3 First Order Methods

3.1 Introduction

In this Chapter we describe fundamental optimization algorithms that leverage the *first derivative* or *gradient* of a function (if unfamiliar with these concepts, the reader is encouraged to review Appendix B before proceeding). These techniques, collectively called *first order optimization methods*, are some of the most popular algorithms used to tackle machine learning problems today. We begin with a discussion of the *first order optimality condition* which (analogous to the zero order conditions outlined in Section 2.2) codifies how the first derivative(s) of a function characterize 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 condition for optimality

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 Section 7.8 if you are not familiar with the notion of Taylor series approximation). Notice in both instances that the tangent line/hyperplane is perfectly flat, indicating that the first derivative(s) is exactly zero at the function's minimum.

This sort of first order behavior is universal 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 derivative(s) or slope(s).

Because the derivative of a single-input function or the gradient of a multi-

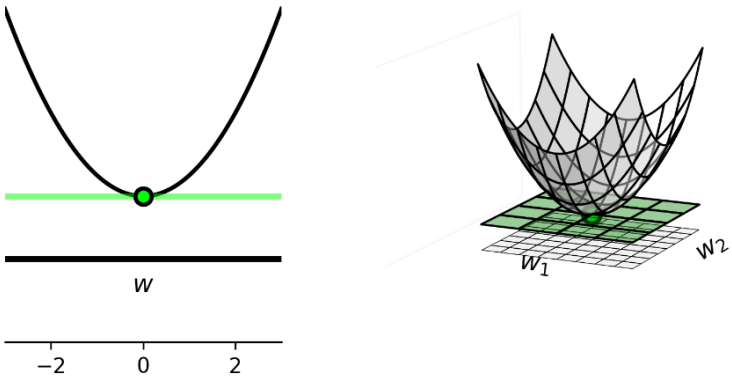

Figure 3.1 A figurative drawing of the gradient descent algorithm. The first order Taylor series approximation provides an excellent and easily computed descent direction at each step of this local method of optimization (here a number of approximations are shown in green). Employing these directions at each step the *gradient descent algorithm* can be used to properly minimize generic functions. Moreover, unlike the zero-order local search algorithms, gradient descent scales very well with input dimension since the descent direction of a hyperplane is much more easily computed in high dimensions than any search approach.

input function (see Section 7.1) 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 where

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

is a potential minimum. Analogously with functions taking in general N dimensional input, any N dimensional point $\mathbf{v}$ where *every* partial derivative of g is zero, that is

$$\begin{array}{c} \frac{\partial}{\partial w_1} g(\mathbf{v}) = 0 \\ \frac{\partial}{\partial w_2} g(\mathbf{v}) = 0 \\ \vdots \\ \frac{\partial}{\partial w_N} g(\mathbf{v}) = 0 \end{array} \tag{3.2}$$

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 (see Section 7.1) as

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

This very useful characterization of minimum points is referred to as the *first order optimality condition* (or the *first order condition* for short) because - at least in principle - it gives us a concrete alternative to seeking out a function's minimum points directly via a zero-order approach (i.e., by solving the first order system of equations). There are however two problems with the first order characterization of minima.

First off, with few exceptions (including some interesting examples we detail below), 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 one can write out on paper). 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. The first order condition also equally characterizes *maxima* and *saddle points* of *non-convex* functions as we see in a few simple examples below. Collectively minima, maxima, and saddle points are often referred to as *stationary* or *critical* points.

Example 3.1 **Finding points of zero derivative for single-input functions graphically**

In the top row of Figure 3.2 we plot the three functions listed below, along with their derivatives in the second row of the same Figure. On each function we mark the points on where its derivative is zero using a green dot (we likewise mark these points on each derivative itself), and show the tangent line / first order Taylor series corresponding to each such point in green as well.

$$\begin{aligned} g(w) &= \sin(2w) \\ g(w) &= w^3 \\ g(w) &= \sin(3w) + 0.1w^2 \end{aligned} \tag{3.4}$$

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. These consist of **i)** *local minima* or points that are the smallest with respect to their immediate neighbors, like the one around the input value $w = 2$ in the upper-right panel **ii)** *local and global maxima* or points that are the largest with respect to their immediate neighbors, like the one around the input value $w = -2$ in the right panel **iii)** *saddle points* like the one shown in the 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 (potentially nonlinear) equations, for which a wide range of algorithmic

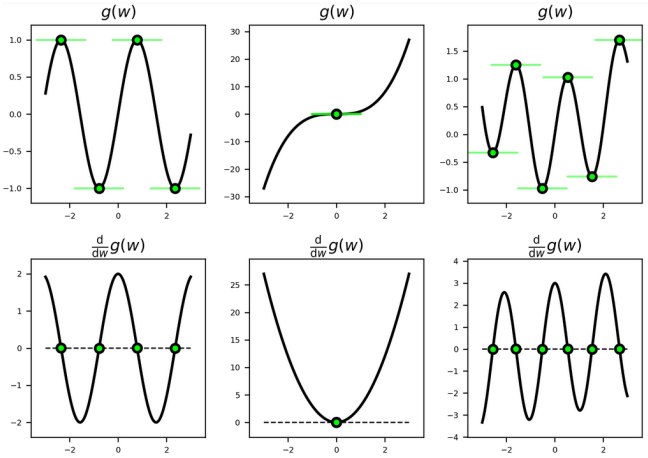

Figure 3.2 Figure for Example 3.1. From left to right the functions $g(w) = \sin(2w)$, w^3, and $\sin(3w) + 0.1w^2$ are shown along with their derivatives in the bottom row. See text for further details.

methods have been designed. The emphasis here on the word *algorithmic* here is key, as solving a system of 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 and the highly related *Rayleigh quotient*, which we discuss below. 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 Calculating stationary points of some single-input functions algebraically

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

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

- $g(w) = w^3$: (plotted in the top-middle panel of Figure 3.2) the first order condition gives $g'(v) = 3v^2 = 0$ which we can visually identify (from the middle-bottom panel of the same Figure) as a saddle point at $v = 0$.
- $g(w) = e^w$, the first order condition gives $g'(v) = e^v = 0$ which is only satisfied as v goes to $-\infty$, giving a minimum.
- $g(w) = \sin(w)$ the first order condition gives stationary points wherever

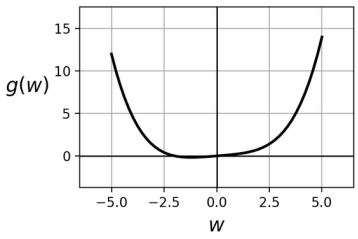

Figure 3.3 Figure associated with Example 3.3. See text for details.

$g'(v) = \cos(v) = 0$ 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.

- $g(w) = w^2$ for which the first order condition gives $g'(v) = 2cv + b = 0$ with a minimum at $v = \frac{-b}{2c}$.

Example 3.3 A simple looking function

As mentioned previously the vast majority of functions - or to be more precise the system of equations derived from functions - 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 very cumbersome to compute by hand.

Take the simple 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.

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 three possible solutions, but the one providing the minimum of the function $g(w)$ is

$$w = \frac{\sqrt[3]{\sqrt{2031} - 45}}{6^{\frac{2}{3}}} - \frac{1}{\sqrt[3]{6\left(\sqrt{2031} - 45\right)}} \tag{3.9}$$

which can be computed - after much toil - using centuries old tricks developed for just such problems.

Example 3.4 **Stationary points of a general multi-input quadratic function**

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 first derivative (gradient) 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 a some (rather important) cases the first order system can be solved *one equation at-a-time* the n^{th} 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 using we first initialize at an input $\mathbf{w}^0$, and begin by updating the first coordinate

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

for the optimal first weight $w_1^\star$. Note importantly in solving this equation for w_1 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 n^{th} 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 at fixed at their current values. We then update the n^{th} 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 k^{th} such sweep we update the n^{th} weight by solving the single equation

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

and update the n^{th} weight of $\mathbf{w}^{k+n-1}$, and so on.

Example 3.5 **Minimizing convex quadratic functions via coordinate descent**

Here we use coordinate descent to minimize the convex quadratic function

$$g(w_0, w_1) = w_0^2 + w_1^2 + 2 \tag{3.16}$$

whose minimum lies at the origin. We make the run initialized at $\mathbf{w} = \begin{bmatrix} 3 \\ 4 \end{bmatrix}$, 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 here is *linear* and so trivial to solve in closed form.

We then apply coordinate descent to minimize the convex quadratic

$$g(w_0, w_1) = 2w_0^2 + 2w_1^2 + 2w_0 w_1 + 20 \tag{3.17}$$

whose contour plot is shown in the right panel of Figure 3.4. Here it takes two full sweeps through the variables to find the global minimizer - which again lies at the origin.

Example 3.6 **Solving systems symmetric linear systems**

In Example 3.4 we saw that the first order system of a convex quadratic function is *symmetric linear* and takes the form

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

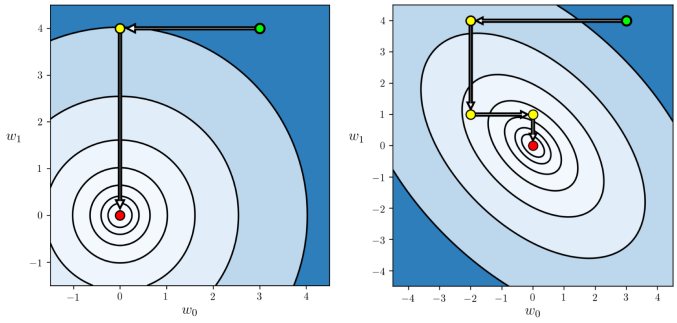

Figure 3.4 Figure associated with Example 3.5

where **a**, **b**, and the symmetric matrix **C** take the general form described in that Example. We can use the coordinate descent algorithm detailed here to solve this system, thereby minimizing the corresponding convex 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 (i.e., where the matrix **C** is positive semi-definite). The solution systems is commonplace e.g., at each and every step of Newton's method (as detailed in Chapter 4).

3.3 The geometric anatomy of lines and hyperplanes

In this Section we describe important characteristics of the *hyperplane* including the concept of the *direction of steepest ascent* and *steepest descent*. In addition we detail how to construct complicated hyperplanes from relatively simple parts. Along the way we will also see several important concepts - most notably the idea of the *direction of steepest ascent* arises naturally from the notion of slope. These concepts are fundamental to the notion of multi-input derivatives (the gradient - see Section 7.1.9), and thus by extension the gradient descent we discussed in Section 3.6.

3.3.1 Single input hyperplanes

The formula for a line

$$g(w) = a + bw \tag{3.19}$$

tells us - for specific choices of a and b - are the point at which it strikes or intersects the vertical axis (given by a) and the steepness or *slope* of that line (given by the coefficient b).

We can visualize the slope of a single line using a horizontal vector with

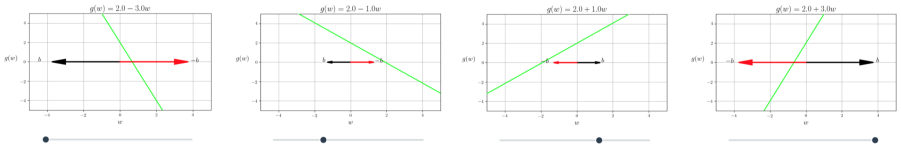

Figure 3.5 Four examples of a line where - from left to right - slope of the line changes from negative to positive. In each instance the slope of the line, its direction of steepest *ascent*, is visualized as a black vector along the input space. Likewise the negative slope, its direction of steepest *descent*, is shown as red vector along the input space of each example.

magnitude b, which we show in Figure 3.5 below in black for several examples. Viewed as a *direction* the slope of a line is often referred to as the *direction of steepest ascent* - since it tells us the direction we must travel on the line to increase its value the fastest. By the same logic the negative slope $-b$ provides the direction of steepest *descent* of the line, that is the direction in which it is decreasing. In Figure 3.5 we draw this value as a red vector along the input axis of each example.

In three dimensions - where we have two input variables w_1 and w_2 - we can form a similar equation using a single input. For example, the formula

$$g(w_1, w_2) = a + bw_1 \tag{3.20}$$

takes in the first input w_1 and treats it like a line, outputting $a + bw_1$. We can still say that this hyperplane - like the corresponding line - has a steepness or slope given by b. The only difference is that this steepness is now defined over a two dimensional input space. However in three-dimensions this is not just a line - its a line in w_1 stretched along the w_2 input dimension - forming a *hyperplane*.

We plot the two and three dimensional versions of the line $g(w_1, w_2) = 2 - w_1$ side by side in the top row of Figure 3.6.

Like the single input / one dimensional example, here we can visualize the *directions of steepest ascent and descent* of the hyperplane - given by this individual direction slope of the hyperplane - as a vector in the input space. Of course here the input space is two-dimensional, hence the vector has two entries. For example, with the hyperplane $g(w_1, w_2) = 2 - 2w_1$ the ascent direction is $(b_1, 0) = (-2, 0)$ and descent direction $-(b_1, 0) = (2, 0)$. In the bottom row of Figure 3.6 show several hyperplanes and these vectors - the ascent vector colored blue, the descent color in red - along a range of values for b_1 beginning with $b_1 = -2$ below.

We can of course define this single input hyperplane along any dimension

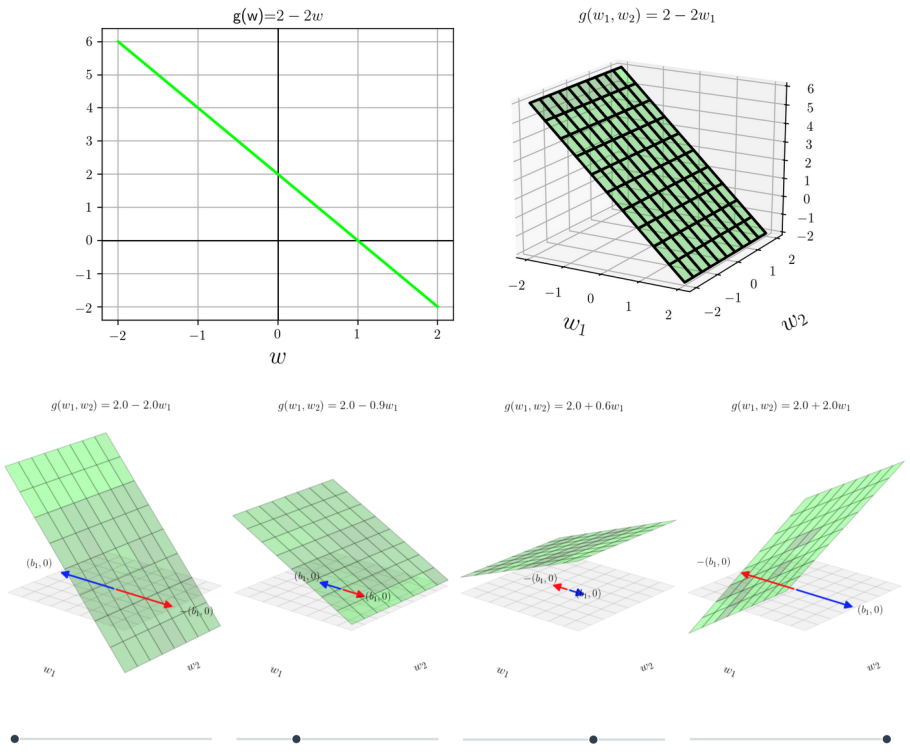

Figure 3.6 (top row) Two and three dimensional versions of the line $g(w_1, w_2) = 2 - w_1$. (bottom row) Change in the directions of steepest ascent and descent as a result of change in the individual direction slope of a 2D hyperplane.

we want. In general if we have N possible inputs $\mathbf{w} = [w_1,\ w_2,\ \cdots w_N]$ we can define it along the n^{th} dimension as $g(\mathbf{w}) = a + bw_n$.

3.3.2 Constructing general hyperplanes

To construct a general hyperplane in N dimensions we can create a set of N of the simple hyperplanes described above (each defined along a single input only) as

$$\begin{aligned} g_1(\mathbf{w}) &= a_1 + b_1 w_1 \\ g_2(\mathbf{w}) &= a_2 + b_2 w_2 \\ &\vdots \\ g_N(\mathbf{w}) &= a_N + b_N w_N \end{aligned} \tag{3.21}$$

and add them up. Summing up the hyperplanes above gives - collecting terms - the hyperplane

$$g(\mathbf{w}) = (a_1 + a_2 + \cdots a_N) + (b_1 w_1 + b_2 w_2 + \cdots b_N w_N). \tag{3.22}$$

We can write this formula more compactly using vector notation, denoting

$$a = \sum_{n=1}^{N} a_n \qquad \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_N \end{bmatrix}. \tag{3.23}$$

With this more compact notation we can write the sum simply as

$$g(\mathbf{w}) = a + \mathbf{w}^T\mathbf{b} \tag{3.24}$$

Whereas each slope and its negative provided the steepest ascent and descent direction for each simple hyperplane, the general N dimensional hyperplane produced by their sum has a direction of steepest ascent given by $\mathbf{b}$, which we can think of as a vector of slopes along each input dimension, and a steepest descent direction $-\mathbf{b}$.

For example, in Figure 3.7 we plot two single-input hyperlanes g_1, g_2, as well as their sum g_1+g_2 in the left, middle, and right panels respectively. The direction of steepest ascent in each simple dimension hyperplane is illustrated as a blue vector and descent vector in red (in the left two panels), while the direction of steepest ascent of the sum is shown in black in the right panel with the general hyperplane. Unlike the simple hyperplanes the ascent / descent directions of the general hyperplane can point in any direction in the input space, and are not limited to single input axes.

3.4 The geometric anatomy of first order Taylor series approximations

In this Section we carry over the interpretation of a slope of a line, or slopes from the previous Section, studying its impact on the scenario of the first order Taylor Series approximation where our hyperplane is defined by a functions derivative(s) (see Section 7.8). This geometric anatomy define the very essence of the extremely popular gradient descent algorithm, introduced in the Section following this one.

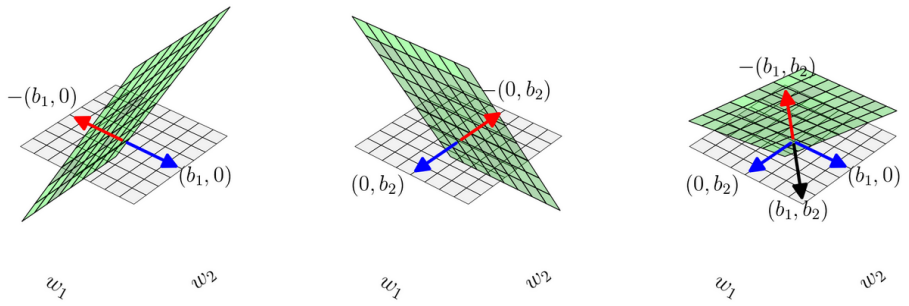

Figure 3.7 (left and middle panels) Two simple hyperplanes, each defined along a single input axis, with their steepest ascent and descent directions illustrated by blue and red vectors respectively in the input space. In each case these directions lie along a single input direction. (right panel) The hyperplane formed by their sum has steepest ascent / descent directions given by the sum of these directions from the simple hyperlanes, here colored black and red respectively, and is not restricted to a single input axis.

3.4.1 Derivatives and the direction of steepest ascent / descent

The derivative of a single-input function $g(w)$ defines a tangent line at each point its input domain - this is called its *first order Taylor series approximation*. At a given point w^0 this tangent line $h(w)$

$$h(w) = g(w^0) + \frac{\mathrm{d}}{\mathrm{d}w}g(w^0)(w - w^0). \tag{3.25}$$

has slope given by the derivative $\frac{\mathrm{d}}{\mathrm{d}w}g(w^0)$. In the context of the previous Section, this derivative provides the steepest *ascent* direction, and its negative the steepest *descent* direction (that is, the directions to travel in to increase or decrease the line as quickly as possible). Because this particular line is constructed to closely *approximate* its underlying function near the piont 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 near the point w^0.

Example 3.7 The derivative as a direction of ascent / descent I

In the top row of Figure 3.8 we visualize the ascent and descent directions, derived from the first order Taylor series approximation, over a set of four values for w^0 for the single-input quadratic $g(w) = 0.5w^2 + 1$. The derivative / steepest ascent direction is plotted as a black vector along the horizontal axis, while the negative derivative / steepest descent direction is similarly shown in red. At each point we can see that the steepest ascent / descent direction for the tangent line are also ascent / descent directions for the underlying function itself (particularly near each value of w^0.

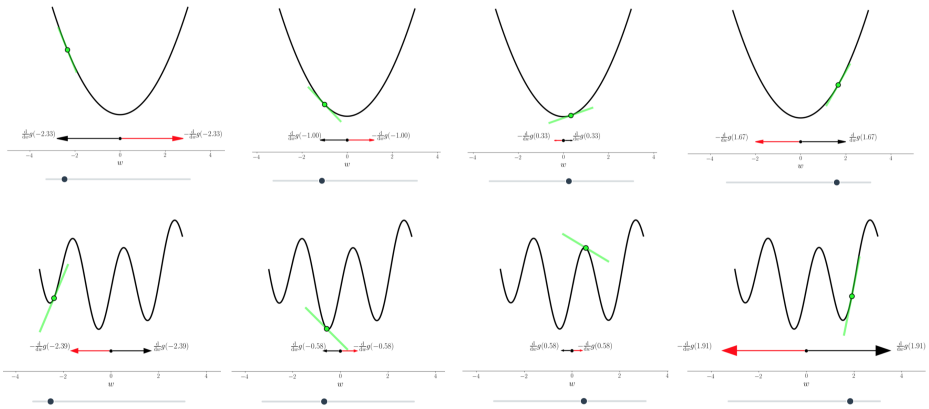

Figure 3.8 Figure associated with Examples 3.7 (top row) and 3.8 (bottom row). See text for details.

Example 3.8 The derivative as a direction of ascent / descent II

In the bottom row of Figure 3.8 we show a similar set of panels for the wavy sinusoidal function

$$g(w) = \sin(3w) + 0.1w^2 + 1.5. \tag{3.26}$$

showing the tangent line, and ascent / descent directions provided by the derivative, for four values of w^0. At each point these directions (provided by the tangent line) provide ascent/descent in the function *locally* near the points w^0 as well. In other words, if we were to follow the descent direction on any of the first order approximations shown here it always leads us to smaller values of on the function itself, provided we do not travel too far.

The same idea we have just explored for differentiable single-input functions holds for multi-input functions as well. With an N dimensional input function $g(\mathbf{w})$ instead of one derivative we have N *partial* derivatives, one in each input direction, stacked into a vector called the *gradient* (see Section 7.1.9)

$$\nabla g(\mathbf{w}) = \begin{bmatrix} \frac{\partial}{\partial w_1} g(\mathbf{w}) \\ \frac{\partial}{\partial w_2} g(\mathbf{w}) \\ \vdots \\ \frac{\partial}{\partial w_N} g(\mathbf{w}). \end{bmatrix} \tag{3.27}$$

Likewise the first order Taylor series is a tangent *hyperplane*, which at a point $\mathbf{w}^0$ has the (analogous to the single input case) formula

$$h(\mathbf{w}) = g(\mathbf{w}^0) + \nabla g(\mathbf{w}^0)^T(\mathbf{w} - \mathbf{w}^0). \tag{3.28}$$

In complete analogy to the single-input case, the steepest ascent / descent direction of this tangent hyperplane is provided by the gradient $\nabla g(\mathbf{w}^0)$ and its negative $-\nabla g(\mathbf{w}^0)$. These directions provide ascent / descent directions for the underlying function itself, locally around the point $\mathbf{w}^0$.

Example 3.9 **The gradient as a direction of ascent / descent III**

In Figure 3.9 we illustrate the quadratic function

$$g(w_1, w_2) = w_1^2 + w_2^2 + 6 \tag{3.29}$$

along with the steepest ascent and descent directions provided by its gradient at the point (top-left panel) $\mathbf{w}^0 = \begin{bmatrix} -1 \\ 1 \end{bmatrix}$ and (top-right panel) $\mathbf{w}^0 = \begin{bmatrix} 1 \\ -1 \end{bmatrix}$. With each we plot the coordinate-wise steepest ascent directions given by the two partial derivatives here in blue, with the gradient / ascent direction shown in black and the negative gradient / descent direction in red. The tangent hyperplane / first order Taylor series approximation is shown in green. Here we can see that indeed both ascent and descent directions of the hyperplane provide directions of ascent / descent in the underlying functions as well.

Example 3.10 **The gradient as a direction of ascent / descent IV**

Here we illustrate the ascent / descent directions - precisely as in the previous example - provided by the first order Taylor series approximation for the wavy sinusoidal function

$$g(w_1, w_2) = 5 + \sin(1.5w_1) - 2w_2. \tag{3.30}$$

In the bottom-left panel of Figure 3.9 we plot everything for the point $\mathbf{w}^0 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$, then in the bottom-right panel at the point $\mathbf{w}^0 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$. In both instances we can see how the ascent / descent directions of the tangent hyperplane provide steepest ascent / descent directions for the underlying function as well locally around the input point.

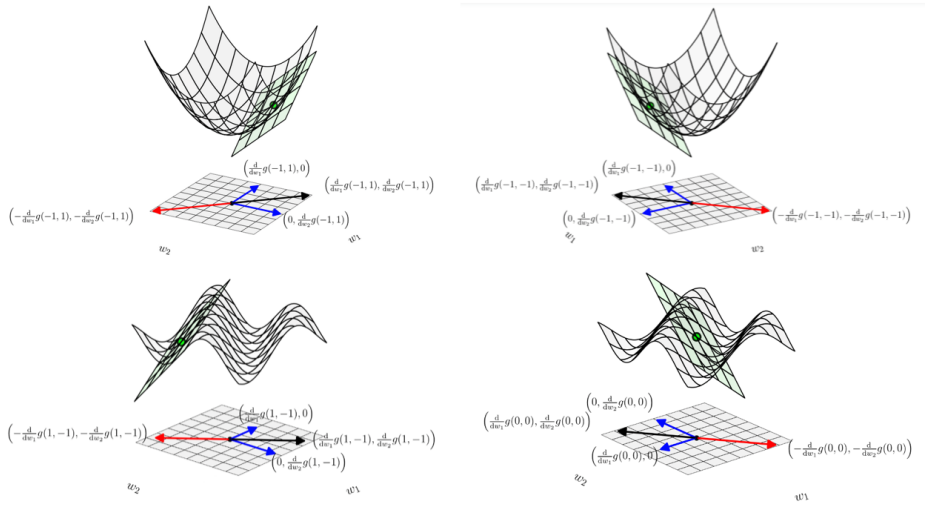

Figure 3.9 Figure associated with Examples 3.9 (top row) and 3.10 (bottom row). See text for details.

3.5 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, like 35 times 21 for instance, you can likely do the multiplication in your head using a combination of *multiplication properties* and *simple multiplication results* you learned in elementary school. In this case you may choose to use the distributive property of multiplication to decompose 35×21 as

$$(30 + 5) \times (20 + 1) = (30 \times 20) + (30 \times 1) + (5 \times 20) + (5 \times 1)$$

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 sinusoidals 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

$$\frac{\mathrm{d}}{\mathrm{d}w}g(w) = \frac{\mathrm{d}}{\mathrm{d}w}w^2\cos(w^2) = 2w\cos(w^2)$$

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, like the one that follows, yourself and *by hand*

$$g\left(w_1, w_2\right) = 2^{\sin\left(0.1w_1^2 + 0.5w_2^2\right)}\tanh\left(\cos\left(0.2w_1w_2\right)\right)\tanh\left(w_1 1 w_2^4 \tanh\left(w_1 0.2 w_2^2\right)\right)$$

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 Chapter 15. For Python users we strongly recommend using the open-source automatic differentiation library called **Autograd**. 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 Numpy operations. In Section 7.6 we provide a brief tutorial on how to get started with **Autograd**, as well as demonstrations of some of its core functionality.

3.6 Gradient descent

In the previous Section we saw how the negative gradient $-\nabla g\left(\mathbf{w}\right)$ of a function $g\left(\mathbf{w}\right)$ 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 2.5), 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

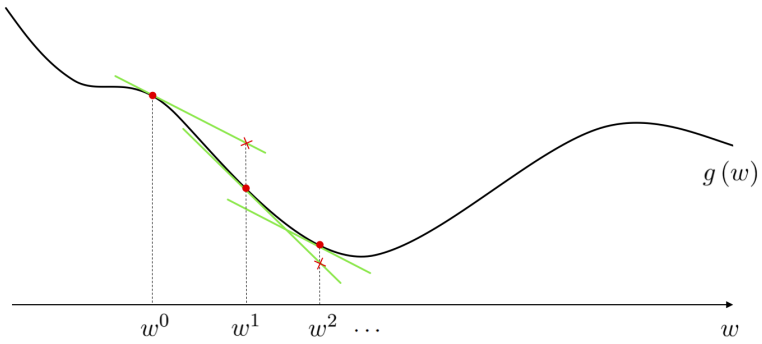

Figure 3.10 A figurative drawing of the gradient descent algorithm. Beginning at the initial point w^0 we make our first approximation to $g(w)$ at the point $(w^0, g(w^0))$ on the function (shown as a red dot) with the first order Taylor series approximation itself drawn in green. Moving in the negative 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 $w^2 = w^1 - \alpha \frac{\mathrm{d}}{\mathrm{d}w} g\left(w^1\right)$, and so forth.

$$\mathbf{w}^k = \mathbf{w}^{k-1} - \alpha \nabla g\left(\mathbf{w}^{k-1}\right). \tag{3.31}$$

It seems intuitive, at least at the outset, that because each and every direction is guaranteed to be one of descent[1] taking such steps could lead us to a point near a local minimizer of the target function. The rather simple update step in Equation (3.31) 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 3.10 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.

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 the remainder of this Chapter, is arguably the most popular local optimization algorithm used in machine learning today. This is largely due to 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 2.5 through 2.7). In other

[1] provided we set α appropriately, as we must always do when using any local optimization method.

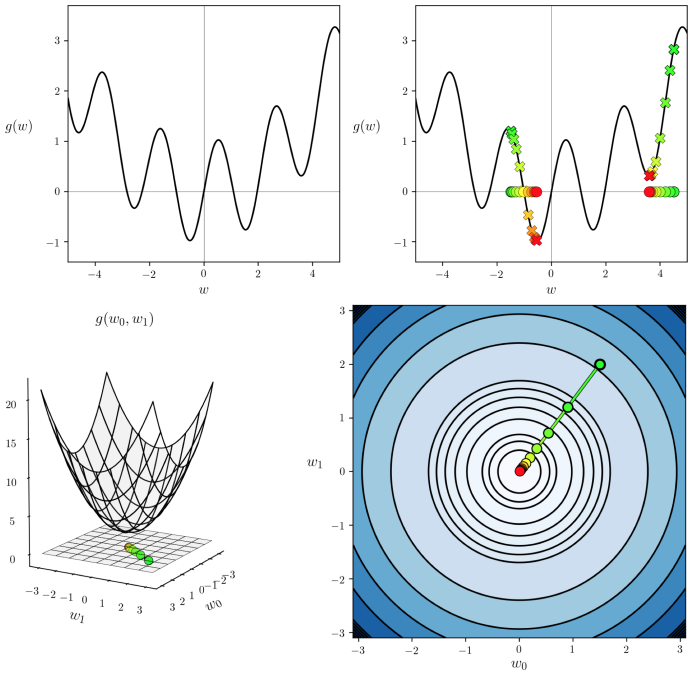

Figure 3.11 (top row) Figure associated with Example 3.11. (bottom row) Figure associated with Example 3.12. See text for details.

words, the fact that the negative gradient direction provides a descent direction for the function locally, combined with the fact that gradients are often cheap to compute makes gradient descent a superb local optimization method.

Example 3.11 **Minimizing a non-convex single-input function using gradient descent**

To find the global minimum of a general non-convex 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 non-convex function

$$g(w) = \sin(3w) + 0.1w^2 \tag{3.32}$$

illustrated in the top-left panel of Figure 3.11. The same function was minimized in Example 2.4 using random search. Here we initialize two runs of gradient descent, one at $w^0 = 4.5$ and another at $w^0 = -1.5$, using a fixed steplength of $\alpha = 0.05$ for both runs. As can be seen by the results in the top-right panel, depending on where we initialize we may end up near a local or global minimizer of the function.

Example 3.12 **Minimizing a convex multi-input function**

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

$$g(w_1, w_2) = w_1^2 + w_2^2 + 2 \tag{3.33}$$

previously used in Example 2.3. We fix the steplength parameter at $\alpha = 0.1$ for all 10 steps of the algorithm. In the bottom row of Figure 3.11 we illustrate the path taken by this run of gradient descent in the input space of the function, 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.

3.6.1 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 2.6. 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 $10^{-\gamma}$ where γ is an integer and ii) using a diminishing steplength like $\alpha = \frac{1}{k}$ where at the k^{th} step of a run.

In both instances our aim in choosing a particular value for the steplength / learning rate 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 3.13 **A fixed steplength selection**

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 3.12 using the simple single-input quadratic $g(w) = w^2$. Here we show the result of taking five gradient descent steps using four different fixed steplength 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

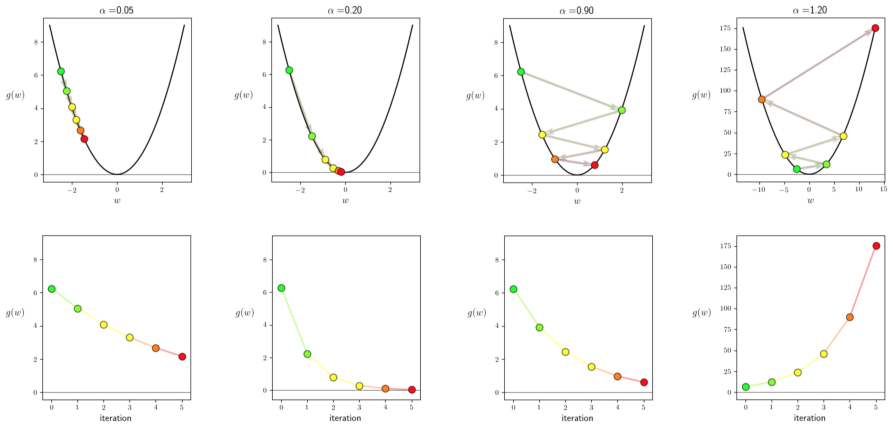

Figure 3.12 Figure associated with Example 3.13. See text for details.

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, whose numerical value is printed above each illustration. In the beginning the steplength is extremely small - so small that we do not descend very much at all. On the other end of the spectrum however, when we set value of α too large the algorithm ascends in the function (ultimately *diverging*).

In the bottom row of the Figure we show what a *cost function plot* we show the cost function plot corresponding to each run of gradient descent plotted on the function itself in the top row of the Figure. We also color these points from green (start of run) to red (end of run).

Example 3.14 Comparing fixed and diminishing steplengths

In Figure 3.13 we illustrate the comparison of a fixed steplength scheme and a the diminishing steplength rule to minimize the function

$$g(w) = |w|\,. \tag{3.34}$$

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 $w^0 = 2$, the first with a fixed steplength rule of $\alpha = 0.5$ (whose evaluation history is 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. This is because the derivative of this function (defined everywhere but at $w = 0$) takes the form

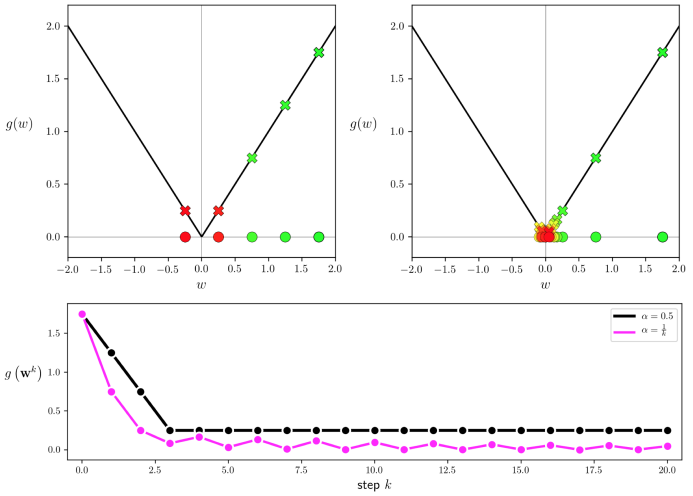

Figure 3.13 Figure associated with Example 3.14. See text for details.

$$\frac{\mathrm{d}}{\mathrm{d}w}g(w) = \begin{cases} +1 & \text{if } w > 0 \\ -1 & \text{if } w < 0. \end{cases} \tag{3.35}$$

which makes the use of any fixed steplength scheme problematic for gradient descent here 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.

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

Remember that in practice in machine learning - since we regularly deal with cost functions that take in far too many inputs / are far to high dimensional to visualize - we use the *cost function history plot* (first described in Section 2.6.2) 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 find the lowest function value possible - which may mean 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.

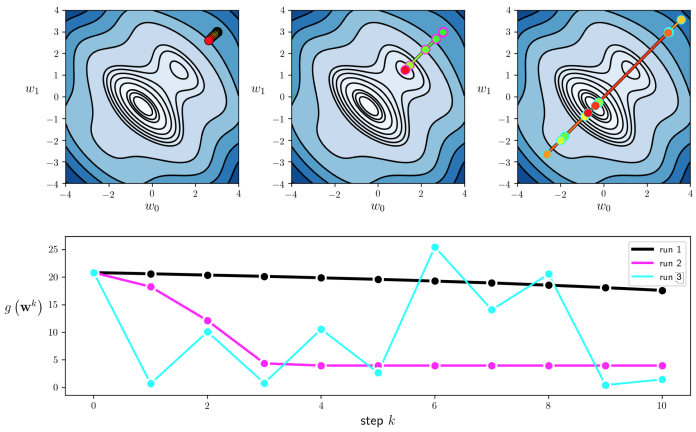

Figure 3.14 Figure associated with Example 3.15

Example 3.15 **Steplength selection**

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

$$g(\mathbf{w}) = w_0^2 + w_1^2 + 2\sin\left(1.5\left(w_0 + w_1\right)\right)^2 + 2 \tag{3.36}$$

whose contour plot is shown in Figure 3.14. You can see a *local minimum* around the point $\begin{bmatrix} 1.5 \\ 1.5 \end{bmatrix}$, and a global minimum near $\begin{bmatrix} -0.5 \\ -0.5 \end{bmatrix}$. All three runs start at the same initial point $\mathbf{w}^0 = \begin{bmatrix} 3 \\ 3 \end{bmatrix}$ and take 10 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) $\alpha = 10^{-1}$, and the third run (shown in the top-right panel) $\alpha = 10^0$.

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 (we can see this in the top-left panel as well), the second choice lead 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 on its first step! So while this run used the largest steplegnth $\alpha = 10^0$, clearly leading to oscillatory (and perhaps - eventually - divergent behavior) - it does indeed find the lowest point out of all three runs performed.

3.6.3 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., $-\nabla g\left(\mathbf{w}^k\right) \approx \mathbf{0}_{N\times 1}$. 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 ensure e.g., that the magnitude of the gradient $\left\|\nabla g\left(\mathbf{w}^{k-1}\right)\right\|_2$ is sufficiently small. However in practice, as with most local optimization schemes, the most practical way to halt gradient descent is to simply run the algorithm for a fixed number of maximum iterations.

What is good number for a 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 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.

3.6.4 Python implementation

Here we provide a basic `Python` implementation of the gradient descent algorithm. There are a number of practical variations one can use in practice - like e.g., different halting conditions other than a maximum number of steps, or what values are returned by the method - which we touch on below as well. Note that this implementation is based off of the general local method pseudo-code described in Section 2.5.4. This implementation requires uses the same general inputs described there (i.e., the function to minimize `g`, a steplength `alpha`, a maximum number of iterations `max_its`, and an initial point `w`) that is typically chosen at random. The computation of the gradient function in line 14, our method of determining the descent direction at each step, employs the open source `Python` Automatic Differentiation library `autograd` (detailed in Sections 3.5 and 7.6 although one can easily replace this line any other method for computing the gradient function. The output of this implementation is a history of the weights and corresponding cost function values at each step of the gradient descent algorithm that can be used to plot a cost function history plot.

```
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     # compute gradient module using autograd
7     gradient = grad(g)
```

```
    # run the gradient descent loop
    weight_history = [w] # weight history container
    cost_history = [g(w)] # cost function history container
    for k in range(max_its):
        # evaluate the gradient
        grad_eval = gradient(w)

        # take gradient descent step
        w = w - alpha*grad_eval

        # record weight and cost
        weight_history.append(w)
        cost_history.append(g(w))
    return weight_history,cost_history
```

Given the input to g is N dimensional a general random initialization - the kind that is often used - can be written as shown below. Here `numpy` function `random.randn` produces samples from a standard normal distribution with mean zero and unit standard deviation. It is also common to scale such initializations by small constants like e.g., 0.1.

```
# a common initialization scheme - a random point
w = np.random.randn(N,1)
```

3.7 Two issues with the negative gradient as a descent direction

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. However the negative gradient is not without its weaknesses as a descent direction, and in this Section we outline two significant problems with it that can arise in practice.

Like any *vector* the negative gradient always consists fundamentally of a *direction* and a *magnitude* (as illustrated in Figure 3.15). 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.

The *direction* of the negative gradient can *rapidly oscillate* or *zig-zag* during a run of gradient descent, often producing *zig-zagging* steps that take considerable time to reach a near minimum point. The *magnitude* of the negative gradient can

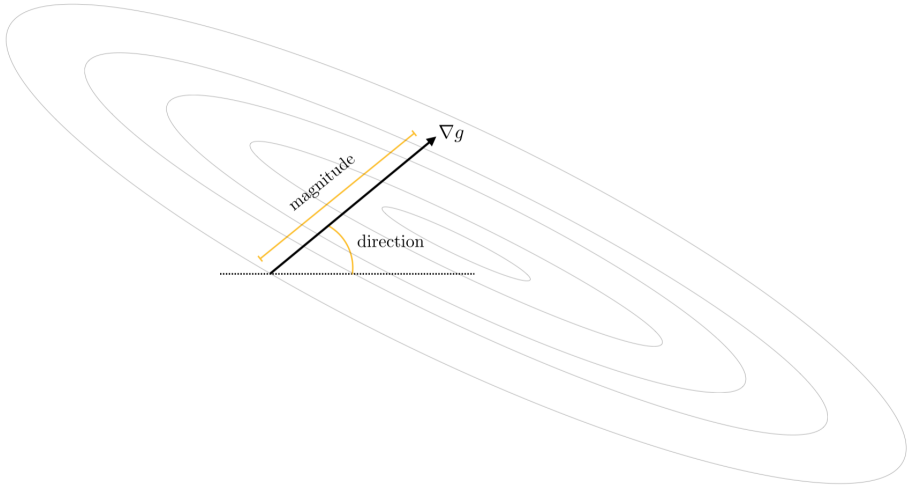

Figure 3.15 The gradient vector of any arbitrary function at any point consists of a *magnitude* and a *direction*.

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 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.

3.7.1 The (negative) gradient direction

A fundamental property of the (negative) gradient direction is that it always points perpendicular the contours of a function. This statement is universally true - and holds for *any* function and at *all* of its inputs.

We illustrate this fact via several examples below, where we show the *contour plot* of several functions along with the negative gradient direction computed and drawn at several points. What we will see is that a gradient ascent / descent direction at an input $\mathbf{w}^\star$ is always perpendicular to the contour $g(\mathbf{w}^\star) = c$. This statement can be rigorously (mathematically) proven to be true as well.[2]

[2] If we suppose $g(\mathbf{w})$ is a differentiable function and $\mathbf{a}$ is some input point, then $\mathbf{a}$ lies on the contour defined by all those points where $g(\mathbf{w}) = g(\mathbf{a}) = c$ for some constant c. If we take another point from this contour $\mathbf{b}$ very close to $\mathbf{a}$ then the vector $\mathbf{a} - \mathbf{b}$ is essentially perpendicular to the gradient $\nabla g(\mathbf{a})$ since $\nabla g(\mathbf{a})^T(\mathbf{a} - \mathbf{b}) = 0$ essentially defines the line in the input space whose normal vector is precisely $\nabla g(\mathbf{a})$. So indeed both the ascent and descent directions defined by the gradient (i.e., the positive and negative gradient directions) of g at $\mathbf{a}$ are perpendicular to the contour there. And since $\mathbf{a}$ was any arbitrary input of g, the same argument holds for each of its inputs.

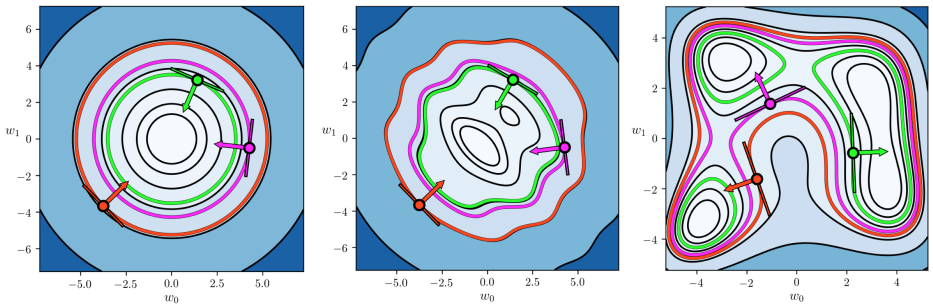

Figure 3.16 Figure associated with Example 3.16. See text for details.

Example 3.16 **The negative gradient direction**

In Figure 3.16 we show the contour plot of (left panel) $g(\mathbf{w}) = w_0^2 + w_1^2 + 2$, (middle panel) $g(\mathbf{w}) = w_0^2 + w_1^2 + 2\sin(1.5(w_0 + w_1))^2 + 2$, and (right panel) $g(\mathbf{w}) = \left(w_0^2 + w_1 - 11\right)^2 + \left(w_0 + w_1^2 - 6\right)^2$.[3] 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 lies 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). Since the gradient direction in each instance is the normal vector for this tangent line they can each be written as

$$\nabla g(\mathbf{w})^T (\mathbf{w} - \mathbf{v}) = 0. \tag{3.37}$$

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.

3.7.2 The 'zig-zagging' behavior of gradient descent

In practice the fact that the negative gradient *always points perpendicular to the contour of a function* can - depending on the function being minimized - make the

[3] With each contour plot is colored *blue* - with darker regions indicating where the function takes on larger values, and lighter regions where it takes on lower values.

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 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.

Example 3.17 **Zig-zagging behavior of gradient descent**

We illustrate the zig-zag behavior of gradient descent with three $N = 2$ dimensional quadratic functions that take the general form we have seen previously $g(\mathbf{w}) = a + \mathbf{b}^T\mathbf{w} + \mathbf{w}^T\mathbf{C}\mathbf{w}$. In each case the constants a and $\mathbf{b}$ are set to zero, and the matrix $\mathbf{C}$ set so each quadratic gets progressively omre narrow as $\mathbf{C} = \begin{bmatrix} 0.5 & 0 \\ 0 & 12 \end{bmatrix}$ (shown in the top panel below), $\mathbf{C} = \begin{bmatrix} 0.1 & 0 \\ 0 & 12 \end{bmatrix}$, and $\mathbf{C} = \begin{bmatrix} 0.01 & 0 \\ 0 & 12 \end{bmatrix}$ respectively. 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 3.17 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 are almost completely parallel to each other near our initialization (an example of a *long narrow valley*).

We then make a run of 25 gradient descent steps to minimize each, and with each we use the same initialization at $\mathbf{w}^0 = \begin{bmatrix} 10 \\ 1 \end{bmatrix}$ and steplength / learning rate 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 is reached), with arrows connecting each step to the one that follows. 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 perpendicular to the contours of the function (this can be especially seen in the third case), and in very narrow functions these contours become almost parallel. This zig-zagging clearly slows down the progress of the algorithm as it aims to find the mininum of each quadratic (which again in all three cases is located at the origin where the quadratics take on a value of zero).

It is the true that we can ameilorate this zig-zagging behavior by *reducing the steplength value*. However this does not solve the underlying problem that zig-zagging produces - which is slow convergence.

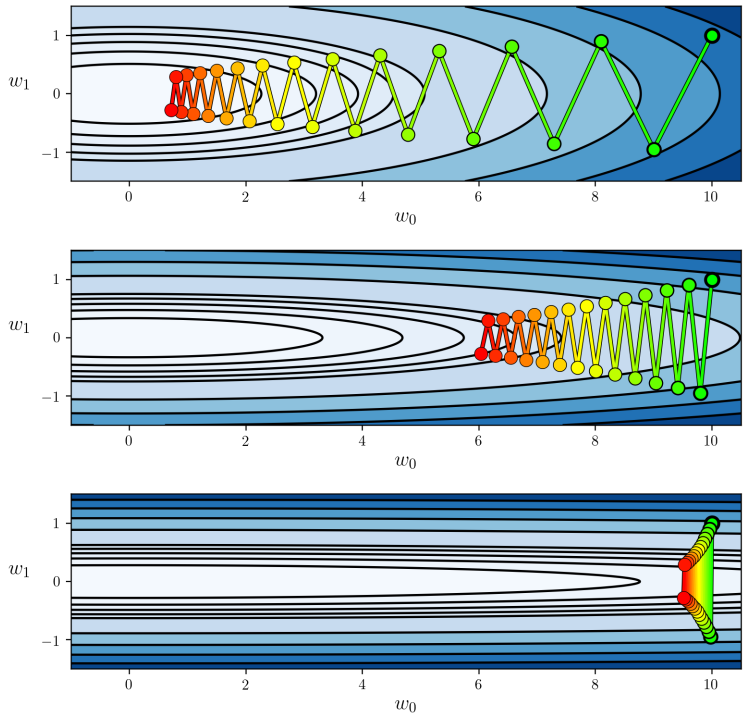

Figure 3.17 Figure associated with Example 3.17. See text for details.

3.7.3 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 the Sections 2.6 and 2.7 (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 / learning rate value α. Indeed we can easily compute that the general distance traveled by a gradient descent step as

$$\left\|\mathbf{w}^k - \mathbf{w}^{k-1}\right\|_2 = \left\|\left(\mathbf{w}^{k-1} - \alpha\nabla g\left(\mathbf{w}^{k-1}\right)\right) - \mathbf{w}^{k-1}\right\|_2 = \alpha\left\|\nabla g\left(\mathbf{w}^{k-1}\right)\right\|_2. \tag{3.38}$$

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 often gradient descent 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 non-convex functions.

Example 3.18 **Slow-crawling behavior near the minimum of a function**

In the top-left panel of Figure 3.18 we plot the function

$$g(w) = w^4 + 0.1 \tag{3.39}$$

whose minimum is at the origin $w = 0$, which we will minimize via gradient descent using a steplength parameter $\alpha = 10^{-1}$ and 10 steps to exaggerate this behavior. We show the results of this run in the top-right panel of the Figure (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 of this function. Both of these behaviors are quite natural, since the magnitude of the gradient is large far from the global minimum and vanishes near it.

Example 3.19 **Slow-crawling behavior of gradient descent near saddle points**

In the bottom row of Figure 3.18 we illustrate the crawling issue of gradient descent near *saddle points* using the non-convex function (illustrated in the bottom-left panel of the Figure)

$$g(w) = \text{maximum}(0, (3w - 2.3)^3 + 1)^2 + \text{maximum}(0, (-3w + 0.7)^3 + 1)^2. \tag{3.40}$$

This function has a minimum at $w = \frac{1}{2}$ and saddle points at $w = \frac{7}{30}$ and $w = \frac{23}{30}$.

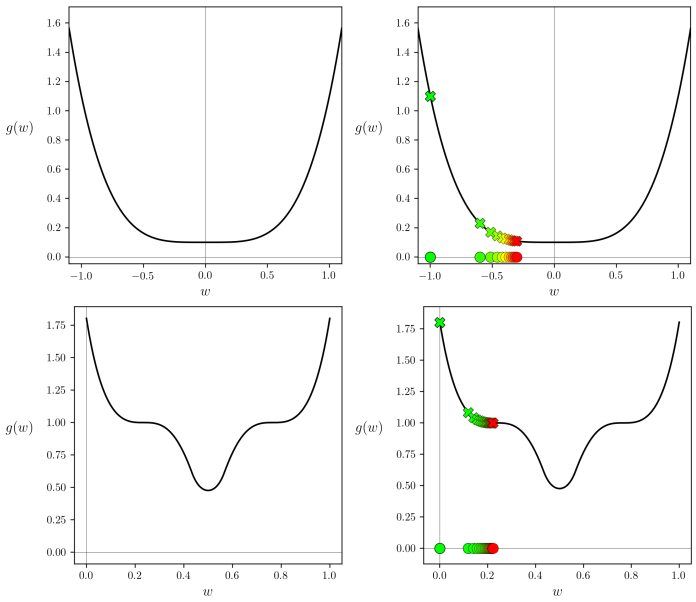

Figure 3.18 Figure associated with Example 3.18 (top row) and Example 3.19 (bottom row). (top/bottom left panels) Simple test function. (top/bottom right panels) Test functions with the resulting steps and function evaluations provided by a run of gradient descent. See text for further details.

We make a run of gradient descent on this function using 50 steps with $\alpha = 10^{-2}$, initialized such that it approaches one of these saddle points and so slows to a halt (as illustrated in the bottom-right panel of the Figure).

Examining the bottom-right panel of the Figure we can see how the gradient descent steps halt near the left-most 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.

3.8 Momentum accelerated gradient descent

In the previous Section we discussed a fundamental issue associated with the *direction* of the negative gradient: it can (depending on the function being minimized) oscillate rapidly, leading to zig-zagging gradient descent steps that slow down minimization. In this Section we describe a popular enhancement to the standard gradient descent step, called *momentum acceleration*, that is specifically designed to ameliorate this issue. The core of this idea comes from the field of *time series analysis*, and in particular is a tool for smoothing time series data known as the *exponential average*. Here we first introduce the exponential average and then detail how it can be integrated into the standard gradient descent step in order to help ameliorate some of this undesirable zig-zagging behavior (when it occurs) and consequently speed up gradient descent.

Figure 3.19 An example of a time series data, that is data ordered in time. See text for further details.

3.8.1 The exponential average

In Figure 3.19 we show an example of a time series data. This particular example shows a real snippet of the price of a financial stock measured at 500 consecutive points in time. In general time series data consists of a sequence of K ordered points w^1, w^2, ..., w^K, meaning that the point w^1 comes before (that is, it is created and/or collected before) w^2, the point w^2 before w^3, and so on. For example, we generate a (potentially multi-dimensional) time series of points whenever we run a local optimization scheme with steps $\mathbf{w}^k = \mathbf{w}^{k-1} + \alpha \mathbf{d}^{k-1}$ since it produces the sequence of ordered points $\mathbf{w}^1$, $\mathbf{w}^2$, ..., $\mathbf{w}^K$.

Because the raw values of a time series often oscillate or *zig-zag*, it is common practice to *smooth* them (in order to remove these zig-zagging motions) for better visualization or prior to further analysis. The *exponential average* is one of the most popular such smoothing techniques for time series, and is used in virtually every application area in which this sort of data arises. We show the result of smoothing the data shown in Figure 3.19 in Figure 3.20 wherein the resulting exponential average shown as a pink curve.

Before we see how the exponential average is computed, it is first helpful to see how to compute a *running average* of K input points w^1, w^2, ..., w^K, that is the average of the first two points, the average of the first three points, and so forth. Denoting the average of the first k points as h^k we can write

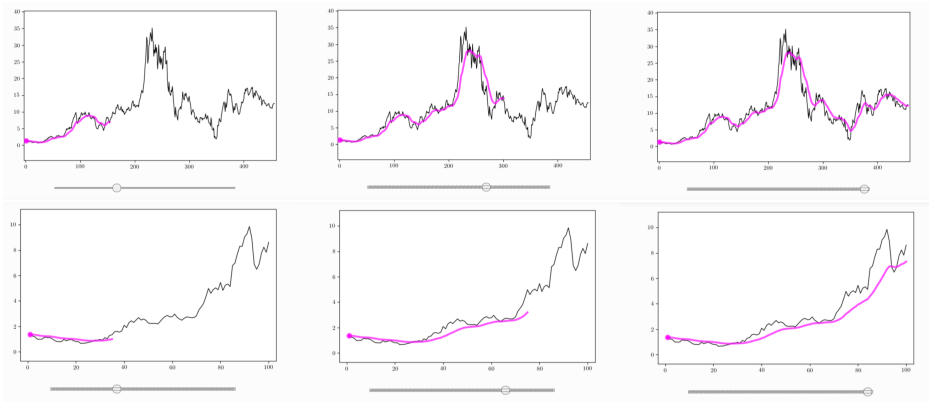

Figure 3.20 An exponential average of the time series data shown in Figure 3.19. (top row) An exponential average (in pink) of the first 150 (left panel), 350 (middle panel), and complete time series (right panel). (bottom row) An exponential average of just the first 50 points of the time series shown as a progression of three stages, similar to that shown for the complete data.

$$\begin{aligned} h^1 &= w^1 \\ h^2 &= \tfrac{w^1+w^2}{2} \\ h^3 &= \tfrac{w^1+w^2+w^3}{3} \\ &\vdots \\ h^K &= \tfrac{w^1+w^2+w^3+w^4+\cdots+w^K}{K} \end{aligned} \tag{3.41}$$

Notice how at each step h^k essentially *summarizes* the input points w^1 through w^k via the simplest statistic: their sample mean. The way the running average is written in Equation (3.41), we need access to every raw point w^1 through w^k in order to compute the k^{th} running average h^k. Alternatively we can write this running average by expressing h^k for $k > 1$ in a recursive manner involving only its preceding running average h^{k-1} and current time series value w^k, as

$$h^k = \frac{k-1}{k}h^{t-1} + \frac{1}{k}w^k. \tag{3.42}$$

From a computational perspective, the recursive way of defining the running average is far more efficient since at the k^{th} step we only need to store and deal with two values as opposed to k of them.

The *exponential average* is a simple twist on the running average formula. Notice, at every step in Equation (3.8.1) that the coefficients on h^{k-1} and w^k *always* sum to 1: that is $\frac{k-1}{k} + \frac{1}{k} = 1$. As k grows larger both coefficients change: the coefficient on h^{k-1} gets closer to 0 while the one on w^k gets closer to 0. With the exponential average we *freeze* these coefficients. That is, we replace the

coefficient on h^{k-1} with a constant value $\beta \in [0, 1]$, and the coefficient on w^k with $1 - \beta$, giving a similar recursive formula for the exponential average, as

$$h^k = \beta\, h^{k-1} + (1 - \beta)\, w^k. \tag{3.43}$$

Clearly the parameter β here controls a trade-off: the smaller we set β the more our exponential average approximates the raw (zig-zagging) time series itself, while the larger we set it the more each subsequent average looks like its predecessor (resulting in a smoother curve). Regardless of how we set β each h^k in an exponential average can still be thought of as a summary for w^k and all time series points that precede it.

Why is this slightly adjusted version of the running average called an *exponential* average? Because if we roll back the update shown in Equation (3.8.1) to express h^k only in terms of preceding time series elements, as we did for running average in Equation (3.41), an exponential (or power) pattern in the coefficients will emerge (see Chapter's exercises).

Also note that in deriving the exponential average we assumed our time series data was *one dimensional*, that is each raw point w^k is a scalar. However this idea holds regardless of the input dimension. We can likewise define the exponential average of a time series of general N dimensional points $\mathbf{w}^1$, $\mathbf{w}^2$, ..., $\mathbf{w}^K$ by initializing $\mathbf{h}^1 = \mathbf{w}^1$, and then for $k > 1$ building $\mathbf{h}^k$ as

$$\mathbf{h}^k = \beta\, \mathbf{h}^{k-1} + (1 - \beta)\, \mathbf{w}^k. \tag{3.44}$$

Here the exponential average $\mathbf{h}^k$ at step k is also N dimensional.

3.8.2 Ameliorating the zig-zag behavior of gradient descent

As mentioned above a sequence of gradient descent steps can be thought of as a *time series*. But - in addition - so too can the sequence of negative gradient directions generated by a run of gradient descent. Indeed if we take K steps of a gradient descent run using the form above we do create a time series of ordered *gradient descent steps* $\mathbf{w}^1$, $\mathbf{w}^1$, ..., $\mathbf{w}^K$ and descent directions $-\nabla g\left(\mathbf{w}^0\right)$, $-\nabla g\left(\mathbf{w}^1\right)$, ..., $-\nabla g\left(\mathbf{w}^{K-1}\right)$.

To attempt to ameliorate some of the zig-zagging behavior of our gradient descent steps $\mathbf{w}^1$, $\mathbf{w}^1$, ..., $\mathbf{w}^K$ - as detailed in Section 3.7.2 - we could compute their *exponential average*. However we do not want to smooth the gradient descent steps *after* they have been created - as the 'damage is already done' in the sense that the zig-zagging has already slowed the progress of a gradient descent run. Instead what we want is to smooth the steps *as they are created*, so that our algorithm makes more progress in minimization.

How do we smooth the steps as the are created? Remember from Section 3.7.2

that the root cause of zig-zagging gradient descent steps zig-zag is the oscillating nature of the (negative) gradient directions themselves. In other words, if the descent directions $-\nabla g\left(\mathbf{w}^0\right)$, $-\nabla g\left(\mathbf{w}^1\right), ..., -\nabla g\left(\mathbf{w}^{K-1}\right)$ zig-zag, so to will the gradient descent steps themselves. Therefore it seems reasonable to suppose that if we smooth out these directions themselves, as they are created during a run of gradient descent, we can as a consequence produce gradient descent steps that do not zig-zag as much and therefore make more progress in minimization.

To do this we first initialize $\mathbf{d}^0 = \nabla g\left(\mathbf{w}^0\right)$ and then for $k - 1 > 0$ the $(k-1)^{th}$ exponentially averaged descent direction $\mathbf{d}^{k-1}$ (using the formula in equation 3.8.1) takes the form

$$\mathbf{d}^{k-1} = \beta\, \mathbf{d}^{k-2} - (1-\beta)\, \nabla g\left(\mathbf{w}^{k-1}\right). \tag{3.45}$$

We can then use this descent direction in our generic local optimization framework to take a step as

$$\mathbf{w}^k = \mathbf{w}^{k-1} + \alpha\, \mathbf{d}^{k-1}. \tag{3.46}$$

Together this exponential averaging adds only a *single extra* step to our basic gradient descent scheme. Together this forms a *momentum accelerated gradient descent step* of the form

$$\begin{aligned} \mathbf{d}^{k-1} &= \beta\, \mathbf{d}^{k-2} - (1-\beta)\, \nabla g\left(\mathbf{w}^{k-1}\right) \\ \mathbf{w}^k &= \mathbf{w}^{k-1} + \alpha\, \mathbf{d}^{k-1}. \end{aligned} \tag{3.47}$$

The term 'momentum' here refers to the new exponentially averaged descent direction $\mathbf{d}^{k-1}$. Notice that by the very definition of the exponential gradient given in the first part of this Section, that this descent direction $\mathbf{d}^{k-1}$ is a function (an exponential average) of *every negative gradient which precedes it* $-\nabla g\left(\mathbf{w}^0\right)$, $-\nabla g\left(\mathbf{w}^1\right), ..., -\nabla g\left(\mathbf{w}^{k-1}\right)$. Hence $\mathbf{d}^{k-1}$ captures the average or 'momentum' of the directions preceding it.

As with any exponential average the choice of $\beta \in [0, 1]$ provides a tradeoff. On the one hand the smaller β is chosen the *more* the exponential average resembles the actual sequence of negative descent directions since *more* of each negative gradient direction is used in the update, but the *less* these descent directions summarize all of the previously seen negative gradients. On the other hand the larger β is chosen the *less* these exponentially averaged descent steps resemble the negative gradient directions, since each update will use *less* of each subsequent negative gradient direction, but the *more* they represent a summary of them. Often in practice larger values of β are used, in the range

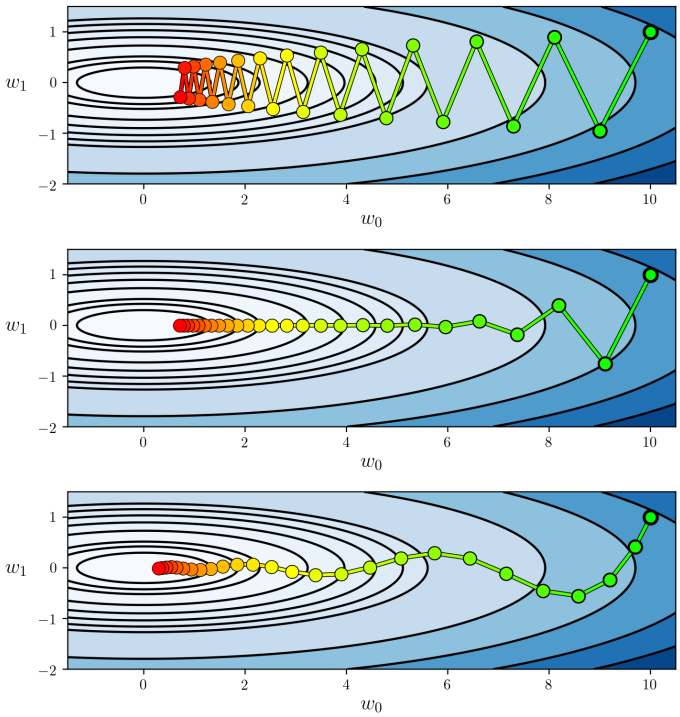

Figure 3.21 Figure associated with Example 3.20.

[0.7, 1], meaning that often in practice a strongly summarizing descent direction $\mathbf{d}^{k-1}$ tends to provide better performance. [4]

Example 3.20 **Accelerating gradient descent on a simple quadratic**

In this Example we compare a run of standard gradient descent to the *momentum accelerated* version detailed above using the first quadratic from Example 3.17 of the previous Section. Here we re-produce the same run of 25 gradient descent steps shown in this prior Example, and compare it to a run of momentum accelerated gradient descent with two choices of $\beta \in \{0.2, 0.7\}$. All three runs are initialized at the same point $\mathbf{w}^0 = \begin{bmatrix} 10 \\ 1 \end{bmatrix}$ and use the same learning rate $\alpha = 10^{-1}$.

We show the resulting steps taken by the standard gradient descent run in

[4] Note in practice this step is also written slightly differently than above, although this notational re-arrangement amounts to exactly the same thing: instead of averaging the *negative* gradient directions the gradient itself is exponentially averaged, and then the *step* is taken in their *negative* direction. This means that we initialize our exponential average at the first *negative* descent direction $\mathbf{d}^0 = -\nabla g\left(\mathbf{w}^0\right)$ and for $k - 1 > 0$ the general descent direction and corresponding step is computed as

$$\begin{aligned} \mathbf{d}^{k-1} &= \beta\, \mathbf{d}^{k-2} + (1-\beta)\nabla g\left(\mathbf{w}^{k-1}\right) \\ \mathbf{w}^k &= \mathbf{w}^{k-1} - \alpha\, \mathbf{d}^{k-1}. \end{aligned} \tag{3.48}$$

the top panel of Figure 3.21 (where significant zig-zagging is present), and the momentum accelerated versions using $\beta = 0.2$ and $\beta = 0.7$ in the middle and bottom panels of this Figure respectively. Both momentum accelerated versions clearly outperform the standard scheme, in that they reach a point closer to the true minimum of the quadratic, and clearly zig-zagging is significantly reduced in both momentum accelerated runs. The choice of $\beta = 0.7$ provides better performance here because it reduces zig-zagging while maintaining larger steplength - since its stronger summarizing descent direction maintains a larger magnitude (from the first steps of the run which are far from the minimum where the magnitude of the negative gradient was large) as the run approaches the minimum of the quadratic. The case where $\beta = 0.2$, which is closer to the original negative gradient, starts taking smaller steps as it approaches the minimum since its descent direction more closely resembles the true negative gradient (whose magnitude is diminishing rapidly).

3.9 Normalized gradient descent

In the Section 3.7.3 we discussed a fundamental issue associated with the *magnitude* of the negative gradient and the fact that it vanishes near stationary points: gradient descent slowly crawls near stationary points. In particular this means - depending on the function being minimized - that it can halt near saddle points. In this Section we describe a popular enhancement to the standard gradient descent scheme, called *normalized gradient descent*, that is specifically designed to ameliorate this issue. The core of this idea lies in a simple inquiry: since the (vanishing) *magnitude* of the negative gradient is what causes gradient descent to slowly crawl near stationary points / halt at saddle points, what happens if we simply ignore the magnitude at each step by *normalizing* it out? In short by normalizing out the gradient magnitude we ameliorate some of the 'slow crawling' problem of standard gradient descent, empowering the method to push through flat regions of a function with much greater ease. This is quite analogous to our discussion about normalizing descent directions produced via zero order methods in Section 2.6.2.

3.9.1 Normalizing out the full gradient magnitude

In Section 3.7.3 we saw how the length of a standard gradient descent step is *proportional to the magnitude of the gradient*, expressed algebraically as $\alpha \left\| \nabla g(\mathbf{w}^{k-1}) \right\|_2$. Moreover we also saw there how this fact explains why gradient descent *slowly crawls* near stationary points, since near such points the *magnitude* of the gradient vanishes.

Since the magnitude of the gradient is to blame for slow crawling near sta-

tionary points, what happens if we simply ignore it by normalizing it out of the update step and just travel in the direction of negative gradient itself?

One way to normalize a (gradient) descent direction is by *normalizing out by its full magnitude*. Doing so gives a *normalized gradient descent step* of the form

$$\mathbf{w}^k = \mathbf{w}^{k-1} - \alpha \frac{\nabla g(\mathbf{w}^{k-1})}{\left\|\nabla g(\mathbf{w}^{k-1})\right\|_2} \tag{3.49}$$

In doing this we do indeed ignore the magnitude of the gradient, since we can easily compute the length of such a step. Provided the magnitude of the gradient is non-zero we have

$$\left\|\mathbf{w}^k - \mathbf{w}^{k-1}\right\|_2 = \left\|\left(\mathbf{w}^{k-1} - \alpha \frac{\nabla g(\mathbf{w}^{k-1})}{\left\|\nabla g(\mathbf{w}^{k-1})\right\|_2}\right) - \mathbf{w}^{k-1}\right\|_2 = \left\|-\alpha \frac{\nabla g(\mathbf{w}^{k-1})}{\left\|\nabla g(\mathbf{w}^{k-1})\right\|_2}\right\|_2 = \alpha. \tag{3.50}$$

In other words, if we normalize out the magnitude of the gradient at each step of gradient descent then the length of each step *is exactly equal to the value of our steplength / learning rate parameter* α. This is precisely what we did with the random search method in Section 2.6.2.

Notice that if we slightly re-write the fully-normalized step above as

$$\mathbf{w}^k = \mathbf{w}^{k-1} - \frac{\alpha}{\left\|\nabla g(\mathbf{w}^{k-1})\right\|_2} \nabla g(\mathbf{w}^{k-1}) \tag{3.51}$$

we can interpret our fully magnitude normalized step as a standard gradient descent step with a steplength / learning rate value $\frac{\alpha}{\left\|\nabla g(\mathbf{w}^{k-1})\right\|_2}$ that *adjusts itself* at each step based on the magnitude of the gradient to ensure that the length of each step is precisely α.

Also notice that in practice it is often useful to add a small constant ϵ (e.g., 10^{-7} or smaller) to the gradient magnitude to avoid potential division by zero (where the magnitude completely vanishes)

$$\mathbf{w}^k = \mathbf{w}^{k-1} - \frac{\alpha}{\left\|\nabla g(\mathbf{w}^{k-1})\right\|_2 + \epsilon} \nabla g(\mathbf{w}^{k-1}) \tag{3.52}$$

Example 3.21 **Ameliorating slow-crawling behavior near minima**

Shown in the top row of Figure 3.22 is a repeat of the run of gradient descent first detailed in Example 3.18, only here we use a fully normalized gradient step. Here we use the same number of steps and steplength / learning rate value used in that Example (which led to slow-crawling with the standard scheme). Here however the normalized step - unaffected by the vanishing gradient magnitude - is able to pass easily through the flat region of this function and find a point very close to the minimum as shown in the top-right panel of Figure 3.22. Comparing

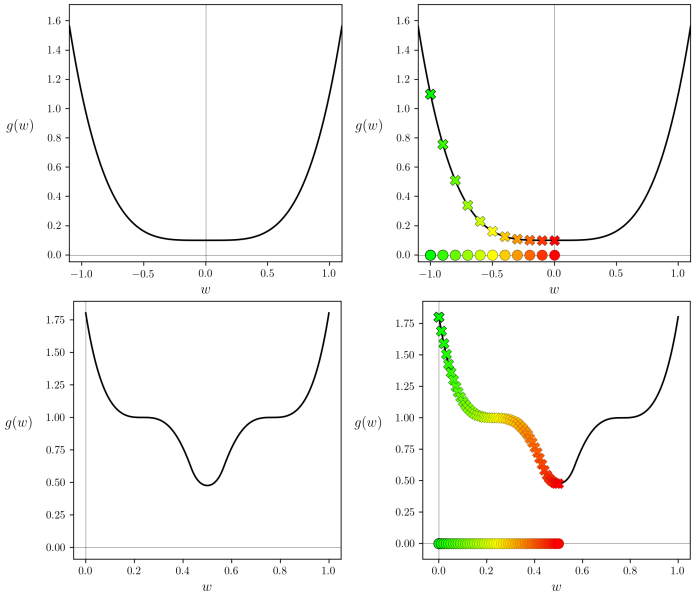

Figure 3.22 Figure associated with Example 3.21 (top row) and Example 3.22. See text for further details.

this run to the original run (of standard gradient descent) in the top-right panel of Figure 3.18 we can see that the normalized run gets considerably closer to the global minimum of the function.

Example 3.22 Ameliorating slow-crawling near saddle points

Shown in the bottom row of Figure 3.22 is a repeat of the run of gradient descent first detailed in Example 3.19, only here we use a fully normalized gradient step. Here we use the same number of steps and steplength / learning rate value used in that Example (which led to halting at a saddle point with the standard scheme). Here we use the same number of steps and steplength / learning rate value used previously (which led to the standard scheme halting at the saddle point). Here however the normalized step - unaffected by the vanishing gradient magnitude - is able to pass easily through the flat region of the saddle point and reach a point of this function close to the minimum.

Comparing this run (in the bottom-right panel of the Figure) to the original run shown in the bottom-right panel of Figure 3.22, the normalized version allows us to get much closer to the global minimum of the function.

Example 3.23 Trade-off when using normalized gradient descent

In this Figure 3.23 we show a comparison of fully-normalized and standard gradient descent (left and right panels respectively) on the simple quadratic function

$$g(w) = w^2 \tag{3.53}$$

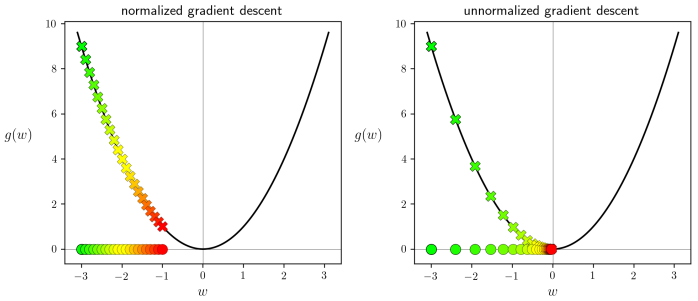

Figure 3.23 Figure associated with Example 3.23. See text for details.

Both algorithms use the same initial point ($w^0 = -3$), steplength parameter ($\alpha = 0.1$), and maximum number of iterations (20 each). Steps are colored from green to red to indicate the starting and ending points of each run, with circles denoting the actual steps in the input space and 'x' marks denoting their respective function evaluations.

Notice how - due to the re-scaling of each step via the derivative length - the standard version races to the global minimum of the function. Meanwhile the normalized version - taking constant length steps - gets only a fraction of the way there. This behavior is indicative of how a normalized step will fail to leverage the gradient when it is large - as the standard method does - in order to take larger steps at the beginning of a run.

3.9.2 Normalizing out the magnitude component-wise

Remember that the gradient is a vector of N *partial derivatives*

$$\nabla g(\mathbf{w}) = \begin{bmatrix} \frac{\partial}{\partial w_1} g(\mathbf{w}) \\ \frac{\partial}{\partial w_2} g(\mathbf{w}) \\ \vdots \\ \frac{\partial}{\partial w_N} g(\mathbf{w}) \end{bmatrix} \tag{3.54}$$

with the j^{th} partial derivative $\frac{\partial}{\partial w_j} g(\mathbf{w})$ defining how the gradient behaves along the j^{th} coordinate axis.

If we then look at what happens to the j^{th} *partial derivative* of the gradient when we normalize off the *full magnitude* of the gradient

$$\frac{\frac{\partial}{\partial w_j} g(\mathbf{w})}{\left\| \nabla g(\mathbf{w}) \right\|_2} = \frac{\frac{\partial}{\partial w_j} g(\mathbf{w})}{\sqrt{\sum_{n=1}^{N} \left(\frac{\partial}{\partial w_n} g(\mathbf{w}) \right)^2}} \tag{3.55}$$

we can see that the j^{th} partial derivative is normalized using a sum of the magnitudes of every partial derivative. This means that if the j^{th} partial derivative

is already small in magnitude itself, doing this will erase virtually all of its contribution to the final descent step. Therefore normalizing by the magnitude of the entire gradient can be problematic when dealing with functions containing regions that are flat with respect to only some of our weights / partial derivative directions, as it *diminishes* the contribution of the very partial derivatives we wish to enhance by ignoring magnitude.

As an alternative we can normalize out the magnitude of the gradient *component-wise*. In other words, instead of normalizing each partial derivative by the magnitude of the entire gradient we can normalize each partial derivative with respect to only itself. Doing this we divide off the magnitude of the j^{th} partial derivative from itself as

$$\frac{\frac{\partial}{\partial w_j} g\left(\mathbf{w}\right)}{\sqrt{\left(\frac{\partial}{\partial w_j} g\left(\mathbf{w}\right)\right)^2}} = \frac{\frac{\partial}{\partial w_j} g\left(\mathbf{w}\right)}{\left|\frac{\partial}{\partial w_j} g\left(\mathbf{w}\right)\right|} = \text{sign}\left(\frac{\partial}{\partial w_j} g\left(\mathbf{w}\right)\right). \tag{3.56}$$

So in the j^{th} direction we can write this component-normalized gradient descent step as

$$w_j^k = w_j^{k-1} - \alpha \frac{\frac{\partial}{\partial w_j} g\left(\mathbf{w}^{k-1}\right)}{\sqrt{\left(\frac{\partial}{\partial w_j} g\left(\mathbf{w}^{k-1}\right)\right)^2}}. \tag{3.57}$$

or equivalently as

$$w_j^k = w_j^{k-1} - \alpha \, \text{sign}\left(\frac{\partial}{\partial w_j} g\left(\mathbf{w}^{k-1}\right)\right). \tag{3.58}$$

We can then write the entire component-wise normalized step can be similarly written as e.g.,

$$\mathbf{w}^k = \mathbf{w}^{k-1} - \alpha \, \text{sign}\left(\nabla g\left(\mathbf{w}^{k-1}\right)\right) \tag{3.59}$$

where here the $\text{sign}\left(\cdot\right)$ acts component-wise on the gradient vector. We can then easily compute the length of a single step of this component-normalized gradient descent step (provided the partial derivatives of the gradient are all non-zero) as

$$\left\|\mathbf{w}^k - \mathbf{w}^{k-1}\right\|_2 = \left\|\left(\mathbf{w}^{k-1} - \alpha \, \text{sign}\left(\nabla g\left(\mathbf{w}^{k-1}\right)\right)\right) - \mathbf{w}^{k-1}\right\|_2 = \sqrt{N}\,\alpha. \tag{3.60}$$

In other words, if we normalize out the magnitude of gradient component-wise at each step of gradient descent then the length of each step is exactly equal to the value of our steplength / learning rate parameter α times the square root of N.

Notice then that if we slightly re-write the j^{th} component-normalized step equivalently as

$$w_j^k = w_j^{k-1} - \frac{\alpha}{\sqrt{\left(\frac{\partial}{\partial w_j} g\left(\mathbf{w}^{k-1}\right)\right)^2}} \frac{\partial}{\partial w_j} g\left(\mathbf{w}^{k-1}\right). \tag{3.61}$$

we can interpret our component normalized step as a standard gradient descent step with an individual steplength / learning rate value $\frac{\alpha}{\sqrt{\left(\frac{\partial}{\partial w_j} g\left(\mathbf{w}^{k-1}\right)\right)^2}}$ per component that all adjusts themselves individually at each step based on component-wise magnitude of the gradient to ensure that the length of each step is precisely $\sqrt{N}\,\alpha$. Indeed if we write

$$\mathbf{a}^{k-1} = \begin{bmatrix} \frac{\alpha}{\sqrt{\left(\frac{\partial}{\partial w_1} g\left(\mathbf{w}^{k-1}\right)\right)^2}} \\ \frac{\alpha}{\sqrt{\left(\frac{\partial}{\partial w_2} g\left(\mathbf{w}^{k-1}\right)\right)^2}} \\ \vdots \\ \frac{\alpha}{\sqrt{\left(\frac{\partial}{\partial w_N} g\left(\mathbf{w}^{k-1}\right)\right)^2}} \end{bmatrix} \tag{3.62}$$

then the full component-normalized descent step can also be written as

$$\mathbf{w}^k = \mathbf{w}^{k-1} - \mathbf{a}^{k-1} \odot \nabla g(\mathbf{w}^{k-1}) \tag{3.63}$$

where the $\odot$ symbol denotes component-wise multiplication. Again note that this is indeed equivalent to the previously written version of the full component-normalized gradient $\mathbf{w}^k = \mathbf{w}^{k-1} - \alpha\, \text{sign}\left(\nabla g\left(\mathbf{w}^{k-1}\right)\right)$. However note in practice if implemented as shown in equation (20) a small $\epsilon > 0$ is added to the denominator of each value of each entry of $\mathbf{a}^{k-1}$ to avoid division by zero.

Example 3.24 **Full versus coordinate normalized gradient descent**

In this example we use the function

$$g(w_0, w_1) = \max\left(0, \tanh(4w_0 + 4w_1)\right) + \max(0, \text{abs}\left(0.4w_0\right)) + 1 \tag{3.64}$$

to show the difference between fully and component normalized gradient descent steps on a function that has a very narrow flat region along only a single dimension of its input. Here this function - whose surface and contour plots can be seen in the left and right panels of Figure 3.24 respectively - is very flat along the w_1 direction for any fixed value of w_0, and has a very narrow valley leading towards its minima the w_1 dimension where $w_0 = 0$. If initialized

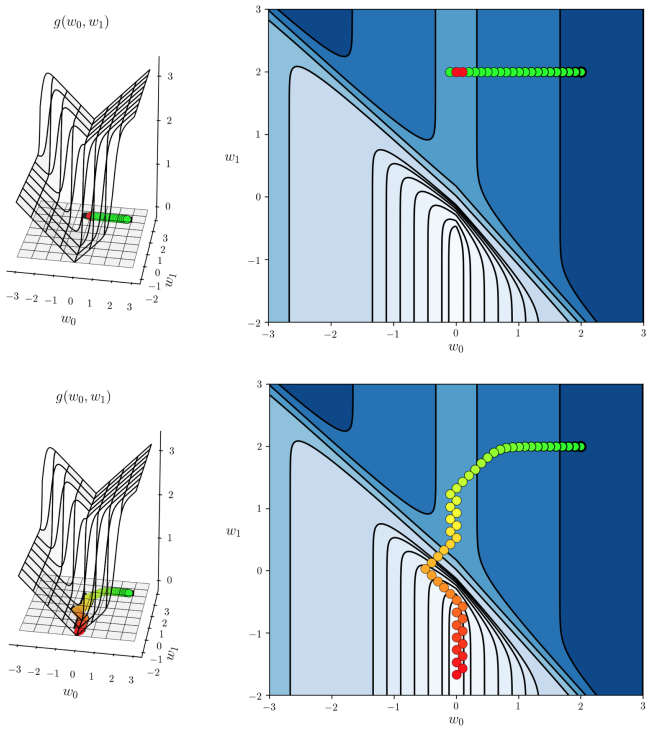

Figure 3.24 Figure associated with Example 3.24

at a point where $w_1 > 2$ this function cannot be minimized very easily using standard gradient descent *or* the fully-normalized version. In the latter case, the magnitude of the partial derivative in w_1 nearly zero everywhere, and so fully-normalizing makes this contribution smaller and halts progress. Below we show the result of 1000 steps of fully-normalized gradient descent starting at the point $\mathbf{w}^0 = \begin{bmatrix} 2 \\ 2 \end{bmatrix}$, colored green (at the start of the run) to red (at its finale). As can be seen, little progress is made because the *direction* provided by the partial derivative in w_1 could not be leveraged properly.

In order to make significant progress from this initial point a gradient descent method needs to enhance the minute sized partial derivative in the w_1 direction, and one way to do this is via the component-normalization scheme. In the bottom row of the Figure we show the results of using this version of normalized gradient descent starting at the same initialization and employing the same steplength / learning rate. Here we only need 50 steps in order to make significant progress.

3.9.3 General usage of normalized gradient descent schemes

Normalizing out the gradient magnitude - using either of the approaches detailed above - ameliorates the 'slow crawling' problem of standard gradient descent and empowers the method to push through flat regions of a function with much greater ease. This includes flat regions of a function that may lead to a local minimum, or the region around a saddle point of a non-convex function where standard gradient descent can halt. However - as highlighted in Example 3.23 - in normalizing every step of standard gradient descent we do *shorten* the first few steps of the run that are typically large (since random initializations are often far from stationary points of a function). This is the trade-off of the normalized step when compared with the standard gradient descent scheme: we trade shorter initial steps for longer ones around stationary points.

3.10 Advanced gradient based methods

In Section 3.8 we described the notion of *momentum accelerated gradient descent*, and how it is a natural remedy for the *zig-zagging* problem the standard gradient descent algorithm suffers when run along *long narrow valleys*. As we the momentum acceleration descent direction $\mathbf{d}^{k-1}$ is simply an *exponential average* of gradient descent directions taking the form

$$\mathbf{d}^{k-1} = \beta\,\mathbf{d}^{k-2} - (1-\beta)\,\nabla g\left(\mathbf{w}^{k-1}\right)\mathbf{w}^{k} = \mathbf{w}^{k-1} + \alpha\,\mathbf{d}^{k-1} \tag{3.65}$$

where $\beta \in [0, 1]$ is typically set at a value of $\beta = 0.8$ or higher.

Then in Section 3.9.2 we saw how *normalizing the gradient descent direction component-wise* helps deal with the problem standard gradient descent has when traversing *flat regions* of a function. We saw there how a component-normalized gradient descent step takes the form (for the j^{th} component of $\mathbf{w}$)

$$w_j^k = w_j^{k-1} - \alpha\,\frac{\frac{\partial}{\partial w_j}g\left(\mathbf{w}^{k-1}\right)}{\sqrt{\left(\frac{\partial}{\partial w_j}g\left(\mathbf{w}\right)\right)^2}} \tag{3.66}$$

where in practice a small fixed value $\epsilon > 0$ is often added to the denominator on the right hand side to avoid division by zero.

With the knowledge that these two additions to the standard gradient descent step help solve two fundamental problems associated with the descent direction used with gradient descent, it is natural to try to *combine them* in order to leverage both enhancements. There are a number of ways one might think to do this. For example, one could momentum accelerate a component-wise normalized direction or - in other words - replace the gradient descent direction in the

exponential average in Equation 3.10 with its component-normalized version shown in Equation 3.10.

Another way of combining the two ideas would be to component-normalize the exponential average descent direction computed in momentum-accelerated gradient descent. That is, compute the exponential average direction in the top line of Equation 3.10 and then normalize *it* (instead of the raw gradient descent direction) as shown in Equation 3.10. With this idea we can write out the update for the j^{th} component of the resulting descent direction as

$$d_j^{k-1} = \beta\, d_j^{k-2} - (1-\beta)\frac{\partial}{\partial w_j} g\left(\mathbf{w}^{k-1}\right) d_j^{k-1} \longleftarrow \frac{d_j^{k-1}}{\sqrt{\left(d_j^{k-1}\right)^2}}. \tag{3.67}$$

Many popular first order steps used to tune machine learning models - particularly those involving employing deep neural networks (see Chapter 13) - combine momentum and normalized gradient descent in this sort of way. Below we list a few examples including the popular *Adam* and *RMSprop* first order steps.

Example 3.25 **Adaptive Moment Estimation (Adam)**

Adaptive Moment Estimation (Adam) is a component-wise normalized gradient step employing independently calculated exponential averages for both the descent direction *and* its magnitude. That is, we compute j^{th} coordinate of the updated descent direction by first computing the exponential average of the gradient descent direction d_j^k squared magnitude h_j^k separately along this coordinate as

$$\begin{aligned} d_j^{k-1} &= \beta_1\, d_j^{k-2} + (1-\beta_1)\frac{\partial}{\partial w_j} g\left(\mathbf{w}^{k-1}\right) \\ h_j^{k-1} &= \beta_2\, h_j^{k-2} + (1-\beta_2)\left(\frac{\partial}{\partial w_j} g\left(\mathbf{w}^{k-1}\right)\right)^2 \end{aligned} \tag{3.68}$$

where β_1 and β_2 are exponential average parameters that lie in the range $[0, 1]$. Popular values the parameters of this update step are $\beta_1 = 0.9$, $\beta_2 = 0.999$. Note as with any exponential average these two updates apply when $k - 1 > 0$ and should be initialized at first values from the series they respectively model: that is the initial descent direction $d_j^0 = \frac{\partial}{\partial w_j} g\left(\mathbf{w}^0\right)$ and its squared magnitude $h_j^0 = \left(\frac{\partial}{\partial w_j} g\left(\mathbf{w}^0\right)\right)^2$. [5]

The *Adam* step is then a component-wise normalized descent step using this

[5] Note: the authors of this particular update step proposed that each exponential average be *inialized at zero* - i.e., as $d_j^0 = 0$ and $h_j^0 = 0$ - instead of the first step in each series they respectively model (i.e., the initial derivative and its squared magnitude). This initialization - along with the values for β_1 and β_2 typically chosen to be greater than 0.9 - cause the the first few update steps of these exponential averages to be 'biased' towards zero as well. Because of this they also

exponentially average descent direction and magnitude. A step in the j^{th} coordinate then takes the form

$$w_j^k = w_j^{k-1} - \alpha \frac{d_j^{k-1}}{\sqrt{h_j^{k-1}}}. \tag{3.69}$$

Notice - as we saw the (component) normalized step in the previous Section - that if we slightly re-write above as $w_j^k = w_j^{k-1} - \frac{\alpha}{\sqrt{h_j^{k-1}}} d_j^{k-1}$ we can interpret the Adam step as a momentum-accelerated gradient descent step with an individual steplength / learning rate value $\frac{\alpha}{\sqrt{h_j^{k-1}}}$ per component that all adjusts themselves individually at each step based on component-wise exponentially normalized magnitude of the gradient.

Example 3.26 **Root Mean Squared Propogation (RMSprop)**

This popular first order step is a variant of the component-wise normalized step where - instead of normalizing each component of the gradient by its magnitude - each component is normalized by the exponential average of the component-wise magnitudes of previous gradient directions.

Denoting h_j^k the exponential average of of the squared magnitude of the j^{th} partial derivative at step k we have

$$h_j^k = \gamma\, h_j^{k-1} + (1-\gamma)\left(\frac{\partial}{\partial w_j} g\left(\mathbf{w}^{k-1}\right)\right)^2 \tag{3.70}$$

The Root Mean Squared Error Propogation (RMSprop) step is then a component-wise normalized descent step using this exponential average. A step in the j^{th} coordinate then takes the form

$$w_j^k = w_j^{k-1} - \alpha \frac{\frac{\partial}{\partial w_j} g\left(\mathbf{w}^{k-1}\right)}{\sqrt{h_j^{k-1}}} \tag{3.71}$$

Popular values the parameters of this update step are $\beta = 0.9$ and $\alpha = 10^{-2}$.

Notice - as we saw the (component) normalized step in the previous Section - that if we slightly re-write above as $w_j^k = w_j^{k-1} - \frac{\alpha}{\sqrt{h_j^{k-1}}} \frac{\partial}{\partial w_j} g\left(\mathbf{w}^{k-1}\right)$. we can interpret the RMSprop step as a standard gradient descent step with an individual steplength / learning rate value $\frac{\alpha}{\sqrt{h_j^{k-1}}}$ per component that all adjusts themselves

employ a 'bias-correction' term to compensate for this initialization of the form $d_j^{k-1} \longleftarrow \frac{d_j^{k-1}}{1-(\beta_1)^{k-1}}$ and $h_j^{k-1} \longleftarrow \frac{h_j^{k-1}}{1-(\beta_2)^{k-1}}$.

individually at each step based on exponential average of component-wise magnitudes of the gradient.

3.11 Mini-batch optimization

In machine learning applications we are almost always tasked with minimizing a *sum* of P functions *of the same form*. Written algebraically such a cost takes the form

$$g(\mathbf{w}) = \sum_{p=1}^{P} g_p(\mathbf{w}). \tag{3.72}$$

where g_1, g_2, ..., g_P are functions of the same type - e.g., they can be convex quadratic functions with different constants (as with Least Squares linear regression discussed in Chapter 5), all parameterized using the same set of weights $\mathbf{w}$.

This special *summation structure* allows for a simple but very effective enhancement to virtually any local optimization scheme called *mini-batch optimization*. Mini-batch optimization is most often used in combination with a gradient-based step like those discussed in this Chapter.

3.11.1 A simple idea with powerful consequences

The motivation for *mini-batch optimization* rests on a simple inquiry: for this sort of function g shown in Equation 3.11, what would happen if instead of taking one descent step in g - that is, one descent step in the entire sum of the functions g_1, g_2, ..., g_P *simultaneously* - we took a sequence of P descent steps in g_1, g_2, ..., g_P *sequentially* by first descending in g_1, then in g_2, etc., until finally we descend in g_P. As we will see empirically throughout this text, starting with the examples below, in many instances this idea can lead to considerably faster optimization of a such a function. While this finding is largely *empirical*, it can be interpreted in the framework of machine learning as we will see in Section BLAH.

The gist of this idea is drawn graphically in Figure 3.25 below for the case $P = 3$, where we graphically compare the idea of taking a the a descent step simultaneously in g_1, g_2, ..., g_P versus a sequence of P descent steps in g_1 then g_2 etc., up to g_P.

Taking the first step of a local method to minimize a cost function g of the form in Equation 3.11, we begin at some initial point $\mathbf{w}^0$, determine a descent direction $\mathbf{d}^0$, and transition to a new point $\mathbf{w}^1$ as

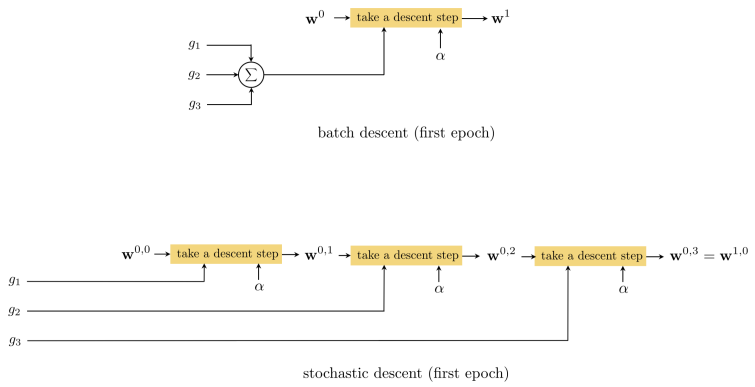

Figure 3.25 An abstract illustration of the full batch (top panel) and stochastic descent approaches to local optimization.

$$\mathbf{w}^1 = \mathbf{w}^0 + \alpha\,\mathbf{d}^0. \tag{3.73}$$

By analogy, if we were to follow the mini-batch idea detailed above this entails taking a sequence of P steps. If we call our initial point $\mathbf{w}^{0,0} = \mathbf{w}^0$, we then first determine a descent direction $\mathbf{d}^{0,1}$ in g_1 alone, the first function in the sum of g, and take a step in this direction as

$$\mathbf{w}^{0,1} = \mathbf{w}^{0,0} + \alpha\,\mathbf{d}^{0,1}. \tag{3.74}$$

Next we determine a descent direction $\mathbf{d}^{0,2}$ in g_2, the second function in the sum for g, and take a step in this direction

$$\mathbf{w}^{0,2} = \mathbf{w}^{0,1} + \alpha\,\mathbf{d}^{0,2} \tag{3.75}$$

and so forth. Continuing this pattern we take a sequence of P steps, where $\mathbf{d}^{0,p}$ is the descent direction found in g_p, that takes the following form

$$\begin{aligned}
\mathbf{w}^{0,1} &= \mathbf{w}^{0,0} + \alpha\,\mathbf{d}^{0,1} \\
\mathbf{w}^{0,2} &= \mathbf{w}^{0,1} + \alpha\,\mathbf{d}^{0,2} \\
&\vdots \\
\mathbf{w}^{0,p} &= \mathbf{w}^{0,p-1} + \alpha\,\mathbf{d}^{0,p} \\
&\vdots \\
\mathbf{w}^{0,P} &= \mathbf{w}^{0,P-1} + \alpha\,\mathbf{d}^{0,P}
\end{aligned} \tag{3.76}$$

This sequence of updates completes one sweep through the functions g_1, g_2, ..., g_P, and is commonly referred to as an *epoch*. If we continued this pattern and took another sweep through each the P functions we perform a second *epoch* of steps, and so on. Below we compare the k^{th} step of a standard descent method (left) to the k^{th} *epoch* of this mini-batch idea (right).

$$\begin{array}{c|c} \text{full (batch) descent step} & \text{mini-batch epoch, batch-size = 1} \\ & \mathbf{w}^{k,1} = \mathbf{w}^{k-1,0} + \alpha\, \mathbf{d}^{k-1,1} \\ & \vdots \\ \mathbf{w}^{k} = \mathbf{w}^{k-1} + \alpha\, \mathbf{d}^{k-1} & \mathbf{w}^{k,p} = \mathbf{w}^{k-1,p-1} + \alpha\, \mathbf{d}^{k-1,p} \\ & \vdots \\ & \mathbf{w}^{k,P} = \mathbf{w}^{k-1,P-1} + \alpha\, \mathbf{d}^{k-1,P} \end{array} \tag{3.77}$$

When employed with gradient-based steps these steps take a very convenient form. For example, noting that $\nabla g\left(\mathbf{w}\right) = \nabla \sum_{p=1}^{P} g_p\left(\mathbf{w}\right)$ the k^{th} standard gradient descent step in all P summands can be written as

$$\mathbf{w}^{k} = \mathbf{w}^{k-1} - \alpha \nabla \sum_{p=1}^{P} g_p\left(\mathbf{w}^{k-1}\right) \tag{3.78}$$

where here the descent direction is $\mathbf{d}^{k-1} = -\nabla \sum_{p=1}^{P} g_p\left(\mathbf{w}^{k-1}\right)$. The gradient descent direction in just a single function g_p at the k^{th} epoch of a mini-batch run is likewise just *negative gradient of this function alone* $\mathbf{d}^{k-1,p} = -\nabla g_p\left(\mathbf{w}^{k-1,p}\right)$, and so the step minibatch step $\mathbf{w}^{k,p} = \mathbf{w}^{k-1,p-1} + \alpha\, \mathbf{d}^{k-1,p}$ is given by

$$\mathbf{w}^{k,p} = \mathbf{w}^{k-1,p-1} - \alpha\, \nabla g_p\left(\mathbf{w}^{k-1,p}\right). \tag{3.79}$$

3.11.2 Descending with larger mini-batch sizes

Instead of taking P sequential steps in single functions g_p one-at-a-time (a mini-batch of *size* 1), we can more generally take fewer steps in one epoch, but take each step with respect to *several* of the functions g_p e.g., two functions at-a-time, or three functions at-a-time, etc.,. With this slight twist on the idea detailed above we take fewer steps per epoch but take each with respect to larger non-overlapping subsets of the functions g_1, g_2, ..., g_P, but still sweep through each cost function of the form in Equation 3.11 exactly once per epoch.

Choosing a general batch size M we split up our set of functions into non-overlapping subsets, where each subset has the same size with perhaps the exception of one (e.g., if $P = 5$ and we choose a batch size of 2 then we will have two subsets of two functions with a final subset containing just one). If we

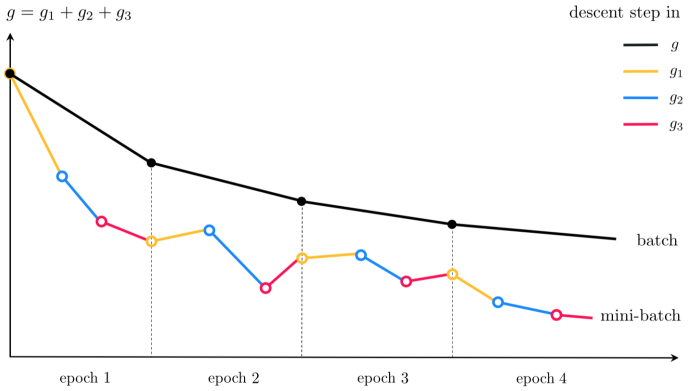

Figure 3.26 Figurative comparison between batch and mini-batch descent

denote Ω_m the set of indices of those summand functions g_1, g_2, ..., g_P in our m^{th} mini-batch then during our k^{th} epoch we take steps like those shown in the table below. Here $\mathbf{d}^{k-1,m}$ is the descent direction computed for the m^{th} minibatch.

The size M of the subset used is called the *batch-size* of the process (mini-batch optimization using a batch-size of 1 is also often referred to as *stochastic optimization*). What batch-size works best in practice - in terms of providing the greatest speed up in optimization - varies and is often problem dependent.

Once again in the case of gradient-based methods a mini-batch step using a batch size greater than 1 can be written out quite clearly, and just as easily implemented. If we denote by Ω_m the set of indices of our summand functions g_1, g_2, ..., g_P in the m^{th} mini-batch, then during the k^{th} epoch of our mini-batch optimization if we use the standard gradient method our descent direction is simply the negative gradient direction in the sum of these functions i.e.,

$$\mathbf{d}^{k-1,m} = -\nabla \sum_{p \in \Omega_m} g_p\left(\mathbf{w}^{k-1,m}\right). \tag{3.80}$$

3.11.3 Mini-batch optimization general performance

Is the trade-off - taking more steps per epoch with a mini-batch approach as opposed a full descent step - worth the extra effort? Typically *yes*. Often in practice when minimizing machine learning functions an epoch of mini-batch steps like those detailed above will drastically outperform an analogous full descent step - often referred to as a *full batch* or simply a *batch* epoch in the context of mini-batch optimization.

A prototypical comparison of a cost function history employing a batch and corresponding epochs of mini-batch optimization applied to the same hypothetical function g with the same initialization $\mathbf{w}^0$ is shown in the Figure 3.26 below. Because we take far more steps with the mini-batch approach and because each g_p takes the same form, each epoch of the mini-batch approach typically outperforms its full batch analog. Even when taking into account that far more descent

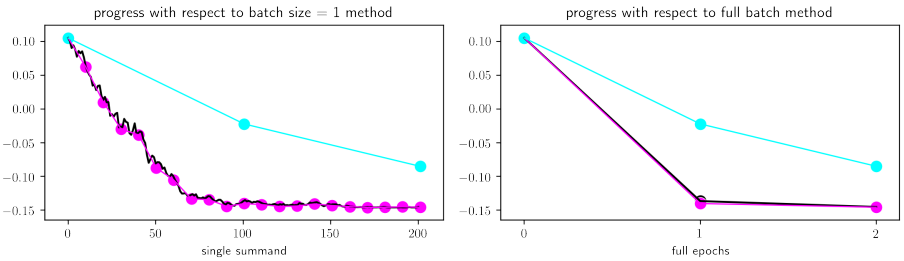

Figure 3.27 Figure associated with Example 3.27

steps are taken during an epoch of mini-batch optimization the method often greatly outperforms its full batch analog.

Example 3.27 **Minimizing a sum of quadratic functions**

In this Example we compare a full batch and two mini-batch runs (using batch-size 1 and 10 respectively) employing the standard gradient descent scheme. The function g we minimize in these various runs is a sum of $P = 100$ single input convex quadratic functions $g_p(w) = a_p + b_p w + c_p w^2$ whose parameters have been set at random[6]. In other words, our function g takes the form

$$g(\mathbf{w}) = \sum_{p=1}^{P} g_p(w) = \sum_{p=1}^{P} \left(a_p + b_p w + c_p w^2\right) \tag{3.81}$$

All three runs use the same (random) initial point $w^0 = 0$ and a steplength / learning rate $\alpha = 10^0$ and we take 2 epochs of each method.

In Figure 3.27 we plot the cost function history plots associated with the full batch (shown in blue), mini-batch with batch-sizes equal to 10 and 1 (in magenta and black respectively). In the left panel we show these three histories with respect to the batch-size = 1 steps, that is we show the cost function value at every single step of the batch-size = 1 method and plot the histories of our other mini-batch method every 10 steps and full batch method every 100 steps for visual comparison. In the right panel we show all three histories only after each full epoch is complete. From these plots can see clearly that both mini-batch approaches descended significantly faster than their full batch version.

---

[6] (by choosing values at random from a normal distribution with zero mean and unit standard deviation)

3.12 Conservative steplength rules*

In Section 3.6 we described how the steplength parameter α for the gradient descent step - whether fixed for all iterations or diminishing - is very often determined by trial and error in machine learning applications. However because gradient descent is built on such foundational first order ideas we can derive proper steplength parameter settings *mathematically* that are guaranteed to produce convergence of the algorithm. However these steplength choices are often quite *conservative*, specifically defined to force descent in the function at *every step*, and are therefore quite expensive computationally speaking. In this Section we briefly review major mathematically determined steplength schemes for the sake of the interested reader.

3.12.1 Gradient descent and simple quadratic surrogates

Crucial to the analysis of theoretically convergent steplength parameter choices for gradient descent is the following symmetric quadratic function

$$h_\alpha(\mathbf{w}) = g\left(\mathbf{w}^0\right) + \nabla g\left(\mathbf{w}^0\right)^T\left(\mathbf{w} - \mathbf{w}^0\right) + \frac{1}{2\alpha}\left\|\mathbf{w} - \mathbf{w}^0\right\|_2^2 \tag{3.82}$$

where $\alpha > 0$. Notice what precisely this is. The first two terms on the right hand side constitute the first order Taylor series approximation to $g(\mathbf{w})$ at a point $\mathbf{w}^0$ or - in other words - the formula for the tangent hyperplane there. The final term on the right hand side is the simplest quadratic component imaginable, turning the tangent hyperplane - regardless of whether or not it is tangent at a point that is locally convex or concave - into a convex and perfectly symmetric quadratic whose curvature is controlled in every dimension by the parameter α. Moreover note: like the hyperplane this quadratic is still tangent to the $g(\mathbf{w})$ at $\mathbf{w}^0$, matching both the function and derivative values at this point.

What happens to this quadratic when we change the value of α? In Figure 3.28 we illustrate the approximation for two different values of α with a generic convex function. Note the connection to α: *the larger the value α the wider the associated quadratic becomes*.

One of the beautiful things about such a simple quadratic approximation as h_α is that we can easily compute a unique global minimum for it, regardless of the value of α, by checking the first order optimality condition (see Section 3.2). Setting its gradient to zero we have

$$\nabla h_\alpha(\mathbf{w}) = \nabla g\left(\mathbf{w}^0\right) + \frac{1}{\alpha}\left(\mathbf{w} - \mathbf{w}^0\right) = \mathbf{0} \tag{3.83}$$

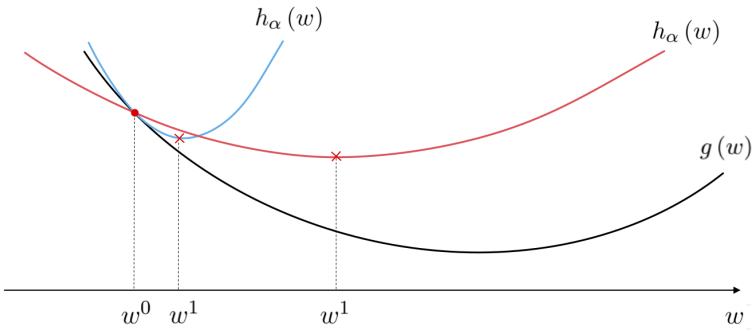

Figure 3.28 Two quadratic functions approximating the function g around $\mathbf{w}^0$ given by the quadratic approximation above. The value of α is larger with the red quadratic than with the blue one.

Rearranging the above and solving for $\mathbf{w}$, we can find the minimizer of h_α, which we call $\mathbf{w}^1$, given as

$$\mathbf{w}^1 = \mathbf{w}^0 - \alpha \nabla g\left(\mathbf{w}^0\right) \tag{3.84}$$

Thus the minimum of our simple quadratic approximation is precisely a standard gradient descent step at $\mathbf{w}^0$ with a steplength parameter α.

If we continue taking steps in this manner the k^{th} update is found as the minimum of the simple quadratic approximation associated with the previous update $\mathbf{w}^{k-1}$, which is likewise

$$h_\alpha(\mathbf{w}) = g\left(\mathbf{w}^{k-1}\right) + \nabla g\left(\mathbf{w}^{k-1}\right)^T \left(\mathbf{w} - \mathbf{w}^{k-1}\right) + \frac{1}{2\alpha}\left\|\mathbf{w} - \mathbf{w}^{k-1}\right\|_2^2 \tag{3.85}$$

where the minimum is once again given as the k^{th} gradient descent step

$$\mathbf{w}^k = \mathbf{w}^{k-1} - \alpha \nabla g\left(\mathbf{w}^{k-1}\right) \tag{3.86}$$

In sum, our exercise with the simple quadratic yields an alternative perspective on the standard gradient descent algorithm (detailed in Section 3.6): we can interpret gradient descent as an algorithm that uses linear approximation to move towards a function's minimum, or simultaneously as an algorithm that uses simple quadratic approximations to do the same. In particular this new perspective says that as we move along the direction of steepest descent of the hyperplane, moving from step to step, we are simultaneously 'hopping' down the global minima of these simple quadratic approximations. These two simultaneous perspectives are illustrated protoypically in the Figure 3.29.

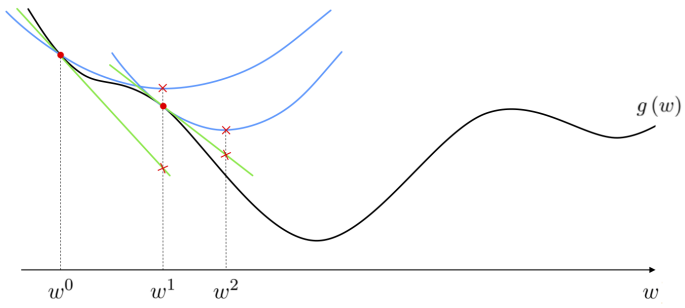

Figure 3.29 Gradient descent can be viewed simultaneously as using either linear or simple quadratic surrogates to find a stationary point of g. At each step the associated step length defines both how far along the linear surrogate we move before hopping back onto the function g, and at the same time the width of the simple quadratic surrogate which we minimize to reach the same point on g.

3.12.2 Backtracking line search

Since the negative gradient is a descent direction, then if we are at a step $\mathbf{w}^{k-1}$ then - if we make α small enough - the gradient descent step to $\mathbf{w}^k$ will decrease the value of g, i.e., $g\left(\mathbf{w}^k\right) \leq g\left(\mathbf{w}^{k-1}\right)$. Our first perspective on gradient descent (detailed in Section 3.6) tells us that this will work because as we shrink α we are traveling a shorter direction in the descent direction of the tangent hyperplane at $\mathbf{w}^{k-1}$, and if we shrink this distance enough the underlying function should also be decreasing in this direction. Our second perspective, given above, gives us a different but completely equivalent take on this issue: it tells us that in shrinking α we are *increasing* the curvature of the associated quadratic approximation shown in Equation 3.12.1 (whose minimum is the point we will move to) so that the minima of the quadratic approximation lies *above* the function. A step to such a point must decrease the function's value because at this point the quadratic is by definition at its lowest, and is in particular lower than where it began tangent to g i.e., at $g\left(\mathbf{w}^{k-1}\right)$.

How can we find a value of α that does just this at the point $\mathbf{w}^{k-1}$? We use the definition of the quadratic approximation there. If our generic gradient descent step is $\mathbf{w}^k = \mathbf{w}^{k-1} - \alpha\nabla g\left(\mathbf{w}^{k-1}\right)$, we want to determine a value of α so that at $\mathbf{w}^k$ the function is lower than the minimum of the quadratic, i.e., $g\left(\mathbf{w}^k\right) \leq h_\alpha\left(\mathbf{w}^k\right)$. We could select a large number of values for α and test this condition, keeping the one that provides the biggest decrease. However this is a computationally expensive and somewhat unwieldy prospect. Instead we test out values of α via an efficient *bisection* process by which we gradually decrease the value of α from some initial value until the inequality is satisfied. This procedure - referred to as *backtracking line search* - generally runs as follows.

1. Choose an initial value for α, for example $\alpha = 1$, and a scalar 'dampening factor' $t \in (0, 1)$.

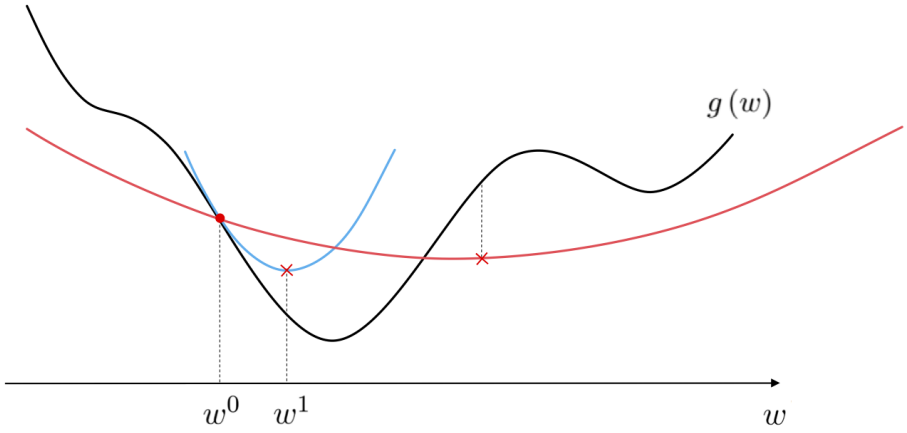

Figure 3.30 An illustration of our second perspective on how to choose a value for the steplength parameter α that is guaranteed to decrease the underlying function's value by taking a single gradient descent step. The value of α should be decreased until its minimum lies over the function. A step to such a point must decrease the function's value because at this point the quadratic is by definition at its lowest, and so is in particular lower than where it began tangent to g. Here the α value associated with the red quadratic is too large, while the one associated with the blue quadratic is small enough so that the quadratic lies above the function. A (gradient descent) step to this point decreases the value of the function g.

2. Create the candidate descent step $\mathbf{w}^k = \mathbf{w}^{k-1} - \alpha \nabla g\left(\mathbf{w}^{k-1}\right)$.

3. Test if $g\left(\mathbf{w}^k\right) \leq h_\alpha\left(\mathbf{w}^k\right)$. If yes, then choose $\mathbf{w}^k$ as the next gradient descent step, otherwise go to step 4.

4. Decrease the value of α as $\alpha \longleftarrow t\alpha$ and go back to step 2.

It is the logic of trying out a large value for α first and then decreasing it until we satisfy the inequality that prevents an otherwise unwieldy number of tests to be performed here. Note however how the dampening factor $t \in (0, 1)$ controls how coarsely we sample α values: in setting t closer to 1 we decrease the amount by which α is shrunk at each failure which could mean more evaluations are required to determine an adequate α value. Conversely setting t closer to 0 here shrinks α considerably with every failure, leading to completion more quickly but at the possible cost of outputting a small value for α (and hence a short gradient descent step).

Backtracking line search is a convenient rule for determining a steplength value at each iteration of gradient descent and works 'right out of the box'. However each gradient step using backtracking line search, compared to using a fixed steplength value, typically includes higher computational cost due to the search for proper step length.

[7]

3.12.3 Exact line search

In thinking on how one could automatically adjust the steplength value α at the $\mathbf{w}^{k-1}$ step of gradient descent, one might also think to try determining the steplength α that literally minimizes the function g directly along the k^{th} gradient descent step $\mathbf{w}^k = \mathbf{w}^{k-1} - \alpha\nabla g\left(\mathbf{w}^{k-1}\right)$ direction, that is

$$\underset{\alpha>0}{\text{minimize}}\; g\left(\mathbf{w}^{k-1} - \alpha\nabla g\left(\mathbf{w}^{k-1}\right)\right) \tag{3.88}$$

This idea is known as *exact line search*. Practically speaking, however, this idea must be implemented via a backtracking line search approach in much the same way we saw above - by successively examining smaller values until we find a value of α at the k^{th} step $\mathbf{w}^k = \mathbf{w}^{k-1} - \alpha\nabla g\left(\mathbf{w}^{k-1}\right)$ such that

$$g\left(\mathbf{w}^k\right) \leq g\left(\mathbf{w}^{k-1}\right) \tag{3.89}$$

This approach can lead to taking *large* steps that offer *little* decrease in the function value itself (an issue that we are guarded against with the main backtracking line search method described above due to the convex curvature of the simple quadratic we use to determine when to stop shrinking α - this is depicted in Figure 3.31).

3.12.4 Conservatively optimal fixed steplength values

Suppose construct the simple quadratic approximation of the form in Equation 3.12.1 and we turn up the value of α in the simple quadratic approximation so that it reflects the greatest amount of curvature or change in the function's first derivative. Setting the quadratic's parameter α to this maximum curvature - called a function's *Lipschitz constant* - means that the quadratic approximation will lie completely above the function everywhere except at its point of tangency with the function at $\left(\mathbf{w}^{k-1}, g\left(\mathbf{w}^{k-1}\right)\right)$.

Since the approximation is convex its minimum at $\mathbf{w}^k$ will of course lie below

[7] The inequality $g\left(\mathbf{w}^k\right) \leq h_\alpha\left(\mathbf{w}^k\right)$ is also written equivalently as

$$g\left(\mathbf{w}^k\right) \leq g\left(\mathbf{w}^{k-1}\right) - \frac{\alpha}{2}\left\|\nabla g\left(\mathbf{w}^{k-1}\right)\right\|_2^2 \tag{3.87}$$

by plugging in the step $\mathbf{w}^k = \mathbf{w}^{k-1} - \alpha\nabla g\left(\mathbf{w}^{k-1}\right)$ into the quadratic $h_\alpha\left(\mathbf{w}^k\right)$, completing the square, and simplifying. This equivalent version tells us that so long as we have not reached a stationary point of g the term $\frac{\alpha}{2}\left\|\nabla g\left(\mathbf{w}^{k-1}\right)\right\|_2^2$ will always be positive, hence finding a value of α that satisfies our inequality means that $g\left(\mathbf{w}^k\right)$ will be strictly smaller than $g\left(\mathbf{w}^{k-1}\right)$.

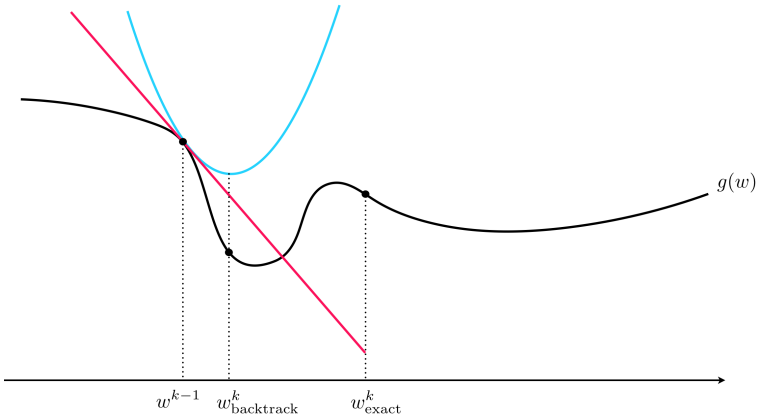

Figure 3.31 With exact line search, unlike the backtracking method, the quadratic associated to gradient descent need not lie entirely above the function g as long as we get a decrease - however small - in the function evaluation.

its point of tangency with the function itself, and since α has been set so that the entire quadratic itself lies above the function this means that in particular this minimum lies above the function. In other words, our gradient descent step must lead to a smaller evaluation of g, i.e., $g\left(\mathbf{w}^k\right) < g\left(\mathbf{w}^{k-1}\right)$.

As detailed in Section 4.2 curvature information of a function is found in its second derivative(s). More specifically, for a single-input function maximum curvature is defined as the maximum (in absolute value) taken by its second derivative

$$\underset{w}{\text{maximize}} \left|\frac{\mathrm{d}^2}{\mathrm{d}w^2}g(w)\right| \tag{3.90}$$

Analogously for a multi-input function $g(\mathbf{w})$ to determine its maximum curvature we must determine the *largest possible eigenvalue (in magnitude) of its Hessian matrix*. Or, written algebraically, employing the spectral norm $\|\cdot\|_2$ (see Section 7.12)

$$\underset{\mathbf{w}}{\text{maximize}} \left\|\nabla^2 g\left(\mathbf{w}\right)\right\|_2. \tag{3.91}$$

As daunting a task as this may seem it can in fact be done analytically for a range of common machine learning functions including as linear regression, (two-class and multi-class) logistic regression, support vector machines, as well as shallow neural networks.

Once determined this maximum curvature - or an upper bound on it - called L gives a fixed steplength $\alpha = \frac{1}{L}$ that can be used so that the k^{th} descent step

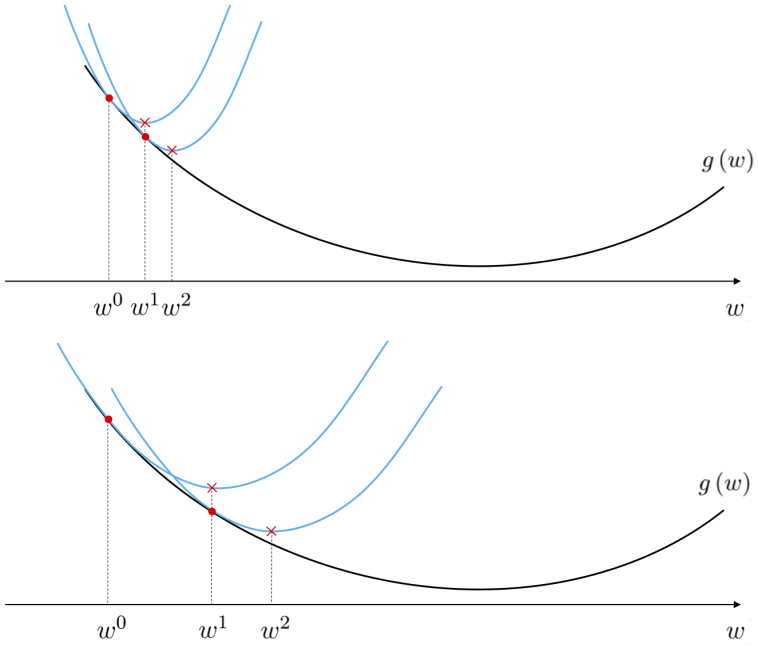

Figure 3.32 (top panel) Too conservative of fixed step length leads to smaller descent steps. (bottom panel) Another conservative fixed step length where the curvature of its associated quadratic just matches the greatest curvature of the hypothetical cost function while still lying entirely above the function. Such a step length is referred to as 'optimally conservative'. Note that while the underlying cost here is drawn convex this applies to non-convex cost functions as well, whose greatest curvature could be negative (on a concave part of the function).

$\mathbf{w}^k = \mathbf{w}^{k-1} - \frac{1}{L}\nabla g\left(\mathbf{w}^{k-1}\right)$ is guaranteed to always descend in the function g [8].

[8] It is fairly easy to rigorously show that the simple quadratic surrogate tangent to g at $\left(\mathbf{w}^{k-1}, g\left(\mathbf{w}^{k-1}\right)\right)$, with $\alpha = \frac{1}{L}$

$$h_{\frac{1}{L}}(\mathbf{w}) = g\left(\mathbf{w}^{k-1}\right) + \nabla g\left(\mathbf{w}^{k-1}\right)^T\left(\mathbf{w} - \mathbf{w}^{k-1}\right) + \frac{L}{2}\left\|\mathbf{w} - \mathbf{w}^{k-1}\right\|_2^2 \tag{3.92}$$

where L is the Lipschitz constant for the function g, indeed lies completely above the function g at all points. Writing out the first order Taylor's formula for g centered at $\mathbf{w}^{k-1}$, we have

$$g(\mathbf{w}) = g\left(\mathbf{w}^{k-1}\right) + \nabla g\left(\mathbf{w}^{k-1}\right)^T\left(\mathbf{w} - \mathbf{w}^{k-1}\right) + \frac{1}{2}\left(\mathbf{w} - \mathbf{w}^{k-1}\right)^T\nabla^2 g(\mathbf{c})\left(\mathbf{w} - \mathbf{w}^{k-1}\right) \tag{3.93}$$

where $\mathbf{c}$ is a point on the line segment connecting $\mathbf{w}$ and $\mathbf{w}^{k-1}$. Since $\nabla^2 g \preceq L\mathbf{I}_{N\times N}$ we can bound the right hand side of the Taylor's formula by replacing $\nabla^2 g(\mathbf{c})$ with $L\mathbf{I}_{N\times N}$, giving

$$g(\mathbf{w}) \leq g\left(\mathbf{w}^{k-1}\right) + \nabla g\left(\mathbf{w}^{k-1}\right)^T\left(\mathbf{w} - \mathbf{w}^{k-1}\right) + \frac{L}{2}\|\mathbf{w} - \mathbf{w}^{k-1}\|_2^2 = h_{\frac{1}{L}}(\mathbf{w}) \tag{3.94}$$

which indeed holds for all $\mathbf{w}$.

With this steplength we can initialize gradient descent anywhere in the input domain of a function and gradient descent will converge to a stationary point.

Example 3.28 **Computing the Lipschitz constant of sinusoid**

Let us compute the Lipschitz constant - or maximum curvature - of the sinusoid function

$$g(w) = \sin(w) \tag{3.95}$$

We can easily compute the second derivative of this function, which is

$$\frac{\mathrm{d}^2}{\mathrm{d}w^2} g(w) = -\sin(w) \tag{3.96}$$

The maximum value this (second derivative) function can take is 1, hence the Lipschitz constant of g can be set to $L = 1$ and so $\alpha = \frac{1}{L} = 1$.

Example 3.29 **Computing the Lipschitz constant of a quadratic in 3d**

Now we look at computing the Lipschitz constant of the quadratic function

$$g(\mathbf{w}) = \mathbf{w}^T \mathbf{A} \mathbf{w} \tag{3.97}$$

where $\mathbf{A} = \begin{bmatrix} 2 & 0 \\ 0 & 1 \end{bmatrix}$. Here the Hessian is simply $\nabla^2 g(\mathbf{w}) = \mathbf{A} + \mathbf{A}^T = 2\mathbf{A}$ for all input $\mathbf{w}$, and since the eigenvalues of a diagonal matrix are precisely its diagonal elements, the maximum (in magnitude) eigenvalue is clearly 4. Thus here we can set the Lipschitz constant to $L = 4$, giving a conservative optimal steplength value of $\alpha = \frac{1}{4}$.

3.12.5 How to use the conservative fixed step length rule

The conservatively optimal Lipschitz constant step length rule will always work "right out of the box' to produce a convergent gradient descent scheme, therefore it can be a very convenient rule to use in practice. However, as its name implies and as described previously, it is indeed a conservative rule by nature.

Therefore in practice, if one has the resources, one should use it as a benchmark to search for a larger convergence-forcing fixed step length rules. In other words with the Lipschitz constant step length $\alpha = \frac{1}{L}$ calculated one can easily test larger step lengths of the form $\alpha = t \cdot \frac{1}{L}$ for any constant $t > 1$. Indeed depending problem values of t ranging from 1 to large values like 100 can work well in practice.

3.12.6 Convergence proofs for gradient descent schemes*

To set the stage for the material of this Section, it will be helpful to briefly point out the specific set of mild conditions satisfied by all of the cost functions we aim to minimize in this book, as these conditions are relied upon explicitly in the upcoming convergence proofs. These three basic conditions are listed below.

1 They have piecewise-differentiable first derivative.

2 They are bounded from below, i.e., they never take on values at $-\infty$.

3 They have bounded curvature.

While the first condition is specific to the set of cost functions we discuss, in particular so that we include the squared margin perceptron which is not smooth and indeed has piecewise differentiable first derivative while the other costs are completely smooth, the latter two conditions are very common assumptions made in the study of mathematical optimization more generally.

Convergence with Lipschitz constant fixed step length

With the gradient of g being Lipschitz continuous with constant L, from Section 3.12.4 we know that at the k^{th} iteration of gradient descent we have a corresponding quadratic upper bound on g of the form

$$g(\mathbf{w}) \leq g\left(\mathbf{w}^{k-1}\right) + \nabla g\left(\mathbf{w}^{k-1}\right)^T\left(\mathbf{w} - \mathbf{w}^{k-1}\right) + \frac{L}{2}\|\mathbf{w} - \mathbf{w}^{k-1}\|_2^2, \tag{3.98}$$

where indeed this inequality holds for all $\mathbf{w}$ in the domain of g. Now plugging in the form of the gradient step $\mathbf{w}^k = \mathbf{w}^{k-1} - \frac{1}{L}\nabla g\left(\mathbf{w}^{k-1}\right)$ into the above and simplifying gives

$$g\left(\mathbf{w}^k\right) \leq g\left(\mathbf{w}^{k-1}\right) - \frac{1}{2L}\|\nabla g\left(\mathbf{w}^{k-1}\right)\|_2^2 \tag{3.99}$$

which, since $\|\nabla g\left(\mathbf{w}^{k-1}\right)\|_2^2 \geq 0$, indeed shows that the sequence of gradient steps with conservative fixed step length is decreasing. To show that it converges to a stationary point where the gradient vanishes we subtract off $g\left(\mathbf{w}^{k-1}\right)$ from both sides of the above, and sum the result over $k = 1...K$ giving

$$\sum_{k=1}^{K} g\left(\mathbf{w}^k\right) - g\left(\mathbf{w}^{k-1}\right) = g\left(\mathbf{w}^K\right) - g\left(\mathbf{w}^0\right) \leq -\frac{1}{2L}\sum_{k=1}^{K}\left\|\nabla g\left(\mathbf{w}^{k-1}\right)\right\|_2^2. \tag{3.100}$$

Note importantly here that since g is bounded below so too is $g\left(\mathbf{w}^K\right)$ for all K, and this implies that, taking $K \longrightarrow \infty$, that we *must* have that

$$\sum_{k=1}^{\infty}\left\|\nabla g\left(\mathbf{w}^{k-1}\right)\right\|_2^2 < \infty. \tag{3.101}$$

If this were not the case then we would contradict the assumption that g has a finite lower bound, since equation (3.100) would say that $g\left(\mathbf{w}^K\right)$ would be negative infinity! Hence the fact that the infinite sum above must be finite implies that as $k \longrightarrow \infty$ we have that

$$\left\|\nabla g\left(\mathbf{w}^{k-1}\right)\right\|_2^2 \longrightarrow 0, \tag{3.102}$$

or that the that the sequence of gradient descent steps with step length determined by the Lipschitz constant of the gradient of g produces a vanishing gradient. Or, in other words, that this sequence indeed converges to a stationary point of g.

Note that we could have made the same argument above using any fixed step length smaller than $\frac{1}{L}$ as well.

Convergence of gradient descent with backtracking line search

With the assumption that g has bounded curvature, stated formally in Section 3.12.4 that g has a Lipschitz continuous gradient with some constant L (even if we cannot calculate L explicitly), it follows that with a fixed choice of initial step length $\alpha > 0$ and $t \in (0, 1)$ for all gradient descent steps we can always find an integer n_0 such that

$$t^{n_0}\alpha \leq \frac{1}{L}. \tag{3.103}$$

Thus we always have the lower bound on an adaptively chosen step length $\hat{t} = t^{n_0}\alpha$, meaning formally that the backtracking found step length at the k^{th} gradient descent step will always be larger than this lower bound, i.e.,

$$\alpha_k \geq \hat{t} > 0. \tag{3.104}$$

Now, recall that by running the backtracking procedure at the k^{th} gradient step we produce a step length α_k that ensures the associated quadratic upper bound

$$g\left(\mathbf{w}\right) \leq g\left(\mathbf{w}^{k-1}\right) + \nabla g\left(\mathbf{w}^{k-1}\right)^T\left(\mathbf{w} - \mathbf{w}^{k-1}\right) + \frac{1}{2\alpha_k}\|\mathbf{w} - \mathbf{w}^{k-1}\|_2^2 \tag{3.105}$$

holds for all $\mathbf{w}$ in the domain of g. Plugging in the gradient step $\mathbf{w}^k = \mathbf{w}^{k-1} - \alpha_k \nabla g\left(\mathbf{w}^{k-1}\right)$ into the above and simplifying we have equivalently that

$$g\left(\mathbf{w}^k\right) \leq g\left(\mathbf{w}^{k-1}\right) - \frac{\alpha_k}{2}\left\|\nabla g\left(\mathbf{w}^{k-1}\right)\right\|_2^2, \tag{3.106}$$

which indeed shows that that step produces decrease in the cost function. To show that the sequence of gradient steps converges to a stationary point of g we first subtract off $g\left(\mathbf{w}^{k-1}\right)$ and sum the above over $k = 1...K$ which gives

$$\sum_{k=1}^{K} g\left(\mathbf{w}^k\right) - g\left(\mathbf{w}^{k-1}\right) = g\left(\mathbf{w}^K\right) - g\left(\mathbf{w}^0\right) \leq -\frac{1}{2}\sum_{k=1}^{K}\alpha_k\left\|\nabla g\left(\mathbf{w}^{k-1}\right)\right\|_2^2. \tag{3.107}$$

Since g is bounded below so too is $g\left(\mathbf{w}^K\right)$ for all K, and therefore taking $K \longrightarrow \infty$ the above says that we must have

$$\sum_{k=1}^{\infty} \alpha_k \left\| \nabla g\left(\mathbf{w}^{k-1}\right) \right\|_2^2 < \infty. \tag{3.108}$$

Now, we know from equation (3.104) that since each $\alpha_k \geq \hat{t} > 0$ for all k. This implies that we must have that

$$\sum_{k=1}^{\infty} \alpha_k = \infty. \tag{3.109}$$

And this is just fine, because in order for the Equation above to hold we *must* have that

$$\left\| \nabla g\left(\mathbf{w}^{k-1}\right) \right\|_2^2 \longrightarrow 0, \tag{3.110}$$

as $k \longrightarrow \infty$, for otherwise equation (3.108) could not be true. This shows that the sequence of gradient steps determined by backtracking line search converges to a stationary point of g.

3.13 Exercises

Section 3.1 exercises

1. **First order condition calculations**

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

   a) $g(\mathbf{w}) = \frac{1}{2}\mathbf{w}^T\mathbf{B}\mathbf{w} + \mathbf{c}^T\mathbf{w}$, $\mathbf{B} = \begin{bmatrix} 2 & 1 \\ 1 & 3 \end{bmatrix}$ and $\mathbf{c} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$

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

   c) $g(w) = \log(1 + \exp(w))$.

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

2. **Stationary points of a simple but common quadratic function**

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

$$g(\mathbf{w}) = a + \mathbf{b}^T\mathbf{w} + \mathbf{w}^T\mathbf{C}\mathbf{w} \tag{3.111}$$

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. Show that the single stationary point of this quadratic is - unlike a general quadratic - not the solution to a linear system, but can be expressed in closed form rather simply (see Example 3.4).

3. **Stationary points of The Rayleigh Quotient**

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

$$g(\mathbf{w}) = \frac{\mathbf{w}^T\mathbf{C}\mathbf{w}}{\mathbf{w}^T\mathbf{w}} \tag{3.112}$$

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

4. **Code up coordinate descent**

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

5. **Coordinate descent is a local optimization scheme**

   Express the coordinate descent method as a local optimization scheme in the general form described in Section 2.5. That is, as a sequence of steps of the form $\mathbf{w}^k = \mathbf{w}^{k-1} + \alpha\,\mathbf{d}^k$.

Section 3.5 exercises

6. **Try out gradient descent**

   Run gradient descent to minimize the following function

$$g(w) = \frac{1}{50}\left(w^4 + w^2 + 10w\right) \tag{3.113}$$

   with an initial point $w_0 = 2$ and 1000 iterations. Make three runs using each of the steplength values: $\alpha = 1$, $\alpha = 10^{-1}$, $\alpha = 10^{-2}$. Compute the derivative of this function by hand, and implement it (and the function itsel) in `Python` using `numpy`. You can base your implementation off the one given in Section 3.6.4.

   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?

7. **Compare fixed and diminishing steplength values for a simple example**

Repeat the comparison experiment described in Example 3.14, producing the cost function history plot comparison shown in the bottom panel of Figure 3.13. You can use the closed form derivative shown there or us an automatic differentiator.

8. **Oscillation in the cost function history plot**

Repeat the experiment described in Example 3.15, producing cost function history plot shown in the bottom panel of Figure 3.14.

9. **Tune fixed step length for gradient descent**

Take the cost function

$$g(\mathbf{w}) = \mathbf{w}^T\mathbf{w} \tag{3.114}$$

where $\mathbf{w}$ is an $N = 10$ dimensional input vector. g is convex with a single global minima at $\mathbf{w} = \mathbf{0}_{N\times 1}$. Code up gradient descent with a maximum iteration stopping criteria of 100 iterations (with no other stopping conditions). Using the initial point $\mathbf{w}^0 = 10 \cdot \mathbf{1}_{N\times 1}$ run gradient descent with step lengths $\alpha_1 = 0.001$, $\alpha_2 = 0.1$ and $\alpha_3 = 1.001$. Produce a cost function history plot to compare the three runs and see which performs best.

10. **Geometry of gradient descent step**

The distance between the $(k-1)^{th}$ and k^{th} gradient step can easily be calculated as $\left\|\mathbf{w}^k - \mathbf{w}^{k-1}\right\|_2 = \left\|\left(\mathbf{w}^{k-1} - \alpha\nabla g\left(\mathbf{w}^{k-1}\right)\right) - \mathbf{w}^{k-1}\right\|_2 = \alpha\left\|\nabla g\left(\mathbf{w}^{k-1}\right)\right\|_2$. In this exercise you will compute the corresponding length traveled along the $(k-1)^{th}$ first order Taylor series approximation $h(\mathbf{w}) = g\left(\mathbf{w}^{k-1}\right) + \nabla g\left(\mathbf{w}^{k-1}\right)^T\left(\mathbf{w} - \mathbf{w}^{k-1}\right)$.

In Figure 3.33 we show a detailed description of the precise geometry involved in taking the k^{th} gradient descent step with a fixed step length α. Use the details of this picture to show that the corresponding length traveled along the linear surrogate, i.e., $\ell = \left\|\begin{bmatrix}\mathbf{w}^k \\ h\left(\mathbf{w}^k\right)\end{bmatrix} - \begin{bmatrix}\mathbf{w}^{k-1} \\ h\left(\mathbf{w}^{k-1}\right)\end{bmatrix}\right\|_2$, is given precisely as

$$\ell = \alpha\sqrt{1 + \left\|\nabla g\left(\mathbf{w}^{k-1}\right)\right\|_2^2}\left\|\nabla g\left(\mathbf{w}^{k-1}\right)\right\|_2 \tag{3.115}$$

Hint: use the Pythagorean Theorem.

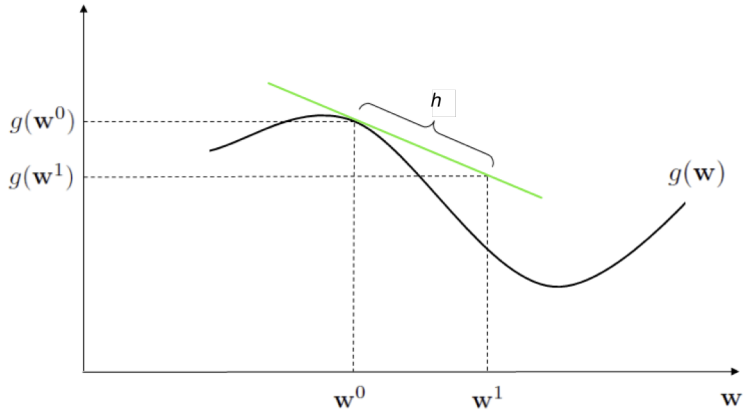

Figure 3.33 The geometry of a gradient descent step. See text for details.

Section 3.7 exercises

11. **Code up momentum accelerated gradient descent**

Code up the momentum accelerated gradient descent scheme described in Section 3.8.2 and use it to repeat the experiments detailed in Example 3.20 using a cost function history plot to come to the same conclusions drawn by studying the contour plots shown in Figure 3.21.

Section 3.8 exercises

12. **Slow-crawling behavior of gradient descent**

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

$$g(w_0, w_1) = \tanh(4w_0 + 4w_1) + \max(1, 0.4w_0^2) + 1 \tag{3.116}$$

shown in Figure 3.34. Using the initialization $\mathbf{w}^0 = \begin{bmatrix} 2 \\ 2 \end{bmatrix}$ make a run of 1000 steps of the standard and fully normalized gradient descent schemes, using a steplength $\alpha = 10^{-1}$ in both instances. The result of the normalized scheme is shown in the Figure. Use a cost function history plot to compare the two runs, noting the progress made with each approach.

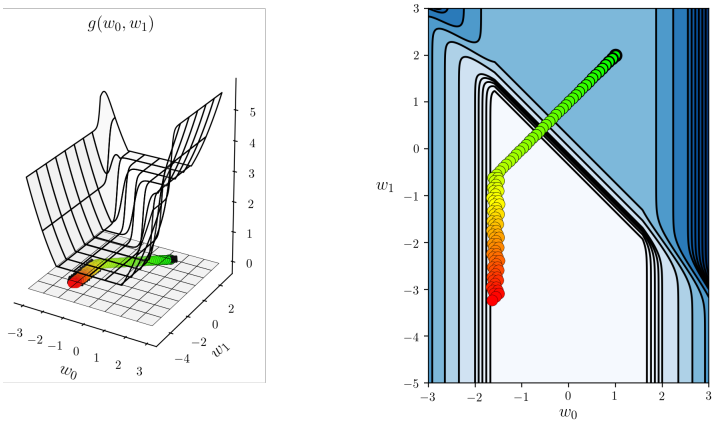

Figure 3.34 Figure associated with Exercise 13. See text for details.

13. **Comparing normalized gradient descent schemes**

Code up the full and component-wise normalized gradient descent schemes and repeat the experiment described in Example 3.24 using a cost function history plot to come to the same conclusions drawn by studying the plots shown in Figure 3.24.

Section 3.10 exercises

14. **Mini-batch gradient descent**

Implement the mini-batch gradient descent scheme and repeat the experiment described in Example 3.27. Create a set of two the cost function history plots like those shown in Figure 3.27 to come to a similar conclusion to that given in the Example.

15. **Mini-batch optimization using zero order method**

Repeat the previous exercise using the zero order coordinate descent method detailed in Section 2.7 instead of gradient descent.

Section 3.10 exercises

16. **Alternative formal definition of Lipschitz gradient**

An alternative to defining the Lipschitz constant by Equation (3.12.4) for functions g with Lipschitz continuous gradient is given by

$$\left\|\nabla g\left(\mathbf{x}\right)-\nabla g\left(\mathbf{y}\right)\right\|_2 \leq L\left\|\mathbf{x}-\mathbf{y}\right\|_2. \tag{3.117}$$

Using the limit definition of the derivative (see Section 7.1.1) show that this definition is equivalent to the one given in Equation 3.12.4.

17. **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 16 show that the composition $f\left(g\right)$ also has Lipschitz continuous gradient. What is the corresponding Lipschitz constant of this composition?