A Advanced First- and Second-Order Optimization Methods

A.1 Introduction

In this chapter we study advanced first- and second-order optimization techniques that are designed to ameliorate the natural weaknesses associated with gradient descent and Newton's method – as detailed previously in Sections 3.6 and 4.4, respectively.

A.2 Momentum-Accelerated Gradient Descent

In Section 3.6 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.

A.2.1 The exponential average

In Figure A.1 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 450 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 (i.e., 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, it is common practice to *smooth* them (in order to remove these zig-zagging motions) for better

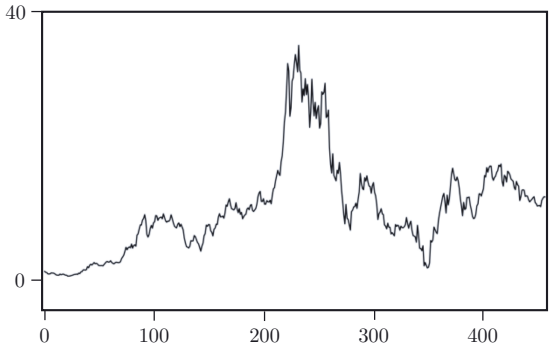

Figure A.1 An example of a time series data, representing the price of a financial stock measured at 450 consecutive points in time.

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. In Figure A.2 we show the result of smoothing the data shown in Figure A.1. Before we see how the exponential average is computed, it is first helpful to see how to compute a *cumulative 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

$$\begin{aligned} h^1 &= w^1 \\ h^2 &= \frac{w^1 + w^2}{2} \\ h^3 &= \frac{w^1 + w^2 + w^3}{3} \\ &\vdots \\ h^K &= \frac{w^1 + w^2 + w^3 + \cdots + w^K}{K}. \end{aligned} \tag{A.1}$$

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 cumulative average is written in Equation (A.1), we need access to every raw point w^1 through w^k in order to compute the kth cumulative average h^k. Alternatively, we can write this cumulative average by expressing h^k for $k > 1$ in a recursive manner involving only its preceding cumulative average h^{k-1} and current time series value w^k, as

$$h^k = \frac{k-1}{k} h^{k-1} + \frac{1}{k} w^k. \tag{A.2}$$

From a computational perspective, the recursive way of defining the cumulative average is far more efficient since at the kth 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 cumulative average formula.

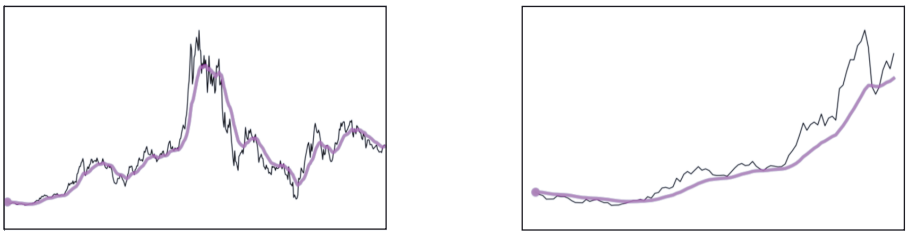

Figure A.2 (left panel) An exponential average (in pink) of the time series data shown in Figure A.1. (right panel) An exponential average of just the first 100 points of the time series, which is a smooth approximation to the underlying time series data.

Notice, at every step in Equation (A.2) that the coefficients on h^{k-1} and w^k *always* sum to 1, i.e., $\frac{k-1}{k} + \frac{1}{k} = 1$. As k grows larger both coefficients change: the coefficient on h^{k-1} gets closer to 1 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

$$h^k = \beta\, h^{k-1} + (1 - \beta)\, w^k. \tag{A.3}$$

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 cumulative average called an *exponential* average? Because if we roll back the update shown in Equation (A.3) to express h^k only in terms of preceding time series elements, as we did for cumulative average in Equation (A.1), an exponential (or power) pattern in the coefficients will emerge.

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

$$\mathbf{h}^k = \beta\, \mathbf{h}^{k-1} + (1 - \beta)\, \mathbf{w}^k. \tag{A.4}$$

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

A.2.2 Ameliorating the zig-zagging behavior of gradient descent

As mentioned previously, a sequence of gradient descent steps can be thought of as a *time series*. Indeed if we take K steps of a gradient descent run we do create a time series of ordered gradient descent steps $\mathbf{w}^1$, $\mathbf{w}^2$, ..., $\mathbf{w}^K$ 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 $\mathbf{w}^1$, $\mathbf{w}^2$, ..., $\mathbf{w}^K$ – as detailed in Section 3.6.3 – 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 they are created? Remember from Section 3.6.3 that the root cause of zig-zagging gradient descent 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 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 $k > 1$ the exponentially averaged descent direction $\mathbf{d}^{k-1}$ (using the formula in Equation (A.4)) takes the form

$$\mathbf{d}^{k-1} = \beta\,\mathbf{d}^{k-2} + (1-\beta)\left(-\nabla g\left(\mathbf{w}^{k-1}\right)\right). \tag{A.5}$$

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

$$\mathbf{w}^k = \mathbf{w}^{k-1} + \alpha\,\mathbf{d}^{k-1}. \tag{A.6}$$

Together this exponential averaging adds only a single extra step to our basic gradient descent scheme, forming a *momentum-accelerated* gradient descent step of the form[1]

[1] Sometimes this step is written slightly differently: 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 $k > 1$ the general descent direction and corresponding step is computed as

$$\begin{aligned}\mathbf{d}^{k-1} &= \beta\,\mathbf{d}^{k-2} + (1-\beta)\,\nabla g\left(\mathbf{w}^{k-1}\right)\\ \mathbf{w}^k &= \mathbf{w}^{k-1} - \alpha\,\mathbf{d}^{k-1}.\end{aligned} \tag{A.7}$$

$$\begin{aligned}\mathbf{d}^{k-1} &= \beta\,\mathbf{d}^{k-2} + (1-\beta)\left(-\nabla g\left(\mathbf{w}^{k-1}\right)\right)\\ \mathbf{w}^{k} &= \mathbf{w}^{k-1} + \alpha\,\mathbf{d}^{k-1}.\end{aligned} \tag{A.8}$$

The term "momentum" here refers to the new exponentially averaged descent direction $\mathbf{d}^{k-1}$, that by definition is a function of *every* negative gradient which precedes it. 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 trade-off. 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, e.g., in the range $[0.7, 1]$.

Example A.1 Accelerating gradient descent on a simple quadratic

In this example we compare a run of standard gradient descent to the *momentum-accelerated* version using a quadratic function of the form

$$g(\mathbf{w}) = a + \mathbf{b}^T\mathbf{w} + \mathbf{w}^T\mathbf{C}\mathbf{w} \tag{A.9}$$

where $a = 0$, $\mathbf{b} = \begin{bmatrix}0\\0\end{bmatrix}$, and $\mathbf{C} = \begin{bmatrix}0.5 & 0\\0 & 9.75\end{bmatrix}$.

Here we make three runs of 25 steps: a run of gradient descent and two runs of momentum-accelerated gradient descent using two choices of the parameter $\beta \in \{0.2, 0.7\}$. All three runs are initialized at the same point $\mathbf{w}^0 = [10\ \ 1]^T$, and use the same steplength $\alpha = 10^{-1}$.

We show the resulting steps taken by the standard gradient descent run in the top panel of Figure A.3 (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. Also note that the overall path taken by gradient descent is smoother in the bottom panel, due to the larger value of its corresponding β.

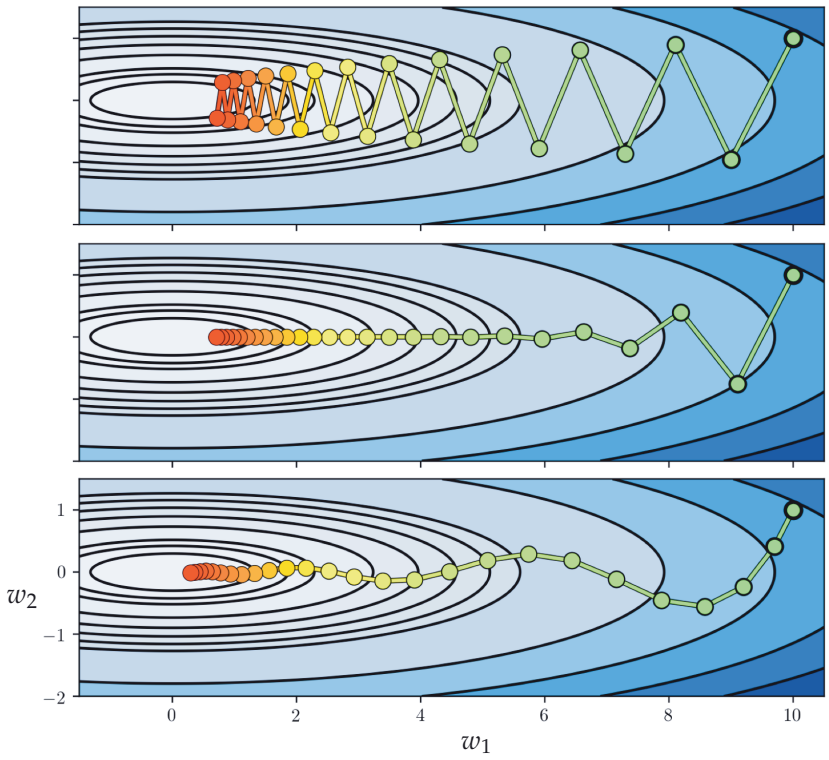

Figure A.3 Figure associated with Example A.1. The zig-zagging behavior of gradient descent can be ameliorated using the momentum-accelerated gradient descent step in Equation (A.8). See text for further details.

A.3 Normalized Gradient Descent

In Section 3.6 we discussed a fundamental issue associated with the *magnitude* of the negative gradient and the fact that it vanishes near stationary points, causing gradient descent to slowly crawl 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 or halt at saddle points, what happens if we simply ignore the magnitude at each step by *normalizing* it out?

A.3.1 Normalizing out the full gradient magnitude

In Section 3.6.4 we saw how the length of a standard gradient descent step is proportional to the magnitude of the gradient, expressed algebraically as

$\alpha \|\nabla g(\mathbf{w}^{k-1})\|_2$. 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 stationary 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 via dividing it by its magnitude. Doing so gives a *normalized gradient descent* step of the form

$$\mathbf{w}^k = \mathbf{w}^{k-1} - \alpha \frac{\nabla g(\mathbf{w}^{k-1})}{\left\|\nabla g(\mathbf{w}^{k-1})\right\|_2}. \tag{A.10}$$

In doing this we do indeed ignore the magnitude of the gradient, since

$$\left\|\mathbf{w}^k - \mathbf{w}^{k-1}\right\|_2 = \left\| -\alpha \frac{\nabla g(\mathbf{w}^{k-1})}{\left\|\nabla g(\mathbf{w}^{k-1})\right\|_2} \right\|_2 = \alpha. \tag{A.11}$$

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 parameter α. This is precisely what we did with the random search method in Section 2.5.2.

Notice that if we slightly rewrite the fully normalized step in Equation (A.10) as

$$\mathbf{w}^k = \mathbf{w}^{k-1} - \frac{\alpha}{\left\|\nabla g(\mathbf{w}^{k-1})\right\|_2} \nabla g(\mathbf{w}^{k-1}) \tag{A.12}$$

we can interpret our fully magnitude-normalized step as a standard gradient descent step with a steplength 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)

$$\mathbf{w}^k = \mathbf{w}^{k-1} - \frac{\alpha}{\left\|\nabla g(\mathbf{w}^{k-1})\right\|_2 + \epsilon} \nabla g(\mathbf{w}^{k-1}). \tag{A.13}$$

Example A.2 Ameliorating slow-crawling near minima and saddle points

Shown in the left panel of Figure A.4 is a repeat of the run of gradient descent first detailed in Example 3.14, only here we use a fully normalized gradient descent step. We use the same number of steps and steplength 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 at the origin. Comparing this run to the original run (of standard gradient descent) in Figure 3.14 we can see that the normalized run gets considerably closer to the global minimum of the function.

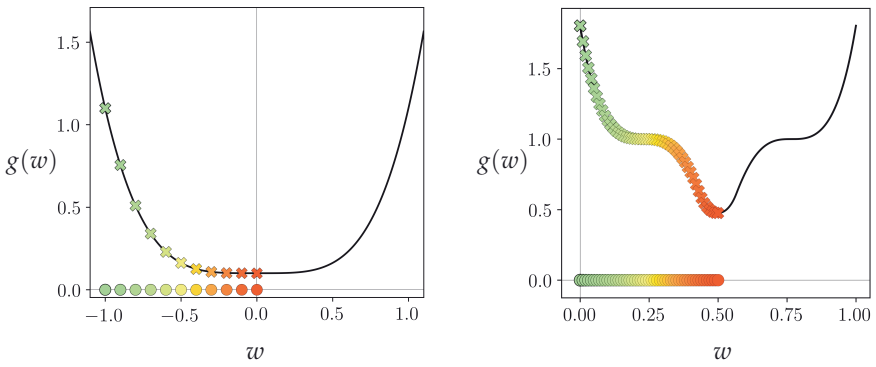

Figure A.4 Figure associated with Example A.2. By normalizing the gradient we can overcome the slow-crawling behavior of gradient descent near a function's minima (left panel) and saddle points (right panel). See text for further details.

Shown in the right panel of Figure A.4 is a repeat of the run of gradient descent first detailed in Example 3.14, only here we use a fully normalized gradient descent step. We use the same number of steps and steplength value used in that example (which led to halting at a saddle point 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 the saddle point and reach a point of this function close to the minimum.

Example A.3 A trade-off when using normalized gradient descent

In Figure A.5 we show a comparison of fully normalized (left panel) and standard (right panel) gradient descent on the simple quadratic function

$$g(w) = w^2. \tag{A.14}$$

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 the standard version races to the global minimum of the function, while the normalized version – taking fixed-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.

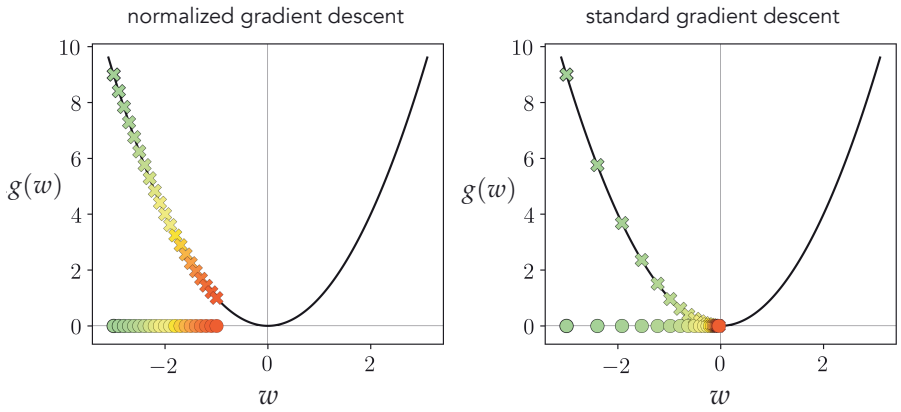

Figure A.5 Figure associated with Example A.3. While normalizing the gradient speeds up gradient descent near flat regions of a function where the magnitude of the gradient is small, it likewise fails to leverage the often large magnitude of the gradient far from a function's minima. See text for further details.

A.3.2 Normalizing out the magnitude component-wise

Remember that the gradient is a vector of N partial derivatives

$$\nabla g(\mathbf{w}) = \begin{bmatrix} \frac{\partial}{\partial w_1} g(\mathbf{w}) \\ \frac{\partial}{\partial w_2} g(\mathbf{w}) \\ \vdots \\ \frac{\partial}{\partial w_N} g(\mathbf{w}) \end{bmatrix} \tag{A.15}$$

with the jth partial derivative $\frac{\partial}{\partial w_j} g(\mathbf{w})$ defining how the gradient behaves along the jth coordinate axis. If we then look at what happens to the jth partial derivative of the gradient when we normalize off the *full magnitude* of the gradient

$$\frac{\frac{\partial}{\partial w_j} g(\mathbf{w})}{\left\| \nabla g(\mathbf{w}) \right\|_2} = \frac{\frac{\partial}{\partial w_j} g(\mathbf{w})}{\sqrt{\sum_{n=1}^{N} \left(\frac{\partial}{\partial w_n} g(\mathbf{w}) \right)^2}} \tag{A.16}$$

we can see that the jth partial derivative is normalized using a sum of the magnitudes of every partial derivative. This means that if the jth 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 the partial derivative direc-

tions, 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 as

$$\frac{\frac{\partial}{\partial w_j} g\left(\mathbf{w}\right)}{\sqrt{\left(\frac{\partial}{\partial w_j} g\left(\mathbf{w}\right)\right)^2}} = \frac{\frac{\partial}{\partial w_j} g\left(\mathbf{w}\right)}{\left|\frac{\partial}{\partial w_j} g\left(\mathbf{w}\right)\right|} = \text{sign}\left(\frac{\partial}{\partial w_j} g\left(\mathbf{w}\right)\right). \tag{A.17}$$

Therefore in the jth direction we can write this component-normalized gradient descent step as

$$w_j^k = w_j^{k-1} - \alpha \, \text{sign}\left(\frac{\partial}{\partial w_j} g\left(\mathbf{w}^{k-1}\right)\right). \tag{A.18}$$

We can then write the entire component-wise normalized step as

$$\mathbf{w}^k = \mathbf{w}^{k-1} - \alpha \, \text{sign}\left(\nabla g\left(\mathbf{w}^{k-1}\right)\right) \tag{A.19}$$

where here the sign function acts component-wise on the gradient vector. We can easily compute the length of a single step of this component-normalized gradient descent step (provided the partial derivatives of the gradient are all nonzero) as

$$\left\|\mathbf{w}^k - \mathbf{w}^{k-1}\right\|_2 = \left\|-\alpha \, \text{sign}\left(\nabla g\left(\mathbf{w}^{k-1}\right)\right)\right\|_2 = \sqrt{N}\,\alpha. \tag{A.20}$$

Notice, additionally, that if we slightly rewrite the jth component-normalized step in Equation (A.18) as

$$w_j^k = w_j^{k-1} - \frac{\alpha}{\sqrt{\left(\frac{\partial}{\partial w_j} g\left(\mathbf{w}^{k-1}\right)\right)^2}} \frac{\partial}{\partial w_j} g\left(\mathbf{w}^{k-1}\right) \tag{A.21}$$

we can interpret our component-normalized step as a standard gradient descent step with an individual steplength value

$$\text{steplength} = \frac{\alpha}{\sqrt{\left(\frac{\partial}{\partial w_j} g\left(\mathbf{w}^{k-1}\right)\right)^2}} \tag{A.22}$$

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

$$\mathbf{a}^{k-1} = \begin{bmatrix} \frac{\alpha}{\sqrt{\left(\frac{\partial}{\partial w_1} g\left(\mathbf{w}^{k-1}\right)\right)^2}} \\ \frac{\alpha}{\sqrt{\left(\frac{\partial}{\partial w_2} g\left(\mathbf{w}^{k-1}\right)\right)^2}} \\ \vdots \\ \frac{\alpha}{\sqrt{\left(\frac{\partial}{\partial w_N} g\left(\mathbf{w}^{k-1}\right)\right)^2}} \end{bmatrix} \tag{A.23}$$

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

$$\mathbf{w}^k = \mathbf{w}^{k-1} - \mathbf{a}^{k-1} \circ \nabla g(\mathbf{w}^{k-1}) \tag{A.24}$$

where the $\circ$ symbol denotes component-wise multiplication (see Appendix Section C.2.3). In practice, 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 A.4 Full versus component-normalized gradient descent
In this example we use the function

$$g(w_1, w_2) = \max(0, \tanh(4w_1 + 4w_2)) + |0.4w_1| + 1 \tag{A.25}$$

to show the difference between full 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 A.6, respectively – is very flat along the w_2 dimension for any fixed value of w_1, and has a very narrow valley leading towards its minima in the w_2 dimension where $w_1 = 0$. If initialized at a point where $w_2 > 2$ 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_2 is nearly zero everywhere, and so fully normalizing makes this contribution smaller, and halts progress. In the top row of the figure we show the result of 1000 steps of fully normalized gradient descent starting at the point $\mathbf{w}^0 = [2 \;\; 2]^T$, colored green (at the start of the run) to red (at its finale). As can be seen, little progress is made.

In the bottom row of the figure we show the results of using component-normalized gradient descent starting at the same initialization and employing the same steplength. Here we only need 50 steps in order to make significant progress.

In summary, normalizing out the gradient magnitude – using either of the approaches detailed previously – ameliorates the "slow-crawling" problem of

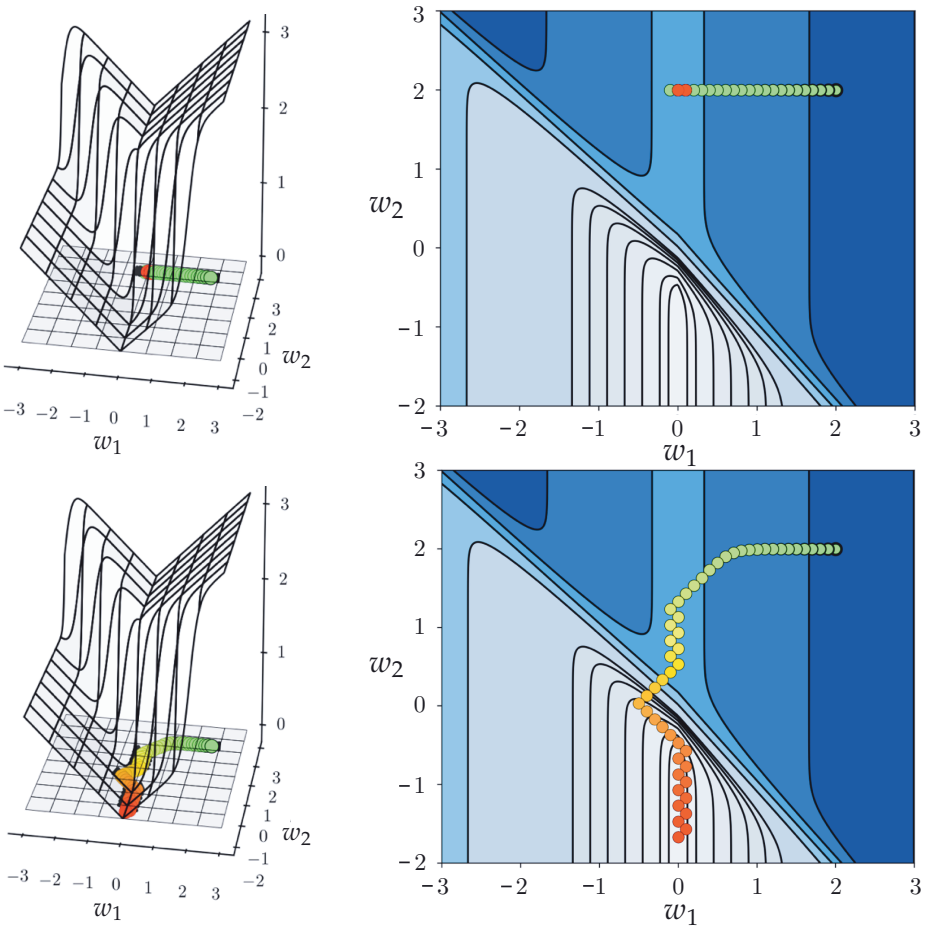

Figure A.6 Figure associated with Example A.4. See text for details.

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 nonconvex function where standard gradient descent can halt. However – as highlighted in Example A.3 – 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.

A.4 Advanced Gradient-Based Methods

In Section A.2 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 from when run along long narrow valleys. As we saw, the momentum-accelerated descent direction $\mathbf{d}^{k-1}$ is simply an exponential average of gradient descent directions taking the form

$$\begin{aligned}\mathbf{d}^{k-1} &= \beta\,\mathbf{d}^{k-2} + (1-\beta)\left(-\nabla g\left(\mathbf{w}^{k-1}\right)\right)\\ \mathbf{w}^{k} &= \mathbf{w}^{k-1} + \alpha\,\mathbf{d}^{k-1}\end{aligned} \tag{A.26}$$

where $\beta \in [0, 1]$ is typically set at a value of $\beta = 0.7$ or higher.

Then in Section A.3.2 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 jth component of $\mathbf{w}$)

$$w_j^k = w_j^{k-1} - \alpha\,\frac{\frac{\partial}{\partial w_j} g\left(\mathbf{w}^{k-1}\right)}{\sqrt{\left(\frac{\partial}{\partial w_j} g\left(\mathbf{w}\right)\right)^2}} \tag{A.27}$$

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.

One 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 (A.8) and then normalize *it* (instead of the raw gradient descent direction). With this idea we can write out the update for the jth component of the resulting descent direction as

$$d_j^{k-1} = \text{sign}\left(\beta\,d_j^{k-2} - (1-\beta)\,\frac{\partial}{\partial w_j} g\left(\mathbf{w}^{k-1}\right)\right). \tag{A.28}$$

Many popular first-order steps used to tune machine learning models – particularly those involving 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 A.5 Adaptive Moment Estimation (Adam)

Adaptive Moment Estimation (Adam) [69] is a component-wise normalized gradient step employing independently-calculated exponential averages for both the descent direction *and* its magnitude. That is, we compute the jth coordinate of the updated descent direction by first computing the exponential average of the gradient descent direction d_j^k and the squared magnitude h_j^k separately along this coordinate as

$$\begin{aligned} d_j^{k-1} &= \beta_1 \, d_j^{k-2} + (1 - \beta_1) \, \frac{\partial}{\partial w_j} g\left(\mathbf{w}^{k-1}\right) \\ h_j^{k-1} &= \beta_2 \, h_j^{k-2} + (1 - \beta_2) \left(\frac{\partial}{\partial w_j} g\left(\mathbf{w}^{k-1}\right) \right)^2 \end{aligned} \tag{A.29}$$

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

The Adam step is then a component-wise normalized descent step using this exponential average descent direction and magnitude, with a step in the jth coordinate taking the form

$$w_j^k = w_j^{k-1} - \alpha \frac{d_j^{k-1}}{\sqrt{h_j^{k-1}}}. \tag{A.30}$$

Example A.6 Root Mean Squared Propagation (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 by h_j^k the exponential average of the squared magnitude of the jth partial derivative at step k, we have

$$h_j^k = \gamma \, h_j^{k-1} + (1 - \gamma) \left(\frac{\partial}{\partial w_j} g\left(\mathbf{w}^{k-1}\right) \right)^2. \tag{A.31}$$

[2] The authors of this particular update step proposed that each exponential average be initialized at zero – i.e., $d_j^0 = 0$ and $h_j^0 = 0$ – instead of the first step in each series they respectively model. This initialization – along with the values for β_1 and β_2 that are typically chosen to be greater than 0.9 – causes the first few update steps of these exponential averages to be "biased" towards zero as well. Because of this they also employ a "bias-correction" term to compensate for this initialization.

The Root Mean Squared Error Propagation (RMSProp) [70] step is then a component-wise normalized descent step using this exponential average, with a step in the jth coordinate taking the form

$$w_j^k = w_j^{k-1} - \alpha \frac{\frac{\partial}{\partial w_j} g\left(\mathbf{w}^{k-1}\right)}{\sqrt{h_j^{k-1}}}. \tag{A.32}$$

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

A.5 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

$$g(\mathbf{w}) = \sum_{p=1}^{P} g_p(\mathbf{w}) \tag{A.33}$$

where $g_1, g_2, ..., g_P$ are functions of the same type, e.g., convex quadratic functions (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.

A.5.1 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 (A.33), 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 throughout this text, 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 7.8.

The gist of this idea is drawn in Figure A.7 for the case $P = 3$, where we graphically compare the idea of taking 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

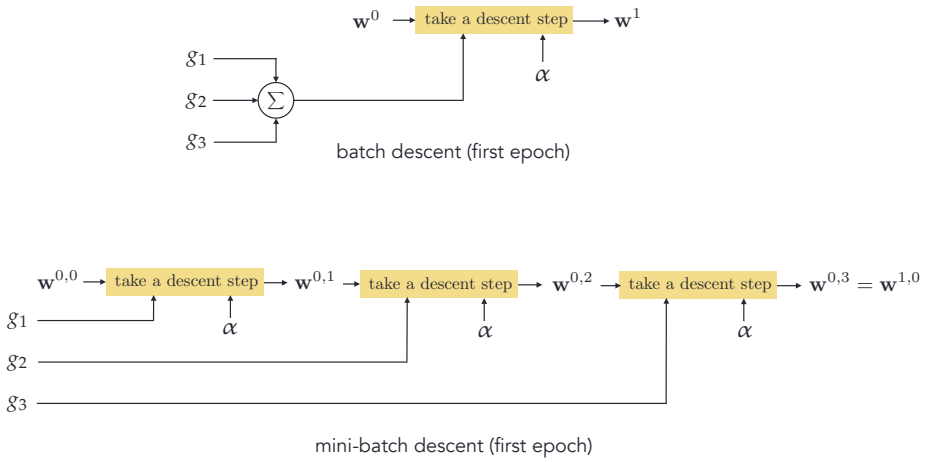

Figure A.7 An abstract illustration of the batch (top panel) and mini-batch descent approaches to local optimization. See text for further details.

Equation (A.33), 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

$$\mathbf{w}^1 = \mathbf{w}^0 + \alpha\, \mathbf{d}^0. \tag{A.34}$$

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 for g, and take a step in this direction as

$$\mathbf{w}^{0,1} = \mathbf{w}^{0,0} + \alpha\, \mathbf{d}^{0,1}. \tag{A.35}$$

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

$$\mathbf{w}^{0,2} = \mathbf{w}^{0,1} + \alpha\, \mathbf{d}^{0,2} \tag{A.36}$$

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

$$\begin{aligned}
\mathbf{w}^{0,1} &= \mathbf{w}^{0,0} + \alpha\, \mathbf{d}^{0,1} \\
\mathbf{w}^{0,2} &= \mathbf{w}^{0,1} + \alpha\, \mathbf{d}^{0,2} \\
&\vdots \\
\mathbf{w}^{0,p} &= \mathbf{w}^{0,p-1} + \alpha\, \mathbf{d}^{0,p} \\
&\vdots \\
\mathbf{w}^{0,P} &= \mathbf{w}^{0,P-1} + \alpha\, \mathbf{d}^{0,P}.
\end{aligned} \tag{A.37}$$

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 of the P functions we perform a second *epoch* of steps, and so on.

A.5.2 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 nonoverlapping subsets of the functions g_1, g_2, ..., g_P, but still sweep through each g_p exactly once per epoch.

The size/cardinality of the subsets 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.

A.5.3 General performance

Is the trade-off – taking more steps per epoch with a mini-batch approach as opposed to 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. This is particularly true when P is large, typically in the thousands or greater.

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) is shown in Figure A.8. 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 steps are taken during an epoch of mini-batch optimization the method often greatly outperforms its full batch analog (see, e.g., Exercise 7.11) – again, particularly when P is large.

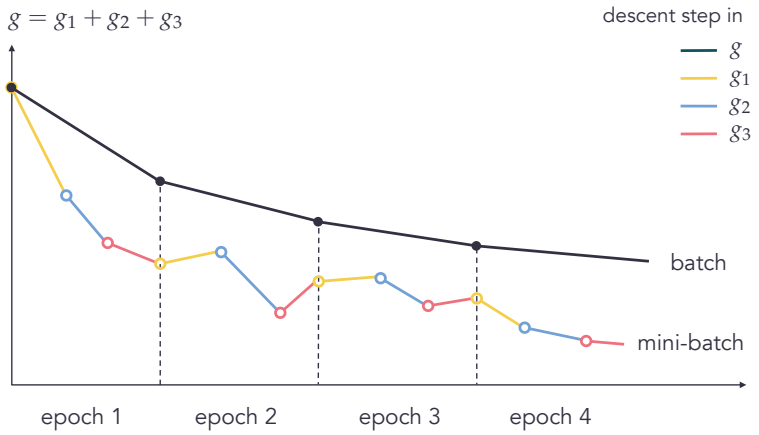

Figure A.8 A prototypical cost function history comparison of batch and mini-batch descent. Here $P = 3$. Epoch per epoch, the mini-batch approach tends to outperform the full batch step, reaching points nearer to local minima of functions of the form in Equation (A.33) faster than the full batch approach.

A.6 Conservative Steplength Rules

In Section 3.5 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, it is possible to derive proper steplength parameter settings mathematically that are guaranteed to produce convergence of the algorithm. These steplength choices are often quite *conservative*, specifically designed to force descent in the function at *every step*, and are therefore quite expensive computationally speaking. In this section we briefly review such steplength schemes for the sake of the interested reader.

A.6.1 Gradient descent and simple quadratic surrogates

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

$$h_\alpha(\mathbf{w}) = g\left(\mathbf{w}^0\right) + \nabla g\left(\mathbf{w}^0\right)^T\left(\mathbf{w} - \mathbf{w}^0\right) + \frac{1}{2\alpha}\left\|\mathbf{w} - \mathbf{w}^0\right\|_2^2 \tag{A.38}$$

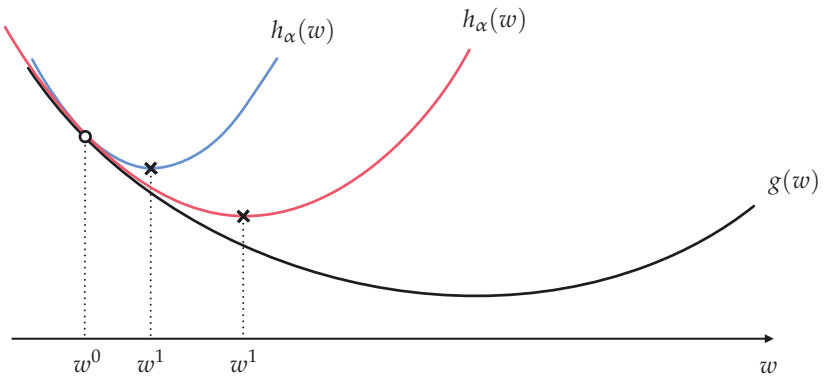

Figure A.9 Two quadratic functions approximating the function g around w^0 given by the quadratic approximation in Equation (A.38). The value of α is larger with the red quadratic than with the blue one.

where $\alpha > 0$. 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 that like the hyperplane, this quadratic is still tangent to $g(\mathbf{w})$ 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 A.9 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

$$\nabla h_\alpha(\mathbf{w}) = \nabla g\left(\mathbf{w}^0\right) + \frac{1}{\alpha}\left(\mathbf{w} - \mathbf{w}^0\right) = \mathbf{0}. \tag{A.39}$$

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

$$\mathbf{w}^1 = \mathbf{w}^0 - \alpha \nabla g\left(\mathbf{w}^0\right). \tag{A.40}$$

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

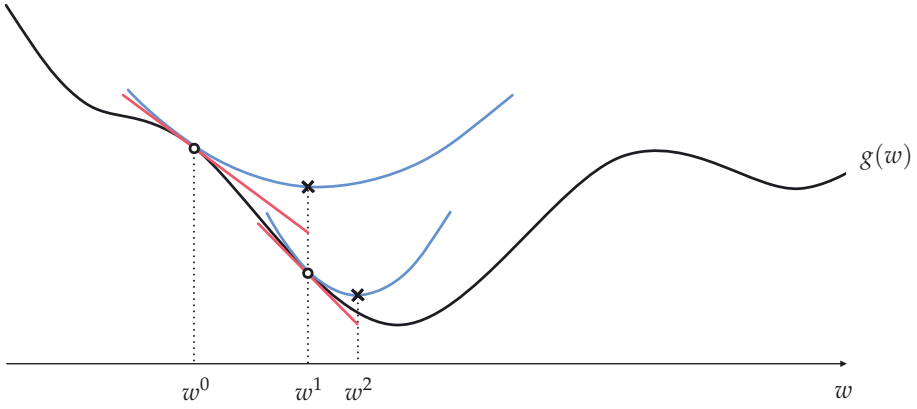

Figure A.10 Gradient descent can be viewed simultaneously as using either linear or simple quadratic surrogates to find a stationary point of g. At each step the associated steplength defines both how far along the linear surrogate we move before hopping back onto the function g, and at the same time the width of the simple quadratic surrogate which we minimize to reach the same point on g.

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

$$h_\alpha(\mathbf{w}) = g\left(\mathbf{w}^{k-1}\right) + \nabla g\left(\mathbf{w}^{k-1}\right)^T\left(\mathbf{w} - \mathbf{w}^{k-1}\right) + \frac{1}{2\alpha}\left\|\mathbf{w} - \mathbf{w}^{k-1}\right\|_2^2 \tag{A.41}$$

where the minimum is once again given as the kth gradient descent step

$$\mathbf{w}^k = \mathbf{w}^{k-1} - \alpha\nabla g\left(\mathbf{w}^{k-1}\right). \tag{A.42}$$

In sum, our exercise with the simple quadratic yields an alternative perspective on the standard gradient descent algorithm (detailed in Section 3.5): 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 prototypically in Figure A.10.

A.6.2 Backtracking line search

Since the negative gradient is a descent direction, if we are at a step $\mathbf{w}^{k-1}$ then – with a small enough α – the gradient descent step to $\mathbf{w}^k$ will decrease the value

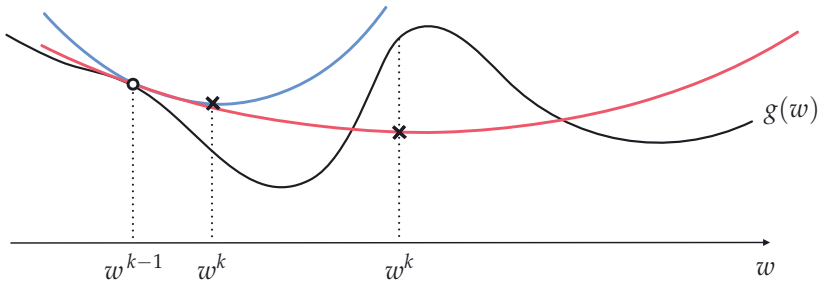

Figure A.11 An illustration of our second perspective on how to choose a value for the steplength parameter α that is guaranteed to decrease the underlying function's value by taking a single gradient descent step. The value of α should be decreased until its minimum lies over the function. A step to 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.

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 3.5) tells us that this will work because as we shrink α we are traveling a shorter distance 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 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 (A.41) (whose minimum is the point we will move to), so that the minimum point on the quadratic approximation lies *above* the function g. 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.

How can we find a value of α that does just this at the point $\mathbf{w}^{k-1}$? 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., $g\left(\mathbf{w}^k\right) \leq h_\alpha\left(\mathbf{w}^k\right)$, as illustrated in Figure A.11. 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, e.g., $\alpha = 1$, and a scalar "dampening factor" $t \in (0, 1)$.

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

Note that the inequality $g\left(\mathbf{w}^k\right) \leq h_\alpha\left(\mathbf{w}^k\right)$ can be written equivalently as

$$g\left(\mathbf{w}^k\right) \leq g\left(\mathbf{w}^{k-1}\right) - \frac{\alpha}{2}\left\|\nabla g\left(\mathbf{w}^{k-1}\right)\right\|_2^2 \tag{A.43}$$

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

$$\frac{\alpha}{2}\left\|\nabla g\left(\mathbf{w}^{k-1}\right)\right\|_2^2 \tag{A.44}$$

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 $g\left(\mathbf{w}^{k-1}\right)$.

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, the way the dampening factor $t \in (0, 1)$ controls how coarsely we sample the 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 steplength.

A.6.3 Exact line search

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

$$\underset{\alpha > 0}{\text{minimize}}\; g\left(\mathbf{w}^{k-1} - \alpha \nabla g\left(\mathbf{w}^{k-1}\right)\right). \tag{A.45}$$

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 previously – by successively examining smaller values until we find a value of α at the kth step $\mathbf{w}^k = \mathbf{w}^{k-1} - \alpha\nabla g\left(\mathbf{w}^{k-1}\right)$ such that

$$g\left(\mathbf{w}^k\right) \leq g\left(\mathbf{w}^{k-1}\right). \tag{A.46}$$

A.6.4 Conservatively optimal fixed steplength values

Suppose we construct the simple quadratic approximation of the form in Equation (A.41) 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 the *Lipschitz constant*, means that the quadratic approximation will lie completely above the function everywhere except at its point of tangency with the function at $\left(\mathbf{w}^{k-1}, g\left(\mathbf{w}^{k-1}\right)\right)$.

When α is set so that the entire quadratic itself lies above the function this means that in particular the quadratic's minimum lies above the function. In other words, our gradient descent step must lead to a smaller evaluation of g since

$$g\left(\mathbf{w}^k\right) < h_\alpha\left(\mathbf{w}^k\right) \leq h_\alpha\left(\mathbf{w}^{k-1}\right) = g\left(\mathbf{w}^{k-1}\right). \tag{A.47}$$

As detailed in Section 4.1, 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, i.e.,

$$\max_w \left|\frac{\mathrm{d}^2}{\mathrm{d}w^2} g(w)\right|. \tag{A.48}$$

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 C.5)

$$\max_{\mathbf{w}} \left\|\nabla^2 g\left(\mathbf{w}\right)\right\|_2. \tag{A.49}$$

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

Once determined this maximum curvature L – or an upper bound on it – gives a fixed[3] steplength $\alpha = \frac{1}{L}$ that can be used so that the kth descent step

$$\mathbf{w}^k = \mathbf{w}^{k-1} - \frac{1}{L}\nabla g\left(\mathbf{w}^{k-1}\right) \tag{A.51}$$

is guaranteed to always descend in the function g.[4] 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.

This conservatively optimal steplength can be a very convenient rule to use in practice. However, as its name implies, it is indeed a conservative rule by nature. Therefore, in practice, one should use it as a benchmark to search for larger convergence-forcing fixed steplength values. In other words, with the steplength $\alpha = \frac{1}{L}$ calculated one can easily test larger steplengths of the form $\alpha = \frac{t}{L}$ for any constant $t > 1$. Indeed depending on the problem values of t ranging from 1 to 100 can work well in practice.

Example A.7 Computing the Lipschitz constant of a single-input sinusoid

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

$$g(w) = \sin(w). \tag{A.55}$$

[3] If we use *local* instead of *global* curvature to define the steplength our corresponding step will take the form

$$\mathbf{w}^k = \mathbf{w}^{k-1} - \frac{1}{\left\|\nabla^2 g\left(\mathbf{w}^{k-1}\right)\right\|_2}\nabla g\left(\mathbf{w}^{k-1}\right) \tag{A.50}$$

which we can interpret as a gradient descent step with self-adjusting steplength (that adjusts itself based on the local curvature of the function g). Newton's method – as discussed in Section 4.3 – can be thought of as an extension of this idea.

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

$$h_{\frac{1}{L}}(\mathbf{w}) = g\left(\mathbf{w}^{k-1}\right) + \nabla g\left(\mathbf{w}^{k-1}\right)^T\left(\mathbf{w} - \mathbf{w}^{k-1}\right) + \frac{L}{2}\left\|\mathbf{w} - \mathbf{w}^{k-1}\right\|_2^2 \tag{A.52}$$

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

$$g(\mathbf{w}) = g\left(\mathbf{w}^{k-1}\right) + \nabla g\left(\mathbf{w}^{k-1}\right)^T\left(\mathbf{w} - \mathbf{w}^{k-1}\right) + \frac{1}{2}\left(\mathbf{w} - \mathbf{w}^{k-1}\right)^T\nabla^2 g(\mathbf{c})\left(\mathbf{w} - \mathbf{w}^{k-1}\right) \tag{A.53}$$

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 have

$$\mathbf{a}^T\nabla^2 g(\mathbf{c})\,\mathbf{a} \leq L\|\mathbf{a}\|_2^2 \tag{A.54}$$

for all $\mathbf{a}$, and in particular for $\mathbf{a} = \mathbf{w} - \mathbf{w}^{k-1}$, which implies $g(\mathbf{w}) \leq h_{\frac{1}{L}}(\mathbf{w})$.

We can easily compute the second derivative of this function as

$$\frac{\mathrm{d}^2}{\mathrm{d}w^2} g(w) = -\sin(w). \tag{A.56}$$

The maximum value this (second derivative) function can take is 1, hence $L = 1$, and therefore $\alpha = \frac{1}{L} = 1$ guarantees descent at every step.

Example A.8 Computing the Lipschitz constant of a multi-input quadratic
In this example we look at computing the Lipschitz constant of the quadratic function

$$g(\mathbf{w}) = a + \mathbf{b}^T\mathbf{w} + \mathbf{w}^T\mathbf{C}\mathbf{w} \tag{A.57}$$

where $a = 1$, $\mathbf{b} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$, and $\mathbf{C} = \begin{bmatrix} 2 & 0 \\ 0 & 1 \end{bmatrix}$.

Here the Hessian is simply $\nabla^2 g(\mathbf{w}) = \mathbf{C} + \mathbf{C}^T = 2\mathbf{C}$ 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 we can set $L = 4$, giving a conservative optimal steplength value of $\alpha = \frac{1}{4}$.

A.6.5 Convergence proofs

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.

3. They have bounded curvature.

Gradient descent with fixed Lipschitz steplength

With the gradient of g being Lipschitz continuous with constant L, from Section A.6.4 we know that at the kth iteration of gradient descent we have a corresponding quadratic upper bound on g of the form

$$g(\mathbf{w}) \leq g\left(\mathbf{w}^{k-1}\right) + \nabla g\left(\mathbf{w}^{k-1}\right)^T\left(\mathbf{w} - \mathbf{w}^{k-1}\right) + \frac{L}{2}\|\mathbf{w} - \mathbf{w}^{k-1}\|_2^2 \tag{A.58}$$

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

$$g\left(\mathbf{w}^k\right) \leq g\left(\mathbf{w}^{k-1}\right) - \frac{1}{2L}\|\nabla g\left(\mathbf{w}^{k-1}\right)\|_2^2 \tag{A.59}$$

which, since $\|\nabla g\left(\mathbf{w}^{k-1}\right)\|_2^2 \geq 0$, indeed shows that the sequence of gradient steps is decreasing. To show that it converges to a stationary point where the gradient vanishes we subtract $g\left(\mathbf{w}^{k-1}\right)$ from both sides of Equation (A.59), and sum the result over $1 \leq k \leq K$, giving

$$\sum_{k=1}^{K}\left[g\left(\mathbf{w}^k\right) - g\left(\mathbf{w}^{k-1}\right)\right] = g\left(\mathbf{w}^K\right) - g\left(\mathbf{w}^0\right) \leq -\frac{1}{2L}\sum_{k=1}^{K}\left\|\nabla g\left(\mathbf{w}^{k-1}\right)\right\|_2^2. \tag{A.60}$$

Note importantly here that since g is bounded from below, taking $K \longrightarrow \infty$, we *must* have that

$$\sum_{k=1}^{\infty}\left\|\nabla g\left(\mathbf{w}^{k-1}\right)\right\|_2^2 < \infty. \tag{A.61}$$

Hence the fact that the infinite sum above must be finite implies that as $k \longrightarrow \infty$ we have that

$$\left\|\nabla g\left(\mathbf{w}^{k-1}\right)\right\|_2^2 \longrightarrow 0 \tag{A.62}$$

meaning that the sequence of gradient descent steps with steplength determined by the Lipschitz constant of the gradient of g produces a sequence with vanishing gradient that converges to a stationary point of g. Note that we could have made the same argument above using any fixed steplength smaller than $\frac{1}{L}$ as well.

Gradient descent with backtracking line search

With the assumption that g has a maximum bounded curvature of L, it follows that with any fixed choice of initial steplength $\alpha > 0$ and $t \in (0, 1)$ we can always find an integer n_0 such that

$$t^{n_0}\alpha \leq \frac{1}{L}. \tag{A.63}$$

Thus the backtracking-found steplength at the kth gradient descent step will always be larger than this lower bound, i.e.,

$$\alpha_k \geq t^{n_0}\alpha > 0 \tag{A.64}$$

for all k.

Recall from Equation (A.43) that by running the backtracking procedure at the kth gradient step we have

$$g\left(\mathbf{w}^k\right) \leq g\left(\mathbf{w}^{k-1}\right) - \frac{\alpha_k}{2}\left\|\nabla g\left(\mathbf{w}^{k-1}\right)\right\|_2^2. \tag{A.65}$$

To show that the sequence of gradient steps converges to a stationary point of g we first subtract $g\left(\mathbf{w}^{k-1}\right)$ from both sides of Equation (A.65), and sum the result over $1 \leq k \leq K$, which gives

$$\sum_{k=1}^{K}\left[g\left(\mathbf{w}^k\right) - g\left(\mathbf{w}^{k-1}\right)\right] = g\left(\mathbf{w}^K\right) - g\left(\mathbf{w}^0\right) \leq -\frac{1}{2}\sum_{k=1}^{K}\alpha_k\left\|\nabla g\left(\mathbf{w}^{k-1}\right)\right\|_2^2. \tag{A.66}$$

Since g is bounded from below, taking $K \longrightarrow \infty$, we must have

$$\sum_{k=1}^{\infty}\alpha_k\left\|\nabla g\left(\mathbf{w}^{k-1}\right)\right\|_2^2 < \infty. \tag{A.67}$$

Now, we know from Equation (A.64) that

$$\sum_{k=1}^{K}\alpha_k \geq K\,t^{n_0}\alpha \tag{A.68}$$

implying

$$\sum_{k=1}^{\infty}\alpha_k = \infty. \tag{A.69}$$

In order for Equations (A.67) and (A.69) to hold simultaneously we *must* have that

$$\left\|\nabla g\left(\mathbf{w}^{k-1}\right)\right\|_2^2 \longrightarrow 0 \tag{A.70}$$

as $k \longrightarrow \infty$. This shows that the sequence of gradient steps determined by backtracking line search converges to a stationary point of g.

A.7 Newton's Method, Regularization, and Nonconvex Functions

As we saw in Section 4.3, Newton's method is naturally incapable of properly minimizing generic nonconvex functions. In this section we describe the regularized Newton step as a common approach to ameliorate this particular issue (which we have essentially seen before albeit without the more in-depth context we provide here).

A.7.1 Turning up ϵ

In Section 4.3.3 we saw how adding a very small positive value ϵ to the second derivative of a single-input function, or analogously a weighted identity matrix of the form $\epsilon \mathbf{I}_{N\times N}$ to the Hessian in the multi-input case, helps Newton's method avoid numerical problems in flat regions of a convex function. This adjusted Newton step, which takes the form

$$\mathbf{w}^k = \mathbf{w}^{k-1} - \left(\nabla^2 g(\mathbf{w}^{k-1}) + \epsilon \mathbf{I}_{N\times N}\right)^{-1} \nabla g(\mathbf{w}^{k-1}), \tag{A.71}$$

can be interpreted as the stationary point of a slightly adjusted second-order Taylor series approximation centered at $\mathbf{w}^{k-1}$

$$h(\mathbf{w}) = g(\mathbf{w}^{k-1}) + \nabla g(\mathbf{w}^{k-1})^T(\mathbf{w} - \mathbf{w}^{k-1}) + \frac{1}{2}(\mathbf{w} - \mathbf{w}^{k-1})^T \nabla^2 g\left(\mathbf{w}^{k-1}\right)(\mathbf{w} - \mathbf{w}^{k-1}) + \frac{\epsilon}{2}\left\|\mathbf{w} - \mathbf{w}^{k-1}\right\|_2^2. \tag{A.72}$$

The first three terms (on the right-hand side of the equality) still represent the second-order Taylor series at $\mathbf{w}^{k-1}$, and to it we have added $\frac{\epsilon}{2}\left\|\mathbf{w} - \mathbf{w}^{k-1}\right\|_2^2$, a *convex* and perfectly symmetric quadratic centered at $\mathbf{w}^{k-1}$ with N positive eigenvalues (each equal to $\frac{\epsilon}{2}$). In other words, we have a sum of two quadratic functions. When $\mathbf{w}^{k-1}$ is at a nonconvex or flat portion of a function the first quadratic is likewise nonconvex or flat. However, the second one is *always* convex, and the larger ϵ is set the greater its (convex) curvature. This means that if we set ϵ larger we can *convexify* the entire approximation, forcing the stationary point we solve for to be a minimum and the direction in which we travel is one of guaranteed descent.

Example A.9 The effect of regularization

In Figure A.12 we illustrate the regularization of a nonconvex function

$$h_1(w_1, w_2) = w_1^2 - w_2^2 \tag{A.73}$$

using the convex regularizer

$$h_2(w_1, w_2) = w_1^2 + w_2^2. \tag{A.74}$$

In particular we show what the resulting sum $h_1 + \epsilon h_2$ looks like over four progressively increasing values of ϵ, from $\epsilon = 0$ (leftmost panel) to $\epsilon = 2$ (rightmost panel).

Since h_1 is nonconvex, and has a single stationary point that is a saddle point at the origin, the addition of h_2 pulls up its downward-facing dimension. Not

surprisingly as ϵ is increased, the shape of the sum is dictated more and more by h_2. Eventually, turning up ϵ sufficiently, the sum becomes convex.

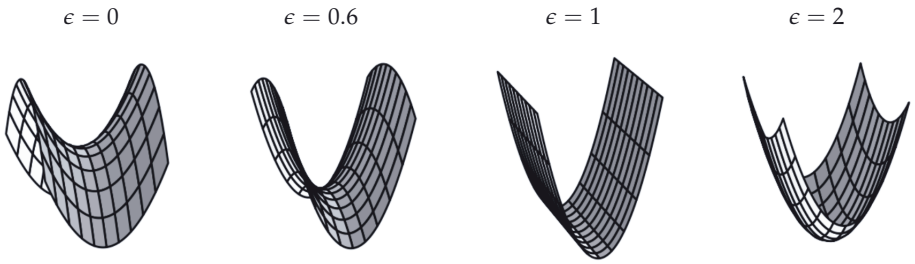

Figure A.12 Figure associated with Example A.9. From left to right, a nonconvex quadratic function is slowly turned into a convex function via the weighted addition of a convex quadratic. See text for further details.

Example A.10 Minimization of a nonconvex function

In Figure A.13 we illustrate five regularized Newton steps (using the update step shown in Equation (A.71)) to minimize the nonconvex function

$$g(w) = 2 - e^{-w^2}. \tag{A.75}$$

We initialize the algorithm at a point of local nonconvexity for this function, and gradually increase ϵ from $\epsilon = 0$ (top panel) where Newton's method diverges, to $\epsilon = 4$ (bottom panel) where it has sufficiently *convexified* the Newton step, leading the algorithm to take downhill steps towards the minimum of the function. In the figure the steps from each run are colored from green (the first step) to red (the final step), with the regularized second-order approximation at each step colored accordingly.

To determine how high we need to turn up ϵ in order to make the regularized second-order Taylor series approximation convex, recall from Section 4.2 that a quadratic function is convex *if and only if* it has all nonnegative eigenvalues. Thus ϵ must be made larger than the magnitude of the smallest eigenvalue of the Hessian $\nabla^2 g\left(\mathbf{w}^{k-1}\right)$ in order for the regularized second-order quadratic to be convex. For a single-input function this reduces to ϵ being larger in magnitude than the value of the function's second derivative at w^{k-1} (if it is negative there).

While ϵ in the regularized Newton step in Equation (A.71) is typically set to a relatively small value, it is interesting to note that as we increase ϵ, the direction we travel in tilts toward the gradient descent direction at $\mathbf{w}^{k-1}$. In other words, when ϵ is large the direction we travel when taking the regularized Newton step becomes the gradient descent direction (albeit with a very small magnitude)

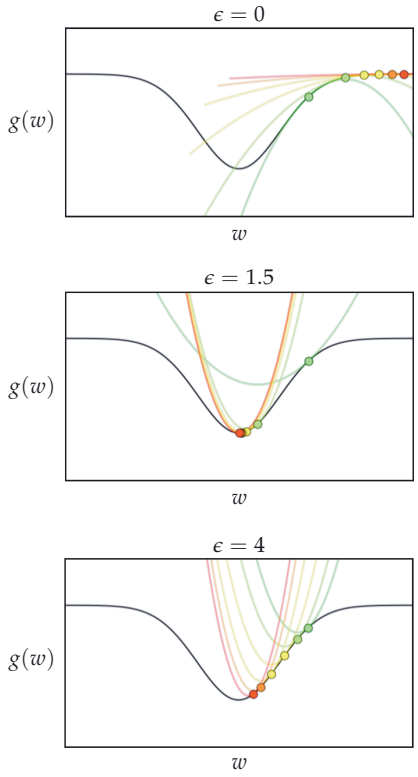

Figure A.13 Figure associated with Example A.10. See text for details.

$$-\left(\nabla^2 g(\mathbf{w}^{k-1}) + \epsilon \mathbf{I}_{N\times N}\right)^{-1} \nabla g(\mathbf{w}^{k-1}) \approx -\left(\epsilon \mathbf{I}_{N\times N}\right)^{-1} \nabla g(\mathbf{w}^{k-1}) = -\frac{1}{\epsilon}\nabla g(\mathbf{w}^{k-1}). \tag{A.76}$$

A.8 Hessian-Free Methods

While Newton's method is a powerful technique that exhibits rapid convergence due to its employment of second-order information, it is naturally constrained by the input dimension N of a general function $g(\mathbf{w})$. More specifically, the $N\times N$ Hessian matrix $\nabla^2 g(\mathbf{w})$, with its N^2 entries, naturally limits Newton's method's use to cases where N is (roughly speaking) in the thousands, since it is difficult to even store such a matrix when N is larger (let alone compute with it).

In this section we discuss two variations on Newton's method, collectively referred to as *Hessian-free* optimization methods, that ameliorate this issue by replacing the Hessian (in each Newton's method step) with a close approximation that does not suffer from the same scaling issue. Because of this, both approaches naturally trade the precision of each Newton's method step with the

ability to scale the basic algorithm to high-dimensional input. The first of these approaches is the simplest conceptually speaking, and involves *subsampling* the Hessian, using only a fraction of its entries. The latter method, often referred to as *quasi-Newton*, involves replacing the Hessian with a *low-rank* approximation that can be computed effectively.

A.8.1 Subsampling the Hessian

The simplest way to deal with the scaling issue inherent with Newton's method, that of the massive number of entries in the $N \times N$ Hessian matrix $\nabla^2 g(\mathbf{w})$ as N increases, is to simply *subsample* the Hessian. That is, instead of using the entire Hessian matrix we use only a fraction of its entries, setting the remainder of the entries to zero. This of course dilutes the power of complete second-order information leveraged at each Newton's step, and thus the corresponding efficacy of each corresponding Newton's step, but salvages the otherwise untenable method when N is too large. There are a variety of ways one can consider subsampling the Hessian that trade-off between maintaining second-order information and the allowance of graceful scaling with input dimension (see, e.g., [15, 71]).

One popular subsampling scheme involves simply retaining only the diagonal entries of the Hessian matrix, i.e., only the N pure partial second derivatives of the form $\frac{\partial^2}{\partial w_n^2} g(w)$ for $n = 1, 2, ..., N$. This drastically reduces the number of entries of the Hessian, and greatly simplifies the Newton's step from the solution to a linear system of equations

$$\mathbf{w}^k = \mathbf{w}^{k-1} - \left(\nabla^2 g(\mathbf{w}^{k-1})\right)^{-1} \nabla g(\mathbf{w}^{k-1}) \tag{A.77}$$

to the straightforward component-wise update for each $n = 1, 2, ..., N$

$$w_n^k = w_n^{k-1} - \frac{\frac{\partial}{\partial w_n} g(\mathbf{w}^{k-1})}{\frac{\partial^2}{\partial w_n^2} g(\mathbf{w}^{k-1})}. \tag{A.78}$$

In other words, keeping only the diagonal entries of the Hessian we decouple each coordinate and no longer have a system of equations to solve. The downside, of course, is that we ignore all cross-partial derivatives retaining only the second-order information corresponding to the curvature along each input dimension independently. Nonetheless, this subsampled Newton's step can be quite effective in practice when used to minimize machine learning cost functions (see, e.g., Exercise 9.8) and can scale as gracefully as a first-order method like gradient descent.

A.8.2 Secant methods

In studying Newton's method as a zero-finding algorithm, at the kth step in finding a zero of the first-order equation $\frac{\mathrm{d}}{\mathrm{d}w}g(w) = 0$ we form the first-order Taylor series

$$h(w) = \frac{\mathrm{d}}{\mathrm{d}w}g\left(w^{k-1}\right) + \frac{\mathrm{d}^2}{\mathrm{d}w^2}g\left(w^{k-1}\right)\left(w - w^{k-1}\right) \tag{A.79}$$

and find where this linear function equals zero, which is given by the corresponding Newton update

$$w^k = w^{k-1} - \frac{\frac{\mathrm{d}}{\mathrm{d}w}g\left(w^{k-1}\right)}{\frac{\mathrm{d}^2}{\mathrm{d}w^2}g\left(w^{k-1}\right)}. \tag{A.80}$$

If we replace the slope of the tangent line – here the second derivative evaluation $\frac{\mathrm{d}^2}{\mathrm{d}w^2}g\left(w^{k-1}\right)$ – with the slope provided by a closely related[5] *secant line*

$$\frac{\mathrm{d}^2}{\mathrm{d}w^2}g\left(w^{k-1}\right) \approx \frac{\frac{\mathrm{d}}{\mathrm{d}w}g\left(w^{k-1}\right) - \frac{\mathrm{d}}{\mathrm{d}w}g\left(w^{k-2}\right)}{w^{k-1} - w^{k-2}} \tag{A.81}$$

we will produce an algorithm highly related to Newton's method (but with no need to employ the second derivative). Replacing the second derivative with this approximation in our Newton's step we have a *secant method* update

$$w^k = w^{k-1} - \frac{\frac{\mathrm{d}}{\mathrm{d}w}g\left(w^{k-1}\right)}{\frac{\frac{\mathrm{d}}{\mathrm{d}w}g\left(w^{k-1}\right) - \frac{\mathrm{d}}{\mathrm{d}w}g\left(w^{k-2}\right)}{w^{k-1} - w^{k-2}}} \tag{A.82}$$

which we can write in a less cumbersome manner as

$$w^k = w^{k-1} - \frac{\frac{\mathrm{d}}{\mathrm{d}w}g\left(w^{k-1}\right)}{s^{k-1}} \tag{A.83}$$

where s^{k-1} has been used to denote the slope of the secant line

$$s^{k-1} = \frac{\frac{\mathrm{d}}{\mathrm{d}w}g\left(w^{k-1}\right) - \frac{\mathrm{d}}{\mathrm{d}w}g\left(w^{k-2}\right)}{w^{k-1} - w^{k-2}}. \tag{A.84}$$

While less accurate than Newton's method this approach, which does not rely directly on the second derivative, can still be generally used to solve the first-order equation and find stationary points of the function $g(w)$. This fact, while fairly inconsequential for a single-input function with $N = 1$, gains significantly

[5] Remember that the derivative of a single-input function defines the slope of the tangent line to the function at the point of tangency. This slope can be roughly approximated as the slope of a nearby *secant* line, that is a line that passes through the same point as well as another point nearby on the function (see Section B.2.1).

more value when this secant method is generalized to multi-input functions. This is because, as we have already discussed, it is the very size of the Hessian matrix that prohibits Newton's method's use for functions with large values of N.

Everything we have discussed for the generic single-input case tracks to the multi-input instance as well. Notice, looking back at Equation (A.84), that by multiplying both sides by $w^{k-1} - w^{k-2}$ we can write it equivalently as

$$s^{k-1}\left(w^{k-1} - w^{k-2}\right) = \frac{\mathrm{d}}{\mathrm{d}w}g\left(w^{k-1}\right) - \frac{\mathrm{d}}{\mathrm{d}w}g\left(w^{k-2}\right). \tag{A.85}$$

This is often referred to as the single-input *secant condition*.
Replacing each component of Equation (A.85) with its multi-input analog gives the multi-input secant condition

$$\mathbf{S}^{k-1}\left(\mathbf{w}^{k-1} - \mathbf{w}^{k-2}\right) = \nabla g\left(\mathbf{w}^{k-1}\right) - \nabla g\left(\mathbf{w}^{k-2}\right). \tag{A.86}$$

Here we have replaced the scalar s^{k-1} with its analog, an $N \times N$ matrix $\mathbf{S}^{k-1}$, and the one-dimensional terms w^{k-1}, w^{k-2}, $\frac{\mathrm{d}}{\mathrm{d}w}g\left(w^{k-1}\right)$, and $\frac{\mathrm{d}}{\mathrm{d}w}g\left(w^{k-2}\right)$ with their respective N-dimensional analogs: $\mathbf{w}^{k-1}$, $\mathbf{w}^{k-2}$, $\nabla g\left(\mathbf{w}^{k-1}\right)$, and $\nabla g\left(\mathbf{w}^{k-2}\right)$. Assuming for a moment that $\mathbf{S}^{k-1}$ is invertible, we can also express the secant condition as

$$\mathbf{w}^{k-1} - \mathbf{w}^{k-2} = \left(\mathbf{S}^{k-1}\right)^{-1}\left(\nabla g\left(\mathbf{w}^{k-1}\right) - \nabla g\left(\mathbf{w}^{k-2}\right)\right). \tag{A.87}$$

In either instance, we can see that the secant method requires we *solve* for the matrix $\mathbf{S}^{k-1}$ or its inverse $\left(\mathbf{S}^{k-1}\right)^{-1}$. Unlike the one-dimensional instance of the secant condition in Equation (A.85) where each update has a unique solution, with the N-dimensional case in Equation (A.86) we must solve a system of equations which will generally have infinitely many solutions, since there are only N equations but N^2 entries to solve for in the matrix $\mathbf{S}^{k-1}$.

A.8.3 Quasi-Newton methods

As discussed in Section 4.3.2, the standard Newton step

$$\mathbf{w}^k = \mathbf{w}^{k-1} - \left(\nabla^2 g\left(\mathbf{w}^{k-1}\right)\right)^{-1}\nabla g\left(\mathbf{w}^{k-1}\right) \tag{A.88}$$

is an example of a local optimization step of the generic form $\mathbf{w}^k = \mathbf{w}^{k-1} + \alpha\mathbf{d}^k$, with the descent direction given by

$$\mathbf{d}^k = -\left(\nabla^2 g\left(\mathbf{w}^{k-1}\right)\right)^{-1}\nabla g\left(\mathbf{w}^{k-1}\right). \tag{A.89}$$

The term "quasi-Newton" method is the jargon phrase used for any descent step of the form

$$\mathbf{d}^k = -\left(\mathbf{S}^{k-1}\right)^{-1} \nabla g\left(\mathbf{w}^{k-1}\right) \tag{A.90}$$

where the true Hessian matrix $\nabla^2 g\left(\mathbf{w}^{k-1}\right)$ is replaced with a secant approximation $\mathbf{S}^{k-1}$. As with the single-input case this kind of update – while less accurate at each step than the true Newton's method – can still define an effective local optimization method depending on how $\mathbf{S}^{k-1}$ is constructed (all while avoiding the need for direct second-order information).

In other words, in taking quasi-Newton steps we are no longer required to compute a sequence of Hessian matrices

$$\nabla^2 g\left(\mathbf{w}^0\right),\ \nabla^2 g\left(\mathbf{w}^1\right),\ \nabla^2 g\left(\mathbf{w}^2\right),\ \text{etc.}, \tag{A.91}$$

and instead we construct a sequence of corresponding secant matrices

$$\mathbf{S}^1,\ \mathbf{S}^2,\ \mathbf{S}^3,\ \text{etc.}, \tag{A.92}$$

as an approximation to the Hessian sequence. To construct this secant sequence, note the following.

- $\mathbf{S}^{k-1}$ **should be a solution to the secant condition.** Defining for notational convenience $\mathbf{a}^k = \mathbf{w}^{k-1} - \mathbf{w}^{k-2}$ and $\mathbf{b}^k = \nabla g\left(\mathbf{w}^{k-1}\right) - \nabla g\left(\mathbf{w}^{k-2}\right)$, the secant condition in Equation (A.86) dictates we must have that $\mathbf{S}^{k-1}\mathbf{a}^k = \mathbf{b}^k$. Letting $\mathbf{F}^k = \left(\mathbf{S}^k\right)^{-1}$ we can write this equivalently as $\mathbf{a}^k = \mathbf{F}^k\mathbf{b}^k$.

- $\mathbf{S}^{k-1}$ **should be symmetric.** Since the Hessian $\nabla^2 g\left(\mathbf{w}^{k-1}\right)$ is always symmetric and ideally we want $\mathbf{S}^{k-1}$ to mimic the Hessian closely, it is reasonable to expect $\mathbf{S}^{k-1}$ to be symmetric as well.

- **The secant sequence should converge.** As the quasi-Newton method progresses, the sequence of steps $\mathbf{w}^{k-1}$ should converge (to a minimum of g), so too should the secant sequence $\mathbf{S}^{k-1}$ (to the Hessian evaluated at that minimum).

We now explore, by example, a number of ways to construct secant sequences that satisfy these conditions. These constructions are *recursive* in nature and take the general form

$$\mathbf{F}^k = \mathbf{F}^{k-1} + \mathbf{D}^{k-1} \tag{A.93}$$

where the $N \times N$ *difference* matrix $\mathbf{D}^{k-1}$ is designed to be *symmetric*, of a particular *low rank*, and *diminishing* in magnitude.

Symmetry of $\mathbf{D}^{k-1}$ guarantees if we initialize the very first $\mathbf{F}^0$ to a symmetric matrix (most commonly the identity matrix), this recursive update will retain our desired symmetry property (since the sum of two symmetric matrices is

always symmetric). Similarly, if $\mathbf{D}^{k-1}$ is designed to be positive-definite and the initialization $\mathbf{F}^0$ is also positive-definite (as the identity matrix is) then this property is inherited by all matrices $\mathbf{F}^k$. Constraining $\mathbf{D}^{k-1}$ to be of low rank makes it structurally simple and allows to compute it in closed form at each step. Finally, the magnitude/norm of $\mathbf{D}^{k-1}$ should shrink as k gets larger, otherwise $\mathbf{F}^k$ simply will not converge.

Example A.11 Rank-1 difference matrix

In this example we describe one of the simplest recursive formula for $\mathbf{S}^k$ (or more precisely its inverse $\mathbf{F}^k$) where the difference matrix $\mathbf{D}^{k-1}$ in Equation (A.93) is a rank-1 outer-product matrix

$$\mathbf{D}^{k-1} = \mathbf{u}\,\mathbf{u}^T. \tag{A.94}$$

By first assuming this form of the difference matrix does indeed satisfy the secant condition, we can then work *backwards* from it to determine the proper value for $\mathbf{u}$.

Substituting $\mathbf{F}^{k-1} + \mathbf{u}\,\mathbf{u}^T$ for $\mathbf{F}^k$ into the secant condition gives

$$\left(\mathbf{F}^{k-1} + \mathbf{u}\,\mathbf{u}^T\right)\mathbf{b}^k = \mathbf{a}^k \tag{A.95}$$

or rearranging equivalently

$$\mathbf{u}\,\mathbf{u}^T\mathbf{b}^k = \mathbf{a}^k - \mathbf{F}^{k-1}\mathbf{b}^k. \tag{A.96}$$

Multiplying both sides by $\left(\mathbf{b}^k\right)^T$

$$\left(\mathbf{b}^k\right)^T \mathbf{u}\,\mathbf{u}^T\mathbf{b}^k = \left(\mathbf{b}^k\right)^T \mathbf{a}^k - \left(\mathbf{b}^k\right)^T \mathbf{F}^{k-1}\mathbf{b}^k \tag{A.97}$$

and taking the square root of both sides gives

$$\mathbf{u}^T\mathbf{b}^k = \left(\left(\mathbf{b}^k\right)^T \mathbf{a}^k - \left(\mathbf{b}^k\right)^T \mathbf{F}^{k-1}\mathbf{b}^k\right)^{\frac{1}{2}}. \tag{A.98}$$

Substituting the value for $\mathbf{u}^T\mathbf{b}^k$ from Equation (A.98) into Equation (A.96), we arrive at the desired form for the vector $\mathbf{u}$

$$\mathbf{u} = \frac{\mathbf{a}^k - \mathbf{F}^{k-1}\mathbf{b}^k}{\left(\left(\mathbf{b}^k\right)^T \mathbf{a}^k - \left(\mathbf{b}^k\right)^T \mathbf{F}^{k-1}\mathbf{b}^k\right)^{\frac{1}{2}}} \tag{A.99}$$

with the corresponding recursive formula for $\mathbf{F}^k$ given as

$$\mathbf{F}^k = \mathbf{F}^{k-1} + \frac{\left(\mathbf{a}^k - \mathbf{F}^{k-1}\mathbf{b}^k\right)\left(\mathbf{a}^k - \mathbf{F}^{k-1}\mathbf{b}^k\right)^T}{\left(\mathbf{b}^k\right)^T \mathbf{a}^k - \left(\mathbf{b}^k\right)^T \mathbf{F}^{k-1}\mathbf{b}^k}. \tag{A.100}$$

Example A.12 The Davidon–Fletcher–Powell (DFP) method
We can use a slightly more complex structure for the difference matrix by constructing it as a sum of two rank-1 matrices

$$\mathbf{D}^{k-1} = \mathbf{u}\,\mathbf{u}^T + \mathbf{v}\,\mathbf{v}^T. \tag{A.101}$$

By allowing for a rank-2 difference between the subsequent matrices (as opposed to a rank-1 difference) we encode an additional level of complexity into our approximation to subsequent inverse Hessian evaluations.

In order to determine the proper values for $\mathbf{u}$ and $\mathbf{v}$, we substitute $\mathbf{F}^{k-1} + \mathbf{D}^{k-1}$ for $\mathbf{F}^k$ into the secant condition (as we did in Example A.11), giving

$$\left(\mathbf{F}^{k-1} + \mathbf{u}\,\mathbf{u}^T + \mathbf{v}\,\mathbf{v}^T\right)\mathbf{b}^k = \mathbf{a}^k, \tag{A.102}$$

or rearranging equivalently

$$\mathbf{u}\,\mathbf{u}^T\mathbf{b}^k + \mathbf{v}\,\mathbf{v}^T\mathbf{b}^k = \mathbf{a}^k - \mathbf{F}^{k-1}\mathbf{b}^k. \tag{A.103}$$

Note that here we have only a *single* equation, and so there are infinitely many choices for our *two* unknown vectors $\mathbf{u}$ and $\mathbf{v}$. A very simple yet common way of determining a single set of values for $\mathbf{u}$ and $\mathbf{v}$ is to suppose the first/second term on the left-hand side of Equation (A.103) equals the corresponding term on the right-hand side, that is

$$\mathbf{u}\,\mathbf{u}^T\mathbf{b}^k = \mathbf{a}^k \qquad \text{and} \qquad \mathbf{v}\,\mathbf{v}^T\mathbf{b}^k = -\mathbf{F}^{k-1}\mathbf{b}^k. \tag{A.104}$$

This added assumption allows us to solve for a valid pair of $\mathbf{u}$ and $\mathbf{v}$ in a way that closely mirrors the solution method from Example A.11. First, we multiply each by $\left(\mathbf{b}^k\right)^T$, giving

$$\left(\mathbf{b}^k\right)^T \mathbf{u}\,\mathbf{u}^T\mathbf{b}^k = \left(\mathbf{b}^k\right)^T \mathbf{a}^k \qquad \text{and} \qquad \left(\mathbf{b}^k\right)^T \mathbf{v}\,\mathbf{v}^T\mathbf{b}^k = -\left(\mathbf{b}^k\right)^T \mathbf{F}^{k-1}\mathbf{b}^k. \tag{A.105}$$

Taking the square root of both sides in both equations, we then have the set of equations

$$\mathbf{u}^T\mathbf{b}^k = \left(\left(\mathbf{b}^k\right)^T \mathbf{a}^k\right)^{\frac{1}{2}} \qquad \text{and} \qquad \mathbf{v}^T\mathbf{b}^k = \left(-\left(\mathbf{b}^k\right)^T \mathbf{F}^{k-1}\mathbf{b}^k\right)^{\frac{1}{2}}. \tag{A.106}$$

Substituting these value for $\mathbf{u}^T\mathbf{b}^k$ and $\mathbf{v}^T\mathbf{b}^k$ from Equation (A.106) into Equation (A.104), we have

$$\mathbf{u} = \frac{\mathbf{a}^k}{\left(\left(\mathbf{b}^k\right)^T \mathbf{a}^k\right)^{\frac{1}{2}}} \qquad \text{and} \qquad \mathbf{v} = \frac{-\mathbf{F}^{k-1}\mathbf{b}^k}{\left(-\left(\mathbf{b}^k\right)^T \mathbf{F}^{k-1}\mathbf{b}^k\right)^{\frac{1}{2}}}. \tag{A.107}$$

with the corresponding recursive formula for $\mathbf{F}^k$ given as

$$\mathbf{F}^k = \mathbf{F}^{k-1} + \frac{\mathbf{a}^k\left(\mathbf{a}^k\right)^T}{\left(\mathbf{b}^k\right)^T \mathbf{a}^k} - \frac{\left(\mathbf{F}^{k-1}\mathbf{b}^k\right)\left(\mathbf{F}^{k-1}\mathbf{b}^k\right)^T}{\left(\mathbf{b}^k\right)^T \mathbf{F}^{k-1}\mathbf{b}^k}. \tag{A.108}$$

This is called the Davidon–Fletcher–Powell (DFP) method based on the authors who first put forth this solution [14, 72].

While the update derived here is for the inverse matrix $\mathbf{F}^k = \left(\mathbf{S}^k\right)^{-1}$, an entirely similar recursive expression can be formulated for $\mathbf{S}^k$ itself, leading to the Broyden–Fletcher–Goldfarb–Shanno (BFGS) update, named after its original authors [14, 72, 73] (see Exercise 4.10).

A.8.4 Low-memory quasi-Newton methods

In the previous examples we saw how to construct a sequence of (inverse) secant matrices via the recursion

$$\left(\mathbf{S}^k\right)^{-1} = \left(\mathbf{S}^{k-1}\right)^{-1} + \mathbf{D}^{k-1} \tag{A.109}$$

to replace the true Hessian sequence. The *descent direction* for the kth step of a quasi-Newton method then takes the form

$$\mathbf{d}^k = -\left(\mathbf{S}^{k-1}\right)^{-1} \nabla g(\mathbf{w}^{k-1}) \tag{A.110}$$

where $\mathbf{S}^{k-1}$ is our approximation to $\nabla^2 g\left(\mathbf{w}^{k-1}\right)$. Notice, however, that as written in Equation (A.110), computing the descent direction still involves the explicit form of an $N \times N$ matrix: the inverse of $\mathbf{S}^{k-1}$. But recall, it was precisely the presence of the $N \times N$ Hessian matrix (with its N^2 values) that drove us to examine quasi-Newton methods to begin with. So, at first glance, it appears that we have not avoided the serious scaling issues associated with employing an $N \times N$ matrix: originally a Hessian matrix, now a secant matrix.

Luckily we can avoid explicit construction of the (inverse) secant matrix by focusing on how it acts on the gradient vector $\nabla g(\mathbf{w}^{k-1})$ in Equation (A.110). For example, denoting for notational convenience $\mathbf{z}^{k-1} = \nabla g(\mathbf{w}^{k-1})$ and $\mathbf{F}^k = \left(\mathbf{S}^k\right)^{-1}$, the descent direction $\mathbf{d}^k$ using the recursive update derived in Example A.11 can be written as

$$\begin{aligned}\mathbf{d}^k = -\mathbf{F}^k\mathbf{z}^{k-1} &= -\left(\mathbf{F}^{k-1} + \frac{\left(\mathbf{a}^k - \mathbf{F}^{k-1}\mathbf{b}^k\right)\left(\mathbf{a}^k - \mathbf{F}^{k-1}\mathbf{b}^k\right)^T}{\left(\mathbf{b}^k\right)^T\mathbf{a}^k - \left(\mathbf{b}^k\right)^T\mathbf{F}^{k-1}\mathbf{b}^k}\right)\mathbf{z}^{k-1} \\ &= -\mathbf{F}^{k-1}\mathbf{z}^{k-1} - \frac{\left(\mathbf{a}^k - \mathbf{F}^{k-1}\mathbf{b}^k\right)^T\mathbf{z}^{k-1}}{\left(\mathbf{b}^k\right)^T\mathbf{a}^k - \left(\mathbf{b}^k\right)^T\mathbf{F}^{k-1}\mathbf{b}^k}\left(\mathbf{a}^k - \mathbf{F}^{k-1}\mathbf{b}^k\right).\end{aligned} \tag{A.111}$$

Notice wherever the matrix $\mathbf{F}^{k-1}$ appears in the final line of Equation (A.111), it is always attached to some other vector, in this case $\mathbf{b}^k$ and $\mathbf{z}^{k-1}$. Thus if we somehow manage to compute $\mathbf{F}^{k-1}\mathbf{b}^k$ and $\mathbf{F}^{k-1}\mathbf{z}^{k-1}$ directly as vectors, we can avoid explicitly forming $\mathbf{F}^{k-1}$.
This is where our particular recursive construction of $\mathbf{F}^k$ as

$$\mathbf{F}^k = \mathbf{F}^{k-1} + \mathbf{u}^{k-1}\left(\mathbf{u}^{k-1}\right)^T \tag{A.112}$$

comes to rescue. Rolling this recursion all the way back to $\mathbf{F}^0$ and initializing $\mathbf{F}^0 = \mathbf{I}_{N\times N}$, we can write it equivalently as

$$\mathbf{F}^k = \mathbf{I}_{N\times N} + \left(\mathbf{u}^0\left(\mathbf{u}^0\right)^T + \cdots + \mathbf{u}^{k-1}\left(\mathbf{u}^{k-1}\right)^T\right). \tag{A.113}$$

Multiplication of any vector $\mathbf{t}$ with $\mathbf{F}^k$ can then be written as

$$\mathbf{F}^k\,\mathbf{t} = \mathbf{I}_{N\times N}\,\mathbf{t} + \left(\mathbf{u}^0\left(\mathbf{u}^0\right)^T + \cdots + \mathbf{u}^{k-1}\left(\mathbf{u}^{k-1}\right)^T\right)\mathbf{t} \tag{A.114}$$

and simplified as

$$\mathbf{F}^k\,\mathbf{t} = \mathbf{t} + \mathbf{u}^0\left(\left(\mathbf{u}^0\right)^T\mathbf{t}\right) + \cdots + \mathbf{u}^{k-1}\left(\left(\mathbf{u}^{k-1}\right)^T\mathbf{t}\right). \tag{A.115}$$

To compute $\mathbf{F}^k\,\mathbf{t}$, as written in Equation (A.115), we only need access to vectors $\mathbf{t}$ as well as $\mathbf{u}^0$ through $\mathbf{u}^{k-1}$, and perform simple vector inner-product.

We can make very similar kinds of arguments with any rank-1 or rank-2 quasi-Newton update formula, meaning that any of the update formulae given in the preceding examples can be implemented in a way that we never need explicitly construct a secant or inverse secant matrix (see, e.g., [14, 72]).