# 8 Optimization

*Parts of this chapter were written by Frederik Kunstner, Si Yi Meng, Aaron Mishkin, Sharan Vaswani, and Mark Schmidt.*

## 8.1 Introduction

We saw in Chapter 4 that the core problem in machine learning is parameter estimation (aka model fitting). This requires solving an **optimization problem**, where we try to find the values for a set of variables $\boldsymbol{\theta} \in \Theta$, that minimize a scalar-valued **loss function** or **cost function** $\mathcal{L} : \Theta \to \mathbb{R}$:

$$\boldsymbol{\theta}^* \in \underset{\boldsymbol{\theta} \in \Theta}{\operatorname{argmin}} \, \mathcal{L}(\boldsymbol{\theta}) \tag{8.1}$$

We will assume that the **parameter space** is given by $\Theta \subseteq \mathbb{R}^D$, where $D$ is the number of variables being optimized over. Thus we are focusing on **continuous optimization**, rather than **discrete optimization**.

If we want to *maximize* a **score function** or **reward function** $R(\boldsymbol{\theta})$, we can equivalently minimize $\mathcal{L}(\boldsymbol{\theta}) = -R(\boldsymbol{\theta})$. We will use the term **objective function** to refer generically to a function we want to maximize or minimize. An algorithm that can find an optimum of an objective function is often called a **solver**.

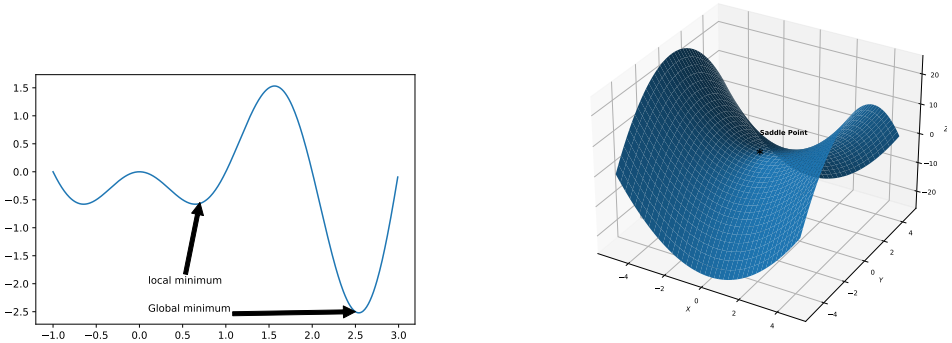
In the rest of this chapter, we discuss different kinds of solvers for different kinds of objective functions, with a focus on methods used in the machine learning community. For more details on optimization, please consult some of the many excellent textbooks, such as [KW19b; BV04; NW06; Ber15; Ber16] as well as various review articles, such as [BCN18; Sun+19b; PPS18; Pey20].

### 8.1.1 Local vs global optimization

A point that satisfies Equation (8.1) is called a **global optimum**. Finding such a point is called **global optimization**.

In general, finding global optima is computationally intractable [Neu04]. In such cases, we will just try to find a **local optimum**. For continuous problems, this is defined to be a point $\boldsymbol{\theta}^*$ which has lower (or equal) cost than "nearby" points. Formally, we say $\boldsymbol{\theta}^*$ is a **local minimum** if

$$\exists \delta > 0, \, \forall \boldsymbol{\theta} \in \Theta \;\; \text{s.t.} \;\; ||\boldsymbol{\theta} - \boldsymbol{\theta}^*|| < \delta, \; \mathcal{L}(\boldsymbol{\theta}^*) \leq \mathcal{L}(\boldsymbol{\theta}) \tag{8.2}$$




(a) (b)

*Figure 8.1: (a) Illustration of local and global minimum in 1d. Generated by extrema_fig_1d.ipynb. (b) Illustration of a saddle point in 2d. Generated by saddle.ipynb.*

A local minimum could be surrounded by other local minima with the same objective value; this is known as a **flat local minimum**. A point is said to be a **strict local minimum** if its cost is strictly lower than those of neighboring points:

$$\exists \delta > 0,\ \forall \boldsymbol{\theta} \in \Theta, \boldsymbol{\theta} \neq \boldsymbol{\theta}^* : ||\boldsymbol{\theta} - \boldsymbol{\theta}^*|| < \delta,\ \mathcal{L}(\boldsymbol{\theta}^*) < \mathcal{L}(\boldsymbol{\theta}) \tag{8.3}$$

We can define a (strict) **local maximum** analogously. See Figure 8.1a for an illustration.

A final note on terminology; if an algorithm is guaranteed to converge to a stationary point from any starting point, it is called **globally convergent**. However, this does not mean (rather confusingly) that it will converge to a global optimum; instead, it just means it will converge to some stationary point.

#### 8.1.1.1 Optimality conditions for local vs global optima

For continuous, twice differentiable functions, we can precisely characterize the points which correspond to local minima. Let $\boldsymbol{g}(\boldsymbol{\theta}) = \nabla \mathcal{L}(\boldsymbol{\theta})$ be the gradient vector, and $\mathbf{H}(\boldsymbol{\theta}) = \nabla^2 \mathcal{L}(\boldsymbol{\theta})$ be the Hessian matrix. (See Section 7.8 for a refresher on these concepts, if necessary.) Consider a point $\boldsymbol{\theta}^* \in \mathbb{R}^D$, and let $\boldsymbol{g}^* = \boldsymbol{g}(\boldsymbol{\theta})|_{\boldsymbol{\theta}^*}$ be the gradient at that point, and $\mathbf{H}^* = \mathbf{H}(\boldsymbol{\theta})|_{\boldsymbol{\theta}^*}$ be the corresponding Hessian. One can show that the following conditions characterize every local minimum:

- Necessary condition: If $\boldsymbol{\theta}^*$ is a local minimum, then we must have $\boldsymbol{g}^* = \mathbf{0}$ (i.e., $\boldsymbol{\theta}^*$ must be a **stationary point**), and $\mathbf{H}^*$ must be positive semi-definite.
- Sufficient condition: If $\boldsymbol{g}^* = \mathbf{0}$ and $\mathbf{H}^*$ is positive definite, then $\boldsymbol{\theta}^*$ is a local optimum.

To see why the first condition is necessary, suppose we were at a point $\boldsymbol{\theta}^*$ at which the gradient is non-zero: at such a point, we could decrease the function by following the negative gradient a small distance, so this would not be optimal. So the gradient must be zero. (In the case of nonsmooth

functions, the necessary condition is that the zero is a local subgradient at the minimum.) To see why a zero gradient is not sufficient, note that the stationary point could be a local minimum, maximum or **saddle point**, which is a point where some directions point downhill, and some uphill (see Figure 8.1b). More precisely, at a saddle point, the eigenvalues of the Hessian will be both positive and negative. However, if the Hessian at a point is positive semi-definite, then some directions may point uphill, while others are flat. Moreover, if the Hessian is strictly positive definite, then we are at the bottom of a "bowl", and all directions point uphill, which is sufficient for this to be a minimum.

### 8.1.2 Constrained vs unconstrained optimization

In **unconstrained optimization**, we define the optimization task as finding any value in the parameter space $\Theta$ that minimizes the loss. However, we often have a set of **constraints** on the allowable values. It is standard to partition the set of constraints $\mathcal{C}$ into **inequality constraints**, $g_j(\boldsymbol{\theta}) \leq 0$ for $j \in \mathcal{I}$ and **equality constraints**, $h_k(\boldsymbol{\theta}) = 0$ for $k \in \mathcal{E}$. For example, we can represent a sum-to-one constraint as an equality constraint $h(\boldsymbol{\theta}) = (1 - \sum_{i=1}^{D} \theta_i) = 0$, and we can represent a non-negativity constraint on the parameters by using $D$ inequality constraints of the form $g_i(\boldsymbol{\theta}) = -\theta_i \leq 0$ .

We define the **feasible set** as the subset of the parameter space that satisfies the constraints:

$$\mathcal{C} = \{\boldsymbol{\theta} : g_j(\boldsymbol{\theta}) \leq 0 : j \in \mathcal{I}, h_k(\boldsymbol{\theta}) = 0 : k \in \mathcal{E}\} \subseteq \mathbb{R}^D \tag{8.4}$$

Our **constrained optimization** problem now becomes

$$\boldsymbol{\theta}^* \in \operatorname*{argmin}_{\boldsymbol{\theta} \in \mathcal{C}} \mathcal{L}(\boldsymbol{\theta}) \tag{8.5}$$

If $\mathcal{C} = \mathbb{R}^D$, it is called **unconstrained optimization**.

The addition of constraints can change the number of optima of a function. For example, a function that was previously unbounded (and hence had no well-defined global maximum or minimum) can "acquire" multiple maxima or minima when we add constraints, as illustrated in Figure 8.2. However, if we add too many constraints, we may find that the feasible set becomes empty. The task of finding any point (regardless of its cost) in the feasible set is called a **feasibility problem**; this can be a hard subproblem in itself.

A common strategy for solving constrained problems is to create penalty terms that measure how much we violate each constraint. We then add these terms to the objective and solve an unconstrained optimization problem. The **Lagrangian** is a special case of such a combined objective (see Section 8.5 for details).

### 8.1.3 Convex vs nonconvex optimization

In **convex optimization**, we require the objective to be a convex function defined over a convex set (we define these terms below). In such problems, every local minimum is also a global minimum. Thus many models are designed so that their training objectives are convex.

#### 8.1.3.1 Convex sets

We say $\mathcal{S}$ is a **convex set** if, for any $\boldsymbol{x}, \boldsymbol{x}' \in \mathcal{S}$, we have

$$\lambda \boldsymbol{x} + (1 - \lambda)\boldsymbol{x}' \in \mathcal{S}, \quad \forall\, \lambda \in [0, 1] \tag{8.6}$$

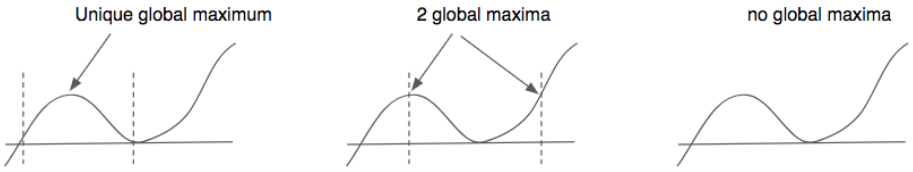



*Figure 8.2: Illustration of constrained maximization of a nonconvex 1d function. The area between the dotted vertical lines represents the feasible set. (a) There is a unique global maximum since the function is concave within the support of the feasible set. (b) There are two global maxima, both occuring at the boundary of the feasible set. (c) In the unconstrained case, this function has no global maximum, since it is unbounded.*

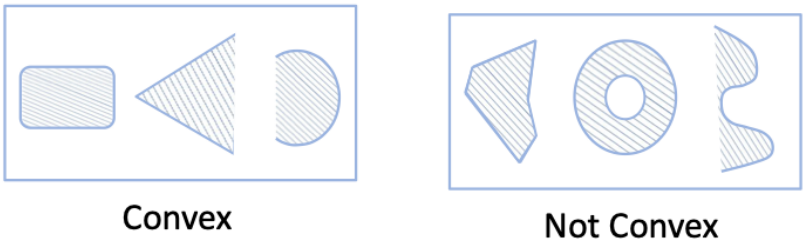



*Figure 8.3: Illustration of some convex and non-convex sets.*

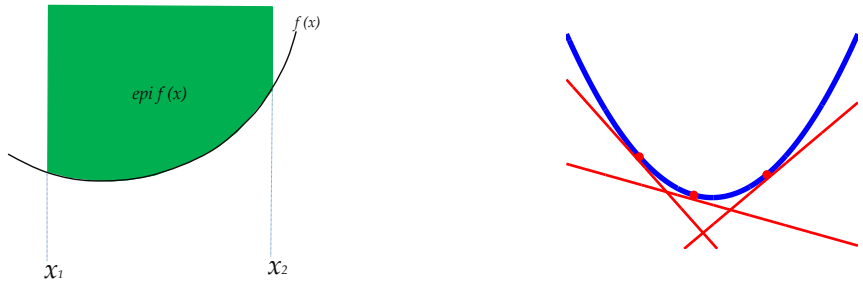
That is, if we draw a line from $\boldsymbol{x}$ to $\boldsymbol{x}'$, all points on the line lie inside the set. See Figure 8.3 for some illustrations of convex and non-convex sets.

#### 8.1.3.2 Convex functions

We say $f$ is a **convex function** if its **epigraph** (the set of points above the function, illustrated in Figure 8.4a) defines a convex set. Equivalently, a function $f(\boldsymbol{x})$ is called convex if it is defined on a convex set and if, for any $\boldsymbol{x}, \boldsymbol{y} \in \mathcal{S}$, and for any $0 \leq \lambda \leq 1$, we have

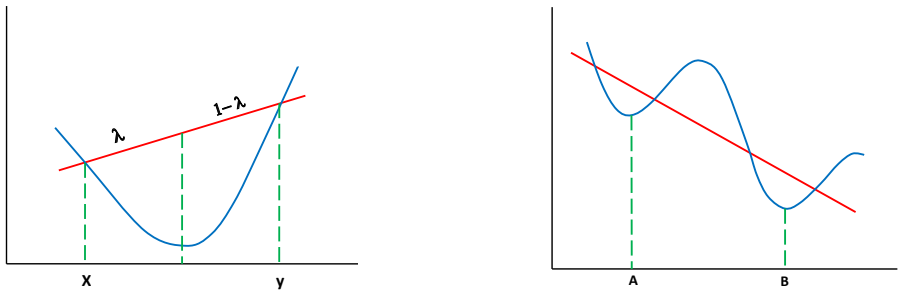
$$f(\lambda \boldsymbol{x} + (1-\lambda)\boldsymbol{y}) \leq \lambda f(\boldsymbol{x}) + (1-\lambda) f(\boldsymbol{y}) \tag{8.7}$$

See Figure 8.5(a) for a 1d example of a convex function. A function is called **strictly convex** if the inequality is strict. A function $f(\boldsymbol{x})$ is **concave** if $-f(\boldsymbol{x})$ is convex, and **strictly concave** if $-f(\boldsymbol{x})$ is strictly convex. See Figure 8.5(b) for a 1d example of a function that is neither convex nor concave.




(a) (b)

*Figure 8.4: (a) Illustration of the epigraph of a function. (b) For a convex function $f(x)$, its epipgraph can be represented as the intersection of half-spaces defined by linear lower bounds derived from the* **conjugate function** $f^*(\lambda) = \max_x \lambda x - f(x)$.




(a) (b)

*Figure 8.5: (a) Illustration of a convex function. We see that the chord joining $(x, f(x))$ to $(y, f(y))$ lies above the function. (b) A function that is neither convex nor concave.* **A** *is a local minimum,* **B** *is a global minimum.*

Here are some examples of 1d convex functions:

$$\begin{aligned} x^2 \\ e^{ax} \\ -\log x \\ x^a,\ a > 1, x > 0 \\ |x|^a,\ a \geq 1 \\ x \log x,\ x > 0 \end{aligned}$$

#### 8.1.3.3 Characterization of convex functions

Intuitively, a convex function is shaped like a bowl. Formally, one can prove the following important result:

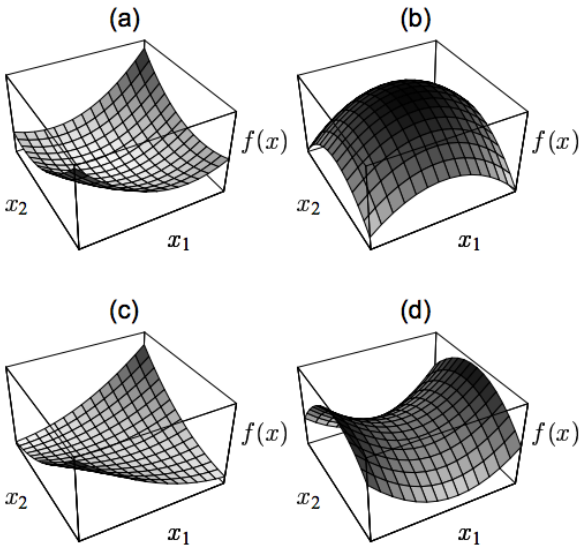



*Figure 8.6: The quadratic form $f(\boldsymbol{x}) = \boldsymbol{x}^\mathsf{T}\mathbf{A}\boldsymbol{x}$ in 2d. (a) $\mathbf{A}$ is positive definite, so $f$ is convex. (b) $\mathbf{A}$ is negative definite, so $f$ is concave. (c) $\mathbf{A}$ is positive semidefinite but singular, so $f$ is convex, but not strictly. Notice the valley of constant height in the middle. (d) $\mathbf{A}$ is indefinite, so $f$ is neither convex nor concave. The stationary point in the middle of the surface is a saddle point. From Figure 5 of [She94].*

**Theorem 8.1.1.** *Suppose $f : \mathbb{R}^n \to \mathbb{R}$ is twice differentiable over its domain. Then $f$ is convex iff $\mathbf{H} = \nabla^2 f(\boldsymbol{x})$ is positive semi definite (Section 7.1.5.3) for all $\boldsymbol{x} \in \text{dom}(f)$. Furthermore, $f$ is strictly convex if $\mathbf{H}$ is positive definite.*

For example, consider the quadratic form

$$f(\boldsymbol{x}) = \boldsymbol{x}^\mathsf{T}\mathbf{A}\boldsymbol{x} \tag{8.8}$$

This is convex if $\mathbf{A}$ is positive semi definite, and is strictly convex if $\mathbf{A}$ is positive definite. It is neither convex nor concave if $\mathbf{A}$ has eigenvalues of mixed sign. See Figure 8.6.

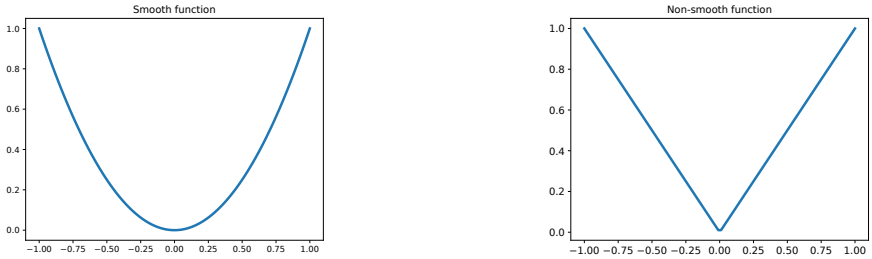
#### 8.1.3.4 Strongly convex functions

We say a function $f$ is **strongly convex** with parameter $m > 0$ if the following holds for all $\boldsymbol{x}$, $\boldsymbol{y}$ in $f$'s domain:

$$(\nabla f(\boldsymbol{x}) - \nabla f(\boldsymbol{y}))^\mathsf{T}(\boldsymbol{x} - \boldsymbol{y}) \geq m||\boldsymbol{x} - \boldsymbol{y}||_2^2 \tag{8.9}$$

A strongly convex function is also strictly convex, but not vice versa.

If the function $f$ is twice continuously differentiable, then it is strongly convex with parameter $m$ if and only if $\nabla^2 f(\boldsymbol{x}) \succeq m\mathbf{I}$ for all $\boldsymbol{x}$ in the domain, where $\mathbf{I}$ is the identity and $\nabla^2 f$ is the Hessian matrix, and the inequality $\succeq$ means that $\nabla^2 f(\boldsymbol{x}) - m\mathbf{I}$ is positive semi-definite. This is equivalent




(a) (b)

*Figure 8.7: (a) Smooth 1d function. (b) Non-smooth 1d function. (There is a discontinuity at the origin.) Generated by smooth-vs-nonsmooth-1d.ipynb.*

to requiring that the minimum eigenvalue of $\nabla^2 f(\boldsymbol{x})$ be at least $m$ for all $\boldsymbol{x}$. If the domain is just the real line, then $\nabla^2 f(x)$ is just the second derivative $f''(x)$, so the condition becomes $f''(x) \geq m$. If $m = 0$, then this means the Hessian is positive semidefinite (or if the domain is the real line, it means that $f''(x) \geq 0$), which implies the function is convex, and perhaps strictly convex, but not strongly convex.

The distinction between convex, strictly convex, and strongly convex is rather subtle. To better understand this, consider the case where $f$ is twice continuously differentiable and the domain is the real line. Then we can characterize the differences as follows:

- $f$ is convex if and only if $f''(x) \geq 0$ for all $x$.
- $f$ is strictly convex if $f''(x) > 0$ for all $x$ (note: this is sufficient, but not necessary).
- $f$ is strongly convex if and only if $f''(x) \geq m > 0$ for all $x$.

Note that it can be shown that a function $f$ is strongly convex with parameter $m$ iff the function

$$J(\boldsymbol{x}) = f(\boldsymbol{x}) - \frac{m}{2}||\boldsymbol{x}||^2 \tag{8.10}$$

is convex.

### 8.1.4 Smooth vs nonsmooth optimization

In **smooth optimization**, the objective and constraints are continuously differentiable functions. For smooth functions, we can quantify the degree of smoothness using the **Lipschitz constant**. In the 1d case, this is defined as any constant $L \geq 0$ such that, for all real $x_1$ and $x_2$, we have

$$|f(x_1) - f(x_2)| \leq L|x_1 - x_2| \tag{8.11}$$

This is illustrated in Figure 8.8: for a given constant $L$, the function output cannot change by more than $L$ if we change the function input by 1 unit. This can be generalized to vector inputs using a suitable norm.

In **nonsmooth optimization**, there are at least some points where the gradient of the objective function or the constraints is not well-defined. See Figure 8.7 for an example. In some optimization

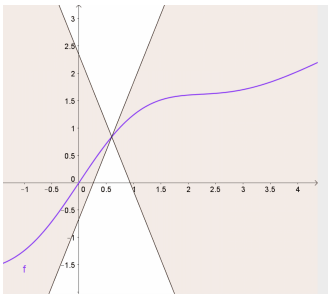


*Figure 8.8: For a Lipschitz continuous function $f$, there exists a double cone (white) whose origin can be moved along the graph of $f$ so that the whole graph always stays outside the double cone. From https://en.wikipedia.org/wiki/Lipschitz_continuity. Used with kind permission of Wikipedia author Taschee.*

problems, we can partition the objective into a part that only contains smooth terms, and a part that contains the nonsmooth terms:

$$\mathcal{L}(\boldsymbol{\theta}) = \mathcal{L}_s(\boldsymbol{\theta}) + \mathcal{L}_r(\boldsymbol{\theta}) \tag{8.12}$$

where $\mathcal{L}_s$ is smooth (differentiable), and $\mathcal{L}_r$ is nonsmooth ("rough"). This is often referred to as a **composite objective**. In machine learning applications, $\mathcal{L}_s$ is usually the training set loss, and $\mathcal{L}_r$ is a regularizer, such as the $\ell_1$ norm of $\boldsymbol{\theta}$. This composite structure can be exploited by various algorithms.

#### 8.1.4.1 Subgradients

In this section, we generalize the notion of a derivative to work with functions which have local discontinuities. In particular, for a convex function of several variables, $f : \mathbb{R}^n \to \mathbb{R}$, we say that $\boldsymbol{g} \in \mathbb{R}^n$ is a **subgradient** of $f$ at $\boldsymbol{x} \in \text{dom}(f)$ if for all $\boldsymbol{z} \in \text{dom}(f)$,

$$f(\boldsymbol{z}) \geq f(\boldsymbol{x}) + \boldsymbol{g}^\mathsf{T}(\boldsymbol{z} - \boldsymbol{x}) \tag{8.13}$$

Note that a subgradient can exist even when $f$ is not differentiable at a point, as shown in Figure 8.9.

A function $f$ is called **subdifferentiable** at $\boldsymbol{x}$ if there is at least one subgradient at $\boldsymbol{x}$. The set of such subgradients is called the **subdifferential** of $f$ at $\boldsymbol{x}$, and is denoted $\partial f(\boldsymbol{x})$.

For example, consider the absolute value function $f(x) = |x|$. Its subdifferential is given by

$$\partial f(x) = \begin{cases} \{-1\} & \text{if } x < 0 \\ [-1, 1] & \text{if } x = 0 \\ \{+1\} & \text{if } x > 0 \end{cases} \tag{8.14}$$

where the notation $[-1, 1]$ means any value between -1 and 1 inclusive. See Figure 8.10 for an illustration.

## 8.2 First-order methods

In this section, we consider iterative optimization methods that leverage **first-order** derivatives of the objective function, i.e., they compute which directions point "downhill", but they ignore curvature

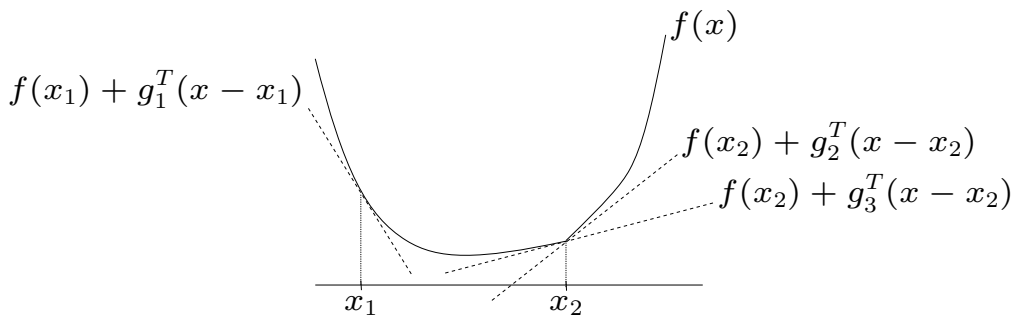



*Figure 8.9: Illustration of subgradients. At $\boldsymbol{x}_1$, the convex function $f$ is differentiable, and $\boldsymbol{g}_1$ (which is the derivative of $f$ at $\boldsymbol{x}_1$) is the unique subgradient at $\boldsymbol{x}_1$. At the point $\boldsymbol{x}_2$, $f$ is not differentiable, because of the "kink". However, there are many subgradients at this point, of which two are shown. From https://web.stanford.edu/class/ee364b/lectures/subgradients_slides.pdf. Used with kind permission of Stephen Boyd.*

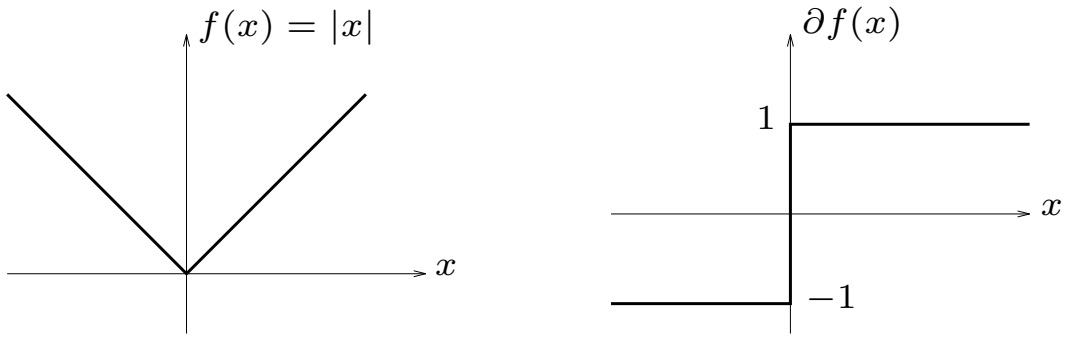



*Figure 8.10: The absolute value function (left) and its subdifferential (right). From https://web.stanford.edu/class/ee364b/lectures/subgradients_slides.pdf. Used with kind permission of Stephen Boyd.*

information. All of these algorithms require that the user specify a starting point $\boldsymbol{\theta}_0$. Then at each iteration $t$, they perform an update of the following form:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \eta_t \boldsymbol{d}_t \tag{8.15}$$

where $\eta_t$ is known as the **step size** or **learning rate**, and $\boldsymbol{d}_t$ is a **descent direction**, such as the negative of the **gradient**, given by $\boldsymbol{g}_t = \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta})|_{\boldsymbol{\theta}_t}$. These update steps are continued until the method reaches a stationary point, where the gradient is zero.

### 8.2.1 Descent direction

We say that a direction $\boldsymbol{d}$ is a **descent direction** if there is a small enough (but nonzero) amount $\eta$ we can move in direction $\boldsymbol{d}$ and be guaranteed to decrease the function value. Formally, we require that there exists an $\eta_{\max} > 0$ such that

$$\mathcal{L}(\boldsymbol{\theta} + \eta \boldsymbol{d}) < \mathcal{L}(\boldsymbol{\theta}) \tag{8.16}$$

for all $0 < \eta < \eta_{\max}$. The gradient at the current iterate, $\boldsymbol{\theta}_t$, is given by

$$\boldsymbol{g}_t \triangleq \nabla \mathcal{L}(\boldsymbol{\theta})|_{\boldsymbol{\theta}_t} = \nabla \mathcal{L}(\boldsymbol{\theta}_t) = \boldsymbol{g}(\boldsymbol{\theta}_t) \tag{8.17}$$

This points in the direction of maximal increase in $f$, so the negative gradient is a descent direction. It can be shown that any direction $\boldsymbol{d}$ is also a descent direction if the angle $\theta$ between $\boldsymbol{d}$ and $-\boldsymbol{g}_t$ is less than 90 degrees and satisfies

$$\boldsymbol{d}^{\mathsf{T}} \boldsymbol{g}_t = ||\boldsymbol{d}||\ ||\boldsymbol{g}_t||\ \cos(\theta) < 0 \tag{8.18}$$

It seems that the best choice would be to pick $\boldsymbol{d}_t = -\boldsymbol{g}_t$. This is known as the direction of **steepest descent**. However, this can be quite slow. We consider faster versions later.

### 8.2.2 Step size (learning rate)

In machine learning, the sequence of step sizes $\{\eta_t\}$ is called the **learning rate schedule**. There are several widely used methods for picking this, some of which we discuss below. (See also Section 8.4.3, where we discuss schedules for stochastic optimization.)

#### 8.2.2.1 Constant step size

The simplest method is to use a constant step size, $\eta_t = \eta$. However, if it is too large, the method may fail to converge, and if it is too small, the method will converge but very slowly.

For example, consider the convex function

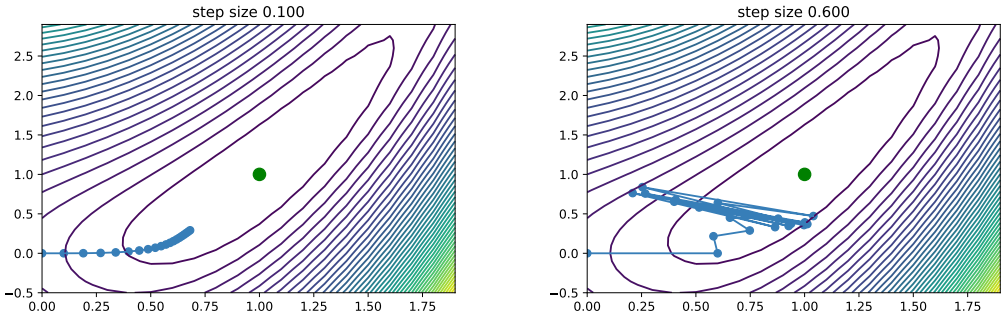
$$\mathcal{L}(\boldsymbol{\theta}) = 0.5(\theta_1^2 - \theta_2)^2 + 0.5(\theta_1 - 1)^2 \tag{8.19}$$

Let us pick as our descent direction $\boldsymbol{d}_t = -\boldsymbol{g}_t$. Figure 8.11 shows what happens if we use this descent direction with a fixed step size, starting from $(0, 0)$. In Figure 8.11(a), we use a small step size of $\eta = 0.1$; we see that the iterates move slowly along the valley. In Figure 8.11(b), we use a larger step size $\eta = 0.6$; we see that the iterates start oscillating up and down the sides of the valley and never converge to the optimum, even though this is a convex problem.

In some cases, we can derive a theoretical upper bound on the maximum step size we can use. For example, consider a quadratic objective, $\mathcal{L}(\boldsymbol{\theta}) = \frac{1}{2}\boldsymbol{\theta}^{\mathsf{T}}\mathbf{A}\boldsymbol{\theta} + \boldsymbol{b}^{\mathsf{T}}\boldsymbol{\theta} + c$ with $\mathbf{A} \succeq \mathbf{0}$. One can show that steepest descent will have global convergence iff the step size satisfies

$$\eta < \frac{2}{\lambda_{\max}(\mathbf{A})} \tag{8.20}$$

where $\lambda_{\max}(\mathbf{A})$ is the largest eigenvalue of $\mathbf{A}$. The intuitive reason for this can be understood by thinking of a ball rolling down a valley. We want to make sure it doesn't take a step that is larger than the slope of the steepest direction, which is what the largest eigenvalue measures (see Section 3.2.2).




*(a)* *(b)*

*Figure 8.11: Steepest descent on a simple convex function, starting from* $(0, 0)$*, for 20 steps, using a fixed step size. The global minimum is at* $(1, 1)$*. (a)* $\eta = 0.1$*. (b)* $\eta = 0.6$*. Generated by steepestDescentDemo.ipynb.*

More generally, setting $\eta < 2/L$, where $L$ is the Lipschitz constant of the gradient (Section 8.1.4), ensures convergence. Since this constant is generally unknown, we usually need to adapt the step size, as we discuss below.

#### 8.2.2.2 Line search

The optimal step size can be found by finding the value that maximally decreases the objective along the chosen direction by solving the 1d minimization problem

$$\eta_t = \underset{\eta>0}{\operatorname{argmin}}\, \phi_t(\eta) = \underset{\eta>0}{\operatorname{argmin}}\, \mathcal{L}(\boldsymbol{\theta}_t + \eta \boldsymbol{d}_t) \tag{8.21}$$

This is known as **line search**, since we are searching along the line defined by $\boldsymbol{d}_t$.

If the loss is convex, this subproblem is also convex, because $\phi_t(\eta) = \mathcal{L}(\boldsymbol{\theta}_t + \eta \boldsymbol{d}_t)$ is a convex function of an affine function of $\eta$, for fixed $\boldsymbol{\theta}_t$ and $\boldsymbol{d}_t$. For example, consider the quadratic loss

$$\mathcal{L}(\boldsymbol{\theta}) = \frac{1}{2}\boldsymbol{\theta}^\mathsf{T}\mathbf{A}\boldsymbol{\theta} + \boldsymbol{b}^\mathsf{T}\boldsymbol{\theta} + c \tag{8.22}$$

Computing the derivative of $\phi$ gives

$$\frac{d\phi(\eta)}{d\eta} = \frac{d}{d\eta}\left[\frac{1}{2}(\boldsymbol{\theta} + \eta\boldsymbol{d})^\mathsf{T}\mathbf{A}(\boldsymbol{\theta} + \eta\boldsymbol{d}) + \boldsymbol{b}^\mathsf{T}(\boldsymbol{\theta} + \eta\boldsymbol{d}) + c\right] \tag{8.23}$$

$$= \boldsymbol{d}^\mathsf{T}\mathbf{A}(\boldsymbol{\theta} + \eta\boldsymbol{d}) + \boldsymbol{d}^\mathsf{T}\boldsymbol{b} \tag{8.24}$$

$$= \boldsymbol{d}^\mathsf{T}(\mathbf{A}\boldsymbol{\theta} + \boldsymbol{b}) + \eta\boldsymbol{d}^\mathsf{T}\mathbf{A}\boldsymbol{d} \tag{8.25}$$

Solving for $\frac{d\phi(\eta)}{d\eta} = 0$ gives

$$\eta = -\frac{\boldsymbol{d}^\mathsf{T}(\mathbf{A}\boldsymbol{\theta} + \boldsymbol{b})}{\boldsymbol{d}^\mathsf{T}\mathbf{A}\boldsymbol{d}} \tag{8.26}$$

Using the optimal step size is known as **exact line search**. However, it is not usually necessary to be so precise. There are several methods, such as the **Armijo backtracking method**, that try to ensure sufficient reduction in the objective function without spending too much time trying to solve Equation (8.21). In particular, we can start with the current stepsize (or some maximum value), and then reduce it by a factor $0 < c < 1$ at each step until we satisfy the following condition, known as the **Armijo-Goldstein** test:

$$\mathcal{L}(\boldsymbol{\theta}_t + \eta \boldsymbol{d}_t) \leq \mathcal{L}(\boldsymbol{\theta}_t) + c\eta \boldsymbol{d}_t^\mathsf{T} \nabla \mathcal{L}(\boldsymbol{\theta}_t) \tag{8.27}$$

where $c \in [0, 1]$ is a constant, typically $c = 10^{-4}$. In practice, the initialization of the line-search and how to backtrack can significantly affect performance. See [NW06, Sec 3.1] for details.

### 8.2.3 Convergence rates

We want to find optimization algorithms that converge quickly to a (local) optimum. For certain convex problems, with a gradient with bounded Lipschitz constant, one can show that gradient descent converges at a **linear rate**. This means that there exists a number $0 < \mu < 1$ such that

$$|\mathcal{L}(\boldsymbol{\theta}_{t+1}) - \mathcal{L}(\boldsymbol{\theta}_*)| \leq \mu |\mathcal{L}(\boldsymbol{\theta}_t) - \mathcal{L}(\boldsymbol{\theta}_*)| \tag{8.28}$$

Here $\mu$ is called the **rate of convergence**.

For some simple problems, we can derive the convergence rate explicitly, For example, consider a quadratic objective $\mathcal{L}(\boldsymbol{\theta}) = \frac{1}{2}\boldsymbol{\theta}^\mathsf{T}\mathbf{A}\boldsymbol{\theta} + \boldsymbol{b}^\mathsf{T}\boldsymbol{\theta} + c$ with $\mathbf{A} \succ 0$. Suppose we use steepest descent with exact line search. One can show (see e.g., [Ber15]) that the convergence rate is given by

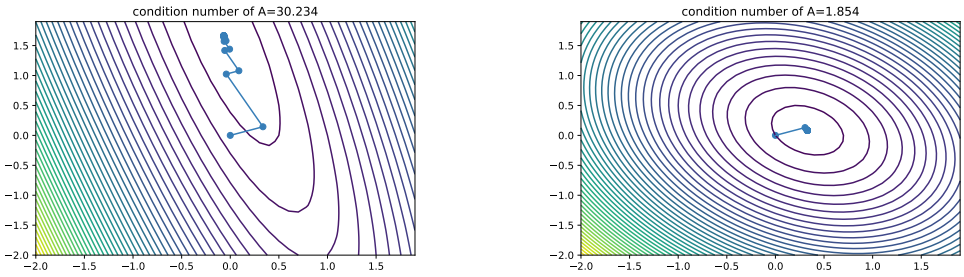
$$\mu = \left( \frac{\lambda_{\max} - \lambda_{\min}}{\lambda_{\max} + \lambda_{\min}} \right)^2 \tag{8.29}$$

where $\lambda_{\max}$ is the largest eigenvalue of $\mathbf{A}$ and $\lambda_{\min}$ is the smallest eigenvalue. We can rewrite this as $\mu = \left(\frac{\kappa-1}{\kappa+1}\right)^2$, where $\kappa = \frac{\lambda_{\max}}{\lambda_{\min}}$ is the condition number of $\mathbf{A}$. Intuitively, the condition number measures how "skewed" the space is, in the sense of being far from a symmetrical "bowl". (See Section 7.1.4.4 for more information on condition numbers.)

Figure 8.12 illustrates the effect of the condition number on the convergence rate. On the left we show an example where $\mathbf{A} = [20, 5; 5, 2]$, $\boldsymbol{b} = [-14; -6]$ and $c = 10$, so $\kappa(\mathbf{A}) = 30.234$. On the right we show an example where $\mathbf{A} = [20, 5; 5, 16]$, $\boldsymbol{b} = [-14; -6]$ and $c = 10$, so $\kappa(\mathbf{A}) = 1.8541$. We see that steepest descent converges much more quickly for the problem with the smaller condition number.

In the more general case of non-quadratic functions, the objective will often be locally quadratic around a local optimum. Hence the convergence rate depends on the condition number of the Hessian, $\kappa(\mathbf{H})$, at that point. We can often improve the convergence speed by optimizing a surrogate objective (or model) at each step which has a Hessian that is close to the Hessian of the objective function as we discuss in Section 8.3.

Although line search works well, we see from Figure 8.12 that the path of steepest descent with an exact line-search exhibits a characteristic **zig-zag** behavior, which is inefficient. This problem can be overcome using a method called **conjugate gradient** descent (see e.g., [She94]).




(a) (b)

*Figure 8.12: Illustration of the effect of condition number $\kappa$ on the convergence speed of steepest descent with exact line searches. (a) Large $\kappa$. (b) Small $\kappa$. Generated by lineSearchConditionNum.ipynb.*

### 8.2.4 Momentum methods

Gradient descent can move very slowly along flat regions of the loss landscape, as we illustrated in Figure 8.11. We discuss some solutions to this below.

#### 8.2.4.1 Momentum

One simple heuristic, known as the **heavy ball** or **momentum** method [Ber99], is to move faster along directions that were previously good, and to slow down along directions where the gradient has suddenly changed, just like a ball rolling downhill. This can be implemented as follows:

$$\boldsymbol{m}_t = \beta \boldsymbol{m}_{t-1} + \boldsymbol{g}_{t-1} \tag{8.30}$$

$$\boldsymbol{\theta}_t = \boldsymbol{\theta}_{t-1} - \eta_t \boldsymbol{m}_t \tag{8.31}$$

where $\boldsymbol{m}_t$ is the momentum (mass times velocity) and $0 < \beta < 1$. A typical value of $\beta$ is 0.9. For $\beta = 0$, the method reduces to gradient descent.

We see that $\boldsymbol{m}_t$ is like an exponentially weighted moving average of the past gradients (see Section 4.4.2.2):

$$\boldsymbol{m}_t = \beta \boldsymbol{m}_{t-1} + \boldsymbol{g}_{t-1} = \beta^2 \boldsymbol{m}_{t-2} + \beta \boldsymbol{g}_{t-2} + \boldsymbol{g}_{t-1} = \cdots = \sum_{\tau=0}^{t-1} \beta^\tau \boldsymbol{g}_{t-\tau-1} \tag{8.32}$$

If all the past gradients are a constant, say $\boldsymbol{g}$, this simplifies to

$$\boldsymbol{m}_t = \boldsymbol{g} \sum_{\tau=0}^{t-1} \beta^\tau \tag{8.33}$$

The scaling factor is a geometric series, whose infinite sum is given by

$$1 + \beta + \beta^2 + \cdots = \sum_{i=0}^{\infty} \beta^i = \frac{1}{1-\beta} \tag{8.34}$$

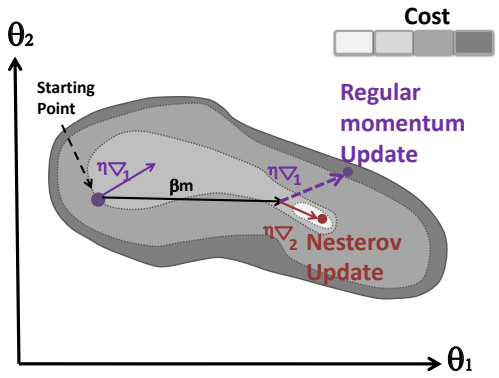



*Figure 8.13: Illustration of the Nesterov update. Adapted from Figure 11.6 of [Gér19].*

Thus in the limit, we multiply the gradient by $1/(1-\beta)$. For example, if $\beta = 0.9$, we scale the gradient up by 10.

Since we update the parameters using the gradient average $\boldsymbol{m}_{t-1}$, rather than just the most recent gradient, $\boldsymbol{g}_{t-1}$, we see that past gradients can exhibit some influence on the present. Furthermore, when momentum is combined with SGD, discussed in Section 8.4, we will see that it can simulate the effects of a larger minibatch, without the computational cost.

#### 8.2.4.2 Nesterov momentum

One problem with the standard momentum method is that it may not slow down enough at the bottom of a valley, causing oscillation. The **Nesterov accelerated gradient** method of [Nes04] instead modifies the gradient descent to include an extrapolation step, as follows:

$$\tilde{\boldsymbol{\theta}}_{t+1} = \boldsymbol{\theta}_t + \beta(\boldsymbol{\theta}_t - \boldsymbol{\theta}_{t-1}) \tag{8.35}$$

$$\boldsymbol{\theta}_{t+1} = \tilde{\boldsymbol{\theta}}_{t+1} - \eta_t \nabla \mathcal{L}(\tilde{\boldsymbol{\theta}}_{t+1}) \tag{8.36}$$

This is essentially a form of one-step "look ahead", that can reduce the amount of oscillation, as illustrated in Figure 8.13.

Nesterov accelerated gradient can also be rewritten in the same format as standard momentum. In this case, the momentum term is updated using the gradient at the predicted new location,

$$\boldsymbol{m}_{t+1} = \beta \boldsymbol{m}_t - \eta_t \nabla \mathcal{L}(\boldsymbol{\theta}_t + \beta \boldsymbol{m}_t) \tag{8.37}$$

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \boldsymbol{m}_{t+1} \tag{8.38}$$

This explains why the Nesterov accelerated gradient method is sometimes called Nesterov momentum. It also shows how this method can be faster than standard momentum: the momentum vector is already roughly pointing in the right direction, so measuring the gradient at the new location, $\boldsymbol{\theta}_t + \beta \boldsymbol{m}_t$, rather than the current location, $\boldsymbol{\theta}_t$, can be more accurate.

The Nesterov accelerated gradient method is provably faster than steepest descent for convex functions when $\beta$ and $\eta_t$ are chosen appropriately. It is called "accelerated" because of this improved

convergence rate, which is optimal for gradient-based methods using only first-order information when the objective function is convex and has Lipschitz-continuous gradients. In practice, however, using Nesterov momentum can be slower than steepest descent, and can even unstable if $\beta$ or $\eta_t$ are misspecified.

## 8.3 Second-order methods

Optimization algorithms that only use the gradient are called **first-order** methods. They have the advantage that the gradient is cheap to compute and to store, but they do not model the curvature of the space, and hence they can be slow to converge, as we have seen in Figure 8.12. **Second-order** optimization methods incorporate curvature in various ways (e.g., via the Hessian), which may yield faster convergence. We discuss some of these methods below.

### 8.3.1 Newton's method

The classic second-order method is **Newton's method**. This consists of updates of the form

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta_t \mathbf{H}_t^{-1} \boldsymbol{g}_t \tag{8.39}$$

where

$$\mathbf{H}_t \triangleq \nabla^2 \mathcal{L}(\boldsymbol{\theta})|_{\boldsymbol{\theta}_t} = \nabla^2 \mathcal{L}(\boldsymbol{\theta}_t) = \mathbf{H}(\boldsymbol{\theta}_t) \tag{8.40}$$

is assumed to be positive-definite to ensure the update is well-defined. The pseudo-code for Newton's method is given in Algorithm 8.1. The intuition for why this is faster than gradient descent is that the matrix inverse $\mathbf{H}^{-1}$ "undoes" any skew in the local curvature, converting a topology like Figure 8.12a to one like Figure 8.12b.

**Algorithm 8.1:** Newton's method for minimizing a function

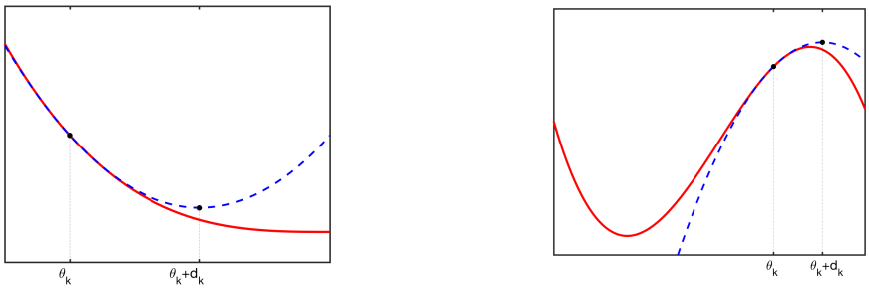
```
Initialize θ_0
for t = 1, 2, ... until convergence do
    Evaluate g_t = ∇L(θ_t)
    Evaluate H_t = ∇²L(θ_t)
    Solve H_t d_t = −g_t for d_t
    Use line search to find stepsize η_t along d_t
    θ_{t+1} = θ_t + η_t d_t
```

This algorithm can be derived as follows. Consider making a second-order Taylor series approximation of $\mathcal{L}(\boldsymbol{\theta})$ around $\boldsymbol{\theta}_t$:

$$\mathcal{L}_{\text{quad}}(\boldsymbol{\theta}) = \mathcal{L}(\boldsymbol{\theta}_t) + \boldsymbol{g}_t^{\mathsf{T}}(\boldsymbol{\theta} - \boldsymbol{\theta}_t) + \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}_t)^{\mathsf{T}} \mathbf{H}_t (\boldsymbol{\theta} - \boldsymbol{\theta}_t) \tag{8.41}$$

The minimum of $\mathcal{L}_{\text{quad}}$ is at

$$\boldsymbol{\theta} = \boldsymbol{\theta}_t - \mathbf{H}_t^{-1} \boldsymbol{g}_t \tag{8.42}$$




(a) (b)

*Figure 8.14: Illustration of Newton's method for minimizing a 1d function. (a) The solid curve is the function $\mathcal{L}(x)$. The dotted line $\mathcal{L}_{\text{quad}}(\theta)$ is its second order approximation at $\theta_t$. The Newton step $d_t$ is what must be added to $\theta_t$ to get to the minimum of $\mathcal{L}_{\text{quad}}(\theta)$. Adapted from Figure 13.4 of [Van06]. Generated by newtonsMethodMinQuad.ipynb. (b) Illustration of Newton's method applied to a nonconvex function. We fit a quadratic function around the current point $\theta_t$ and move to its stationary point, $\theta_{t+1} = \theta_t + d_t$. Unfortunately, this takes us near a local maximum of $f$, not minimum. This means we need to be careful about the extent of our quadratic approximation. Adapted from Figure 13.11 of [Van06]. Generated by newtonsMethodNonConvex.ipynb.*

So if the quadratic approximation is a good one, we should pick $\boldsymbol{d}_t = -\mathbf{H}_t^{-1}\boldsymbol{g}_t$ as our descent direction. See Figure 8.14(a) for an illustration. Note that, in a "pure" Newton method, we use $\eta_t = 1$ as our stepsize. However, we can also use linesearch to find the best stepsize; this tends to be more robust as using $\eta_t = 1$ may not always converge globally.

If we apply this method to linear regression, we get to the optimum in one step, since (as we show in Section 11.2.2.1) we have $\mathbf{H} = \mathbf{X}^\mathsf{T}\mathbf{X}$ and $\boldsymbol{g} = \mathbf{X}^\mathsf{T}\mathbf{X}\boldsymbol{w} - \mathbf{X}^\mathsf{T}\boldsymbol{y}$, so the Newton update becomes

$$\boldsymbol{w}_1 = \boldsymbol{w}_0 - \mathbf{H}^{-1}\boldsymbol{g} = \boldsymbol{w}_0 - (\mathbf{X}^\mathsf{T}\mathbf{X})^{-1}(\mathbf{X}^\mathsf{T}\mathbf{X}\boldsymbol{w}_0 - \mathbf{X}^\mathsf{T}\boldsymbol{y}) = \boldsymbol{w}_0 - \boldsymbol{w}_0 + (\mathbf{X}^\mathsf{T}\mathbf{X})^{-1}\mathbf{X}^\mathsf{T}\boldsymbol{y} \tag{8.43}$$

which is the OLS estimate. However, when we apply this method to logistic regression, it may take multiple iterations to converge to the global optimum, as we discuss in Section 10.2.6.

### 8.3.2 BFGS and other quasi-Newton methods

**Quasi-Newton** methods, sometimes called **variable metric** methods, iteratively build up an approximation to the Hessian using information gleaned from the gradient vector at each step. The most common method is called **BFGS** (named after its simultaneous inventors, Broyden, Fletcher, Goldfarb and Shanno), which updates the approximation to the Hessian $\mathbf{B}_t \approx \mathbf{H}_t$ as follows:

$$\mathbf{B}_{t+1} = \mathbf{B}_t + \frac{\boldsymbol{y}_t\boldsymbol{y}_t^\mathsf{T}}{\boldsymbol{y}_t^\mathsf{T}\boldsymbol{s}_t} - \frac{(\mathbf{B}_t\boldsymbol{s}_t)(\mathbf{B}_t\boldsymbol{s}_t)^\mathsf{T}}{\boldsymbol{s}_t^\mathsf{T}\mathbf{B}_t\boldsymbol{s}_t} \tag{8.44}$$

$$\boldsymbol{s}_t = \boldsymbol{\theta}_t - \boldsymbol{\theta}_{t-1} \tag{8.45}$$

$$\boldsymbol{y}_t = \boldsymbol{g}_t - \boldsymbol{g}_{t-1} \tag{8.46}$$

This is a rank-two update to the matrix. If $\mathbf{B}_0$ is positive-definite, and the step size $\eta$ is chosen via line search satisfying both the Armijo condition in Equation (8.27) and the following curvature

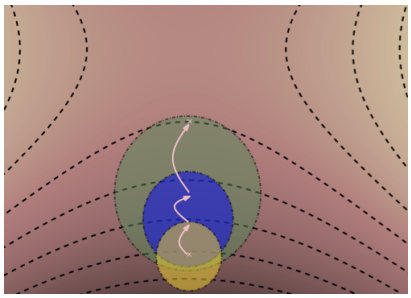


*Figure 8.15: Illustration of the trust region approach. The dashed lines represents contours of the original nonconvex objective. The circles represent successive quadratic approximations. From Figure 4.2 of [Pas14]. Used with kind permission of Razvan Pascanu.*

condition

$$\nabla \mathcal{L}(\boldsymbol{\theta}_t + \eta \boldsymbol{d}_t) \geq c_2 \eta \boldsymbol{d}_t^\mathsf{T} \nabla \mathcal{L}(\boldsymbol{\theta}_t) \tag{8.47}$$

then $\mathbf{B}_{t+1}$ will remain positive definite. The constant $c_2$ is chosen within $(c, 1)$ where $c$ is the tunable parameter in Equation (8.27). The two step size conditions are together known as the **Wolfe conditions**. We typically start with a diagonal approximation, $\mathbf{B}_0 = \mathbf{I}$. Thus BFGS can be thought of as a "diagonal plus low-rank" approximation to the Hessian.

Alternatively, BFGS can iteratively update an approximation to the inverse Hessian, $\mathbf{C}_t \approx \mathbf{H}_t^{-1}$, as follows:

$$\mathbf{C}_{t+1} = \left(\mathbf{I} - \frac{\boldsymbol{s}_t \boldsymbol{y}_t^\mathsf{T}}{\boldsymbol{y}_t^\mathsf{T} \boldsymbol{s}_t}\right) \mathbf{C}_t \left(\mathbf{I} - \frac{\boldsymbol{y}_t \boldsymbol{s}_t^\mathsf{T}}{\boldsymbol{y}_t^\mathsf{T} \boldsymbol{s}_t}\right) + \frac{\boldsymbol{s}_t \boldsymbol{s}_t^\mathsf{T}}{\boldsymbol{y}_t^\mathsf{T} \boldsymbol{s}_t} \tag{8.48}$$

Since storing the Hessian approximation still takes $O(D^2)$ space, for very large problems, one can use **limited memory BFGS**, or **L-BFGS**, where we control the rank of the approximation by only using the $M$ most recent $(\boldsymbol{s}_t, \boldsymbol{y}_t)$ pairs while ignoring older information. Rather than storing $\mathbf{B}_t$ explicitly, we just store these vectors in memory, and then approximate $\mathbf{H}_t^{-1}\boldsymbol{g}_t$ by performing a sequence of inner products with the stored $\boldsymbol{s}_t$ and $\boldsymbol{y}_t$ vectors. The storage requirements are therefore $O(MD)$. Typically choosing $M$ to be between 5–20 suffices for good performance [NW06, p177].

Note that sklearn uses LBFGS as its default solver for logistic regression.[1]

### 8.3.3 Trust region methods

If the objective function is nonconvex, then the Hessian $\mathbf{H}_t$ may not be positive definite, so $\boldsymbol{d}_t = -\mathbf{H}_t^{-1}\boldsymbol{g}_t$ may not be a descent direction. This is illustrated in 1d in Figure 8.14(b), which shows that Newton's method can end up in a local maximum rather than a local minimum.

In general, any time the quadratic approximation made by Newton's method becomes invalid, we are in trouble. However, there is usually a local region around the current iterate where we can safely

1. See https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html.

approximate the objective by a quadratic. Let us call this region $\mathcal{R}_t$, and let us call $M(\boldsymbol{\delta})$ the model (or approximation) to the objective, where $\boldsymbol{\delta} = \boldsymbol{\theta} - \boldsymbol{\theta}_t$. Then at each step we can solve

$$\boldsymbol{\delta}^* = \underset{\boldsymbol{\delta} \in \mathcal{R}_t}{\operatorname{argmin}} \, M_t(\boldsymbol{\delta}) \tag{8.49}$$

This is called **trust-region optimization**. (This can be seen as the "opposite" of line search, in the sense that we pick a distance we want to travel, determined by $\mathcal{R}_t$, and then solve for the optimal direction, rather than picking the direction and then solving for the optimal distance.)

We usually assume that $M_t(\boldsymbol{\delta})$ is a quadratic approximation:

$$M_t(\boldsymbol{\delta}) = \mathcal{L}(\boldsymbol{\theta}_t) + \boldsymbol{g}_t^\mathsf{T} \boldsymbol{\delta} + \frac{1}{2} \boldsymbol{\delta}^\mathsf{T} \mathbf{H}_t \boldsymbol{\delta} \tag{8.50}$$

where $\boldsymbol{g}_t = \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta})|_{\boldsymbol{\theta}_t}$ is the gradient, and $\mathbf{H}_t = \nabla^2_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta})|_{\boldsymbol{\theta}_t}$ is the Hessian. Furthermore, it is common to assume that $\mathcal{R}_t$ is a ball of radius $r$, i.e., $\mathcal{R}_t = \{\boldsymbol{\delta} : ||\boldsymbol{\delta}||_2 \leq r\}$. Using this, we can convert the constrained problem into an unconstrained one as follows:

$$\boldsymbol{\delta}^* = \underset{\boldsymbol{\delta}}{\operatorname{argmin}} \, M(\boldsymbol{\delta}) + \lambda ||\boldsymbol{\delta}||_2^2 = \underset{\boldsymbol{\delta}}{\operatorname{argmin}} \, \boldsymbol{g}^\mathsf{T} \boldsymbol{\delta} + \frac{1}{2} \boldsymbol{\delta}^\mathsf{T} (\mathbf{H} + \lambda \mathbf{I}) \boldsymbol{\delta} \tag{8.51}$$

for some Lagrange multiplier $\lambda > 0$ which depends on the radius $r$ (see Section 8.5.1 for a discussion of Lagrange multipliers). We can solve this using

$$\boldsymbol{\delta} = -(\mathbf{H} + \lambda \mathbf{I})^{-1} \boldsymbol{g} \tag{8.52}$$

This is called **Tikhonov damping** or **Tikhonov regularization**. See Figure 8.15 for an illustration.

Note that adding a sufficiently large $\lambda \mathbf{I}$ to $\mathbf{H}$ ensures the resulting matrix is always positive definite. As $\lambda \to 0$, this trust method reduces to Newton's method, but for $\lambda$ large enough, it will make all the negative eigenvalues positive (and all the 0 eigenvalues become equal to $\lambda$).

## 8.4 Stochastic gradient descent

In this section, we consider **stochastic optimization**, where the goal is to minimize the average value of a function:

$$\mathcal{L}(\boldsymbol{\theta}) = \mathbb{E}_{q(\boldsymbol{z})} \left[ \mathcal{L}(\boldsymbol{\theta}, \boldsymbol{z}) \right] \tag{8.53}$$

where $\boldsymbol{z}$ is a random input to the objective. This could be a "noise" term, coming from the environment, or it could be a training example drawn randomly from the training set, as we explain below.

At each iteration, we assume we observe $\mathcal{L}_t(\boldsymbol{\theta}) = \mathcal{L}(\boldsymbol{\theta}, \boldsymbol{z}_t)$, where $\boldsymbol{z}_t \sim q$. We also assume a way to compute an unbiased estimate of the gradient of $\mathcal{L}$. If the distribution $q(\boldsymbol{z})$ is independent of the parameters we are optimizing, we can use $\boldsymbol{g}_t = \nabla_{\boldsymbol{\theta}} \mathcal{L}_t(\boldsymbol{\theta}_t)$. In this case, the resulting algorithm can be written as follows:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta_t \nabla \mathcal{L}(\boldsymbol{\theta}_t, \boldsymbol{z}_t) = \boldsymbol{\theta}_t - \eta_t \boldsymbol{g}_t \tag{8.54}$$

This method is known as **stochastic gradient descent** or **SGD**. As long as the gradient estimate is unbiased, then this method will converge to a stationary point, providing we decay the step size $\eta_t$ at a certain rate, as we discuss in Section 8.4.3.

### 8.4.1 Application to finite sum problems

SGD is very widely used in machine learning. To see why, recall from Section 4.3 that many model fitting procedures are based on empirical risk minimization, which involve minimizing the following loss:

$$\mathcal{L}(\boldsymbol{\theta}_t) = \frac{1}{N}\sum_{n=1}^{N} \ell(\boldsymbol{y}_n, f(\boldsymbol{x}_n; \boldsymbol{\theta}_t)) = \frac{1}{N}\sum_{n=1}^{N} \mathcal{L}_n(\boldsymbol{\theta}_t) \tag{8.55}$$

This is called a **finite sum problem**. The gradient of this objective has the form

$$\boldsymbol{g}_t = \frac{1}{N}\sum_{n=1}^{N} \nabla_{\boldsymbol{\theta}} \mathcal{L}_n(\boldsymbol{\theta}_t) = \frac{1}{N}\sum_{n=1}^{N} \nabla_{\boldsymbol{\theta}} \ell(\boldsymbol{y}_n, f(\boldsymbol{x}_n; \boldsymbol{\theta}_t)) \tag{8.56}$$

This requires summing over all $N$ training examples, and thus can be slow if $N$ is large. Fortunately we can approximate this by sampling a **minibatch** of $B \ll N$ samples to get

$$\boldsymbol{g}_t \approx \frac{1}{|\mathcal{B}_t|}\sum_{n\in\mathcal{B}_t} \nabla_{\boldsymbol{\theta}} \mathcal{L}_n(\boldsymbol{\theta}_t) = \frac{1}{|\mathcal{B}_t|}\sum_{n\in\mathcal{B}_t} \nabla_{\boldsymbol{\theta}} \ell(\boldsymbol{y}_n, f(\boldsymbol{x}_n; \boldsymbol{\theta}_t)) \tag{8.57}$$

where $\mathcal{B}_t$ is a set of randomly chosen examples to use at iteration $t$.[2] This is an unbiased approximation to the empirical average in Equation (8.56). Hence we can safely use this with SGD.

Although the theoretical rate of convergence of SGD is slower than batch GD (in particular, SGD has a sublinear convergence rate), in practice SGD is often faster, since the per-step time is much lower [BB08; BB11]. To see why SGD can make faster progress than full batch GD, suppose we have a dataset consisting of a single example duplicated $K$ times. Batch training will be (at least) $K$ times slower than SGD, since it will waste time computing the gradient for the repeated examples. Even if there are no duplicates, batch training can be wasteful, since early on in training the parameters are not well estimated, so it is not worth carefully evaluating the gradient.

### 8.4.2 Example: SGD for fitting linear regression

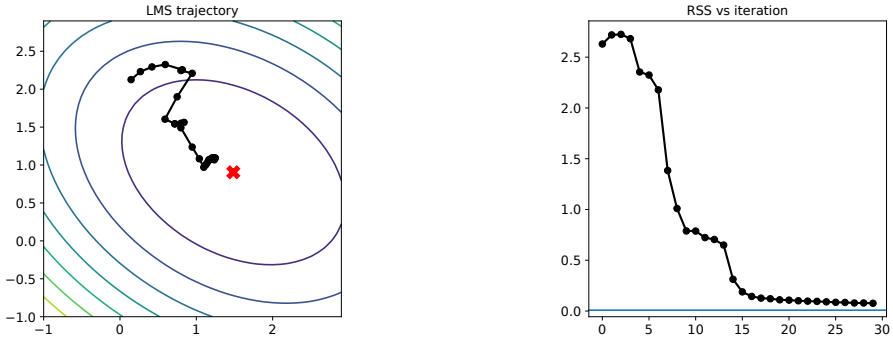
In this section, we show how to use SGD to fit a linear regression model. Recall from Section 4.2.7 that the objective has the form

$$\mathcal{L}(\boldsymbol{\theta}) = \frac{1}{2N}\sum_{n=1}^{N} (\boldsymbol{x}_n^\mathsf{T}\boldsymbol{\theta} - y_n)^2 = \frac{1}{2N}||\mathbf{X}\boldsymbol{\theta} - \boldsymbol{y}||_2^2 \tag{8.58}$$

The gradient is

$$\boldsymbol{g}_t = \frac{1}{N}\sum_{n=1}^{N} (\boldsymbol{\theta}_t^\mathsf{T}\boldsymbol{x}_n - y_n)\boldsymbol{x}_n \tag{8.59}$$

2. In practice we usually sample $\mathcal{B}_t$ without replacement. However, once we reach the end of the dataset (i.e., after a single training **epoch**), we can perform a random shuffling of the examples, to ensure that each minibatch on the next epoch is different from the last. This version of SGD is analyzed in [HS19].




*(a)* *(b)*

*Figure 8.16: Illustration of the LMS algorithm. Left: we start from $\boldsymbol{\theta} = (-0.5, 2)$ and slowly converging to the least squares solution of $\hat{\boldsymbol{\theta}} = (1.45, 0.93)$ (red cross). Right: plot of objective function over time. Note that it does not decrease monotonically. Generated by lms_demo.ipynb.*

Now consider using SGD with a minibatch size of $B = 1$. The update becomes

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta_t(\boldsymbol{\theta}_t^\mathsf{T}\boldsymbol{x}_n - y_n)\boldsymbol{x}_n \tag{8.60}$$

where $n = n(t)$ is the index of the example chosen at iteration $t$. The overall algorithm is called the **least mean squares** (**LMS**) algorithm, and is also known as the **delta rule**, or the **Widrow-Hoff rule**.

Figure 8.16 shows the results of applying this algorithm to the data shown in Figure 11.2. We start at $\boldsymbol{\theta} = (-0.5, 2)$ and converge (in the sense that $||\boldsymbol{\theta}_t - \boldsymbol{\theta}_{t-1}||_2^2$ drops below a threshold of $10^{-2}$) in about 26 iterations. Note that SGD (and hence LMS) may require multiple passes through the data to find the optimum.

### 8.4.3 Choosing the step size (learning rate)

When using SGD, we need to be careful in how we choose the learning rate in order to achieve convergence. For example, in Figure 8.17 we plot the loss vs the learning rate when we apply SGD to a deep neural network classifier (see Chapter 13 for details). We see a U-shaped curve, where an overly small learning rate results in underfitting, and overly large learning rate results in instability of the model (c.f., Figure 8.11(b)); in both cases, we fail to converge to a local optimum.

One heuristic for choosing a good learning rate, proposed in [Smi18], is to start with a small learning rate and gradually increase it, evaluating performance using a small number of minibatches. We then make a plot like the one in Figure 8.17, and pick the learning rate with the lowest loss. (In practice, it is better to pick a rate that is slightly smaller than (i.e., to the left of) the one with the lowest loss, to ensure stability.)

Rather than choosing a single constant learning rate, we can use a **learning rate schedule**, in which we adjust the step size over time. Theoretically, a sufficient condition for SGD to achieve

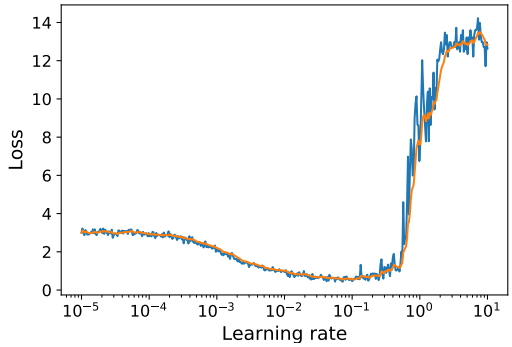
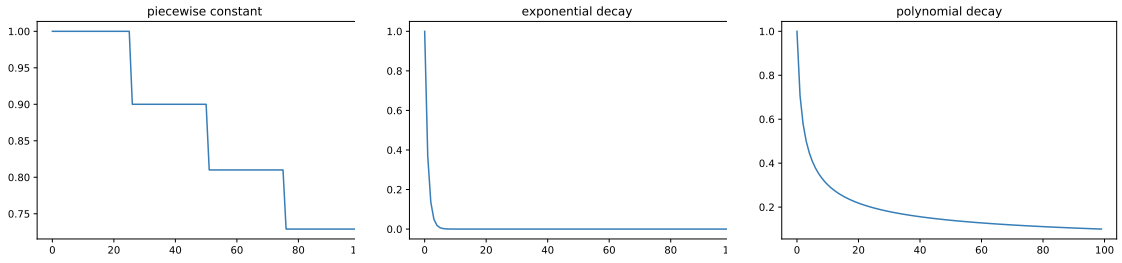



*Figure 8.17: Loss vs learning rate (horizontal axis). Training loss vs learning rate for a small MLP fit to FashionMNIST using vanilla SGD. (Raw loss in blue, EWMA smoothed version in orange). Generated by lrschedule_tf.ipynb.*




*(a)* *(b)* *(c)*

*Figure 8.18: Illustration of some common learning rate schedules. (a) Piecewise constant. (b) Exponential decay. (c) Polynomial decay. Generated by learning_rate_plot.ipynb.*

convergence is if the learning rate schedule satisfies the **Robbins-Monro conditions**:

$$\eta_t \to 0, \ \frac{\sum_{t=1}^{\infty} \eta_t^2}{\sum_{t=1}^{\infty} \eta_t} \to 0 \tag{8.61}$$

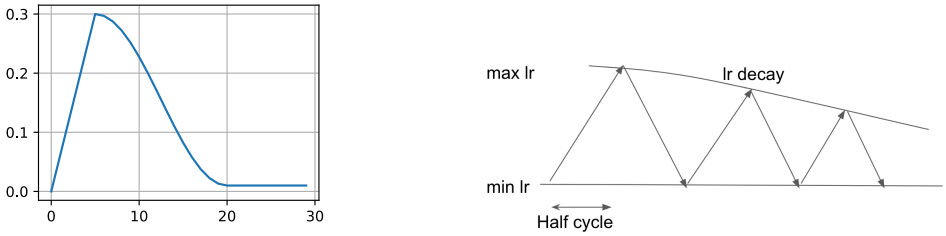
Some common examples of learning rate schedules are listed below:

$$\eta_t = \eta_i \text{ if } t_i \leq t \leq t_{i+1} \quad \text{piecewise constant} \tag{8.62}$$

$$\eta_t = \eta_0 e^{-\lambda t} \text{ exponential decay} \tag{8.63}$$

$$\eta_t = \eta_0 (\beta t + 1)^{-\alpha} \text{ polynomial decay} \tag{8.64}$$

In the piecewise constant schedule, $t_i$ are a set of time points at which we adjust the learning rate to a specified value. For example, we may set $\eta_i = \eta_0 \gamma^i$, which reduces the initial learning rate by a factor of $\gamma$ for each threshold (or milestone) that we pass. Figure 8.18a illustrates this for $\eta_0 = 1$




*(a)* *(b)*

*Figure 8.19: (a) Linear warm-up followed by cosine cool-down. (b) Cyclical learning rate schedule.*

and $\gamma = 0.9$. This is called **step decay**. Sometimes the threshold times are computed adaptively, by estimating when the train or validation loss has plateaued; this is called **reduce-on-plateau**. Exponential decay is typically too fast, as illustrated in Figure 8.18b. A common choice is polynomial decay, with $\alpha = 0.5$ and $\beta = 1$, as illustrated in Figure 8.18c; this corresponds to a **square-root schedule**, $\eta_t = \eta_0 \frac{1}{\sqrt{t+1}}$.

In the deep learning community, another common schedule is to quickly increase the learning rate and then gradually decrease it again, as shown in Figure 8.19a. This is called **learning rate warmup**, or the **one-cycle learning rate schedule** [Smi18]. The motivation for this is the following: initially the parameters may be in a part of the loss landscape that is poorly conditioned, so a large step size will "bounce around" too much (c.f., Figure 8.11(b)) and fail to make progress downhill. However, with a slow learning rate, the algorithm can discover flatter regions of space, where a larger step size can be used. Once there, fast progress can be made. However, to ensure convergence to a point, we must reduce the learning rate to 0. See [Got+19; Gil+21] for more details.

It is also possible to increase and decrease the learning rate multiple times, in a cyclical fashion. This is called a **cyclical learning rate** [Smi18], and was popularized by the `fast.ai` course. See Figure 8.19b for an illustration using triangular shapes. The motivation behind this approach is to escape local minima. The minimum and maximum learning rates can be found based on the initial "dry run" described above, and the half-cycle can be chosen based on how many restarts you want to do with your training budget. A related approach, known as **stochastic gradient descent with warm restarts**, was proposed in [LH17]; they proposed storing all the checkpoints visited after each cool down, and using all of them as members of a model ensemble. (See Section 18.2 for a discussion of ensemble learning.)

An alternative to using heuristics for estimating the learning rate is to use line search (Section 8.2.2.2). This is tricky when using SGD, because the noisy gradients make the computation of the Armijo condition difficult [CS20]. However, [Vas+19] show that it can be made to work if the variance of the gradient noise goes to zero over time. This can happen if the model is sufficiently flexible that it can perfectly interpolate the training set.

### 8.4.4 Iterate averaging

The parameter estimates produced by SGD can be very unstable over time. To reduce the variance of the estimate, we can compute the average using

$$\overline{\boldsymbol{\theta}}_t = \frac{1}{t}\sum_{i=1}^{t}\boldsymbol{\theta}_i = \frac{1}{t}\boldsymbol{\theta}_t + \frac{t-1}{t}\overline{\boldsymbol{\theta}}_{t-1} \tag{8.65}$$

where $\boldsymbol{\theta}_t$ are the usual SGD iterates. This is called **iterate averaging** or **Polyak-Ruppert averaging** [Rup88].

In [PJ92], they prove that the estimate $\overline{\boldsymbol{\theta}}_t$ achieves the best possible asymptotic convergence rate among SGD algorithms, matching that of variants using second-order information, such as Hessians.

This averaging can also have statistical benefits. For example, in [NR18], they prove that, in the case of linear regression, this method is equivalent to $\ell_2$ regularization (i.e., ridge regression).

Rather than an exponential moving average of SGD iterates, **Stochastic Weight Averaging** (SWA) [Izm+18] uses an *equal* average in conjunction with a modified learning rate schedule. In contrast to standard Polyak-Ruppert averaging, which was motivated for faster convergence rates, SWA exploits the flatness in objectives used to train deep neural networks, to find solutions which provide better generalization.

### 8.4.5 Variance reduction *

In this section, we discuss various ways to reduce the variance in SGD. In some cases, this can improve the theoretical convergence rate from sublinear to linear (i.e., the same as full-batch gradient descent) [SLRB17; JZ13; DBLJ14]. These methods reduce the variance of the gradients, rather than the parameters themselves and are designed to work for finite sum problems.

#### 8.4.5.1 SVRG

The basic idea of **stochastic variance reduced gradient** (**SVRG**) [JZ13] is to use a control variate, in which we estimate a baseline value of the gradient based on the full batch, which we then use to compare the stochastic gradients to.

More precisely, ever so often (e.g., once per epoch), we compute the full gradient at a "snapshot" of the model parameters $\tilde{\boldsymbol{\theta}}$; the corresponding "exact" gradient is therefore $\nabla\mathcal{L}(\tilde{\boldsymbol{\theta}})$. At step $t$, we compute the usual stochastic gradient at the current parameters, $\nabla\mathcal{L}_t(\boldsymbol{\theta}_t)$, but also at the snapshot parameters, $\nabla\mathcal{L}_t(\tilde{\boldsymbol{\theta}})$, which we use as a baseline. We can then use the following improved gradient estimate

$$\boldsymbol{g}_t = \nabla\mathcal{L}_t(\boldsymbol{\theta}_t) - \nabla\mathcal{L}_t(\tilde{\boldsymbol{\theta}}) + \nabla\mathcal{L}(\tilde{\boldsymbol{\theta}}) \tag{8.66}$$

to compute $\boldsymbol{\theta}_{t+1}$. This is unbiased because $\mathbb{E}\left[\nabla\mathcal{L}_t(\tilde{\boldsymbol{\theta}})\right] = \nabla\mathcal{L}(\tilde{\boldsymbol{\theta}})$. Furthermore, the update only involves two gradient computations, since we can compute $\nabla\mathcal{L}(\tilde{\boldsymbol{\theta}})$ once per epoch. At the end of the epoch, we update the snapshot parameters, $\tilde{\boldsymbol{\theta}}$, based on the most recent value of $\boldsymbol{\theta}_t$, or a running average of the iterates, and update the expected baseline. (We can compute snapshots less often, but then the baseline will not be correlated with the objective and can hurt performance, as shown in [DB18].)

Iterations of SVRG are computationally faster than those of full-batch GD, but SVRG can still match the theoretical convergence rate of GD.

#### 8.4.5.2 SAGA

In this section, we describe the **stochastic averaged gradient accelerated** (**SAGA**) algorithm of [DBLJ14]. Unlike SVRG, it only requires one full batch gradient computation, at the start of the algorithm. However, it "pays" for this saving in time by using more memory. In particular, it must store $N$ gradient vectors. This enables the method to maintain an approximation of the global gradient by removing the old local gradient from the overall sum and replacing it with the new local gradient. This is called an **aggregated gradient** method.

More precisely, we first initialize by computing $\boldsymbol{g}_n^{\text{local}} = \nabla \mathcal{L}_n(\boldsymbol{\theta}_0)$ for all $n$, and the average, $\boldsymbol{g}^{\text{avg}} = \frac{1}{N}\sum_{n=1}^{N} \boldsymbol{g}_n^{\text{local}}$. Then, at iteration $t$, we use the gradient estimate

$$\boldsymbol{g}_t = \nabla \mathcal{L}_n(\boldsymbol{\theta}_t) - \boldsymbol{g}_n^{\text{local}} + \boldsymbol{g}^{\text{avg}} \tag{8.67}$$

where $n \sim \text{Unif}\{1, \ldots, N\}$ is the example index sampled at iteration $t$. We then update $\boldsymbol{g}_n^{\text{local}} = \nabla \mathcal{L}_n(\boldsymbol{\theta}_t)$ and $\boldsymbol{g}^{\text{avg}}$ by replacing the old $\boldsymbol{g}_n^{\text{local}}$ by its new value.

This has an advantage over SVRG since it only has to do one full batch sweep at the start. (In fact, the initial sweep is not necessary, since we can compute $\boldsymbol{g}^{\text{avg}}$ "lazily", by only incorporating gradients we have seen so far.) The downside is the large extra memory cost. However, if the features (and hence gradients) are sparse, the memory cost can be reasonable. Indeed, the SAGA algorithm is recommended for use in the sklearn logistic regression code when $N$ is large and $\boldsymbol{x}$ is sparse.[3]

#### 8.4.5.3 Application to deep learning

Variance reduction methods are widely used for fitting ML models with convex objectives, such as linear models. However, there are various difficulties associated with using SVRG with conventional deep learning training practices. For example, the use of batch normalization (Section 14.2.4.1), data augmentation (Section 19.1) and dropout (Section 13.5.4) all break the assumptions of the method, since the loss will differ randomly in ways that depend not just on the parameters and the data index $n$. For more details, see e.g., [DB18; Arn+19].

### 8.4.6 Preconditioned SGD

In this section, we consider **preconditioned SGD**, which involves the following update:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta_t \mathbf{M}_t^{-1} \boldsymbol{g}_t, \tag{8.68}$$

where $\mathbf{M}_t$ is a **preconditioning matrix**, or simply the **preconditioner**, typically chosen to be positive-definite. Unfortunately the noise in the gradient estimates make it difficult to reliably estimate the Hessian, which makes it difficult to use the methods from Section 8.3. In addition, it is expensive to solve for the update direction with a full preconditioning matrix. Therefore most practitioners use a diagonal preconditioner $\mathbf{M}_t$. Such preconditioners do not necessarily use second-order information, but often result in speedups compared to vanilla SGD. See also [Roo+21]

3. See https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression.

for a probabilitic interpretation of these heuristics, and sgd_comparison.ipynb for an empirical comparison on some simple datasets.

#### 8.4.6.1 AdaGrad

The **AdaGrad** (short for "adaptive gradient") method of [DHS11] was originally designed for optimizing convex objectives where many elements of the gradient vector are zero; these might correspond to features that are rarely present in the input, such as rare words. The update has the following form

$$\theta_{t+1,d} = \theta_{t,d} - \eta_t \frac{1}{\sqrt{s_{t,d} + \epsilon}} g_{t,d} \tag{8.69}$$

where $d = 1 : D$ indexes the dimensions of the parameter vector, and

$$s_{t,d} = \sum_{i=1}^{t} g_{i,d}^2 \tag{8.70}$$

is the sum of the squared gradients and $\epsilon > 0$ is a small term to avoid dividing by zero. Equivalently we can write the update in vector form as follows:

$$\Delta \boldsymbol{\theta}_t = -\eta_t \frac{1}{\sqrt{\boldsymbol{s}_t + \epsilon}} \boldsymbol{g}_t \tag{8.71}$$

where the square root and division is performed elementwise. Viewed as preconditioned SGD, this is equivalent to taking $\mathbf{M}_t = \text{diag}(\boldsymbol{s}_t + \boldsymbol{\epsilon})^{1/2}$. This is an example of an **adaptive learning rate**; the overall stepsize $\eta_t$ still needs to be chosen, but the results are less sensitive to it compared to vanilla GD. In particular, we usually fix $\eta_t = \eta_0$.

#### 8.4.6.2 RMSProp and AdaDelta

A defining feature of AdaGrad is that the term in the denominator gets larger over time, so the effective learning rate drops. While it is necessary to ensure convergence, it might hurt performance as the denominator gets large too fast.

An alternative is to use an exponentially weighted moving average (EWMA, Section 4.4.2.2) of the past squared gradients, rather than their sum:

$$s_{t+1,d} = \beta s_{t,d} + (1 - \beta) g_{t,d}^2 \tag{8.72}$$

In practice we usually use $\beta \sim 0.9$, which puts more weight on recent examples. In this case,

$$\sqrt{s_{t,d}} \approx \text{RMS}(\boldsymbol{g}_{1:t,d}) = \sqrt{\frac{1}{t} \sum_{\tau=1}^{t} g_{\tau,d}^2} \tag{8.73}$$

where RMS stands for "root mean squared". Hence this method, (which is based on the earlier **RPROP** method of [RB93]) is known as **RMSProp** [Hin14]. The overall update of RMSProp is

$$\Delta \boldsymbol{\theta}_t = -\eta_t \frac{1}{\sqrt{\boldsymbol{s}_t + \epsilon}} \boldsymbol{g}_t. \tag{8.74}$$

The **AdaDelta** method was independently introduced in [Zei12], and is similar to RMSprop. However, in addition to accumulating an EWMA of the gradients in $\hat{\boldsymbol{s}}$, it also keeps an EWMA of the updates $\boldsymbol{\delta}_t$ to obtain an update of the form

$$\Delta\boldsymbol{\theta}_t = -\eta_t \frac{\sqrt{\boldsymbol{\delta}_{t-1} + \epsilon}}{\sqrt{\boldsymbol{s}_t + \epsilon}} \boldsymbol{g}_t \tag{8.75}$$

where

$$\boldsymbol{\delta}_t = \beta \boldsymbol{\delta}_{t-1} + (1-\beta)(\Delta\boldsymbol{\theta}_t)^2 \tag{8.76}$$

and $\boldsymbol{s}_t$ is the same as in RMSProp. This has the advantage that the "units" of the numerator and denominator cancel, so we are just elementwise-multiplying the gradient by a scalar. This eliminates the need to tune the learning rate $\eta_t$, which means one can simply set $\eta_t = 1$, although popular implementations of AdaDelta still keep $\eta_t$ as a tunable hyperparameter. However, since these adaptive learning rates need not decrease with time (unless we choose $\eta_t$ to explicitly do so), these methods are not guaranteed to converge to a solution.

#### 8.4.6.3 Adam

It is possible to combine RMSProp with momentum. In particular, let us compute an EWMA of the gradients (as in momentum) and squared gradients (as in RMSProp)

$$\boldsymbol{m}_t = \beta_1 \boldsymbol{m}_{t-1} + (1-\beta_1)\boldsymbol{g}_t \tag{8.77}$$

$$\boldsymbol{s}_t = \beta_2 \boldsymbol{s}_{t-1} + (1-\beta_2)\boldsymbol{g}_t^2 \tag{8.78}$$

We then perform the following update:

$$\Delta\boldsymbol{\theta}_t = -\eta_t \frac{1}{\sqrt{\boldsymbol{s}_t} + \epsilon} \boldsymbol{m}_t \tag{8.79}$$

The resulting method is known as **Adam**, which stands for "adaptive moment estimation" [KB15].

The standard values for the various constants are $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-6}$. (If we set $\beta_1 = 0$ and no bias-correction, we recover RMSProp, which does not use momentum.) For the overall learning rate, it is common to use a fixed value such as $\eta_t = 0.001$. Again, as the adaptive learning rate may not decrease over time, convergence is not guaranteed (see Section 8.4.6.4).

If we initialize with $\boldsymbol{m}_0 = \boldsymbol{s}_0 = \mathbf{0}$, then initial estimates will be biased towards small values. The authors therefore recommend using the bias-corrected moments, which increase the values early in the optimization process. These estimates are given by

$$\hat{\boldsymbol{m}}_t = \boldsymbol{m}_t / (1-\beta_1^t) \tag{8.80}$$

$$\hat{\boldsymbol{s}}_t = \boldsymbol{s}_t / (1-\beta_2^t) \tag{8.81}$$

The advantage of bias-correction is shown in Figure 4.3.

#### 8.4.6.4 Issues with adaptive learning rates

When using diagonal scaling methods, the overall learning rate is determined by $\eta_0 \mathbf{M}_t^{-1}$, which changes with time. Hence these methods are often called **adaptive learning rate** methods. However, they still require setting the base learning rate $\eta_0$.

Since the EWMA methods are typically used in the stochastic setting where the gradient estimates are noisy, their learning rate adaptation can result in non-convergence even on convex problems [RKK18]. Various solutions to this problem have been proposed, including AMSGRAD [RKK18], PADAM [CG18; Zho+18], and YOGI [Zah+18]. For example, the YOGI update modifies ADAM by replacing

$$\boldsymbol{s}_t = \beta_2 \boldsymbol{s}_{t-1} + (1-\beta_2)\boldsymbol{g}_t^2 = \boldsymbol{s}_{t-1} + (1-\beta_2)(\boldsymbol{g}_t^2 - \boldsymbol{s}_{t-1}) \tag{8.82}$$

with

$$\boldsymbol{s}_t = \boldsymbol{s}_{t-1} + (1-\beta_2)\boldsymbol{g}_t^2 \odot \operatorname{sgn}(\boldsymbol{g}_t^2 - \boldsymbol{s}_{t-1}) \tag{8.83}$$

However, more recent work [Zha+22] has shown that vanilla Adam can be made to always converge provided the $\beta_1$ and $\beta_2$ parameters are tuned on a per-dataset basis. (In practice, it is common to fix $\beta_1 = 0.9$ and just tune $\beta_2$.)

#### 8.4.6.5 Non-diagonal preconditioning matrices

Although the methods we have discussed above can adapt the learning rate of each parameter, they do not solve the more fundamental problem of ill-conditioning due to correlation of the parameters, and hence do not always provide as much of a speed boost over vanilla SGD as one may hope.

One way to get faster convergence is to use the following preconditioning matrix, known as **full-matrix Adagrad** [DHS11]:

$$\mathbf{M}_t = [(\mathbf{G}_t \mathbf{G}_t^\mathsf{T})^{\frac{1}{2}} + \epsilon \mathbf{I}_D]^{-1} \tag{8.84}$$

where

$$\mathbf{G}_t = [\boldsymbol{g}_t, \ldots, \boldsymbol{g}_1] \tag{8.85}$$

Here $\boldsymbol{g}_i = \nabla_{\boldsymbol{\psi}} c(\boldsymbol{\psi}_i)$ is the $D$-dimensional gradient vector computed at step $i$. Unfortunately, $\mathbf{M}_t$ is a $D \times D$ matrix, which is expensive to store and invert.

The **Shampoo** algorithm [GKS18] makes a block diagonal approximation to $\mathbf{M}$, one per layer of the model, and then exploits Kronecker product structure to efficiently invert it. (It is called "shampoo" because it uses a conditioner.) Recently, [Ani+20] scaled this method up to fit very large deep models in record time.

## 8.5 Constrained optimization

In this section, we consider the following **constrained optimization problem**:

$$\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta} \in \mathcal{C}} \mathcal{L}(\boldsymbol{\theta}) \tag{8.86}$$

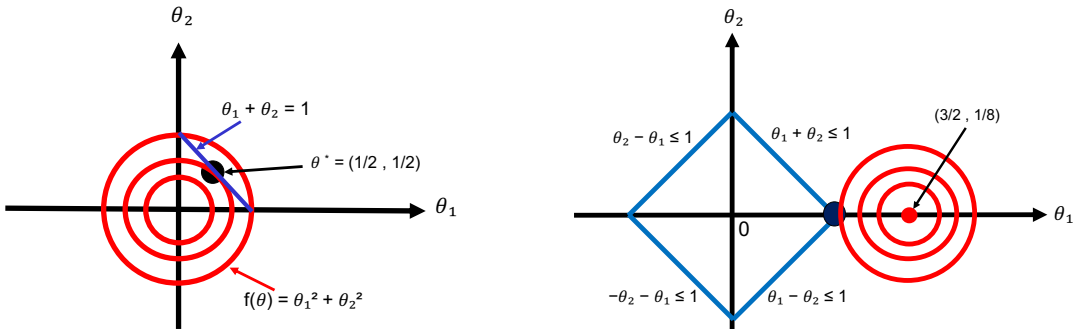



*Figure 8.20: Illustration of some constrained optimization problems. Red contours are the level sets of the objective function $\mathcal{L}(\boldsymbol{\theta})$. Optimal constrained solution is the black dot, (a) Blue line is the equality constraint $h(\boldsymbol{\theta}) = 0$. (b) Blue lines denote the inequality constraints $|\theta_1| + |\theta_2| \leq 1$. (Compare to Figure 11.8 (left).)*

where the feasible set, or constraint set, is

$$\mathcal{C} = \{\boldsymbol{\theta} \in \mathbb{R}^D : h_i(\boldsymbol{\theta}) = 0, i \in \mathcal{E},\ g_j(\boldsymbol{\theta}) \leq 0, j \in \mathcal{I}\} \tag{8.87}$$

where $\mathcal{E}$ is the set of **equality constraints**, and $\mathcal{I}$ is the set of **inequality constraints**.

For example, suppose we have a quadratic objective, $\mathcal{L}(\boldsymbol{\theta}) = \theta_1^2 + \theta_2^2$, subject to a linear equality constraint, $h(\boldsymbol{\theta}) = 1 - \theta_1 - \theta_2 = 0$. Figure 8.20(a) plots the level sets of $\mathcal{L}$, as well as the constraint surface. What we are trying to do is find the point $\boldsymbol{\theta}^*$ that lives on the line, but which is closest to the origin. It is clear from the geometry that the optimal solution is $\boldsymbol{\theta} = (0.5, 0.5)$, indicated by the solid black dot.

In the following sections, we briefly describe some of the theory and algorithms underlying constrained optimization. More details can be found in other books, such as [BV04; NW06; Ber15; Ber16].

### 8.5.1 Lagrange multipliers

In this section, we discuss how to solve equality contrained optimization problems. We initially assume that we have just one equality constraint, $h(\boldsymbol{\theta}) = 0$.

First note that for any point on the constraint surface, $\nabla h(\boldsymbol{\theta})$ will be orthogonal to the constraint surface. To see why, consider another point nearby, $\boldsymbol{\theta} + \boldsymbol{\epsilon}$, that also lies on the surface. If we make a first-order Taylor expansion around $\boldsymbol{\theta}$ we have

$$h(\boldsymbol{\theta} + \boldsymbol{\epsilon}) \approx h(\boldsymbol{\theta}) + \boldsymbol{\epsilon}^\mathsf{T} \nabla h(\boldsymbol{\theta}) \tag{8.88}$$

Since both $\boldsymbol{\theta}$ and $\boldsymbol{\theta} + \boldsymbol{\epsilon}$ are on the constraint surface, we must have $h(\boldsymbol{\theta}) = h(\boldsymbol{\theta} + \boldsymbol{\epsilon})$ and hence $\boldsymbol{\epsilon}^\mathsf{T} \nabla h(\boldsymbol{\theta}) \approx 0$. Since $\boldsymbol{\epsilon}$ is parallel to the constraint surface, $\nabla h(\boldsymbol{\theta})$ must be perpendicular to it.

We seek a point $\boldsymbol{\theta}^*$ on the constraint surface such that $\mathcal{L}(\boldsymbol{\theta})$ is minimized. We just showed that it must satisfy the condition that $\nabla h(\boldsymbol{\theta}^*)$ is orthogonal to the constraint surface. In addition, such a point must have the property that $\nabla \mathcal{L}(\boldsymbol{\theta})$ is also orthogonal to the constraint surface, as otherwise we could decrease $\mathcal{L}(\boldsymbol{\theta})$ by moving a short distance along the constraint surface. Since both $\nabla h(\boldsymbol{\theta})$

and $\nabla \mathcal{L}(\boldsymbol{\theta})$ are orthogonal to the constraint surface at $\boldsymbol{\theta}^*$, they must be parallel (or anti-parallel) to each other. Hence there must exist a constant $\lambda^* \in \mathbb{R}$ such that

$$\nabla \mathcal{L}(\boldsymbol{\theta}^*) = \lambda^* \nabla h(\boldsymbol{\theta}^*) \tag{8.89}$$

(We cannot just equate the gradient vectors, since they may have different magnitudes.) The constant $\lambda^*$ is called a **Lagrange multiplier**, and can be positive, negative, or zero. This latter case occurs when $\nabla \mathcal{L}(\boldsymbol{\theta}^*) = 0$.

We can convert Equation (8.89) into an objective, known as the **Lagrangian**, that we should find a stationary point of the following:

$$L(\boldsymbol{\theta}, \lambda) \triangleq \mathcal{L}(\boldsymbol{\theta}) + \lambda h(\boldsymbol{\theta}) \tag{8.90}$$

At a stationary point of the Lagrangian, we have

$$\nabla_{\boldsymbol{\theta},\lambda} L(\boldsymbol{\theta}, \lambda) = \mathbf{0} \iff \lambda \nabla_{\boldsymbol{\theta}} h(\boldsymbol{\theta}) = \nabla \mathcal{L}(\boldsymbol{\theta}),\ h(\boldsymbol{\theta}) = 0 \tag{8.91}$$

This is called a **critical point**, and satisfies the original constraint $h(\boldsymbol{\theta}) = 0$ and Equation (8.89).

If we have $m > 1$ constraints, we can form a new constraint function by addition, as follows:

$$L(\boldsymbol{\theta}, \boldsymbol{\lambda}) = \mathcal{L}(\boldsymbol{\theta}) + \sum_{j=1}^{m} \lambda_j h_j(\boldsymbol{\theta}) \tag{8.92}$$

We now have $D+m$ equations in $D+m$ unknowns and we can use standard unconstrained optimization methods to find a stationary point. We give some examples below.

#### 8.5.1.1 Example: 2d Quadratic objective with one linear equality constraint

Consider minimizing $\mathcal{L}(\boldsymbol{\theta}) = \theta_1^2 + \theta_2^2$ subject to the constraint that $\theta_1 + \theta_2 = 1$.

(This is the problem illustrated in Figure 8.20(a).) The Lagrangian is

$$L(\theta_1, \theta_2, \lambda) = \theta_1^2 + \theta_2^2 + \lambda(\theta_1 + \theta_2 - 1) \tag{8.93}$$

We have the following conditions for a stationary point:

$$\frac{\partial}{\partial \theta_1} L(\theta_1, \theta_2, \lambda) = 2\theta_1 + \lambda = 0 \tag{8.94}$$

$$\frac{\partial}{\partial \theta_2} L(\theta_1, \theta_2, \lambda) = 2\theta_2 + \lambda = 0 \tag{8.95}$$

$$\frac{\partial}{\partial \lambda} L(\theta_1, \theta_2, \lambda) = \theta_1 + \theta_2 - 1 = 0 \tag{8.96}$$

From Equations 8.94 and 8.95 we find $2\theta_1 = -\lambda = 2\theta_2$, so $\theta_1 = \theta_2$. Also, from Equation (8.96), we find $2\theta_1 = 1$. So $\boldsymbol{\theta}^* = (0.5, 0.5)$, as we claimed earlier. Furthermore, this is the global minimum since the objective is convex and the constraint is affine.

### 8.5.2 The KKT conditions

In this section, we generalize the concept of Lagrange multipliers to additionally handle inequality constraints.

First consider the case where we have a single inequality constraint $g(\boldsymbol{\theta}) \leq 0$. To find the optimum, one approach would be to consider an unconstrained problem where we add the penalty as an infinite step function:

$$\hat{\mathcal{L}}(\boldsymbol{\theta}) = \mathcal{L}(\boldsymbol{\theta}) + \infty \mathbb{I}\left(g(\boldsymbol{\theta}) > 0\right) \tag{8.97}$$

However, this is a discontinuous function that is hard to optimize.

Instead, we create a lower bound of the form $\mu g(\boldsymbol{\theta})$, where $\mu \geq 0$. This gives us the following Lagrangian:

$$L(\boldsymbol{\theta}, \mu) = \mathcal{L}(\boldsymbol{\theta}) + \mu g(\boldsymbol{\theta}) \tag{8.98}$$

Note that the step function can be recovered using

$$\hat{\mathcal{L}}(\boldsymbol{\theta}) = \max_{\mu \geq 0} L(\boldsymbol{\theta}, \mu) = \begin{cases} \infty & \text{if } g(\boldsymbol{\theta}) > 0, \\ \mathcal{L}(\boldsymbol{\theta}) & \text{otherwise} \end{cases} \tag{8.99}$$

Thus our optimization problem becomes

$$\min_{\boldsymbol{\theta}} \max_{\mu \geq 0} L(\boldsymbol{\theta}, \mu) \tag{8.100}$$

Now consider the general case where we have multiple inequality constraints, $\boldsymbol{g}(\boldsymbol{\theta}) \leq \mathbf{0}$, and multiple equality constraints, $\boldsymbol{h}(\boldsymbol{\theta}) = \mathbf{0}$. The **generalized Lagrangian** becomes

$$L(\boldsymbol{\theta}, \boldsymbol{\mu}, \boldsymbol{\lambda}) = \mathcal{L}(\boldsymbol{\theta}) + \sum_i \mu_i g_i(\boldsymbol{\theta}) + \sum_j \lambda_j h_j(\boldsymbol{\theta}) \tag{8.101}$$

(We are free to change $-\lambda_j h_j$ to $+\lambda_j h_j$ since the sign is arbitrary.) Our optimization problem becomes

$$\min_{\boldsymbol{\theta}} \max_{\boldsymbol{\mu} \geq \mathbf{0}, \boldsymbol{\lambda}} L(\boldsymbol{\theta}, \boldsymbol{\mu}, \boldsymbol{\lambda}) \tag{8.102}$$

When $\mathcal{L}$ and $g$ are convex, then all critical points of this problem must satisfy the following criteria (under some conditions [BV04, Sec.5.2.3]):

- All constraints are satisfied (this is called **feasibility**):

$$\boldsymbol{g}(\boldsymbol{\theta}) \leq \mathbf{0}, \ \boldsymbol{h}(\boldsymbol{\theta}) = \mathbf{0} \tag{8.103}$$

- The solution is a stationary point:

$$\nabla \mathcal{L}(\boldsymbol{\theta}^*) + \sum_i \mu_i \nabla g_i(\boldsymbol{\theta}^*) + \sum_j \lambda_j \nabla h_j(\boldsymbol{\theta}^*) = \mathbf{0} \tag{8.104}$$

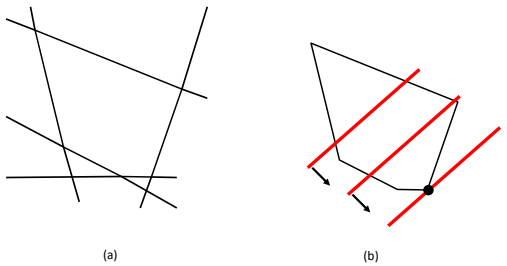



*Figure 8.21: (a) A convex polytope in 2d defined by the intersection of linear constraints. (b) Depiction of the feasible set as well as the linear objective function. The red line is a level set of the objective, and the arrow indicates the direction in which it is improving. We see that the optimal solution lies at a vertex of the polytope.*

- The penalty for the inequality constraint points in the right direction (this is called **dual feasibility**):

$$\boldsymbol{\mu} \geq \mathbf{0} \tag{8.105}$$

- The Lagrange multipliers pick up any slack in the inactive constraints, i.e., either $\mu_i = 0$ or $g_i(\boldsymbol{\theta}^*) = 0$, so

$$\boldsymbol{\mu} \odot \boldsymbol{g} = \mathbf{0} \tag{8.106}$$

  This is called **complementary slackness**.

To see why the last condition holds, consider (for simplicity) the case of a single inequality constraint, $g(\boldsymbol{\theta}) \leq 0$. Either it is **active**, meaning $g(\boldsymbol{\theta}) = 0$, or it is inactive, meaning $g(\boldsymbol{\theta}) < 0$. In the active case, the solution lies on the constraint boundary, and $g(\boldsymbol{\theta}) = 0$ becomes an equality constraint; then we have $\nabla \mathcal{L} = \mu \nabla g$ for some constant $\mu \neq 0$, because of Equation (8.89). In the inactive case, the solution is not on the constraint boundary; we still have $\nabla \mathcal{L} = \mu \nabla g$, but now $\mu = 0$.

These are called the **Karush-Kuhn-Tucker** (**KKT**) conditions. If $\mathcal{L}$ is a convex function, and the constraints define a convex set, the KKT conditions are sufficient for (global) optimality, as well as necessary.

### 8.5.3 Linear programming

Consider optimizing a linear function subject to linear constraints. When written in **standard form**, this can be represented as

$$\min_{\boldsymbol{\theta}} \boldsymbol{c}^\mathsf{T} \boldsymbol{\theta} \quad \text{s.t.} \quad \mathbf{A}\boldsymbol{\theta} \leq \boldsymbol{b}, \ \boldsymbol{\theta} \geq 0 \tag{8.107}$$

The feasible set defines a convex **polytope**, which is a convex set defined as the intersection of half spaces. See Figure 8.21(a) for a 2d example. Figure 8.21(b) shows a linear cost function that

decreases as we move to the bottom right. We see that the lowest point that is in the feasible set is a vertex. In fact, it can be proved that the optimum point always occurs at a vertex of the polytope, assuming the solution is unique. If there are multiple solutions, the line will be parallel to a face. There may also be no optima inside the feasible set; in this case, the problem is said to be infeasible.

#### 8.5.3.1 The simplex algorithm

It can be shown that the optima of an LP occur at vertices of the polytope defining the feasible set (see Figure 8.21(b) for an example). The **simplex algorithm** solves LPs by moving from vertex to vertex, each time seeking the edge which most improves the objective.

In the worst-case scenario, the simplex algorithm can take time exponential in $D$, although in practice it is usually very efficient. There are also various polynomial-time algorithms, such as the interior point method, although these are often slower in practice.

#### 8.5.3.2 Applications

There are many applications of linear programming in science, engineering and business. It is also useful in some machine learning problems. For example, Section 11.6.1.1 shows how to use it to solve robust linear regression. It is also useful for state estimation in graphical models (see e.g., [SGJ11]).

### 8.5.4 Quadratic programming

Consider minimizing a quadratic objective subject to linear equality and inequality constraints. This kind of problem is known as a **quadratic program** or **QP**, and can be written as follows:

$$\min_{\boldsymbol{\theta}} \frac{1}{2}\boldsymbol{\theta}^\mathsf{T}\mathbf{H}\boldsymbol{\theta} + \boldsymbol{c}^\mathsf{T}\boldsymbol{\theta} \quad \text{s.t.} \quad \mathbf{A}\boldsymbol{\theta} \leq \boldsymbol{b},\ \mathbf{C}\boldsymbol{\theta} = \boldsymbol{d} \tag{8.108}$$

If $\mathbf{H}$ is positive semidefinite, then this is a convex optimization problem.

#### 8.5.4.1 Example: 2d quadratic objective with linear inequality constraints

As a concrete example, suppose we want to minimize

$$\mathcal{L}(\boldsymbol{\theta}) = (\theta_1 - \frac{3}{2})^2 + (\theta_2 - \frac{1}{8})^2 = \frac{1}{2}\boldsymbol{\theta}^\mathsf{T}\mathbf{H}\boldsymbol{\theta} + \boldsymbol{c}^\mathsf{T}\boldsymbol{\theta} + \text{const} \tag{8.109}$$

where $\mathbf{H} = 2\mathbf{I}$ and $\boldsymbol{c} = -(3, 1/4)$, subject to

$$|\theta_1| + |\theta_2| \leq 1 \tag{8.110}$$

See Figure 8.20(b) for an illustration.

We can rewrite the constraints as

$$\theta_1 + \theta_2 \leq 1,\ \ \theta_1 - \theta_2 \leq 1,\ \ -\theta_1 + \theta_2 \leq 1,\ \ -\theta_1 - \theta_2 \leq 1 \tag{8.111}$$

which we can write more compactly as

$$\mathbf{A}\boldsymbol{\theta} \leq \boldsymbol{b} \tag{8.112}$$

where $\boldsymbol{b} = \mathbf{1}$ and

$$\mathbf{A} = \begin{pmatrix} 1 & 1 \\ 1 & -1 \\ -1 & 1 \\ -1 & -1 \end{pmatrix} \tag{8.113}$$

This is now in the standard QP form.

From the geometry of the problem, shown in Figure 8.20(b), we see that the constraints corresponding to the two left faces of the diamond) are inactive (since we are trying to get as close to the center of the circle as possible, which is outside of, and to the right of, the constrained feasible region). Denoting $g_i(\boldsymbol{\theta})$ as the inequality constraint corresponding to row $i$ of $\mathbf{A}$, this means $g_3(\boldsymbol{\theta}^*) > 0$ and $g_4(\boldsymbol{\theta}^*) > 0$, and hence, by complementarity, $\mu_3^* = \mu_4^* = 0$. We can therefore remove these inactive constraints.

From the KKT conditions we know that

$$\mathbf{H}\boldsymbol{\theta} + \boldsymbol{c} + \mathbf{A}^\mathsf{T}\boldsymbol{\mu} = \mathbf{0} \tag{8.114}$$

Using these for the actively constrained subproblem, we get

$$\begin{pmatrix} 2 & 0 & 1 & 1 \\ 0 & 2 & 1 & -1 \\ 1 & 1 & 0 & 0 \\ 1 & -1 & 0 & 0 \end{pmatrix} \begin{pmatrix} \theta_1 \\ \theta_2 \\ \mu_1 \\ \mu_2 \end{pmatrix} = \begin{pmatrix} 3 \\ 1/4 \\ 1 \\ 1 \end{pmatrix} \tag{8.115}$$

Hence the solution is

$$\boldsymbol{\theta}_* = (1, 0)^\mathsf{T}, \boldsymbol{\mu}_* = (0.625, 0.375, 0, 0)^\mathsf{T} \tag{8.116}$$

Notice that the optimal value of $\boldsymbol{\theta}$ occurs at one of the vertices of the $\ell_1$ "ball" (the diamond shape).

#### 8.5.4.2 Applications

There are several applications of quadratic programming in ML. For example, in Section 11.4, we discuss the lasso method for sparse linear regression, which amounts to optimizing $\mathcal{L}(\boldsymbol{w}) = ||\mathbf{X}\boldsymbol{w} - \boldsymbol{y}||_2^2 + \lambda||\boldsymbol{w}||_1$, which can be reformulated into a QP. And in Section 17.3, we show how to use QP for SVMs (support vector machines).

### 8.5.5 Mixed integer linear programming *

**Integer linear programming** or **ILP** corresponds to minimizing a linear objective, subject to linear constraints, where the optimization variables are discrete integers instead of reals. In standard form, the problem is as follows:

$$\min_{\boldsymbol{\theta}} \boldsymbol{c}^\mathsf{T}\boldsymbol{\theta} \qquad \text{s.t.} \quad \mathbf{A}\boldsymbol{\theta} \leq \boldsymbol{b},\ \boldsymbol{\theta} \geq 0, \boldsymbol{\theta} \in \mathbb{Z}^D \tag{8.117}$$

where $\mathbb{Z}$ is the set of integers. If some of the optimization variables are real-valued, it is called a **mixed ILP**, often called a **MIP** for short. (If all of the variables are real-valued, it becomes a standard LP.)

MIPs have a large number of applications, such as in vehicle routing, scheduling and packing. They are also useful for some ML applications, such as formally verifying the behavior of certain kinds of deep neural networks [And+18], and proving robustness properties of DNNs to adversarial (worst-case) perturbations [TXT19].

## 8.6 Proximal gradient method *

We are often interested in optimizing an objective of the form

$$\mathcal{L}(\boldsymbol{\theta}) = \mathcal{L}_s(\boldsymbol{\theta}) + \mathcal{L}_r(\boldsymbol{\theta}) \tag{8.118}$$

where $\mathcal{L}_s$ is differentiable (smooth), and $\mathcal{L}_r$ is convex but not necessarily differentiable (i.e., it may be non-smooth or "rough"). For example, $\mathcal{L}_s$ might be the negative log likelihood (NLL), and $\mathcal{L}_r$ might be an indicator function that is infinite if a constraint is violated (see Section 8.6.1), or $\mathcal{L}_r$ might be the $\ell_1$ norm of some parameters (see Section 8.6.2), or $\mathcal{L}_r$ might measure how far the parameters are from a set of allowed quantized values (see Section 8.6.3).

One way to tackle such problems is to use the **proximal gradient method** (see e.g., [PB+14; PSW15]). Roughly speaking, this takes a step of size $\eta$ in the direction of the gradient, and then projects the resulting parameter update into a space that respects $\mathcal{L}_r$. More precisely, the update is as follows

$$\boldsymbol{\theta}_{t+1} = \text{prox}_{\eta_t \mathcal{L}_r}(\boldsymbol{\theta}_t - \eta_t \nabla \mathcal{L}_s(\boldsymbol{\theta}_t)) \tag{8.119}$$

where $\text{prox}_{\eta \mathcal{L}_r}(\boldsymbol{\theta})$ is the **proximal operator** of $\mathcal{L}_r$ (scaled by $\eta$) evaluated at $\boldsymbol{\theta}$:

$$\text{prox}_{\eta \mathcal{L}_r}(\boldsymbol{\theta}) \triangleq \underset{\boldsymbol{z}}{\text{argmin}} \left( \mathcal{L}_r(\boldsymbol{z}) + \frac{1}{2\eta} ||\boldsymbol{z} - \boldsymbol{\theta}||_2^2 \right) \tag{8.120}$$

(The factor of $\frac{1}{2}$ is an arbitrary convention.) We can rewrite the proximal operator as solving a constrained optimization problem, as follows:

$$\text{prox}_{\eta \mathcal{L}_r}(\boldsymbol{\theta}) = \underset{\boldsymbol{z}}{\text{argmin}}\, \mathcal{L}_r(\boldsymbol{z}) \;\; \text{s.t.} \;\; ||\boldsymbol{z} - \boldsymbol{\theta}||_2 \leq \rho \tag{8.121}$$

where the bound $\rho$ depends on the scaling factor $\eta$. Thus we see that the proximal projection minimizes the function while staying close to (i.e., proximal to) the current iterate. We give some examples below.

### 8.6.1 Projected gradient descent

Suppose we want to solve the problem

$$\underset{\boldsymbol{\theta}}{\text{argmin}}\, \mathcal{L}_s(\boldsymbol{\theta}) \;\; \text{s.t.} \;\; \boldsymbol{\theta} \in \mathcal{C} \tag{8.122}$$

where $\mathcal{C}$ is a convex set. For example, we may have the **box constraints** $\mathcal{C} = \{\boldsymbol{\theta} : \boldsymbol{l} \leq \boldsymbol{\theta} \leq \boldsymbol{u}\}$, where we specify lower and upper bounds on each element. These bounds can be infinite for certain

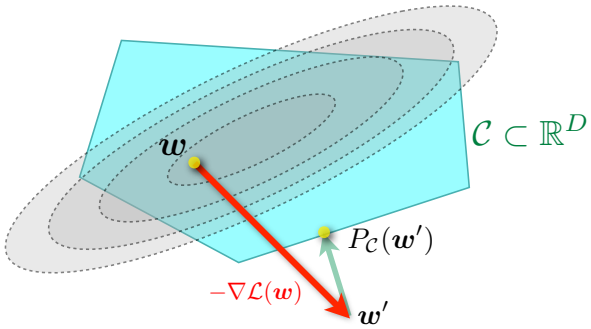



*Figure 8.22: Illustration of projected gradient descent. $\boldsymbol{w}$ is the current parameter estimate, $\boldsymbol{w}'$ is the update after a gradient step, and $P_{\mathcal{C}}(\boldsymbol{w}')$ projects this onto the constraint set $\mathcal{C}$. From https://bit.ly/3eJ3BhZ Used with kind permission of Martin Jaggi.*

elements if we don't want to constrain values along that dimension. For example, if we just want to ensure the parameters are non-negative, we set $l_d = 0$ and $u_d = \infty$ for each dimension $d$.

We can convert the constrained optimization problem into an unconstrained one by adding a **penalty term** to the original objective:

$$\mathcal{L}(\boldsymbol{\theta}) = \mathcal{L}_s(\boldsymbol{\theta}) + \mathcal{L}_r(\boldsymbol{\theta}) \tag{8.123}$$

where $\mathcal{L}_r(\boldsymbol{\theta})$ is the indicator function for the convex set $\mathcal{C}$, i.e.,

$$\mathcal{L}_r(\boldsymbol{\theta}) = I_{\mathcal{C}}(\boldsymbol{\theta}) = \begin{cases} 0 & \text{if } \boldsymbol{\theta} \in \mathcal{C} \\ \infty & \text{if } \boldsymbol{\theta} \notin \mathcal{C} \end{cases} \tag{8.124}$$

We can use proximal gradient descent to solve Equation (8.123). The proximal operator for the indicator function is equivalent to projection onto the set $\mathcal{C}$:

$$\text{proj}_{\mathcal{C}}(\boldsymbol{\theta}) = \underset{\boldsymbol{\theta}' \in \mathcal{C}}{\text{argmin}} \, ||\boldsymbol{\theta}' - \boldsymbol{\theta}||_2 \tag{8.125}$$

This method is known as **projected gradient descent**. See Figure 8.22 for an illustration.

For example, consider the box constraints $\mathcal{C} = \{\boldsymbol{\theta} : \boldsymbol{l} \leq \boldsymbol{\theta} \leq \boldsymbol{u}\}$. The projection operator in this case can be computed elementwise by simply thresholding at the boundaries:

$$\text{proj}_{\mathcal{C}}(\boldsymbol{\theta})_d = \begin{cases} l_d & \text{if } \theta_d \leq l_d \\ \theta_d & \text{if } l_d \leq \theta_d \leq u_d \\ u_d & \text{if } \theta_d \geq u_d \end{cases} \tag{8.126}$$

For example, if we want to ensure all elements are non-negative, we can use

$$\text{proj}_{\mathcal{C}}(\boldsymbol{\theta}) = \boldsymbol{\theta}_+ = [\max(\theta_1, 0), \ldots, \max(\theta_D, 0)] \tag{8.127}$$

See Section 11.4.9.2 for an application of this method to sparse linear regression.

### 8.6.2 Proximal operator for $\ell_1$-norm regularizer

Consider a linear predictor of the form $f(\boldsymbol{x};\boldsymbol{\theta}) = \sum_{d=1}^{D} \theta_d x_d$. If we have $\theta_d = 0$ for any dimension $d$, we ignore the corresponding feature $x_d$. This is a form of **feature selection**, which can be useful both as a way to reduce overfitting as well as way to improve model interpretability. We can encourage weights to be zero (and not just small) by penalizing the $\ell_1$ norm,

$$||\boldsymbol{\theta}||_1 = \sum_{d=1}^{D} |\theta_d| \tag{8.128}$$

This is called a **sparsity inducing regularizer**.

To see why this induces sparsity, consider two possible parameter vectors, one which is sparse, $\boldsymbol{\theta} = (1, 0)$, and one which is non-sparse, $\boldsymbol{\theta}' = (1/\sqrt{2}, 1/\sqrt{2})$. Both have the same $\ell_2$ norm

$$||(1,0)||_2^2 = ||(1/\sqrt{2}, 1/\sqrt{2})||_2^2 = 1 \tag{8.129}$$

Hence $\ell_2$ regularization (Section 4.5.3) will not favor the sparse solution over the dense solution. However, when using $\ell_1$ regularization, the sparse solution is cheaper, since

$$||(1,0)||_1 = 1 < ||(1/\sqrt{2}, 1/\sqrt{2})||_1 = \sqrt{2} \tag{8.130}$$

See Section 11.4 for more details on sparse regression.

If we combine this regularizer with our smooth loss, we get

$$\mathcal{L}(\boldsymbol{\theta}) = \text{NLL}(\boldsymbol{\theta}) + \lambda ||\boldsymbol{\theta}||_1 \tag{8.131}$$

We can optimize this objective using proximal gradient descent. The key question is how to compute the prox operator for the function $f(\boldsymbol{\theta}) = ||\boldsymbol{\theta}||_1$. Since this function decomposes over dimensions $d$, the proximal projection can be computed componentwise. From Equation (8.120), with $\eta = 1$, we have

$$\text{prox}_{\lambda f}(\theta) = \underset{z}{\text{argmin}}\, |z| + \frac{1}{2\lambda}(z-\theta)^2 = \underset{z}{\text{argmin}}\, \lambda|z| + \frac{1}{2}(z-\theta)^2 \tag{8.132}$$

In Section 11.4.3, we show that the solution to this is given by

$$\text{prox}_{\lambda f}(\theta) = \begin{cases} \theta - \lambda & \text{if } \theta \geq \lambda \\ 0 & \text{if } |\theta| \leq \lambda \\ \theta + \lambda & \text{if } \theta \leq -\lambda \end{cases} \tag{8.133}$$

This is known as the **soft thresholding operator**, since values less than $\lambda$ in absolute value are set to 0 (thresholded), but in a continuous way. Note that soft thresholding can be written more compactly as

$$\text{SoftThreshold}(\theta, \lambda) = \text{sign}(\theta)\,(|\theta| - \lambda)_+ \tag{8.134}$$

where $\theta_+ = \max(\theta, 0)$ is the positive part of $\theta$. In the vector case, we perform this elementwise:

$$\text{SoftThreshold}(\boldsymbol{\theta}, \lambda) = \text{sign}(\boldsymbol{\theta}) \odot (|\boldsymbol{\theta}| - \lambda)_+ \tag{8.135}$$

See Section 11.4.9.3 for an application of this method to sparse linear regression.

### 8.6.3 Proximal operator for quantization

In some applications (e.g., when training deep neural networks to run on memory-limited **edge devices**, such as mobile phones) we want to ensure that the parameters are **quantized**. For example, in the extreme case where each parameter can only be -1 or +1, the state space becomes $\mathcal{C} = \{-1, +1\}^D$.

Let us define a regularizer that measures distance to the nearest quantized version of the parameter vector:

$$\mathcal{L}_r(\boldsymbol{\theta}) = \inf_{\boldsymbol{\theta}_0 \in \mathcal{C}} ||\boldsymbol{\theta} - \boldsymbol{\theta}_0||_1 \tag{8.136}$$

(We could also use the $\ell_2$ norm.) In the case of $\mathcal{C} = \{-1, +1\}^D$, this becomes

$$\mathcal{L}_r(\boldsymbol{\theta}) = \sum_{d=1}^{D} \inf_{[\theta_0]_d \in \{\pm 1\}} |\theta_d - [\theta_0]_d| = \sum_{d=1}^{D} \min\{|\theta_d - 1|, |\theta_d + 1|\} = ||\boldsymbol{\theta} - \text{sign}(\boldsymbol{\theta})||_1 \tag{8.137}$$

Let us define the corresponding quantization operator to be

$$q(\boldsymbol{\theta}) = \text{proj}_{\mathcal{C}}(\boldsymbol{\theta}) = \text{argmin}\, \mathcal{L}_r(\boldsymbol{\theta}) = \text{sign}(\boldsymbol{\theta}) \tag{8.138}$$

The core difficulty with quantized learning is that quantization is not a differentiable operation. A popular solution to this is to use the **straight-through estimator**, which uses the approximation $\frac{\partial \mathcal{L}}{\partial q(\boldsymbol{\theta})} \approx \frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}}$ (see e.g., [Yin+19]). The corresponding update can be done in two steps: first compute the gradient vector at the quantized version of the current parameters, and then update the unconstrained parameters using this approximate gradient:

$$\tilde{\boldsymbol{\theta}}_t = \text{proj}_{\mathcal{C}}(\boldsymbol{\theta}_t) = q(\boldsymbol{\theta}_t) \tag{8.139}$$

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta_t \nabla \mathcal{L}_s(\tilde{\boldsymbol{\theta}}_t) \tag{8.140}$$

When applied to $\mathcal{C} = \{-1, +1\}^D$, this is known as the **binary connect** method [CBD15].

We can get better results using proximal gradient descent, in which we treat quantization as a regularizer, rather than a hard constraint; this is known as **ProxQuant** [BWL19]. The update becomes

$$\tilde{\boldsymbol{\theta}}_t = \text{prox}_{\lambda \mathcal{L}_r}\left(\boldsymbol{\theta}_t - \eta_t \nabla \mathcal{L}_s(\boldsymbol{\theta}_t)\right) \tag{8.141}$$

In the case that $\mathcal{C} = \{-1, +1\}^D$, one can show that the proximal operator is a generalization of the soft thresholding operator in Equation (8.135):

$$\text{prox}_{\lambda \mathcal{L}_r}(\boldsymbol{\theta}) = \text{SoftThreshold}(\boldsymbol{\theta}, \lambda, \text{sign}(\boldsymbol{\theta})) \tag{8.142}$$

$$= \text{sign}(\boldsymbol{\theta}) + \text{sign}(\boldsymbol{\theta} - \text{sign}(\boldsymbol{\theta})) \odot (|\boldsymbol{\theta} - \text{sign}(\boldsymbol{\theta})| - \lambda)_+ \tag{8.143}$$

This can be generalized to other forms of quantization; see [Yin+19] for details.

### 8.6.4 Incremental (online) proximal methods

Many ML problems have an objective function which is a sum of losses, one per example. Such problems can be solved incrementally; this is a special case of **online learning**. It is possible to extend proximal methods to this setting. For a probabilistic perspective on such methods (in terms of Kalman filtering), see [AEM18; Aky+19].