Chapter 4

Numerical Computation

Machine learning algorithms usually require a high amount of numerical computation. This typically refers to algorithms that solve mathematical problems by methods that update estimates of the solution via an iterative process, rather than analytically deriving a formula providing a symbolic expression for the correct solution. Common operations include optimization (finding the value of an argument that minimizes or maximizes a function) and solving systems of linear equations. Even just evaluating a mathematical function on a digital computer can be difficult when the function involves real numbers, which cannot be represented precisely using a finite amount of memory.

4.1 Overflow and Underflow

The fundamental difficulty in performing continuous math on a digital computer is that we need to represent infinitely many real numbers with a finite number of bit patterns. This means that for almost all real numbers, we incur some approximation error when we represent the number in the computer. In many cases, this is just rounding error. Rounding error is problematic, especially when it compounds across many operations, and can cause algorithms that work in theory to fail in practice if they are not designed to minimize the accumulation of rounding error.

One form of rounding error that is particularly devastating is *underflow*. Underflow occurs when numbers near zero are rounded to zero. Many functions behave qualitatively differently when their argument is zero rather than a small positive number. For example, we usually want to avoid division by zero (some software

environments will raise exceptions when this occurs, others will return a result with a placeholder not-a-number value) or taking the logarithm of zero (this is usually treated as $-\infty$, which then becomes not-a-number if it is used for many further arithmetic operations).

Another highly damaging form of numerical error is *overflow*. Overflow occurs when numbers with large magnitude are approximated as ∞ or $-\infty$. Further arithmetic will usually change these infinite values into not-a-number values.

One example of a function that must be stabilized against underflow and overflow is the softmax function. The softmax function is often used to predict the probabilities associated with a multinoulli distribution. The softmax function is defined to be

$$\text{softmax}(\boldsymbol{x})_i = \frac{\exp(x_i)}{\sum_{j=1}^{n} \exp(x_j)}. \tag{4.1}$$

Consider what happens when all of the x_i are equal to some constant c. Analytically, we can see that all of the outputs should be equal to $\frac{1}{n}$. Numerically, this may not occur when c has large magnitude. If c is very negative, then $\exp(c)$ will underflow. This means the denominator of the softmax will become 0, so the final result is undefined. When c is very large and positive, $\exp(c)$ will overflow, again resulting in the expression as a whole being undefined. Both of these difficulties can be resolved by instead evaluating $\text{softmax}(\boldsymbol{z})$ where $\boldsymbol{z} = \boldsymbol{x} - \max_i x_i$. Simple algebra shows that the value of the softmax function is not changed analytically by adding or subtracting a scalar from the input vector. Subtracting $\max_i x_i$ results in the largest argument to exp being 0, which rules out the possibility of overflow. Likewise, at least one term in the denominator has a value of 1, which rules out the possibility of underflow in the denominator leading to a division by zero.

There is still one small problem. Underflow in the numerator can still cause the expression as a whole to evaluate to zero. This means that if we implement $\log \text{softmax}(\boldsymbol{x})$ by first running the softmax subroutine then passing the result to the log function, we could erroneously obtain $-\infty$. Instead, we must implement a separate function that calculates $\log \text{softmax}$ in a numerically stable way. The $\log \text{softmax}$ function can be stabilized using the same trick as we used to stabilize the softmax function.

For the most part, we do not explicitly detail all of the numerical considerations involved in implementing the various algorithms described in this book. Developers of low-level libraries should keep numerical issues in mind when implementing deep learning algorithms. Most readers of this book can simply rely on low-level libraries that provide stable implementations. In some cases, it is possible to implement a new algorithm and have the new implementation automatically

stabilized. Theano (Bergstra *et al.*, 2010; Bastien *et al.*, 2012) is an example of a software package that automatically detects and stabilizes many common numerically unstable expressions that arise in the context of deep learning.

4.2 Poor Conditioning

Conditioning refers to how rapidly a function changes with respect to small changes in its inputs. Functions that change rapidly when their inputs are perturbed slightly can be problematic for scientific computation because rounding errors in the inputs can result in large changes in the output.

Consider the function $f(\boldsymbol{x}) = \boldsymbol{A}^{-1}\boldsymbol{x}$. When $\boldsymbol{A} \in \mathbb{R}^{n \times n}$ has an eigenvalue decomposition, its *condition number* is

$$\max_{i,j} \left| \frac{\lambda_i}{\lambda_j} \right|. \tag{4.2}$$

This is the ratio of the magnitude of the largest and smallest eigenvalue. When this number is large, matrix inversion is particularly sensitive to error in the input.

This sensitivity is an intrinsic property of the matrix itself, not the result of rounding error during matrix inversion. Poorly conditioned matrices amplify pre-existing errors when we multiply by the true matrix inverse. In practice, the error will be compounded further by numerical errors in the inversion process itself.

4.3 Gradient-Based Optimization

Most deep learning algorithms involve optimization of some sort. Optimization refers to the task of either minimizing or maximizing some function $f(\boldsymbol{x})$ by altering $\boldsymbol{x}$. We usually phrase most optimization problems in terms of minimizing $f(\boldsymbol{x})$. Maximization may be accomplished via a minimization algorithm by minimizing $-f(\boldsymbol{x})$.

The function we want to minimize or maximize is called the *objective function* or *criterion*. When we are minimizing it, we may also call it the *cost function*, *loss function*, or *error function*. In this book, we use these terms interchangeably, though some machine learning publications assign special meaning to some of these terms.

We often denote the value that minimizes or maximizes a function with a superscript $*$. For example, we might say $\boldsymbol{x}^* = \arg\min f(\boldsymbol{x})$.

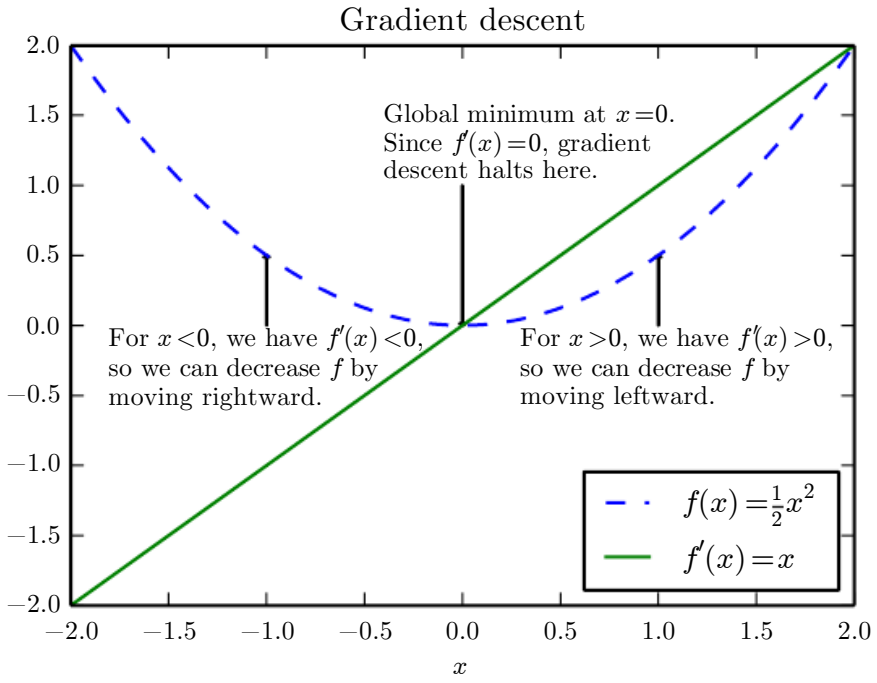

Figure 4.1: An illustration of how the derivatives of a function can be used to follow the function downhill to a minimum. This technique is called *gradient descent*.

We assume the reader is already familiar with calculus, but provide a brief review of how calculus concepts relate to optimization here.

Suppose we have a function $y = f(x)$, where both x and y are real numbers. The *derivative* of this function is denoted as $f'(x)$ or as $\frac{dy}{dx}$. The derivative $f'(x)$ gives the slope of $f(x)$ at the point x. In other words, it specifies how to scale a small change in the input in order to obtain the corresponding change in the output: $f(x + \epsilon) \approx f(x) + \epsilon f'(x)$.

The derivative is therefore useful for minimizing a function because it tells us how to change x in order to make a small improvement in y. For example, we know that $f(x - \epsilon \operatorname{sign}(f'(x)))$ is less than $f(x)$ for small enough ϵ. We can thus reduce $f(x)$ by moving x in small steps with opposite sign of the derivative. This technique is called *gradient descent* (Cauchy, 1847). See Fig. 4.1 for an example of this technique.

When $f'(x) = 0$, the derivative provides no information about which direction to move. Points where $f'(x) = 0$ are known as *critical points* or *stationary points*. A *local minimum* is a point where $f(x)$ is lower than at all neighboring points, so it is no longer possible to decrease $f(x)$ by making infinitesimal steps. A *local maximum* is a point where $f(x)$ is higher than at all neighboring points, so it is

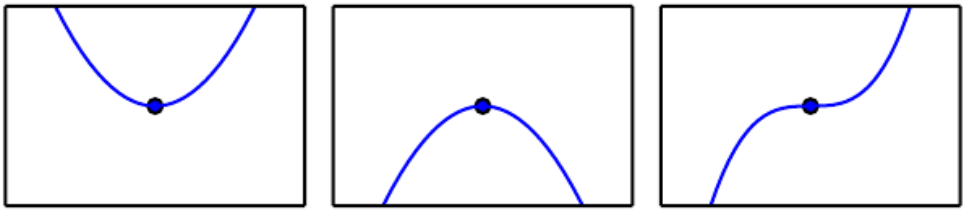

Figure 4.2: Examples of each of the three types of critical points in 1-D. A critical point is a point with zero slope. Such a point can either be a local minimum, which is lower than the neighboring points, a local maximum, which is higher than the neighboring points, or a saddle point, which has neighbors that are both higher and lower than the point itself.

not possible to increase $f(x)$ by making infinitesimal steps. Some critical points are neither maxima nor minima. These are known as *saddle points*. See Fig. 4.2 for examples of each type of critical point.

A point that obtains the absolute lowest value of $f(x)$ is a *global minimum*. It is possible for there to be only one global minimum or multiple global minima of the function. It is also possible for there to be local minima that are not globally optimal. In the context of deep learning, we optimize functions that may have many local minima that are not optimal, and many saddle points surrounded by very flat regions. All of this makes optimization very difficult, especially when the input to the function is multidimensional. We therefore usually settle for finding a value of f that is very low, but not necessarily minimal in any formal sense. See Fig. 4.3 for an example.

We often minimize functions that have multiple inputs: $f : \mathbb{R}^n \to \mathbb{R}$. For the concept of "minimization" to make sense, there must still be only one (scalar) output.

For functions with multiple inputs, we must make use of the concept of *partial derivatives*. The partial derivative $\frac{\partial}{\partial x_i} f(\boldsymbol{x})$ measures how f changes as only the variable x_i increases at point $\boldsymbol{x}$. The *gradient* generalizes the notion of derivative to the case where the derivative is with respect to a vector: the gradient of f is the vector containing all of the partial derivatives, denoted $\nabla f(\boldsymbol{x})$. Element i of the gradient is the partial derivative of f with respect to x_i. In multiple dimensions,

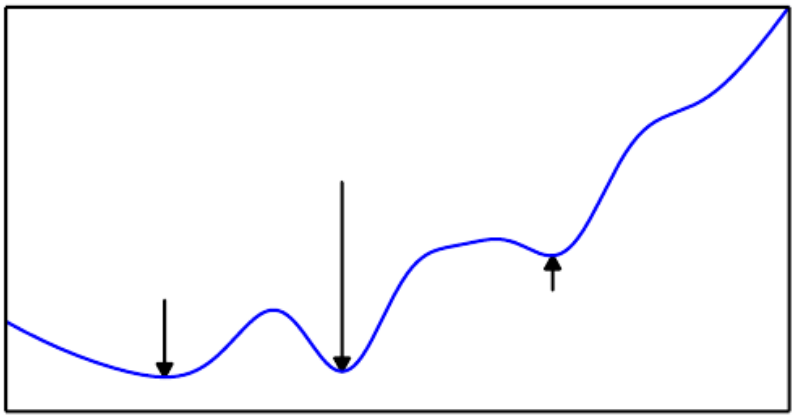

Figure 4.3: Optimization algorithms may fail to find a global minimum when there are multiple local minima or plateaus present. In the context of deep learning, we generally accept such solutions even though they are not truly minimal, so long as they correspond to significantly low values of the cost function.

critical points are points where every element of the gradient is equal to zero.

The *directional derivative* in direction $\boldsymbol{u}$ (a unit vector) is the slope of the function f in direction u. In other words, the directional derivative is the derivative of the function $f(\boldsymbol{x}+\alpha\boldsymbol{u})$ with respect to α, evaluated at $\alpha=0$. Using the chain rule, we can see that $\frac{\partial}{\partial\alpha}f(\boldsymbol{x}+\alpha\boldsymbol{u})=\boldsymbol{u}^\top\nabla\ f(\boldsymbol{x})$.

To minimize f, we would like to find the direction in which f decreases the fastest. We can do this using the directional derivative:

$$\min_{,\ =1}\ \boldsymbol{u}^\top\nabla\ f(\boldsymbol{x}) \tag{4.3}$$

$$=\ \min_{,\ =1}\ ||\boldsymbol{u}|_2||\nabla\ f(\boldsymbol{x})||_2\cos\theta \tag{4.4}$$

where θ is the angle between $\boldsymbol{u}$ and the gradient. Substituting in $||\boldsymbol{u}||_2=1$ and ignoring factors that do not depend on $\boldsymbol{u}$, this simplifies to min $\cos\theta$. This is minimized when $\boldsymbol{u}$ points in the opposite direction as the gradient. In other words, the gradient points directly uphill, and the negative gradient points directly downhill. We can decrease f by moving in the direction of the negative gradient. This is known as the *method of steepest descent* or *gradient descent*.

Steepest descent proposes a new point

$$\boldsymbol{x}'=\boldsymbol{x}-\epsilon\nabla\ f(\boldsymbol{x}) \tag{4.5}$$

where ϵ is the *learning rate*, a positive scalar determining the size of the step. We can choose ϵ in several different ways. A popular approach is to set ϵ to a small constant. Sometimes, we can solve for the step size that makes the directional derivative vanish. Another approach is to evaluate $f\left(\boldsymbol{x} - \epsilon \nabla \ f(\boldsymbol{x})\right)$ for several values of ϵ and choose the one that results in the smallest objective function value. This last strategy is called a *line search*.

Steepest descent converges when every element of the gradient is zero (or, in practice, very close to zero). In some cases, we may be able to avoid running this iterative algorithm, and just jump directly to the critical point by solving the equation $\nabla \ f(\boldsymbol{x}) = 0$ for $\boldsymbol{x}$.

Although gradient descent is limited to optimization in continuous spaces, the general concept of making small moves (that are approximately the best small move) towards better configurations can be generalized to discrete spaces. Ascending an objective function of discrete parameters is called *hill climbing* (Russel and Norvig, 2003).

Sometimes we need to find all of the partial derivatives of a function whose input and output are both vectors. The matrix containing all such partial derivatives is known as a *Jacobian matrix*. Specifically, if we have a function $\boldsymbol{f} : \mathbb{R}^m \rightarrow \mathbb{R}^n$, then the Jacobian matrix $\boldsymbol{J} \in \mathbb{R}^{n \times m}$ of $\boldsymbol{f}$ is defined such that $J_{i,j} = \frac{\partial}{\partial x} f(\boldsymbol{x})_i$.

We are also sometimes interested in a derivative of a derivative. This is known as a *second derivative*. For example, for a function $f : \mathbb{R}^n \rightarrow \mathbb{R}$, the derivative with respect to x_i of the derivative of f with respect to x_j is denoted as $\frac{\partial}{\partial x \ \partial x} f$. In a single dimension, we can denote $\frac{d}{dx} f$ by $f''(x)$. The second derivative tells us how the first derivative will change as we vary the input. This is important because it tells us whether a gradient step will cause as much of an improvement as we would expect based on the gradient alone. We can think of the second derivative as measuring *curvature*. Suppose we have a quadratic function (many functions that arise in practice are not quadratic but can be approximated well as quadratic, at least locally). If such a function has a second derivative of zero, then there is no curvature. It is a perfectly flat line, and its value can be predicted using only the gradient. If the gradient is 1, then we can make a step of size ϵ along the negative gradient, and the cost function will decrease by ϵ. If the second derivative is negative, the function curves downward, so the cost function will actually decrease by more than ϵ. Finally, if the second derivative is positive, the function curves upward, so the cost function can decrease by less than ϵ. See Fig.

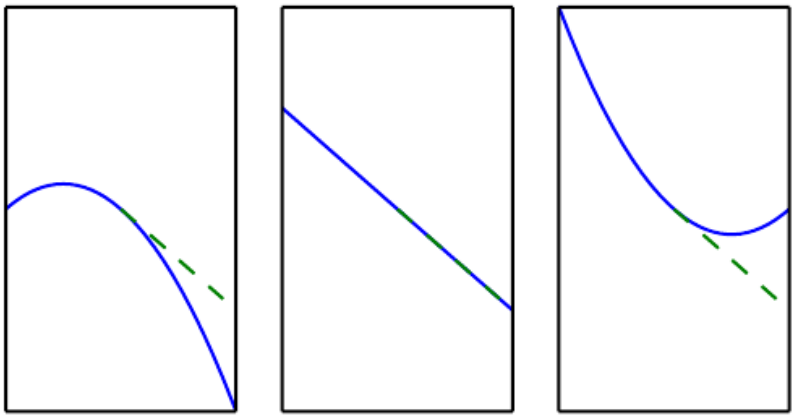

Figure 4.4: The second derivative determines the curvature of a function. Here we show quadratic functions with various curvature. The dashed line indicates the value of the cost function we would expect based on the gradient information alone as we make a gradient step downhill. In the case of negative curvature, the cost function actually decreases faster than the gradient predicts. In the case of no curvature, the gradient predicts the decrease correctly. In the case of positive curvature, the function decreases slower than expected and eventually begins to increase, so too large of step sizes can actually increase the function inadvertently.

4.4 to see how different forms of curvature affect the relationship between the value of the cost function predicted by the gradient and the true value.

When our function has multiple input dimensions, there are many second derivatives. These derivatives can be collected together into a matrix called the *Hessian matrix*. The Hessian matrix $\boldsymbol{H}(f)(\boldsymbol{x})$ is defined such that

$$\boldsymbol{H}(f)(\boldsymbol{x})_{i,j} = \frac{\partial^2}{\partial x_i \partial x_j} f(\boldsymbol{x}). \tag{4.6}$$

Equivalently, the Hessian is the Jacobian of the gradient.

Anywhere that the second partial derivatives are continuous, the differential operators are commutative, i.e. their order can be swapped:

$$\frac{\partial^2}{\partial x_i \partial x_j} f(\boldsymbol{x}) = \frac{\partial^2}{\partial x_j \partial x_i} f(\boldsymbol{x}). \tag{4.7}$$

This implies that $H_{i,j} = H_{j,i}$, so the Hessian matrix is symmetric at such points. Most of the functions we encounter in the context of deep learning have a symmetric Hessian almost everywhere. Because the Hessian matrix is real and symmetric, we can decompose it into a set of real eigenvalues and an orthogonal basis of

eigenvectors. The second derivative in a specific direction represented by a unit vector $\boldsymbol{d}$ is given by $\boldsymbol{d}^\top \boldsymbol{H}\boldsymbol{d}$. When $\boldsymbol{d}$ is an eigenvector of $\boldsymbol{H}$, the second derivative in that direction is given by the corresponding eigenvalue. For other directions of $\boldsymbol{d}$, the directional second derivative is a weighted average of all of the eigenvalues, with weights between 0 and 1, and eigenvectors that have smaller angle with $\boldsymbol{d}$ receiving more weight. The maximum eigenvalue determines the maximum second derivative and the minimum eigenvalue determines the minimum second derivative.

The (directional) second derivative tells us how well we can expect a gradient descent step to perform. We can make a second-order Taylor series approximation to the function $f(\boldsymbol{x})$ around the current point $\boldsymbol{x}^{(0)}$:

$$f(\boldsymbol{x}) \approx f(\boldsymbol{x}^{(0)}) + (\boldsymbol{x} - \boldsymbol{x}^{(0)})^\top \boldsymbol{g} + \frac{1}{2}(\boldsymbol{x} - \boldsymbol{x}^{(0)})^\top \boldsymbol{H}(\boldsymbol{x} - \boldsymbol{x}^{(0)}). \tag{4.8}$$

where $\boldsymbol{g}$ is the gradient and $\boldsymbol{H}$ is the Hessian at $\boldsymbol{x}^{(0)}$. If we use a learning rate of ϵ, then the new point $\boldsymbol{x}$ will be given by $\boldsymbol{x}^{(0)} - \epsilon \boldsymbol{g}$. Substituting this into our approximation, we obtain

$$f(\boldsymbol{x}^{(0)} - \epsilon \boldsymbol{g}) \approx f(\boldsymbol{x}^{(0)}) - \epsilon \boldsymbol{g}^\top \boldsymbol{g} + \frac{1}{2}\epsilon^2 \boldsymbol{g}^\top \boldsymbol{H}\boldsymbol{g}. \tag{4.9}$$

There are three terms here: the original value of the function, the expected improvement due to the slope of the function, and the correction we must apply to account for the curvature of the function. When this last term is too large, the gradient descent step can actually move uphill. When $\boldsymbol{g}^\top \boldsymbol{H}\boldsymbol{g}$ is zero or negative, the Taylor series approximation predicts that increasing ϵ forever will decrease f forever. In practice, the Taylor series is unlikely to remain accurate for large ϵ, so one must resort to more heuristic choices of ϵ in this case. When $\boldsymbol{g}^\top \boldsymbol{H}\boldsymbol{g}$ is positive, solving for the optimal step size that decreases the Taylor series approximation of the function the most yields

$$\epsilon^* = \frac{\boldsymbol{g}^\top \boldsymbol{g}}{\boldsymbol{g}^\top \boldsymbol{H}\boldsymbol{g}}. \tag{4.10}$$

In the worst case, when $\boldsymbol{g}$ aligns with the eigenvector of $\boldsymbol{H}$ corresponding to the maximal eigenvalue $\lambda_{\max}$, then this optimal step size is given by $\frac{1}{\lambda}$. To the extent that the function we minimize can be approximated well by a quadratic function, the eigenvalues of the Hessian thus determine the scale of the learning rate.

The second derivative can be used to determine whether a critical point is a local maximum, a local minimum, or saddle point. Recall that on a critical point, $f'(x) = 0$. When $f''(x) > 0$, this means that $f'(x)$ increases as we move to the right, and $f'(x)$ decreases as we move to the left. This means $f'(x - \epsilon) < 0$ and

$f'(x+\epsilon) > 0$ for small enough ϵ. In other words, as we move right, the slope begins to point uphill to the right, and as we move left, the slope begins to point uphill to the left. Thus, when $f'(x) = 0$ and $f''(x) > 0$, we can conclude that x is a local minimum. Similarly, when $f'(x) = 0$ and $f''(x) < 0$, we can conclude that x is a local maximum. This is known as the *second derivative test*. Unfortunately, when $f''(x) = 0$, the test is inconclusive. In this case x may be a saddle point, or a part of a flat region.

In multiple dimensions, we need to examine all of the second derivatives of the function. Using the eigendecomposition of the Hessian matrix, we can generalize the second derivative test to multiple dimensions. At a critical point, where $\nabla f(\boldsymbol{x}) = 0$, we can examine the eigenvalues of the Hessian to determine whether the critical point is a local maximum, local minimum, or saddle point. When the Hessian is positive definite (all its eigenvalues are positive), the point is a local minimum. This can be seen by observing that the directional second derivative in any direction must be positive, and making reference to the univariate second derivative test. Likewise, when the Hessian is negative definite (all its eigenvalues are negative), the point is a local maximum. In multiple dimensions, it is actually possible to find positive evidence of saddle points in some cases. When at least one eigenvalue is positive and at least one eigenvalue is negative, we know that $\boldsymbol{x}$ is a local maximum on one cross section of f but a local minimum on another cross section. See Fig. 4.5 for an example. Finally, the multidimensional second derivative test can be inconclusive, just like the univariate version. The test is inconclusive whenever all of the non-zero eigenvalues have the same sign, but at least one eigenvalue is zero. This is because the univariate second derivative test is inconclusive in the cross section corresponding to the zero eigenvalue.

In multiple dimensions, there can be a wide variety of different second derivatives at a single point, because there is a different second derivative for each direction. The condition number of the Hessian measures how much the second derivatives vary. When the Hessian has a poor condition number, gradient descent performs poorly. This is because in one direction, the derivative increases rapidly, while in another direction, it increases slowly. Gradient descent is unaware of this change in the derivative so it does not know that it needs to explore preferentially in the direction where the derivative remains negative for longer. It also makes it difficult to choose a good step size. The step size must be small enough to avoid overshooting the minimum and going uphill in directions with strong positive curvature. This usually means that the step size is too small to make significant progress in other directions with less curvature. See Fig. 4.6 for an example.

This issue can be resolved by using information from the Hessian matrix to

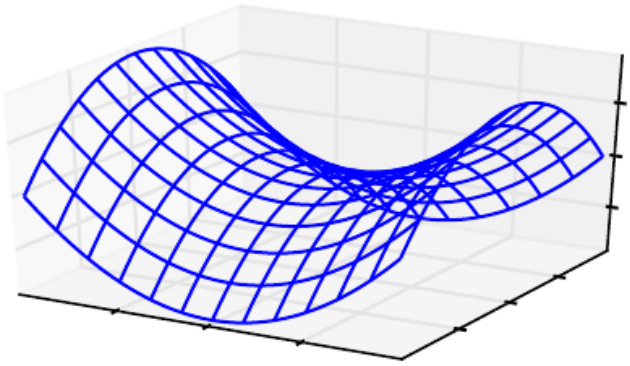

Figure 4.5: A saddle point containing both positive and negative curvature. The function in this example is $f(\boldsymbol{x}) = x \ - x$. Along the axis corresponding to x , the function curves upward. This axis is an eigenvector of the Hessian and has a positive eigenvalue. Along the axis corresponding to x , the function curves downward. This direction is an eigenvector of the Hessian with negative eigenvalue. The name "saddle point" derives from the saddle-like shape of this function. This is the quintessential example of a function with a saddle point. In more than one dimension, it is not necessary to have an eigenvalue of 0 in order to get a saddle point: it is only necessary to have both positive and negative eigenvalues. We can think of a saddle point with both signs of eigenvalues as being a local maximum within one cross section and a local minimum within another cross section.

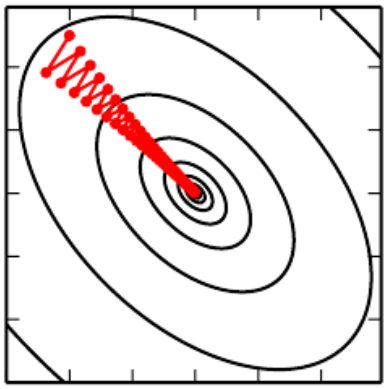

Figure 4.6: Gradient descent fails to exploit the curvature information contained in the Hessian matrix. Here we use gradient descent to minimize a quadratic function $f(\boldsymbol{x})$ whose Hessian matrix has condition number 5. This means that the direction of most curvature has five times more curvature than the direction of least curvature. In this case, the most curvature is in the direction $[1, 1]$ and the least curvature is in the direction $[1, -1]$. The red lines indicate the path followed by gradient descent. This very elongated quadratic function resembles a long canyon. Gradient descent wastes time repeatedly descending canyon walls, because they are the steepest feature. Because the step size is somewhat too large, it has a tendency to overshoot the bottom of the function and thus needs to descend the opposite canyon wall on the next iteration. The large positive eigenvalue of the Hessian corresponding to the eigenvector pointed in this direction indicates that this directional derivative is rapidly increasing, so an optimization algorithm based on the Hessian could predict that the steepest direction is not actually a promising search direction in this context.

guide the search. The simplest method for doing so is known as *Newton's method*. Newton's method is based on using a second-order Taylor series expansion to approximate $f(\boldsymbol{x})$ near some point $\boldsymbol{x}^{(0)}$:

$$f(\boldsymbol{x}) \approx f(\boldsymbol{x}^{(0)}) + (\boldsymbol{x} - \boldsymbol{x}^{(0)})^\top \nabla_{\boldsymbol{x}} f(\boldsymbol{x}^{(0)}) + \frac{1}{2}(\boldsymbol{x} - \boldsymbol{x}^{(0)})^\top \boldsymbol{H}(f)(\boldsymbol{x}^{(0)})(\boldsymbol{x} - \boldsymbol{x}^{(0)}). \quad (4.11)$$

If we then solve for the critical point of this function, we obtain:

$$\boldsymbol{x}^* = \boldsymbol{x}^{(0)} - \boldsymbol{H}(f)(\boldsymbol{x}^{(0)})^{-1} \nabla_{\boldsymbol{x}} f(\boldsymbol{x}^{(0)}). \quad (4.12)$$

When f is a positive definite quadratic function, Newton's method consists of applying Eq. 4.12 once to jump to the minimum of the function directly. When f is not truly quadratic but can be locally approximated as a positive definite quadratic, Newton's method consists of applying Eq. 4.12 multiple times. Iteratively updating the approximation and jumping to the minimum of the approximation can reach the critical point much faster than gradient descent would. This is a useful property near a local minimum, but it can be a harmful property near a saddle point. As discussed in Sec. 8.2.3, Newton's method is only appropriate when the nearby critical point is a minimum (all the eigenvalues of the Hessian are positive), whereas gradient descent is not attracted to saddle points unless the gradient points toward them.

Optimization algorithms such as gradient descent that use only the gradient are called *first-order optimization algorithms*. Optimization algorithms such as Newton's method that also use the Hessian matrix are called *second-order optimization algorithms* (Nocedal and Wright, 2006).

The optimization algorithms employed in most contexts in this book are applicable to a wide variety of functions, but come with almost no guarantees. This is because the family of functions used in deep learning is quite complicated. In many other fields, the dominant approach to optimization is to design optimization algorithms for a limited family of functions.

In the context of deep learning, we sometimes gain some guarantees by restricting ourselves to functions that are either *Lipschitz continuous* or have Lipschitz continuous derivatives. A Lipschitz continuous function is a function f whose rate of change is bounded by a *Lipschitz constant* $\mathcal{L}$:

$$\forall \boldsymbol{x}, \forall \boldsymbol{y}, |f(\boldsymbol{x}) - f(\boldsymbol{y})| \leq \mathcal{L} ||\boldsymbol{x} - \boldsymbol{y}||_2 . \quad (4.13)$$

This property is useful because it allows us to quantify our assumption that a small change in the input made by an algorithm such as gradient descent will have a small change in the output. Lipschitz continuity is also a fairly weak constraint,