

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 iteratively update estimates of the solution, rather than analytically deriving a formula providing a symbolic expression for the correct solution. Common operations include solving systems of linear equations and finding the value of an argument that minimizes a function. 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, otherwise 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 further arithmetic).

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.

For an example of the need to design software implementations to deal with overflow and underflow, consider the softmax function, typically used to predict the probabilities associated with a multinoulli distribution:

$$\text{softmax}(\mathbf{x})_i = \frac{\exp(x_i)}{\sum_j^n \exp(x_j)}.$$

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 $\text{softmax}(\mathbf{z})$ where $\mathbf{z} = \mathbf{x} - \max_i x_i$. 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 $\log \text{softmax}(\mathbf{x})$ 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. Implementors should keep numerical issues in mind when developing implementations. Many numerical issues can be avoided by using Theano (Bergstra *et al.*, 2010a; Bastien *et al.*, 2012), 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 per-

turbed slightly can be problematic for scientific computation because rounding errors in the inputs can result in large changes in the output.

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

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

i.e. 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.

Note that this 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. With iterative algorithms such as solving a linear system (or the worked-out example of linear least square by gradient descent, Section 4.5) ill-conditioning (in that case of the linear system matrix) yields very slow convergence of the iterative algorithm, i.e., more iterations are needed to achieve some given degree of approximation to the final solution.

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 $f(\mathbf{x})$ by altering \mathbf{x} . We usually phrase most optimization problems in terms of minimizing $f(\mathbf{x})$. Maximization may be accomplished via a minimization algorithm by minimizing $-f(\mathbf{x})$.

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 $x^* = \arg \min f(\mathbf{x})$.

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: $f(x + \epsilon) \approx f(x) + \epsilon f'(x)$.

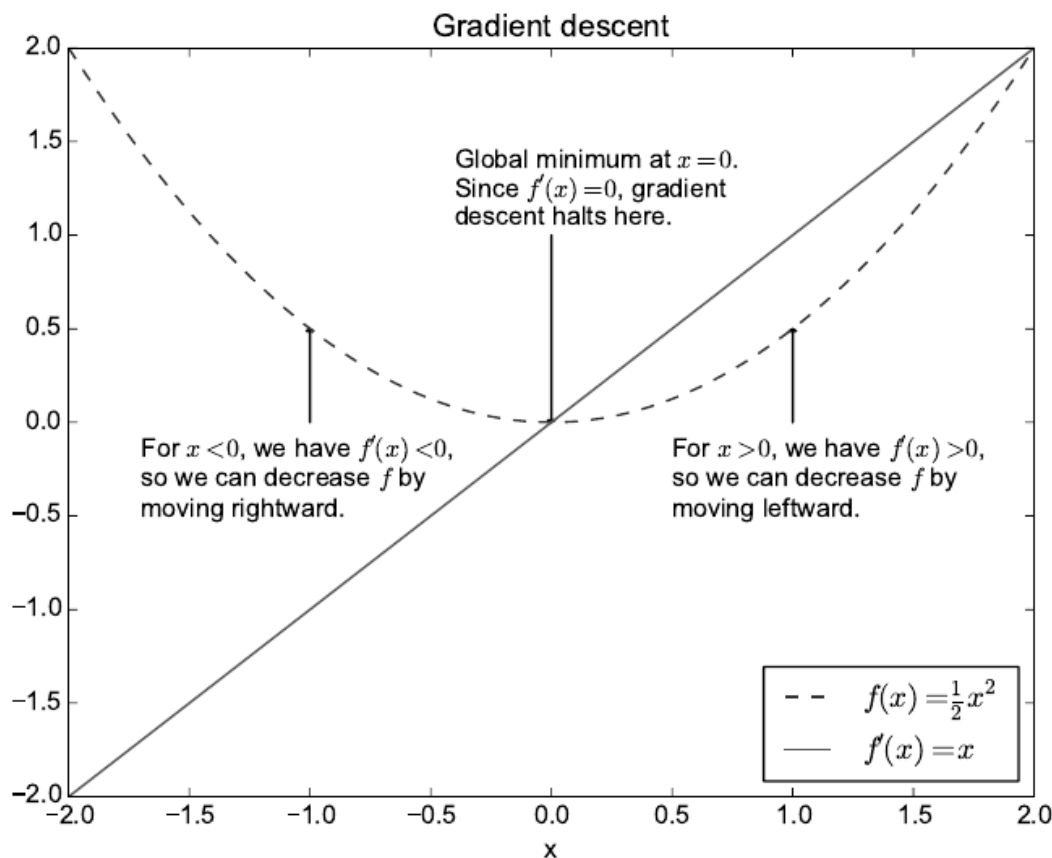


Figure 4.1: An illustration of how the derivatives of a function can be used to follow the function downhill to a minimum. This technique is called *gradient descent*.

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 $f(x - \epsilon \text{sign}(f'(x)))$ 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 the derivative. This technique is called gradient descent (Cauchy, 1847a). 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 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

Types of critical points

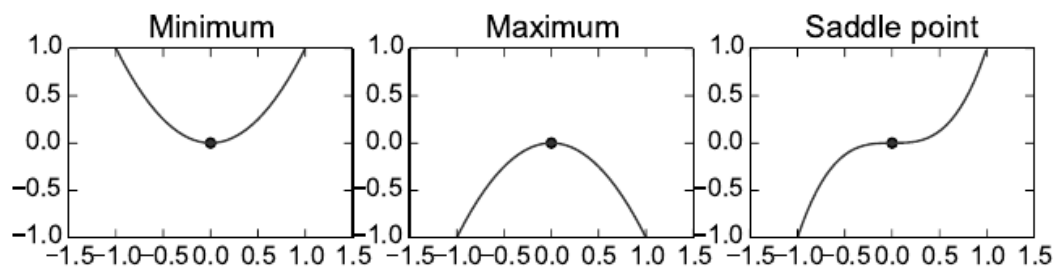


Figure 4.2: Examples of each of the three types of critical points in 1-D. A critical point is a point with zero slope. Such a point can either be a local minimum, which is lower than the neighboring points, a local maximum, which is higher than the neighboring points, or a saddle point, which has neighbors that are both higher and lower than the point itself. The situation in higher dimension is qualitatively different, especially for saddle points: see Figures 4.4 and 4.5.

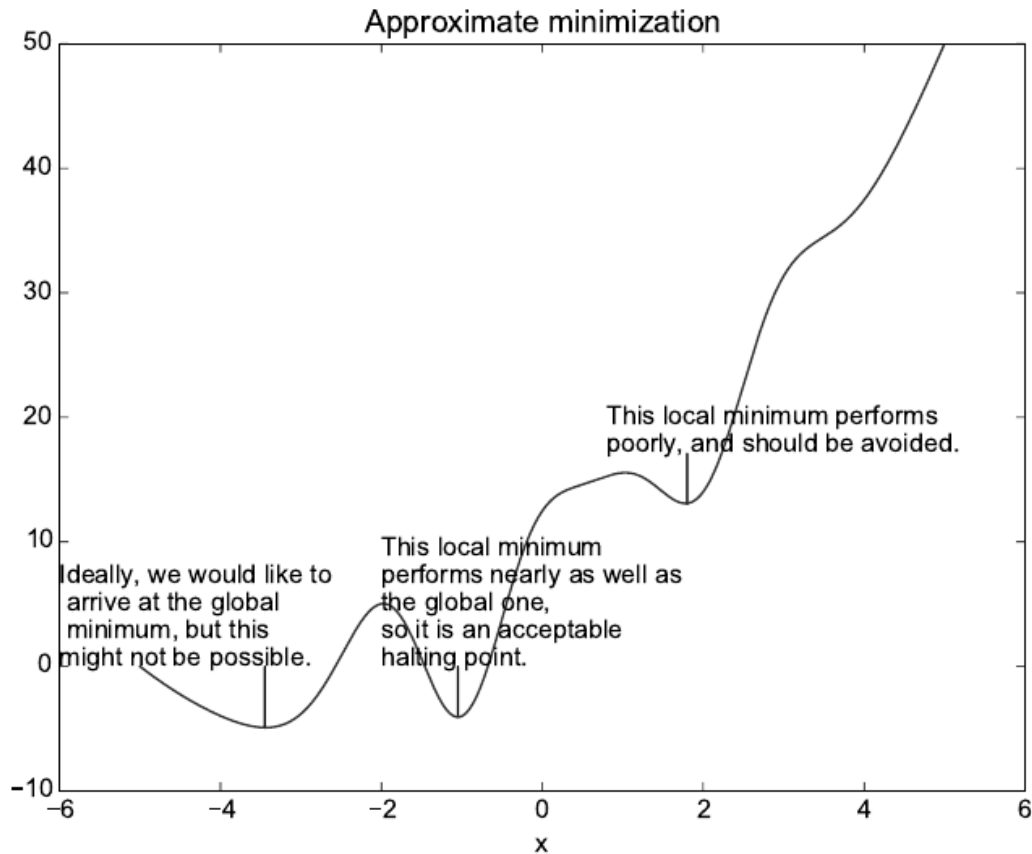


Figure 4.3: Optimization algorithms may fail to find a global minimum when there are multiple local minima or plateaus present. In the context of deep learning, we generally accept such solutions even though they are not truly minimal, so long as they correspond to significantly low values of the cost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

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: $f : \mathbb{R}^n \rightarrow \mathbb{R}$. Note that for the concept of “minimization” to make sense, there must still be only one output.

For these functions, we must make use of the concept of *partial derivatives*. The partial derivative $\frac{\partial}{\partial x_i} f(\mathbf{x})$ measures how f changes as only the variable x_i

increases at point \mathbf{x} . The *gradient* generalizes the notion of derivative to the case where the derivative is with respect to a vector: \mathbf{f} is the vector containing all of the partial derivatives, denoted $\nabla_{\mathbf{x}} f(\mathbf{x})$. Element i of the gradient is the partial derivative of f with respect to x_i . In multiple dimensions, critical points are points where every element of the gradient is equal to zero.

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

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

$$\begin{aligned} & \min_{\mathbf{u}, \mathbf{u}^\top \mathbf{u} = 1} \mathbf{u}^\top \nabla_{\mathbf{x}} f(\mathbf{x}) \\ &= \min_{\mathbf{u}, \mathbf{u}^\top \mathbf{u} = 1} \|\mathbf{u}\|_2 \|\nabla_{\mathbf{x}} f(\mathbf{x})\|_2 \cos \theta \end{aligned}$$

where θ is the angle between \mathbf{u} and the gradient. Substituting in $\|\mathbf{u}\|_2 = 1$ and ignoring factors that don't depend on \mathbf{u} , this simplifies to $\min_{\mathbf{u}} \cos \theta$. This is minimized when \mathbf{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

$$\mathbf{x}' = \mathbf{x} - \epsilon \nabla_{\mathbf{x}} f(\mathbf{x})$$

where ϵ is 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 $f(\mathbf{x} - \epsilon \nabla_{\mathbf{x}} f(\mathbf{x}))$ 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 $\nabla_{\mathbf{x}} f(\mathbf{x}) = 0$ for \mathbf{x} .

Sometimes we need to find all of the partial derivatives of all of the elements of a vector-valued function. The matrix containing all such partial derivatives is known as a *Jacobian matrix*. Specifically, if we have a function $\mathbf{f} : \mathbb{R}^m \rightarrow \mathbb{R}^n$, then the Jacobian matrix $\mathbf{J} \in \mathbb{R}^{n \times m}$ of \mathbf{f} is defined such that $J_{i,j} = \frac{\partial}{\partial x_j} f(\mathbf{x})_i$.

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 $\frac{\partial^2}{\partial x_i \partial x_j} f$.

In a single dimension, we can denote $\frac{d^2}{dx^2}f$ by $f''(x)$.

The second derivative tells us how the first derivative will change as we vary the input. This means it can be useful for determining 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. These derivatives can be collected together into a matrix called the *Hessian matrix*. The Hessian matrix $\mathbf{H}(f)(\mathbf{x})$ is defined such that

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

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:

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

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.

Using the eigendecomposition of the Hessian matrix, we can generalize the second derivative test to multiple dimensions. At a critical point, where $\nabla_{\mathbf{x}} f(\mathbf{x}) = 0$, 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 definite¹, 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 definite², the point is a local maximum. In multiple dimensions, it is

¹all its eigenvalues are positive

²all its eigenvalues are negative

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 \mathbf{x} is a local maximum on one cross section of f but a local minimum on another cross section. See Fig. 4.4 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.

The Hessian can also be useful for understanding the performance of gradient descent. 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. See Fig. 4.5 for an example.

This issue can be resolved by using information from the Hessian matrix to 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 $f(\mathbf{x})$ near some point \mathbf{x}_0 , ignoring derivatives of higher order:

$$f(\mathbf{x}) \approx f(\mathbf{x}_0) + (\mathbf{x} - \mathbf{x}_0)^\top \nabla_{\mathbf{x}} f(\mathbf{x}_0) + \frac{1}{2}(\mathbf{x} - \mathbf{x}_0)^\top H(f)(\mathbf{x}_0)(\mathbf{x} - \mathbf{x}_0).$$

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

$$\mathbf{x}^* = \mathbf{x}_0 - H(f)(\mathbf{x}_0)^{-1} \nabla_{\mathbf{x}} f(\mathbf{x}_0).$$

When the function can be locally approximated as quadratic, 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 Section 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 can in principle escape a saddle point, although it may take a lot of time if the negative eigenvalues are very small in magnitude, producing a kind of plateau around the saddle point.

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

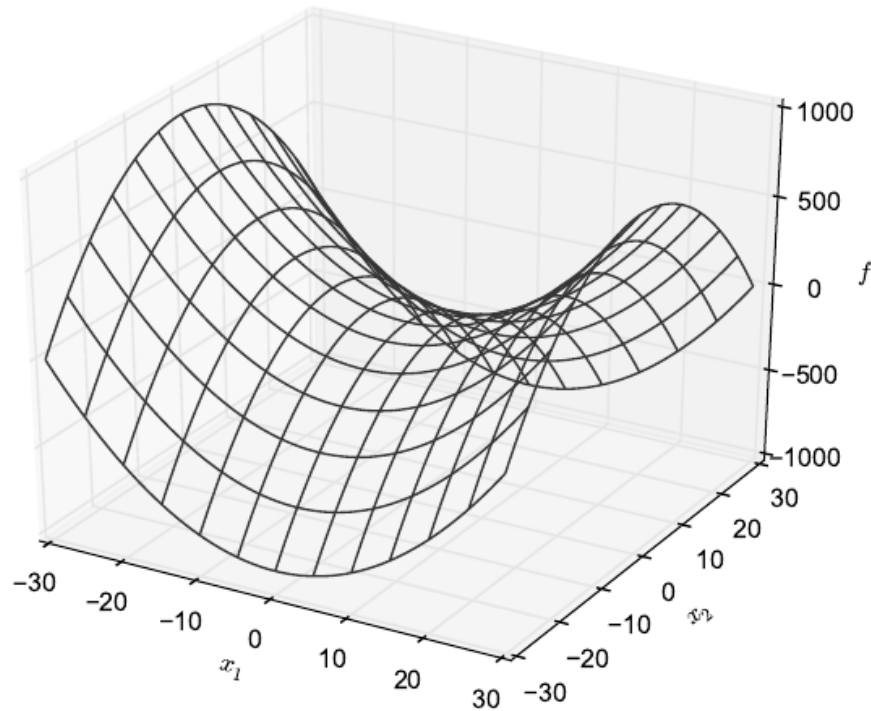


Figure 4.4: A saddle point containing both positive and negative curvature. The function in this example is $f(\mathbf{x}) = x_1^2 - x_2^2$. Along the axis corresponding to x_1 , the function curves upward. This axis is an eigenvector of the Hessian and has a positive eigenvalue. Along the axis corresponding to x_2 , 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. Note that 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.

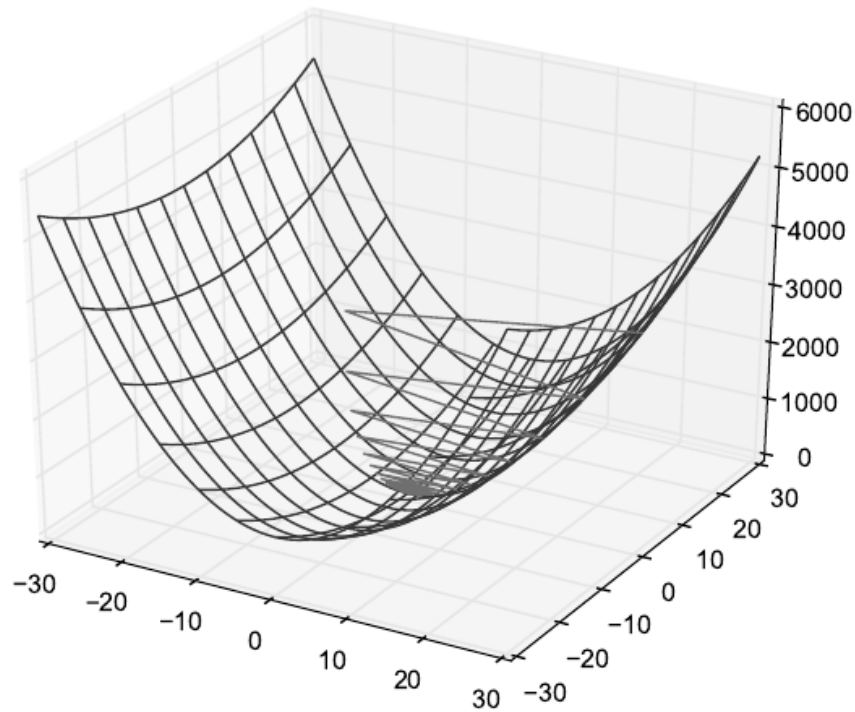


Figure 4.5: Gradient descent fails to exploit the curvature information contained in the Hessian matrix. Here we use gradient descent on a quadratic function whose Hessian matrix has condition number 5 (curvature is 5 times larger in one direction than in some other direction). The lines above the mesh 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.

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. Perhaps the most successful field of specialized optimization is *convex optimization*. Convex optimization algorithms are able to provide many more guarantees, but are applicable only to functions for which the Hessian is positive definite everywhere. Such functions are well-behaved because they lack saddle points and all of their local minima are necessarily global minima. However, most problems in deep learning are difficult to express in terms of convex optimization. Convex optimization is used only as a subroutine of some deep learning algorithms. Ideas from the analysis of convex optimization algorithms can be useful for proving the convergence of deep learning algorithms. However, in general, the importance of convex optimization is greatly diminished in the context of deep learning. For more information about convex optimization, see Boyd and Vandenberghe (2004) or Rockafellar (1997).

4.4 Constrained Optimization

Sometimes we wish not only to maximize or minimize a function $f(\mathbf{x})$ over all possible values of \mathbf{x} . Instead we may wish to find the maximal or minimal value of $f(\mathbf{x})$ for values of \mathbf{x} in some set S . This is known as *constrained optimization*. Points \mathbf{x} that lie within the set S are called *feasible* points in constrained optimization terminology.

One simple approach to constrained optimization is simply to modify gradient descent taking the constraint into account. If we use a small constant step size ϵ , we can make gradient descent steps, then project the result back into S . If we use a line search (see previous section), we can search only over step sizes ϵ that yield new \mathbf{x} points that are feasible, or we can project each point on the line back into the constraint region. When possible, this method can be made more efficient by projecting the gradient into the tangent space of the feasible region before taking the step or beginning the line search (Rosen, 1960).

A more sophisticated approach is to design a different, unconstrained optimization problem whose solution can be converted into a solution to the original, constrained optimization problem. For example, if we want to minimize $f(\mathbf{x})$ for $\mathbf{x} \in \mathbb{R}^2$ with \mathbf{x} constrained to have exactly unit L^2 norm, we can instead minimize $g(\theta) = f([\cos \theta, \sin \theta]^T)$ with respect to θ , then return $[\cos \theta, \sin \theta]$ as the solution to the original problem. This approach requires creativity; the transformation between optimization problems must be designed specifically for each case we encounter.

The *Karush–Kuhn–Tucker (KKT) approach* ³ provides a very general solu-

³The KKT approach generalizes the method of *Lagrange multipliers* which only allows equal-

tion to constrained optimization. With the KKT approach, we introduce a new function called the *generalized Lagrangian* or *generalized Lagrange function*.

To define the Lagrangian, we first need to describe S in terms of equations and inequalities. We want a description of S in terms of m functions g_i and n functions h_j so that $S = \{\mathbf{x} \mid \forall i, g_i(\mathbf{x}) = 0 \text{ and } \forall j, h_j(\mathbf{x}) \leq 0\}$. The equations involving g_i are called the *equality constraints* and the inequalities involving h_j are called *inequality constraints*.

We introduce new variables λ_i and α_j for each constraint, these are called the KKT multipliers. The generalized Lagrangian is then defined as

$$L(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\alpha}) = f(\mathbf{x}) + \sum_i \lambda_i g_i(\mathbf{x}) + \sum_j \alpha_j h_j(\mathbf{x}).$$

We can now solve a constrained minimization problem using unconstrained optimization of the generalized Lagrangian. Observe that, so long as at least one feasible point exists and $f(\mathbf{x})$ is not permitted to have value ∞ , then

$$\min_{\mathbf{x}} \max_{\boldsymbol{\lambda}} \max_{\boldsymbol{\alpha}, \boldsymbol{\alpha} \geq 0} L(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\alpha}).$$

has the same optimal objective function value and set of optimal points \mathbf{x} as

$$\min_{\mathbf{x} \in S} f(\mathbf{x}).$$

This follows because any time the constraints are satisfied,

$$\max_{\boldsymbol{\lambda}} \max_{\boldsymbol{\alpha}, \boldsymbol{\alpha} \geq 0} L(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\alpha}) = f(\mathbf{x}),$$

while any time a constraint is violated,

$$\max_{\boldsymbol{\lambda}} \max_{\boldsymbol{\alpha}, \boldsymbol{\alpha} \geq 0} L(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\alpha}) = \infty.$$

These properties guarantee that no infeasible point will ever be optimal, and that the optimum within the feasible points is unchanged.

To perform constrained maximization, we can construct the generalized Lagrange function of $-f(\mathbf{x})$, which leads to this optimization problem:

$$\min_{\mathbf{x}} \max_{\boldsymbol{\lambda}} \max_{\boldsymbol{\alpha}, \boldsymbol{\alpha} \geq 0} -f(\mathbf{x}) + \sum_i \lambda_i g_i(\mathbf{x}) + \sum_j \alpha_j h_j(\mathbf{x}).$$

We may also convert this to a problem with maximization in the outer loop:

$$\max_{\mathbf{x}} \min_{\boldsymbol{\lambda}} \min_{\boldsymbol{\alpha}, \boldsymbol{\alpha} \geq 0} f(\mathbf{x}) + \sum_i \lambda_i g_i(\mathbf{x}) - \sum_j \alpha_j h_j(\mathbf{x}).$$

ity constraints

$$\sum_i \quad \sum_j$$

Note that the sign of the term for the equality constraints does not matter; we may define it with addition or subtraction as we wish, because the optimization is free to choose any sign for each λ_i .

The inequality constraints are particularly interesting. We say that a constraint $h_i(\mathbf{x})$ is *active* if $h_i(\mathbf{x}^*) = 0$. If a constraint is not active, then the solution to the problem is the same whether or not that constraint exists. Because an inactive h_i has negative value, then the solution to $\min_{\mathbf{x}} \max_{\boldsymbol{\lambda}} \max_{\boldsymbol{\alpha}, \alpha \geq 0} L(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\alpha})$ will have $\alpha_i = 0$. We can thus observe that at the solution, $\boldsymbol{\alpha} \mathbf{h}(\mathbf{x}) = \mathbf{0}$. In other words, for all i , we know that at least one of the constraints $\alpha_i \geq 0$ and $h_i(\mathbf{x}) \leq 0$ must be active at the solution. To gain some intuition for this idea, we can say that either the solution is on the boundary imposed by the inequality and we must use its KKT multiplier to influence the solution to \mathbf{x} , or the inequality has no influence on the solution and we represent this by zeroing out its KKT multiplier.

The properties that the gradient of the generalized Lagrangian is zero, all constraints on both \mathbf{x} and the KKT multipliers are satisfied, and $\boldsymbol{\alpha} \odot \mathbf{h}(\mathbf{x}) = \mathbf{0}$ are called the *Karush-Kuhn-Tucker* (KKT) conditions (Karush, 1939; Kuhn and Tucker, 1951). Together, these properties describe the optimal points of constrained optimization problems.

In the case where there are no inequality constraints, the KKT approach simplifies to the method of Lagrange multipliers. For more information about the KKT approach, see Nocedal and Wright (2006).

4.5 Example: Linear Least Squares

Suppose we want to find the value of \mathbf{x} that minimizes

$$f(\mathbf{x}) = \frac{1}{2} \|\mathbf{Ax} - \mathbf{b}\|_2^2.$$

There are specialized linear algebra algorithms that can solve this problem efficiently. However, we can also explore how to solve it using gradient-based optimization as a simple example of how these techniques work.

First, we need to obtain the gradient:

$$\nabla_{\mathbf{x}} f(\mathbf{x}) = \mathbf{A}^\top (\mathbf{Ax} - \mathbf{b}) = \mathbf{A}^\top \mathbf{Ax} - \mathbf{A}^\top \mathbf{b}.$$

We can then follow this gradient downhill, taking small steps. See Algorithm 4.1 for details.

One can also solve this problem using Newton's method. In this case, because the true function is quadratic, the quadratic approximation employed by Newton's

Algorithm 4.1 An algorithm to minimize $f(\mathbf{x}) = \frac{1}{2}\|\mathbf{Ax} - \mathbf{b}\|_2^2$ with respect to \mathbf{x} using gradient descent.

Set ϵ , the step size, and δ , the tolerance, to small, positive numbers.

while $\|\mathbf{A}^\top \mathbf{Ax} - \mathbf{A}^\top \mathbf{b}\|_2 > \delta$ **do**

$\mathbf{x} \leftarrow \mathbf{x} - \epsilon (\mathbf{A}^\top \mathbf{Ax} - \mathbf{A}^\top \mathbf{b})$

end while

method is exact, and the algorithm converges to the global minimum in a single step.

Now suppose we wish to minimize the same function, but subject to the constraint $\mathbf{x}^\top \mathbf{x} \leq 1$. To do so, we introduce the Lagrangian

$$L(\mathbf{x}, \lambda) = f(\mathbf{x}) + \lambda (\mathbf{x}^\top \mathbf{x} - 1).$$

We can now solve the problem

$$\min_{\mathbf{x}} \max_{\lambda, \lambda \geq 0} L(\mathbf{x}, \lambda).$$

The solution to the unconstrained least squares problem is given by $\mathbf{x} = \mathbf{A}^+ \mathbf{b}$. If this point is feasible, then it is the solution to the constrained problem. Otherwise, we must find a solution where the constraint is active. By differentiating the Lagrangian with respect to \mathbf{x} , we obtain the equation

$$\mathbf{A}^\top \mathbf{Ax} - \mathbf{A}^\top \mathbf{b} + 2\lambda \mathbf{x} = 0.$$

This tells us that the solution will take the form

$$\mathbf{x} = (\mathbf{A}^\top \mathbf{A} + 2\lambda \mathbf{I})^{-1} \mathbf{A}^\top \mathbf{b}.$$

The magnitude of λ must be chosen such that the result obeys the constraint. We can find this value by performing gradient ascent on λ . To do so, observe

$$\frac{\partial}{\partial \lambda} L(\mathbf{x}, \lambda) = \mathbf{x}^\top \mathbf{x} - 1.$$

When the norm of \mathbf{x} exceeds 1, this derivative is positive, so to ascend the gradient and increase the Lagrangian with respect to λ , we increase λ . This will in turn shrink the optimal \mathbf{x} . The process continues until \mathbf{x} has the correct norm and the derivative on λ is 0.