Dauphine | PSL
UNIVERSITÉ PARIS

Optimization for Machine Learning

Clément W. Royer

Lecture notes - M2 IASD Apprentissage - 2024/2025

- The last version of these notes can be found at:
  https://www.lamsade.dauphine.fr/∼croyer/ensdocs/OAA/PolyOAA.pdf.
- Comments, typos, etc, can be sent to `clement.royer@lamsade.dauphine.fr`.
  Thanks to Florentin Goyens for feedback on an earlier version.
- **Major updates of the document**
  - 2024.12.08: First version of these notes.
- **Learning goals:**
  - Understand the nature and structure of optimization problems arising in machine learning.
  - Select an algorithm tailored to solving a particular instance among those seen in class based on theoretical and practical concerns.
  - Know the underlying motivation behind the design of optimization algorithms for machine learning.

Contents

Chapter 1

Introduction

This course is concerned with optimization problems arising in data-related applications. Such formulations have gained tremendous interest in recent years, due to the increase in computational power that enable significant advances in fields such as image processing. One of the most fundamental tools behind data science is optimization,that combines mathematical formulations and algorithmic procedures. We describe below the motivation behind studying optimization techniques tailored to data-related applications, as well as the characteristics of the associated problems.

1.1 Motivation

The words *machine learning* are widely used as a way to characterize any task that involves manipulating data : nevertheless, their precise meaning can be difficult to formalize, as other keywords such as *data mining*, *data analysis*, *artificial intelligence* or *Big Data* also denote fields that involve data and/or a learning process. In these notes, we focus on the link between data-related tasks and optimization; although we will denote our applications of interest as pertaining to machine learning, we point out that a more general, possibly better suited categorization would be that of **data science**. For the purpose of these lectures, we will indeed consider machine learning through two main goals:

1) Extract patterns from data, possibly in terms of statistical properties;

2) Use this information to infer or make predictions about yet unseen data.

A number of such machine learning tasks involve an optimization component, as shown Figure 1.1. As a result, for the purpose of these notes, we will view machine learning as a field making use of statistics and optimization, with the latter being our area of interest. Nevertheless, we point out that computer science features such as data management and parallel computing have also been instrumental to the success of machine learning, and thus should eventually be integrated with optimization to form efficient algorithms.

1.2 Notations

1.2.1 Generic notations

- Scalars (i.e. reals) are denoted by lowercase letters: $a, b, c, \alpha, \beta, \gamma$.

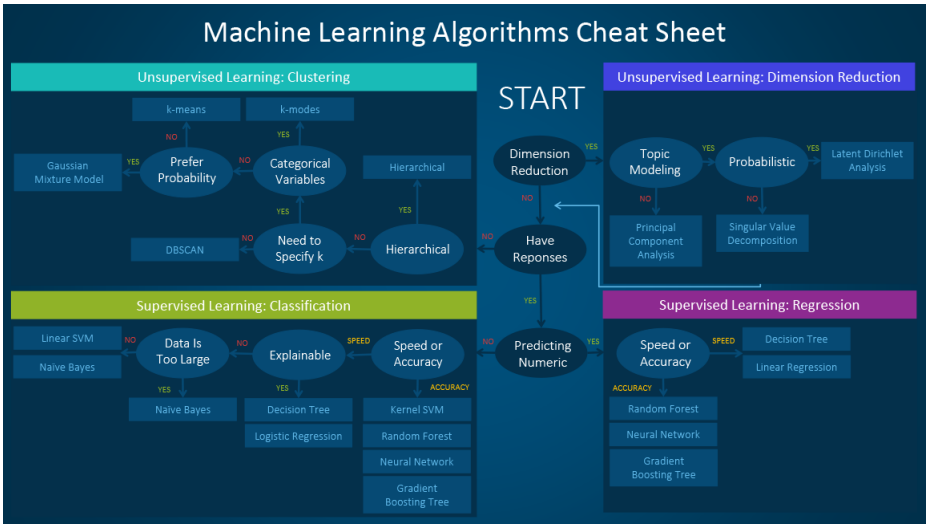

Figure 1.1: A diagram for choosing a machine learning technique appropriate to a given problem; about half of the leaves (Linear SVM, Logistic regression, etc) are directly connected to optimization. *Source: https://blogs.sas.com/content/subconsciousmusings/2017/04/12/machine-learning-algorithm-use/*

- Vectors are denoted by **bold** lowercase letters: $\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}, \boldsymbol{\alpha}, \boldsymbol{\beta}, \boldsymbol{\gamma}$.
- Matrices are denoted by **bold** uppercase letters: $\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}$.
- Sets are denoted by **bold** uppercase cursive letters : $\mathcal{A}, \mathcal{B}, \mathcal{C}$.
- A new operator or quantity is defined using $:=$.
- The following quantifiers are used throughout the notes: $\forall$ (for every), $\exists$ (it exists), $\exists!$ (it exists a unique), $\in$ (belongs to), $\subseteq$ (subset of), $\subset$ (proper subset).
- The Σ operator is used for sums. To lighten the notation, and in the absence of ambiguity, we may omit the first and last indices, or use one sum over multiple indices. As a result, the notations $\sum_{i=1}^{m} \sum_{j=1}^{n}$, $\sum_i \sum_j$ and $\sum_{i,j}$ may be used interchangeably.
- The notation $i = 1, \ldots, m$ indicates that the variable i takes all integer values between 1 and m.

1.2.2 Scalar and vector notations

- The set of natural numbers (nonnegative integers) is denoted by $\mathbb{N}$; the set of integers is denoted by $\mathbb{Z}$.
- The set of real numbers is denoted by $\mathbb{R}$. Our notations for the subset of nonnegative real numbers and the set of positive real numbers are $\mathbb{R}_+$ and $\mathbb{R}_{++}$, respectively. We also define the extended real line $\overline{\mathbb{R}} := \mathbb{R} \cup \{-\infty, \infty\}$.
- The notation $\mathbb{R}^d$ is used for the set of vectors with $d \in \mathbb{N}$ real components; although we do not explicitly indicate it in the rest of these notes, we always assume that $d \geq 1$.

- A vector $\boldsymbol{w} \in \mathbb{R}^d$ is thought as a column vector, with $w_i \in \mathbb{R}$ denoting its i-th coordinate in the canonical basis of $\mathbb{R}^d$. We thus write $\boldsymbol{w} = \begin{bmatrix} w_1 \\ \vdots \\ w_d \end{bmatrix}$, or, in a compact form, $\boldsymbol{w} = [w_i]_{1 \leq 1 \leq d}$.
- Given a column vector $\boldsymbol{w} \in \mathbb{R}^d$, the corresponding row vector is denoted by $\boldsymbol{w}^{\mathrm{T}}$, so that $\boldsymbol{w}^{\mathrm{T}} = [w_1 \ \cdots \ w_d]$ and $[\boldsymbol{w}^{\mathrm{T}}]^{\mathrm{T}} = \boldsymbol{w}$.
- For any integer $d \geq 1$, the vectors $\mathbf{0}_d$ and $\mathbf{1}_d$ correspond to the vectors of $\mathbb{R}^d$ for which all elements are 0 or 1, respectively.

1.2.3 Matrix notations

- We use $\mathbb{R}^{m \times n}$ to denote the set of real rectangular matrices with m rows and n columns, where m et n will always be assumed to be at least 1. If $m = n$, $\mathbb{R}^{n \times n}$ refers to the set of square matrices of size n.
- We identify a matrix in $\mathbb{R}^{m \times 1}$ with its corresponding column vector in $\mathbb{R}^m$.
- Given a matrix $\boldsymbol{A} \in \mathbb{R}^{m \times n}$, $\boldsymbol{A}_{ij}$ refers to the coefficient from the i-th row and the j-th column of $\boldsymbol{A}$: the diagonal of $\boldsymbol{A}$ is given by the coefficients $\boldsymbol{A}_{ii}$. Provided this notation is not ambiguous, we use the notations $\boldsymbol{A}$, $[\boldsymbol{A}_{ij}]_{\substack{1 \leq i \leq m \\ 1 \leq j \leq n}}$ and $[\boldsymbol{A}_{ij}]$ interchangeably.
- Depending on the context, we may use $\boldsymbol{a}_i^{\mathrm{T}}$ to denote the i-th row of $\boldsymbol{A}$ or $\boldsymbol{a}_j$ to denote the j-th column of $\boldsymbol{A}$, leading to $\boldsymbol{A} = \begin{bmatrix} \boldsymbol{a}_1^{\mathrm{T}} \\ \vdots \\ \boldsymbol{a}_m^{\mathrm{T}} \end{bmatrix}$ or $\boldsymbol{A} = [\boldsymbol{a}_1 \ \cdots \ \boldsymbol{a}_n]$, respectively.
- Given $\boldsymbol{A} = [\boldsymbol{A}_{ij}] \in \mathbb{R}^{m \times n}$, the *transpose of matrix* $\boldsymbol{A}$, denoted by $\boldsymbol{A}^{\mathrm{T}}$ (read "$\boldsymbol{A}$ transpose"), is defined as the matrix in $\mathbb{R}^{n \times m}$ (or "n-by-m matrix") such that

  $$\forall i = 1 \ldots m, \ \forall j = 1 \ldots n, \quad \boldsymbol{A}_{ji}^{\mathrm{T}} = \boldsymbol{A}_{ij}.$$

  Note that this generalizes the notation used for row vectors.
- For every $n \geq 1$, $\mathbf{I}_n$ refers to the identity matrix in $\mathbb{R}^{n \times n}$ (with 1s on the diagonal and 0s elsewhere).

1.3 The optimization problem

We now introduce the mathematical foundations behind optimization.

Definition 1.3.1 *Optimization is the field of applied mathematics study concerned with making the best decision out of a set of alternatives.*

Mathematically, we write an optimization problem using three components:

- An **objective function**, i.e. a criterion that measures how good a given decision is, that we want to minimize or maximize depending on the context;

- **Decision variables**, that represent the knobs we can turn to change the decision;
- **Constraints**, i.e. conditions that the decision variables must satisfy in order for the decision to be acceptable.

The general form of the optimization problems considered in these notes will be the following

$$\underset{\boldsymbol{w}\in\mathbb{R}^d}{\text{minimize}} \; f(\boldsymbol{w}) \quad \text{subject to} \quad \boldsymbol{w} \in \mathcal{F}. \tag{1.3.1}$$

In problem (1.3.1), f is the objective function (to be minimized), $\boldsymbol{w}$ is the vector of decision variables and $\mathcal{F}$ is a set encompassing all the constraints on the decision variables. This set is called the feasible set, and is often characterized using mathematical expressions.

1.3.1 Mathematical background

Optimization draws from several fields of mathematics, mostly pertaining to linear algebra, topology and differential calculus. We briefly review the key definitions below.

We will always consider $\mathbb{R}^d$ and $\mathbb{R}^{n\times d}$ as endowed with their canonical vector space structure; in particular, this means that we will be able to add two vectors (or two matrices), and to multiply a vector (or a matrix) by a scalar value. We will also use the following norm.

Definition 1.3.2 (Euclidean norm on $\mathbb{R}^d$) *The Euclidean norm (or ℓ_2 norm) of a vector $\boldsymbol{w} \in \mathbb{R}^d$ is given by:*

$$\|\boldsymbol{w}\| := \sqrt{\sum_{i=1}^{d} w_i^2}.$$

Definition 1.3.3 (Scalar product on $\mathbb{R}^d$) *The scalar product is defined for every $\boldsymbol{w}, \boldsymbol{z} \in \mathbb{R}^d$ by:*

$$\boldsymbol{w}^{\mathrm{T}}\boldsymbol{z} := \sum_{i=1}^{d} w_i \, z_i.$$

One thus has $\boldsymbol{w}^{\mathrm{T}}\boldsymbol{z} = \boldsymbol{z}^{\mathrm{T}}\boldsymbol{w}$ and $\boldsymbol{w}^{\mathrm{T}}\boldsymbol{w} = \|\boldsymbol{w}\|^2$.

The notation T comes from the concept of transpose in matrix linear algebra.

Definition 1.3.4 (Transpose matrix) *Let $\boldsymbol{A} = [\boldsymbol{A}_{ij}] \in \mathbb{R}^{n\times d}$ be a matrix with n rows and d columns.*
The transpose matrix of $\boldsymbol{A}$, *denoted by $\boldsymbol{A}^{\mathrm{T}}$, is the matrix with d rows and n columns such that*

$$\forall i = 1, \dots, n, \; \forall j = 1, \dots, d, \qquad \left[\boldsymbol{A}^{\mathrm{T}}\right]_{ij} = \boldsymbol{A}_{ji}.$$

A square matrix $\boldsymbol{A} \in \mathbb{R}^{d\times d}$ such that $\boldsymbol{A}^{\mathrm{T}} = \boldsymbol{A}$ is called a symmetric matrix.

Definition 1.3.5 (Matrix inversion) *A matrix $\boldsymbol{A} \in \mathbb{R}^{d\times d}$ is* invertible *if it exists $\boldsymbol{B} \in \mathbb{R}^{d\times d}$ such that $\boldsymbol{B}\boldsymbol{A} = \boldsymbol{A}\boldsymbol{B} = \mathbf{I}_d$, where $\mathbf{I}_d$ is the identity matrix of $\mathbb{R}^{d\times d}$.*

In this case, $\boldsymbol{B}$ is the unique matrix with this property: $\boldsymbol{B}$ is called the inverse matrix of $\boldsymbol{A}$, *and is denoted by $\boldsymbol{A}^{-1}$.*

Definition 1.3.6 (Positive (semi-)definiteness) *A matrix $\boldsymbol{A} \in \mathbb{R}^{d\times d}$ is* positive semidefinite *if*

$$\forall \boldsymbol{x} \in \mathbb{R}^n, \quad \boldsymbol{x}^{\mathrm{T}}\boldsymbol{A}\boldsymbol{x} \geq 0.$$

It is called positive definite *when $\boldsymbol{x}^{\mathrm{T}}\boldsymbol{A}\boldsymbol{x} > 0$ for every nonzero vector $\boldsymbol{x}$.*

Definition 1.3.7 (Eigenvalues and eigenvectors) *Let $\boldsymbol{A} \in \mathbb{R}^{d\times d}$. A real λ is called an* eigenvalue of $\boldsymbol{A}$ *if*

$$\exists \boldsymbol{v} \in \mathbb{R}^d, \|\boldsymbol{v}\| \neq 0, \qquad \boldsymbol{A}\boldsymbol{v} = \lambda \boldsymbol{v}.$$

The vector $\boldsymbol{v}$ is then called an eigenvector of $\boldsymbol{A}$ *associated to the eigenvalue λ.*

Theorem 1.3.1 *Any symmetric matrix in $\mathbb{R}^{d\times d}$ possesses d real eigenvalues.*

Notation 1.3.1 *Given two symmetric matrices $(\boldsymbol{A}, \boldsymbol{B}) \in \mathbb{R}^{d\times d}$, we introduce the following notations:*

- $\lambda_{\min}(\boldsymbol{A})/\lambda_{\max}(\boldsymbol{A})$*: smallest/largest eigenvalue of $\boldsymbol{A}$;*
- $\boldsymbol{A} \succeq \boldsymbol{B} \;\Leftrightarrow\; \lambda_{\min}(\boldsymbol{A}) \geq \lambda_{\max}(\boldsymbol{B})$*;*
- $\boldsymbol{A} \succ \boldsymbol{B} \Leftrightarrow \lambda_{\min}(\boldsymbol{A}) > \lambda_{\max}(\boldsymbol{B})$.

Following these notations, a matrix $\boldsymbol{A}$ is called **positive semi-definite** (resp. positive definite) if and only if $\boldsymbol{A} \succeq \boldsymbol{0}$ (resp. $\boldsymbol{A} \succ \boldsymbol{0}$).

Differential calculus We will mostly consider minimization problems involving a smooth objective function: the term "smooth" can be loosely defined in the optimization or learning literature, but generally means that the function is as regular as needed for the desired algorithms and analysis to be applicable. In these notes, we will consider that a smooth function is at least continuously differentiable, sometimes twice continuously differentiable. Those concepts are recalled below.

Definition 1.3.8 (Continuous function) *A function $f : \mathbb{R}^d \to \mathbb{R}^m$ is continuous at $\boldsymbol{w} \in \mathbb{R}^d$ if for every $\epsilon > 0$, it exists $\delta > 0$ such that*

$$\forall \boldsymbol{v} \in \mathbb{R}^d, \; \|\boldsymbol{v} - \boldsymbol{w}\| \leq \delta \implies \|f(\boldsymbol{v}) - f(\boldsymbol{w})\| \leq \epsilon.$$

Definition 1.3.9 (Lipschitz continuous function) *A function $f : \mathbb{R}^d \to \mathbb{R}^m$ is L-Lipschitz continuous over $\mathbb{R}^d$ if*

$$\forall (\boldsymbol{u}, \boldsymbol{v}) \in \left(\mathbb{R}^d\right)^2, \quad \|f(\boldsymbol{u}) - f(\boldsymbol{v})\| \leq L \|\boldsymbol{u} - \boldsymbol{v}\|,$$

where $L > 0$ is called a Lipschitz constant.

Lipschitz continuous functions can be sandwiched between two linear functions, which is particularly useful for optimization purposes. Note that every Lipschitz continuous function is continuous.

Derivatives are ubiquitous in continuous optimization, as they allow to characterize the local behavior of a function. We assume that the reader is familiar with the concept of derivative of a function from $\mathbb{R} \to \mathbb{R}$. A function $f : \mathbb{R}^d \to \mathbb{R}$ is called *differentiable* at $\boldsymbol{w} \in \mathbb{R}^d$ if all its partial derivatives at $\boldsymbol{w}$ exist.

Definition 1.3.10 (Classes of functions)
- *A function $f : \mathbb{R}^d \to \mathbb{R}$ is continuously differentiable if its first-order derivative exists and is continuous. The set of continously differentiable functions is denoted by $\mathcal{C}^1(\mathbb{R}^d)$.*
- *A function $f : \mathbb{R}^d \to \mathbb{R}$ is twice continuously differentiable if $f \in \mathcal{C}^1(\mathbb{R}^d)$, the second-order derivative of f exists and is continuous. The set of twice continously differentiable functions is denoted by $\mathcal{C}^2(\mathbb{R}^d)$.*

Definition 1.3.11 (First-order derivative) *Let $f \in \mathcal{C}^1(\mathbb{R}^d)$ be a continuously differentiable function. For any $\boldsymbol{w} \in \mathbb{R}^d$, the* **gradient of f at $\boldsymbol{w}$** *is given by*

$$\nabla f(\boldsymbol{w}) := \left[\frac{\partial f}{\partial w_i}(\boldsymbol{w})\right]_{1\leq i\leq d} \in \mathbb{R}^d.$$

Definition 1.3.12 (Second-order derivative) *Let $f \in \mathcal{C}^2(\mathbb{R}^d)$ be a twice continuously differentiable function. For any $\boldsymbol{w} \in \mathbb{R}^d$, the* **Hessian of f at $\boldsymbol{w}$** *is given by*

$$\nabla^2 f(\boldsymbol{w}) := \left[\frac{\partial^2 f}{\partial w_i \partial w_j}(\boldsymbol{w})\right]_{1\leq i,j\leq d} \in \mathbb{R}^{d\times d}.$$

The Hessian matrix is symmetric.

Finally, we define an important class of problems involving a Lipschitz continuity assumption.

Definition 1.3.13 (Smooth functions with Lipschitz derivatives)
- *Given $L > 0$, the set $\mathcal{C}_L^{1,1}(\mathbb{R}^d)$ represents the set of all functions $f : \mathbb{R}^d \to \mathbb{R}$ that belong to $\mathcal{C}^1(\mathbb{R}^d)$ such that ∇f is L-Lipschitz continuous.*
- *Given $L > 0$, the set $\mathcal{C}_L^{2,2}(\mathbb{R}^d)$ represents the set of all functions $f : \mathbb{R}^d \to \mathbb{R}$ that belong to $\mathcal{C}^2(\mathbb{R}^d)$ such that $\nabla^2 f$ is L-Lipschitz continuous.*

An important property of such functions is that one can derive upper approximations on their values, as shown by the following theorem.

Theorem 1.3.2 (First-order Taylor expansion) *Let $f \in C_L^{1,1}(\mathbb{R}^d)$ with $L > 0$. For any vectors $\boldsymbol{w}, \boldsymbol{z} \in \mathbb{R}^d$, one has:*

$$f(\boldsymbol{z}) \leq f(\boldsymbol{w}) + \nabla f(\boldsymbol{w})^{\mathrm{T}}(\boldsymbol{z} - \boldsymbol{w}) + \frac{L}{2}\|\boldsymbol{z} - \boldsymbol{w}\|^2. \tag{1.3.2}$$

This expansion is crucial in analyzing the performance of first-order algorithms, as we will do in Chapter 2.

1.3.2 Solution and optimality conditions

In the rest of this section, we will focus on unconstrained optimization formulations of the form

$$\underset{\boldsymbol{w}\in\mathbb{R}^d}{\text{minimize}}\, f(\boldsymbol{w}) \quad \text{subject to} \quad \boldsymbol{w} \in \mathcal{F}, \tag{1.3.3}$$

and characterize properties of solutions of such problems. Since there can be more than one solution, we denote the set of solutions of (1.3.3) by

$$\operatorname*{argmin}_{\boldsymbol{w}\in\mathbb{R}^d} \{f(\boldsymbol{w}) \mid \boldsymbol{w} \in \mathcal{F}\} \subseteq \mathbb{R}^d. \tag{1.3.4}$$

The minimal value of problem (1.3.6) will be denoted by

$$\min_{\boldsymbol{w}\in\mathbb{R}^d} \{f(\boldsymbol{w}) \mid \boldsymbol{w} \in \mathcal{F}\} \in \mathbb{R} \cup \{-\infty, \infty\}. \tag{1.3.5}$$

If the problem is unbounded (i.e. there always exist a better $\boldsymbol{w}$), we set the minimum value to be $-\infty$, whereas if the feasible set $\mathcal{F}$ is empty, we set the minimum to be $+\infty$.

We now provide two definitions of solutions of (1.3.3), or approximations thereof.

Definition 1.3.14 (Local minimum) *Given a function $f : \mathbb{R}^d \to \mathbb{R}$, a point $\boldsymbol{w}^* \in \mathbb{R}^d$ is called a local minimum of the problem (1.3.3) if it possesses the lowest value of f in a neighborhood of feasible points, i.e. if $\boldsymbol{w}^* \in \mathcal{F}$ and there exists $\delta > 0$ such that*

$$\forall \boldsymbol{w} \in \mathcal{B}_\delta(\boldsymbol{w}^*) \cap \mathcal{F}, \qquad f(\boldsymbol{w}^*) \leq f(\boldsymbol{w}).$$

Local minima are local approximations of solutions: a stronger notion, much harder to guarantee in practice, is that of global minima.

Definition 1.3.15 (Global minimum) *Given a function $f : \mathbb{R}^d \to \mathbb{R}$, a point $\boldsymbol{w}^* \in \mathbb{R}^d$ is called a global minimum of f over $\mathcal{F}$ if $\boldsymbol{w}^* \in \mathcal{F}$*

$$\forall \boldsymbol{w} \in \mathcal{F}, \qquad f(\boldsymbol{w}^*) \leq f(\boldsymbol{w}).$$

Optimality conditions In general, finding global or even local minima is a hard problem. For this reason, researchers in optimization have developed optimality conditions: these are mathematical expressions that can be checked at a given point (unlike the conditions above) and help assessing whether a given point is a local minimum or not.

In this introductory chapter, we will present these conditions in the context of an unconstrained optimization problem

$$\operatorname*{minimize}_{\boldsymbol{w}\in\mathbb{R}^d} f(\boldsymbol{w}). \tag{1.3.6}$$

Theorem 1.3.3 (First-order necessary condition) *Suppose that the objective function f in problem (1.3.6) belongs to $\mathcal{C}^1(\mathbb{R}^d)$. Then,*

$$[\,\boldsymbol{w}^* \text{ is a local minimum of } f\,] \implies \|\nabla f(\boldsymbol{w}^*)\| = 0. \tag{1.3.7}$$

Note that this condition is only necessary: there may exist points with zero gradient that are not local minima. Indeed, the set of points with zero gradient, called *first-order stationary points*, also includes local maxima and saddle points[1].

Provided we strengthen our smoothness requirements on f, we can establish stronger optimality conditions for problem (2.1.1).

[1] A vector is a saddle point of a function if it is a local minimum with respect to certain directions and a local maximum with respect to other directions of the space.

Theorem 1.3.4 (Second-order necessary condition) *Suppose that the objective function f in problem (1.3.6) belongs to $\mathcal{C}^2(\mathbb{R}^d)$. Then,*

$$[\,\boldsymbol{w}^* \text{ is a local minimum of } f\,] \implies \left[\|\nabla f(\boldsymbol{w}^*)\| = 0 \quad \text{and} \quad \nabla^2 f(\boldsymbol{w}^*) \succeq \mathbf{0}\right]. \tag{1.3.8}$$

From Theorem 1.3.3, first-order stationary points that violate the condition $\nabla^2 f(\boldsymbol{w}^*) \succeq \mathbf{0}$ cannot be local minima: conversely, a stronger version of this property guarantees that we are in presence of a local minimum.

Theorem 1.3.5 (Second-order sufficient condition) *Suppose that the objective function f in problem (1.3.6) belongs to $\mathcal{C}^2(\mathbb{R}^d)$. Then,*

$$\left[\|\nabla f(\boldsymbol{w}^*)\| = 0 \quad \text{and} \quad \nabla^2 f(\boldsymbol{w}^*) \succ \mathbf{0}\right] \implies [\,\boldsymbol{w}^* \text{ is a local minimum of } f\,] \tag{1.3.9}$$

By exploiting the second-order derivative, it is thus possible to certify whether a point is a local minima (note that there could be local or even global minima such that $\nabla^2 f(\boldsymbol{w}^*) \succeq \mathbf{0}$). With further assumptions on the structure of the problem, these optimality conditions can be more informative about minima. This is the case when the objective function is convex: we detail this property in the next section.

1.3.3 Convexity

Convexity is at its core a geometric notion: before defining what a convex function is, we describe the corresponding property for a set.

Definition 1.3.16 (Convex set) *A set $\mathcal{C} \in \mathbb{R}^d$ is called* **convex** *if*

$$\forall (\boldsymbol{u}, \boldsymbol{v}) \in \mathcal{C}^2,\ \forall t \in [0,1], \qquad t\boldsymbol{u} + (1-t)\boldsymbol{v} \in \mathcal{C}.$$

Example 1.3.1 (Examples of convex sets) *The following sets are convex:*

- *The entire space $\mathbb{R}^d$;*
- *Every line segment of the form $\{t\boldsymbol{w} | t \in \mathbb{R}\}$ for some $\boldsymbol{w} \in \mathbb{R}^d$;*
- *Every (Euclidean) ball of the form $\left\{\boldsymbol{w} \in \mathbb{R}^d \,\middle|\, \|\boldsymbol{w}\|^2 = \sum_{i=1}^d [\boldsymbol{w}]_i^2 \leq 1\right\}$.*

We now provide the basic definition of a convex function.

Definition 1.3.17 (Convex function) *A function $f : \mathbb{R}^d \to \mathbb{R}$ is* **convex** *if*

$$\forall (\boldsymbol{u}, \boldsymbol{v}) \in (\mathbb{R}^d)^2,\ \forall t \in [0,1], \qquad f(t\boldsymbol{u} + (1-t)\boldsymbol{v}) \leq t\, f(\boldsymbol{u}) + (1-t)\, f(\boldsymbol{v}).$$

Example 1.3.2 *The following functions are convex :*

- *Linear functions of the form $\boldsymbol{w} \mapsto \boldsymbol{a}^{\mathrm{T}}\boldsymbol{w} + b$, with $\boldsymbol{a} \in \mathbb{R}^d$ and $b \in \mathbb{R}$;*
- *Squared Euclidean norm: $\boldsymbol{w} \mapsto \|\boldsymbol{w}\|^2 = \boldsymbol{w}^{\mathrm{T}}\boldsymbol{w}$.*

If we consider differentiable functions, it is possible to characterize convexity using the derivatives of the function.

Theorem 1.3.6 *Let $f : \mathbb{R}^d \to \mathbb{R}$ be an element of $\mathcal{C}^1(\mathbb{R}^d)$. Then, the function f is convex if and only if*

$$\forall \boldsymbol{u}, \boldsymbol{v} \in \mathbb{R}^d, \quad f(v) \geq f(u) + \nabla f(u)^{\mathrm{T}}(v - u). \tag{1.3.10}$$

The inequality (1.3.10) is fundamental in analyzing convex optimization algorithms, as it provides an underestimator for the variation of a (convex) objective function.

Convexity can also be characterized using the Hessian matrix (provided the function is sufficiently regular).

Theorem 1.3.7 *Let $f : \mathbb{R}^d \to \mathbb{R}$ be an element of $\mathcal{C}^2(\mathbb{R}^d)$. Then, the function f is convex if and only if*

$$\forall \boldsymbol{w} \in \mathbb{R}^d, \quad \nabla^2 f(\boldsymbol{w}) \succeq \mathbf{0}. \tag{1.3.11}$$

Convex functions are particularly suitable for minimization problems as they satisfy the following property.

Theorem 1.3.8 *If f is a convex function, then every local minimum of f is a global minimum.*

If the function is differentiable, the optimality conditions as well as the characterization of convexity lead us to the following result.

Corollary 1.3.1 *If f is continuously differentiable, every point $\boldsymbol{w}^*$ such that $\|\nabla f(\boldsymbol{w}^*)\| = 0$ is a global minimum of f.*

Strong convexity The results above can be further improved by assuming that a convex function is strongly convex, as defined below.

Definition 1.3.18 (Strongly convex function) *A function $f : \mathbb{R}^d \to \mathbb{R}$ in $\mathcal{C}^1$ is μ-strongly convex (or* strongly convex of modulus $\mu > 0$*) if for all $(\boldsymbol{u}, \boldsymbol{v}) \in (\mathbb{R}^d)^2$ and $t \in [0, 1]$,*

$$f(t\boldsymbol{u} + (1-t)\boldsymbol{v}) \;\leq\; t\, f(\boldsymbol{u}) + (1-t) f(\boldsymbol{v}) - \frac{\mu}{2} t(1-t)\|\boldsymbol{v} - \boldsymbol{u}\|^2.$$

Strong convexity leads to an even more desirable property in terms of optimization landscape.

Theorem 1.3.9 *Any strongly convex function has a unique global minimizer.*

Similarly to convex functions, it is possible to characterize strong convexity using first- and second-order derivatives.

Theorem 1.3.10 *Let $f : \mathbb{R}^d \to \mathbb{R}$ be an element of $\mathcal{C}^1(\mathbb{R}^d)$. Then, the function f is μ-strongly convex if and only if*

$$\forall \boldsymbol{u}, \boldsymbol{v} \in \mathbb{R}^d, \quad f(\boldsymbol{v}) \geq f(\boldsymbol{u}) + \nabla f(\boldsymbol{u})^{\mathrm{T}}(\boldsymbol{v} - \boldsymbol{u}) + \frac{\mu}{2}\|\boldsymbol{v} - \boldsymbol{u}\|^2. \tag{1.3.12}$$

Theorem 1.3.11 *Let $f : \mathbb{R}^d \to \mathbb{R}$ be an element of $\mathcal{C}^2(\mathbb{R}^d)$. Then, the function f is μ-strongly convex if and only if*

$$\forall \boldsymbol{w} \in \mathbb{R}^d, \quad \nabla^2 f(\boldsymbol{w}) \succeq \mu \mathbf{I}. \tag{1.3.13}$$

We end this section by giving two examples of strongly convex optimization problems.

Example 1.3.3 (Convex quadratic problems) *Consider*

$$\underset{\boldsymbol{w}\in\mathbb{R}^d}{\text{minimize}}\ f(\boldsymbol{w}) := \frac{1}{2}\boldsymbol{w}^{\mathrm{T}}\boldsymbol{A}\boldsymbol{w} + \boldsymbol{b}^{\mathrm{T}}\boldsymbol{w}, \quad \boldsymbol{A} \succeq \mathbf{0}.$$

The function f belongs to $\mathcal{C}^2(\mathbb{R}^d)$, with $\nabla^2 f(\boldsymbol{w}) = \boldsymbol{A}$ for every $\boldsymbol{w} \in \mathbb{R}^d$. As a result, this function is convex. Moreover, if we assume that $\boldsymbol{A} \succ \mathbf{0}$, then the function is $\lambda_{\min}(\boldsymbol{A})$-strongly convex.

Example 1.3.4 (Projection onto a closed, convex set) *Let $\mathcal{X} \subseteq \mathbb{R}^d$ be a convex, closed[2] set, and $\boldsymbol{a} \in \mathbb{R}^d$. The problem of computing the projection of $\boldsymbol{a}$ onto $\mathcal{X}$ is formulated as*

$$\underset{\boldsymbol{w}\in\mathcal{X}}{\text{minimize}}\ \frac{1}{2}\|\boldsymbol{w} - \boldsymbol{a}\|^2.$$

The objective function of this problem is 1-strongly convex, which implies that the problem has a unique solution (i. e. the projection is unique).

1.4 Examples of optimization problems in ML

1.4.1 Linear regression

Linear least squares is arguably the most classical problem in data analysis. We consider a dataset $\{(\boldsymbol{x}_i, y_i)\}_{i=1}^n$ with $\boldsymbol{x}_i \in \mathbb{R}^d$ and $y_i \in \mathbb{R}$. Our goal is to compute a linear model that best fits (or explains) the data. We define this model as a function $h : \mathbb{R}^d \to \mathbb{R}$, and we parameterize it through a vector $\boldsymbol{w} \in \mathbb{R}^d$, so that for any $\boldsymbol{x} \in \mathbb{R}^d$, we have $h(\boldsymbol{x}) = \boldsymbol{x}^{\mathrm{T}}\boldsymbol{w}$. For every example $(\boldsymbol{x}_i, y_i)$ in the dataset, we evaluate how we fit the data based on the squared error $(\boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{w} - y_i)^2$. We then compute a model by solving the following optimization problem

$$\underset{\boldsymbol{w}\in\mathbb{R}^d}{\text{minimize}}\ \frac{1}{2n}\|\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y}\|^2 + \frac{\lambda}{2}\|\boldsymbol{w}\|^2. = \frac{1}{n}\sum_{i=1}^n \tfrac{1}{2}\left[(\boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{w} - y_i)^2 + \lambda\|\boldsymbol{w}\|^2\right], \tag{1.4.1}$$

where $\lambda > 0$ is a regularization parameter. From an optimizer's point of view, problem (1.4.1) is well understood: this is a strongly convex, quadratic problem, and its solution can be computed in close form.

In a typical linear regression setting, one assumes that there exists an underlying truth but that the measurements are noisy, i.e.

$$\boldsymbol{y} = \boldsymbol{X}\boldsymbol{w}^* + \boldsymbol{\epsilon},$$

where $\boldsymbol{\epsilon} \in \mathcal{N}(\mathbf{0}, \mathbf{I})$ is a vector with i.i.d. entries following a standard normal distribution: this is illustrated in Figure 1.2.

In this setting, we wish to compute the most likely value for $\boldsymbol{w}^*$, while being robust to variance in the data. To this end, we suppose that $\boldsymbol{y}$ follows a Gaussian distribution of mean $\boldsymbol{X}\boldsymbol{w}$ and of covariance matrix $\mathbf{I}$. We also assume a prior Gaussian distribution on the entries of $\boldsymbol{w}$, in order to

[2] A set $\mathcal{X} \subseteq \mathbb{R}^d$ is closed if for every converging subsequence of $\{\boldsymbol{x}_n\}_n$, the limit of this sequence belongs to $\mathcal{X}$.

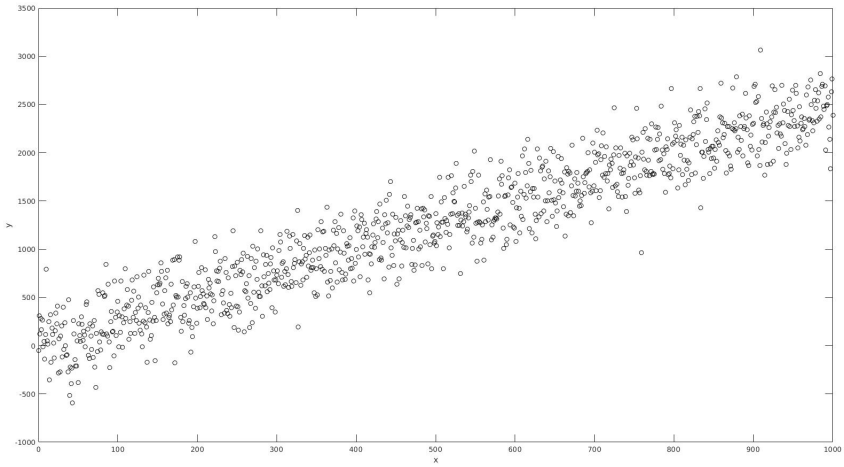

Figure 1.2: Noisy data generated from a linear model with Gaussian noise.

reduce the variance with respect to the data. As a result, an estimate of $\boldsymbol{w}^*$, called the maximum a posteriori estimator, can be computed by solving

$$\underset{\boldsymbol{w}\in\mathbb{R}^d}{\text{maximize}}\, L(y_1,\ldots,y_n;\boldsymbol{w}) := \left[\frac{1}{\sqrt{2\pi}}\right]^m \exp\left(-\frac{1}{2}\sum_{i=1}^{m}(\boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{w}-y_i)^2-\frac{\lambda}{2}\|\boldsymbol{w}\|^2\right). \tag{1.4.2}$$

The solutions of this maximization problem are the same than the solutions of the linear least-squares problem (1.4.1). The resulting solution can be shown to possess very favorable statistical properties: in particular, for λ close to 0, its expected value is close to $\boldsymbol{w}^*$.

Linear regression (with or without regularization) has been extensively studied in optimization and statistics; however, when the number of samples is extremely large, it still poses a number of challenges in practice, as the solution of the problem cannot be computed exactly.

1.4.2 Logistic regression

As in Section 1.4.1, we consider a dataset $\{(\boldsymbol{x}_i, y_i)\}_{i=1}^n$ where $\boldsymbol{x}_i \in \mathbb{R}^d$ are feature vectors, and the y_is represent binary labels. We wish to build a linear classifier $\boldsymbol{x} \mapsto \boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}$ to perform this classification, i. e. identify the correct label from the feature. We first suppose that $y_i \in \{-1,+1\}$. To model these discrete-valued labels, we introduce an *odds-like* function

$$p(\boldsymbol{x};\boldsymbol{w}) = (1+e^{\boldsymbol{x}^{\mathrm{T}}\boldsymbol{w}})^{-1} \in (0,1).$$

Given this function, our goal is to choose the model $\boldsymbol{w}$ such that

$$\begin{cases} p(\boldsymbol{x}_i;\boldsymbol{w}) \approx 1 & \text{if } y_i = +1; \\ p(\boldsymbol{x}_i;\boldsymbol{w}) \approx 0 & \text{if } y_i = -1. \end{cases}$$

Given this goal, we want to build an objective function that measures the error between our model and the labels according to the property above. Therefore, we penalize situations in which $y_i = +1$

and $p(\boldsymbol{x}_i; \boldsymbol{w})$ is close to 0, or $y_i = -1$ and $p(\boldsymbol{x}_i; \boldsymbol{w})$ is close to 1. This results in the so-called logistic loss, which is a function from $\mathbb{R}^d$ to $\mathbb{R}$ defined by

$$\forall \boldsymbol{w} \in \mathbb{R}^d, f(\boldsymbol{w}) = \frac{1}{n} \left\{ \sum_{y_i=-1} \ln\left(1 + e^{-\boldsymbol{x}_i^{\mathrm{T}} \boldsymbol{w}}\right) + \sum_{y_i=+1} \ln\left(1 + e^{\boldsymbol{x}_i^{\mathrm{T}} \boldsymbol{w}}\right) \right\}. \tag{1.4.3}$$

The motivation behind introducing the logarithm of the function p is twofold. On the one hand, it provides a statistical interpretation of the loss as a joint distribution; on the other hand, the derivatives of this function have a more favorable structure.

Given this objective function, the **logistic regression** problem is given by

$$\underset{\boldsymbol{w} \in \mathbb{R}^d}{\text{minimize}} \frac{1}{n} \left\{ \sum_{y_i=-1} \ln\left(1 + e^{-\boldsymbol{x}_i^{\mathrm{T}} \boldsymbol{w}}\right) + \sum_{y_i=+1} \ln\left(1 + e^{\boldsymbol{x}_i^{\mathrm{T}} \boldsymbol{w}}\right) \right\} \tag{1.4.4}$$

This is a convex, smooth problem (though not a strongly convex one), that can be made strongly convex by adding a regularizing term (see Chapter 3).

1.4.3 Linear SVM

To illustrate the role of optimization in data-related applications, we consider a binary classification problem, illustrated in Figure 1.3.

The red circles and blue squares appear at the same locations on all three figures : they represent data samples identified by their coordinates, while their color or shape represents a certain class they belong to. Our goal is to compute a linear classifier, that is, a separator of the two classes corresponding to a linear function. Each of the three figures shows a separator that achieves the task of classifying the data (the separator is the same for the middle and right plots): in that sense, the task involving the *samples* has been performed. However, if we envision the samples as being part of a (much) larger dataset, represented by the blue and red blobs, it becomes clear that the best classifier is the one on the rightmost figure.

These observations can be modeled and summarized using a mathematical framework. Let $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$ be n vectors of $\mathbb{R}^d$, and suppose that each vector $\boldsymbol{x}_i$ is given a label $y(\boldsymbol{x}_i) \in \{-1, 1\}$ (say, -1 for a red point and 1 for a blue point). Then, one can translate the binary classification problem into the following optimization problem:

$$\underset{\substack{\boldsymbol{u} \in \mathbb{R}^d \\ v \in \mathbb{R}}}{\text{minimize}} \frac{1}{n} \sum_{i=1}^{n} \max\left\{1 - y(\boldsymbol{x}_i)(\boldsymbol{x}_i^{\mathrm{T}} \boldsymbol{u} - v), 0\right\} \tag{1.4.5}$$

Here we seek a linear function defined by $\boldsymbol{x} \mapsto \boldsymbol{x}^{\mathrm{T}} \boldsymbol{u} - v$: it thus suffices to compute its coefficients, given by the vector $\boldsymbol{u}$ and the scalar v (see Section 1.2 for a formal definition of these concepts).

1.4.4 Neural networks

Neural networks have enabled the most impressive, recent advances in perceptual tasks such as image recognition and classification. Thanks to the increase in computational capabilities over the past decade, it is now possible to train extremely deep and wide neural networks, so that they can learn efficient representations of the data.

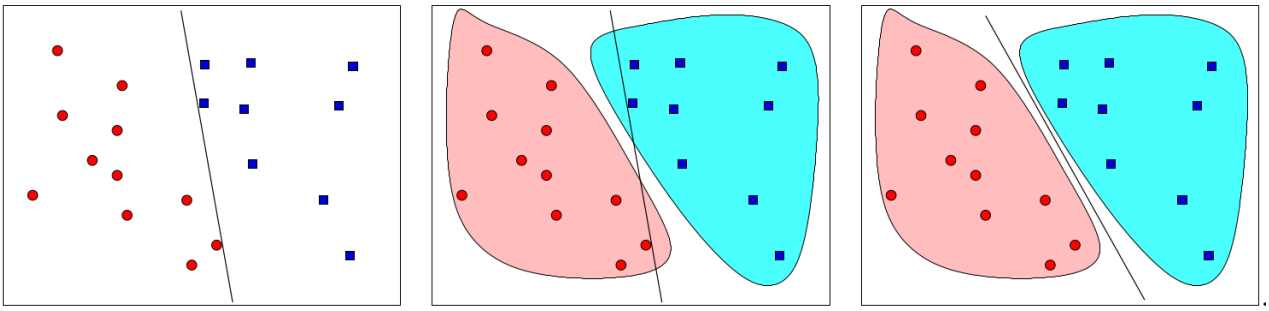

Figure 1.3: A sample for binary classification (red circles/blue squares) and the associated distribution (light red set/light blue set). A same linear classifier is shown on the left and middle plot: although it classifies the samples correctly, its closeness to several data points make it sensitive to the data, and prevents it from correctly classifying the distribution. On the contrary, the linear classifier on the right plot (that has a maximal margin of separation) provides a better classification, and is able to generalize to the distributions. *Source : S. J. Wright and B. Recht, Optimization for Data Analysis [4].*

Given an input vector $\boldsymbol{x}_i \in \mathbb{R}^{d_0}$, a neural network represents a prediction function $h : \mathbb{R}^{d_0} \to \mathbb{R}^{d_J}$, which applies a series of transformations in layers $\boldsymbol{x}_i = \boldsymbol{x}_i^{(0)} \mapsto \boldsymbol{x}_i^{(1)} \mapsto \cdots \mapsto \boldsymbol{x}_i^{(J-1)} \mapsto \boldsymbol{x}_i^{(J)}$. The j-th layer typically performs the following transformation:

$$\boldsymbol{x}_i^{(j)} = \boldsymbol{\sigma}\left(\boldsymbol{W}_j \boldsymbol{x}_i^{(j-1)} + \boldsymbol{b}_j\right) \in \mathbb{R}^{d_j}, \tag{1.4.6}$$

where $\boldsymbol{W}_j \in \mathbb{R}^{d_j \times d_{j-1}}$, $\boldsymbol{b}_j \in \mathbb{R}^{d_j}$ and $\boldsymbol{\sigma} : \mathbb{R}^{d_j} \to \mathbb{R}^{d_j}$ is a componentwise nonlinear function, e.g. $\boldsymbol{\sigma}(\boldsymbol{y}) = \left[\frac{1}{1+\exp(-y_i)}\right]_i$ (sigmoid function) or $\boldsymbol{\sigma}(\boldsymbol{y}) = [\max(0, y_i)]_i$. As a result, we have $\boldsymbol{x}_i^{(J)} = h(\boldsymbol{x}_i; \boldsymbol{w})$, where $\boldsymbol{w} \in \mathbb{R}^d$ gathers all the parameters $\{(\boldsymbol{W}_1, \boldsymbol{b}_1), \ldots, (\boldsymbol{W}_J, \boldsymbol{b}_J)\}$ of the layers.

The optimization problem corresponding to training this neural network architecture involves a training set $\{(\boldsymbol{x}_i, y_i)\}_{i=1}^n$ and the choice of a loss function ℓ. It usually results in the following formulation

$$\underset{\boldsymbol{w} \in \mathbb{R}^d}{\text{minimize}} \frac{1}{n} \sum_{i=1}^{n} \ell\left(h(\boldsymbol{x}_i; \boldsymbol{w}), y_i\right). \tag{1.4.7}$$

This optimization problem is highly nonlinear and nonconvex in nature, which makes it particularly difficult to solve using algorithms such as gradient descent. Moreover, it typically involves costly algebraic operations, as the number of layers and/or parameters is tremendously large in modern deep neural network architectures. Therefore, problem (1.4.7) also possesses characteristics that are not accounted for in its formulation. The optimization algorithms that efficiently tackle this problem are those that can both guarantee convergence and perform well in practice.

1.5 Optimization algorithms

The field of optimization can be broadly divided into three categories:

- Mathematical optimization is concerned with the theoretical study of complex optimization formulations, and the proof of well-posedness of such problems (for instance, prove that their exist solutions);

- Computational optimization deals with the development of software that can solve a family of optimization problems, through careful implementation of efficient methods;
- **Algorithmic** optimization lies in-between the previous two categories, and aims at proposing new algorithms that address a particular issue, with theoretical guarantees and/or validation of their practical interest.

These notes cover material from the third category of optimization activities. The design of optimization algorithms (also called methods, or schemes) is a particularly subtle process, as an algorithm must exploit the theoretical properties of the problem while being amenable to implementation on a computer.

1.5.1 The algorithmic process

Most numerical optimization algorithms do not attempt to find a solution of a problem in a direct way, and rather proceed in an *iterative* fashion. Given a current point, that represents the current approximation to the solution, an optimization procedure attempts to move towards a (potentially) better point: to this end, the method generally requires a certain amount of calculation.

Suppose we apply such a process to the problem $\text{minimize}_{\boldsymbol{w}\in\mathbb{R}^d} f(\boldsymbol{w})$, resulting in a sequence of iterates $\{\boldsymbol{w}_k\}_k$. Ideally, these iterates obey one of the scenarios below:

1. The iterates get increasingly close to a solution, i. e.

$$\|\boldsymbol{w}_k - \boldsymbol{w}^*\| \to 0 \quad \text{when } k \to \infty.$$

   Although $\boldsymbol{w}^*$ is generally not known in practice, such results can be guaranteed by the theory, for instance on strongly convex problems.

2. The function values associated with the iterates get increasingly close to the optimum, i. e.

$$f(\boldsymbol{w}_k) \to f^* \quad \text{when } k \to \infty,$$

   As for the case above, f^* may not be known, but it can still be possible to prove convergence for certain algorithms and function classes (typically strongly convex, smooth functions).

3. The first-order optimality condition gets close to being satisfied, that is, $f \in \mathcal{C}^1(\mathbb{R}^d)$ and

$$\|\nabla f(\boldsymbol{w}_k)\| \to 0 \quad \text{when } k \to \infty.$$

Out of the three conditions, the last one is the easiest to track as the algorithm unfolds: it is, however, only a necessary condition, and does not guarantee convergence to a local minimum for generic, nonconvex functions. On the other hand, the first two conditions can only be measured approximately (by looking at the behavior of the iterates and enforcing decrease in the function values), but lead to stronger guarantees.

1.5.2 Convergence and convergence rates

The typical theoretical results that optimizers aim at proving for algorithms are asymptotic, as shown above: they only provide a guarantee in the limit. In practice, one may want to obtain more precise guarantees, that relate to a certain accuracy target that the practitioner would like to achieve. This led to the development of **global convergence rates**.

Example 1.5.1 (Global convergence rate for the gradient norm) *Given an algorithm applied to minimize$_{\boldsymbol{w}\in\mathbb{R}^d} f(\boldsymbol{w})$ that produces a sequence of iterates $\{\boldsymbol{w}_k\}$, we say that the method is $\mathcal{O}(1/k)$ for the gradient norm, or $\|\nabla f(\boldsymbol{w}_k)\| = \mathcal{O}\left(\frac{1}{k}\right)$ if*

$$\exists C > 0, \quad \|\nabla f(\boldsymbol{w}_k)\| \leq \frac{C}{k} \; \forall k.$$

Such rates allow to quantify how much effort (in terms of iterations) is needed to reach a certain target accuracy $\epsilon > 0$. This leads to the companion notion of **worst-case complexity bound**.

Example 1.5.2 (Worst-case complexity for the gradient norm) *Given an algorithm applied to minimize$_{\boldsymbol{w}\in\mathbb{R}^d} f(\boldsymbol{w})$ that produces a sequence of iterates $\{\boldsymbol{w}_k\}$, we say that the method has a worst-case complexity of $\mathcal{O}\left(\epsilon^{-1}\right)$ for the gradient norm if*

$$\exists C > 0, \; \|\nabla f(\boldsymbol{w}_k)\| \leq \epsilon \text{ when } k \geq \frac{C}{\epsilon}.$$

Such results are quite common in theoretical computer science or statistics, which partly explain their popularity in machine learning. In optimization, they have been developed for a number of years in the context of convex optimization but have only gained momentum in general optimization over the last decade.

Remark 1.5.1 (The computational side of optimization) *The most popular programming languages for optimization are C/C++/Fortran for high performance implementations, with Python and Julia raising increasing interest. The use of MATLAB is also widespread throughout the optimization community.*

In addition to programming languages, optimizers have developed **modeling** *languages that help bringing the code and the mathematical formulation of a problem closer. The broad-spectrum languages GAMS/AMPL/CVX are reknown examples; other languages, that are more domain-oriented, include MATPOWER and PyTorch.*

Finally, there are many commercial solvers available (with CPLEX and Gurobi being arguably some of the most efficient for certain classes of problems), along with open-source codes (the COIN-OR platform provides a good interface to all of these methods).

1.6 Summary

Optimization is a key component of modern science, with many tasks in machine learning and related fields involving an optimization problem of some form. The specifics of dealing with massive amounts of data, yet possibly not enough to perfectly model the task at hand, poses a challenge to optimizers. Still, optimization algorithms can prove quite useful to help practitioners in data science (and beyond) in making better decisions.

Optimization begins by a modeling phase, in which a given problem must be stated in terms of objective, variable and constraints. This allows to characterize the properties of the problem, and most importantly its solutions. Properties such as differentiability or convexity lead to specific conditions that one can exploit to identify solutions of this problem.

In general, it is not possible to directly compute a solution of an optimization problem from its formulation; one must thus design a method that will try to compute an approximate solution of the problem. By analyzing this method, it is often possible to identify how fast a method can be at getting close to a solution.

Chapter 2

Smooth optimization methods

In this chapter, we review the main methods for solving smooth unconstrained optimization problems. Our starting point will be the Gradient Descent (GD) algorithm, which we study from a theoretical and computational viewpoint in Section 2.1. We will then focus on convex problems and investigate accelerated techniques in Section 2.2.

2.1 Gradient descent

In this section, we investigate more general, nonlinear unconstrained problems of the form

$$\underset{\boldsymbol{w}\in\mathbb{R}^d}{\text{minimize}}\, f(\boldsymbol{w}). \tag{2.1.1}$$

We will assume that $f \in \mathcal{C}^1(\mathbb{R}^d)$, therefore the gradient mapping for f exists, is continuous: we will also assume that it can be used in an algorithm. We will develop an algorithm that primarily relies on the use of gradient information, termed gradient descent. For such a method, we will derive theoretical guarantees with and without the assumption of convexity: in the latter case, we will see that better results are obtained compared to the general, nonconvex setting.

2.1.1 Algorithm

Because we consider a problem with a continuously differentiable function, we know from the optimality conditions that for any local minimum $\boldsymbol{w}^*$, we necessarily have $\nabla f(\boldsymbol{w}^*) = 0$. As a result, given any point $\boldsymbol{w} \in \mathbb{R}^d$, only one of the two properties below holds:

1. Either $\nabla f(\boldsymbol{w}) = 0$, and $\boldsymbol{w}$ can be a local minimum;
2. Or $\nabla f(\boldsymbol{w}) \neq 0$ and the function f decreases *locally* from $\boldsymbol{w}$ in the direction of -$\nabla f(\boldsymbol{w})$.

We will formally establish the second property in the next section, thanks to the Taylor expansions we derived in Section 1.3.1. Using this result, we can design the update rule

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \alpha \nabla f(\boldsymbol{w}), \tag{2.1.2}$$

where $\alpha > 0$ is a stepsize parameter. If $\nabla f(\boldsymbol{w}) = 0$, the vector $\boldsymbol{w}$ does not change: this is consistent with the notion of first-order stationarity (we cannot get more information by using the gradient).

On the contrary, when $\nabla f(\boldsymbol{w}) \neq 0$, we expect that there exists a range of values for $\alpha > 0$ for which such an update leads to a point with a lower objective value.

Using the updating rule (2.1.2), we can design an algorithm for the minimization of the function f: this method is called **gradient descent**[1] and described in Algorithm 1.

Algorithm 1: Gradient descent algorithm.

Initialization: $\boldsymbol{w}_0 \in \mathbb{R}^d$.
for $k = 0, 1, \ldots$ **do**

1. Compute the gradient $\nabla f(\boldsymbol{w}_k)$.
2. Compute a steplength $\alpha_k > 0$.
3. Set $\boldsymbol{w}_{k+1} = \boldsymbol{w}_k - \alpha_k \nabla f(\boldsymbol{w}_k)$.

end

As written, Algorithm 1 does not have any stopping criterion, and a number of variants can be derived depending on the choice of this stopping criterion, that of the initial point and that of the sequence $\{\alpha_k\}_k$. We comment on these aspects below.

Stopping criterion In general numerical algorithms operate under a certain budget (of floating-point operations, time, number of iterations), thus any reasonable numerical algorithm will have an embedded stopping criterion, that forces the method to terminate if this budget is reached. In Algorithm 1, for instance, we could have stopped the method after $k_{\max}$ iterations.

In addition to these practical concerns, algorithms are run in the hope of reaching a prescribed level of accuracy, corresponding to the metrics we described in Section 1.3. For instance, a typical stopping criterion (also called convergence criterion) for gradient descent is

$$\|\nabla f(\boldsymbol{w}_k)\| < \epsilon, \tag{2.1.3}$$

where $\epsilon > 0$ is a prescribed tolerance, convergence being supposedly harder to achieve as ϵ gets smaller.

Finally, additional safety checks can be added to the algorithm. For instance, if the difference between two successive points falls below machine precision, it may not be worth running the method for more iterations.

Choosing the initial point Good initialization can lead to significant gains in performance, that must however be put in perspective with the cost of this initialization. For general problems, there could be no incentive to choose one point over another: in this case, random multistart (i.e. running multiple versions of the method with randomly generated starting points) can be used with a small budget to determine a suitable initial point. However, in many applications, the practitioner might already have a reference point, or take an educated guess at what values the decision variables could take: using this as a starting point can be quite valuable, as it will represent a reference value the method is trying to improve upon.

[1] Although "gradient descent" is the most common terminology in data science, the historical name used in optimization is "steepest descent", because the gradient is the direction of steepest change at a given point.

2.1.2 Choosing the stepsize

There are numerous techniques used to select the stepsize[2]. We review the most general below, but point out that those are generally combined with knowledge about the problem in practice.

Constant stepsize One possible strategy is to maintain a constant step size throughout the entire algorithmic run, i. e. set $\alpha_k = \alpha > 0$. If the budget allows for it, several values of α can be tested for comparison. Under regularity assumptions on f, one can guarantee that there exists a value below which a constant stepsize will lead to complexity guarantees (see Section 2.1.3). For instance, when $f \in \mathcal{C}_L^{1,1}(\mathbb{R}^d)$, the choice

$$\alpha_k = \alpha = \tfrac{1}{L} \tag{2.1.4}$$

leads to such guarantees. Because of its dependence in L, this choice is tailored to the problem at hand. Note that the rule (2.1.4) requires knowledge of the Lipschitz constant, but this information may not be available in practice.

Decreasing stepsize Another popular choice consist in choosing the entire sequence $\{\alpha_k\}$ in advance so as to guarantee that $\alpha_k \to 0$ as $k \to \infty$. This also enables the derivation of theoretical results, under some conditions that can help designing the formula for the α_ks. However, this process forces the steps to get increasingly smaller, which may prevent fast progress towards the end of the algorithm.

Adaptive choice with line search Line-search techniques have been widely used in continuous optimization: at every iteration, they aim at computing the value of α_k that leads to the largest decrease in the function value in the direction $-\nabla f(\boldsymbol{w}_k)$. In general, such exact line searches are not practical, and thus an inexact process is preferred. The most popular method is backtracking, that proceeds by testing a set of decreasing values: a simple version of a backtracking line search is described in Algorithm 2.

Algorithm 2: Basic backtracking line search in direction d.

Inputs: $\boldsymbol{w} \in \mathbb{R}^d$, $\boldsymbol{d} \in \mathbb{R}^d$, $\alpha_0 \in \mathbb{R}^d$.
Initialization: Set $\alpha = \alpha_0$ and $j = 0$.
while $f(\boldsymbol{w} + \alpha_j \boldsymbol{d}) > f(\boldsymbol{w})$ **do**
  Set $\alpha_j = \frac{\alpha_j}{2}$ and $j = j + 1$.
end
Output: α_j.

We can thus incorporate this line-search technique in step 2 of Algorithm 1 by calling the method with $\boldsymbol{w} = \boldsymbol{w}_k$, $\boldsymbol{d} = -\nabla f(\boldsymbol{w}_k)$ and (for instance) $\alpha_0 = 1$. Many variants can be build upon this simple framework. One drawback of line-search methods is that they require to evaluate the objective function, which can be deemed too expensive in certain applications.

[2] Or *learning rate* in machine learning.

2.1.3 Convergence rate analysis of gradient descent

In this section, we present several convergence rates for gradient descent, in the case of a smooth objective function. We will see that the nonconvex, convex and strongly convex cases exhibit different behavior.

Proposition 2.1.1 *Consider the k-th iteration of Algorithm 1 applied to $f \in \mathcal{C}_L^{1,1}(\mathbb{R}^d)$, and suppose that $\nabla f(\boldsymbol{w}_k) \neq 0$. Then, if $0 < \alpha_k < \frac{2}{L}$, we have*

$$f(\boldsymbol{w}_k - \alpha_k \nabla f(\boldsymbol{w}_k)) < f(\boldsymbol{w}_k).$$

In particular, choosing $\alpha_k = \frac{1}{L}$ leads to

$$f(\boldsymbol{w}_k - \frac{1}{L}\nabla f(\boldsymbol{w}_k)) < f(\boldsymbol{w}_k) - \frac{1}{2L}\|\nabla f(\boldsymbol{w}_k)\|^2. \tag{2.1.5}$$

Proof. We use the inequality (1.3.2) with the vectors $(\boldsymbol{w}_k, \boldsymbol{w}_k - \alpha_k \nabla f(\boldsymbol{w}_k))$:

$$\begin{aligned} f(\boldsymbol{w}_k - \alpha_l \nabla f(\boldsymbol{w}_k)) &\leq f(\boldsymbol{w}_k) + \nabla f(\boldsymbol{w}_k)^{\mathrm{T}} [-\alpha_k \nabla f(\boldsymbol{w}_k)] + \frac{L}{2}\| - \alpha_k \nabla f(\boldsymbol{w}_k)\|^2 \\ &= f(\boldsymbol{w}_k) - \alpha_k \nabla f(\boldsymbol{w}_k)^{\mathrm{T}} \nabla f(\boldsymbol{w}_k) + \frac{L}{2}\alpha_k^2 \|\nabla f(\boldsymbol{w}_k)\|^2 \\ &= f(\boldsymbol{w}_k) + \left(-\alpha_k + \frac{L}{2}\alpha_k^2\right) \|\nabla f(\boldsymbol{w}_k)\|^2. \end{aligned}$$

If $-\alpha_k + \frac{L}{2}\alpha_k^2 < 0$, the second term on the right-hand side will be negative, thus we will have $f(\boldsymbol{w}_k - \alpha_l \nabla f(\boldsymbol{w}_k)) < f(\boldsymbol{w}_k)$. Since $-\alpha_k + \frac{L}{2}\alpha_k^2 < 0 \Leftrightarrow \alpha_k < \frac{2}{L}$ and $\alpha_k > 0$ by definition, this proves the first part of the result.

To obtain (2.1.5), one simply needs to use $\alpha_k = \frac{1}{L}$ in the series of equations above. □

The result of Proposition 2.1.1 will be instrumental to obtain complexity guarantees on Algorithm 1 in three different settings (nonconvex, convex, strongly convex): this analysis will be performed under the following assumption.

Assumption 2.1.1 *The objective function f belongs to $\mathcal{C}_L^{1,1}(\mathbb{R}^d)$ for $L > 0$ and there exists $f_{low} \in \mathbb{R}$ such that for every $\boldsymbol{w} \in \mathbb{R}^d$, $f(\boldsymbol{w}) \geq f_{low}$ (i. e. f is bounded below on $\mathbb{R}^d$).*

Nonconvex case In the nonconvex case, we aim at bounding the number of iterations required to drive the gradient norm below some threshold $\epsilon > 0$: this means that we should be able to show that the gradient norm actually goes below this threshold, which is a guarantee of convergence.

Theorem 2.1.1 (Complexity of gradient descent for nonconvex functions) *Let f be a nonconvex function satisfying Assumption 2.1.1. Suppose that Algorithm 1 is applied with $\alpha_k = \frac{1}{L}$. Then, for any $K \geq 1$, we have*

$$\min_{0 \leq k \leq K-1} \|\nabla f(\boldsymbol{w}_k)\| \leq \mathcal{O}\left(\frac{1}{\sqrt{K}}\right). \tag{2.1.6}$$

Proof. Let K be an iteration index such that for every $k = 0, \ldots, K-1$, we have $\|\nabla f(\boldsymbol{w}_k)\| > \epsilon$. From Proposition 2.1.1, we have that

$$\forall k = 0, \ldots, K-1, \quad f(\boldsymbol{w}_{k+1}) \leq f(\boldsymbol{w}_k) - \frac{1}{2L}\|\nabla f(\boldsymbol{w}_k)\|^2 \leq f(\boldsymbol{w}_k) - \frac{1}{2L}\left(\min_{0 \leq k \leq K-1} \|\nabla f(\boldsymbol{w}_k)\|\right)^2.$$

By summing across all such iterations, we obtain :

$$\sum_{k=0}^{K-1} f(\boldsymbol{w}_{k+1}) \leq \sum_{k=0}^{K-1} f(\boldsymbol{w}_k) - \frac{K}{2L}\left(\min_{0\leq k\leq K-1} \|\nabla f(\boldsymbol{w}_k)\|\right)^2.$$

Removing identical terms on both sides yields

$$f(\boldsymbol{w}_K) \leq f(\boldsymbol{w}_0) - \frac{K}{2L}\left(\min_{0\leq k\leq K-1} \|\nabla f(\boldsymbol{w}_k)\|\right)^2.$$

Using $f(\boldsymbol{w}_K) \geq f_{low}$ (which holds by Assumption 2.1.1) and re-arranging the terms leads to

$$\min_{0\leq k\leq K-1} \|\nabla f(\boldsymbol{w}_k)\| \leq \left[\frac{2L(f(\boldsymbol{w}_0) - f_{low})}{K}\right]^{1/2} = \mathcal{O}\left(\frac{1}{\sqrt{K}}\right).$$

□

Equivalently, we say that the worst-case complexity of gradient descent is $\mathcal{O}\left(\epsilon^{-2}\right)$, because for any $\epsilon > 0$, a reasoning similar to the proof of Theorem 2.1.1 guarantees that $\min_{0\leq k\leq K-1} \|\nabla f(\boldsymbol{w}_k)\| \leq \epsilon$ after at most

$$\left\lceil 2L(f(\boldsymbol{w}_0) - f_{low})\epsilon^{-2}\right\rceil = \mathcal{O}(\epsilon^{-2})$$

iterations.

Convex/Strongly convex case In addition to Assumption 2.1.1, if we further assume that the objective is convex or strongly convex, we can show that stronger guarantees than that of the nonconvex case can be obtained at a lower cost. This improvement illustrates the interest of convex functions in optimization.

In this paragraph, we let $f^* = \min_{\boldsymbol{w}\in\mathbb{R}^d} f(\boldsymbol{w})$ denote the minimal value of f (note that $f^* \geq f_{low}$) and we assume that there exists $\boldsymbol{w}^* \in \mathbb{R}^d$ such that $f(\boldsymbol{w}^*) = f^*$ (i.e. the set of minima is not empty). Given an accuracy threshold $\epsilon > 0$, we are interested in bounding the number of iterations necessary to reach an iterate $\boldsymbol{w}_k$ such that $f(\boldsymbol{w}_k) - f^* \leq \epsilon$.

Theorem 2.1.2 *Convergence of gradient descent for convex functions Let f be a convex function satisfying Assumption 2.1.1. Suppose that Algorithm 1 is applied with $\alpha_k = \frac{1}{L}$. Then, for any $K \geq 1$, the iterate $\boldsymbol{w}_K$ satisfies*

$$f(\boldsymbol{w}_k) - f^* \leq \mathcal{O}\left(\frac{1}{K}\right). \tag{2.1.7}$$

method runs for at most $\mathcal{O}(\epsilon^{-1})$ iterations before computing $\boldsymbol{w}_k$ such that $f(\boldsymbol{w}_k) - f^* \leq \epsilon$.

Proof. Let K be an index such that for every $k = 0, \ldots, K-1$, $f(\boldsymbol{w}_k) - f^* > \epsilon$.
For any $k = 0, \ldots, K-1$, the characterization of convexity (1.3.10) at $\boldsymbol{w}_k$ and $\boldsymbol{w}^*$ gives

$$f(\boldsymbol{w}^*) \geq f(\boldsymbol{w}_k) + \nabla f(\boldsymbol{w}_k)^{\mathrm{T}}(\boldsymbol{w}^* - \boldsymbol{w}_k).$$

Combining this property with (2.1.5), we obtain:

$$\begin{aligned} f(\boldsymbol{w}_{k+1}) &\leq f(\boldsymbol{w}_k) - \frac{1}{2L}\|\nabla f(\boldsymbol{w}_k)\|^2 \\ &\leq f(\boldsymbol{w}^*) + \nabla f(\boldsymbol{w}_k)^{\mathrm{T}}(\boldsymbol{w}_k - \boldsymbol{w}^*) - \frac{1}{2L}\|\nabla f(\boldsymbol{w}_k)\|^2. \end{aligned}$$

To proceed onto the next step, one notices that

$$\nabla f(\boldsymbol{w}_k)^{\mathrm{T}}(\boldsymbol{w}_k - \boldsymbol{w}^*) - \frac{1}{2L}\|\nabla f(\boldsymbol{w}_k)\|^2 = \frac{L}{2}\left(\|\boldsymbol{w}_k - \boldsymbol{w}^*\|^2 - \|\boldsymbol{w}_k - \boldsymbol{w}^* - \frac{1}{L}\nabla f(\boldsymbol{w}_k)\|^2\right).$$

Thus, recalling that $\boldsymbol{w}_{k+1} = \boldsymbol{w}_k - \frac{1}{L}\nabla f(\boldsymbol{w}_k)$, we arrive at

$$\begin{aligned} f(\boldsymbol{w}_{k+1}) &\leq f(\boldsymbol{w}^*) + \frac{L}{2}\left(\|\boldsymbol{w}_k - \boldsymbol{w}^*\|^2 - \|\boldsymbol{w}_k - \boldsymbol{w}^* - \frac{1}{L}\nabla f(\boldsymbol{w}_k)\|^2\right) \\ &= f(\boldsymbol{w}^*) + \frac{L}{2}\left(\|\boldsymbol{w}_k - \boldsymbol{w}^*\|^2 - \|\boldsymbol{w}_{k+1} - \boldsymbol{w}^*\|^2\right). \end{aligned}$$

Hence,

$$f(\boldsymbol{w}_{k+1}) - f(\boldsymbol{w}^*) \leq \frac{L}{2}\left(\|\boldsymbol{w}_k - \boldsymbol{w}^*\|^2 - \|\boldsymbol{w}_{k+1} - \boldsymbol{w}^*\|^2\right). \tag{2.1.8}$$

By summing (2.1.8) on all indices k between 0 and $K-1$, we obtain

$$\sum_{k=0}^{K-1} f(\boldsymbol{w}_{k+1}) - f(\boldsymbol{w}^*) \leq \frac{L}{2}\left(\|\boldsymbol{w}_0 - \boldsymbol{w}^*\|^2 - \|\boldsymbol{w}_K - \boldsymbol{w}^*\|^2\right) \leq \frac{L}{2}\|\boldsymbol{w}_0 - \boldsymbol{w}^*\|^2.$$

Finally, using $f(\boldsymbol{w}_0) \geq f(\boldsymbol{w}_1) \geq \ldots \geq f(\boldsymbol{w}_K)$ (a consequence of Proposition 2.1.1, we obtain that

$$\sum_{k=0}^{K-1} f(\boldsymbol{w}_{k+1}) - f(\boldsymbol{w}^*) \geq K\left(f(\boldsymbol{w}_K) - f^*\right).$$

Injecting this formula into the previous equation finally yields the desired outcome:

$$f(\boldsymbol{w}_k) - f(\boldsymbol{w}^*) \leq \frac{L\|\boldsymbol{w}_0 - \boldsymbol{w}^*\|^2}{2}\frac{1}{K}.$$

□

Equivalently, we say that the worst-case complexity of gradient descent is $\mathcal{O}\left(\epsilon^{-1}\right)$, which means here that there exist a positive constant C (that depends on $\|\boldsymbol{w}_0 - \boldsymbol{w}^*\|$ and L) such that

$$f(\boldsymbol{w}_K) - f_{low} \leq \epsilon.$$

after at most $C\epsilon^{-1}$ iterations.

We now turn to the strongly convex case.

Theorem 2.1.3 *Convergence of gradient descent for strongly convex functions Let f be a μ-strongly convex function satisfying Assumption 2.1.1, with $\mu \in (0, L]$. Suppose that Algorithm 1 is applied with $\alpha_k = \frac{1}{L}$ and let $\epsilon > 0$. Then, for any $K \in \mathbb{N}$, we have*

$$f(\boldsymbol{w}_k) - f^* \leq \mathcal{O}\left((1 - \tfrac{\mu}{L})^k\right) \tag{2.1.9}$$

for at most $\mathcal{O}(\frac{L}{\mu}\ln(\frac{1}{\epsilon}))$ iterations before computing $\boldsymbol{w}_k$ such that $f(\boldsymbol{w}_k) - f^* \leq \epsilon$.

Equivalently, we say that the convergence rate of gradient descent is $\mathcal{O}\left((1 - \frac{\mu}{L})^k\right)$.

Proof. We exploit the strong convexity property (1.3.12). For any $(\boldsymbol{x}, \boldsymbol{y}) \in (\mathbb{R}^n)^2$, we have

$$f(\boldsymbol{y}) \geq f(\boldsymbol{x}) + \nabla f(\boldsymbol{x})^{\mathrm{T}}(\boldsymbol{y} - \boldsymbol{x}) + \frac{\mu}{2}\|\boldsymbol{y} - \boldsymbol{x}\|^2.$$

Minimizing both sides with respect to $\boldsymbol{y}$ lead to $\boldsymbol{y} = \boldsymbol{w}^*$ on the left-hand side, and $\boldsymbol{y} = \boldsymbol{x} - \frac{1}{\mu}\nabla f(\boldsymbol{x})$ on the right-hand side (see Example **??**). As a result, we obtain

$$\begin{aligned} f^* &\geq f(\boldsymbol{x}) + \nabla f(\boldsymbol{x})^{\mathrm{T}}\left[-\frac{1}{\mu}\nabla f(\boldsymbol{x})\right] + \frac{\mu}{2}\|-\frac{1}{\mu}\nabla f(\boldsymbol{x})\|^2 \\ f^* &\geq f(\boldsymbol{x}) - \frac{1}{2\mu}\|\nabla f(\boldsymbol{x})\|^2. \end{aligned}$$

By re-arranging the terms, we arrive at

$$\|\nabla f(\boldsymbol{x})\|^2 \geq 2\mu\left[f(\boldsymbol{x}) - f^*\right], \tag{2.1.10}$$

which is valid for any $\boldsymbol{x} \in \mathbb{R}^n$. Using (2.1.10) together with (2.1.5) thus gives

$$f(\boldsymbol{w}_{k+1}) \leq f(\boldsymbol{w}_k) - \frac{1}{2L}\|\nabla f(\boldsymbol{w}_k)\|^2 \leq f(\boldsymbol{w}_k) - \frac{\mu}{L}(f(\boldsymbol{w}_k) - f^*).$$

This leads to

$$f(\boldsymbol{w}_{k+1}) - f^* \leq \left(1 - \frac{\mu}{L}\right)(f(\boldsymbol{w}_k) - f^*),$$

which we can iterate in order to obtain

$$f(\boldsymbol{w}_K) - f^* \leq \left(1 - \frac{\mu}{L}\right)^K (f(\boldsymbol{w}_0) - f^*).$$

It then suffices to note that the bound is also valid for $K = 0$. □

Equivalently, we can show a worst-case complexity result: the method computes $\boldsymbol{w}_k$ such that $f(\boldsymbol{w}_k) - f^* \leq \epsilon$ in at most $\mathcal{O}(\frac{L}{\mu}\ln(\frac{1}{\epsilon}))$ iterations.

Similar results can be shown for the criterion $\|\boldsymbol{w}_k - \boldsymbol{w}^*\|$: in other words, the distance between the current iterate and the (unique) global optimum decreases at a rate $\mathcal{O}\left((1 - \frac{\mu}{L})^k\right)$.

Remark 2.1.1 *Proofs of convergence rates are typically more technical for convex and strongly convex problems: in order to obtain better bounds than in the nonconvex setting, one must make careful use of the (strong) convexity inequalities. In this course, we do not focus on these aspects, but rather draw insights from the final complexity bounds or convergence rates.*

2.1.4 Application: regression with logistic and sigmoid losses

As in Section **??**, we consider a dataset $\{(\boldsymbol{x}_i, y_i)\}_{i=1}^n$ where $\boldsymbol{x}_i \in \mathbb{R}^d$ are feature vectors, and the y_is represent binary labels. We wish to build a linear classifier $\boldsymbol{x} \mapsto \boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}$ to perform this classification, i. e. identify the correct label from the feature.

Logistic loss We first suppose that $y_i \in \{-1, +1\}$; to model these discrete-valued labels, we introduce an *odds-like* function

$$p(\boldsymbol{x}; \boldsymbol{w}) = (1 + e^{\boldsymbol{x}^{\mathrm{T}}\boldsymbol{w}})^{-1} \in (0, 1).$$

Given this function, our goal is to choose the model $\boldsymbol{w}$ such that

$$\begin{cases} p(\boldsymbol{x}_i; \boldsymbol{w}) \approx 1 & \text{if } y_i = +1; \\ p(\boldsymbol{x}_i; \boldsymbol{w}) \approx 0 & \text{if } y_i = -1. \end{cases}$$

Given this goal, we want to build an objective function that measures the error between our model and the labels according to the property above. Therefore, we penalize situations in which $y_i = +1$ and $p(\boldsymbol{x}_i; \boldsymbol{w})$ is close to 0, or $y_i = -1$ and $p(\boldsymbol{x}_i; \boldsymbol{w})$ is close to 1. This results in the so-called logistic loss, which is a function from $\mathbb{R}^d$ to $\mathbb{R}$ defined by

$$\forall \boldsymbol{w} \in \mathbb{R}^d, f(\boldsymbol{w}) = \frac{1}{n} \left\{ \sum_{y_i=-1} \ln\left(1 + e^{-\boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{w}}\right) + \sum_{y_i=+1} \ln\left(1 + e^{\boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{w}}\right) \right\}. \tag{2.1.11}$$

The motivation behind introducing the logarithm of the function p is twofold. On the one hand, it provides a statistical interpretation of the loss as a joint distribution; on the other hand, the derivatives of this function have a more favorable structure.

Given this objective function, the **logistic regression** problem is given by

$$\min_{\boldsymbol{w} \in \mathbb{R}^d} \frac{1}{n} \left\{ \sum_{y_i=-1} \ln\left(1 + e^{-\boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{w}}\right) + \sum_{y_i=+1} \ln\left(1 + e^{\boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{w}}\right) \right\} \tag{2.1.12}$$

This is a convex, smooth problem (though not a strongly convex one), that can be made strongly convex by adding a regularizing term, which will be done in a subsequent chapter. In both cases, we can apply gradient descent with guaranteed convergence rates.

Sigmoid loss We now assume that $y_i \in \{0, 1\}$ for every i.In this case, and for similar reasons than in the case of the logistic loss, we can measure agreement between the model and the label for example i by looking at the sigmoid function

$$\phi(\boldsymbol{x}_i; \boldsymbol{w}) = \left(1 + e^{-\boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{w}}\right)^{-1};$$

Drawing inspiration from Section **??**, we may want to penalize the average of the squared errors $(y_i - \phi(\boldsymbol{x}_i; \boldsymbol{w}))^2$. This is the philosophy behind the nonlinear regression problem:

$$\min_{\boldsymbol{w} \in \mathbb{R}^d} \frac{1}{n} \sum_{i=1}^{n} \left(y_i - \frac{1}{1+e^{-\boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{w}}} \right)^2. \tag{2.1.13}$$

This problem is a *nonlinear* least-squares problems: it is twice continuously differentiable, but non-convex. Therefore, we can apply gradient descent to this problem, but we will only be guaranteed to reach a first-order stationary point.

2.2 Acceleration techniques

2.2.1 Introduction: the concept of momentum

In Section 2.1.3, we derive complexity bounds for the gradient descent algorithm, and we saw in particular that assuming that the function was convex (respectively, strongly convex) improved the complexity. These results are called *upper* complexity bounds, in the sense that they reflect the worst possible convergence rate that this algorithm could exhibit on a given problem. The issue of *lower* bounds, that show a rate that cannot be improved upon, has been the subject to a lot of attention, particularly in the convex optimization community.

For nonconvex optimization, it is known that there exists a function for which gradient descent converges exactly at the $\mathcal{O}(\frac{1}{\sqrt{K}})$ rate: in this case, the lower bound matches the upper bound. On the contrary, for convex functions, the lower bound is actually $\mathcal{O}(\frac{1}{K^2})$, which is a sensible improvement over the bound in $\mathcal{O}(\frac{1}{K})$ of Theorem 2.1.2. There are methods that can achieve this bound, thanks to an algorithmic technique called **acceleration**.

The underlying idea of acceleration is that, at a given iteration and given the available information from previous iterations (in particular, the latest displacement), one can move along a better step than that given by the current gradient.

2.2.2 Nesterov's accelerated gradient method

Among the existing methods based on acceleration, the accelerated gradient algorithm proposed by Yurii Nesterov in 1983 is the most famous, to the point that it has been termed "Nesterov's algorithm".

Algorithm 3: Accelerated gradient method.

Initialization: $\boldsymbol{w}_0 \in \mathbb{R}^d$, $\boldsymbol{w}_{-1} = \boldsymbol{w}_0$.
for $k = 0, 1, \ldots$ **do**

1. Compute a steplength $\alpha_k > 0$ and a parameter $\beta_k > 0$.
2. Compute the new iterate as

$$\boldsymbol{w}_{k+1} = \boldsymbol{w}_k - \alpha_k \nabla f\left(\boldsymbol{w}_k + \beta_k(\boldsymbol{w}_k - \boldsymbol{w}_{k-1})\right) + \beta_k(\boldsymbol{w}_k - \boldsymbol{w}_{k-1}). \tag{2.2.1}$$

end

Algorithm 3 provides a description of the method. Like the gradient descent method of Section 2.1, it requires a single gradient calculation per iteration; however, unlike in gradient descent, the gradient is not evaluated at the current iterate $\boldsymbol{w}_k$, but at a combination of this iterate with the previous step $\boldsymbol{w}_k - \boldsymbol{w}_{k-1}$: this term is called the **momentum term**, and is key to the performance of accelerated gradient techniques.

Another view of the accelerated gradient descent is that of a two-loop recursion: given $\boldsymbol{w}_0$ and

$\boldsymbol{z}_0 = \boldsymbol{w}_0$, the update (2.2.1) can be rewritten as

$$\begin{cases} \boldsymbol{w}_{k+1} &= \boldsymbol{z}_k - \alpha_k \nabla f(\boldsymbol{z}_k) \\ \boldsymbol{z}_{k+1} &= \boldsymbol{w}_{k+1} + \beta_{k+1}(\boldsymbol{w}_{k+1} - \boldsymbol{w}_k). \end{cases} \tag{2.2.2}$$

This formulation decouples the two steps behind the accelerated gradient update: a gradient step on $\boldsymbol{z}_k$, combined with a momentum step on $\boldsymbol{w}_{k+1}$.

Choosing the parameters We now comment on the choice of the stepsize α_k and the momentum parameter β_k. The same techniques than those presented in Section 2.1.2 can be considered for the choice of α_k (stepsize parameter). As in the gradient descent case, the choice $\alpha_k = \frac{1}{L}$ is a standard one.

The choice of β_k is most crucial to obtaining the improved complexity bound. The standard values proposed by Nesterov depend on the nature of the objective function:

- If f is a μ-strongly convex, we set

$$\beta_k = \beta = \frac{\sqrt{L}-\sqrt{\mu}}{\sqrt{L}+\sqrt{\mu}} \tag{2.2.3}$$

  for every k. Note that this requires the knowledge of both the Lipschitz constant of the gradient and the strong convexity constant.

- For a general convex function f, β_k is computed in an adaptive way using two sequences, as follows:

$$t_{k+1} = \frac{1}{2}(1 + \sqrt{1 + 4t_k^2}), t_0 = 0, \quad \beta_k = \frac{t_k - 1}{t_{k+1}}. \tag{2.2.4}$$

The following informal theorem summarizes the complexity results that can be proven for Algorithm 3.

Theorem 2.2.1 *Consider Algorithm 3 applied to a convex function f satisfying Assumption 2.1.1, with $\alpha_k = \frac{1}{L}$, and let $\epsilon > 0$. Then, for any $K \geq 1$, the iterate $\boldsymbol{w}_K$ computed by Algorithm 3 satisfies*

i) $f(\boldsymbol{w}_K) - f^* \leq \mathcal{O}(\frac{1}{K^2})$ for a generic convex function if β_k is set according to the adaptive rule (2.2.4);

ii) At most $f(\boldsymbol{w}_K) - f^* \leq \left((1 - \sqrt{\frac{\mu}{L}})^K\right)$ for a μ-strongly convex function, provided β_k is set to the constant value given by (2.2.3).

Note that we can also derive worst-case complexity bounds for the accelerated gradient method, that show the same improvement. For instance, for strongly convex functions, we can establish that $f(\boldsymbol{w}_k) - f^* \leq \epsilon$ after at most $\mathcal{O}\left(\sqrt{\frac{L}{\mu}}\ln(\epsilon^{-1})\right)$ $\mathcal{O}\left(\frac{L}{\mu}\ln(\epsilon^{-1})\right)$.

2.2.3 Other accelerated methods

Heavy ball method The heavy ball method is a precursor of the accelerated gradient algorithm, that was proposed by Boris T. Polyak in 1964. Its k-th iteration can be written as

$$\boldsymbol{w}_{k+1} = \boldsymbol{w}_k - \alpha \nabla f(\boldsymbol{w}_k) + \beta(\boldsymbol{w}_k - \boldsymbol{w}_{k+1}),$$

where the stepsize and momentum parameters are chosen to be constant values. The key difference between this iteration and Nesterov's lies in the gradient evaluation, which the heavy ball method performs at the current point: in that sense, the heavy ball method performs first the gradient update, then the momentum step, while Nesterov's method adopts the inverse approach. This method achieves the optimal rate of convergence on strongly convex quadratic functions, but can fail on general strongly convex functions.

Conjugate gradient The (linear) conjugate gradient method, proposed by Hestenes and Stiefel in 1952, has remained to this day one of the preferred methods to solve linear systems of equations and strongly convex quadratic minimization problems. Unlike Polyak's method, the conjugate gradient algorithm does not require knowledge of the Lipschitz constant L nor the parameter μ, because it exploits knowledge from the past iterations. The k-th iteration of conjugate gradient can be written as:

$$\boldsymbol{w}_{k+1} = \boldsymbol{w}_k + \alpha_k p_k, \quad p_k = -\nabla f(x_k) + \beta_k p_{k-1}.$$

In a standard conjugate gradient algorithm, α_k and β_k are computed using formulas tailored to the problem: this contributes to their convergence rate analysis, which leads to a rate similar to that of accelerated gradient. However, unlike accelerated gradient, the conjugate gradient is guaranteed to terminate after d iterations on a d-dimensional problem. When d is very large, the bound for conjugate gradient matches that of the other methods, and in that sense does not depend on the problem dimension.

Example 2.2.1 (Strongly convex quadratic minimization) *A strongly convex quadratic minimization problem is an optimization problem of the form*

$$\underset{\boldsymbol{w}\in\mathbb{R}^d}{\text{minimize}} \; q(\boldsymbol{w}) := \tfrac{1}{2}\boldsymbol{w}^{\mathrm{T}}\boldsymbol{A}\boldsymbol{w} - \boldsymbol{b}^{\mathrm{T}}\boldsymbol{w}$$

where $\boldsymbol{A} \in \mathbb{R}^{d\times d}$ is a symmetric positive definite matrix and $\boldsymbol{b} \in \mathbb{R}^d$. This problem is smooth (because the objective is polynomial in all of the decision variables) and $\nabla^2 f(\boldsymbol{w}) \succ \boldsymbol{0}$ for every $\boldsymbol{w}$, meaning that the problem is μ-strongly convex with μ denoting the minimum eigenvalue of $\boldsymbol{A}$. As a result, there exist a unique global minimum given by the solution of $\nabla q(\boldsymbol{w}) = \boldsymbol{A}\boldsymbol{w} - \boldsymbol{b} = 0$. This equation is a linear system but the cost of inverting this system and computing a solution can be prohibitive. For this reason, one can replace the exact solve by an iterative, gradient-based approach, and apply Algorithm 1 or Algorithm 3. Note that $q \in \mathcal{C}^{1,1}_{\|\boldsymbol{A}\|}(\mathbb{R}^d)$, hence the choice of steplength 2.1.4 is a valid one.

If gradient descent is applied, then an ϵ-accuracy in the objective value can be reached in at most $\mathcal{O}\left(\frac{L}{\mu}\ln(\frac{1}{\epsilon})\right)$ iterations, while if one applies the accelerated gradient or the heavy ball method with appropriately chosen parameters, this bound improves to $\mathcal{O}\left(\frac{L}{\mu}\ln(\frac{1}{\epsilon}\right)$. Finally, if we aim at using conjugate gradient, the result bound will be in $\mathcal{O}\left(\min\{d, \frac{L}{\mu}\ln(\frac{1}{\epsilon})\}\right)$.

2.3 Conclusion

The most classical optimization problems involve linear algebra: this is the case for linear least squares as well as eigenvalue and singular value calculations, that can be viewed as solutions of optimization problems. For these problems, it is possible to compute the solution explicitly (or in closed form). Linear least squares is a particular case of such instances.

For general unconstrained optimization problems, it is not possible to obtain a closed-form expression of the solution(s). As a result, one must construct algorithms that proceed iteratively to move from a starting point towards a solution. The gradient descent method is the canonical example of such a framework: many variants have been built on this paradigm, especially regarding the choice of the stepsize (or learning rate in machine learning applications). To analyze the behavior of gradient descent, one can establish global convergence rates (or, equivalently, global complexity bounds) that can be refined depending on the nature of the objective function. Indeed, gradient descent can be shown to converge faster on convex problems than on nonconvex ones, and even faster on strongly convex problems.

A natural question arising from these convergence rates results is whether those are optimal. For gradient descent applied to nonconvex, differentiable functions, it is not possible to improve over the rate established in Section 2.1.3. However, one can design accelerated methods for strongly convex and convex functions that possess better rates, a fact that reflects on the practical performance. These methods all rely on the concept of momentum, which is also exploited in state-of-the-art algorithms used to learn complex models in machine learning (e. g. Adagrad).

Chapter 3

Regularization

In this chapter, we investigate several challenges that can be posed while trying to apply stochastic gradient techniques to machine learning problems. To motivate these issues further, we will begin with an introductory example and method.

3.1 Introduction : The perceptron algorithm

Recall that in section 1.1, we introduced a linear SVM problem of the following form :

$$\min_{\boldsymbol{w}\in\mathbb{R}^d} \frac{1}{n}\sum_{i=1}^{n} \max\{1 - y_i \boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{w}, 0\} + \frac{\lambda}{2}\|\boldsymbol{w}\|_2^2 \tag{3.1.1}$$

where $\{(\boldsymbol{x}_i, y_i)\}_{i=1}^n$ represents the dataset, and $\lambda > 0$.

One of the earliest methods that was proposed to solve this algorithm is the **perceptron algorithm**, given in Algorithm 4.

Algorithm 4: Perceptron algorithm for problem 3.1.2.

Initialization: $\boldsymbol{w}_0 \in \mathbb{R}^d$, $\alpha > 0$.
for $k = 0, 1, \ldots$ **do**

1. Draw an index $i_k \in \{1, \ldots, n\}$ at random.
2. Compute the new iterate as

$$\boldsymbol{w}_{k+1} = \left(1 - \frac{\alpha\lambda}{n}\right)\boldsymbol{w}_k + \begin{cases} \alpha y_{i_k}\boldsymbol{x}_{i_k} & \text{if } 1 - y_{i_k}\boldsymbol{x}_{i_k}^{\mathrm{T}}\boldsymbol{w}_k > 0 \\ 0 & \text{otherwise,} \end{cases} \tag{3.1.2}$$

end

In its basic form, the preceptron algorithm is quite similar to stochastic gradient with a constant step size, in that it selects a single sample at every iteration and performs an update based on this

value. In fact, this would be exactly the stochastic gradient method if the problem were

$$\min_{\boldsymbol{w}\in\mathbb{R}^d} \frac{1}{n}\sum_{i=1}^{n}(1 - y_i \boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{w}) + \frac{\lambda}{2}\|\boldsymbol{w}\|_2^2.$$

However, this choice of loss function would not satisfy our desired requirements (see section 1.1). The *hinge loss* is a more meaningful quantity, however it is **nonsmooth**, i.e. the gradient does not exist at every point. In this situation, and with structured functions such as the hinge loss, it is possible to define quantities that act as a proxy for the gradient, and can thus drive the optimization process : we detail these aspects in Section 3.2.

Another interesting property of the problem (3.1.1) is that the objective function involves two terms: the hinge loss term, which depends on the data and a regularizing term, which does not depend on the data and serves to enforce structural properties on the solution. We will address this topic and the associated algorithms in Section 3.3.

3.2 Nonsmooth optimization

3.2.1 From nonsmooth functions to nonsmooth problems

Problems such as (3.1.1), that involve a function possibly not differentiable, are termed *nonsmooth problems*. They involve functions that we will call nonsmooth (by opposition with smooth) : for the purpose of these notes, we will define nonsmooth functions as follows.

Definition 3.2.1 (Nonsmooth functions) *A function $f : \mathbb{R}^d \to \mathbb{R}$ is called* **nonsmooth** *if it is not differentiable everywhere.*

Remark 3.2.1 *A nonsmooth function can be continuous (this is the case for the hinge loss above).*

Example 3.2.1 *Examples of nonsmooth functions*

- $w \mapsto |w|$ *from* $\mathbb{R}$ *to* $\mathbb{R}$*;*
- $w \mapsto \|w\|_1$ *from* $\mathbb{R}^d$ *to* $\mathbb{R}$*;*
- *ReLU:* $w \mapsto \max\{w, 0\}$ *from* $\mathbb{R}^d$ *to* $\mathbb{R}$*.*

Since nonsmooth functions are not differentiable everywhere, optimization problems that involve nonsmooth functions may be impossible to solve via gradient-based methods. Still, several approaches can be used to tackle these problems.

One useful technique consists in reformulating a nonsmooth problem as a smooth one when possible. For instance, the problem $\min_{w\in\mathbb{R}} |w|$ is equivalent to

$$\min_{w,t^+,t^-\in\mathbb{R}} t^+ + t^- \quad \text{s. t.} \quad w = t^+ - t^-, t^+ \geq 0, t^- \geq 0.$$

This reformulation is a smooth problem involving only linear objective and constraints, which is easily solvable by smooth solvers.

Another technique, frequently employed in practice, consists in working with functions that are nonsmooth but Lipschitz continuous (denoted by $\mathcal{C}_L^{0,0}$, by analogy with $\mathcal{C}_L^{1,1}$) and using a gradient-based scheme. This approach is motivated by the following property.

Theorem 3.2.1 *Let $f : \mathbb{R}^d \to \mathbb{R}$ be a Lipschitz continuous function. Then it is differentiable at almost every point in $\mathbb{R}^d$.*

For instance, the ReLU function is Lipschitz continuous (not differentiable at 0) thus most constructions involving ReLU (such as neural networks) would not be differentiable everywhere. However, most algorithms will operate under the assumption that the function is indeed differentiable. This is the case for most points (in fact, almost every point), but nonsmooth functions are likely to be non-differentiable at their minima, should they possess one.

3.2.2 Subgradient methods

In the case of convex functions, one can define a proxy for the gradient called the subgradient.

Definition 3.2.2 (Subgradient and subdifferential) *Let $f : \mathbb{R}^d \to \mathbb{R}$ be a convex function. A vector $\boldsymbol{g} \in \mathbb{R}^d$ is called a subgradient of f at $\boldsymbol{w} \in \mathbb{R}^d$ if*

$$\forall \boldsymbol{z} \in \mathbb{R}^n, \qquad f(\boldsymbol{z}) \;\geq\; f(\boldsymbol{w}) + \boldsymbol{g}^{\mathrm{T}}(\boldsymbol{z} - \boldsymbol{w}).$$

The set of all subgradients of f at $\boldsymbol{w}$ is called the subdifferential of f at $\boldsymbol{w}$, and denoted by $\partial f(\boldsymbol{w})$.

Note that when the function f is differentiable at $\boldsymbol{w}$, we have $\partial f(\boldsymbol{w}) = \{\nabla f(\boldsymbol{w})\}$, thus the notion of subdifferential matches that of the gradient for differentiable functions.

The interest of subgradients is further illustrated by the following result.

Theorem 3.2.2 *Let $f : \mathbb{R}^d \to \mathbb{R}$ be a convex function, and $\boldsymbol{w} \in \mathbb{R}^d$.*

$$\mathbf{0} \in \partial f(\boldsymbol{w}) \quad \Leftrightarrow \quad \boldsymbol{w} \text{ minimum of } f.$$

Example 3.2.2 *Let $f : \mathbb{R} \to \mathbb{R}$, $f(w) = |w|$.*

$$\partial f(w) \;=\; \begin{cases} -1 & \text{if } w < 0 \\ 1 & \text{if } w > 0 \\ [-1, 1] & \text{if } w = 0. \end{cases}$$

The set $[-1, 1]$ contains 0, which confirms that $w^* = 0$ is the minimum of f.

Remark 3.2.2 *Subgradients can also be defined for nonconvex functions, however in that case the subdifferential may be empty (typically at local maxima of the function).*

By analogy with gradient descent, we can design a subgradient method, as shown by Algorithm 5.

Such a method offers a flexibility in choosing the subgradient, which can be an issue. Moreover, choosing the stepsize is more difficult than for gradient descent, due to the nonsmooth nature of the problem. In fact, a subgradient can lead to increase in the function value for any stepsize, hence the choice of subgradient is critical to the success of this method.

Variants of subgradient method Based on the existing variants on the gradient descent paradigm, one can build algorithms that incorporate momentum and/or stochastic aspects; however, their analysis is also more intricate.

Algorithm 5: Subgradient descent method.

Initialization: $\boldsymbol{w}_0 \in \mathbb{R}^d$.
for $k = 0, 1, \ldots$ **do**

1. Compute a subgradient $\boldsymbol{g}_k \in \partial f(\boldsymbol{w}_k)$.
2. Compute a steplength $\alpha_k > 0$.
3. Set $\boldsymbol{w}_{k+1} = \boldsymbol{w}_k - \alpha_k \boldsymbol{g}_k$.

end

3.3 Regularization

3.3.1 Regularized problems

As we mentioned in introduction, a common practice in machine learning problems consists in enforcing a specific structure of the machine learning model through the objective function. Such regularized problems have the following form :

$$\underset{\boldsymbol{w}\in R^d}{\text{minimize}} \quad \underbrace{f(\boldsymbol{w})}_{loss\ function} + \underbrace{\lambda\Omega(\boldsymbol{w})}_{regularization\ term}.$$

where $\lambda > 0$ is called a regularization parameter.

Example 3.3.1 (Ridge regularization) *A problem with ridge regularization has the following form:*

$$\underset{\boldsymbol{w}\in\mathbb{R}^d}{\textit{minimize}}\, f(\boldsymbol{w}) + \frac{\lambda}{2}\|\boldsymbol{w}\|^2.$$

The ridge regularizer $\boldsymbol{w} \mapsto \frac{1}{2}\|\boldsymbol{w}\|^2$ has several interpretations. It effectively penalizes $\boldsymbol{w}$s with large components, and can be shown to be equivalent to a constraint on the squared norm $\|\boldsymbol{w}\|^2$. In addition, a ridge regularizer has the effect to reduce the variance of the problem solution with respect to the data. Finally, when the regularizer $\lambda > 0$ is big enough, this often turns the objective function into a strongly convex one, with the positive implications in terms of convergence speed and uniqueness of the (global) minimum.

3.3.2 Sparsity-inducing regularizers

While computing a model to explain some data, we might want to compute a model that explains the data using as few features as possible[1]. Mathematically speaking, if our model is parameterized by a vector $\boldsymbol{w} \in \mathbb{R}^d$, our goal is to compute a vector that explains the data with as few nonzero coordinates as possible.

There exists a regularizer that penalized vectors with nonzero components (not just large as opposed to the ridge regularizer), called the ℓ_0 norm [2]. An ℓ_0-regularized problem has the form

$$\underset{\boldsymbol{w}}{\text{minimize}}\, f(\boldsymbol{w}) + \lambda\|\boldsymbol{w}\|_0, \quad \|\boldsymbol{v}\|_0 = |\{i|[\boldsymbol{v}]_i \neq 0\}|.$$

[1] The goal of this process is *feature selection*.
[2] Though technically this function defines a semi-norm.

However, this function is nonsmooth and discontinuous; its combinatorial nature also introduces more complexity to the original problem. As a result, researchers have turned to an intermediate regularization term, the ℓ_1 norm defined by

$$\|\boldsymbol{w}\|_1 = \sum_{i=1}^{d} |w_i|. \tag{3.3.1}$$

This function is continuous and convex; moreover, it is a norm function, which endows it with many desirable properties.

An illustration of this method is given below.

Example 3.3.2 *LASSO (Least Absolute Shrinkage and Selection Operator) Consider the setting of linear regression with data $\boldsymbol{X} \in \mathbb{R}^{n\times d}$ and $\boldsymbol{y} \in \mathbb{R}^n$. With an ℓ_1 regularizer, the problem becomes:*

$$\underset{\boldsymbol{w}\in\mathbb{R}^d}{\text{minimize}}\, \frac{1}{2}\|\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y}\|^2 + \lambda\|\boldsymbol{w}\|_1.$$

The solution of this problem is known to possess fewer nonzero elements than the un-regularized, least-squares solution.

3.3.3 Proximal methods

Following our introduction of regularized problems in the previous section, we now describe optimization algorithms tailored to such formulations.

We begin by describing our problem class of interest.

Definition 3.3.1 (Composite optimization) *A composite optimization problem is of the form:*

$$\underset{\boldsymbol{w}\in\mathbb{R}^d}{\text{minimize}}\, f(\boldsymbol{w}) + \lambda\Omega(\boldsymbol{w}),$$

where $f : \mathbb{R}^d \to \mathbb{R}$ is a smooth, $C^{1,1}$ function, $\lambda > 0$ and $\Omega : \mathbb{R}^d \to \mathbb{R}$ is a convex, nonsmooth regularizer.

The **proximal approach** follows a classical optimization paradigm, in which a given problem is replaced by a sequence of easier problems called subproblems (note that all methods that we covered in these notes implicitly rely on these techniques). In the case of proximal methods, one aims at exploiting the smoothness of f to obtain easier problems, while using the structure of Ω directly into the subproblems.

Algorithm 6 gives a sketch of a proximal gradient method. The cost of an iteration of this algorithm is clearly more than that of other methods we have seen so far, given that it includes a gradient calculation as well as solving an auxiliary optimization problem (3.3.2), called the proximal subproblem.

Remark 3.3.1 *If $\Omega \equiv 0$ (i. e. Ω is the zero function and the problem is un-regularized), one can show that the solution of (3.3.2) is given by*

$$\boldsymbol{w}_{k+1} = \boldsymbol{w}_k - \alpha_k \nabla f(\boldsymbol{w}_k).$$

We thus recognize the gradient iteration of Algorithm 1.

Algorithm 6: Proximal gradient method.

Initialization: $\boldsymbol{w}_0 \in \mathbb{R}^d$.
for $k = 0, 1, \ldots$ **do**

1. Compute the gradient of the smooth part $\nabla f(\boldsymbol{w}_k)$.
2. Compute a steplength $\alpha_k > 0$.
3. Compute $\boldsymbol{w}_{k+1}$ such that

$$\boldsymbol{w}_{k+1} \in \underset{\boldsymbol{w}\in\mathbb{R}^d}{\operatorname{argmin}} \left\{ f(\boldsymbol{w}_k) + \nabla f(\boldsymbol{w}_k)^{\mathrm{T}}(\boldsymbol{w} - \boldsymbol{w}_k) + \tfrac{1}{2\alpha_k}\|\boldsymbol{w} - \boldsymbol{w}_k\|_2^2 + \lambda\Omega(\boldsymbol{w}) \right\}. \tag{3.3.2}$$

end

Proximal gradient methods can be designed using most of the tools that can be applied to gradient descent : this includes stepsize choices, acceleration as well as stochastic aspects. Moreover, complexity results exist for nonconvex and convex f, though the latter has attracted more attention in the literature.

Example of proximal method: ISTA We end this section on proximal methods by a instance of Algorithm 6 that has proven successful in signal and image processing. This method is dedicated to solving problems with an ℓ_1 regularization term, of the form:

$$\underset{\boldsymbol{w}\in\mathbb{R}^d}{\text{minimize}}\, f(\boldsymbol{w}) + \lambda\|\boldsymbol{w}\|_1.$$

Unlike for general regularizers, one can obtain a closed-form solution of the subproblem (3.3.2). Indeed, the proximal subproblem, given by

$$\underset{\boldsymbol{w}\in\mathbb{R}^d}{\text{minimize}} \left\{ f(\boldsymbol{w}_k) + \nabla f(\boldsymbol{w}_k)^{\mathrm{T}}(\boldsymbol{w} - \boldsymbol{w}_k) + \tfrac{1}{2\alpha_k}\|\boldsymbol{w} - \boldsymbol{w}_k\|_2^2 + \lambda\|\boldsymbol{w}\|_1 \right\},$$

has a unique solution. To obtain it, one computes the usual gradient step $\boldsymbol{w}_k - \alpha_k\nabla f(\boldsymbol{w}_k)$, then one applies the **soft-thresholding function** $s_{\alpha_k\lambda}(\bullet)$ to each component, where this function is given by

$$\forall \mu > 0, \forall t \in \mathbb{R}, \qquad s_\mu(t) = \begin{cases} t + \mu & \text{if } t < -\mu \\ t - \mu & \text{if } t > \mu \\ 0 & \text{otherwise.} \end{cases}$$

As a result, the solution of the proximal subproblem is defined component-wise according to the components of the gradient step. The resulting update is at the heart of the corresponding proximal algorithm, called ISTA (Iterative Soft-Thresholding Algorithm): a description of ISTA is given in Algorithm 7.

It can be shown that the use of the soft-thresholding function does promote zero components in the new iterates, which results in sparser solutions at the end of the algorithmic run.

Algorithm 7: ISTA: Iterative Soft-Thresholding Algorithm.

Initialization: $\boldsymbol{w}_0 \in \mathbb{R}^d$.
for $k = 0, 1, \ldots$ **do**

1. Compute the gradient of the smooth par $\nabla f(\boldsymbol{w}_k)$.
2. Compute a steplength $\alpha_k > 0$.
3. Compute $\boldsymbol{w}_{k+1}$ component-wise through the following rule

$$[\boldsymbol{w}_{k+1}]_i = \begin{cases} [\boldsymbol{w}_k - \alpha_k \nabla f(\boldsymbol{w}_k)]_i + \alpha_k \lambda & \text{if } [\boldsymbol{w}_k - \alpha_k \nabla f(\boldsymbol{w}_k)]_i < -\alpha_k \lambda \\ [\boldsymbol{w}_k - \alpha_k \nabla f(\boldsymbol{w}_k)]_i - \alpha_k \lambda & \text{if } [\boldsymbol{w}_k - \alpha_k \nabla f(\boldsymbol{w}_k)]_i > \alpha_k \lambda \\ 0 & \text{if } [\boldsymbol{w}_k - \alpha_k \nabla f(\boldsymbol{w}_k)]_i \in [-\alpha_k \lambda, \alpha_k \lambda]. \end{cases} \tag{3.3.3}$$

end

Remark 3.3.2 *A notable improvement on ISTA was the inclusion of momentum, which resulted in a new algorithm called FISTA (Fast ISTA): this method is now the most widely used instance of ISTA.*

3.4 Conclusion

Nonsmoothness is a very common property in optimization, that can lead to mild or major challenges in implementing algorithms to minimize nonsmooth functions. In certain cases, the structure and the impact of nonsmoothness are well understood; in other cases, generalized notions of derivative such as subgradients may have to come into play.

Nonsmoothness frequently arises in regularized problem, where the goal is to enforce properties for a model, that do not depend on the data. The optimization schemes of choice for these problems are proximal gradient methods, that proceed by solving subproblems involving the regularizer. For instance, the ℓ_1 regularizer, that promotes sparsity of the solution, can be tackled using the ISTA method. Note that a regularizer need not be nonsmooth, in which case a classical gradient method could be applied. This is for instance the case with the ℓ_2 regularizer, that aims at reducing variance with respect to the data, and leads to a smooth, possibly strongly convex problem.

Chapter 4

Stochastic optimization methods

4.1 Motivation

In this chapter, we will leverage the structure inherent to data science problems. More formally, we suppose that we have access to data samples $\{(\boldsymbol{x}_i, y_i)\}_{i=1}^n$, $\boldsymbol{x}_i \in \mathbb{R}^d$, $y_i \in \mathbb{R}$, that are drawn from an unknown distribution. As in the regression examples studied above, we seek a predictor function or a model h such that $h(\boldsymbol{x}_i) \approx y_i$ for every $i = 1, \dots, n$. Rather than optimizing over a space of models, we assume that a given model is defined by means of a vector $\boldsymbol{w} \in \mathbb{R}^d$ (i.e. $h(\boldsymbol{x}_i) = h(\boldsymbol{w}; \boldsymbol{x}_i)$). Therefore, we only need to determine the vector $\boldsymbol{w}$ in order to obtain the model.

To assess the accuracy of our model in predicting the data, we define a loss function, i.e. a mapping $\ell : (h, y) \mapsto \ell(h, y)$, that penalize pairs (h, y) such that $h \neq y$. We have already seen several examples of such losses (least-squares loss, sigmoid loss, etc). The loss at a given sample of the dataset thus is $\ell(h(\boldsymbol{w}; \boldsymbol{x}_i), y_i)$: in order to account for all samples, we consider the average of all losses as our objective to be minimized. This gives rise to the following optimization problem.

Definition 4.1.1 (Finite-sum optimization problem) *Given a dataset $\{(\boldsymbol{x}_i, y_i)\}_{i=1}^n$, $\boldsymbol{x}_i \in \mathbb{R}^d$, $y_i \in \mathbb{R}$, a class of predictor functions $\{h(\boldsymbol{w}; \cdot)\}_{\boldsymbol{w} \in \mathbb{R}^d}$ and a loss function ℓ, we define the corresponding optimization problem:*

$$\min_{\boldsymbol{w} \in \mathbb{R}^d} f(\boldsymbol{w}) = \frac{1}{n} \sum_{i=1}^n \ell(h(\boldsymbol{w}; \boldsymbol{x}_i), y_i) = \frac{1}{n} \sum_{i=1}^n f_i(\boldsymbol{w}). \tag{4.1.1}$$

Suppose that we apply gradient descent (Algorithm 1) to that problem, assuming all f_i are differentiable. The k-th iteration of this method is

$$\boldsymbol{w}_{k+1} = \boldsymbol{w}_k - \alpha_k \nabla f(\boldsymbol{w}_k) = \boldsymbol{w}_k - \frac{\alpha_k}{n} \sum_{i=1}^n \nabla f_i(\boldsymbol{w}).$$

From this update, we see that one iteration of gradient descent requires to look over the **entire dataset** in order to compute the gradient vector. In a big data setting where the number of samples n is very large, this cost can be prohibitive.

Remark 4.1.1 *In stochastic optimization, the data samples might be generated directly from the distribution, and be available in a streaming fashion. Instead of involving a discrete average on the*

sample, the resulting optimization problem would involve a mathematical expectation of the form

$$\min_{\boldsymbol{w}\in\mathbb{R}^d} \mathbb{E}_{(\boldsymbol{x},y)}\left[f_{(\boldsymbol{x},y)}(\boldsymbol{w})\right].$$

In such a context, the full gradient cannot be computed exactly. However, most of the reasoning of stochastic gradient will still be applicable.

4.2 Stochastic gradient algorithm

4.2.1 Algorithm

At its core, the idea of the stochastic gradient method is remarkably simple. Starting from the problem $\min_{\boldsymbol{w}\in\mathbb{R}^d} \frac{1}{n}\sum_{i=1}^n f_i(\boldsymbol{w})$, and assuming each component function f_i is differentiable, the method picks an index i at random and takes a step in the direction of the negative gradient of the component function f_i.

Algorithm 8: Stochastic gradient method.

Initialization: $\boldsymbol{w}_0 \in \mathbb{R}^d$.
for $k = 0, 1, \ldots$ **do**

1. Compute a steplength $\alpha_k > 0$.
2. Draw a random index $i_k \in \{1, \ldots, n\}$.
3. Compute the new iterate as

$$\boldsymbol{w}_{k+1} = \boldsymbol{w}_k - \alpha_k \nabla f_{i_k}(\boldsymbol{w}_k). \tag{4.2.1}$$

end

The key motivation for this process is that using a single data point at a time results in updates that are n **times cheaper than a full gradient step**.

Remark 4.2.1 *In general, considering independent updates may not be desirable. Consider for instance the problem* $\min_{w\in\mathbb{R}} \frac{1}{2}(f_1(w) + f_2(w))$ *with* $f_1(w) = 2w^2$ *and* $f_2 = -w^2$. *Starting from* $w_k > 0$, *drawing* $i_k = 2$ *will necessarily lead to an increase in the function value.*

In finite-sum problems arising from machine learning, the data samples are correlated enough that an update according to one sample might lead to improvement with respect to other samples as well: this is a key reason for the success of stochastic gradient methods in this setting.

Remark 4.2.2 *Algorithm 8 is often referred to as Stochastic Gradient Descent, or SGD, by analogy with Gradient Descent. However, for the reason mentioned in the previous remark, the stochastic gradient algorithm is not a descent method in general (as we will see in the next section, it can however produce descent in expectation). In these notes, we will adopt the terminology stochastic gradient.*

4.2.2 Analysis

We now describe the main arguments in deriving convergence rates for stochastic gradient, under a slightly modified version of Assumption 2.1.1.

Assumption 4.2.1 *The objective function $f = \frac{1}{n}\sum_{i=1}^n f_i$ belongs to $\mathcal{C}_L^{1,1}(\mathbb{R}^d)$ for $L > 0$ and there exists $f_{low} \in \mathbb{R}$ such that for every $\boldsymbol{w} \in \mathbb{R}^d$, $f(\boldsymbol{w}) \geq f_{low}$. Moreover, every function f_i belongs to $\mathcal{C}^1(\mathbb{R}^d)$.*

Recall that, for gradient descent, the key result was Proposition 2.1.1, which gave

$$f(\boldsymbol{w}_{k+1}) \leq f(\boldsymbol{w}_k) + \nabla f(\boldsymbol{w}_k)^{\mathrm{T}}(\boldsymbol{w}_{k+1} - \boldsymbol{w}_k) + \frac{L}{2}\|\boldsymbol{w}_{k+1} - \boldsymbol{w}_k\|^2.$$

A similar result can be shown for stochastic gradient under certain assumptions on how the random components are drawn. Those are summarized below.

Assumption 4.2.2 (Assumptions on stochastic gradient) *At any iteration of Algorithm 8 of index k, the index i_k is drawn independently from the previous indices $i_0, \ldots, i_{k-1}$ so that the following properties are satisfied:*

1. $\mathbb{E}_{i_k}\left[\nabla f_{i_k}(\boldsymbol{w}_k)\right] = \nabla f(\boldsymbol{w}_k)$;
2. $\mathbb{E}_{i_k}\left[\|\nabla f_{i_k}(\boldsymbol{w}_k)\|^2\right] \leq \sigma^2 + \|\nabla f(\boldsymbol{w}_k)\|^2$ *with* $\sigma^2 > 0$.

The first property of Assumption 4.2.2 forces the stochastic gradient $\nabla f_{i_k}(\boldsymbol{w}_k)$ to be an unbiased estimate of the true gradient $\nabla f(\boldsymbol{w}_k)$. The second property controls the variance of the norm of this stochastic gradient, so as to control the variations in its magnitude due to noise. Several strategies can be designed to draw an index i_k that satisfies these properties, the most classical of which is given below.

Example 4.2.1 (Uniform sampling) *Suppose that the k-th iteration of stochastic gradient draws the index i_k uniformly at random in $\{1, \ldots, n\}$. Then Algorithm 8 satisfies Assumption 4.2.2.*

Proposition 4.2.1 *Under Assumptions 2.1.1 and 4.2.2, consider the k-th iteration of Algorithm 8. Then,*

$$\mathbb{E}_{i_k}\left[f(\boldsymbol{w}_{k+1})\right] - f(\boldsymbol{w}_k) \leq \nabla f(\boldsymbol{w}_k)^{\mathrm{T}}\,\mathbb{E}_{i_k}\left[\boldsymbol{w}_{k+1} - \boldsymbol{w}_k\right] + \frac{L}{2}\,\mathbb{E}_{i_k}\left[\|\boldsymbol{w}_{k+1} - \boldsymbol{w}_k\|^2\right].$$

A stochastic gradient update will thus lead to decrease **in expectation**. Such a property suffices to derive convergence rates (or complexity results) for stochastic gradient applied to strongly convex, convex or nonconvex problems. Those results heavily depend upon the formula for the step sizes $\{\alpha_k\}_k$. In fact, one of the major problems in machine learning consists in tuning the learning rate, which corresponds to choosing the step size in stochastic gradient. We will illustrate the various challenges posed by this choice in the context of strongly convex functions.

Assumption 4.2.3 *The objective function is μ-strongly convex and possesses a unique global minimizer $\boldsymbol{w}^*$. We let $f^* = f(\boldsymbol{w}^*)$.*

We first provide a global rate result in the case of a constant step size.

Theorem 4.2.1 (SG with constant stepsize) *Let Assumptions 2.1.1, 4.2.2 and 4.2.3, and consider Algorithm 8 applied with a constant stepsize*

$$\alpha_k = \alpha \in (0, \tfrac{1}{2\mu}) \forall k.$$

Then,

$$\mathbb{E}\left[f(\boldsymbol{w}_k) - f^*\right] \leq \frac{\alpha L \sigma^2}{2\mu} + (1 - 2\alpha\mu)^k \left[f(\boldsymbol{w}_0) - f^* - \frac{\alpha L \sigma^2}{2\mu}\right]. \tag{4.2.2}$$

We note that this convergence rate corresponds to guaranteeing $\mathbb{E}\left[f(\boldsymbol{w}_k) - f^*\right] \leq \epsilon$ after at most $\mathcal{O}\left(\ln(1/\epsilon)\right)$ iterations. However, unlike in the gradient descent case, the tolerance ϵ cannot be arbitrarily close to zero. In fact, the use of stochastic gradients introduces an additional (bias) term $\frac{\alpha L \sigma^2}{2\mu}$. As a result, SG with constant stepsize can only be guaranteed to converge towards a **neighborhood** of the optimal function value f^*. On the other hand, such a method is capable of taking long steps, as opposed to the next technique based on decreasing step sizes.

In the original stochastic gradient method (proposed by Robbins and Monro in 1951), the stepsize sequence was required to satisfy

$$\sum_{k=0}^{\infty} \alpha_k = \infty \quad \text{and} \quad \sum_{k=0}^{\infty} \alpha_k^2 < \infty,$$

which implies that $\alpha_k \to 0$. In our next result, we thus consider the case of diminishing stepsizes.

Theorem 4.2.2 (SG with diminishing stepsize) *Let Assumptions 2.1.1, 4.2.2 and 4.2.3, and consider Algorithm 8 applied with a decreasing stepsize sequence $\{\alpha_k\}_k$ satisfying*

$$\alpha_k = \frac{\beta}{k + \gamma},$$

where $\beta > \frac{1}{\mu}$ and $\gamma > 0$ is chosen such that $\alpha_0 = \frac{\beta}{\gamma} \leq \frac{1}{L}$. Then,

$$\mathbb{E}\left[f(\boldsymbol{w}_k) - f^*\right] \leq \frac{\nu}{\gamma + k}, \tag{4.2.3}$$

where

$$\nu = \max\left\{\gamma(f(\boldsymbol{w}_0) - f^*), \frac{\beta^2 L \sigma^2}{2(\beta\mu - 1)}\right\}.$$

The decreasing stepsize choice possesses the same drawbacks than for gradient descent, namely that it results in increasingly small steps. It also provides a global convergence rate that is sublinear, as opposed to linear with a constant stepsize. Note, however, that SG with a decreasing stepsize is guaranteed to reach any neighborhood of a solution, unlike its variant with a constant stepsize.

Remark 4.2.3 (A practical constant stepsize approach) *A common practical strategy in machine learning consists in running the algorithm with a value α until the method stalls (which can indicate that the smallest neighborhood attainable with this stepsize choice has been reached). When that occurs, the stepsize can be reduced, and the algorithmic run can continue until it stalls again, then the stepsize will be further reduced, etc (say $\alpha, \alpha/2, \alpha/4$, etc). This process can lead to*

convergence guarantees, however the convergence is slower than that produced by constant stepsize SG:

$$\mathbb{E}\left[f(\boldsymbol{w}_k) - f^*\right] \leq \epsilon \quad \textit{after} \quad \mathcal{O}(1/\epsilon) \textit{ iterations.}$$

This choice of stepsize is adaptive, in that it is designed to reach closer and closer neighborhoods as the algorithm proceeds. However, it requires the method to be able to detect stalling, and act upon it.

Stepsize choice in the nonconvex setting Stochastic gradient (or some variant thereof) is the method of choice for training neural networks, which is usually a nonconvex problem. It is thus natural to ask whether global rates can be obtained for stochastic gradient in the nonconvex setting. The situation is significantly more complicated, as we get guarantees on

- $\mathbb{E}\left[\frac{1}{K}\sum_{i=1}^{K}\|\nabla f(\boldsymbol{w}_k)\|^2\right]$ for constant stepsizes;
- $\mathbb{E}\left[\frac{1}{\sum_{i=1}^{K}\alpha_k}\sum_{i=1}^{K}\alpha_k\|\nabla f(\boldsymbol{w}_k)\|^2\right]$ for decreasing stepsizes.

Similarly to the strongly convex case, the complexity bounds are affected by a residual term which in turns lead to worse rates than in the deterministic setting.

Example 4.2.2 *A typical stochastic gradient method with constant stepsize will satisfy*

$$\mathbb{E}\left[\frac{1}{K}\sum_{i=1}^{K}\|\nabla f(\boldsymbol{w}_k)\|^2\right] \leq \epsilon$$

in at most $\mathcal{O}(\epsilon^{-4})$ iterations, where ϵ is a sufficient large threshold of accuracy.

Remark 4.2.4 (What about momentum?) *The most successful implementations of stochastic gradient, such as ADAM, rely on some form of momentum incorporated in the stochastic gradient update. Intuitively, the hope is that incorporating momentum will allow the method to promote moves along good directions of decrease, while steps in bad directions will eventually cancel out. Some theory has been developed in the recent years to accelerate stochastic gradient yet, unlike in the deterministic setting, theory is still decorrelated from practice.*

4.3 Variance reduction

As we saw in the previous section, the theory for stochastic gradient is based on Assumption 4.2.2, and in particular on the fact that the variance of stochastic gradient estimates is bounded (by σ^2). It can clearly be seen from bounds such as (4.2.2) that the bigger σ is, the looser the bound becomes. More practically, this means that gradient estimates with high variance are unlikely to yield fast convergence.

Variance reduction techniques have precisely been developed in the aim of diminishing the variance of traditional stochastic gradient estimates. They can be categorized in two families, that either exploit more sampled gradients at every iteration, or use past history of the method. In these notes, we will focus on the former category.

4.3.1 Batch variants

We recall that the main part of Algorithm 8 consists in the update

$$\boldsymbol{w}_{k+1} = \boldsymbol{w}_k - \alpha_k \nabla f_{i_k}(\boldsymbol{w}_k),$$

where the index i_k is drawn at random. The use of a **single** sample is partially responsible for the importance of the variance term σ^2 in Assumption 4.2.2. One can thus consider stochastic gradient estimates that are built using *several* samples at once : this is the idea behind batch stochastic gradient.

Formally, the update of a batch stochastic gradient method is given by

$$\boldsymbol{w}_{k+1} = \boldsymbol{w}_k - \alpha_k \frac{1}{|S_k|} \sum_{i \in S_k} \nabla f_i(\boldsymbol{w}_k) \tag{4.3.1}$$

where $S_k \subset \{1, \dots, n\}$ is drawn at random. When S_k consists in a single index, we recover the usual stochastic gradient algorithm; conceptually, one could also consider a set S_k of cardinality n, in which case we would recover the usual gradient method.

Overall, two batch regimes can be distinguished:

- $|S_k| \approx n$, which has a cost essentially equivalent to that of a full gradient update;
- $|S_k| = n_b << n$, also called mini-batching, which may be advantageous in theory and variance reduction while still being affordable in practice. The resulting method is called mini-batch SG.

In fact, if we assume that $|S_k| = n_b \ \forall k$, it is possible to show that with the same stepsize, mini-batch SG requires n_b less iterations than SG. Moreover, mini-batch SGD can exploit parallel computing, by computing the n_b stochastic gradients on distributed processors. Moreover, we have the following property.

Proposition 4.3.1 *Under Assumptions 2.1.1 and 4.2.2, the variance of a mini-batch stochastic gradient estimate is given by*

$$\mathrm{Var}_{S_k}\left[\left\| \frac{1}{|S_k|} \sum_{i \in S_k} \nabla f_i(\boldsymbol{w}_k) \right\|_2\right] \leq \frac{\sigma^2}{n_b}.$$

As a final note, we mention that batch techniques are still more expensive than stochastic gradient, while being more sensitive to redundancies in the data. Tuning the best batch size is not necessarily an easy task. These concerns partly explain why stochastic gradient (or other schemes based on his sampling paradigm) remains the preferred approach.

4.3.2 Other variants

Gradient aggregation methods have attracted a lot of attention in the learning and optimization theory, because of the nice theory and algorithms that have been proposed and guarantee linear convergence rates. Their main principle consists in computing a **full gradient step** once in a while during the algorithmic run, in order to correct high-variance components. Despite their strong guarantees, they have not been widely exploited in practice, due to the cost of full gradient evaluations, that is still too prohibitive in certain applications.

Iterate averaging is another popular technique, that can be easier to implement. The underlying idea consists in analyzing (and possibly returning as output) the average iterate of a run of stochastic gradient, given by $\frac{1}{K}\sum_{k=0}^{K-1} \boldsymbol{w}_k$. In certain contexts (e.g. $\alpha = \frac{1}{\mu(k+1)}$ and f μ-strongly convex), this average has good properties with respect to the optimization, and is also a more robust solution than the last iterate obtained. However, returning this average either requires to store the history of iterates, or to maintain an average which can be prone to cancellation or numerical errors.

4.4 Stochastic gradient methods for deep learning

In this section, we focus on stochastic gradient algorithms that have proven useful in training deep learning models (though the methods we will present are not tailored to a particular architecture).

We again consider a finite-sum problem of the form (4.1.1) under Assumption 4.2.1. Our objective is to analyze several variants on the basic scheme

$$\boldsymbol{w}_{k+1} = \boldsymbol{w}_k - \alpha \boldsymbol{g}_k, \tag{4.4.1}$$

where $\alpha > 0$ is a stepsize (also known as learning rate in the machine learning community) and $\boldsymbol{g}_k$ is a stochastic gradient estimator, that either corresponds to a single gradient component (as in vanilla stochastic gradient) or a batch of indices.

We will present all our variants within a unified framework that highlights the key features of these methods: this framework is given by the iteration

$$\boldsymbol{w}_{k+1} = \boldsymbol{w}_k - \alpha \boldsymbol{m}_k \oslash \boldsymbol{v}_k, \tag{4.4.2}$$

where $\alpha > 0$, $\boldsymbol{m}_k, \boldsymbol{v}_k \in \mathbb{R}^d$ and $\oslash$ denotes the componentwise division, i. e.

$$\boldsymbol{m}_k \oslash \boldsymbol{v}_k := \left[\frac{[\boldsymbol{m}_k]_i}{[\boldsymbol{v}_k]_i}\right]_{i=1,\dots,d}.$$

Note that by letting $\boldsymbol{m}_k = \boldsymbol{g}_k$ and $\boldsymbol{v}_k = \mathbf{1}_{\mathbb{R}^d}$, we recover the classical stochastic gradient iteration (4.4.1).

4.4.1 Stochastic gradient with momentum

Inspired by the accelerated methods that we investigated in Chapter 2, we first consider adding momentum to the basic iteration (4.4.1). The most common approach, called **stochastic gradient with momentum**, corresponds to the iteration:

$$\boldsymbol{w}_{k+1} = \boldsymbol{w}_k - \alpha(1-\beta_1)\boldsymbol{g}_k + \alpha\beta_1 (\boldsymbol{w}_k - \boldsymbol{w}_{k-1}), \tag{4.4.3}$$

where $\beta_1 \in (0,1)$ is a constant parameter ($\beta_1 = 0$ would correspond to the classical stochastic gradient method). This method is a (stochastic) variant on Polyak's heavy-ball method, for which the gradient step is combined with the previous displacement. As in momentum-based methods, the idea consists in accumulating information from the previous iteration. In practice, the iteration (4.4.3) tends to accumulate good directions (in the optimization sense) while "bad" directions tend to cancel out.

The method (4.4.3) is a special case of (4.4.2), corresponding to $\boldsymbol{v}_k = \mathbf{1}_{\mathbb{R}^d}$ and $\boldsymbol{m}_k$ defined recursively by $\boldsymbol{m}_{-1} = \mathbf{0}_{\mathbb{R}^d}$ and

$$\boldsymbol{m}_k = (1-\beta_1)\boldsymbol{g}_k - \beta_1 \boldsymbol{m}_{k-1} \quad \forall k \in \mathbb{N}.$$

where β_1 is the constant defined in (4.4.3).

Stochastic gradient with momentum is implemented in standard deep learning librairies such as PyTorch. It is particularly useful in training deep neural netwroks on computer vision tasks, and, as such, played a role in the outbreak of deep learning circa 2012.

Remark 4.4.1 *It is less straightforward to derive theoretical guarantees for the method* (4.4.3) *than for accelerated gradient descent, even in a strongly convex setting. Nevertheless, adding momentum to the stochastic gradient iteration is a popular practice in solving nonconvex problems such as those arising from training neural networks.*

4.4.2 AdaGrad

The adaptive gradient method, or AdaGrad, was proposed in 2011 to address the issue of selecting the learning rate α in stochastic gradient without relying on adaptive approaches like line searches. In AdaGrad, every component of the stochastic gradient is scaled according to a running average of the values taken by that component over all iterations. The method maintains a sequence $\{\boldsymbol{r}_k\}_k$ given by

$$\forall i = 1, \ldots, d, \quad \begin{cases} [\boldsymbol{r}_{-1}]_i = 0 \\ [\boldsymbol{r}_k]_i = [\boldsymbol{r}_{k-1}]_i + [\boldsymbol{g}_k]_i^2 \quad \forall k \geq 0, \end{cases} \tag{4.4.4}$$

The AdaGrad iteration is thus

$$\boldsymbol{w}_{k+1} = \boldsymbol{w}_k - \alpha \boldsymbol{g}_k \oslash \sqrt{\boldsymbol{r}_k}, \tag{4.4.5}$$

where the square root is applied to every component of $\boldsymbol{r}_k$. This iteration matches (4.4.2) with $\boldsymbol{m}_k = \boldsymbol{g}_k$ and $\boldsymbol{v}_k = \sqrt{\boldsymbol{r}_k}$. The contribution of AdaGrad thus consists in using a different stepsize for each coordinate, leading the sequence :

$$\left\{ \left[\frac{\alpha}{\sqrt{[\boldsymbol{r}_k]_i}} \right]_{i=1}^{d} \right\}_k .$$

The method performs a *diagonal scaling* of the components of the stochastic gradient $\boldsymbol{g}_k$, which is particularly well suited for ill-conditioned problems where the components have a high variance. However, such stepsizes typically decrease very quickly towards 0.

Remark 4.4.2 *In practice, we replace* $\boldsymbol{r}_k$ *by* $\boldsymbol{r}_k + \eta \mathbf{1}_{\mathbb{R}^d}$ *where* $\eta > 0$ *is a small quantity, so that the algorithm is numerically stable.*

AdaGrad is particularly suited for problems with *sparse gradients*, for which stochastic graidents also tend to have many zero components. In this situation, computing $\boldsymbol{r}_k$ will only change the stepsize for the nonzero coordinates. Problems from recommender systems typically come with sparse gradients, which explains the popularity of AdaGrad in this setting.

4.4.3 RMSProp

The Root Mean Square Propagation algorithm, or RMSProp, is similar to AdaGrad in that it scales the stochastic gradient components. To this end, the method computes a vector sequence $\{\boldsymbol{r}_k\}_k$ as follows:

$$\forall i = 1, \ldots, d, \quad \begin{cases} [\boldsymbol{r}_{-1}]_i = 0 \\ [\boldsymbol{r}_k]_i = (1-\lambda)[\boldsymbol{r}_{k-1}]_i + \lambda[\boldsymbol{g}_k]_i^2 \quad \forall k \geq 0, \end{cases} \tag{4.4.6}$$

where $\lambda \in (0,1)$. The value of λ controls how much weight is given to the past stochastic gradient components over the current stochastic gradient components. This idea leads to a slower decrease in the stepsizes compared to the values of AdaGrad.

As for AdaGrad, the iteration of RMSProp corresponds to a special case of (4.4.2) using $\boldsymbol{m}_k = \boldsymbol{g}_k$ and $\boldsymbol{v}_k = \sqrt{\boldsymbol{r}_k}$.

Remark 4.4.3 *In practice, and as in* AdaGrad, *the vector $\boldsymbol{r}_k$ is replaced by $\boldsymbol{r}_k + \eta \mathbf{1}_{\mathbb{R}^d}$ for a small value $\eta > 0$.*

The RMSProp algorithm has been successfully applied to training very deep neural networks.

4.4.4 Adam

The Adam algorithm[1] was proposed in 2013, and has been one of the most popular stochastic gradient technique in pratice. This method can be viewed as combining the idea of momentum together with scaling: scaling will be performed according to the past gradients, and the search direction will also include pas gradient information. The Adam iteration corresponds to applying (4.4.2) with

$$\boldsymbol{m}_k = \frac{(1-\beta_1)\sum_{j=0}^{k} \beta_1^{k-j} \boldsymbol{g}_j}{1-\beta_1^{k+1}} \tag{4.4.7}$$

for $\beta_1 \in (0,1)$. This is indeed a momentum-type iteration, since we can obtain $\boldsymbol{m}_k$ from $\boldsymbol{m}_{k-1}$ and $\boldsymbol{g}_k$ through the formula

$$\boldsymbol{m}_k = \beta_1 \frac{1-\beta_1^k}{1-\beta_1^{k+1}} \boldsymbol{m}_{k-1} + \frac{1-\beta_1}{1-\beta_1^{k+1}} \boldsymbol{g}_k.$$

The other component of the Adam update is given by

$$\boldsymbol{v}_k = \sqrt{\frac{(1-\beta_2)\sum_{j=0}^{k} \beta_2^{k-j} \boldsymbol{g}_j \odot \boldsymbol{g}_j}{1-\beta_2^{k+1}}}. \tag{4.4.8}$$

where $\beta_2 \in (0,1)$ and $\odot$ denoting the componentwise or Hadamard product given by

$$\boldsymbol{g}_k \odot \boldsymbol{g}_k = \left[[\boldsymbol{g}_k]_i^2 \right]_{i=1}^d.$$

Remark 4.4.4 *In practice, a vector of the form $\boldsymbol{v}_k + \eta \mathbf{1}_{\mathbb{R}^d}$ will be used in lieu of $\boldsymbol{v}_k$, with η being a small positive number.*

The above formulae amount to combining previously employed directions with the latest stochastic gradient vector, and normalizing the components of the obtained vector according to the history of these components. In both cases, more importance is given to the latest values that have been computed. This is a key feature of the method, that has statistical motivations, and may explain the impressive performance of Adam. In practice, Adam (and its variant AdamW based on regularization) are among the most efficient methods for training architectures on Natural Language Processing tasks.

[1] The name Adam is derived from ADAptive Momentum estimation.

4.5 Conclusion

From a pure optimization perspective, stochastic gradient methods may not seem so attractive, as they only rely on partial information from the gradient and possess worse convergence guarantees than gradient descent. However, they have encountered tremendous success in data-related applications, where computing gradients involves looking at the entire data and is thus too prohibitive. On the contrary, using stochastic gradient estimates represents a significantly cheaper cost per iteration; in a data science setting, where there can be redundancies (or even underlying randomness) in the data, such updates do not necessarily hinder the progress of the algorithm, but rather lead to faster convergence in practice.

Still, the stochastic gradient approach suffers from high-variance estimates. For this reason, practical variants typically incorporate enhancements to reduce the variance. The most prominent technique for finite-sum and stochastic problems consist in using a batch of samples, which provably reduces the variance and can improve the performance. Meanwhile, the most efficient stochastic gradient techniques, such as those used in deep learning, employ both momentum terms and diagonal scaling to improve the quality of the steps. These techniques may not be endowed with better (if any) theoretical guarantees, especially when applied to nonconvex training problems. However, methods such as Stochastic Gradient with Momentum or ADAM have been widely adopted by the learning community because of their practical efficiency.

Chapter 5

Second-order methods

5.1 Motivation

In nonlinear optimization, it is known that exploiting second-order information can enhance the performance of first-order algorithms. Indeed, first-order methods such as gradient descent can be quite sensitive to the conditioning of the optimization problem, because such methods are not *scale invariant*.

5.1.1 Ill-conditioning

Consider the problem $\min_{\boldsymbol{w}\in\mathbb{R}^d} f(\boldsymbol{w})$, where f is a continuously differentiable function, and a scaled version of the problem, $\min_{\tilde{\boldsymbol{w}}\in\mathbb{R}^d} f(\boldsymbol{A}\tilde{\boldsymbol{w}})$, where $\boldsymbol{A} \in \mathbb{R}^{d\times d}$ is a linear invertible transformation of the input (typical of neural networks). The k-th iteration of gradient descent for the first problem is

$$\boldsymbol{w}_{k+1} = \boldsymbol{w}_k - \alpha_k \nabla f(\boldsymbol{w}_k),$$

while for the second problem, it is

$$\tilde{\boldsymbol{w}}_{k+1} = \tilde{\boldsymbol{w}}_k - \alpha_k \boldsymbol{A}\nabla f(\boldsymbol{A}\tilde{\boldsymbol{w}}_k).$$

If we set $\boldsymbol{w}_k = \boldsymbol{A}\tilde{\boldsymbol{w}}_k$ in order to obtain equivalent solutions, the second iteration becomes

$$\boldsymbol{w}_{k+1} = \boldsymbol{w}_k - \alpha_k \boldsymbol{A}^2\nabla f(\boldsymbol{w}_k),$$

which can have quite a different behavior than the original iteration, depending on the properties of the matrix $\boldsymbol{A}$.

Second-order methods, on the other hand, can be designed so as to be insensitive to such changes: Newton's method is for instance invariant to linear transformations. This is one of the reasons for which the optimization community started to focus on second-order methods in the 80s.

5.1.2 Drawbacks of second-order methods

The cost of second-order methods has always been raised as an issue with these schemes. Indeed, a basic second-order method would require the computation of the Hessian matrix at the current point, along with the manipulation of that matrix in certain ways (solving a linear system, computing eigenvalues, etc). In the context of machine learning, this can be prohibitive due to the cost of

accessing the data necessary for these calculations. In Section 5.4, we will describe several approaches that attempt to use second-order information at a reasonable cost.

If the problem at hand is not sufficiently smooth, it may also seem impossible to use second-order information. Like with first-order methods, equivalents of the second-order derivatives can be defined in multiple contexts, that allow for generalizing many second-order algorithms to this setting. This is particularly important in variational analysis, but is, however, out of the scope of this lecture.

5.2 Newton's method

We begin by presenting the most classical second-order algorithm for optimization, Newton's method. In order to provide justification for Newton's method in optimization, we first recall its motivation in the context of nonlinear equations.

5.2.1 Newton's method for nonlinear equations

This method was originally designed to solve systems of nonlinear equations. Considering the equation

$$\boldsymbol{\phi}(\boldsymbol{s}) = 0, \qquad \boldsymbol{\phi} : \mathbb{R}^n \to \mathbb{R}^n \tag{5.2.1}$$

and assuming that $\boldsymbol{\phi}$ is continuous, Newton's method attempts to find a solution of (5.2.1) through the iteration $\boldsymbol{s}_{k+1} = \boldsymbol{s}_k - \boldsymbol{J}(\boldsymbol{s}_k)^{-1}\boldsymbol{\phi}(\boldsymbol{s}_k)$, where $\boldsymbol{J}(\boldsymbol{s})$ denotes the Jacobian matrix of $\boldsymbol{\phi}$ at $\boldsymbol{s}$, i.e. $\boldsymbol{J}(\boldsymbol{s}) = \left[\frac{\partial \boldsymbol{\phi}}{\partial \boldsymbol{w}_j}(\boldsymbol{w})\right]_{\substack{i=1,\dots,d \\ j=1,\dots,d}}$. Note that this iteration assumes that the Jacobian can be inverted. Note also that no stepsize is used in the iteration. Rather, the iteration is built so that the inversion of the Jacobian matrix provides a scaling of the components of $\boldsymbol{\phi}$ according to the derivative information.

5.2.2 Newton's method in nonlinear optimization

In nonlinear optimization, we consider $\min_{\boldsymbol{w}\in\mathbb{R}^d} f(\boldsymbol{w})$ where $f \in \mathcal{C}^2$. In that context, Newton's method corresponds to applying Newton's method for nonlinear equations to the system $\nabla f(\boldsymbol{w}_k) = 0$. The iteration of Newton's method thus is

$$\boldsymbol{w}_{k+1} = \boldsymbol{w}_k - \left[\nabla^2 f(\boldsymbol{w}_k)\right]^{-1} \nabla f(\boldsymbol{w}_k), \tag{5.2.2}$$

whenever the Hessian matrix of f at $\boldsymbol{w}_k$ is not singular. This is for instance the case when f is a strongly convex function.

Two equivalent formulations of Newton's method are of importance. The first one does not explicitly use the inverse of the Hessian matrix (which may not exist if the function is not strongly convex):

$$\boldsymbol{w}_{k+1} = \boldsymbol{w}_k + \boldsymbol{s}_k, \quad \nabla^2 f(\boldsymbol{w}_k)\boldsymbol{s}_k = -\nabla f(\boldsymbol{w}_k). \tag{5.2.3}$$

Note that the linear system $\nabla^2 f(\boldsymbol{w}_k)\boldsymbol{s}_k = -\nabla f(\boldsymbol{w}_k)$ is commonly called the *Newton system* or the *Newton equations*. Solutions of this system can exist even when the Hessian matrix is indefinite.

The second one formulates the Newton step as a solution of a quadratic problem:

$$\boldsymbol{w}_{k+1} = \arg\min_{\boldsymbol{s}\in\mathbb{R}^d} f(\boldsymbol{w}_k) + \nabla f(\boldsymbol{w}_k)^{\mathrm{T}}\boldsymbol{s} + \tfrac{1}{2}\boldsymbol{s}^{\mathrm{T}}\nabla^2 f(\boldsymbol{w}_k)\boldsymbol{s}. \tag{5.2.4}$$

This problem may not have a solution when f is not convex.

5.2.3 Local convergence of Newton's method

One of the main characteristics of Newton's method is its fast local convergence rate guarantees. When f is a strongly convex quadratic, Newton's method converges in one iteration. If f is strongly convex, it can be shown that Newton's method converges at a *local quadratic rate*, i.e. if $\boldsymbol{w}_0$ is close enough to the global minimum $\boldsymbol{w}^*$ of f, we have:

$$\|\boldsymbol{w}_{k+1} - \boldsymbol{w}^*\| \leq C \|\boldsymbol{w}_k - \boldsymbol{w}^*\|^2,$$

where $C > 0$. This reflects on the practical performance of Newton's method, which commonly takes only a few iterations to converge on such functions.

Note that similar rates can also be obtained for convex or nonconvex functions, provided the function is strongly convex in a neighborhood of a local minimum: this illustrates that this result is essentially local (and shows the importance of a good initialization).

5.3 Globalization techniques

As described in the previous section, Newton's method possesses local convergence guarantees. It can be shown that those guarantees are not global, in the sense that for general nonconvex functions (or even convex functions that are not strongly convex), the choice of the initial point determines whether or not the method will converge, or be well-defined. In practice, finite-precision arithmetic can even prevent Newton from converging on strongly convex functions.

Globalization techniques were developed to guarantee that Newton's method would converge independently of its starting point, and regardless of the convexity of the problem. They have proven to be quite useful in multiple settings, even though they rely on seemingly costly operations at every iteration (compute minimum eigenvalue, solve a minimization problem, etc). We present thereafter the three main approaches for globally convergent Newton methods, then discuss their theoretical properties.

5.3.1 Line search

Line-search algorithms proceed by first computing a suitable direction of decrease for the objective function, then by selecting an appropriate stepsize along this direction. Line-search Newton methods construct this direction by modifying the Newton system (5.2.3) so that it has a solution.

A natural idea to guarantee that a step can be uniquely obtained from the Newton equations consists in regularizing the Hessian so that it becomes positive definite. At every iteration k, one can indeed select a nonnegative value λ_k such that $\nabla^2 f(\boldsymbol{w}_k) + \lambda_k \boldsymbol{I} \succ 0$ and compute a (regularized) Newton step as $\boldsymbol{s}_k = -[\nabla^2 f(\boldsymbol{w}_k) + \lambda_k \boldsymbol{I}]^{-1} \nabla f(\boldsymbol{w}_k)$.

Once the direction has been set, a line-search process is executed to compute the most appropriate steplength along that direction. In the case of Newton-type methods, it is usually a good idea to try the unit step first (especially if $\lambda_k = 0$). A classical process, called *backtracking*, tries geometrically decreasing values (e.g. $1, 1/2, 1/4, \dots$) until finding a value that sufficiently decreases the objective. Algorithmically, this means that given $\boldsymbol{s}_k$, the method determines $\alpha_k > 0$ such that (at least) $f(\boldsymbol{w}_k + \alpha_k \boldsymbol{s}_k) < f(\boldsymbol{w}_k)$, then set $\boldsymbol{w}_{k+1} = \boldsymbol{w}_k + \alpha_k \boldsymbol{s}_k$.

With more precise definitions, it is possible to show convergence of this framework regardless of the starting point. Under some additional conditions, it is also possible to obtain local convergence results, that are usually worse than Newton's method because of the use of λ_k. Still, line-search

methods are widely used in nonlinear optimization, and Newton-type directions turn out to be less sensitive to the choice of stepsize than gradient-based methods.

5.3.2 Trust region

In the optimization community, trust-region methods are often preferred to line-search methods for handling nonconvex problems. Rather than regularizing the Newton system, those techniques modify the subproblem formulation of Newton's method (5.2.4). At every iteration, the step $\boldsymbol{s}_k$ is thus obtained as the solution of a *trust-region subproblem*:

$$\boldsymbol{s}_k = \arg\min_{\boldsymbol{s}\in\mathbb{R}^d} \nabla f(\boldsymbol{w}_k)^{\mathrm{T}} s + \frac{1}{2}\boldsymbol{s}^{\mathrm{T}}\nabla^2 f(\boldsymbol{w}_k)\boldsymbol{s} \quad \text{subject to} \quad \|\boldsymbol{s}\| \leq \delta_k. \tag{5.3.1}$$

The parameter δ_k is called the trust-region radius, and prevents the solution to go to infinity if the Hessian of f at $\boldsymbol{w}_k$ is indefinite. Depending on whether or not $\boldsymbol{s}_k$ leads to a decrease in the function value, the step is accepted or rejected, and δ_k is increased or decreased.

Trust-region algorithms have been endowed with similar global convergence guarantees than line-search methods and can even have faster local convergence rates, if the management of the trust-region radius allows for taking Newton steps close to a minimum.

5.3.3 Cubic regularization

This more recent technique has gained popularity because of its attractive complexity properties. It proceeds similarly to the trust-region paradigm, by choosing $\boldsymbol{s}_k$ as the solution to the subproblem

$$\boldsymbol{s}_k = \arg\min_{\boldsymbol{s}\in\mathbb{R}^d} \nabla f(\boldsymbol{w}_k)^{\mathrm{T}} s + \frac{1}{2}\boldsymbol{s}^{\mathrm{T}}\nabla^2 f(\boldsymbol{w}_k)\boldsymbol{s} + \frac{\sigma_k}{3}\|\boldsymbol{s}\|^3. \tag{5.3.2}$$

The parameter $\sigma_k > 0$ ensures that this subproblem always has a finite solution: this value of σ_k is updated in an inverse way compared to the trust-region radius. Note that cubic regularization can be seen as an implicit regularization of the Hessian, that is not known before $\boldsymbol{s}_k$ is computed. On the contrary, the regularizing parameter in the line-search framework is known before computing the step.

5.3.4 Convergence and complexity

Under appropriate assumptions, these three methods can be shown to converge to a stationary point in the nonconvex and convex cases. In a convex setting (and even more so in a strongly convex setting), local convergence results are usually of interest, and Newton-type methods can be designed to take advantage of Newton steps for fast local rates.

In the nonconvex case, recent research has focused on bounding the number of iterations required to satisfy $\|\nabla f(\boldsymbol{w}_k)\| \leq \epsilon$ for $\epsilon \in (0, 1)$. In their standard versions, the line-search and trust-region variants require at most $\mathcal{O}(\epsilon^{-2})$ iterations to reach such a point, while cubic regularization requires at most $\mathcal{O}(\epsilon^{-3/2})$ iterations (this is the optimal bound for Newton-type methods). This has motivated a significant amount of research on cubic regularization techniques, even though those turned out to be less efficient in practice than trust-region methods (in part because solving cubic subproblems can be harder than solving trust-region subproblems).

5.4 Practical second-order methods

In their most basic form, Newton's method and its variants assume that the exact Hessian matrix is available, which is not reasonable in a number of settings (including machine learning, but also many physical-based applications like control). For this reason, inexact variants of Newton's method have been developed, that do not require full storage or computation of the Hessian.

5.4.1 Hessian-free inexact Newton methods

One approach that has become quite popular in large-scale optimization is to compute an *inexact Newton step* by approximately solving the Newton equations at every iteration. That is, at iteration k, the step s_k is obtained such that

$$\|\nabla^2 f(\boldsymbol{w}_k)\boldsymbol{s}_k + \nabla f(\boldsymbol{w}_k)\| \leq \eta_k, \tag{5.4.1}$$

where $\eta_k > 0$ (with $\eta_k = 0$, we would recover the exact Newton iteration). One advantage of using (5.4.1) is that it can be satisfied even when the linear system does not have a solution. Moreover, it allows for applying iterative linear algebra techniques to the linear system, that are *matrix-free* in nature: they do not require access to the full matrix, only to *products of this matrix with a vector*. One example of such an algorithm is the (linear) conjugate gradient method, that is guaranteed to solve a positive definite linear system of dimension d in d iterations in exact arithmetic. In practice, such a method is applied for a certain number of iterations, and stops when (5.4.1) is satisfied or this budget is exhausted.

Computing Hessian-vector products With Hessian-free techniques, all that is required is the computation of $\nabla^2 f(\boldsymbol{w}_k)\boldsymbol{v}$ for any $\boldsymbol{v} \in \mathbb{R}^d$. One possibility to obtain these quantities without access to the Hessian is to use finite-difference estimators based on several gradient evaluations. If this is still too expensive, automatic differentiation techniques might be useful. If a code for computing the numerical value of a gradient is available,automatic differentiation obtains numerical values of Hessian-vector products at a cost that is only more expensive than a gradient evaluation by a constant factor. Automatic differentiation is used in machine learning packages such as `PyTorch` for computing gradients, and can also be employed to calculate Hessian-vector products.

5.4.2 Subsampling Hessian-free methods

Consider the following finite-sum optimization problem:[1]

$$\min_{\boldsymbol{w}\in\mathbb{R}^d} f(\boldsymbol{w}) := \frac{1}{n}\sum_{i=1}^{n} f_i(\boldsymbol{w}). \tag{5.4.2}$$

In data-driven applications, it is quite common to encounter problems of this form, for which each f_i depends on one data point. As a result, evaluating the entire function f (or its derivatives) is extremely costly, and practitioners rely on subsampling techniques, that draw a generally small subset of the data points on which derivatives are calculated.

[1] A similar analysis holds for general stochastic optimization problems of the form $f(\boldsymbol{w}) = \mathbb{E}_\xi\left[f(\boldsymbol{w};\xi)\right]$.

This idea can be applied to inexact Newton methods as well. At every iteration k of the algorithm, given the current iterate $\boldsymbol{w}_k$, one computes subsampling derivatives as

$$\nabla f_{\mathcal{S}_k}(\boldsymbol{w}_k) = \frac{1}{|\mathcal{S}_k|}\sum_{i\in\mathcal{S}_k}\nabla f_i(\boldsymbol{w}_k), \quad \nabla^2 f_{\mathcal{S}_k^H}(\boldsymbol{w}_k) = \frac{1}{|\mathcal{S}_k^H|}\sum_{i\in\mathcal{S}_k^H}\nabla^2 f_i(\boldsymbol{w}_k), \tag{5.4.3}$$

where $\mathcal{S}_k$ and $\mathcal{S}_k^H$ are drawn randomly in $\{1,\dots,n\}$. One can then apply an inexact Newton step approach to the system defined by these derivatives. For instance, a Newton-conjugate gradient method would apply the linear conjugate gradient algorithm to the linear system $\nabla^2 f_{\mathcal{S}_k^H}(\boldsymbol{w}_k)s = -\nabla f_{\mathcal{S}_k}(\boldsymbol{w}_k)$ until a budget of conjugate gradient iterations has been exhausted, or a vector s is found such that

$$\|\nabla^2 f_{\mathcal{S}_k^H}(\boldsymbol{w}_k)s_k + \nabla f_{\mathcal{S}_k}(\boldsymbol{w}_k)\| \leq \rho\|\nabla f_{\mathcal{S}_k}(\boldsymbol{w}_k)\| \tag{5.4.4}$$

is satisfied. As for the classical Newton-CG approach, globalization techniques would then be used to guarantee convergence even for nonconvex functions. In order for such guarantees to hold, the sample sizes $|\mathcal{S}_k|$ and $|\mathcal{S}_k^H|$ must be sufficiently large, which contrasts with the typical choices adopted in practice. In particular, if $|\mathcal{S}_k|$ is not large enough, it is likely that the noise within the gradient estimation will be amplified by the Newton step, and that the resulting step will not be informative; still, small sample sizes can lead to good practical performance.

Computing subsampled Hessian-vector products In addition to the aforementioned techniques, the particular form of the objective function can be used to compute Hessian-vector products efficiently. Considering for instance a logistic loss objective of the form

$$f(\boldsymbol{w}) = \frac{1}{n}\sum_{i=1}^{n}\log\left(1+\exp(-y_i\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}_i)\right),$$

we obtain the following formula for subsampled Hessian-vector products:

$$\nabla^2 f_{\mathcal{S}_k^H}(\boldsymbol{w}_k)\boldsymbol{d} = \frac{1}{|\mathcal{S}_k^H|}\sum_{i\in\mathcal{S}_k^H}\frac{\exp(-y_i\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}_i)}{(1+\exp(-y_i\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}_i))^2}(\boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{d})\boldsymbol{x}_i.$$

The field of subsampling Newton methods is an active area of research, particularly in the optimization community, but reconciling theory and practice remains an open question.

5.4.3 Quasi-Newton methods

Along with inexact Newton techniques, *quasi-Newton schemes* have been quite successful in large-scale optimization problems (and even when second-order derivatives do not exist!). Because of the way they approximate second-order information without computing Hessian values, they have also been favored from a practical viewpoint.

For the problem $\min_{\boldsymbol{w}\in\mathbb{R}^d} f(\boldsymbol{w})$, the k-th iteration of a quasi-Newton method typically has the following form:

$$\boldsymbol{w}_{k+1} \;=\; \boldsymbol{w}_k - \alpha_k \boldsymbol{H}_k\nabla f(\boldsymbol{w}_k), \tag{5.4.5}$$

where $\boldsymbol{H}_k$ is a symmetric, positive-definite matrix such that $\boldsymbol{H}_k^{-1} \approx \nabla^2 f(\boldsymbol{w}_k)$. The quasi-Newton matrix $\boldsymbol{H}_k$ is updated dynamically at every iteration. Several formulas for such an update have

been proposed in the literature, such as the DFP (Davidon[2]-Fletcher-Powell) and BFGS (Broyden-Fletcher-Goldfarb-Shanno), the latter of which is detailed below.

BFGS Quasi-Newton Start with $\boldsymbol{w}_0$ and $\boldsymbol{H}_0 = \boldsymbol{I}$ (the identity matrix). At every iteration k, compute $\boldsymbol{w}_{k+1}$ according to (5.4.5), then define $\boldsymbol{s}_k = \boldsymbol{w}_{k+1} - \boldsymbol{w}_k$ and $\boldsymbol{v}_k = \nabla f(\boldsymbol{w}_{k+1}) - \nabla f(\boldsymbol{w}_k)$. Finally, update the quasi-Newton matrix as follows:

$$\boldsymbol{H}_{k+1} = \left(\boldsymbol{I} - \frac{\boldsymbol{v}_k \boldsymbol{s}_k^{\mathrm{T}}}{\boldsymbol{s}_k^{\mathrm{T}} \boldsymbol{v}_k}\right)^{\mathrm{T}} \boldsymbol{H}_k \left(\boldsymbol{I} - \frac{\boldsymbol{v}_k \boldsymbol{s}_k^{\mathrm{T}}}{\boldsymbol{s}_k^{\mathrm{T}} \boldsymbol{v}_k}\right) + \frac{\boldsymbol{s}_k \boldsymbol{s}_k^{\mathrm{T}}}{\boldsymbol{s}_k^{\mathrm{T}} \boldsymbol{v}_k}. \tag{5.4.6}$$

Such an update guarantees that $\boldsymbol{H}_{k+1}^{-1} \boldsymbol{s}_k = \boldsymbol{v}_k$: that is, the new Hessian formula is consistent with the most recent displacement (such a relation is often called a *secant equation*).

In the case of a strongly convex function f, it can be shown that quasi-Newton methods such as BFGS possess a *local superlinear rate*. That is, for $\boldsymbol{w}_0$ sufficiently close to the optimum $\boldsymbol{w}^*$, the iterates satisfy the relation

$$\|\boldsymbol{w}_{k+1} - \boldsymbol{w}^*\| \leq C \|\boldsymbol{w}_k - \boldsymbol{w}^*\|^{1+t},$$

where $t \in (0, 1)$, $C > 0$ and $\boldsymbol{w}^*$ is the global minimum of the function.

L-BFGS A quasi-Newton matrix is likely to become dense as the iteration unfolds, and this can be an issue in a large-dimensional setting. The *limited-memory* variant of the BFGS Quasi-Newton update removes the need for storing a matrix $\boldsymbol{H}_k$, by considering the BFGS updates (5.4.6) in a recursive fashion. Indeed, the formula for $\boldsymbol{H}_{k+1}$ only depends on $\boldsymbol{H}_0$ and the pairs $(\boldsymbol{s}_0, \boldsymbol{v}_0), (\boldsymbol{s}_1, \boldsymbol{v}_1), \ldots, (\boldsymbol{s}_k, \boldsymbol{v}_k)$. Instead of performing the $k+1$ updates, the limited-memory L-BFGS update only computes $\boldsymbol{H}_{k+1}$ based on the latest m pairs $\{(\boldsymbol{s}_{k-i}, \boldsymbol{v}_{k-i})\}_{i=0}^{\max\{0,m-1\}}$. This can be implemented very efficiently: the value $m = 5$ is commonly used in implementations, and is sufficient to observe significant improvement compared to a gradient descent method.

A remarkable property of quasi-Newton methods is that they were used for quite some time before a convergence proof became available, because they performed remarkably well in practice. This performance continues to be demonstrated with variants such as L-BFGS, that can be extremely efficient on data science problems arising from robust statistics and matrix approximation. It would seem that a full explanation for this performance, in particular regarding the limited memory variants, remains to be found.

For problems exhibiting a structure similar to (5.4.2), it is possible to implement *subsampling quasi-Newton methods*, that only depend on subsampled gradients. For this reason, these methods require smaller batch sizes that what is theoretically needed for Hessian-free Newton methods (typically subsampling L-BFGS with $m = 5$ would be more expensive than stochastic gradient by a factor of 20).

[2]William C. Davidon (1927-2013) is credited for the original idea behind quasi-Newton methods in the late 50s. As a physicist, he applied this technique because he could not afford to compute second-order derivatives in his application. His work was celebrated in 1991, when the first issue of *SIAM Journal on Optimization* published his original technical report, that was rejected by a journal at the time Davidon submitted it, and that he had given up on publishing. Davidon also has an interesting backstory, that can be found on his Wikipedia page.

5.4.4 Gauss-Newton methods

By using the structure of a given problem, it is generally possible to use more information. This is a general optimization principle that turns out to be quite important for machine learning (the most prominent example being exploiting finite-sum structures). Here we present a way to extract second-order information from gradient information when the optimization problem possesses a particular formulation.

Consider the problem $\min_{\boldsymbol{w}\in\mathbb{R}^d} f(\boldsymbol{w})$, where f is a *least-squares objective* of the form

$$f(\boldsymbol{w}) = \frac{1}{2}\|\boldsymbol{\phi}(\boldsymbol{w})\|^2,$$

with $\boldsymbol{\phi} = [\phi_j]_{j=1}^m \in \mathcal{C}^2(\mathbb{R}^d, \mathbb{R}^m)$. The gradient and Hessian of f are given by the following formula:

$$\nabla f(\boldsymbol{w}) = \boldsymbol{J}_{\boldsymbol{\phi}}(\boldsymbol{w})^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{w}), \qquad \nabla^2 f(\boldsymbol{w}) = \boldsymbol{J}_{\boldsymbol{\phi}}(\boldsymbol{w})^{\mathrm{T}}\boldsymbol{J}_{\boldsymbol{\phi}}(\boldsymbol{w}) + \sum_{j=1}^m \phi_j(\boldsymbol{w})\nabla^2\phi_j(\boldsymbol{w}), \tag{5.4.7}$$

where $\boldsymbol{J}_{\boldsymbol{\phi}}(\boldsymbol{w}) = \begin{bmatrix} \nabla\phi_1(\boldsymbol{w})^{\mathrm{T}} \\ \cdots \\ \nabla\phi_m(\boldsymbol{w})^{\mathrm{T}} \end{bmatrix}$ is the Jacobian matrix of $\boldsymbol{\phi}$ at $\boldsymbol{w}$. Because this Jacobian is needed for computing ∇f, the first term of $\nabla^2 f$ is available "for free" once the gradient has been computed. In addition, if the solution of the problem $\boldsymbol{w}^*$ is such that $\boldsymbol{\phi}(\boldsymbol{w}^*) = \boldsymbol{0}$, the second term in the formula of $\nabla^2 f$ can be neglected around the solution. This is the underlying idea of Gauss-Newton methods, that rely on the following iteration:

$$\boldsymbol{w}_{k+1} = \boldsymbol{w}_k + \boldsymbol{s}_k, \qquad \boldsymbol{J}_{\boldsymbol{\phi}}(\boldsymbol{w}_k)^{\mathrm{T}}\boldsymbol{J}_{\boldsymbol{\phi}}(\boldsymbol{w}_k)\boldsymbol{s}_k = \boldsymbol{J}_{\boldsymbol{\phi}}(\boldsymbol{w}_k)\boldsymbol{\phi}(\boldsymbol{w}_k). \tag{5.4.8}$$

or a globalized variant thereof. The matrix $\boldsymbol{J}_{\boldsymbol{\phi}}(\boldsymbol{w}_k)^{\mathrm{T}}\boldsymbol{J}_{\boldsymbol{\phi}}(\boldsymbol{w}_k)$ is called the *Gauss-Newton approximation* to the Hessian matrix. Although Gauss-Newton techniques lack the fast convergence rate guarantees of Newton variants near an optimum, they can lead to better steps away from the solution.

Gauss-Newton techniques have been applied to finite-sum problems with differentiable losses, of the following form:

$$\min_{\boldsymbol{w}\in\mathbb{R}^d} f(\boldsymbol{w}) := \frac{1}{n}\sum_{i=1}^n \ell(\boldsymbol{h}(\boldsymbol{x}_i;\boldsymbol{w}), y_i)), \tag{5.4.9}$$

where $\boldsymbol{h} : \mathbb{R}^d \to \mathbb{R}^d$ and $\ell : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}^d$ are both twice continously differentiable. In that case, one usually selects a batch $\mathcal{S}_k$ for the gradient and a batch $\mathcal{S}_k^H$ for the Gauss-Newton approximation of the Hessian: the latter is given by

$$\frac{1}{|\mathcal{S}_k^H|}\sum_{i\in\mathcal{S}_k^H} \boldsymbol{J}_{\boldsymbol{h}}(\boldsymbol{x}_i;\boldsymbol{w})^{\mathrm{T}}\frac{\partial^2\ell(\boldsymbol{h}(\boldsymbol{x}_i;\boldsymbol{w}_k), y_i)}{\partial\boldsymbol{w}^2}\boldsymbol{J}_{\boldsymbol{h}}(\boldsymbol{x}_i;\boldsymbol{w}).$$

5.4.5 Diagonal scaling

We mention in passing another kind of second-order methods, that resembles the quasi-Newton idea. The iteration takes the same form as (5.4.5) with a diagonal matrix $\boldsymbol{H}_k$: a different scaling is thys applied to every component of the gradient vector. In the optimization community, one of the most classical methods of this form is the *Barzilai-Borwein* method. In machine learning applications,

several approaches based on using the diagonal of a (subsampled) Gauss-Newton-type matrix have also been proposed. The idea is connected with *batch normalization*, a procedure that is often inserted between layers of a neural network.

Beyond second-order considerations, most successful optimization schemes in deep learning involve some form of diagonal scaling. We highlight three popular routines below.

RMSProp The *Root Mean Square Propagation* algorithm, or RMSProp, maintains an average of the magnitude of every component of the (stochastic) gradient through the following recursion:

$$\forall i = 1, \ldots, d, \quad \left\{ \begin{array}{l} [R_{-1}]_i = 0 \\ [R_k]_i = (1-\lambda)[R_{k-1}]_i + \lambda [g_k]_i^2 \quad \forall k \geq 0, \end{array} \right. \tag{5.4.10}$$

where g_k denotes the stochastic gradient vector used in the stochastic gradient step and $\lambda \in (0,1)$. Assuming that a constant stepsize/learning rate $\alpha > 0$ is used, the update of RMSProp is defined componentwise by

$$\forall i = 1, \ldots, d, \quad [\boldsymbol{w}_{k+1}]_i \; = \; [\boldsymbol{w}_k]_i - \frac{\alpha}{\sqrt{[R_k]_i + \mu}} [g_k]_i \tag{5.4.11}$$

The constant $\mu > 0$ serves as a further regularization parameter. This approach has been observed to be quite efficient for optimizing deep neural networks.

Adagrad The *Adaptive gradient* method, or Adagrad, follows a similar approach than RMSProp but simply uses the sum formula

$$\forall i = 1, \ldots, d, \quad \left\{ \begin{array}{l} [R_{-1}]_i = 0 \\ [R_k]_i = [R_{k-1}]_i + [g_k]_i^2 \quad \forall k \geq 0, \end{array} \right. \tag{5.4.12}$$

The stepsize sequence $\left\{ \left[\frac{\alpha}{\sqrt{[R_k]_i + \mu}} \right]_{i=1}^{d} \right\}_k$ is thus decreasing for each component of the parameter vector. This method has also encountered success in deep neural networks, particularly on problems that exhibit sparse gradients, a common feature in computer vision and natural language processing.

Adam One of the most common methods to train neural networks is the Adam optimizer proposed in 2013. It was described as combining the features of both RMSProp and Adagrad, through the estimation of first and second moments of the stochastic gradients. The Adam framework also relies on momentum, which makes it a particularly popular algorithm to train deep neural networks (and the default method in numerous architectures). Interestingly, failures in its original convergence analysis were pointed out in 2018, while the method was already being widely used. Its performance overcame its lack of strong theoretical results, particularly in the nonconvex setting. Developing a complete understanding of this method (as well as other stochastic gradient techniques based on momentum) is an active area of research, as theory remains decorrelated from practice.

5.5 Conclusion

Second-order methods have been widely used in numerical analysis (PDEs, optimization, control) on very large-scale instances, but are not as common in data-driven applications. Nevertheless, there is an undeniable practical value in incorporating second-order aspects, not necessarily at the cost of

evaluating second-order derivatives, and this trend of research is still being explored in a variety of data science problems.

Checkpoint questions

1. What is the main computational issue with using second-order methods?
2. How can this cost be mitigating on finite-sum problems?
3. What technique to incorporate curvature is at the heart of modern implementations of stochastic gradient?

Bibliography

[1] L. Bottou, F. E. Curtis, and J. Nocedal. Optimization Methods for Large-Scale Machine Learning. *SIAM Rev.*, 60:223–311, 2018.

[2] S. Boyd and L. Vandenberghe. *Convex optimization*. Cambridge University Press, Cambridge, United Kingdom, 2004.

[3] J. Nocedal and S. J. Wright. *Numerical Optimization*. Springer Series in Operations Research and Financial Engineering. Springer-Verlag, New York, second edition, 2006.

[4] S. J. Wright and B. Recht. *Optimization for Data Analysis*. Cambridge University Press, 2022.