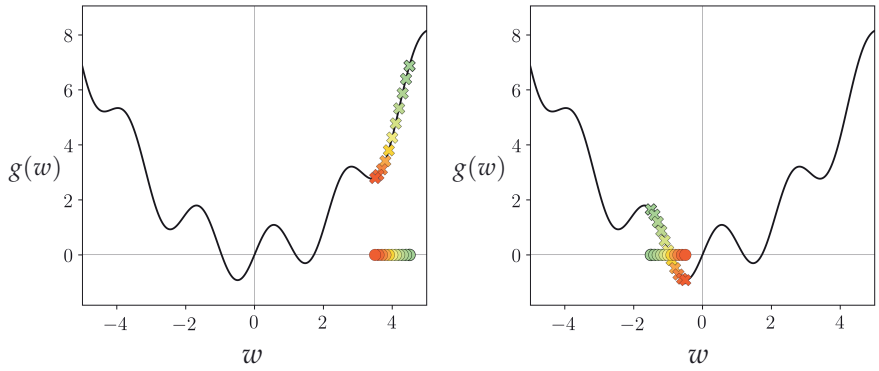

Figure 2.8 Figure associated with Example 2.4. Properly minimizing such a nonconvex function like the one shown here requires multiple runs of local optimization with different initializations. See text for further details.

2.5.3 Exploring fundamental steplength rules

In Examples 2.3 and 2.4 we set the steplength parameter α fixed for all steps of each run. Known as a *fixed steplength rule*, this is a very common choice of steplength parameter for local optimization methods in general. It is also possible to change the value of α from one step to another with what is often referred to as an *adjustable steplength rule*. Before exploring a very common adjustable steplength rule used in machine learning, called the *diminishing* steplength rule, we first show the importance of steplength tuning through a simple example.

Example 2.5 A failure to converge

In this example we use random search to minimize the quadratic function

$$g(w_1, w_2) = w_1^2 + w_2^2 + 2 \tag{2.27}$$

using the steplength $\alpha = 1$ (as we used in Example 2.3) but with a different initialization at $\mathbf{w}^0 = [1.5 \;\; 2]^T$. However, with this initialization and as shown by examining the contour plot of this function in the left panel of Figure 2.9, the algorithm gets stuck at a non-optimal point (colored red) away from the global minimum point located at the origin, where here the contour plot is shown without color for better visualization. Also drawn in the same plot is a blue unit circle centered at the final red point, representing the location of all possible points the algorithm could take us to if we decided to take another step and move from where it halts at the red point. Notice how this blue circle encompasses one of the contours of the quadratic (in dashed red) on which the final red point lies. This means that every possible direction provides ascent, not descent, and the algorithm must therefore halt.

We need to be careful when choosing the steplength value with this simple

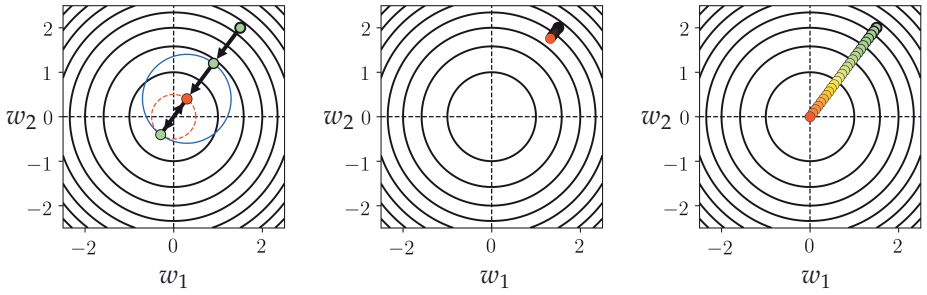

Figure 2.9 Figure associated with Example 2.5. Determining a proper steplength is crucial to optimal performance with random search – and by extension many local optimization algorithms. Here choosing the steplength too large leads to the method halting at a suboptimal point (left panel), setting it too small leads to very slow convergence towards the minimum of the function (middle panel), and setting it "just right" leads to ideal convergence to a point close to the function's minimum (right panel). See text for further details.

quadratic function, and by extension any general function. If, as shown in the middle panel of Figure 2.9, we repeat the same experiment but cut the steplength down to $\alpha = 0.01$, we do not reach a point anywhere near the global minimum within the same number of steps.

Setting the steplength parameter a little larger to $\alpha = 0.1$ for all steps, we make another run mirroring the previous one with results shown in the right panel of Figure 2.9. The algorithm now converges to a point much closer to the global minimum of the function at the origin.

In general, the combination of steplength and maximum number of iterations are best chosen together. The trade-off here is straightforward: a small steplength combined with a large number of steps can guarantee convergence to a local minimum, but can be computationally expensive. Conversely, a large steplength and small number of maximum iterations can be cheaper but less effective at finding the optimal. Often, in practice, these kinds of choices are made by making several runs of an algorithm and plotting their corresponding cost function histories to determine optimal parameter settings.

2.5.4 Diminishing steplength rules

A commonly used alternative to fixed steplength rules are the so-called *diminishing steplength rules* wherein we shrink or *diminish* the size of the steplength at each step of local optimization. One common way of producing a diminishing steplength rule is simply to set $\alpha = \frac{1}{k}$ at the kth step of the process. This provides the benefit of shrinking the distance between subsequent steps as we progress

on a run, since with this choice of steplength and a unit-length descent direction vector we have

$$\|\mathbf{w}^k - \mathbf{w}^{k-1}\|_2 = \alpha\|\mathbf{d}^{k-1}\|_2 = \frac{1}{k}. \tag{2.28}$$

At the same time, if we sum up the total distance the algorithm travels in K steps (provided we indeed move at each step) we have

$$\sum_{k=1}^{K} \|\mathbf{w}^k - \mathbf{w}^{k-1}\|_2 = \sum_{k=1}^{K} \frac{1}{k}. \tag{2.29}$$

The beauty of this sort of diminishing steplength is that while the steplength $\alpha = \frac{1}{k}$ decreases to zero as k increases, the total distance traveled by the algorithm goes to infinity.[1] This means that a local algorithm employing this sort of diminishing steplength rule can – in theory – move around an infinite distance in search of a minimum all the while taking smaller and smaller steps, which allows it to work into any small nooks and crannies a function might have where any minimum lies.

2.5.5 Random search and the curse of dimensionality

As with the global optimization approach discussed in Section 2.3, the curse of dimensionality also poses a major obstacle to the practical application of the random search algorithm as the dimension of a function's input increases. In other words, for most functions it becomes *exponentially* more difficult to find a descent direction *at random* at a given point as its input dimension increases.

Take, for example, the single-input quadratic function $g(w) = w^2 + 2$ and suppose we take a single step using the random search algorithm beginning at $w^0 = 1$ with steplength $\alpha = 1$. As illustrated in the top panel of Figure 2.10, because the input dimension in this case is $N = 1$, to determine a descent direction we only have two directions to consider: the negative and positive directions from our initial point. One of these two directions will provide descent

[1] The sum $\sum_{k=1}^{\infty} \frac{1}{k}$ is often called the *harmonic series*, and one way to see that it diverges to infinity is by lumping together consecutive terms as

$$\begin{aligned}\sum_{k=1}^{\infty} \frac{1}{k} &= 1 + \frac{1}{2} + \left(\frac{1}{3} + \frac{1}{4}\right) + \left(\frac{1}{5} + \frac{1}{6} + \frac{1}{7} + \frac{1}{8}\right) + \cdots \\ &\geq 1 + \frac{1}{2} + 2\left(\frac{1}{4}\right) + 4\left(\frac{1}{8}\right) + \cdots \\ &= 1 + \frac{1}{2} + \frac{1}{2} + \frac{1}{2} + \cdots.\end{aligned} \tag{2.30}$$

In other words, the harmonic series is lower-bounded by an infinite sum of $\frac{1}{2}$ values, and thus diverges to infinity.

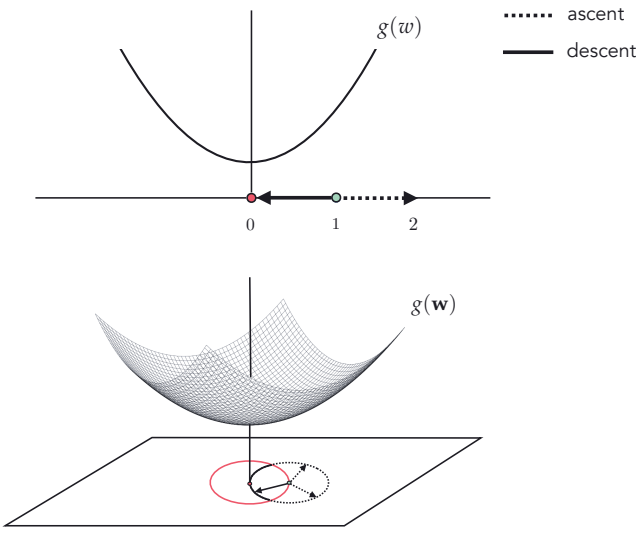

Figure 2.10 (top panel) When the input dimension is $N = 1$, there are only two unit directions we can move in, only one of which (the solid arrow) is a descent direction. (bottom panel) When the input dimension is $N = 2$, there are infinitely many unit directions to choose from, only a fraction of which whose endpoint lies inside the unit circle (points on the solid portion of the arc) are descent directions.

(here, the negative direction). In other words, we would have a 50 percent chance of finding a descent direction if were to choose one at random.

Now let us examine the same sort of quadratic function, this time one that takes in an $N = 2$ dimensional vector $\mathbf{w}$ as input: $g(\mathbf{w}) = \mathbf{w}^T\mathbf{w} + 2$, and imagine taking a single random search step starting at $\mathbf{w}^0 = [1 \;\; 0]^T$, a two-dimensional analog of the initialization we used for our one-dimensional quadratic. As illustrated in the bottom panel of Figure 2.10, we now have infinitely many unit directions to choose from, but only a fraction of which (less than 50 percent) provide descent. In other words, in two dimensions the chance of randomly selecting a descent direction drops with respect to its analogous value in one dimension. This decrease in the probability of randomly choosing a descent direction decreases *exponentially* as the input dimension N of this quadratic increases. Indeed one can compute that for a general N, the probability of choosing a descent direction at random starting at

$$\mathbf{w}^0 = \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}_{N\times 1} \tag{2.31}$$

for the quadratic function at hand is upper-bounded by $\frac{1}{2}\left(\frac{\sqrt{3}}{2}\right)^{N-1}$ (see Exercise 2.5). This means, for example, that when $N = 30$ the descent probability falls below 1 percent, making random search incredibly inefficient for minimizing even a simple quadratic function.

2.6 Coordinate Search and Descent

The coordinate search and descent algorithms are additional zero-order local methods that get around the inherent scaling issues of random local search by restricting the set of search directions to the coordinate axes of the input space. The concept is simple: random search was designed to minimize a function $g(w_1, w_2, ..., w_N)$ with respect to all of its parameters *simultaneously*. With coordinate-wise algorithms we attempt to minimize such a function with respect to one coordinate or parameter at a time (or, more generally, one subset of coordinates or parameters at a time) keeping all others fixed.

While this limits the diversity of descent directions that can be potentially discovered, and thus more steps are often required to determine approximate minima, these algorithms are far more scalable than random search.

2.6.1 Coordinate search

As illustrated in the left panel of Figure 2.11 for a prototypical $N = 2$ dimensional example, with *coordinate search* we seek out a descent direction by searching randomly among only the coordinate axes of the input space. This means in general that, for a function of input dimension N, we only look over the $2N$ directions from the set $\{\pm \mathbf{e}_n\}_{n=1}^{N}$, where $\mathbf{e}_n$ is a *standard basis vector* whose entries are set to zero except its nth entry which is set to 1.

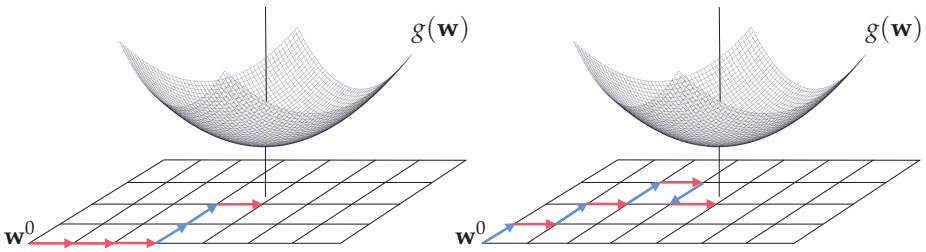

Figure 2.11 (left panel) With *coordinate search* we seek out descent directions only among the coordinates axes: at each step, colored alternately blue and red for better visualization, we try $2N = 4$ directions along the $N = 2$ coordinate axes, and pick the one resulting in the largest decrease in the function's evaluation. (right panel) With *coordinate descent* we (ideally) take a step immediately after examining the positive and negative directions along each coordinate.

It is this restricted set of directions we are searching over that distinguishes the coordinate search approach from the random search approach described in the previous section, where the set of directions at each step was made up of random directions. While the (lack of) diversity of the coordinate axes may limit the effectiveness of the possible descent directions it can encounter (and thus more steps are required to determine an approximate minimum), the restricted search makes coordinate search far more scalable than the random search method since at each step only $2N$ directions must be tested.

2.6.2 Coordinate descent

A slight twist on the coordinate search produces a much more effective algorithm at precisely the same computational cost, called *coordinate descent*. Instead of collecting each coordinate direction (along with its negative version), and then choosing a single best direction from this entire set, we can simply examine one coordinate direction (and its negative) at a time and step in this direction if it produces descent. This idea is illustrated in the right panel of Figure 2.11.

Whereas with coordinate search we evaluate the cost function $2N$ times (twice per coordinate) to produce a single step, this alternative approach takes the same number of function evaluations but potentially moves N steps in doing so. In other words, with coordinate descent we can, for the same cost as coordinate search, potentially minimize a function much faster. Indeed of all the zero-order methods detailed in this chapter, coordinate descent is by far the most practical.

Example 2.6 Coordinate search versus coordinate descent

In this example we compare the efficacy of coordinate search and the coordinate descent algorithms on the simple quadratic function

$$g(w_1, w_2) = 0.26\left(w_1^2 + w_2^2\right) - 0.48\, w_1 w_2. \tag{2.32}$$

In Figure 2.12 we compare 20 steps of coordinate search (left panel) and coordinate descent (right panel), using a diminishing steplength for both runs. Because coordinate descent effectively takes two steps for every single step taken by coordinate search, we get significantly closer to the function's minimum using the same number of function evaluations.

2.7 Conclusion

This chapter laid the groundwork for a wide range of fundamental ideas related to mathematical optimization (motivated in Section 1.4) that we will see repeat-

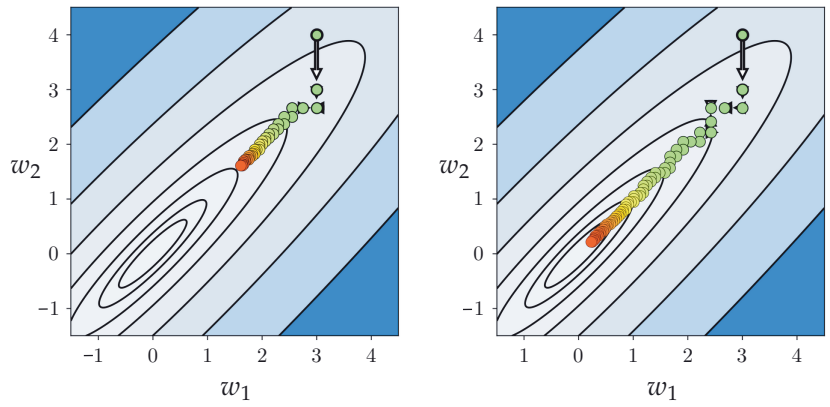

Figure 2.12 Figure associated with Example 2.6. A run of coordinate search (left panel) and coordinate descent (right panel) employed to minimize the same function. While both runs have the same computational cost, coordinate descent makes significantly greater progress. See text for further details.

edly not only in the next two chapters, but throughout the remainder of the text as well.

We began by introducing the concept of mathematical optimization, which is the mathematical/computational pursuit of a function's minima or maxima. Then in Section 2.2 we translated our informal definition of a function's minima and maxima into the language of mathematics, with this formal translation referred to as the *zero-order condition for optimality*. Leveraging this definition we then described global optimization methods (essentially the raw evaluation of a function over a fine grid of input points) in Section 2.3 which we saw – due to the curse of dimensionality – scales very poorly in terms of input dimension, and are thus not often very practical. Section 2.4 introduced the alternative to this limited framework – *local optimization* – which encompasses an enormous family of algorithms we discuss in the remainder of this chapter, as well as the two chapters that follow. Finally, in Sections 2.5 and 2.6 we described a number of examples of zero-order local optimization algorithms – including random search and coordinate search/descent. While the latter schemes can be very useful in particular applications, in general these zero-order local schemes are not as widely used in machine learning when compared to those we will see in the chapters to come, which leverage a function's first and/or second derivatives to more immediately formulate descent directions (instead of the raw search required by zero-order algorithms). However, the relative simplicity of zero-order schemes allowed us to flush out a range of critical concepts associated with local optimization – ideas that we will see echo repeatedly throughout the chapters that follow in a comparatively uncluttered setting – including the notions of descent directions, steplength/learning rates, diminishing steplength schemes, and cost function history plots.

2.8 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`

2.1 Minimizing a quadratic function and the curse of dimensionality
Consider the simple quadratic function

$$g(\mathbf{w}) = \mathbf{w}^T\mathbf{w} \tag{2.33}$$

whose minimum is always at the origin regardless of the input dimension N.

(a) Create a range of these quadratics for input dimension $N = 1$ to $N = 100$, sample the input space of each $P = 100$ times uniformly on the hypercube $[-1, 1] \times [-1, 1] \times \cdots \times [-1, 1]$ (this hypercube has N sides), and plot the *minimum* value attained for each quadratic against the input dimension N.

(b) Repeat part (a) using $P = 100$, $P = 1000$, and $P = 10,000$ samples, and plot all three curves in the same figure. What sort of trend can you see in this plot as N and P increase?

(c) Repeat parts (a) and (b), this time replacing uniformly chosen samples with randomly selected ones.

2.2 Implementing random search in `Python`
Implement the random search algorithm in `Python` and repeat the experiment discussed in Example 2.4.

2.3 Using random search to minimize a nonconvex function
Use your implementation of random search in Exercise 2.2 to minimize the function

$$g(w_1, w_2) = \tanh(4w_1 + 4w_2) + \max\left(0.4w_1^2, 1\right) + 1. \tag{2.34}$$

Take a maximum of eight steps and search through $P = 1000$ random directions at each step, with a steplength $\alpha = 1$ and an initial point $\mathbf{w}^0 = [2 \;\; 2]^T$.

2.4 Random search with diminishing steplength
In this exercise you will use random search and a diminishing steplength to minimize the *Rosenbrock function*

$$g(w_1, w_2) = 100\left(w_2 - w_1^2\right)^2 + (w_1 - 1)^2. \tag{2.35}$$

This function has a global minimum at the point $\mathbf{w}^\star = [1 \;\; 1]^T$ located in a very narrow and curved valley.

Make two runs of random search using $P = 1000$, the initial point $\mathbf{w}^0 = [-2 \;\; -2]^T$, and $K = 50$ steps. With the first run use a fixed steplength $\alpha = 1$, and – with the second run – a diminishing steplength as detailed in Section 2.5.4. Compare the two runs by either plotting the contour plot of the cost function (with each run plotted on top), or by constructing a cost function history plot.

2.5 Random descent probabilities

Consider the quadratic function $g(\mathbf{w}) = \mathbf{w}^T\mathbf{w} + 2$, which we aim to minimize using random search starting at $\mathbf{w}^0$ defined in Equation (2.31), with $\alpha = 1$ and $\left\|\mathbf{d}^0\right\|_2 = 1$.

(a) When $N = 2$, show that the *probability of descent* – i.e., the probability that $g(\mathbf{w}^0 + \alpha\mathbf{d}^0) < g(\mathbf{w}^0)$ for a randomly chosen unit direction $\mathbf{d}^0$ – is upper-bounded by $\frac{\sqrt{3}}{4}$. *Hint: see Figure 2.13.*

(b) Extend your argument in part (a) to find an upper-bound on the probability of descent for general N.

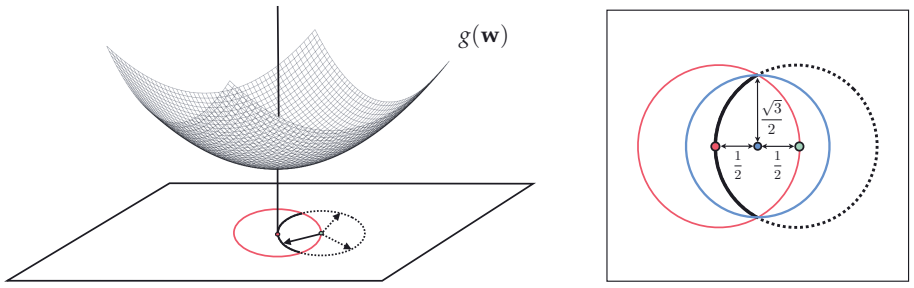

Figure 2.13 Figure associated with Exercise 2.5.

2.6 Revisiting the curse of dimensionality

In this exercise you will empirically confirm the curse of dimensionality problem described in Section 2.5.5 for the simple quadratic $g(\mathbf{w}) = \mathbf{w}^T\mathbf{w} + 2$.

Starting at the N-dimensional input point $\mathbf{w}^0$ defined in Equation (2.31), create P random unit directions $\{\mathbf{d}^p\}_{p=1}^P$ (where $\|\mathbf{d}^p\|_2 = 1$) and evaluate the point $\mathbf{w}^0 + \mathbf{d}^p$ via the quadratic for all p.

Next, produce a plot illustrating the portion of the sampled directions that

provide a decrease in function evaluation (i.e., the number of *descent directions* divided by P) against N (for $N = 1, 2, ..., 25$) and for three values of P: $P = 100$, $P = 1000$, and $P = 10,000$. Describe the trend you observe.

2.7 Pseudo-code for the coordinate search algorithm

Devise pseudo-code for the coordinate search algorithm described in Section 2.6.1.

2.8 Coordinate search applied to minimize a simple quadratic

Compare five steps of the random search algorithm (with $P = 1000$ random directions tested at each step) to seven steps of coordinate search, using the same starting point $\mathbf{w}^0 = [3 \;\; 4]^T$ and fixed steplength parameter value $\alpha = 1$ to minimize the quadratic function

$$g(w_1, w_2) = w_1^2 + w_2^2 + 2. \tag{2.36}$$

Plot this function along with the resulting runs of both algorithms, and describe any differences in behavior between the two runs.

2.9 Coordinate search with diminishing steplength

Implement the coordinate search algorithm detailed in Section 2.6.1 and use it to minimize the function

$$g(w_1, w_2) = 0.26\left(w_1^2 + w_2^2\right) - 0.48\, w_1 w_2 \tag{2.37}$$

using the diminishing steplength rule beginning at a random initial point. The global minimum of this function lies at the origin. Test your implementation by making sure it can reach a point significantly close (e.g., within 10^{-2}) to the origin from various random initializations.

2.10 Coordinate search versus coordinate descent

Implement the coordinate search and coordinate descent algorithms, and repeat the experiment discussed in Example 2.6.