

# CONVEX OPTIMIZATION FOR MACHINE LEARNING

CHANGHO SUH

*Published, sold and distributed by:*

now Publishers Inc.  
PO Box 1024  
Hanover, MA 02339  
United States  
Tel. +1-781-985-4510  
[www.nowpublishers.com](http://www.nowpublishers.com)  
[sales@nowpublishers.com](mailto:sales@nowpublishers.com)

*Outside North America:*

now Publishers Inc.  
PO Box 179  
2600 AD Delft  
The Netherlands  
Tel. +31-6-51115274

ISBN: 978-1-63828-052-1

E-ISBN: 978-1-63828-053-8

DOI: 10.1561/9781638280538

Copyright © 2022 Changho Suh

Suggested citation: Changho Suh. (2022). *Convex Optimization for Machine Learning*. Boston–Delft: Now Publishers

The work will be available online open access and governed by the Creative Commons “Attribution-Non Commercial” License (CC BY-NC), according to <https://creativecommons.org/licenses/by-nc/4.0/>

We would like to express our sincere gratitude to Hyun Seung Suh, who has provided a flurry of constructive comments and feedback on the structure of the book, writing, and readability from a beginner’s viewpoint.

This work was supported by the 2019 Google Education Grant; and Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (2020-0-00626, Ensuring high AI learning performance with only a small amount of training data).

# Table of Contents

Preface	v
<b>Chapter 1 Convex Optimization Basics</b>	1
1.1 Overview of the Book .....	1
1.2 Definition of Convex Optimization .....	10
1.3 Tractability of Convex Optimization and Gradient Descent.....	18
Problem Set 1 .....	24
1.4 Linear Program (LP) .....	28
1.5 LP: Examples and Relaxation .....	35
1.6 LP: Algorithms .....	42
1.7 LP: CVXPY Implementation .....	50
Problem Set 2 .....	54
1.8 Least Squares (LS).....	58
1.9 LS: Test Error, Regularization and CVXPY Implementation .....	64
1.10 LS: Computed Tomography .....	72
Problem Set 3 .....	80
1.11 Quadratic Program .....	86
1.12 Second-order Cone Program .....	93
1.13 Semi-definite Program (SDP) .....	103
1.14 SDP Relaxation .....	109
Problem Set 4 .....	116

<b>Chapter 2 Duality</b>	124
2.1 Strong Duality.....	124
2.2 Interior Point Method.....	131
Problem Set 5 .....	140
2.3 Proof of Strong Duality Theorem (1/2) .....	143
2.4 Proof of Strong Duality Theorem (2/2) .....	150
Problem Set 6 .....	157
2.5 Weak Duality.....	162
2.6 Lagrange Relaxation for Boolean Problems .....	169
2.7 Lagrange Relaxation for the MAXCUT Problem .....	175
Problem Set 7 .....	182
<b>Chapter 3 Machine Learning Applications</b>	185
3.1 Supervised Learning and Optimization.....	185
3.2 Logistic Regression .....	193
3.3 Deep Learning I .....	201
3.4 Deep Learning II .....	210
3.5 Deep Learning: TensorFlow Implementation .....	221
Problem Set 8 .....	229
3.6 Unsupervised Learning: Generative Modeling.....	240
3.7 Generative Adversarial Networks (GANs) .....	246
3.8 GANs: TensorFlow Implementation .....	252
Problem Set 9 .....	262
3.9 Wasserstein GAN I .....	271
3.10 Wasserstein GAN II .....	278
3.11 Wasserstein GAN: TensorFlow Implementation .....	285
Problem Set 10 .....	295
3.12 Fair Machine Learning .....	299
3.13 A Fair Classifier and Its Connection to GANs .....	307
3.14 A Fair Classifier: TensorFlow Implementation .....	313
Problem Set 11 .....	323
<b>Appendix A Python Basics</b>	329
A.1 Jupyter Notebook .....	329
A.2 Basic Python Syntaxes .....	334
<b>Appendix B CVXPY Basics</b>	343
<b>Appendix C TensorFlow and Keras Basics</b>	348
<b>References .....</b>	357
<b>Index .....</b>	362
<b>About the Author .....</b>	370

# Preface

**Features of the book** The writing of this book was prompted by the huge surge of research activities in machine learning and deep learning, and the crucial roles of convex optimization in the spotlight fields. This forms the motivation of this book, enabling three key features.

The first feature of this book is the focus of optimization contents tailored for modern applications in machine learning and deep learning. Since the optimization field has contributed to many disciplines, ranging from statistics, dynamical systems & control, complexity theory to algorithms, there are numerous optimization books that have been published with great interest during the past decades. In addition, the optimization discipline has been applied to a widening array of contexts, including science, engineering, economics, finance and management. Hence, the books contain a broad spectrum of contents to cover many applications in various areas. On the other hand, this book focuses on a single yet prominent field: *machine learning*. Among the vast contents, we put a special emphasis on the concepts and theories concerning machine learning and deep learning. Moreover, we employ many toy examples to ease illustration of some important concepts. Examples include historical and canonical instances, as well as the ones that arise in machine learning.

Second, this book is written in a lecture style. A majority of optimization books deal with many mathematical concepts and theories together with numerous applications in a wide variety of domains. So the concepts and relevant theories are simply enumerated to present topics sequentially, following a dictionary style organization. While the dictionary style eases search for targeted materials, it often lacks coherent stories that may help motivate readers. This book aims at motivating readers who are interested in machine learning inspired by optimization fundamentals. So we intend to make an interesting storyline that well demonstrates the

role of fundamentals in the field. In order to establish a motivating storyline, this book adopts a lecture style organization. Each section serves as a note for a lecture spanning around 80 minutes, and an intimate connection is made across sections, centered around coherent themes and concepts. To make a smooth transition from one section to another, we feature two special paragraphs in each section: (i) the “recap” paragraph that summarizes what have been done so far, thereby motivating the contents in the current section; and (ii) the “look ahead” paragraph that introduces upcoming contents by connecting with the past materials.

The last feature of this book is the inclusion of many programming exercises via three prominent software languages: (i) Python; (ii) CVXPY; and (iii) TensorFlow. Since one of the key tools in convex optimization is an *algorithm* that requires computation on a computer, it is crucial to know how to implement algorithms using software tools. We employ Python as a major programming platform. To solve traditional convex optimization problems such as linear program, least squares, and semi-definite program, we utilize an easy-to-use and high-level language, CVXPY, running in Python. To implement machine learning and deep learning algorithms, we employ TensorFlow, one of the most popular deep learning frameworks. TensorFlow provides numerous powerful built-in functions that ease performing many important procedures in deep learning. One of the key benefits of TensorFlow is that it is fully integrated with Keras, the most high-level library with a focus on enabling fast user experimentation. Keras allows us to go from idea to implementation with very few steps.

**Structure of the book** This book is made up of course materials that we developed for the following two courses at KAIST: (i) EE523 Convex Optimization (offered in Spring 2019); and (ii) EE424 Introduction to Optimization (offered in Fall 2020 and 2021). It consists of three parts, each being comprised of many sections. Each section contains materials covered by a single lecture with the duration of approximately 80 minutes. Each problem set (which served as a homework in the courses) is included every three or four sections. The contents for the three parts are summarized as below.

- I. *Convex optimization basics (14 sections and 4 problem sets)*: A brief history of convex optimization; basic concepts on convex sets and convex functions, and the definition of convex optimization; gradient descent; linear program (LP), LP relaxation, least squares, quadratic program, second-order cone program, semi-definite program (SDP) and SDP relaxation; CVXPY implementation.
- II. *Duality (7 sections and 3 problem sets)*: The Lagrange function, the dual function and the dual problem; strong duality, KKT conditions, and the interior point method; weak duality and Lagrange relaxation.

- III. *Machine learning applications (14 sections and 4 problem sets):* Supervised learning and the role of optimization in logistic regression and deep learning; backpropagation and its Python implementation; unsupervised learning, Generative Adversarial Networks (GANs), Wasserstein GAN, and the role of LP and duality theories; fair machine learning and the role of the regularization technique and the KKT conditions; TensorFlow implementation of a deep learning classifier, GANs, Wasserstein GAN and a fair machine learning algorithm.

At the end, we offer three appendices for brief tutorials of the employed programming languages (Python, CVXPY and TensorFlow). We also provide a list of references that are related to the contents discussed in the book. But we do not explain details, since we do not aim to exhaust the immense research literature.

**How to use this book** This book is written as a textbook for a senior-level undergraduate course, yet it is also suitable for a first-year graduate course. The expected background is solid undergraduate courses in linear algebra and probability, together with basic familiarity with Python.

For students and interested readers, we provide some guidelines:

1. *Study one section per day and two sections per week:* Since each section is designed for a single lecture and two lectures are normal per week in a course offering, we recommend this way of reading.
2. *Go through all the contents in Parts I and II:* One of the most important concepts in optimization is *duality*. So if you are familiar with convex optimization basics, then it is okay to directly dive into Part II without exploring Part I. However, if it is not the case, we recommend you to go through all the contents in Parts I and II in a sequential manner. A motivating storyline is made across sections, and proper exercise problems are placed adequately in between. We believe this way of reading maximizes your motivation, interest and understanding on the materials.
3. *Explore Part III in part depending on your interest:* Since Part III is dedicated to applications, you may want to read them in part. Nonetheless, we made a logical storyline assuming that every section is read sequentially. One of the key features in Part III is TensorFlow implementation. You may be able to implement all the covered algorithms, consulting with a guideline in the main body together with skeleton codes offered in problem sets and appendices.
4. *Solve four to five basic problems in each problem set:* Around 90 problems (more than 200 subproblems) are provided. Most of them elaborate on concepts discussed in the main text. The exercises range from basics on linear

algebra and probability, relatively straightforward derivations of results in the main text, in-depth exploration on non-trivial concepts not fully explained in the main text, and to programming implementation via Python, CVXPY or TensorFlow. All of the problems are tightly coupled with the storyline established. So working on at least some of the problems is essential in understanding the materials.

In the course offerings at KAIST, we have been able to cover most of the materials in Parts I and II, yet only two to three applications in Part III. Depending on the background and interest of students, and time availability, one can envision several other ways to structure a course around this book. Examples include:

1. *Semester-based course (24–26 lectures)*: Cover all the sections in Parts I and II, and two to three applications in Part III, e.g., (i) supervised learning and GANs (or Wasserstein GAN), or (ii) supervised learning and fair machine learning.
2. *Quarter-based course (18–20 lectures)*: Cover almost all the materials in Parts I and II, except some topics like LP relaxation, simplex algorithm, least squares for computed tomography, SDP relaxation, and the proof of strong duality theorem. Investigate two applications picked up from Part III.
3. *A graduate course for students with convex optimization basics*: Briefly review the contents in Part I spending around four to six lectures. Cover all the materials in Parts II and III.

Programming exercises may be covered through homeworks to save time.

## Chapter 1

# Convex Optimization Basics

## 1.1 Overview of the Book

---

**Outline** In this section, we will cover two basic stuffs. The first is logistics of this book. We will explain details as to how the book is organized and will proceed. The second thing to cover is a brief overview to this book. We are going to explore a story of how optimization was developed, as well as what we will cover throughout this book.

**Prerequisite** The key prerequisite for this book is to have a good background in linear algebra. In terms of a course, this means that you should have taken an introductory-level course on linear algebra. The course is usually offered in the Department of Mathematics, e.g., MAS109 in KAIST. Some of you might take a different yet equivalent course from other departments. This is also okay. Taking a somewhat advanced-level course (e.g., MAS212 Linear Algebra in KAIST) is optional although it is recommended. If you feel uncomfortable although you took the relevant course(s), you may want to consult with some exercise problems that we will put in proper places while preceding the book.

Another prerequisite for the book is a familiarity with the concept on probability. In terms of a course, this means that you are expected to be comfortable with the contents dealt in an undergraduate-level probability course, e.g., EE210 in

KAIST. Taking an advanced course like Random Processes (e.g., EE528 in KAIST) is optional. This is not a strong prerequisite. In fact, the optimization concept itself has nothing to do with probability. But some problems (especially the ones that arise in machine learning and deep learning, and we will also touch upon in this book) deal with some quantities which are random and therefore are described with probability distributions. This is the only reason that understanding the probability is needed. Hence, reviewing any easy-level probability book that you choose may suffice.

There must be a reason as to why linear algebra is crucial for this book. The reason is related to the definition of optimization. A somewhat casual definition of optimization is to make the best choice among many candidates or alternatives. A more math-oriented definition of optimization which we will rely upon is to choose an *optimization variable* (or a *decision variable*) so as to *minimize* or *maximize* a *certain quantity* of interest possibly given some constraint(s). Here the optimization variable and the certain quantity are the ones that relate the optimization to linear algebra. In many situations, the optimization variables are multiple real values which can be succinctly represented as a vector (just a collection of numbers). Also the certain quantity (which is a function of the optimization variables) can be represented as a function that involves matrix-vector multiplication and/or matrix-matrix multiplication, which are basic operations in linear algebra. In fact, many operations and techniques in linear algebra help formulate an optimization problem in a very succinct manner, and therefore serve to theorize the optimization field. This is the very reason that this book requires a good understanding and manipulation techniques on linear algebra. Some of you may not be well trained with expressing an interested quantity with vector/matrix forms. Please don't be offended. You will have lots of chances to be trained via some examples that will be covered throughout the book, particularly in many problem sets. Whenever some advanced techniques are needed, we will provide detailed explanations and/or relevant exercise problems which serve you to understand the required techniques.

**Problem sets** Each problem set is offered per three or four sections. So there would be 11 problem sets in total. We encourage you to cooperate with your colleagues in solving the problem sets. The problem sets are vehicles for learning, and whatever maximizes learning for you is desirable. This usually includes discussion, teaching of others and learning from others. Solutions will be available only to instructors upon request. Some problems may require programming tools like Python, CVXPY and TensorFlow. We will use Jupyter notebook. Please refer to the installation guide provided in Appendix A.1. Or you can consult with:

<https://jupyter.readthedocs.io/en/latest/install.html>

We also provide tutorials for the programming tools in appendices: (i) Appendix A for Python; (ii) Appendix B for CVXPY; and (iii) Appendix C for TensorFlow.

**Optimization** Let us investigate how the theory of optimization was developed in what contexts. Based on this, we will present detailed topics that we will learn about throughout the book.

Let us start with a story of how the theory of optimization was developed. What is optimization? As mentioned earlier, a casual informal definition of optimization is to make the best choice out of possible candidates. It comes up often in our daily life. For example, we may want to figure out a scheduling strategy for airplanes so that the total waiting time is minimized under some constraints, e.g., a certain airplane with emergency should take off no later than a specific time. Or family members may want to choose a restaurant to visit for dinner, so as to maximize the happiness of the members (if it can be quantified) given a distance constraint, e.g., a chosen restaurant should be within a few kilometers.

A mathematical definition of optimization that we are interested in here is to choose an optimization variable that minimizes (or maximizes) a certain quantity of interest possibly given some constraints. We aim at learning a theory concerning such a formal definition. Specifically we are interested in learning a *mathematical theory of optimization* which has been extensively developed and explored for a few past centuries. In fact, the birth of the theory traced back to an astronomy problem in the 1800s (Serio et al., 2002). So let us first talk about the problem so that you can readily figure out how the theory was developed.

**An astronomy problem in the early 1800s (Serio et al., 2002)** In the early 1800s, astronomers discovered a new planetoid (or called dwarf planet), which was later named *Ceres*. See Fig. 1.1. Giuseppe Piazzi is the first astronomer who discovered the planetoid. At that time, he wished to figure out an orbit of Ceres.

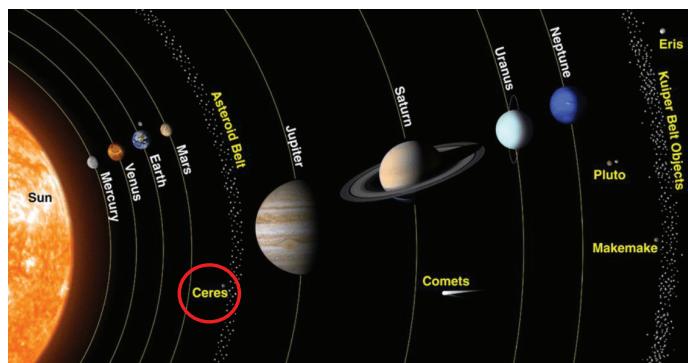


Figure 1.1. A new planetoid, named Ceres, discovered in the early 1800s.



Carl Friedrich Gauss (1777 ~ 1855)

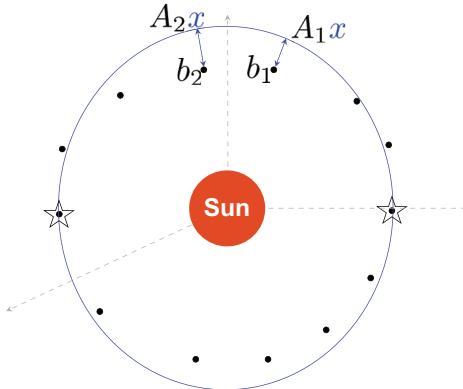
**Figure 1.2.** Carl Friedrich Gauss, a German mathematician, is the father of the optimization field. In fact, he is well-known as an inventor of the *Gaussian* distribution (one of the very famous and useful distributions in probability and statistics) as well as the *Gaussian elimination* (an efficient method which allows us to do matrix inversion or to solve linear equations).

So as an effort, he made 19 observations (of its locations) over 42 days. However, something happened in locating the trajectory. The object was lost due to the glare of the Sun. So many astronomers wanted to *predict* the hidden trajectory only from the partial 19 observations.

One interesting trial was made by a young German mathematician, named Carl Friedrich Gauss (1777 ~ 1855) ([Gauss, 1887](#)). See Fig. 1.2 for his portrait. He made a specific yet smart approach to figure out the trajectory successfully. In the process, he could develop a mathematical problem that later laid the foundation of the optimization field.

**Gauss's approach (Gauss, 1887)** Here is what Gauss did. See Fig. 1.3 for illustration. First of all, he gathered all the observations (each pointing to a location of Ceres measured at a certain time) scattered around the Sun. Let  $b_i \in \mathbf{R}^3$  indicate a coordinate of the location of the  $i$ th observation where  $i \in \{1, 2, \dots, m\}$ . Here  $m$  denotes the total number of observations that could be up to 19 in the astronomy problem context. Remember that Piazzi made 19 observations.

Next he fixed an arbitrary number of observation points, say two points, marked in the hallowed star sign in the figure. You may wonder why two. This will be explained in detail soon. He then drew an orbit that crosses the two fixed points.



**Figure 1.3.** Gauss's approach for searching the orbit of Ceres.

Actually it was well studied in the astronomy field that an orbit can be fully represented with six parameters. So the orbits that cross the two fixed points can be represented with four parameters. Depending on the choice of the free parameters, there are many ways to draw such orbits. Let's denote by  $x \in \mathbf{R}^4$  a vector that stacks up the four parameters. Here we see that the dimension of  $x$  is six minus the number of observation points that we fixed earlier. Increasing the number of fixed points, a drawn orbit would more fit into the fixed points, and this may incur the risk of not well fitting into other observations. On the other hand, without fixing any points, we need to figure out six parameters and this may be challenging only with 19 observations. As a number that well balances these two issues, Gauss chose the number two.

In terms of the vector  $x \in \mathbf{R}^4$ , Gauss could represent a point on the orbit which is the closest to each  $b_i$ . He could approximate the point as a vector that comprises linear combinations of the components of  $x$ , i.e.,  $A_i x$ . Here  $A_i \in \mathbf{R}^{3 \times 4}$  indicates a certain matrix that relates  $b_i$  to the nearest point on the orbit. He then believed that if the orbit were the ground truth that we wish to figure out, then the distance to  $b_i$ , reflected in  $\|A_i x - b_i\|$ , must be only within a location-measurement error. Here  $\|x\|$  denotes the Euclidean norm (or called the  $\ell_2$  norm), defined as  $\|x\| := \sqrt{x_1^2 + \dots + x_d^2}$  where  $d$  is the dimension of  $x$ . This motivated him to formulate the following optimization problem:

$$\min_{x \in \mathbf{R}^4} \sum_{i=1}^m \|A_i x - b_i\|^2. \quad (1.1)$$

Since we fixed the two points that an orbit must pass,  $\|A_i x - b_i\| = 0$  for the fixed points.

Gauss then observed that  $\sum_{i=1}^m \|A_i x - b_i\|^2$  can be simplified as:

$$\begin{aligned}\sum_{i=1}^m \|A_i x - b_i\|^2 &= \left\| \begin{bmatrix} A_1 x - b_1 \\ \vdots \\ A_m x - b_m \end{bmatrix} \right\|^2 \\ &= \left\| \begin{bmatrix} A_1 \\ \vdots \\ A_m \end{bmatrix} x - \begin{bmatrix} b_1 \\ \vdots \\ b_m \end{bmatrix} \right\|^2.\end{aligned}\tag{1.2}$$

Let

$$A = \begin{bmatrix} A_1 \\ \vdots \\ A_m \end{bmatrix} \in \mathbf{R}^{3m \times 4}, \quad b = \begin{bmatrix} b_1 \\ \vdots \\ b_m \end{bmatrix} \in \mathbf{R}^{3m}.\tag{1.3}$$

Then, the optimization problem can be re-written as:

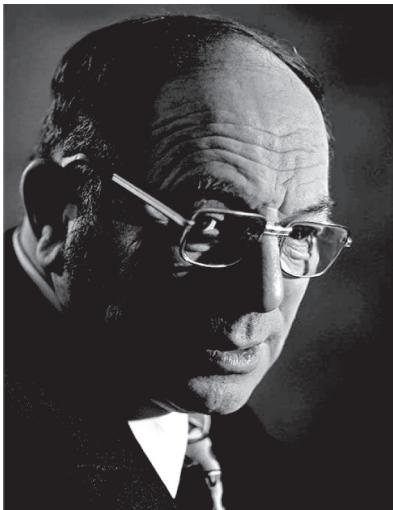
$$\min_{x \in \mathbf{R}^4} \|Ax - b\|^2.\tag{1.4}$$

**Least squares** The problem (1.4) is the famous problem that is now known as *least squares*. Notice that the word “*least*” comes from “*min*” and “*squares*” is due to the square exponent placed above the Euclidean norm. You may wonder why Gauss employed the square as an exponent in the objective function. Why not other exponents like 1, 3 or 4? This was due to the mathematical beauty that Gauss was obsessed with. Notice that using other exponents like 1 or 3 or 4, one cannot do the beautiful simplification like (1.2). If you cannot see why, then check in Prob 1.1.

The least-squares problem could open up the optimization field (since then, people have tried to theorize the field with passion) and also has played a significant role in the field. There are two reasons as to why the problem played such a big role. The first is that (1.4) has the beautiful closed-form solution:

$$x^* = (A^T A)^{-1} A^T b\tag{1.5}$$

where  $(\cdot)^T$  indicates a transpose of a matrix and  $(\cdot)^{-1}$  denotes a matrix inversion. We will later show why the solution is of the form (1.5). Please be patient until we get to the point. The second reason is that there are efficient algorithms and software that enable us to compute the solution involving matrix inverse. Even in the 1800s, there was an efficient matrix-inversion algorithm, based on the *Gaussian elimination* due to again Gauss.



Leonid Kantorovich (1912 ~ 1986)

**Figure 1.4.** Leonid Kantorovich is a Soviet economist. He is known as the father of Linear Program (LP), one of the prominent optimization classes that we will study soon in depth.

Since the development of the least squares, people tried to translate any problem of their interest to a least-squares problem. So a variety of translation techniques (that we will also study in this book) have been developed. However, people encountered many situations in which such translation does not work. This challenge was expected. As you can easily image, the least-squares problem is just a single tiny class of the entire optimization problems that can be formulated in the world.

**A breakthrough by Kantorovich** Unfortunately there was no significant progress on the optimization theory for more than a century. But another history was made in 1939 by a Soviet economist, named Leonid Kantorovich ([Kantorovich, 1960](#)). See Fig. 1.4 for his portrait.

He made a breakthrough in the course of solving a military-related problem during World War II (sort of forced to do so by the Soviet Union government). The problem that he was trying to solve was to plan expenditures and returns of soldiers to minimize the entire cost of the Soviet Union Army as well as to maximize the losses imposed on the enemy.

In the process, he could formulate an optimization problem now known as the very famous *linear program* (LP).<sup>1</sup> Simply put, the LP is an optimization problem

---

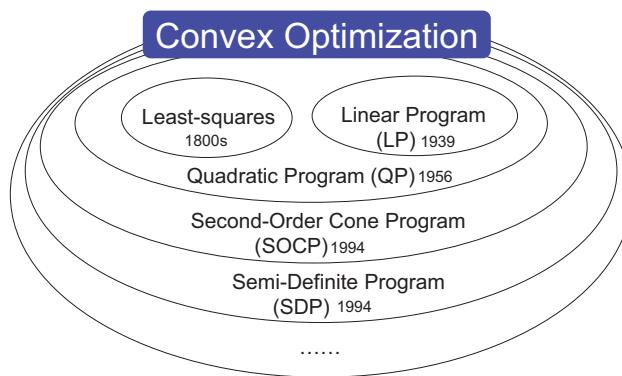
1. The “program” (or “programming”) is a jargon frequently used in the field, which refers to an optimization problem. So the formal name of LP is a linear optimization problem.

in which the objective function and the functions that appear in constraints are all linear. We will delve into its formal definition later on. Unlike the least-squares problem, the LP has no closed form solution. But the good news is that Kantorovich could develop a very efficient algorithm that achieves the optimal solution. Actually having a closed form solution is not that important as long as we know how to get to the optimal solution. This achievement won him the *Nobel Prize in Economics* in 1975 ([Kantorovich, 1989](#)).

The development of LP made people become very excited again, trying to translate an interested problem into a least-squares problem or LP. While many LP-translation techniques (that we will also study) have been developed, people encountered still many situations in which the translation is not doable.

**A class of tractable optimization problems** Inspired by the development of LP, people tried to come up with a class of *tractable* optimization problems which can be solved reliably & efficiently. LP is one such instance. In a decade, another tractable problem, called quadratic program (QP), was developed ([Frank and Wolfe, 1956](#)). It is a generalized version, as it includes as special cases the least-squares problem and LP. See Fig. 1.5.

In the 1990s, another problem, called second-order cone program (SOCP), was developed which subsumes as special cases all of the prior problems ([Nesterov and Nemirovskii, 1994](#)). Around at the same time, a larger class of problem, called semi-definite program (SDP), was developed ([Alizadeh, 1991](#); [Nesterov and Nemirovskii, 1994](#)). More and more tractable problems have been developed so far. It turns out that all of the tractable problems share the common property (concerning the word “convex”), and this property established the class of tractable problems, named *convex optimization*, which we will focus mostly on throughout this book.



**Figure 1.5.** A class of tractable optimization problems: Convex optimization.

**Book outline** This book consists of three parts. In Part I, we will study the basic concepts and several mathematical definitions required to understand what convex optimization is as well as how to translate an interested problem into a convex problem. We will then explore five instances of convex optimization problems: LP, least squares, QP, SOCP and SDP. We will focus on techniques which serve recognizing (and translating to) such problems. We will also study some prominent algorithms for solving the problems. In Part II, we will study one of the key theories in the optimization field, called *duality*. There are two types of dualities: (1) strong duality; and (2) weak duality. The strong duality is quite useful for gaining algorithmic insights for convex problems. The weak duality helps dealing with difficult non-convex problems, by providing an approximated solution. In the last third part, we will explore applications that arise in machine learning: (i) supervised learning, one of the most popular machine learning methodologies; (ii) Generative Adversarial Networks (GANs), one of the breakthrough models for unsupervised learning; and (iii) fair classifiers, which is one of the trending topics in machine learning.

## 1.2 Definition of Convex Optimization

**Recap** In the last section, we figured out how the optimization theory was developed. There were two breakthroughs in the history of optimization. The first was made by the famous Gauss. In the process of solving an astronomy problem of figuring out the orbit of Ceres (which many astronomers were trying to address in the 1800s), he could develop an optimization problem, which is now known as the least-squares problem. The beauty of the least-squares problem is two folded: (i) it has a closed form solution; and (ii) there is an algorithm which enables us to efficiently do matrix inversion required for computing the solution. The beauty of the problem opened up the optimization field and has played a significant role in the field.

The second breakthrough was made by Leonid Kantorovich. In the process of solving a military-related problem, he could formulate a problem which is now known as linear program (LP). The good thing about LP is that there is an efficient *algorithm* which allows us to compute the optimal solution reliably and efficiently although the closed form solution is unknown. In other words, Kantorovich came up with the concept of *tractable* optimization problems which can be solved via an algorithm without the knowledge of the concrete form of the optimal solution. This motivated many followers to mimic his approach, thereby coming up with a class of tractable optimization problems: convex optimization. See the class of convex optimization problems in Fig. 1.6.

**Outline** The goal of this section is to understand what convex optimization is. To this end, we will cover four stuffs. First we will study a standard mathematical formulation of optimization problems. It turns out the definition of *convex optimization problems* requires the knowledge of *convex functions*. But the definition of convex functions relies upon the concept of *convex sets*. So in the second part, we

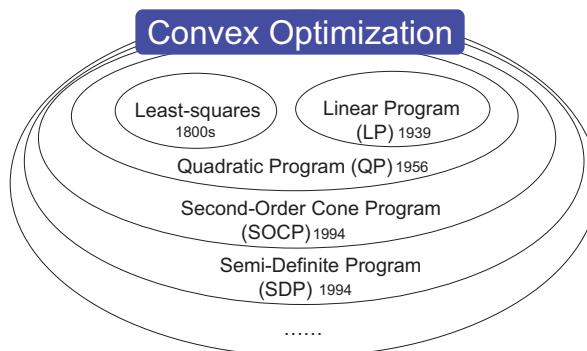


Figure 1.6. A class of tractable optimization problems: Convex optimization.

will study what the convex set is. We will also investigate some important examples which help us to be familiar with the concept as well as which play an important role in defining convex optimization. Next we will study the definition of convex functions, building upon a couple of examples and crucial properties to be explored in depth. Using all of these, we will finally investigate a standard mathematical formulation of convex optimization problems.

**Optimization problem in standard form** Let us start by recalling the definition of optimization: Choosing an optimization variable that minimizes (or maximizes) a certain quantity of interest possibly given constraints. We denote the optimization variable by  $x := [x_1, \dots, x_d]^T \in \mathbf{R}^d$  where  $d$  indicates the dimension of the variable. Denote the objective function (the certain quantity) by  $f(x) \in \mathbf{R}$ . Notice that the objective function should always be a real number, not a vector. There are two types of constraints: (i) inequality constraints; and (ii) equality constraints. The inequality constraints are represented by the form like  $f_i(x) \leq c_i$  where  $i \in \{1, \dots, m\}$ . Here  $m$  indicates the number of the constraints. Without loss of generality (WLOG), the constant  $c_i$  can be merged with  $f_i(x)$  and hence the form can be simplified as:  $f_i(x) \leq 0$ . Here the WLOG means that the general case can readily be covered with some proper modification to the simplified case of focus. That's why here we say that there is no loss of generality although we focus on the simplified case. Similarly the equality constraints can be represented by:  $h_i(x) = 0$  where  $i \in \{1, \dots, p\}$  and  $p$  denotes the number of the equality constraints.

Using these notations, one can write the standard form of optimization problems as:

$$\begin{aligned} & \min_{x \in \mathbf{R}^d} f(x) \\ & \text{subject to } f_i(x) \leq 0, i \in \{1, 2, \dots, m\}, \\ & \quad h_i(x) = 0, i \in \{1, 2, \dots, p\}. \end{aligned} \tag{1.6}$$

WLOG, it suffices to consider the minimization problem, since the maximization problem can readily come by flipping the sign of  $f(x)$ :  $\min f(x)$  is equivalent to  $\max -f(x)$ . Here we have two conventions that allow us to simplify the above form (1.6). First we use the colon “:” to indicate the “subject to”. Second,  $x \in \mathbf{R}^d$  placed below  $\min$  is often omitted since the role of  $x$  is clear enough from the context. Hence, the simpler form reads:

$$\begin{aligned} & \min f(x) : f_i(x) \leq 0, i \in \{1, 2, \dots, m\}, \\ & \quad h_i(x) = 0, i \in \{1, 2, \dots, p\}. \end{aligned} \tag{1.7}$$

Two more things to note. One is the *optimal value*, denoted by  $p^* := \min f(x)$ . The other is the *optimal solution*, denoted by  $x^* := \arg \min f(x)$ . Or it is called the *minimizer*. Here “ $\arg \min$ ” stands for “*argues the one that minimizes*”.

**Convex set** Now what is *convex optimization* that we wish to figure out in this section? As mentioned in the beginning, to define this, we need to know about the concept of convex functions. But the definition of convex functions requires the knowledge of convex sets. So we will first study the definition of the convex set.

A set  $\mathcal{S}$  is said to be *convex* if and only if

$$x, y \in \mathcal{S} \implies \lambda x + (1 - \lambda)y \in \mathcal{S}, \quad \forall \lambda \in [0, 1] \quad (1.8)$$

where the sign “ $\forall$ ” means “for all”.

**Examples: Point, line, plane, line segment, ...** To get a concrete feel about what the convex set means, let us explore several examples. The first simplest example is the set containing a single point. This is obviously a convex set, as any convex combination that lies in between  $x$  and  $y$ , represented as  $\lambda x + (1 - \lambda)y$ , is just the single point.

The second simplest example is perhaps the set that contains a *line* that lives in a 2-dimensional ambient space. This is also convex because any convex combination of two points lying on a line should also lie on the line. Here let us investigate how to represent the convex set. This representation will help us to understand the concept of convex optimization later on. Notice that the line in a 2-dimensional space can be represented as:  $y = ax + b$  where  $a$  and  $b$  indicate the slope and  $y$ -intercept, respectively. Hence, one can represent the set as:

$$\mathcal{S} = \{x := [x_1, x_2]^T : x_2 = a_1 x_1 + b_1\}. \quad (1.9)$$

Here we use the notations  $(x_1, x_2)$  instead of  $(x, y)$ ; also  $(a_1, b_1)$  instead of  $(a, b)$ . Using vector notations, one can define  $a := [-a_1, 1]^T$  and  $b := b_1$ , which in turn simplifies the representation (1.9) as:

$$\mathcal{S} = \{x : a^T x - b = 0\}. \quad (1.10)$$

The third example is the naive extension of the second example: *a plane living in a 3-dimensional space*. This is also obviously a convex set, as any combination of two points lying on a plane also lies on the plane. The representation of the convex set is exactly the same as (1.10), except that now the dimension of  $x$  and  $a$  are 3. Why?

The fourth example is the one in which the dimension of an object of interest differs from  $d \geq 2$  by 2. One such example is the set that contains a *line* living in a 3-dimensional space. This is also a convex set, since the object of interest is a

line. But the representation of such a convex set is different from that of (1.10). A line is actually the *intersection* of two planes in a 3-dimensional space. So the representation of the set should read:

$$\mathcal{S} = \{x : a_1^T x - b_1 = 0, a_2^T x - b_2 = 0\}. \quad (1.11)$$

Defining  $A := [a_1, a_2]^T$  and  $b := [b_1, b_2]^T$ , one can simplify this as:

$$\mathcal{S} = \{x : Ax - b = \mathbf{0}\} \quad (1.12)$$

where the thick  $\mathbf{0}$  indicates the all-zero vector  $[0, 0]^T$ . For illustrative simplicity, we will use the normal 0 even to denote the all-zero vector, as it may be clear from the context.

Looking carefully at these examples, one can see that the representation of a line, a plane or a hyperplane (a subspace whose dimension is one less than that of its ambient space) lying in a larger-dimensional ambient space reads the form like (1.12). Depending on the dimension of the matrix  $A \in \mathbb{R}^{p \times d}$ ,  $\mathcal{S}$  may refer to the set containing a line, a plane or a higher-dimensional plane. For instance, when  $d - p = 1$ ,  $\mathcal{S}$  refers to a line. When  $d - p = 2$ ,  $\mathcal{S}$  indicates a plane. The set represented by the form (1.12) is called an *affine* set. An affine function is a linear function that allows for having a bias constant term; the formal definition will be given later on. Since the set in (1.12) includes the affine function  $f(x) = Ax - b$ , it is called an affine set.

Another example that we would like to mention is the *line segment*; see the left top in Fig. 1.7. Again this is obviously a convex set. On the other hand, a broken line, a line that is broken in the middle, is not convex, since some convex combination of two points in the broken line may fall into to somewhere in the broken place; see the right top in Fig. 1.7.

**More examples: Convex polygon, polyhedron, polytope, ...** You may wonder if there are any other examples beyond point/line/plane. Of course, there are many. One object that you may be interested in is: a *polygon* living in a

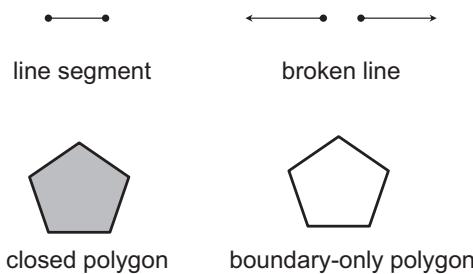


Figure 1.7. Examples of convex sets (left two) and non-convex sets (right two).

2-dimensional space. In particular, the closed polygon in which points inside (and boundary at) the polygon are included in the set is a convex set. See the left bottom in Fig. 1.7 for illustration. On the other hand, the boundary-only polygon is not a convex set. See the right bottom in Fig. 1.7.

As you may imagine, the representation of the closed-polygon convex set is different from the form (1.12) of affine sets. The closed-polygon can actually be represented as the *intersection* of half-planes. A half-plane is a planar region consisting of all points on one side of an infinite straight line, and no points on the other hand. It is represented by  $a_i^T x - b_i \leq 0$ . Hence, the representation of such a set reads:

$$\mathcal{S} = \{x : Ax - b \leq 0\} \quad (1.13)$$

where the inequality indicates a component-wise inequality.

Similarly, a *polyhedron* living in a 3-dimensional ambient space (or a *polytope* living in an  $d$ -dimension space and hence difficult to visualize) is a convex set and can also be represented as the form like (1.13).

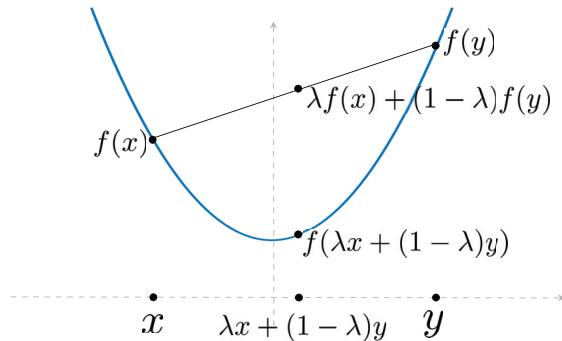
**Convex function** We are now ready to define the convex function. A real-valued function  $f(x)$  is said to be *convex* if the following two conditions are satisfied:

- (i) the domain of the function  $f$ , denoted by  $\text{dom } f$  (the set in which the input  $x$  of the function lies in), is a convex set; and
- (ii) for  $x, y \in \text{dom } f$  and  $\lambda \in [0, 1]$ :

$$f(\lambda x + (1 - \lambda)y) \leq \lambda f(x) + (1 - \lambda)f(y). \quad (1.14)$$

Here you can see why we needed to know about the concept of the *convex set*. The concept appears while mentioning the first condition that  $\text{dom } f$  should satisfy. Actually this “convex set” condition is required; otherwise, a problem occurs when it comes to stating the second key condition in (1.14), because the function in the left hand side cannot be defined. Notice that the input argument in the function is  $\lambda x + (1 - \lambda)y$  and this should be in  $\text{dom } f$  (meaning that  $\text{dom } f$  should be convex); otherwise,  $f(\lambda x + (1 - \lambda)y)$  cannot be defined.

If you think about some picture that reflects the second key condition (1.14), then you can readily get a feel about why the convex function should be defined in such a manner. The meaning of “convex” is “bowl-shaped”. So we can think about a bowl-shaped curve like the one as illustrated in Fig. 1.8. Consider two points, say  $x$  and  $y$ , and a  $\lambda$ -weighted convex combination,  $\lambda x + (1 - \lambda)y$ . The function evaluated at  $\lambda x + (1 - \lambda)y$  is on the bowl-shaped curve while the same-weighted convex combination  $\lambda f(x) + (1 - \lambda)f(y)$  of the two functions evaluated at  $x$  and



**Figure 1.8.** A geometric intuition behind convex functions.

$y$  is above  $f(\lambda x + (1 - \lambda)y)$ . Hence, the key condition (1.14) comes naturally as a consequence of the bowl-shaped feature of the curve.

There are tons of examples of convex functions and also many of these are the ones that you should be familiar with if you wish to be an expert in this field. Or at least you may want to know some of them in which the problems of your interest can be linked to. But exploring many of such examples may be too much now – it is just the beginning of the book, so that way you will be exhausted shortly. Thus we will here investigate only a couple of examples. One example of a convex function is:

$$f(x) = \frac{1}{x} \quad x > 0. \quad (1.15)$$

Here the function is indeed bowl-shaped, so it respects the second condition (1.14). Also  $\text{dom } f = \{x : x > 0\}$  is a convex set, satisfying the first condition. Hence, the function is convex.

Now what about a slightly different function:

$$f(x) = \frac{1}{x} ? \quad (1.16)$$

Here the distinction is that  $\text{dom } f$  is not explicitly defined. In this case, we should think about an *implicit* constraint that  $x$  should satisfy. The implicit constraint is:  $x \neq 0$ , thus yielding:

$$\text{dom } f = (-\infty, 0) \cup (0, \infty).$$

Since  $\text{dom } f$  is not convex, the function is not convex either.

There is a way to handle this issue to make such a non-trivial function convex. The way is to make  $\text{dom } f$  span the entire region (making it convex) while setting the function to some arbitrary quantities for newly added regions. For instance, we

can define the function as:

$$f(x) = \begin{cases} \frac{1}{x}, & x > 0; \\ +\infty, & x \leq 0. \end{cases} \quad (1.17)$$

Notice that now  $\text{dom} = (-\infty, \infty)$  is a convex set. Also one can readily verify that the key condition (1.14) is satisfied. Hence, the function is convex.

There is another function which is defined very similarly to the convex function. That is, a *concave function*. We say that  $f(x)$  is *concave* if  $-f(x)$  is convex. The geometric intuition says that the function of a bell shape is concave. Also a function is said to be *affine* (linear plus bias) if it is convex and concave.

*On a side note:* Here you may wonder if there is a corresponding set with respect to (w.r.t.) concave functions, like a concave set. Unlike a convex set, people do not introduce any definition for a set regarding concave functions. Remember in the definition of convex functions that the convex set was employed only for the purpose of making  $f(\lambda x + (1 - \lambda)y)$  definable at the convex combination. Hence, the same convex set suffices to define concave functions.

**Convex sets defined in terms of general convex functions** Previously we investigated a bunch of examples of convex sets where only affine functions,  $Ax - b$  (linear and bias-allowing functions), are introduced. There are many convex sets concerning general convex functions. Here we list a couple of such examples.

One such example is:

$$\mathcal{S} = \{x : f(x) \leq 0\} \quad (1.18)$$

where  $f(x)$  is a convex function. Here is the proof that  $\mathcal{S}$  is a convex set. Suppose  $x, y \in \mathcal{S}$ . Then,  $f(x) \leq 0$  and  $f(y) \leq 0$ . This together with the convexity of  $f$ , reflected in the condition (1.14), gives:

$$f(\lambda x + (1 - \lambda)y) \leq 0,$$

which in turn implies that  $\lambda x + (1 - \lambda)y \in \mathcal{S}$ . This completes the proof.

Another example is the intersection of such convex sets:

$$\begin{aligned} \mathcal{S} &= \mathcal{S}_1 \cap \mathcal{S}_2 \\ \mathcal{S}_1 &= \{x : f_1(x) \leq 0\}, \quad \mathcal{S}_2 = \{x : f_2(x) \leq 0\}. \end{aligned} \quad (1.19)$$

Try the proof in Prob 1.4. Actually the intersection of arbitrary convex sets is also convex – check in Prob 1.4 as well.

**Convex optimization problem in standard form** We are now ready to define the convex optimization problem. It is an optimization problem which satisfies the following three: (i) The objective function is convex; (ii) The set induced by inequality constraints is convex; and (iii) The set induced by equality constraints is convex. So the standard form of the convex optimization problem reads (1.7) in which (i)  $f(x)$  is convex; (ii)  $f_i(x)$  is convex; and (iii)  $h_i(x)$  is affine. Notice that the set induced by affine equality constraints  $\mathcal{S} = \{x : Ax - b = 0\}$  is a convex set as we studied earlier.

*On a side note:* Notice that the standard form exhibits just a *sufficient* condition under which the set induced by inequality constraints is convex. There are indeed a bunch of convex sets which take the form like (1.18) yet having a non-convex function  $f(x)$ . But it turns out that in many cases, convex sets can be represented as the form like (1.18) with a convex function  $f(x)$ . This is one of the main reasons that people define convex optimization in such a manner.

**Look ahead** There is another reason that the convex optimization problem is defined in such a manner. This is because the way of definition makes the problem *tractable*. In the next section, we will provide an intuition as to why convex optimization is tractable. We will then start investigating one instance of convex optimization: Linear Program (LP).

## 1.3 Tractability of Convex Optimization and Gradient Descent

---

**Recap** In the last section, we studied the concept of convex sets and convex functions to understand what convex optimization is. We then figured out that convex optimization is defined as:

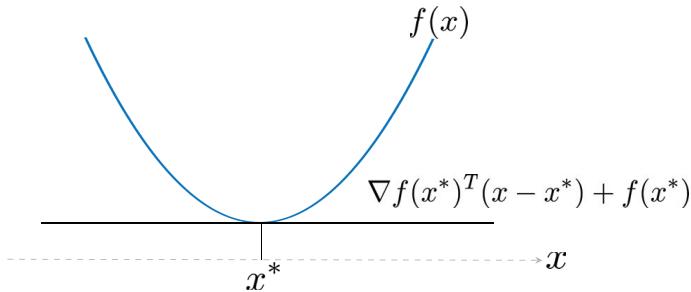
$$\begin{aligned} \min f(x) : f_i(x) \leq 0, i \in \{1, \dots, m\}, \\ h_i(x) = 0, i \in \{1, \dots, p\} \end{aligned} \quad (1.20)$$

where  $f(x)$  and  $f_i(x)$ 's are convex and  $h_i(x)$ 's are affine functions. There was a reason that many people have been interested in such convex optimization defined particularly as above. The reason is that the way of defining the problem makes the problem *tractable*. Here what it means by *tractable* is that the optimal solution can be achieved via an *algorithm* (often with the help of a computer) even if the closed form solution is unknown.

**Outline** The main goal of this section is to understand why such a particular way of definition enables *tractability*. We will try to understand this by focusing on two cases: (i) *unconstrained* minimization; and (ii) *constrained* minimization. For the unconstrained case, we will present an explicit intuition as to why the convex objective function yields a tractable way of solving the problem. Specifically we will first derive a simple-looking necessary and sufficient condition for the optimality of a solution and then argue that there are efficient algorithms that allow us to satisfy the condition, thus obtaining the optimal solution. We will also study one very prominent algorithm, named *gradient descent*. For the constrained case, on the other hand, we will rely upon a well-known theory (to be studied in depth in Part II) to argue that there are also efficient algorithms that allow us to solve the problem.

Another goal of this section is to give an overview to the contents regarding one instance of convex optimization problems: Linear Program (LP). This is what we will cover through a couple of upcoming sections.

**Assumption** While investigating the two cases, we will assume that: (i) the objective function  $f(x)$  is *differentiable* at every point  $x$  in  $\text{dom } f$ ; (ii)  $\text{dom } f$  is open; and (iii)  $f(x)$  has a *stationary* point, i.e., there exists  $x^*$  such that  $\nabla f(x^*) = 0$ . The rationale behind this assumption is two folded. The first is that the assumption represents many practically-relevant scenarios. Second, the assumption allows us to easily explain an intuition behind the tractability of convex optimization. This will be clearer as this section progresses.



**Figure 1.9.** 1st order condition of convex functions:  $f(x) \geq \nabla f(x^*)^T(x - x^*) + f(x^*)$ ,  $\forall x \in \text{dom}f$ .

**Unconstrained minimization** Let us start by investigating the unconstrained convex optimization:

$$\min f(x).$$

Recall the meaning of *convex*: “bowl-shaped”. So one can think of a graph illustrated in Fig. 1.9. This graph also respects our assumption that there exists a stationary point. Here you can easily see that the slope of the objective function at the optimal point  $x^*$  is 0, and also vice versa (meaning that the point with the slope being 0 is the optimal solution). This naturally guides us to conjecture that  $\nabla f(x^*) = 0$  is a sufficient and necessary condition in order for  $x^*$  to be optimal:

$$\nabla f(x^*) = 0 \iff f(x) \geq f(x^*) \quad \forall x \in \text{dom}f. \quad (1.21)$$

It turns out this conjecture indeed holds. Here is the proof.

*Proof of the direct part ( $\implies$ ) in (1.21):* To gain some insights, let us see a convex function  $f(x)$  in Fig. 1.9. Pick up a point  $(x^*, f(x^*))$ . Now consider a line that passes through the point  $(x^*, f(x^*))$  with a slope  $\nabla f(x^*)$  so that it is tangent to  $f(x)$ . Then, the line should read:  $\nabla f(x^*)^T(x - x^*) + f(x^*)$ . Here the picture suggests that the convex function  $f(x)$  is above (or touching) the line:

$$f(x) \geq \nabla f(x^*)^T(x - x^*) + f(x^*) \quad \forall x \in \text{dom}f. \quad (1.22)$$

It turns out this is indeed the case, meaning that the condition (1.22) (together with  $\text{dom}f$  being convex) holds for any  $x, y \in \text{dom}f$  if and only if  $f(x)$  is convex. This is one of the crucial properties of convex functions, called the “1st order condition of convex functions”. The proof of this is omitted here, but you will have a chance to prove this in Prob 1.5. This together with the hypothesis  $(\nabla f(x^*) = 0)$  gives:  $f(x) \geq f(x^*)$ ,  $\forall x \in \text{dom}f$ .

*Proof of the converse part ( $\Leftarrow$ ) in (1.21):* The converse proof relies upon the following fact:

$$f(x) \geq f(x^*) \quad \forall x \in \text{dom}f \implies \nabla f(x^*)^T(x - x^*) \geq 0 \quad \forall x \in \text{dom}f. \quad (1.23)$$

Let us adopt this for the time being. We will prove this fact once we finalize the converse proof.

Suppose  $\nabla f(x^*) \neq 0$ . Here the key thing to note is that there is no constraint on  $x$ , except that  $x \in \text{dom}f$ . So one can choose  $x$  such that  $x - x^*$  points to an *arbitrary* direction. This implies that we can easily choose  $x$  such that

$$\nabla f(x^*)^T(x - x^*) < 0. \quad (1.24)$$

This contradicts with the RHS of (1.23). Hence, it must be that  $\nabla f(x^*) = 0$ . This completes the proof.

Let us now prove (1.23) which we deferred proving earlier.

*Proof of (1.23):* The proof idea is by contradiction. Suppose that there exists  $x$  (i.e.,  $\exists x \in \text{dom}f$ ) such that:

$$\nabla f(x^*)^T(x - x^*) < 0. \quad (1.25)$$

Consider a point:  $z(\lambda) := \lambda x + (1 - \lambda)x^*$  where  $\lambda \in [0, 1]$ . Notice that  $z(\lambda) \in \text{dom}f$ , as the function  $f$  is convex and therefore its domain is a convex set. Here what we want to show is that for a very small  $\lambda \approx 0$ ,  $f(z(\lambda)) < f(x^*)$ . This is because  $f(z(\lambda)) < f(x^*)$  contradicts with the fact that  $x^*$  is an optimal solution, thus leading to contradiction. To show this, we consider the following quantity:

$$\begin{aligned} \frac{d}{d\lambda}f(z(\lambda)) &\stackrel{(a)}{=} \nabla f(z(\lambda))^T \frac{d}{d\lambda}z(\lambda) \\ &\stackrel{(b)}{=} \nabla f(z(\lambda))^T(x - x^*) \end{aligned}$$

where (a) follows from a chain rule and (b) is due to the definition of  $z(\lambda) := \lambda x + (1 - \lambda)x^*$ . Now evaluating both sides at  $\lambda = 0$ , we get:

$$\left. \frac{d}{d\lambda}f(z(\lambda)) \right|_{\lambda=0} = \nabla f(x^*)^T(x - x^*) < 0 \quad (1.26)$$

where the last inequality comes from our assumption (1.25). Here the derivative of  $f(z(\lambda))$  being negative at  $\lambda = 0$  implies that  $f(z(\lambda))$  decreases with  $\lambda$  and therefore:

$$f(z(\lambda)) < f(x^*). \quad (1.27)$$

This contradicts with the hypothesis  $f(x) \geq f(x^*) \ \forall x \in \text{dom}f$ . This completes the proof.

**Gradient descent** So what we can conclude with respect to (w.r.t.) unconstrained minimization is that:

$\nabla f(x^*) = 0$  is a sufficient and necessary condition for  $x^*$  to be optimal.

This suggests that it suffices to find a point such that (s.t.) its gradient is 0. But there are some issues in obtaining such a point. Two issues. One is that computing  $\nabla f(x)$  may not be that simple. The second is that analytically finding such a point may not be doable even if one can explicitly compute the gradient. However, there is a good news. The good news is that there are several algorithms which allow us to find such a point numerically without the knowledge of the closed form solution. One prominent algorithm that has been widely employed in a variety of fields is: *gradient descent*.

Here is how the algorithm works. The gradient descent is an *iterative* algorithm. Suppose that at the  $t$ th iteration, we have an estimate of  $x^*$ , say  $x^{(t)}$ . We then compute the gradient of the function evaluated at the estimate:  $\nabla f(x^{(t)})$ . Next we update the estimate along a direction being *opposite* to the direction of the gradient:

$$x^{(t+1)} \leftarrow x^{(t)} - \alpha^{(t)} \nabla f(x^{(t)}) \quad (1.28)$$

where  $\alpha^{(t)} > 0$  indicates the learning rate (or called a step size) that usually decays like  $\alpha^{(t)} = \frac{1}{2^t}$ .

If you think about it, this update rule makes an intuitive sense. Suppose  $x^{(t)}$  is placed right relative to the optimal point  $x^*$ , as in the two-dimensional case<sup>2</sup> illustrated in Fig. 1.10.

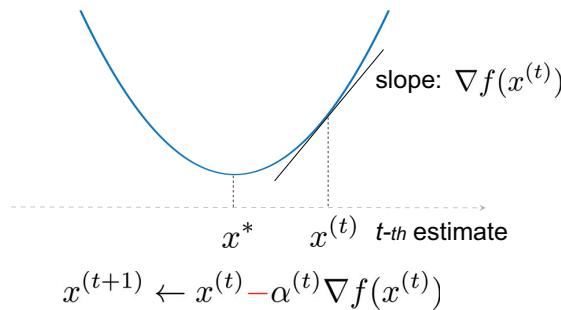
Then, we should move  $x^{(t)}$  to the *left* so that it becomes closer to  $x^*$ . The update rule actually does this, as we *subtract* by  $\alpha^{(t)} \nabla f(x^{(t)})$ . Notice that  $\nabla f(x^{(t)})$  points to the *right* direction given that  $x^{(t)}$  is placed right relative to  $x^*$ . We repeat this procedure until it converges. It turns out: as  $t \rightarrow \infty$ , it converges:

$$x^{(t)} \longrightarrow x^*, \quad (1.29)$$

as long as the learning rate is chosen properly, like the one decaying exponentially. We will not touch upon the proof of this convergence. In fact, the proof is difficult – even there is a big field in statistics which intends to prove the convergence of a variety of algorithms (if the convergence holds).

---

2. In a higher-dimensional case, it is difficult to visualize how  $x^{(t)}$  is placed. Hence, we focus on the two-dimensional case. It turns out that gradient descent works even for high-dimensional settings although it is not 100% intuitive.



**Figure 1.10.** How gradient descent works.

**Constrained minimization** As mentioned in the beginning of this section, to give insights in the constrained minimization, we will rely upon a well-known theory that we will touch upon in depth later in Part II. The theory is called *strong duality*. There is a rigorous mathematical statement of strong duality. But here we will state only an implication of strong duality because in that way we can even understand a rationale without introducing any further notations and concepts. The implication of strong duality is:

*Convex-constrained problem can be translated into an *unconstrained convex optimization without loss of optimality*.*

This suggests that it suffices to consider a translated *unconstrained* minimization; hence, we can rely on efficient algorithms like gradient descent to solve the problem. In summary, we have efficient algorithms for solving both unconstrained and constrained convex optimization. This is exactly the reason why we define convex optimization in the particular manner mentioned at the beginning.

You may be eager to know about *strong duality* right now because we relied heavily on the theory for the purpose of explaining the tractability of convex optimization. However, we will not touch upon it now because the proof requires many deep backgrounds. If we do so now, you will be easily distracted, potentially losing an interest in the optimization field. Please be patient until we get to that point in Part II.

**Overview of Linear Program (LP)** From now on, we will start investigating several instances of convex optimization problems. One instance that we will take first is: Linear Program (LP).

$$\begin{aligned} \min f(x) : \quad & f_i(x) \leq 0, \quad i \in \{1, \dots, m\}, \\ & h_i(x) = 0, \quad i \in \{1, \dots, p\}. \end{aligned} \tag{1.30}$$

Here we say that (1.30) is an LP if all functions  $f(x)$ ,  $f_i(x)$ 's,  $h_i(x)$ 's are *affine*.

Since Kantorovich's breakthrough, people realized that many interesting/important problems can be formulated as LPs such as: (i) resource allocation problems (like the military-related problem that Kantorovich considered); (ii) transportation problems ([Monge, 1781](#)) (important problems that arise in economics and other relevant areas); (iii) the classification problem (one of the most classical and popular problems in machine learning); (iv) the network flow problem (a fundamental problem in computer networks); and so on and so forth.

Moreover, some very difficult problems in which the optimization variable is boolean (binary values) can be approximated as an LP via a relaxation technique. Very interestingly, in some cases, the LP relaxation yields the *exact* solution to the original problem, meaning it comes without loss of optimality.

**Look ahead** Through a couple of upcoming sections, we will deal with the above examples together with algorithms and software implementation. Specifically we are going to cover four stuffs. First we will study a few examples that can be formulated as LPs. Second, we will study the LP relaxation technique which we claimed useful for some very difficult problems. Third, we will investigate efficient algorithms for solving LPs. Lastly we will study how to implement such algorithms using software tools like CVXPY running in Python.

## Problem Set 1

---

**Prob 1.1 (Least Squares)** Let  $A := [a_1 \ \cdots \ a_m]^T$  and  $b := [b_1 \ \cdots \ b_m]^T$  where  $a_i \in \mathbf{R}^d$  and  $b_i \in \mathbf{R}$ ,  $i \in \{1, \dots, m\}$ .

- (a) Consider a function  $f : \mathbf{R}^d \rightarrow \mathbf{R}$ :  $f(x) = \|a_1^T x - b_1\| + \cdots + \|a_m^T x - b_m\|$  where  $x \in \mathbf{R}^d$  and  $\|\cdot\|$  denotes the Euclidean norm. A student claims that the function  $f$  can be represented as:

$$f(x) = \|Ax - b\|. \quad (1.31)$$

Prove or disprove the claim.

- (b) Consider another function  $f : \mathbf{R}^d \rightarrow \mathbf{R}$ :

$$f(x) = \|Ax - b\|^2. \quad (1.32)$$

Using only the definition of a convex function, show that  $f(x)$  is convex in  $x \in \mathbf{R}^d$ .

**Prob 1.2 (Basics on gradients)** Let  $f : \mathbf{R}^d \rightarrow \mathbf{R}$  be a differentiable function. Let  $x = [x_1, x_2, \dots, x_d]^T \in \mathbf{R}^d$ ,  $B \in \mathbf{R}^{d \times d}$  and  $c \in \mathbf{R}^d$ .

- (a) Explain the concept of *gradient*  $\nabla f$  with respect to (w.r.t.)  $x$  (or denoted by  $\nabla_x f$ ).
- (b) Suppose  $f(x) = x^T x$ . Derive  $\nabla f$  w.r.t.  $x$ . Show all of your detailed derivation.
- (c) Suppose  $f(x) = x^T B x$ . Derive  $\nabla f$  w.r.t.  $x$ . Show all of your detailed derivation.
- (d) Suppose  $f(x) = c^T x$ . Derive  $\nabla f$  w.r.t.  $x$ . Show all of your detailed derivation.
- (e) Suppose  $f(x) = x^T c$ . Derive  $\nabla f$  w.r.t.  $x$ . Show all of your detailed derivation.

**Prob 1.3 (Representation of convex sets)**

- (a) In Section 1.2, it was claimed that a plane in a 3-dimensional space can be represented as

$$\mathcal{S} = \{x : a^T x - b = 0\}. \quad (1.33)$$

where  $x, a \in \mathbf{R}^3$  and  $b \in \mathbf{R}$ . Explain why.

- (b) State the definition of a *hyperplane*. What is the representation of a set that indicates a hyperplane in a  $d$ -dimensional ambient space?

Specify the dimension of matrices and vectors (if any) employed in your representation.

- (c) State the definition of a *polytope*. What is the representation of a set that indicates a polytope in a  $d$ -dimensional ambient space? Specify the dimension of matrices and vectors (if any) employed in your representation.

**Prob 1.4 (Convex sets)** Let  $f_1 : \mathbf{R}^d \rightarrow \mathbf{R}$  and  $f_2 : \mathbf{R}^d \rightarrow \mathbf{R}$  be convex functions.

- (a) In Section 1.2, we proved that  $S_i = \{x : f_i(x) \leq 0\}$  is convex for  $i \in \{1, 2\}$ . Show that  $\mathcal{S} = S_1 \cap S_2$  is convex.
- (b) Suppose  $S_1$  and  $S_2$  are arbitrary convex sets. Prove that  $\mathcal{S} = S_1 \cap S_2$  is convex.
- (c) Suppose  $f : \mathbf{R}^d \rightarrow \mathbf{R}$  be a concave function. A student claims that  $\mathcal{C} = \{x : f(x) \leq 0\}$  is not convex. Prove or disprove the claim.

**Prob 1.5 (1st-order condition of convexity)** Suppose  $f : \mathbf{R}^d \rightarrow \mathbf{R}$  is differentiable, i.e., its gradient  $\nabla f$  exists at each point in  $\text{dom } f$ . In Section 1.3, it was claimed that  $f$  is convex if and only if

$$\begin{aligned} \text{dom } f &\text{ is convex;} \\ f(x) &\geq \nabla f(x^*)^T(x - x^*) + f(x^*) \quad \forall x, x^* \in \text{dom } f. \end{aligned} \tag{1.34}$$

This problem explores the proof of the above.

- (a) Suppose  $d = 1$ . Show that if  $f(x)$  is convex, then (1.34) holds.
- (b) Suppose  $d = 1$ . Show that if (1.34) holds, then  $f(x)$  is convex.
- (c) Prove the claim for arbitrary  $d \in \mathbf{N}$ .

**Prob 1.6 (Composition of convex functions)** Let  $f : \mathbf{R}^d \rightarrow \mathbf{R}$  be a real-valued function. The same holds for  $g, f_1$  and  $f_2$ .

- (a) Show that if the functions  $f(x)$  and  $g(x)$  are convex, so is  $f(x) + g(x)$ .
- (b) Let  $A$  and  $b$  be a matrix and a vector of compatible size. Show that if  $f(x)$  is convex, so is  $f(Ax + b)$ .
- (c) Show that if  $f_1$  and  $f_2$  are convex, so is  $\max\{f_1(x), f_2(x)\}$ .
- (d) Show that if  $f(x)$  is convex, then  $-\log(-f(x))$  is convex on  $\{x : f(x) < 0\}$ .

**Prob 1.7 (Jensen's inequality)** Suppose that  $f : \mathbf{R}^d \rightarrow \mathbf{R}$  is convex,  $x_1, \dots, x_k \in \text{dom } f$ , and  $\lambda_1, \dots, \lambda_k \geq 0$  with  $\lambda_1 + \dots + \lambda_k = 1$ . Show that

$$f(\lambda_1 x_1 + \dots + \lambda_k x_k) \leq \lambda_1 f(x_1) + \dots + \lambda_k f(x_k). \tag{1.35}$$

Also, identify the conditions under which the equality holds.

**Prob 1.8 (Logistic function)** Let  $x, w \in \mathbf{R}^d$ . Let

$$\sigma(t) := \frac{1}{1 + e^{-t}}, \quad t \in \mathbf{R}. \quad (1.36)$$

- (a) A student claims that  $L(w) := -\log \sigma(w^T x)$  is convex in  $w$ . Prove or disprove the claim.
- (b) A student claims that  $L(w) := \log(1 - \sigma(w^T x))$  is convex in  $w$ . Prove or disprove the claim.

**Prob 1.9 (Convex function vs convex set)** Let  $f : \mathbf{R}^d \rightarrow \mathbf{R}$  be a real-valued function. In Section 1.2, we proved that if  $f(x)$  is convex,  $\mathcal{S} = \{x : f(x) \leq 0\}$  is a convex set. A student claims that if  $\mathcal{S}$  is a convex set,  $f(x)$  must be convex. Prove or disprove the claim.

**Prob 1.10 (Gradient descent)**

- (a) Consider an optimization problem:

$$\min x^2 - 2x \quad (1.37)$$

where  $x \in \mathbf{R}$ . Suppose we perform gradient descent with an initial point  $x^{(0)} = 5$  and the learning rate  $\alpha^{(t)} = \frac{1}{2^t}$ . Using Python to plot the objective function evaluated at  $x^{(t)}$  as a function of  $t$  where  $x^{(t)}$  denotes the estimate at the  $t$ -th iteration. What is the limiting value of  $x^{(t)}$ ? Does it converge to the optimal solution that you can obtain analytically?

- (b) Now consider:

$$\max -x^2 + 4x \quad (1.38)$$

where  $x \in \mathbf{R}$ . Can you apply gradient descent to approach the optimal solution numerically (without converting max into min optimization)? If so, redo part (a) with the same initial point and the learning rate. Otherwise, come up with another iterative algorithm which bears similarity to gradient descent, yet which allows us to obtain the optimal solution. Also explain why. In addition, plot the objective function evaluated at  $x^{(t)}$  as a function of  $t$  where  $x^{(t)}$  denotes the  $t$ -th estimate of your algorithm.

**Prob 1.11 (True or False?)**

- (a) A line is represented as:

$$\mathcal{S} = \{x : a^T x - b = 0\} \quad (1.39)$$

where  $a, x \in \mathbf{R}^d$  and  $b \in \mathbf{R}$ .

- (b) Consider a set  $\{(x_1, x_2) \in \mathbf{R}^2 : \frac{x_1}{x_2} \geq 1, x_1 > 0, x_2 > 0\}$  where  $x_1, x_2 \in \mathbf{R}$ .  
The set is convex.
- (c) Consider a set  $\{(x, t) \in \mathbf{R}^{d+1} : x^T x \leq t\}$  where  $x \in \mathbf{R}^d, t \in \mathbf{R}, d \in \mathbf{N}$ .  
The set is convex.
- (d) Suppose that  $f_1, f_2, f_3, f_4 : \mathbf{R}^d \rightarrow \mathbf{R}$  are concave functions. Then,

$$\max\{\min\{f_1(x), f_2(x)\}, \min\{f_3(x), f_4(x)\}\} \quad (1.40)$$

is convex.

- (e) Consider functions  $f_i(x)$ ,  $i \in \{1, 2, \dots, m\}$ . Then, the following function

$$\max_{x \in \mathbf{dom}f} \sum_{i=1}^m \lambda_i f_i(x) \quad (1.41)$$

is always convex in  $\lambda := [\lambda_1, \dots, \lambda_m]^T \in \mathbf{R}^m$ .

## 1.4 Linear Program (LP)

---

**Recap** In Section 1.3, we tried to understand why convex optimization stated below is *tractable*, i.e., it can be solved efficiently via an algorithm even if the closed form solution is unknown:

$$\begin{aligned} \min f(x) : f_i(x) \leq 0, i \in \{1, \dots, m\}, \\ h_i(x) = 0, i \in \{1, \dots, p\} \end{aligned} \quad (1.42)$$

where  $f(x)$  and  $f_i(x)$ 's are *convex* and  $h_i(x)$ 's are *affine* functions. To this end, we considered two cases: (i) unconstrained case; and (ii) constrained case. For unconstrained minimization, we first derived the necessary and sufficient condition for  $x^*$  to be optimal:  $\nabla f(x^*) = 0$ , assuming that there exists a stationary point and that  $f$  is differentiable at every point  $\in \text{dom } f$ , which is open. We then investigated one prominent algorithm for finding such a solution: *gradient descent*. The gradient descent is an iterative algorithm which allows us to achieve  $x^*$  numerically. For constrained minimization, on the other hand, we relied upon a theory so called *strong duality* which enables translating a convex-constrained problem into an unconstrained convex problem *without loss of optimality*. So with this theory, we can use algorithms like gradient descent to address the convex-constrained problem. This is how we understood why convex optimization is tractable.

We then moved onto one simple instance of convex optimization: Linear Program (LP). The standard form reads:

$$\begin{aligned} \min w^T x : Ax - b \leq 0, \\ Cx - e = 0 \end{aligned} \quad (1.43)$$

where  $w, A, b, C$  and  $e$  are of compatible size and the inequality is component-wise one. At the end, we claimed that many interesting and important problems can be translated into LPs.

**Outline** The goal of this section is to defend the claim. To this end, we will study two prominent and historical examples which can be formulated as LPs. The first is a resource allocation problem which is also called an *optimal planning problem*. This has been the most important problem in economics and operation research in the 20th century. Specifically we will investigate a historical problem (explored by the father of LP, Leonid Kantorovich ([Gardner, 1990](#))), which later gave inspirations to the development of LP. The second is a transportation problem which has been playing a crucial role in a variety of fields for centuries. In particular we will study a problem explored by the farther of Transportation Theory, Gaspard

	wood 1	wood 2
machine 1	10 units/time	10
machine 2	20	40

Figure 1.11. Machine capabilities of peeling woods.

Monge, a French mathematician in the 18th century (Monge, 1781). While investigating these examples, we will learn a couple of techniques that serve to formulate an LP: (i) how to express conditions (given in a problem) in terms of vector and matrix notations; (ii) how to set up a proper optimization variable that yields an LP formulation; and (iii) how to translate a convex function into an affine function that arises in the standard form of LP.

**Kantorovich's plywood cutting problem (Gardner, 1990)** One of the problems that Kantorovich considered is the *plywood cutting problem*. Kantorovich encountered the problem in 1937 while interacting with plywood engineers. For simplicity, we consider a much simpler version of the original problem.

The problem is about allocating the time for the use of different machines for peeling different kinds of woods. Suppose there are two kinds of woods to peel, say wood 1 and wood 2. Also there are two different peeling machines: machine 1 and machine 2. Each machine has a different capability of peeling. Machine 1 can peel 10 units/time for either type of wood. On the other hand, machine 2 can peel 20 units/time for wood 1 while peeling 40 units/time for wood 2. See Fig. 1.11.

The goal of the problem is to maximize the total wood production. But there is a constraint. The constraint is that production is desired to meet the equal proportion, i.e., the amount of wood 1 peeled is desired to be the same as that of wood 2. If there is a remnant part which exceeds the equal proportion, then it is simply discarded. So the objective is to maximize the *minimum* of wood 1 and 2 products:

$$\max \min \{ \text{wood 1 product, wood 2 product} \}. \quad (1.44)$$

What is an optimization variable? In other words, what is a quantity that affects the objective function? That is, the time that we use for peeling each wood with a certain machine. Let  $x_1$  be machine 1's time for peeling wood 1. Normalizing the time, we can assume that  $0 \leq x_1 \leq 1$ . Assuming that machines are always operating, machine 1's time for wood 2 would be  $1 - x_1$ . Similarly define  $0 \leq x_2 \leq 1$  as machine 2's time for peeling wood 1. Using these notations together with machine capabilities illustrated in Fig. 1.11, we get: wood 1 product =  $10x_1 + 20x_2$  (units); wood 2 product =  $10(1 - x_1) + 40(1 - x_2)$  (units). Now applying this to (1.44) and flipping the sign of the objective function, we obtain a minimization

problem as follows:

$$\begin{aligned} \min \max \{-10x_1 - 20x_2, 10(x_1 - 1) + 40(x_2 - 1)\} : \\ 0 \leq x_1 \leq 1, 0 \leq x_2 \leq 1. \end{aligned} \quad (1.45)$$

**Translation to an LP** Notice in (1.45) that the objective function  $\max\{\cdot, \cdot\}$  (marked in red) is convex (why? check in Prob 1.6), but it is *not affine*, so it is not an LP. This is exactly where one important technique kicks in. One technique that allows us to convert such a convex function into an affine function is to introduce another variable, say  $x_3$ , such that:

$$x_3 \geq -10x_1 - 20x_2, \quad x_3 \geq 10(x_1 - 1) + 40(x_2 - 1). \quad (1.46)$$

Now consider the following optimization:

$$\begin{aligned} \min x_3 : \\ 0 \leq x_1 \leq 1, \quad 0 \leq x_2 \leq 1, \\ -10x_1 - 20x_2 - x_3 \leq 0, \\ 10x_1 + 40x_2 - x_3 - 50 \leq 0. \end{aligned} \quad (1.47)$$

Here the key observation is that the minimizer, say  $x_3^*$ , is achieved at:

$$x_3^* = \max \{-10x_1 - 20x_2, 10(x_1 - 1) + 40(x_2 - 1)\}. \quad (1.48)$$

Otherwise, i.e., if  $x_3^* > \max \{-10x_1 - 20x_2, 10(x_1 - 1) + 40(x_2 - 1)\}$ , it contradicts with the hypothesis that  $x_3^*$  is the minimizer. Hence, the translated problem (1.47) is equivalent to the original problem (1.45).

Note that the objective function and all of the functions that appear in inequality constraints are affine. Hence, the problem is an LP. Using vector/matrix notations, we can also represent this in the following standard form:

$$\min w^T x : Ax - b \leq 0 \quad (1.49)$$

where:

$$w = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}, A = \begin{bmatrix} -1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 1 & 0 \\ -10 & -20 & -1 \\ 10 & 40 & -1 \end{bmatrix}, b = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \\ 0 \\ 50 \end{bmatrix}. \quad (1.50)$$

The first rows of  $A$  and  $b$  come from  $-x_1 \leq 0$ , and the second rows are due to  $x_1 - 1 \leq 0$ , and so on and so forth.

**Monge's transportation problem (Monge, 1781)** The second problem that we will study is another historical problem: *Monge's problem*, explored by Gaspard Monge, a French mathematician who lived in the 18th century. In 1781, he published a memoir, titled: *Mémoire sur la théorie des déblais et des remblais* (Monge, 1781). In the memoir, he introduced a transportation problem which later laid the foundation of *transportation theory*. In particular, the field of transportation theory was revolutionized in the 20th century by the recognition that the Monge's transportation problem can be translated into an LP. This recognition was made by Kantorovich (Kantorovich, 1960). So here we will figure out how Kantorovich recognized it as an LP.

Monge's problem is about transporting soils (mined in several grounds) into construction sites, each of which demands a certain amount of soils for construction purpose. For instance, let us consider an example illustrated in Fig. 1.12. Suppose there are three grounds (marked in black squares) and four construction sites (marked in hollowed circles). For each ground, a certain amount of soils can be mined. Let  $s_i$  be the amount of soils mined in ground  $i \in \{1, 2, 3\}$ . For simplicity, we assume that  $s_i$ 's are normalized such that  $s_1 + s_2 + s_3 = 1$ . Let  $d_j$  indicate the amount of soils demanded at construction  $j \in \{1, 2, 3, 4\}$ . Assume that the total demand is the same as the total supply. Then,  $d_1 + d_2 + d_3 + d_4 = s_1 + s_2 + s_3 = 1$ .

The goal of the problem is to find an optimal *coupling* such that the *transportation cost* is minimized. To figure out how to achieve this, we first need to understand how the transportation cost is determined. We assume that the cost is proportional to two factors: (i) distance between a ground and a corresponding construction site and (ii) the amount of the soils sent. To quantify the distance, we need to define coordinates of locations of grounds and constructions. Let  $x_i$  and  $y_j$  denote location coordinates of ground  $i$  and construction  $j$ , respectively, where  $i \in \{1, 2, 3\}$

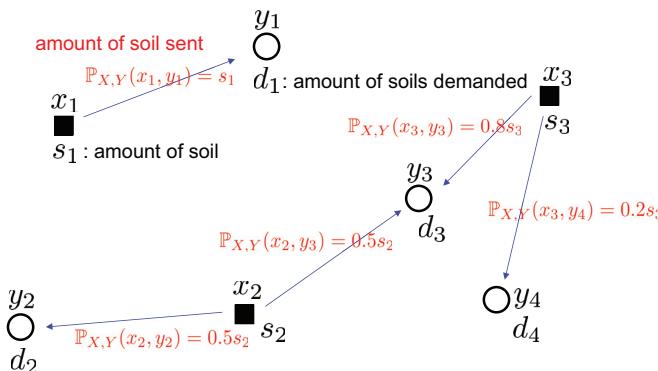


Figure 1.12. A particular coupling in Monge's transportation problem.

and  $j \in \{1, 2, 3, 4\}$ . Then, the distance between ground  $i$  and construction  $j$  can be written as:

$$\text{dist}(\text{ground } i, \text{construction } j) = \|x_i - y_j\| \quad (1.51)$$

where  $\|\cdot\|$  denotes the Euclidean norm.

How to represent the amount of soils delivered from ground  $i$  to construction  $j$ ? For ease of illustration, let us consider a particular *coupling* illustrated in Fig. 1.12. This coupling suggests that the soils mined in ground 1 are transmitted only to construction 1. So the amount of soils must read  $s_1$ . Let's denote that by  $\mathbb{P}_{X,Y}(x_1, y_1) = s_1$ . Later you will see why we use this complicated-looking notation  $\mathbb{P}_{X,Y}(\cdot, \cdot)$ . For ground 2, the soils are split into two construction sites: constructions 2 and 3. Assume the equal split. We can then represent the splitting by  $\mathbb{P}_{X,Y}(x_2, y_2) = 0.5s_2$  and  $\mathbb{P}_{X,Y}(x_2, y_3) = 0.5s_2$ . Similarly for ground 3, the soils are split into constructions 3 and 4, but with an asymmetric split, say 8:2. So we have  $\mathbb{P}_{X,Y}(x_3, y_3) = 0.8s_3$  and  $\mathbb{P}_{X,Y}(x_3, y_4) = 0.2s_3$ . Notice that the soil allocation, determined by the values of  $\mathbb{P}_{X,Y}(x_i, y_j)$ 's, is the one that we can control over. So this is an *optimization variable*. It is a 12-dimensional vector in this example.

Next think about constraints posed in the problem. The constraints are two folded: (i) all the soils mined in each ground should be transmitted to construction sites; and (ii) the demands of all the constructions should be satisfied. In terms of mathematical notations, this means that:

$$\sum_{j=1}^4 \mathbb{P}_{X,Y}(x_i, y_j) = s_i \quad i \in \{1, 2, 3\}, \quad (1.52)$$

$$\sum_{i=1}^3 \mathbb{P}_{X,Y}(x_i, y_j) = d_j \quad j \in \{1, 2, 3, 4\}. \quad (1.53)$$

We can then write down the optimization problem as follows. Given  $(s_i, d_j)$ 's and  $(x_i, y_j)$ 's:

$$\begin{aligned} & \min \sum_{i=1}^3 \sum_{j=1}^4 \underbrace{\mathbb{P}_{X,Y}(x_i, y_j)}_{\text{amount of soils}} \cdot \underbrace{\|x_i - y_j\|}_{\text{distance}} : \\ & \sum_{j=1}^4 \mathbb{P}_{X,Y}(x_i, y_j) = s_i \quad i \in \{1, 2, 3\}, \\ & \sum_{i=1}^3 \mathbb{P}_{X,Y}(x_i, y_j) = d_j \quad j \in \{1, 2, 3, 4\}. \end{aligned} \quad (1.54)$$

The objective function and all the functions that appear in the constraints are *affine* w.r.t. the optimization variable  $\mathbb{P}_{X,Y}(x_i, y_j)$ . Hence, it is an LP.

**Wasserstein distance** If you think about the formula (1.54), we see that one can succinctly represent it by relying upon the concept of *probability distribution*. The succinct form that you will see soon gives an insightful interpretation on the transportation problem. Remember that  $s_i$ 's and  $d_j$ 's are normalized:  $s_1 + s_2 + s_3 = 1$  and  $d_1 + d_2 + d_3 + d_4 = 1$ . So one can view this as a valid probability distribution. For example, defining  $\mathbb{P}_X(x_i) := s_i$ , we see that  $\mathbb{P}_X(x_i)$  is a probability distribution. Similarly,  $\mathbb{P}_Y(y_j) := d_j$  is another valid probability distribution.

Keeping these in our mind, let us take a careful look at the constraints in the above optimization problem (1.54). What do the constraints remind you of? They remind you of the *total probability law*! Hence, one can view  $\mathbb{P}_{X,Y}(x_i, y_j)$  as another valid probability distribution. This probabilistic viewpoint then allows us to simplify the optimization problem (1.54) as follows. Given  $(\mathbb{P}_X, \mathbb{P}_Y)$ ,

$$W(\mathbb{P}_X, \mathbb{P}_Y) := \min_{\mathbb{P}_{X,Y}} \mathbb{E} [\|X - Y\|] \quad (1.55)$$

where the minimization is over all joint distributions  $\mathbb{P}_{X,Y}$  which respect marginals:

$$\begin{aligned} \sum_{y_j} \mathbb{P}_{X,Y}(x_i, y_j) &= \mathbb{P}_X(x_i) \quad \forall x_i, \\ \sum_{x_i} \mathbb{P}_{X,Y}(x_i, y_j) &= \mathbb{P}_Y(y_j) \quad \forall y_j. \end{aligned} \quad (1.56)$$

Here  $W(\mathbb{P}_X, \mathbb{P}_Y)$  is a function of  $\mathbb{P}_X$  and  $\mathbb{P}_Y$ . So one nice interpretation is that it can be viewed as sort of *distance* between the two distributions. Notice that  $\mathbb{P}_X = \mathbb{P}_Y$  gives  $W(\mathbb{P}_X, \mathbb{P}_Y) = 0$ , while distinct marginal distributions yield larger  $W(\cdot, \cdot)$ . This succinct expression (1.55) was recognized by Kantorovich and other person, named Rubinstein. So it is called the *Kantorovich-Rubinstein distance*. The distance measure was generalized later by incorporating an arbitrary  $p$ th-order exponent in  $\|X - Y\|$  (i.e.,  $\|X - Y\|^p$ ) (Vaserstein, 1969). The general measure concerning  $\|X - Y\|^p$  is called the  $p$ th-order Wasserstein distance. So  $W(\mathbb{P}_X, \mathbb{P}_Y)$  is called the 1st-order Wasserstein distance.

It turns out the Wasserstein distance appears in many of the optimal transportation problems as a key measure, thus revolutionizing the field of transportation theory. Very interestingly, the Wasserstein distance played a crucial role in designing a famous machine learning model, called Generative Adversarial Networks (GANs) (Goodfellow et al., 2014), thus leading to the development of Wasserstein

GAN ([Arjovsky et al., 2017](#)) that has been proved powerful in many applications. We will investigate details on this in Part III of this book.

**Look ahead** In the next section, we will study one more example for LP. We will also do what we were planning to do for LP: Studying an LP relaxation technique which turns out to be instrumental in addressing some very difficult problems.

## 1.5 LP: Examples and Relaxation

**Recap** In the previous section, we explored two historical examples that can be translated into LPs, as well as have played a big role in the fields like economics, operation research and transportation:

1. Kantorovich's plywood cutting problem;
2. Monge's transportation problem.

In the process, we learned about a couple of techniques which allow us to translate problems into LPs. We also gained some insights as to how to recognize LPs.

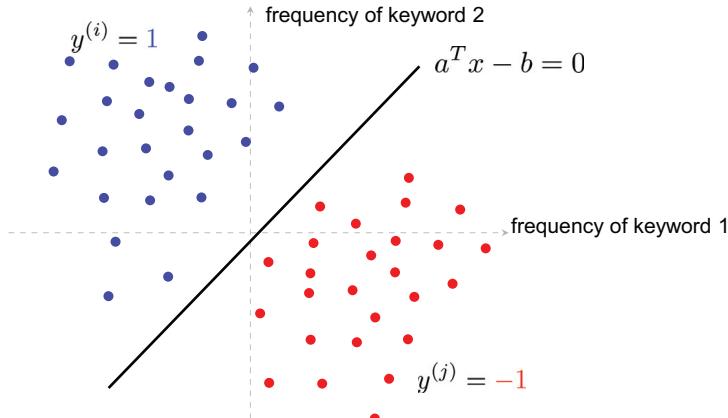
**Outline** In this section, we will study another interesting and prominent example:

3. Classification problem.

This is arguably the most classical problem that arises in machine learning. Next we will deal with another claim that we made in Section 1.3. Some very difficult problems can be solved via LP relaxation. Specifically we will study a canonical example in which the LP relaxation can provide the exact solution in some cases.

**Classification problem** The last example that we will touch upon is a very popular problem that arises in a widening array of fields such as machine learning, artificial intelligence, finance, and real estates. In particular, it is the most classical and canonical problem in machine learning.

For illustrative purpose, let us explain what the problem is under a simple task setting: classifying legitimate emails against spam emails. Suppose there are two datasets. One dataset contains data points (also called *samples* or *examples* in the machine learning field) concerning *spam* emails. The other includes those concerning *legitimate* emails. Assume that each data point is represented by two *features*: (i) frequency of keyword 1 (say, *dollar signs \$\$*); and (ii) frequency of keyword 2 (say, *winner*). In machine learning, the feature is a frequently used terminology which refers to a key component that well describes characteristics of data. Denote each data point by  $x^{(i)} := (x_1^{(i)}, x_2^{(i)})$  where  $x_1^{(i)}$  and  $x_2^{(i)}$  indicate the two features: frequencies of keyword 1 and 2 contained in the  $i$ th email, respectively. See Fig. 1.13 for data points in two datasets, **blue** (legitimate) and **red** (spam) datasets. Here we are also given a *label* which indicates whether data point  $x^{(i)}$  comes from a legitimate email ( $y^{(i)} = 1$ ) or from a spam email ( $y^{(i)} = -1$ ). Assume that we have  $m$  such paired samples.



**Figure 1.13.** Visualization of two datasets for email classification: one for [spam](#) emails; the other for [legitimate](#) emails.

Given these data points together with labels  $\{(x^{(i)}, y^{(i)})\}_{i=1}^m$ , the goal of the classification problem is to find a *line* that separates two datasets. We consider the simplest classification approach, called *linear* classification, which pursues to find a linear function. A line can be represented as a linear equation in the two-dimensional space:  $a^T x - b = 0$ . Note that  $a^T x - b \geq 0$  is a half-space that covers the region above the line (where the blue data points reside) while  $a^T x - b \leq 0$  indicates another half-space spanning the bottom region (where the red data points reside). Hence, in order for a line to separate the two datasets, it must hold that:

$$y^{(i)}(a^T x^{(i)} - b) \geq 0 \quad i \in \{1, \dots, m\}. \quad (1.57)$$

Notice that the *optimization variables* are  $(a, b)$  instead of  $(x^{(i)}, y^{(i)})$ 's. You may be confused about the notations because we have so far used  $x$  notation for the optimization variable. Here we use the  $x$  notation to indicate *data points*, which is a sort of convention in machine learning.

Notice in (1.57) that  $(a, b) = (0, 0)$  always satisfies the constraint. However, it is obviously not of interest, since the trivial solution  $(a, b) = (0, 0)$  makes the classifier play no role. Hence, one may want *strict separability*, meaning that a strict inequality may be preferred in (1.57):

$$y^{(i)}(a^T x^{(i)} - b) > 0 \quad i \in \{1, \dots, m\}. \quad (1.58)$$

However, this is also problematic. Remember the standard form of optimization problems. The inequality constraints should be of “ $\leq$ ”. In fact, the rationale behind the use of this form is related to the strong duality that we will study in depth in

Part II. For now, let us just adopt this form. Given this form, the strict inequality in (1.58) is indeed problematic.

In order to address this, one can introduce a tiny value, say  $\epsilon$ , so that we obtain an inequality constraint like:

$$y^{(i)}(a^T x^{(i)} - b) \geq \epsilon. \quad (1.59)$$

Here the scale of the values that appear in the inequality does not affect finding the optimal solution  $(a^*, b^*)$  that we will seek shortly. Hence, without loss of generality, we consider the normalized version instead:

$$y^{(i)}(a^T x^{(i)} - b) \geq 1 \quad i \in \{1, \dots, m\}. \quad (1.60)$$

Again, whenever we have a strict inequality as in (1.58), one can obtain the inequality like (1.60) by properly scaling  $(a, b)$ .

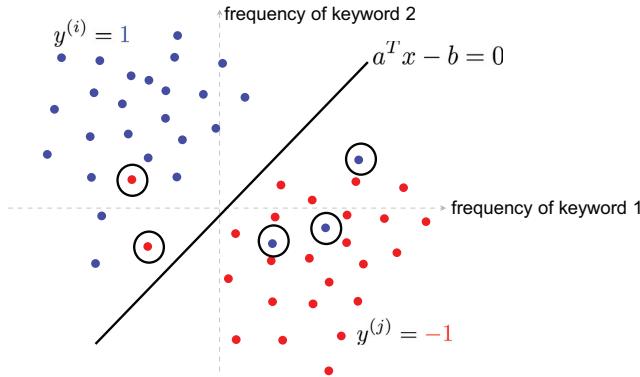
As long as the above constraints are satisfied for all data points  $(x^{(i)}, y^{(i)})$ 's, we are done, meaning that there is nothing to minimize or maximize. One traditional way to represent an optimization problem in such a case is to set an arbitrary yet constant value as an objective function. So one such problem reads: Given  $\{(x^{(i)}, y^{(i)})\}_{i=1}^m$ ,

$$\min_{a,b} 0 : y^{(i)}(a^T x^{(i)} - b) \geq 1 \quad i \in \{1, \dots, m\} \quad (1.61)$$

where the objective function is simply set to 0. Notice that we have nothing to minimize. Observe that all the functions that appear are affine. Hence, it is an LP.

*On a side note:* You may wonder how we can apply an algorithm like *gradient descent*, since we have a *constant objective* with 0 derivative all the time. Notice that this is a *constrained optimization*. Remember the claim that we stated in Section 1.3. A convex constrained optimization can be translated into an unconstrained version thanks to the consequence of strong duality. It turns out the objective function in the unconstrained counterpart is indeed a function of the optimization variable. This point will be clearer in Part II.

**Non-separable case** In the classification setting, one may ask the following natural question. What if datasets are *not linearly separable*, as illustrated in Fig. 1.14? Notice in Fig. 1.14 that some **red** points reside near a cluster of **blue** points, and also some **blue** points are mingled with a majority of **red** points. Obviously there is no line that separates the two datasets. This non-separable case often occurs in various tasks in machine learning. For instance, in the cat-dog classification problem, the boundary that separates cat dataset and dog dataset is highly non-linear.



**Figure 1.14.** Non-separable case of email classification.

One naive<sup>3</sup> yet natural way to handle this non-separable case is to introduce the concept of *margin*. For some outlier data points  $(x^{(i)}, y^{(i)})$ , we introduce margins, say  $v^{(i)} \geq 0$ , such that

$$y^{(i)}(a^T x^{(i)} - b) + v^{(i)} \geq 1 \quad i \in \{1, \dots, m\}. \quad (1.62)$$

Whenever  $y^{(i)}(a^T x^{(i)} - b)$  is strictly less than 1 (which is undesirable), we introduce a positive margin  $v^{(i)}$  so that the sum of them is greater than or equal to 1. Obviously the smaller the margin, the better the situation.

We can then set out our new goal as: minimizing the aggregated margins while respecting the above constraint (1.62). Hence, one can formulate the optimization problem as:

$$\begin{aligned} & \min_{a, b, v^{(i)}} \sum_{i=1}^m v^{(i)} : \\ & y^{(i)}(a^T x^{(i)} - b) + v^{(i)} \geq 1 \quad i \in \{1, \dots, m\}, \\ & v^{(i)} \geq 0 \quad i \in \{1, \dots, m\}. \end{aligned} \quad (1.63)$$

Again this is still an LP.

3. A more powerful yet sophisticated way is to employ *deep neural networks*. During the past decades, there has been a breakthrough in machine learning. It has been shown that *deep neural networks* can well represent any *arbitrary* (possibly highly non-linear) functions with reasonable computational complexity in view of current technologies. So one can use such a network to implement a non-linear classifier to do much better. We will get to this point in depth in Part III. *Neural networks* are systems with one or multiple layers in which each layer consists of an affine operation and an arbitrary (possibly non-linear) operation (called the activation function in the literature). Input and output in each layer are high-dimensional vectors and each component in the vectors is represented as a circle and called a neuron. The naming was originated from the fact that the structure looks like that of brain networks. Deep neural networks refer to the one having at least one hidden layer between input and output layers.

**A class of difficult yet solvable problems** Recall the claim that we made earlier. Some very difficult problems can be solved via LP relaxation. So let us study the technique in the context of such a difficult problem class. One such prominent class is: a class of *boolean problems*, which can be formally stated as below.

$$\begin{aligned} p^* := \min w^T x : \\ Ax - b \leq 0, \quad Cx - e = 0, \\ x_i \in \{0, 1\}, \quad i \in \{1, \dots, d\}. \end{aligned} \tag{1.64}$$

We see the last additional constraint, saying that the optimization variable is constrained to be *boolean*. We often use the following shorthand notation:  $x \in \{0, 1\}^d$ . There are many problems that can be formulated as above in the real world. To get some feeling about the problem nature, let us explore one popular example.

**Example: Shortest path problem (Dijkstra et al., 1959)** The popular problem that we will delve into is a fundamental problem in combinatorial optimization, named the *shortest path problem*. The problem is about finding a *path* from a source to a destination in a graph.

For ease of illustration, let us consider an example in Fig. 1.15. To understand what this problem is, we need to first know about the concept of *graphs*. A graph, denoted by  $\mathcal{G}$ , is a collection of two sets: (i) a vertex (or node) set, denoted by  $\mathcal{V}$ ; and (ii) an edge set, denoted by  $\mathcal{E}$ . The vertex set includes many nodes indicating some objects of interest. The edge set includes many edges indicating some connections between two nodes. Here we have a graph  $\mathcal{G} = (\mathcal{V}, \mathcal{E})$  where:

$$\begin{aligned} \mathcal{V} &= \{1, 2, 3, 4, 5, 6\}; \\ \mathcal{E} &= \{(1, 2), (2, 1), (1, 5), (5, 1), (2, 3), (3, 2), (2, 5), (5, 2), \\ &\quad (3, 4), (4, 3), (4, 5), (5, 4), (4, 6), (6, 4)\}. \end{aligned}$$

We consider a *bi-directed* graph in which the edges are bi-directional. Let node 1 and node 6 be source and destination, respectively. A *path* is defined as a sequence of edges that connects the source to the destination. See an example path in Fig. 1.15, marked in a green line:  $(1, 5) \rightarrow (5, 4) \rightarrow (4, 6)$ . Let  $x_{ij}$  indicate whether the edge

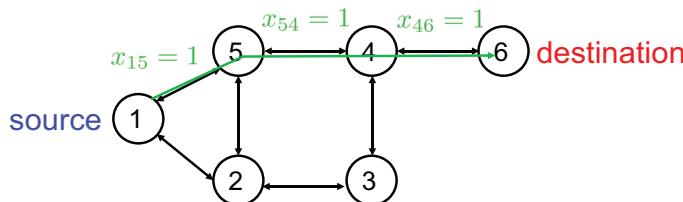


Figure 1.15. Illustration of the shortest path problem.

$(i, j)$  is participated in the path. So in this example,  $x_{15} = x_{54} = x_{46} = 1$  while all others are 0, i.e.,  $x_{51} = 0, x_{12} = 0, \dots, x_{34} = 0, x_{64} = 0$ . Notice that the path has a *direction*. So  $x_{51} = x_{45} = x_{64} = 0$ , since those are reverse to the direction of the path flow.

The goal of the problem is to find a path such that the total cost is minimized. It is assumed that the cost is decided by the weight  $w_{ij}$  (non-negative) that comes with an edge  $(i, j)$ . Hence, the objective can be stated as:

$$\min_{x_{ij} \forall (i,j) \in \mathcal{E}} \sum_{(i,j) \in \mathcal{E}} w_{ij} x_{ij}. \quad (1.65)$$

Here the constraint is that  $x_{ij}$ 's should be set to ensure a valid path, i.e., connecting the source to the destination. Then, how to check whether or not a path is valid? The validation is a bit tricky. But if you think about the nature of the flow of a valid path, you can come up with an easy way to check.

Consider the flow at source node 1 in the example. One key observation is that the flow is just *outgoing*, i.e., there is no ingoing flow. So we should read:

$$\text{outgoing flow} - \text{ingoing flow} = 1$$

$$\iff \sum_{(1,j) \in \mathcal{E}} x_{1j} - \sum_{(j,1) \in \mathcal{E}} x_{j1} = 1 \quad (1.66)$$

where  $\sum_{(1,j) \in \mathcal{E}} x_{1j}$  indicates the entire flow that comes out of source 1 and  $\sum_{(j,1) \in \mathcal{E}} x_{j1}$  denotes the aggregated flow that goes into source 1. On the other hand, at the destination, the situation is reversed:

$$\text{outgoing flow} - \text{ingoing flow} = -1$$

$$\iff \sum_{(6,j) \in \mathcal{E}} x_{6j} - \sum_{(j,6) \in \mathcal{E}} x_{j6} = -1. \quad (1.67)$$

For other node, say  $u$  (neither source nor destination), the flow is *just passing*, meaning that the flow coming in must go out. So we have:

$$\text{outgoing flow} - \text{ingoing flow} = 0$$

$$\iff \sum_{(u,j) \in \mathcal{E}} x_{uj} - \sum_{(j,u) \in \mathcal{E}} x_{ju} = 0. \quad (1.68)$$

Using the above, we can formulate the optimization problem as:

$$\min_{x_{ij} \forall (i,j) \in \mathcal{E}} \sum_{(i,j) \in \mathcal{E}} w_{ij} x_{ij} :$$

$$\begin{aligned}
& \sum_{(\textcolor{blue}{1},j) \in \mathcal{E}} x_{\textcolor{blue}{1}j} - \sum_{(j,\textcolor{blue}{1}) \in \mathcal{E}} x_{j\textcolor{blue}{1}} = \textcolor{blue}{1} \\
& \sum_{(\textcolor{red}{6},j) \in \mathcal{E}} x_{\textcolor{red}{6}j} - \sum_{(j,\textcolor{red}{6}) \in \mathcal{E}} x_{j\textcolor{red}{6}} = -1 \\
& \sum_{(\textcolor{green}{u},j) \in \mathcal{E}} x_{uj} - \sum_{(j,\textcolor{green}{u}) \in \mathcal{E}} x_{j\textcolor{green}{u}} = \textcolor{green}{0} \quad \forall u \in \{2, 3, 4, 5\} \\
x_{ij} & \in \{0, 1\} \quad \forall (i,j) \in \mathcal{E}.
\end{aligned} \tag{1.69}$$

We see that this is a boolean problem (1.64).

**LP relaxation** The boolean problem (1.64) is known to be notoriously difficult in general. In most cases, we need to search over all possible binary choices of  $x$  to figure out the optimal solution. So the complexity scales like  $2^d$ , growing exponentially with the dimension  $d$ .

To deal with such difficult problems, people thought about a way to move forward. One natural way is to just ignore the binary value constraint (the very cause of making the problem intractable). This natural way is indeed *relaxation*. Simply ignoring the binary value constraint in (1.64), we obtain:

$$\begin{aligned}
p_{\text{LP}}^* := \min w^T x : \\
Ax - b \leq 0, \quad Cx - e = 0, \\
0 \leq x_i \leq 1, \quad i \in \{1, \dots, d\}
\end{aligned} \tag{1.70}$$

where  $x_i$  is *relaxed* to be any real value  $\in [0, 1]$ .

Since it is a more relaxed problem, we can do better, so in general,

$$p^* \geq p_{\text{LP}}^*. \tag{1.71}$$

Interestingly, it turns out that under some situations of shortest path problems, the optimal solution for the relaxed problem is binary:  $x_{ij, \text{LP}}^* \in \{0, 1\}$ , thus implying that  $p^* = p_{\text{LP}}^*$ . We will not prove this here. Instead you may want to check this numerically via programming tools. If you are interested in further details, you may want to take a graph theory course offered in math and/or computer science departments.

**Look ahead** So far we have studied the three historical examples that can be formulated as LPs, as well as some boolean problems which can be solved via LP relaxation. In the next section, we will cover one remaining part among what we were planning to do for LP. That is, investigating efficient algorithms.

## 1.6 LP: Algorithms

---

**Recap** During the past two sections, we have studied three important problems that can be formulated as LPs: (1) Kantorovich's plywood cutting problem ([Gardner, 1990](#)); (2) Monge's transportation problem ([Monge, 1781](#)); and (3) the linear classification problem. We also investigated a class of difficult problems which can be however solved via LP relaxation: Boolean problems. As an example, we explored the shortest path problem ([Dijkstra et al., 1959](#)).

**Outline** In this section, we are going to study an efficient algorithm for LP. We will focus on one particular algorithm, called the *simplex algorithm*. Specifically what we are going to do are three folded. There are a couple of historical and famous algorithms for LP. First off, let us provide the rationale behind why the simplex algorithm is picked up among the algorithms. We will then investigate the standard form that the algorithm relies upon. Lastly we will study how the algorithm works in detail. The algorithm is intuitive and beautiful. You will check this soon.

**Algorithms for LP** Three major algorithms have been recognized for LP. The first is obviously the one that the father of LP, Kantorovich, developed. The second is a very famous and faster algorithm, called the *simplex algorithm* ([Dantzig, 1951](#)). The algorithm was developed in 1947 by an American mathematician, named George Dantzig. Actually some scientists and mathematicians in the West, especially at Berkeley (where Dantzig obtained PhD) and Stanford (where he was a Professor), claimed that the inventor of LP is Dantzig. The claim was based on the fact that the simplex algorithm is the first one that solves any LP in a finite number of steps (which was not revealed by Kantorovich) as well as the fact that the naming of LP was first used in print by Dantzig. However, many people did not accept the claim, and perhaps more importantly, the Nobel Prize committee was silent on this. Kantorovich's contribution for the Nobel Prize was actually on the *optimal allocation of scarce resources*, which is not the invention of LP although highly related ([Kantorovich, 1989](#)). If the committee had wanted to award a prize for LP, then Dantzig should have been included. The last algorithm is a very generic algorithm, called the *interior point method* ([Dikin, 1967](#); [Wright, 2005](#)), which can be applied to general convex optimization problems not limited to LP. The algorithm is based on strong duality. Since strong duality will be covered later in Part II, we will study the algorithm around at the time. This is the reason that the simplex algorithm is picked up for the focus of this section.

**Standard form for the simplex algorithm** Remember the standard form of LP:

$$\min w^T x : Ax - b \leq 0, Cx - e = 0. \quad (1.72)$$

The simplex algorithm relies upon a differently-looking yet equivalent form, called the standard form for the simplex algorithm:

$$\begin{aligned} \max w^T x : & Ax \leq b, \\ & x \geq 0. \end{aligned} \quad (1.73)$$

Later you will see why this form (1.73) helps run the algorithm. One can readily see that (1.73) belongs to the class of (1.72). It turns out the other direction is also true, i.e., any form like (1.72) can always be converted into the form like (1.73).

**How to convert (1.72) into the standard form (1.73)?** To show the conversion from (1.72) to (1.73), we need to demonstrate four things. The first is to convert min to max. This can be done very easily by flipping the sign of the objective function. The second is to convert the equality constraint into inequality one(s). This is also immediate because:

$$Cx - e = 0 \iff Cx \leq e, \quad Cx \geq e.$$

The last is to ensure that all the optimization variables are non-negative, i.e.,  $x \geq 0$ . This can be done in the following manipulation. Suppose there is no sign constraint on a variable, say  $x_1$ . Then, by introducing two new non-negative variables, say  $x_2, x_3 \geq 0$ , we can cover the case by setting:

$$x_1 = x_2 - x_3, \quad x_2, x_3 \geq 0.$$

Here one important note is that using the equality  $x_1 = x_2 - x_3$ , we should replace all  $x_1$ 's (that appear in other constraints if any) with  $x_2 - x_3$ , so that there is no  $x_1$  in the final form.

**Simplex algorithm: Conversion into the slack form** In fact, the precise description of the algorithm is complicated although the idea is simple and insightful. So we will focus only on grasping the key idea through the following example:

$$\begin{aligned} \max & 5x_1 + 4x_2 : \\ & 3x_1 + 5x_2 \leq 78 \\ & 4x_1 + x_2 \leq 36 \\ & x_1, x_2 \geq 0. \end{aligned} \quad (1.74)$$

The algorithm starts with conversion into another form, called the *slack form*. In the slack form, two types of new variables are introduced. One is the target variable, usually denoted by  $z$ , which indicates the objective function itself:

$$\textcolor{red}{z} = 5x_1 + 4x_2.$$

The others are the slack variables, usually denoted by  $s_i$ 's. These are used for making the inequality constraints equality ones. For instance,  $3x_1 + 5x_2 \leq 78$  can be equivalently written as:

$$3x_1 + 5x_2 + \textcolor{blue}{s}_1 = 78, \quad \textcolor{blue}{s}_1 \geq 0.$$

You may wonder why we convert back into equality constraints, since we had already converted the equality constraints in (1.72) into inequality constraints as in (1.73). The reason is relevant to a certain condition (to be described in the sequel) that the translated equality constraints should respect.

To see this clearly, we re-write (1.74) as follows with the target and slack variables:

$$\max z : \tag{1.75}$$

$$z - 5x_1 - 4x_2 = 0 \tag{1.76}$$

$$3x_1 + 5x_2 + \textcolor{green}{s}_1 = \textcolor{green}{78} \tag{1.77}$$

$$4x_1 + x_2 + \textcolor{green}{s}_2 = \textcolor{green}{36} \tag{1.78}$$

$$x_1, x_2, s_1, s_2 \geq 0. \tag{1.79}$$

In the slack form, we see that the right hand side (RHS) terms in the translated equality constraints are *non-negative*, as marked in green in the above. Here the RHS being non-negative is the certain condition that the translated equality constraints should satisfy. Actually we could obtain such non-negative RHS terms by going through the two-step conversion: (i) converting (1.72) into (1.73); and then (ii) converting inequality constraints back into equality ones with slack variables.

However, if you think about it, you may see that the RHS in the translated equality constraint is not always guaranteed to be *non-negative*. For example, suppose we have the following inequality instead:

$$-2x_1 - 4x_2 \leq \textcolor{red}{-34}.$$

In this case, if we naively apply the slack-variable trick that we did earlier, then we would get  $-34$  in the RHS, violating the certain condition. Hence, to avoid this, we should take a slightly different slack-variable trick. We first flip the sign of both sides to obtain:

$$2x_1 + 4x_2 \geq \textcolor{red}{34}.$$

We then *subtract* a non-negative slack variable, say  $s_1$ , so as to obtain:

$$2x_1 + 4x_2 - s_1 = 34.$$

This way, we can ensure that all the RHSs are non-negative while  $s_1$  being non-negative.

**Simplex algorithm: The overall procedure** The simplex algorithm is an iterative algorithm. For each iteration, we do the following:

1. Start with an initial feasible solution;
2. Perturb the solution along a direction that maximizes the target variable  $z$ .

Once we obtain a newly perturbed solution, we set it as another initial point in the next iteration, and again do perturbation along a new  $z$ -maximizing direction in view of the new initial point. We repeat this procedure until any perturbation does not increase  $z$  further.

**Iteration 1** How to set up an initial solution? There could be many ways. One natural way comes from a particular form of the two equality constraints that involve the two slack variables: (1.77) and (1.78). Notice that  $s_1$  appears only in (1.77), similarly  $s_2$  appears only in (1.78); on the other hand,  $(x_1, x_2)$  appear *both* in the two equations. So one easily-derived feasible point is the one in which we set the both-appearing variables to zero, i.e.,  $x_1 = x_2 = 0$ . This way, one can readily see that the following is a feasible solution:

$$(x_1, x_2, s_1, s_2) = (0, 0, 78, 36). \quad (1.80)$$

Notice in this case that  $z = 0$  due to (1.76).

A natural question arises. Can we do better? To figure this out, we consider the equality constraint (1.76) that includes  $z$  of interest:

$$z = 5x_1 + 4x_2. \quad (1.81)$$

We see that increasing  $x_1$  and/or  $x_2$  (from the initial point  $x_1 = x_2 = 0$ ) yields an increase in  $z$ . So one may wonder which direction leads to maximizing  $z$ ? There are three possible options that one can think of: (i) increasing  $x_1$  only while maintaining  $x_2 = 0$ , i.e.,  $(x_1, x_2) = (\delta, 0)$  where  $\delta \geq 0$ ; (ii) the other way around, i.e.,  $(x_1, x_2) = (0, \delta)$ ; or (iii) increasing both  $x_1$  and  $x_2$ , i.e.,  $(x_1, x_2) = (\delta_1, \delta_2)$  where  $\delta_i \geq 0$ . The simplex algorithm takes only the first two options. You may wonder why the last option is ignored – the reason will be explored in depth in Prob 2.6.

The first option seems the  $z$ -maximizing direction because the slope 5 placed in front of  $x_1$  is larger than the slope 4 in front of  $x_2$  in (1.81). However, it is not that clear if taking that direction is indeed the best way to go. The reason is that the maximum values of  $\delta$  that we can push through can be *different* across distinct

directions. Notice that we still need to respect the feasible region induced by the constraints while searching for the maximum values of  $\delta$ . So we need to investigate the two options carefully.

First consider  $(x_1, x_2) = (\delta, 0)$ . The constraints of (1.77) and (1.78) then give:  $s_1 = 78 - 3\delta \geq 0$  and  $s_2 = 36 - 4\delta \geq 0$ , which in turn yields:  $\delta \leq \min\{26, 9\} = 9$ . So  $x_1$  can be maximally set to 9 where  $z = 45$ . On the other hand,  $(x_1, x_2) = (0, \delta)$  gives:  $s_1 = 78 - 5\delta \geq 0$  and  $s_2 = 36 - \delta \geq 0$ . Hence,  $\delta \leq \min\{\frac{78}{5}, 36\} = \frac{78}{5}$ , which yields  $z = 62.4$ . So from this, we see that the second option is better, although the slope 4 is smaller than the other slope 5. This naturally motivates us to choose the following feasible point:

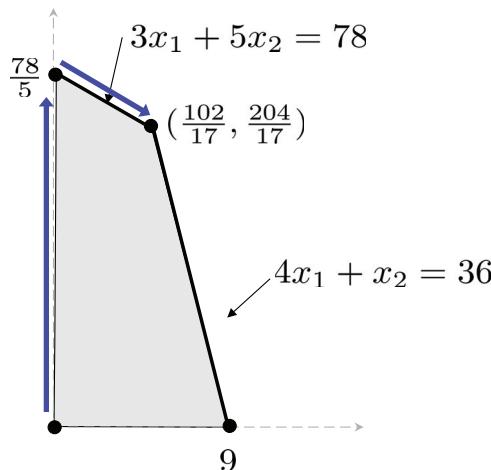
$$(x_1, \textcolor{blue}{x}_2, s_1, s_2) = \left(0, \frac{78}{5}, 0, \frac{102}{5}\right) \quad (1.82)$$

where  $s_1$  and  $s_2$  are set according to the constraints of (1.77) and (1.78). The geometric picture for this is illustrated in Fig. 1.16. We move from  $(x_1, x_2) = (0, 0)$  to  $(x_1, x_2) = (0, \frac{78}{5})$ .

**Iteration 2** We now take the solution (1.82) as an initial feasible solution. The question is: Can we do better than this? To check this, let us ponder (1.82) again:

$$(\textcolor{red}{x}_1, x_2, \textcolor{red}{s}_1, s_2) = \left(0, \frac{78}{5}, 0, \frac{102}{5}\right). \quad (1.83)$$

Remember in the initial feasible point (1.80) that  $(x_1, x_2) = (0, 0)$ . On the other hand, in the second feasible point (1.83), variables that are set to zero are different:



**Figure 1.16.** A geometric insight behind how the simplex algorithm leads to the solution. The beauty of the simple algorithm is that it can obtain the optimal solution although it searches *only along the edges* of the feasible region.

$(x_1, s_1) = (0, 0)$ . Notice that whenever an iteration occurs, we have a new set of two variables being set to zeros. This is because only one variable ( $x_2$  in this example) is perturbed while the other ( $x_1$  in this example) being zero and in the new initial point, one equality is obtained, thus forcing another variable ( $s_1$  in this example) to be zero.

Given that  $(x_1, s_1) = (0, 0)$  in the new initial point, one can now think of two options for perturbation: (i)  $(x_1, s_1) = (\delta, 0)$ ; and (ii)  $(x_1, s_1) = (0, \delta)$ . You may wonder why other variables ( $x_2, s_2$ ) are not taken into consideration for perturbation. Sure, we can include such additional options for perturbation. However, the beauty of the simplex algorithm is that only the restricted perturbation w.r.t. the zero-set variables suffices for us to obtain the optimal solution. Of course, it is not crystal clear why that is the case. The proof of the optimality of the simplex algorithm may be helpful for you to obtain a better understanding although we will not cover the proof here.

In order to check which direction is better between the two options, we first need to ponder (1.76) to see how  $x_1$  and  $s_1$  affect  $z$ :

$$z - 5x_1 - 4x_2 = 0. \quad (1.84)$$

But there is a problem here. The problem is that it is difficult to see how  $s_1$  affect  $z$ , since  $s_1$  does not appear in the equation (1.84).

A very famous technique, called the *Gaussian elimination*, helps us to see the effect of  $s_1$  upon  $z$ . Massaging (1.76) and (1.77) properly, we can cancel  $x_2$  out to obtain:

$$\begin{aligned} & 5 \times [z - 5x_1 - 4x_2 = 0] \\ + & 4 \times [3x_1 + 5x_2 + s_1 = 78] \\ \hline \rightarrow & 5z - 13x_1 + 4s_1 = 312. \end{aligned} \quad (1.85)$$

This then immediately rules out the second option:  $(x_1, s_1) = (0, \delta)$ . Why? We see that increasing  $s_1$  yields a *decrease* in  $z$ . So taking the first option  $(x_1, s_1) = (\delta, 0)$  is the right way to go. Now the question is: How maximally can we set  $\delta$ ? To check this, let us ponder the constraints (1.77) and (1.78) again:

$$3x_1 + 5x_2 + s_1 = 78; \quad (1.86)$$

$$4x_1 + x_2 + s_2 = 36. \quad (1.87)$$

Here (1.86) looks okay because we can immediately see how  $x_2$  is changed depending on  $(x_1, s_1)$  and this helps us to easily identify the limit of  $\delta$ . On the other hand, the form like (1.87) is not desirable because the form does not allow us to directly

see how  $s_2$  is changed depending on  $(x_1, s_1)$ . Hence, it is not that simple to identify the limit of  $\delta$ . Actually the following form is preferred instead:  $?x_1 + ?s_1 + ?s_2 = ?$ . Again the *Gaussian elimination* helps us to obtain the form:

$$\begin{aligned} & 5 \times [4x_1 + x_2 + s_2 = 36] \\ - & \quad [3x_1 + 5x_2 + s_1 = 78] \\ \hline \rightarrow & \quad 17x_1 - s_1 + 5s_2 = 102. \end{aligned} \tag{1.88}$$

With (1.88) and (1.85), we can re-write the optimization problem as:

$$\max z : \tag{1.89}$$

$$5z - 13x_1 + 4s_1 = 312 \tag{1.90}$$

$$3x_1 + 5x_2 + s_1 = 78 \tag{1.91}$$

$$17x_1 - s_1 + 5s_2 = 102 \tag{1.92}$$

$$x_1, x_2, s_1, s_2 \geq 0. \tag{1.93}$$

Remember that our second feasible point was:

$$(x_1, x_2, s_1, s_2) = \left(0, \frac{78}{5}, 0, \frac{102}{5}\right). \tag{1.94}$$

As mentioned earlier, we can rule out the second option  $(x_1, s_1) = (0, \delta)$ . So taking the first option  $(x_1, s_1) = (\delta, 0)$ , we get:  $5x_2 = 78 - 3\delta \geq 0$  and  $5s_2 = 102 - 17\delta \geq 0$ , which then yields:  $\delta \leq \min\{26, \frac{102}{17}\} = \frac{102}{17}$ . Hence, we obtain:  $z = 78$ . Since  $z = 78$  is strictly larger than  $z = 62.4$  (obtained under (1.94)), this motivates us to choose the following feasible point:

$$(x_1, x_2, s_1, s_2) = \left(\frac{102}{17}, \frac{204}{17}, 0, 0\right) = (6, 12, 0, 0) \tag{1.95}$$

where  $(x_2, s_2)$  are set according to (1.91) and (1.92). The geometric picture for this is illustrated in Fig. 1.16. We move from  $(x_1, x_2) = (0, \frac{78}{5})$  to  $(x_1, x_2) = (\frac{102}{17}, \frac{204}{17})$ .

**Iteration 3** Can we do better? To check this, again ponder (1.95). We now have the following two options for perturbation: (i)  $(s_1, s_2) = (\delta, 0)$ ; and (ii)  $(s_1, s_2) = (0, \delta)$ , due to  $(s_1, s_2) = (0, 0)$ . To check which direction is better, again consider (1.90) to see how  $(s_1, s_2)$  affect  $z$ :  $5z - 13x_1 + 4s_1 = 312$ . Here it is difficult to see how  $s_2$  affects  $z$ . Again use the Gaussian elimination to obtain the

following where one can see the effect immediately:

$$\begin{aligned}
 & 17 \times [5z - 13x_1 + 4s_1 = 312] \\
 - & 13 \times [17x_1 - s_1 + 5s_2 = 102] \\
 \hline
 \rightarrow & 85z + 55s_1 + 65s_2 = 6630. \tag{1.96}
 \end{aligned}$$

We see from (1.96) that increasing  $(s_1, s_2)$  yields a *decrease* in  $z$ , meaning that any perturbation does not increase  $z$  further. Hence, we stop here, obtaining:

$$(x_1^*, x_2^*) = (6, 12) \implies z^* = 78. \tag{1.97}$$

This is how the simplex algorithm works. We stop when increasing such zero-variables does not increase  $z$ . It turns out this way of iteration enables us to achieve the optimal solution in a finite number of steps. In many practical applications, it has been shown that the finite number of steps required is much less than the total number of vertices in the polytope formed by the constraints, meaning that the simplex algorithm arrives at the optimal point *very fast*.

**Look ahead** In the next section, we will study how to solve LPs using a particular yet useful programming tool: CVXPY. We will then move onto the second instance of convex optimization problems: *Least Squares*.

## 1.7 LP: CVXPY Implementation

**Recap** So far we have studied several stuffs: (1) the concept of convex optimization; (2) typical categorization of convex instances; (3) why convex optimization problems are tractable (efficiently solvable on a computer); (4) a bunch of historical and classical examples that can be translated into LPs or that can be solved via LP relaxation; and (5) the simplex algorithm for LPs. However, there is one content that we missed for LP. That is, how to implement the algorithm via programming tools such as CVXPY that runs in a widely-used open source platform, Python.

**Outline** In this section, we are going to cover the missing stuff: studying how to implement several interested problems via CVXPY. Specifically what we are going to do are four folded. First off, we will learn how to install CVXPY in Python. We will then investigate basic CVXPY syntaxes via a simple example. Next we will do some exercises for code implementation to get familiar with CVXPY. We will do this in the context of the two prior examples: (i) Kantorovich's plywood cutting problem (see (1.47) in Section 1.4); and (ii) the toy example introduced in the course of explaining the simplex algorithm (see (1.74) in Section 1.6).

**CVXPY in Python** There are two popular software tools depending on platforms that we use: (1) CVX (running in MATLAB); (2) CVXPY (running in Python). While MATLAB is much more user-friendly and hence much easier to use, it requires a license. So we will use a free software, CVXPY.

CVXPY is a library running in a recent version of Python, Version 3. So you should first install Version 3 by downloading it from <https://www.python.org/downloads/>. Or you can use it via virtual environment tools like Anaconda:

<https://www.anaconda.com/products/individual>

For more details, refer to a Python tutorial in Appendix A.

In order to install CVXPY, you should rely upon the library manager called pip. Here is a command for installation:

```
pip install cvxpy
```

To check the list of installed libraries, you can type:

```
pip list
```

If you have difficulties during installation, please refer to

[https://www\\_cvxpy.org/install/index.html](https://www_cvxpy.org/install/index.html)

**How to use CVXPY?** To give you a rough idea as to how to use CVXPY, let us give you a CVXPY script for a simple example:

$$\begin{aligned} \min_x (x_1 - 3x_2)^2 : \\ x_1 - x_2 &\leq 3, \\ x_1 + x_2 &= 10. \end{aligned} \tag{1.98}$$

With some straightforward yet tedious calculation, you can easily figure out that the optimal solution is:  $(x_1^*, x_2^*) = (6.5, 3.5)$ . Let's do sanity check with the following CVXPY script:

```
import cvxpy as cp
# optimization variable
x1 = cp.Variable()
x2 = cp.Variable()
# constraints
constraints = [x1-x2<=3, x1+x2==10]
# objective function
obj_min = cp.Minimize((x1-3*x2)**2)
# set up a problem
prob = cp.Problem(obj_min, constraints)
# solve the problem
prob.solve()
# print the solution
print('status:', prob.status)
print('optimal value:', prob.value)
print('optimal variables:', x1.value, x2.value)
```

Here the blank parenthesis in `cp.Variable()` is for generating a *scalar* optimization variable. To create a vector with a higher dimension, say  $d$ , one can put the dimension inside the parenthesis like `cp.Variable(d)`. Sometimes, an optimization variable is of a matrix form in particular in the case of Semi-Definite Program (SDP) that we will study in Section 1.13. In such a case, we use: `cp.Variable(d1,d2)` where  $(d1,d2)$  are the row and column sizes of an interested matrix. For equality constraints, we use the symbol “ $=$ ” instead of “ $\leq$ ”. The command `cp.Minimize` is for the use of *minimization*, while `cp.Maximize` is employed for maximization. The command `cp.Problem` is an optimization problem object that takes the objective function and constraints as input arguments. The command `prob.solve()` solves the optimization problem and prints the optimal value accordingly. The last three lines are for checking whether the solution is optimal as well as returning

the optimal value together with the optimal solution. In this case, we should read:

```
status: optimal
optimal value: 16.0
optimal variables: 6.5 3.5
```

Notice that the optimal solution is the same as what we calculated by hand.

You may now have a very rough idea as to how to implement CVXPY. Of course, this may not be enough for scripting some problems that you may be interested in. So we do more exercises yet in the context of the two examples that we investigated earlier.

[Exercise 1: Kantorovich's plywood cutting problem \(1.47\)](#) Recall the problem:

$$\begin{aligned} \min x_3 : \\ 0 \leq x_1 \leq 1, \quad 0 \leq x_2 \leq 1, \\ -10x_1 - 20x_2 - x_3 \leq 0, \\ 10x_1 + 40x_2 - x_3 - 50 \leq 0. \end{aligned} \tag{1.99}$$

Using the syntaxes that we learned above, we can readily implement a code as below:

```
import cvxpy as cp
x1 = cp.Variable()
x2 = cp.Variable()
x3 = cp.Variable()

constraints = [x1>=0, x1<=1,
               x2>=0, x2<=1,
               -10*x1-20*x2-x3<=0,
               10*x1+40*x2-x3-50<=0]
obj_min = cp.Minimize(x3)
prob = cp.Problem(obj_min, constraints)
prob.solve()
print('status: ', prob.status)
print('optimal value: ', prob.value)
print('optimal variables: ', x1.value, x2.value, x3.value)
```

```
status: optimal
optimal value: -20.00000000023447
optimal variables: 1.0000000001581 0.4999999998885342
-20.00000000023447
```

Actually, the optimal value is  $-20$  and the optimal variables should read  $(x_1^*, x_2^*, x_3^*) = (1, 0.5, -20)$ . So there are some minor distinctions in the values relative to the above numerical solutions. This is because CVXPY runs on a computer via an algorithm, incurring some numerical distinctions.

**Exercise 2: The toy example used for the simplex algorithm** Recall the toy example:

$$\begin{aligned} \max & 5x_1 + 4x_2 : \\ & 3x_1 + 5x_2 \leq 78 \\ & 4x_1 + x_2 \leq 36 \\ & x_1, x_2 \geq 0. \end{aligned} \tag{1.100}$$

The code implementation is also very simple. See below.

```
import cvxpy as cp
x1 = cp.Variable()
x2 = cp.Variable()
constraints = [3*x1+5*x2<=78,
               4*x1+x2<=36,
               x1>=0, x2>=0]
obj_max = cp.Maximize(5*x1 + 4*x2)
prob = cp.Problem(obj_max, constraints)
prob.solve()
print('status: ', prob.status)
print('optimal value: ', prob.value)
print('optimal variables: ', x1.value, x2.value)
```

```
status: optimal
optimal value: 77.99999988422522
optimal variables: 5.99999990155606 11.99999983361798
```

Similarly, we see some very minor numerical differences, compared to the optimal value  $78$  and the optimal variables  $(x_1^*, x_2^*) = (6, 12)$ .

You may want to exercise more. Don't worry. We have prepared a couple of exercise problems in upcoming problem sets. So you will have some chances to do more.

**Look ahead** We are now done with all the contents in LP. In the next section, we are going to move onto the second instance of convex optimization: *Least Squares* (*LS*). Specifically what we are going to do for LS are three folded. First we will review what the LS problem is. We will then present a geometric insight which can help us to understand what the LS solution means and therefore why the solution makes sense in light of our intuition. Lastly we will study one very important application in machine learning: the classification problem.

## Problem Set 2

---

**Prob 2.1 (Total probability law)** Consider discrete random variables  $X \in \mathcal{X}$  and  $Y \in \mathcal{Y}$  with probability distributions  $\mathbb{P}_X$  and  $\mathbb{P}_Y$ , respectively.

- (a) State the total probability law, and prove it.
- (b) Suppose  $\mathbb{P}_{X,Y}$  is a joint distribution for  $(X, Y)$ . Show that

$$\sum_{y \in \mathcal{Y}} \mathbb{P}_{X,Y}(x, y) = \mathbb{P}_X(x) \quad \forall x \in \mathcal{X}. \quad (1.101)$$

- (c) Suppose (1.101) holds. Show that  $\mathbb{P}_{X,Y}$  satisfies one of probability axioms:

$$\sum_{x \in \mathcal{X}} \sum_{y \in \mathcal{Y}} \mathbb{P}_{X,Y}(x, y) = 1.$$

**Prob 2.2 (1st-order Wasserstein distance)** Let  $X \in \mathcal{X} := \{-2, 0, 2\}$  be a discrete random variable with probability distribution:  $\mathbb{P}_X(x) = \frac{1}{4}, \frac{1}{2}, \frac{1}{4}$  for  $x = -2, 0, 2$ , respectively. Let  $Y \in \mathcal{Y} := \{-4, -1, 1, 4\}$  be another discrete random variable with:  $\mathbb{P}_Y(y) = \frac{3}{8}, \frac{1}{8}, \frac{1}{8}, \frac{3}{8}$  for  $y = -4, -1, 1, 4$ , respectively. Consider Monge's problem which can be formulated as follows. Given  $\mathbb{P}_X$  and  $\mathbb{P}_Y$ ,

$$W(\mathbb{P}_X, \mathbb{P}_Y) := \min_{\mathbb{P}_{X,Y}} \mathbb{E} [\|X - Y\|] \quad (1.102)$$

where the minimization is over all joint distributions  $\mathbb{P}_{X,Y}$  respecting the marginals  $\mathbb{P}_X$  and  $\mathbb{P}_Y$ :

$$\sum_{y \in \mathcal{Y}} \mathbb{P}_{X,Y}(x, y) = \mathbb{P}_X(x) \quad \forall x \in \mathcal{X}; \quad (1.103)$$

$$\sum_{x \in \mathcal{X}} \mathbb{P}_{X,Y}(x, y) = \mathbb{P}_Y(y) \quad \forall y \in \mathcal{Y}. \quad (1.104)$$

- (a) Translate the above optimization problem into an LP in *standard form*.
- (b) Solve the optimization problem using CVXPY. Also write a script for CVXPY implementation.

**Prob 2.3 (Linear classification)** Consider the linear classification problem wherein the goal is to find a boundary of the line form that can distinguish legitimate emails from spams. We are given  $\{(x^{(i)}, y^{(i)})\}_{i=1}^m$  training dataset placed in the file “train.csv”. The file is uploaded on

Here  $x^{(i)} := (x_1^{(i)}, x_2^{(i)})$  indicates the  $i$ th example and  $y^{(i)}$  denotes its corresponding label (legitimate = +1; spam = -1).

(a) What are  $m$  and dimension of  $x^{(i)}$ ?

(b) Use Python to visualize the data points in the two-dimensional space.

*Hint: You may want to use a Python package matplotlib.pyplot. For how to use it, refer to Appendix A.2.*

(c) Formulate an optimization problem for the linear classifier. Solve the problem using CVXPY. Also write a script for CVXPY implementation.

**Prob 2.4 (Bipartite matching problem)** Consider the bipartite matching problem that assigns three people to three tasks in an one-to-one fashion. The matching costs are given as:

$$(w_{11}, w_{12}, w_{13}) = (1, 2, 4)$$

$$(w_{21}, w_{22}, w_{23}) = (1, 4, 2)$$

$$(w_{31}, w_{32}, w_{33}) = (1, 3, 3)$$

where  $w_{ij}$  indicates the cost w.r.t. the assignment of task  $j$  to person  $i$ . The objective function is:

$$\min_{x_{ij}} \sum_{i=1}^N \sum_{j=1}^N w_{ij} x_{ij} \quad (1.105)$$

where  $x_{ij} \in \{0, 1\}$  indicates whether person  $i$  is assigned to task  $j$  and  $N = 4$  is the number of people (or tasks). Here the constraint is that each person must be assigned, and so each task must be:

$$\sum_{j=1}^N x_{ij} = 1 \quad \forall i \in \{1, \dots, N\} \text{ (each person must be assigned to one task);} \quad (1.106)$$

$$\sum_{i=1}^N x_{ij} = 1 \quad \forall j \in \{1, \dots, N\} \text{ (each task must be assigned to one person).} \quad (1.107)$$

- (a) Formulate a Boolean optimization problem. Derive the optimal value  $p^*$  and the optimal solution  $x^*$ . You can do this by hand or by computer.
- (b) Formulate an LP relaxation problem. Solve this problem (deriving  $p_{LP}^*$  and  $x_{LP}^*$ ) using CVXPY. Also write a script for CVXPY implementation. Is  $p_{LP}^* = p^*$ ?

**Prob 2.5 (Standard and slack forms for the simplex algorithm)** Consider the following LP:

$$\min x_1 + x_2 : 2x_1 + x_2 \geq 1$$

$$x_1 + 3x_2 \geq 1$$

$$x_1 \geq 0, x_2 \geq 0.$$

- (a) Convert this into the *standard form* for the simplex algorithm.
- (b) Convert the standard form (derived in part (a)) into the *slack form*.

**Prob 2.6 (Simplex algorithm:  $z$ -maximizing direction)** Consider the example that we investigated in Section 1.6:

$$\max z : \quad (1.108)$$

$$z - 5x_1 - 4x_2 = 0 \quad (1.109)$$

$$3x_1 + 5x_2 + s_1 = 78 \quad (1.110)$$

$$4x_1 + x_2 + s_2 = 36 \quad (1.111)$$

$$x_1, x_2, s_1, s_2 \geq 0. \quad (1.112)$$

Suppose we set an initial feasible point as:

$$(x_1, x_2, s_1, s_2) = (0, 0, 78, 36). \quad (1.113)$$

Consider three possible options for a direction along which we perturb the initial point:

- (i)  $(x_1, x_2) = (\delta, 0);$
- (ii)  $(x_1, x_2) = (0, \delta);$
- (iii)  $(x_1, x_2) = (\delta_1, \delta_2)$

where  $\delta, \delta_1, \delta_2 \geq 0$ .

- (a) Suppose that  $\delta = \delta_1 + \delta_2 = 1$ . Which is the  $z$ -maximizing direction?
- (b) Suppose that  $(\delta, \delta_1, \delta_2)$  can be chosen *arbitrarily* subject to the constraints. Is  $(\delta_1, \delta_2) = (\frac{102}{17}, \frac{204}{17})$  a valid choice, i.e., respecting the constraints? If so, what is  $z$  under the choice? What is the  $z$ -maximizing direction in this case?
- (c) When taking the third option, discuss whether it is easy to find the best choice of  $(\delta_1, \delta_2)$  (in a sense of maximizing  $z$ ) relative to finding the best  $\delta$  in the first (or second) option. What if we have more constraints that come with more slack variables?

**Prob 2.7 (Simplex algorithm: Exercise 1)** Consider the following LP:

$$\begin{aligned} \max 4x_1 + 3x_2 : \quad & 2x_1 + 3x_2 \leq 6 \\ & -3x_1 + 2x_2 \leq 3 \\ & 2x_2 \leq 5 \\ & 2x_1 + x_2 \leq 4 \\ & x_1 \geq 0, x_2 \geq 0. \end{aligned}$$

- (a) Solve this using the *simplex algorithm* that we learned in Section 1.6. You should do it by hand.
- (b) Solve this using CVXPY to do sanity check for your solution in part (a).

**Prob 2.8 (Simplex algorithm: Exercise 2)** Consider the following LP:

$$\begin{aligned} \min 3x_1 + 9x_2 : \quad & 2x_1 + x_2 \geq 8; \\ & x_1 - 2x_2 \geq -1; \\ & x_1 \geq 0, x_2 \geq 0. \end{aligned} \tag{1.114}$$

- (a) Convert the problem into the slack form.
- (b) Use the simplex algorithm to solve this problem. Show all the detailed procedures as per the rule that we learned in Section 1.6.
- (c) Write a CVXPY script for solving the above optimization (1.114).

**Prob 2.9 (True or False?)**

- (a) Consider Monge's problem that we studied in Section 1.4. Let  $s_i \geq 0$  be the amount of soils mined in ground  $i \in \{1, 2, \dots, m\}$ . Let  $d_j \geq 0$  be the amount of soils demanded at construction site  $j \in \{1, 2, \dots, n\}$ . In Section 1.4, assuming that  $\sum_{i=1}^m s_i = \sum_{j=1}^n d_j = 1$ , it was shown that the problem can be formulated as an LP. In fact, without the assumption on  $s_i$ 's and  $d_j$ 's, the problem can also be formulated as an LP.
- (b) The LP relaxation technique yields the exact solution for the shortest path problem.

## 1.8 Least Squares (LS)

---

**Recap** So far we have studied many contents ranging from the concept of convex optimization to all the details for LP including CVXPY implementation.

**Outline** In this section, we will move onto the second instance of convex optimization: *Least Squares (LS)*. Specifically we will cover three stuffs. First we will review what the LS problem is. We will then present a geometric insight which helps us to understand what the LS solution means and therefore why the solution makes sense in light of our intuition. Lastly we will study one very important application in machine learning: the classification problem.

**Review: Least-squares problem** Since we studied LS in Section 1.1 (a bit far from here), let us start by reviewing what the problem is. The problem is formulated as:

$$\min \|Ax - b\|^2. \quad (1.115)$$

As mentioned earlier, one of the most important things that we can benefit from this problem is that it has the *closed-form solution*:

$$x^* = (A^T A)^{-1} A^T b. \quad (1.116)$$

In Section 1.1, we just claimed that this is the solution. Now let us prove it. The first thing to notice is that the objective function  $\|Ax - b\|^2$  is *convex*. Actually this was dealt in Prob 1.1(b). Next, remember the optimality condition for  $x^*$  that we learned in Section 1.3 w.r.t. unconstrained optimization. That is,  $x^*$  must be the stationary point. By applying this, we obtain:

$$\nabla \|Ax^* - b\|^2 = 0. \quad (1.117)$$

Now consider the gradient w.r.t.  $x$ :

$$\begin{aligned} \nabla \|Ax - b\|^2 &= \nabla (Ax - b)^T (Ax - b) \\ &= \nabla (x^T A^T Ax - 2x^T A^T b + b^T b) \\ &= 2A^T Ax - 2A^T b \end{aligned}$$

where the last follows from the definition of the gradient w.r.t. a vector. Please exercise Prob 1.2 if you are not convinced with the gradient computation. Applying this to the optimality condition (1.117), we get:

$$x^* = (A^T A)^{-1} A^T b. \quad (1.118)$$

**Dimensions of  $(x, A, b)$**  Let  $d$  be the dimension of the optimization variable  $x$ . Then,  $x \in \mathbf{R}^d$ . Let  $A := [a_1 \ \cdots \ a_d] \in \mathbf{R}^{m \times d}$ . Then,  $b \in \mathbf{R}^m$ . We now have two cases depending on the values of  $m$  and  $d$ .

One is:  $m < d$ , i.e.,  $A$  is a *wide matrix*.<sup>4</sup> Suppose all the row vectors in  $A$  are linearly independent, i.e.,  $\text{rank}(A) = m$ , which usually holds in practice. In this case, we have a larger number  $d$  of unknowns than the number  $m$  of linear equations in  $Ax - b = 0$ . This implies that there are infinitely many solutions that respect  $Ax - b = 0$ . So in this case, the optimal value  $p^* = 0$ . In typical scenarios, the optimization solution  $x^*$  that we seek to find is unique. For instance, consider the astronomy problem that we investigated in Section 1.1, wherein  $x^*$  concerns the ground-truth trajectory of the orbit of Ceres. In such a case,  $x^*$  must be unique. However, if  $m < d$ , there are infinitely many solutions for  $x^*$ . Then, it would be very much unlikely that one solution among infinitely many coincides with the ground-truth trajectory. This is definitely not an interested scenario.

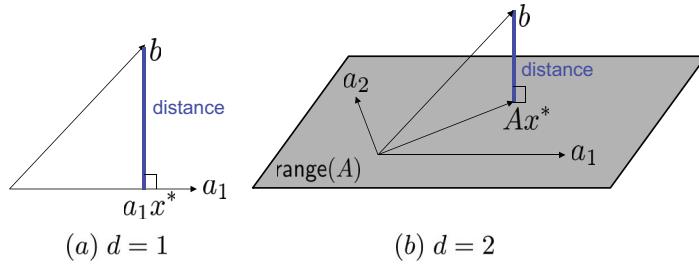
The second case is:  $m \geq d$ , i.e.,  $A$  is a *tall matrix*. Suppose that  $b$  does not lie in the *range space* of  $A$ ,  $\text{range}(A)$  (the space spanned by all the column vectors of  $A$ ). Actually the case  $b \notin \text{range}(A)$  is typical in practice. Again think about the astronomy problem in which  $b$  indicates a collection of observed location coordinates and  $A$  is an observed-location-dependent matrix. Since observations often contain measurement errors subject to random directions, it is very much likely that  $b$  is not in a certain space, say  $\text{range}(A)$ . Hence,  $b \notin \text{range}(A)$ . In the case of  $m \geq d$ , obviously there is *no solution* that satisfies  $Ax - b = 0$ . But what we can say is that it has a solution that minimizes  $\|Ax - b\|^2$  though, and this forms the basic idea behind the least-squares problem. So what we are interested in is the second case:  $m \geq d$ .

**Geometric insight** We present a geometric insight behind the least-squares problem. From this, you will see what the least-squares solution means, as well as why the problem is important accordingly.

Let us first consider the simplest setting in which  $d = 1$ . In this case,  $A$  is simply a single-column vector and  $x$  is a scalar. Suppose we have two vectors  $a_1$  and  $b$ , as illustrated in Fig. 1.17(a), in which  $b$  is not aligned with  $a_1$ . This is due to the assumption that often holds in practice:  $b \notin \text{range}(A)$ . Notice that  $\|a_1x - b\|^2$  is minimized when the vector  $a_1x - b$  (marked in the blue thick line) is *perpendicular* to the direction of  $a_1$ . So from this, one can interpret the least-squares solution

---

4. In fact, a majority of people use the terminology like a *fat matrix* instead. But Prof. Stephen Boyd at Stanford whom I interacted with while being on sabbatical recommended the use of a different terminology: a *wide matrix*. His rationale was that the wide matrix has sort of *positive* nuance, while the fat matrix looks *negative*. In light of his positive attitude, let us use the “wide matrix” terminology.



**Figure 1.17.** Geometric insight behind the least-squares solutions.

as the *distance-minimizing solution*. The distance-minimizing solution is obviously what we want. So it is sort of a good solution which well matches our natural demand.

We can have the same interpretation for a slightly more general case, say  $d = 2$ . In this case,  $A = [a_1 \ a_2] \in \mathbf{R}^{m \times 2}$ . The vector  $Ax$  now lies in the *plane*, which is the range space of  $A$ :  $Ax \in \text{range}(A)$ ; see Fig. 1.17(b). Similarly  $\|Ax - b\|^2$  is minimized when the vector  $Ax - b$  (marked in the blue thick line) is perpendicular to the plane, as illustrated in Fig. 1.17(b). So again one can interpret  $Ax^*$  as the distance-minimizing solution.

**An application: Classification problem** As mentioned earlier, the least-squares problem is a very popular and powerful problem which has played a significant role in the optimization field since the birth of the problem in the 1800s. It has been employed for addressing many important problems that arise in a wide variety of applications.

In this section, we would like to put a particular emphasis on one important application that arises in machine learning as well as that we have already investigated earlier: the *classification problem*.

Remember the classification example that we studied in Section 1.5: legitimate-vs-spam emails classification, in which we are given  $m$  data points  $\{(x^{(i)}, y^{(i)})\}_{i=1}^m$ . Here  $x^{(i)}$  indicates a *feature vector*. In the example, we considered a two-dimensional case where  $x^{(i)} := (x_1^{(i)}, x_2^{(i)})$  and  $(x_1^{(i)}, x_2^{(i)})$  denote the frequencies of keywords 1 and 2 that appear in the  $i$ th email, respectively. Here  $y^{(i)}$  is a *label*, indicating an identity of the  $i$ th email:  $y^{(i)} = +1$  (legitimate email),  $y^{(i)} = -1$  (spam email).

For the above setting, we considered *linear classifiers*. For the separable case, we formulated an LP which intends to find a line that separates two datasets (legitimate vs. spam). For the non-separable case (which is typical in practice), we formulated a slightly different LP which finds a line that minimizes the aggregated *margin*.

In this section, we will consider a different classifier which is based on the least-squares problem and therefore called: the *least-squares classifier*. The idea of the



**Figure 1.18.** A block diagram of the least-squares classifier.

least-squares classifier is to find a *linear projector* that minimizes the aggregated *squared error*.

**Least-squares classifier** To see what the idea means, let us consider a block diagram for the classifier, illustrated in Fig. 1.18. The least-squares classifier is parameterized by a weight vector, say  $w \in \mathbf{R}^d$ . Given input  $x^{(i)}$ , it computes a linear projection w.r.t. the weight vector  $w$ ; hence, it outputs  $x^{(i)T} w$ . You may want to consider a slightly more general setting where we allow for having a bias term, like  $x^{(i)T} w + b$ . It turns out one can deal with this case easily with a slight modification to the classifier. This will be explored later. The way to design  $w$  is as follows. Using the corresponding label  $y^{(i)}$ , we first compute its squared error:  $\|x^{(i)T} w - y^{(i)}\|^2$ . Next compute the aggregated squared error with all of the  $m$  data points given. Finally we formulate an optimization problem which minimizes the aggregation:

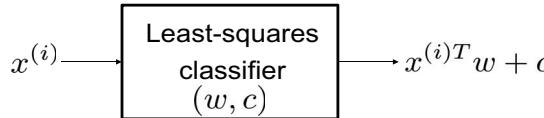
$$\min_{w \in \mathbf{R}^d} \sum_{i=1}^m \|x^{(i)T} w - y^{(i)}\|^2. \quad (1.119)$$

Notice that the objective function is very similar to the one that we saw in Section 1.1. Yes, that is the objective function that Gauss came up with in the process of addressing the astronomy problem. So we can use the same simplification trick that Gauss did, thus obtaining:

$$\begin{aligned} \sum_{i=1}^m \|x^{(i)T} w - y^{(i)}\|^2 &= \left\| \begin{bmatrix} x^{(1)T} w - y^{(1)} \\ \vdots \\ x^{(m)T} w - y^{(m)} \end{bmatrix} \right\|^2 \\ &= \left\| \begin{bmatrix} x^{(1)T} \\ \vdots \\ x^{(m)T} \end{bmatrix} w - \begin{bmatrix} y^{(1)} \\ \vdots \\ y^{(m)} \end{bmatrix} \right\|^2. \end{aligned} \quad (1.120)$$

Define

$$A := \begin{bmatrix} x^{(1)T} \\ \vdots \\ x^{(m)T} \end{bmatrix}, \quad b := \begin{bmatrix} y^{(1)T} \\ \vdots \\ y^{(m)T} \end{bmatrix}. \quad (1.121)$$



**Figure 1.19.** A block diagram of the bias-allowing least-squares classifier.

Using these notations, we can re-write the optimization problem (1.120) as:

$$\min_w \|Aw - b\|^2, \quad (1.122)$$

which is the least-squares problem. So we obtain  $w^*$  as:

$$w^* = (A^T A)^{-1} A^T b. \quad (1.123)$$

**Bias-allowing LS classifier** Consider a more general setting in which we allow for a bias in the LS classifier. See Fig. 1.19. In case a bias term is allowed, the linear classifier outputs:

$$x^{(i)T} w + c \quad (1.124)$$

where  $c \in \mathbf{R}$  indicates the bias. We can then formulate an optimization problem as:

$$\min_{w \in \mathbf{R}^d, c \in \mathbf{R}} \sum_{i=1}^m \|x^{(i)T} w + c - y^{(i)}\|^2. \quad (1.125)$$

Using exactly the same manipulation that we did in (1.120), we get:

$$\begin{aligned} \sum_{i=1}^m \|x^{(i)T} w + c - y^{(i)}\|^2 &= \left\| \begin{bmatrix} x^{(1)T} w + c - y^{(1)} \\ \vdots \\ x^{(m)T} w + c - y^{(m)} \end{bmatrix} \right\|^2 \\ &= \left\| \begin{bmatrix} x^{(1)T} & 1 \\ \vdots & \\ x^{(m)T} & 1 \end{bmatrix} \begin{bmatrix} w \\ c \end{bmatrix} - \begin{bmatrix} y^{(1)} \\ \vdots \\ y^{(m)} \end{bmatrix} \right\|^2. \end{aligned} \quad (1.126)$$

By defining

$$A := \begin{bmatrix} x^{(1)T} & 1 \\ \vdots & \\ x^{(m)T} & 1 \end{bmatrix}, \quad b := \begin{bmatrix} y^{(1)} \\ \vdots \\ y^{(m)} \end{bmatrix},$$

we see that the optimal solution is exactly of the same formula as before:

$$\begin{bmatrix} w^* \\ c^* \end{bmatrix} = (A^T A)^{-1} A^T b. \quad (1.127)$$

**Question** One natural question arises. Can the least-squares classifier perform better than the linear classifier that we developed earlier using LP? To answer this question, first of all, we need to know what is a proper performance measure that one can use.

**Look ahead** In the next section, we will cover this topic in depth, thus addressing the question. Specifically we will introduce a prominent performance measure named *test error*. We will then study how to evaluate test error, thereby making a comparison between the LS classifier and the linear classifier. We will also study a popular technique employed for improving the test error performance: *regularization*.

## 1.9 LS: Test Error, Regularization and CVXPY Implementation

---

**Recap** In the prior section, we embarked on the second instance of convex optimization: Least Squares (LS). We studied the LS problem in the context of the spam filter design. We showed that the spam filter design problem can be formulated as an LS optimization. The formulated LS problem reads: Given data points  $\{(x^{(i)}, y^{(i)})\}_{i=1}^m$ ,

$$\min_w \|Aw - b\|^2 \quad (1.128)$$

where

$$A := \begin{bmatrix} x^{(1)T} \\ \vdots \\ x^{(m)T} \end{bmatrix}, \quad b := \begin{bmatrix} y^{(1)} \\ \vdots \\ y^{(m)} \end{bmatrix}.$$

This then yields  $w^* = (A^T A)^{-1} A^T b$ . We also proved the closed-form solution using the optimality condition of unconstrained convex optimization (derived in Section 1.3). In addition, we extended into a general setting that allows for a bias term, say  $c \in \mathbf{R}$ :

$$\min_{\bar{w}:=(w,c) \in \mathbf{R}^{d+1}, c \in \mathbf{R}} \sum_{i=1}^m \|A\bar{w} - b\|^2 \quad (1.129)$$

where  $A$  now reads:

$$A := \begin{bmatrix} x^{(1)T} & 1 \\ \vdots & \vdots \\ x^{(m)T} & 1 \end{bmatrix} \in \mathbb{R}^{m \times (d+1)}. \quad (1.130)$$

At the end of the last section, we raised two questions: (i) Is the LS classifier better than the margin-based linear classifier?; and (ii) What is a proper performance measure?

**Outline** In this section, we will answer these two questions. Specifically what we are going to do are four folded. First we will introduce a prominent performance measure called *test error*. We will then study how to evaluate the test error. With this measure, we will make a comparison between the LS classifier and the linear classifier. Next, we will study a useful technique employed for the purpose of improving the test error performance: *regularization*. Lastly we will learn how to solve an LS problem via CVXPY.

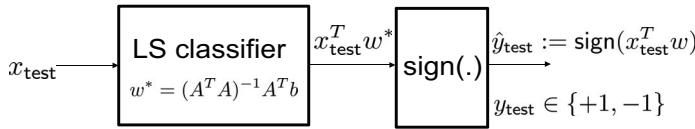


Figure 1.20. A block diagram of the least-squares classifier.

**Test data** In order to introduce a proper performance measure, we first need to study one concept on the data that is employed for the purpose of performance evaluation. In fact, what we are interested in is the performance w.r.t. data which is never used while training an interested model. Such data is so called *unseen data*. A formal name of the unseen data is *test data*, as the unseen data is employed for the purpose of testing. In contrast, we call the data used for training *train data*. Train data is usually denoted by  $\{(x^{(i)}, y^{(i)})\}_{i=1}^m$ , while test data is denoted by  $\{(x_{\text{test}}^{(i)}, y_{\text{test}}^{(i)})\}_{i=1}^{m_{\text{test}}}$  where  $m_{\text{test}}$  indicates the number of examples in the test dataset.

**Test error** The test error indicates the error computed w.r.t. such unseen data. To give you a mathematical definition, consider the LS classifier illustrated in Fig. 1.20. Here an unseen test example, say  $x_{\text{test}}$ , is fed as an input, yielding a linearly projected output  $x_{\text{test}}^T w^*$ . Notice that the output is a real value, while the label  $y_{\text{test}}$  is binary  $\in \{+1, -1\}$ . Hence, we take a transformation of  $x_{\text{test}}^T w^*$  in an effort to make an apple-to-apple comparison with such a binary label. To this end, if  $x_{\text{test}}^T w^* \geq 0$ , we declare a legitimate email, outputting  $\hat{y}_{\text{test}} = +1$ . Otherwise, we declare a spam email, setting  $\hat{y}_{\text{test}} = -1$ . In other words, we take the *sign* of the output:

$$\hat{y}_{\text{test}} = \text{sign}(x_{\text{test}}^T w). \quad (1.131)$$

Comparing this to the ground-truth label  $y_{\text{test}}$ , we compute a loss function, so called an *0/1 loss*. It takes 0 if they coincide and 1 otherwise. Considering many such data points, say  $m_{\text{test}}$  examples, we compute the test error as the average of the 0/1 losses:

$$\text{TestError} = \frac{1}{m_{\text{test}}} \sum_{i=1}^{m_{\text{test}}} \mathbf{1}\{\hat{y}_{\text{test}}^{(i)} \neq y_{\text{test}}^{(i)}\} \quad (1.132)$$

where  $\mathbf{1}\{\cdot\}$  denotes an indicator function which outputs 1 when the event  $\{\cdot\}$  is true and 0 otherwise.

**An example of test error computation** Here is an example that demonstrates the test error performance of a trained classifier conducted on a test dataset. Suppose that the test dataset has: (1) 1139 legitimate emails; and (2) 127 spam emails. Applying the trained classifier, we obtain 1120 correct and 19 wrong answers for

	$\hat{y} = +1$	$\hat{y} = -1$
$y = +1$ (legitimate)	1120 true positive	19 <b>false negative</b> (misdetection)
$y = -1$ (spam)	32 <b>false positive</b>	95 true negative <b>(false alarm)</b>

Figure 1.21. Test error performance of a trained classifier conducted on a test dataset.

1139 legitimate emails. For 127 spam emails, we have 95 correct and 32 wrong answers. See Fig. 1.21 for the summary of the results.

Then, the test error is computed as:

$$\text{TestError} = \frac{19 + 32}{1139 + 127} \approx 4\%. \quad (1.133)$$

**Four types of interested events and two types of errors** The test error is categorized into two types depending on what the ground truth is, and a different emphasis should be put on the two types depending on applications. To explain what it means, let us first introduce four types of interested events to understand the different types of errors. The first is the *true positive* case which indicates the event  $\{\hat{y} = +1 | y = +1\}$ . The second refers to another desirable situation: the *true negative* case which indicates  $\{\hat{y} = -1 | y = -1\}$ .

Now the third and fourth are relevant to *error* events. The third is the *false negative* (or *misdetection*) case indicating  $\{\hat{y} = -1 | y = +1\}$ . The third is the *false positive* (or *false alarm*) case, which refers to  $\{\hat{y} = +1 | y = -1\}$ . The first type of error is concerned about the false negative case and therefore called the *false negative rate (FNR)*:

$$\text{FNR} := \mathbb{P}\{\hat{y} = -1 | y = +1\}. \quad (1.134)$$

The second type of error refers to the false positive case, so it is called the *false positive rate (FPR)*:

$$\text{FPR} := \mathbb{P}\{\hat{y} = +1 | y = -1\}. \quad (1.135)$$

*On a side note:* The above naming (e.g., FNR and FPR) may be a bit confused to some readers. One way to remember the naming is as follows. If the test result is  $+1$  (or  $-1$ ), we say *positive* (or *negative*). If the prediction is correct (or wrong), then it reads *true* (or *false*).

In the above example exhibited in Fig. 1.21, the two types of error can be computed as:

$$\text{FNR} = \frac{19}{19 + 1120} \approx 1.7\%;$$

$$\text{FPR} = \frac{32}{32 + 95} \approx 25\%.$$

Notice that the two errors are highly imbalanced; one is much smaller than the other. If you think about it, it is sort of a *desired* situation. In reality, it is crucial to protect against missing legitimate emails. In other words, we should be able to well declare legitimate emails if they are indeed legitimate, meaning that we should reduce FNR as much as possible. In this case, it is around 1.7%, which is more or less okay.

On the other hand, we may be okay with declaring as actually spam emails as legitimate ones, meaning that a moderate value of FPR may be acceptable in reality. In this case, it is around 25%, which is more or less okay.

**Margin-based linear classifier vs. least-squares classifier** Since we figure out the concept of the test error which is a proper performance measure, we are now ready to compare performances of the two classifiers: the margin-based linear classifier and the least-squares classifier.

To this end, we first gather training dataset:  $\{(x^{(i)}, y^{(i)})\}_{i=1}^m$ . We then use this to design the margin-based linear classifier (using LP) as well as the least-squares classifier. Next we test the classifiers on test dataset  $\{(x_{\text{test}}^{(i)}, y_{\text{test}}^{(i)})\}_{i=1}^{m_{\text{test}}}$  to compute  $\text{TestError}_{\text{linear}}$  and  $\text{TestError}_{\text{LeastSquares}}$ . This is how we compare performances. You may wonder which is better in terms of the test error measure. It turns out the answer depends on datasets. In other words, there is often no concrete answer like: one is always better than the other. In Prob 3.1, you will have a chance to check just one case.

**Regularization technique** We are faced with an issue in applying the least-squares classifier if no modification is made. In reality, a data point, say  $x$ , contains some *noise*. Data points are usually obtained from measurements made by humans or sensors. But humans and sensors are not perfect in reality, so  $x$  definitely contains some error. This error incurs an issue: Large values of  $\|w^*\|$  can *boost up such noise*.

To avoid this, we somehow want to make those values small. One way to implement this is to minimize  $\|w^*\|^2$ . But obviously at the same time, we want to make  $\|Aw - b\|^2$  small; otherwise,  $w^*$  would be always 0 – this is obviously what we do not want to get. This motivates people to come up with a natural idea, which is to

regulate the two objectives at the same time:

$$\min_{w \in \mathbb{R}^d} \|Aw - b\|^2 + \lambda \|w\|^2 \quad (1.136)$$

where  $\lambda \geq 0$ . Notice that for one extreme case of  $\lambda = 0$ , we obtain the conventional least-squared solution while for the other extreme case of  $\lambda = \infty$ , we get  $w^* = 0$  in which we declare spam emails randomly. In case a prior knowledge is given, one can take a smarter action. For instance, if the probability of a randomly selected email being spam (called *a priori probability*) is known, then one can declare any randomly selected email is spam with the known probability.

The above technique is called *regularization* and  $\lambda$  is called the *regularization factor*. In the machine learning community, the regularization factor is considered as a *hyperparameter* that can be arbitrarily chosen as per our design.

**How to choose  $\lambda$ ?** One natural question that arises is then: how to choose such a hyperparameter  $\lambda$ . To figure this out, we need to understand how performances vary in terms of the regularization factor  $\lambda$ . First consider the *training error* which is defined as:

$$\text{TrainError} = \frac{1}{m} \sum_{i=1}^m \mathbf{1} \left\{ \hat{y}^{(i)} \neq y^{(i)} \right\} \quad (1.137)$$

where  $\hat{y}^{(i)} = \text{sign}(x^{(i)T} w^*)$ . Obviously the training error is minimized at  $\lambda = 0$  because the case focuses only on the error induced by the training dataset. And it monotonically increases with an increase in  $\lambda$ . Notice that the larger  $\lambda$ , the more we regulate, penalizing more on the training error. So we will get something like a blue curve, as plotted in Fig. 1.22.

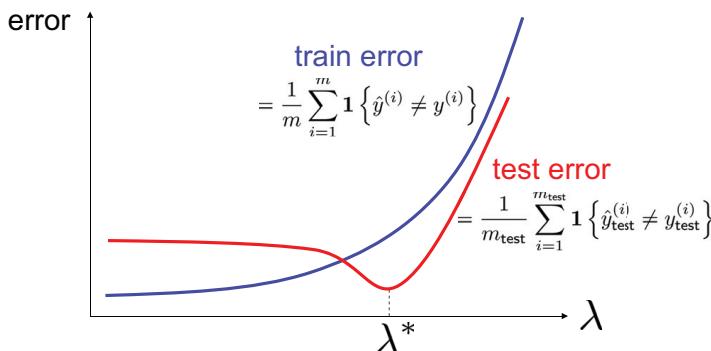


Figure 1.22. Training and test errors as a function of regularization factor  $\lambda$ .

On the other hand, the situation is different for the test error:

$$\text{TestError} = \frac{1}{m_{\text{test}}} \sum_{i=1}^{m_{\text{test}}} \mathbf{1} \left\{ \hat{y}_{\text{test}}^{(i)} \neq y_{\text{test}}^{(i)} \right\}. \quad (1.138)$$

When  $\lambda = 0$ , the test error would be small but larger than the training error because the  $\lambda = 0$  case focuses only on the training error. Increasing  $\lambda$ , we would have a regularization effect, so we can make the performance less dependent on the train data. This then puts an *indirect emphasis* on the test dataset, thus yielding a smaller test error. But if  $\lambda$  is too big, the classifier would be close to  $w^* = 0$  in which the test error would be obviously very large. So one can expect there is a sweet spot on  $\lambda$  that minimizes the test error. Hence, we may obtain something like a red curve, as plotted in Fig. 1.22. Actually this is indeed the case. This suggests a natural idea: Choosing  $\lambda^*$  that minimizes the test error.

**Validation data** But you may recognize an issue here. Remember that test data is defined as *unseen* data which is never employed during training. However, finding such  $\lambda^*$  is included in the process of designing a model; hence, any data employed for finding  $\lambda^*$  is actually “seen data”. So we may need another dataset for searching such a hyperparameter. The data used for that purpose is *validation* data. It is still “seen data”. But it is employed for the purpose of validating the goodness of a trained model. In other words, it is for the hyperparameter search. One important note on validation data: While constructing datasets, it is important to ensure that the distribution of validation dataset is similar to that of test dataset as much as possible. This is because hyperparameters (tuned due to validation data) are desired to be set so as to yield a good performance w.r.t. unseen test data.

**How to solve the regularized problem?** Going back to the regularized least-squares problem (1.136), how can we solve the problem? If you think about it, this is nothing but another least-squares problem. Why? Applying the same simplification trick as Gauss did earlier, we obtain:

$$\|Aw - b\|^2 + \lambda \|w\|^2 = \left\| \begin{bmatrix} A \\ \sqrt{\lambda} I \end{bmatrix} w - \begin{bmatrix} b \\ 0 \end{bmatrix} \right\|^2 = \|A'w - b'\|^2$$

where

$$A' := \begin{bmatrix} A \\ \sqrt{\lambda} I \end{bmatrix} \in \mathbf{R}^{(m+d) \times d} \text{ and } b' := \begin{bmatrix} b \\ 0 \end{bmatrix} \in \mathbf{R}^{m+d}. \quad (1.139)$$

Hence, we get:

$$\min_{w \in \mathbf{R}^d} \|A'w - b'\|^2. \quad (1.140)$$

So the solution would be:

$$w^* = (A'^T A')^{-1} A'^T b'. \quad (1.141)$$

**CVXPY implementation** We have figured out that many types of the LS problems with (or without) regularization and/or bias terms can be cast into:

$$\min_w \|Aw - b\|^2. \quad (1.142)$$

So constructing  $(A, b)$  from data and formulating an objective function accordingly is the key to CVXPY implementation. For illustrative purpose, let us dig into implementation details under a simple example in which there are no regularization and bias terms, and a dataset is given by:

$$\{(x^{(i)}, y^{(i)})\}^m = \left\{ \left( \begin{bmatrix} 2 \\ 3 \\ 1 \end{bmatrix}, 5 \right), \left( \begin{bmatrix} -1 \\ 2 \\ 3 \end{bmatrix}, -7 \right) \right. \\ \left. \left( \begin{bmatrix} 3 \\ 1 \\ 4 \end{bmatrix}, -9 \right), \left( \begin{bmatrix} 7 \\ 4 \\ 2 \end{bmatrix}, -12 \right) \right\}.$$

Here is a code for implementation:

```
import cvxpy as cp
import numpy as np
# optimization variable
w = cp.Variable(3)
# construct (A,b)
A = np.array([[2, 3, 1],
              [-1, 2, 3],
              [3, 1, 4],
              [7, 4, 2]])
b = np.array([5,-7,-9,-12])
# objective function
cost = cp.sum_squares(A @ w - b)
obj_min = cp.Minimize(cost)
# set up a problem
prob = cp.Problem(obj_min)
# solve the problem
prob.solve()
#print the solution
print('status:', prob.status)
print('optimal value:', prob.value)
print('optimal w*:', w.value)
```

```
status: optimal
optimal value: 71.90174092652698
optimal w*: [-1.23989377 1.23517262 -2.36470935]
```

Here a good thing about CVXPY is that the square of the matrix norm (the key operation that arises in the LS problem) is already implemented as `cp.sum_squares(A @ w - b)`. So the only thing that you should do is to construct  $(A, b)$  with `numpy.array` and then to plug the objective function properly. One can also readily check that the above  $w^*$  coincides with the closed-form solution  $(A^T A)^{-1} A^T b$ .

**Look ahead** In the next section, we will study another application in which the least-squares problem has played a crucial role. The application is the one that arises in a completely different field, the medical field. That is, *Computed Tomography* that you often hear of simply as CT.

## 1.10 LS: Computed Tomography

---

**Recap** During the past two sections, we have studied a couple of stuffs on the 2nd instance of convex optimization, Least Squares (LS):

$$\min \|Ax - b\|^2 \quad (1.143)$$

where  $x \in \mathbf{R}^d$ ,  $A \in \mathbf{R}^{m \times d}$ ,  $b \in \mathbf{R}^m$  and  $m \geq d$ . In particular, we studied those stuffs in the context of one very important machine learning application: the *classification problem*. Specifically we developed another linear classifier which can be designed via an LS problem. We also discussed a popular performance measure, called the *test error*, which is instrumental in making a fair comparison between distinct classifiers.

**Outline** In this section, we are going to study another application that arises in quite a different domain: the *medical imaging* field. The application that we will investigate is: *Computed Tomography*, CT for short. Very interestingly, there is a mathematical principle behind the idea of CT and the principle is based on LS. To see how the LS problem is related to CT, we need to understand the principle of CT. But to this end, we first need to understand the principle of X-ray which forms the basis of CT. So we are going to proceed in the following sequence. We will first study what X-ray is together with its key principle. We will then figure out the motivating context behind the invention of CT. Next we will explore the idea of CT. Finally we will formulate CT as an LS problem.

**X-ray** X-ray is a form of electromagnetic radiation. It was discovered in 1895 by a German physicist, named Wilhelm Röntgen ([Rosenbusch and de Knecht-van Eekelen, 2019](#)). The discovery opened up a new medical field, called *radiology*. The radiology is now a very well-known and well-established field in medical imaging, but there was no such field before the discovery of X-ray. In addition, the X-ray played a significant role in other areas beyond the medical field, like Physics and Chemistry. So the discovery won Röntgen the *first Nobel Prize* in Physics in 1901. Take a look at the very short 5-year gap between the discovery year 1895 and the award year 1901. It is a very rare case because it usually takes much longer time (around more than 10~20 years) to receive a Nobel Prize since the discovery or invention.

**Principle of X-ray** The principle of X-ray that we will focus on was discovered through Röntgen's key observation made while he was working in his laboratory.

While Röntgen was dealing with experimental tubes, he observed a type of *unidentified* radiation emanating from the tubes. The naming “X-ray” was originated from the nature of the *unidentified* radiation, since “X” typically refers to an unknown. And he made an interesting observation about the mysterious radiation. When passing through an object, it absorbs photons and its energy (typically quantified as the *intensity*) is proportional to the number of photons absorbed: the more photons (denser), the stronger intensity.

In fact, he did lots of experiments which support the observation, and he tested even on his wife’s hand, obtaining a scary picture that shows the bone structure of the hand.<sup>5</sup> It was a sort of the first historical medical X-ray image. The discovery together with such medical images made many people excited about the X-ray. In particular, people used the X-ray for the purpose of investigating the inside structure of interested objects (like human bodies) without drilling it, and this could open up a new medical field: *radiology*, which later opened up a broader field of medical imaging.

**Limitations of X-ray** However, the X-ray has some limitations in figuring out an internal structure of interested objects. Usually an object of interest is 3-dimensional. But the X-ray can generate only a projected 2-dimensional image, so this gives a challenge in identifying a complicated 3D object structure. For example, it is hard to spot tumors behind bones. Another clear example is illustrated in Fig. 1.23. Here a 2D image projected on the wall looks like a human hand, but



**Figure 1.23.** An example in which the projected 2D image does not well represent the 3D structure.

5. At that time, Röntgen had no idea of how detrimental X-ray is to human bodies. Perhaps this may be the reason that his wife passed away 6 years later since the discovery. Who knows?

the actual 3D object is a rabbit. This implies that much of the key structure-related information can be lost while being projected. This clearly shows the limitations of X-ray.

**Invention of Computed Tomography (CT)** Many people tried to solve such information-missing problem. A history was made in 1967 among such efforts. At the time, a smart way to address the problem was developed, named *Computed Tomography (CT)*. The technique together with a computer-aided machine was invented by two people: one is a British electrical engineer, named Godfrey Hounsfield; and the other is a South African-American physicist, named Allan Cormack ([Cormack and Hounsfield](#)). In fact, this invention was not done in a cooperative manner – rather it was done independently. While Hounsfield's invention was slightly earlier, the credit was also fully given to Cormack, so they could be co-awarded the Nobel Prize in Physiology or Medicine in 1979.

**Idea of CT** The idea of CT is very well reflected in the name. “Tomos” is a Greek word which means “projected section (or slice)”; and “graphy” means “describe”. So in words, it means “describing an object using slices.” Details on the idea are the following:

1. Project X-ray beams onto an object from many different angles;
2. Calculate the intensities of the projected images (*slices*);
3. Use them to reconstruct (*describe*) the object.

**A simple example** To explain what it means in detail, let us consider the following example in which an object of interest is comprised of four equal-sized grids – see Fig. 1.24. For illustrative purpose, we consider a 2D object, although many of interested objects are 3D. Once you grasp the idea, you will soon understand that

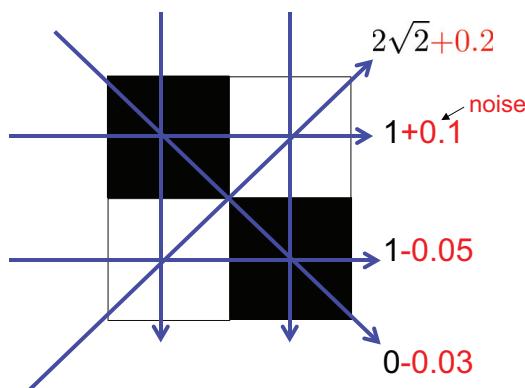


Figure 1.24. A simple grid example that well illustrates the idea of CT.

the idea can readily be extended to a 3D object case. The object has two black grids on left upper and right bottom, while having two white grids on right upper and left bottom. Suppose that X-ray absorbs no photon when passing through a black grid, i.e., the density is 0. Here we define the unit of the density as the number of photons *per unit length*. On the other hand, assume that X-ray absorbs something (say, the density is 1) when passing through a white grid.

Suppose we project a horizontal X-ray beam to the upper part of the object so that it passes through the two upper grids. Then, it would absorb nothing in the black grid zone while absorbing something in the white grid zone. Since the unit of the density that we define here is w.r.t. the unit length, the intensity of the X-ray beam would be proportional to the width of the white grid zone. For simplicity, assume that the width is 1 unit length. Then, the intensity would be 1. But there is always an error in measurement, say that the measurement noise here is **0.1** (marked in red in Fig. 1.24). We also project another horizontal X-ray beam to the bottom part of the object. We then measure the corresponding intensity, say  $1 - \textcolor{red}{0.05}$ . Projecting two vertical downward beams, we get two measurements, say:  $1 + \textcolor{red}{0.02}$  and  $1 - \textcolor{red}{0.1}$ .

Shooting a top-left to bottom-right diagonal beam to the object, we would absorb nothing since it passes only through the black grids, so the measurement would be close to 0, say  $0 - \textcolor{red}{0.03}$ . On the other hand, the other bottom-left to top-right diagonal beam would pass only through the white grids. Since the length of the passing zone is  $2\sqrt{2}$ , the intensity measurement would be close to  $2\sqrt{2}$ , say  $2\sqrt{2} + \textcolor{red}{0.2}$ .

What we want to figure out are the densities of the four grids, so let us denote those by unknown variables, say  $d_1$  (top left),  $d_2$  (top right),  $d_3$  (bottom left),  $d_4$  (bottom right). Using these notations, we can express the above six measurements as the following linear equations:

$$\begin{aligned} d_1 + d_2 &= 1.1 \\ d_3 + d_4 &= 0.95 \\ d_1 + d_3 &= 1.02 \\ d_2 + d_4 &= 0.9 \end{aligned} \tag{1.144}$$

$$\sqrt{2}d_1 + \sqrt{2}d_4 = -0.03$$

$$\sqrt{2}d_2 + \sqrt{2}d_3 = 2\sqrt{2} + 0.2.$$

**Least squares formulation** Notice in (1.144) that we have six equations while having four unknowns. So there is *no solution* in general. This is indeed the no-solution case. But we can invoke Gauss's idea to address this case. In other words,

we can formulate it as an LS problem. Defining

$$A := \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ \sqrt{2} & 0 & 0 & \sqrt{2} \\ 0 & \sqrt{2} & \sqrt{2} & 0 \end{bmatrix} \text{ and } b := \begin{bmatrix} 1.1 \\ 0.95 \\ 1.02 \\ 0.9 \\ -0.03 \\ 2\sqrt{2} + 0.2 \end{bmatrix},$$

we can formulate it as:

$$\min \|Ad - b\|^2. \quad (1.145)$$

Then, the solution would be  $d^* = (A^T A)^{-1} A^T b$ . For the above example, we obtain:

$$(d_1^*, d_2^*, d_3^*, d_4^*) = (0.0400, 1.0613, 1.0463, -0.0950).$$

The solution makes an intuitive sense, as it is close to the ground truth  $(0, 1, 1, 0)$ .

**A more realistic example** In fact, the example in Fig. 1.24 is too simple. In reality, an object is of an arbitrary shape and also the density of the object *continuously* changes over regions. To understand how to apply the idea into a more realistic object, let us consider another example in Fig. 1.25.

Here what we want to figure out is the density, say  $d(x, y)$ , which indicates the density at a coordinate  $(x, y)$ . Suppose we project to the object an X-ray beam (with a bottom-left to top-right diagonal direction), as illustrated in Fig. 1.25. Let  $t$  be the length of the beam trajectory from the starting point  $(x_0, y_0)$  at  $t = 0$ . Let  $\theta$  be the angle of the beam in reference to the  $x$ -axis. Then, the coordinate  $p(t)$  that the

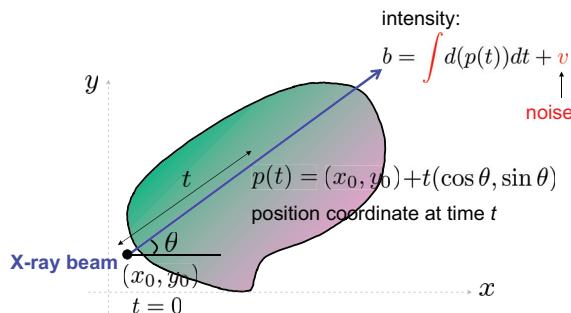


Figure 1.25. A more realistic example for illustration of CT idea.

beam indicates when the beam length is  $t$  would be:

$$p(t) = (x_0, y_0) + t(\cos \theta, \sin \theta). \quad (1.146)$$

Remember that the intensity is: (density)  $\times$  (length that the beam traverses). Here the density changes over the regions that the beam swipes. So it can be represented as  $d(p(t))$ . And the length of the beam w.r.t. the very small region where the density is almost constant can be represented by  $dt$ . So the intensity measurement would be:

$$b = \int d(p(t))dt + v \quad (1.147)$$

where  $v$  indicates a measurement noise.

**Discretization** A question arises. How to estimate the density  $d(p(t))$  only from such measurement (1.147)? Specifically, one can ask: How is it related to the LS problem which does not deal with such integral-involved term? The idea is applying the *discretization* illustrated in Fig. 1.26. We make many minuscule grids in the space so that the density for each tiny grid can be assumed to be constant. Let  $d_i$  be the density of the  $i$ th grid. Denoting by  $a_i$  the length of the beam traversed at the  $i$ th grid, we can approximate the intensity absorbed through the  $i$ th grid as  $a_i d_i$ . Letting  $S_{\text{beam}}$  the set of the indices of the grids that the beam travels, we can approximate the aggregated intensity measured as:

$$b \approx \sum_{i \in S_{\text{beam}}} a_i d_i + v. \quad (1.148)$$

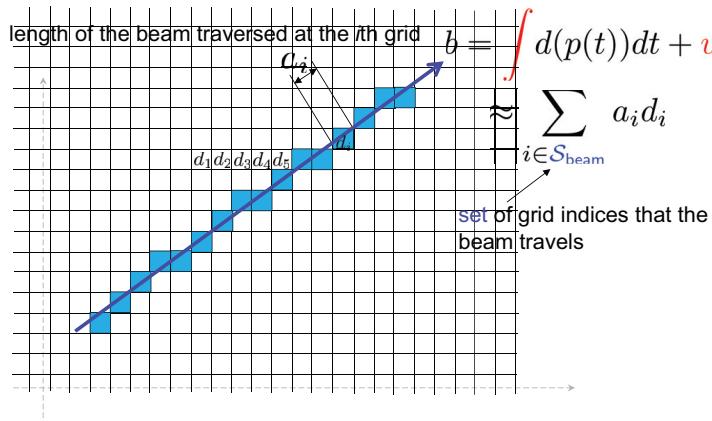


Figure 1.26. The discretization idea for CT.

Now shooting many X-ray beams from many different angles, we obtain the following measurements:

$$\begin{aligned} b_1 &\approx \sum_{i \in \mathcal{S}_{\text{beam}1}} a_i d_i + v_1 \\ b_2 &\approx \sum_{i \in \mathcal{S}_{\text{beam}2}} a_i d_i + v_2 \\ &\vdots \\ b_m &\approx \sum_{i \in \mathcal{S}_{\text{beam}\cdot m}} a_i d_i + v_m. \end{aligned} \tag{1.149}$$

**LS formulation** Notice in (1.149) that we have  $m$  equations and the number of unknowns is the same as the number of grids that the object spans. With a sufficiently large number  $m$  of measurements (this is subject to our design), we can make  $m$  always larger than the number of unknowns. And for this setting, we can again apply Gauss's idea to formulate an LS problem as follows:

$$\min \|Ad - b\|^2 \tag{1.150}$$

where

$$A := \begin{bmatrix} \{a_i\}_{i \in \mathcal{S}_{\text{beam}1}} \\ \{a_i\}_{i \in \mathcal{S}_{\text{beam}2}} \\ \vdots \\ \{a_i\}_{i \in \mathcal{S}_{\text{beam}\cdot m}} \end{bmatrix}, \quad b := \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix}.$$

Here  $\{a_i\}_{i \in \mathcal{S}_{\text{beam}1}}$  an  $n_{\text{grid}}$ -dimensional row vector wherein we read  $a_i$ 's for  $i \in \mathcal{S}_{\text{beam}1}$  while 0 otherwise, and  $n_{\text{grid}}$  denotes the total number of grids. For instance, if  $n_{\text{grid}} = 10$  and  $\mathcal{S}_{\text{beam}1} = \{1, 5, 9\}$ , then

$$\{a_i\}_{i \in \mathcal{S}_{\text{beam}1}} = [a_1 \ 0 \ 0 \ 0 \ a_5 \ 0 \ 0 \ 0 \ a_9 \ 0]. \tag{1.151}$$

**History of CT scanners** This is the idea that Hounsfield came up with. While he mimicked Gauss's idea, we believe that the way to mimick is highly non-trivial. Applying this idea, Hounsfield could also develop a prototype CT scanner in 1971 (Beckmann, 2006); see the first left picture in Fig. 1.27.

Remember that he was an *electrical engineer* – he was good enough to build an electrical computer-aided machine. The prototype supported  $m = 160$  measurements (X-ray beams). The scanning time for each beam was a little over 5 minutes. So the total scanning time was around 13 hours. Also, the computation time for reconstructing an object with measurements (solving the LS problem) was around



Prototype CT scanner   A historical EMI-scanner   CT scanner nowadays

Figure 1.27. History of CT scanners.

2.5 hours on a computer that he had. So it could not be commercialized as it took lots of time.

Fortunately, at that time, Hounsfield was working at a big and supportive company: EMI (Electric & Music Industries). While EMI was a music-record company, it was rich enough to invest some money to a field which has nothing to do with the music industries. Actually EMI was even going further. There was a rumour that with tons of money from the sales of *The Beatles* records in the 1960s, EMI helped fund the development of CT scanners. Anyhow the fact is that in the same year 1971, Hounsfield could develop the first commercial CT scanner with generous support from EMI, named the EMI-scanner (Bhattacharyya, 2016) – see the middle picture in Fig. 1.27. The performance of the scanner was remarkable relative to the prototype scanner. The scanning and reconstruction times were around 4 minutes and 7 minutes, respectively. So it could be commercialized.

CT scanners nowadays are beyond remarkable. For example, Siemens CT scanner (2017 model) in Fig. 1.27 takes only  $\sim 0.33$  seconds for the whole process.

**Look ahead** So far, we have studied two instances of convex optimization: LP and LS. In the next section, we will study another instance which subsumes LP and LS as special cases: Quadratic Program.

## Problem Set 3

---

**Prob 3.1 (Margin-based linear vs. Least-Squares classifiers)** Consider the legitimate-vs-spam email classification. We are given  $\{(x^{(i)}, y^{(i)})\}_{i=1}^m$  training dataset. This is the same as that in the file “train.csv” that we employed in Prob 2.3. Remember the file is placed in:

[http://csuh.kaist.ac.kr/convex\\_book/PS2/train.csv](http://csuh.kaist.ac.kr/convex_book/PS2/train.csv)

In this problem, you will build up two classifiers: (i) the margin-based linear classifier; and (ii) the Least-Squares classifier. You will also compare the test error performances with test dataset  $\{(x_{\text{test}}^{(i)}, y_{\text{test}}^{(i)})\}_{i=1}^{m_{\text{test}}}$ . The test dataset is another file “test.csv” uploaded on:

[http://csuh.kaist.ac.kr/convex\\_book/PS3/test.csv](http://csuh.kaist.ac.kr/convex_book/PS3/test.csv)

You need to write a script for CYXPY implementation.

- (a) Formulate an LP for the margin-based linear classifier. Solve the problem (using the training data) to design the classifier.
- (b) Formulate an optimization for the least-squares classifier. Solve the problem to design the classifier.
- (c) Use Python to compute the test errors of the two classifiers designed in parts (a) and (b). Which classifier is better in terms of test error?
- (d) State the two types of errors together with their definitions. Also explain which type should be further minimized relative to the other in this problem.

**Prob 3.2 (Bias-allowing least-squares problem)** In this problem, you will design least squares classifiers which allow for bias terms, for legitimate-vs-spam email classification. Suppose that a linear classifier reads:

$$y = x^T w + c \quad (1.152)$$

where  $x \in \mathbf{R}^d$  and  $y \in \mathbf{R}$  denote the input and output of the classifier, respectively. Also  $w \in \mathbf{R}^d$  and  $c \in \mathbf{R}$  indicate the linear weight and bias, respectively. Let

$$A := \begin{bmatrix} x^{(1)T} & 1 \\ x^{(2)T} & 1 \\ \vdots & \vdots \\ x^{(m)T} & 1 \end{bmatrix} \in \mathbf{R}^{m \times (d+1)},$$

$$\bar{w} := \begin{bmatrix} w \\ c \end{bmatrix} \in \mathbf{R}^{d+1},$$

$$b := \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(m)} \end{bmatrix} \in \mathbf{R}^m$$

where  $\{(x^{(i)}, y^{(i)})\}_{i=1}^m$  indicate the training dataset employed in Prob 3.1. You need to write a script for CVXPY implementation.

- (a) Formulate the bias-allowing least squares problem. Use the training dataset to design a classifier.
- (b) Use the trained classifier together with the test dataset in Prob 3.1 to compute the test error. Also compare its performance to that of the ordinary least squares classifier designed in Prob 3.1.

**Prob 3.3 (Regularized least-squares problem)** In this problem, you will design regularized least-squares classifiers for legitimate-vs-spam email classification. Use the same datasets employed in Prob 3.1. You also need to write a script for CVXPY implementation.

- (a) Formulate a regularized least squares problem. Use the training dataset to design classifiers for different regularization factors  $\lambda$ . You may want to set up a range of  $\lambda$  like:

```
import numpy as np
lambda_ = np.logspace(-2,3,51)
```

- (b) Define the training error. Use Python to plot the training error as a function of  $\lambda$ . Also explain the shape of the curve and describe why.
- (c) Define the test error. Use the classifiers (trained in part (a)) together with the test dataset to plot the test error as a function of  $\lambda$ . Also explain the shape of the curve and describe why.

**Prob 3.4 (2nd-order condition of convexity)** Suppose  $f : \mathbf{R}^d \rightarrow \mathbf{R}$  is twice differentiable, i.e., its second derivative  $\nabla^2 f$  (also called Hessian) exists at each point in  $\text{dom } f$ . In addition to the 1st-order condition of convexity that we proved in Prob 1.5, another well-known fact w.r.t. convexity is:  $f$  is convex if and only if

$$\begin{aligned} \text{dom } f &\text{ is convex;} \\ \nabla^2 f(x) &\text{ is positive semidefinite, i.e., } \nabla^2 f(x) \succeq 0 \quad \forall x \in \text{dom } f. \end{aligned} \tag{1.153}$$

This problem explores the proof of this via the following subproblems.

- (a) State the definition of a *positive semidefinite matrix*.
- (b) Suppose  $d = 1$ . Show that if  $f(x)$  is convex, then (1.153) holds.
- (c) Suppose  $d = 1$ . Show that if (1.153) holds, then  $f(x)$  is convex.
- (d) Prove the 2nd-order condition for arbitrary  $d$ .

### Prob 3.5 (Concepts)

- (a) In binary classification, four types of events are emphasized in Section 1.9. What are those? Also explain the rationale behind the naming of two types of errors.
- (b) In the legitimate-vs-spam email classification, which type of errors do we need to minimize further? Also explain why.
- (c) State the definitions of hyperparameter and validation set.
- (d) What are the meanings of *tomos* and *graphy*? Describe the idea of Computed Tomography (CT).
- (e) In the least squares problem for CT, what is the size of the matrix  $A$ ? What is the condition on the size? Can the condition easily be satisfied? Also explain why.

**Prob 3.6 (Regularized CT)** Consider the LS problem that we formulated for CT in Section 1.10:

$$\min_d \|Ad - b\|^2 \quad (1.154)$$

where  $d \in \mathbf{R}^n$  indicates the density vector,  $A \in \mathbf{R}^{m \times n}$  denotes the measurement matrix, and  $b \in \mathbf{R}^m$  is the measurement vector. Here  $m$  denotes the number of X-ray beams projected to an object of interest, and  $n$  indicates the number of grids that span the object. We assume that  $d$  is a vectorized version of a matrix which discretizes the object with  $n_1$  horizontal grids and  $n_2$  vertical grids. See Fig. 1.28 for such vectorization. In the example,  $n_1 = 50$ ,  $n_2 = 50$  and  $n = n_1 n_2 = 2500$ .

In practice, people use a slightly different version of the ordinary LS problem (1.154), in an effort to exploit some *prior* information on  $d$ . The density vector  $d$  is a conversion of *image* information. One of the common properties of natural images is that it is *smooth*; neighboring pixel values are not very much different from each other. One way to enforce the smoothness is to minimize the differences across *horizontally*-neighboring pixel values, i.e., minimize:

$$\begin{aligned} & \|d_1 - d_2\|^2 + \|d_2 - d_3\|^2 + \cdots + \|d_{n_1-1} - d_{n_1}\|^2 \\ & + \|d_{n_1+1} - d_{n_1+2}\|^2 + \|d_{n_1+2} - d_{n_1+3}\|^2 + \cdots + \|d_{2n_1-1} - d_{2n_1}\|^2 \end{aligned}$$

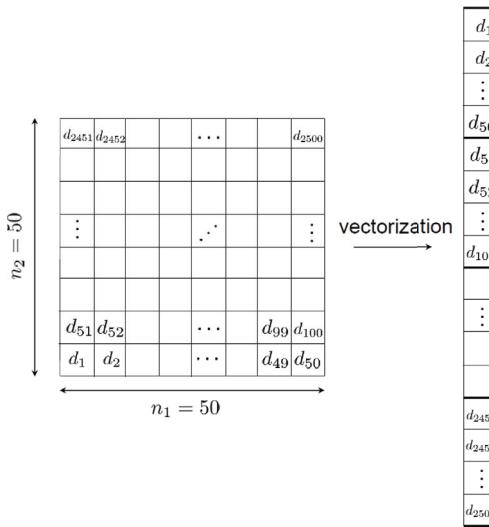


Figure 1.28. Vectorization of a discretized matrix for a 2-dimensional object.

$$+ \quad \vdots$$

$$+ \|d_{n_1(n_2-1)+1} - d_{n_1(n_2-1)+2}\|^2 + \cdots + \|d_{n_1n_2-1} - d_{n_1n_2}\|^2. \quad (1.155)$$

(a) Suppose we want to express (1.155) into a matrix-vector form as follows:

$$(1.155) = \|D_h d\|^2. \quad (1.156)$$

What is  $D_h$ ?

(b) Another way to enforce the smoothness is to make sure that the *vertically*-neighboring pixel values are small, i.e., minimize the following quantity:

$$\begin{aligned} & \|d_1 - d_{n_1+1}\|^2 + \|d_{n_1+1} - d_{2n_1+1}\|^2 + \\ & \quad \cdots + \|d_{n_1(n_2-2)+1} - d_{n_1(n_2-1)+1}\|^2 \\ & + \|d_2 - d_{n_1+2}\|^2 + \|d_{n_1+2} - d_{2n_1+2}\|^2 + \\ & \quad \cdots + \|d_{n_1(n_2-2)+2} - d_{n_1(n_2-1)+2}\|^2 \\ & + \quad \vdots \\ & + \|d_{n_1} - d_{2n_1}\|^2 + \|d_{2n_1+1} - d_{3n_1}\|^2 + \cdots + \|d_{n_1(n_2-1)} - d_{n_1n_2}\|^2. \end{aligned} \quad (1.157)$$

Again we want to express (1.157) into a matrix-vector form like:

$$(1.157) = \|D_v d\|^2. \quad (1.158)$$

What is  $D_v$ ?

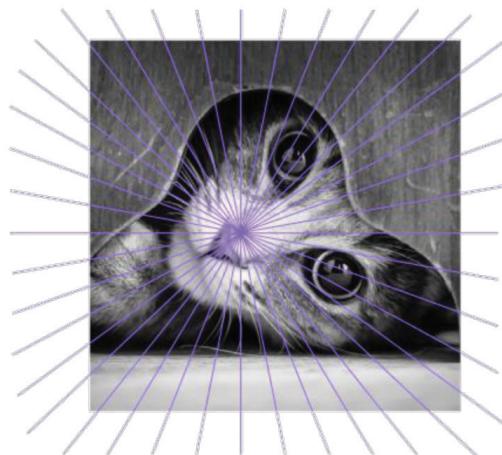
- (c) As we did before, we want to formulate a *regularized* LS problem which minimizes  $\|Ad - b\|^2$  and  $\|D_h d\|^2 + \|D_v d\|^2$  at the same time. Suppose we employ a regularization factor  $\lambda$  which would be multiplied to  $\|D_h d\|^2 + \|D_v d\|^2$ . Formulate such regularized LS problem – your final form should follow the standard form of an LS.

**Prob 3.7 (Implementation of regularized CT)** We implement the regularized CT formulated in Prob 3.6. Suppose we project many X-ray beams to an interested 2-dimensional image from different angles, as illustrated in Fig. 1.29.

Let  $m$  be the number of X-ray beams projected. We discretize the 2D image with  $n_1$  horizontal and  $n_2$  vertical grids so that the total number  $n$  of grids is  $n_1 n_2$ . We apply this method to obtain a measurement matrix  $A \in \mathbf{R}^{m \times n}$  and a measurement vector  $b \in \mathbf{R}^m$ . This data is given in the files: (i) “CT\_data1.csv” (for image 1); and (ii) “CT\_data2.csv” (for image 2). These are uploaded on:

[http://csuh.kaist.ac.kr/convex\\_book/PS3/CT\\_data1.csv](http://csuh.kaist.ac.kr/convex_book/PS3/CT_data1.csv)

[http://csuh.kaist.ac.kr/convex\\_book/PS3/CT\\_data2.csv](http://csuh.kaist.ac.kr/convex_book/PS3/CT_data2.csv)



**Figure 1.29.** Projecting many X-ray beams to an interested 2D image from different angles.

- (a) What are  $m$ ,  $n_1$ ,  $n_2$  and  $n$ ?
- (b) For image 1, consider the regularized least squares problem formulated in Prob 3.3. For the regularization factor  $\lambda \in \{0, 1, 100\}$ , solve the problem to estimate the density vector  $d$ . Also visualize it. For visualization, you may want to use the following Python script:

```
import numpy as np
import matplotlib.pyplot as plt
Img = np.reshape(dhat,(n_1,n_2))
Img = np.rot90(Img,3)
Img = np.fliplr(Img)
plt.imshow(Img,'gray')
plt.show()
```

where  $dhat$  indicates the estimated density vector. Also write a script for CVXPY implementation.

- (c) Repeat part (b) for image 2.

**Prob 3.8 (CT)** Let  $d(x, y)$  be the density of an object at coordinate  $(x, y)$ . Suppose we project to the object an X-ray beam. Let  $t$  be the length of the beam trajectory from the starting from  $(x_0, y_0)$  at  $t = 0$ . Let  $\theta$  be the angle of the beam in reference to the  $x$ -axis. Let  $v$  be an additive noise in the intensity measurement. Express the intensity measurement in terms of  $d(\cdot, \cdot)$  and  $v$ . Then, explain the *discretization* idea (that we learned in Section 1.10) to express an approximation of the intensity measurement. Moreover, use the idea of Computed Tomography (CT) to formulate a least-squares problem. Specify a condition on the size of the matrix (multiplied to the density vector) and discuss if the condition can be easily satisfied in reality.

### Prob 3.9 (True or False?)

- (a) Consider the regularized least squares classifier that we learned in Section 1.9:

$$\min_{w \in \mathbf{R}^d} \|Aw - b\|^2 + \lambda \|w\|^2 \quad (1.159)$$

where  $A \in \mathbf{R}^{m \times d}$ ,  $m$  denotes the number of training samples, and  $\lambda \geq 0$  indicates the regularization factor. There exists a sweet spot, say  $\lambda^* > 0$ , such that the test error is minimized, and this can be found via test data in practice.

- (b) In real applications, the misdetection error is desired to be smaller than the false positive error.
- (c) The regularized CT can be formulated as an *ordinary* least-squares problem.

## 1.11 Quadratic Program

---

**Recap** During several past sections, we have studied two instances of convex optimization: LP and LS. For LP, we investigated a bunch of important examples that can be formulated as LPs or can be solved via LP relaxation. For LS, we explored two applications to demonstrate the power of the LS problem.

**Outline** In this section, we are going to study the follow-up instance that includes LP and LS as special cases: Quadratic Program (QP). Specifically we are going to cover five stuffs. First we will study what QP is and show its convexity. We will then verify that QP indeed subsumes LP and LS. Next, we will investigate a special case of QP which exhibits a closed-form solution. The focused problem is *Constrained Least Squares*. In the fourth part, we will discuss how to deal with general QP yet in a very brief manner. Lastly we will investigate CVXPY implementation.

**Quadratic Program** The standard form of QP reads:

$$\begin{aligned} & \min w^T x + x^T Qx : \\ & Ax - b \leq 0 \\ & Cx - e = 0 \end{aligned} \tag{1.160}$$

where  $Q = Q^T \in \mathbf{R}^{d \times d} \succeq 0$  is a positive semi-definite (PSD) matrix. We say that a *symmetric* matrix, say  $Q = Q^T \in \mathbf{R}^{d \times d}$ , is positive semi-definite if  $v^T Q v \geq 0$ ,  $\forall v \in \mathbf{R}^d$ . Equivalently, one often uses the following condition instead: All the eigenvalues of  $Q$  are non-negative. One can prove the equivalence between the two conditions, relying upon eigenvalue value decomposition (w.r.t.  $Q$ ) together with some manipulation. Please exercise by yourself if you are not convinced. The PSD condition is simply denoted by  $Q \succeq 0$ . Here  $(w, A, b, C, e)$  are of compatible size. Using the 2nd-order condition of convexity (check in Prob 3.4), one can readily show that QP is indeed a convex optimization problem:

$$\begin{aligned} \nabla^2(w^T x + x^T Qx) &= \nabla(w + (Q + Q^T)x) \\ &= \nabla(w + 2Qx) \\ &= 2Q \succeq 0 \end{aligned}$$

where the 1st and 3rd equalities are due to the definition of gradient w.r.t. a vector  $x \in \mathbf{R}^d$  (please check Prob 1.2); and the 2nd and last come from our hypothesis ( $Q^T = Q \succeq 0$ ).

Obviously QP includes LP as a special case in which  $Q = 0$ . To check whether it subsumes LS, consider:

$$\min_{x \in \mathbf{R}^d} \|Ax - b\|^2 = \min_{x \in \mathbf{R}^d} x^T (A^T A)x - 2b^T Ax + b^T b. \quad (1.161)$$

First observe that  $b^T b$  does not alter the optimal solution. The second to notice is that  $A^T A$  is PSD. Why? Notice that

$$x^T A^T A x = (Ax)^T (Ax) = \|Ax\|^2 \geq 0 \quad \forall x \in \mathbf{R}^d. \quad (1.162)$$

**Equality-constrained LS** As mentioned in the beginning, there is a special (yet important) case in which the closed-form solution exists. The special case is very similar to the ordinary LS – the only distinction is that we additionally include an *equality* constraint. Let us call such a problem *equality-constrained LS*:

$$\begin{aligned} \min_{x \in \mathbf{R}^d} & \|Ax - b\|^2 : \\ & Cx - e = 0 \end{aligned} \quad (1.163)$$

where  $A \in \mathbf{R}^{m \times d}$ ,  $b \in \mathbf{R}^d$ ,  $C \in \mathbf{R}^{p \times d}$  and  $e \in \mathbf{R}^p$ .

Obviously we are interested in the case of  $m \geq d$ . Why? Remember what we discussed in Section 1.8. Depending on the values of  $p$  and  $d$ , we can now think of two cases: (i)  $p \geq d$ ; and (ii)  $p < d$ . The first is not an interesting case because in the case  $x^*$  is simply determined solely by the equality constraint (so it has nothing to do with minimizing the squared error) or there is no solution. Hence, the second case  $p < d$  is of interest. Regarding the wide (or square) matrix  $C$ , assume that

$$\text{rank}(C) = p. \quad (1.164)$$

We also assume that

$$\text{rank} \left( \begin{bmatrix} A \\ C \end{bmatrix} \right) = d, \quad (1.165)$$

meaning that all the columns of  $\begin{bmatrix} A \\ C \end{bmatrix}$  are linearly independent. Actually these often hold in reality. Later you will figure out the assumptions (1.164) and (1.165) are instrumental in deriving the closed-form solution.

**Closed-form solution** Consider the case in which  $m \geq d$ ,  $p < d$ , and (1.164) and (1.165) hold. Under the case, the closed-form solution to (1.163) reads:

$$x^* = d\text{-Components} \left\{ \begin{bmatrix} 2A^T A & C^T \\ C & 0 \end{bmatrix}^{-1} \begin{bmatrix} 2A^T b \\ e \end{bmatrix} \right\} \quad (1.166)$$

where  $d\text{-Components}(\cdot)$  is an operator that takes the first  $d$  components of  $(\cdot)$ . Notice that the inside of the operator is an  $(d+p)$ -dimensional vector.

In fact, once we start with the given formula (1.166) for  $x^*$ , the proof is straightforward, although directly deriving the formula is highly non-trivial. Here we will take an easy way, formally stated below.

1. Show that if  $\exists(x^*, z) \in \mathbf{R}^{d+p}$  such that

$$\begin{bmatrix} 2A^T A & C^T \\ C & 0 \end{bmatrix} \begin{bmatrix} x^* \\ z \end{bmatrix} = \begin{bmatrix} 2A^T b \\ e \end{bmatrix}, \quad (1.167)$$

then  $x^*$  must be the optimal solution, i.e.,  $\|Ax - b\|^2 \geq \|Ax^* - b\|^2, \forall x$  subject to  $Cx - e = 0$ .

2. Show that

$$\begin{bmatrix} 2A^T A & C^T \\ C & 0 \end{bmatrix} \text{ is invertible due to (1.164) and (1.165).} \quad (1.168)$$

*A site note:* The direct derivation of the formula (1.166) requires the identification of the optimality condition for  $x^*$  (which is non-straightforward to derive) as well as some concepts (e.g., Lagrange functions) that we did not study yet. Lagrange functions will be covered in Part II. Hence, this derivation is omitted herein. But the detailed derivation will be given in Section 2.2. You will also have a chance to get some sense of the direct derivation in Prob 4.2.

**Remark** Prior to proving (1.166), let us say a few words about the interested matrix that appears in (1.166):

$$\begin{bmatrix} 2A^T A & C^T \\ C & 0 \end{bmatrix}. \quad (1.169)$$

This is a very famous matrix, named the KKT matrix. Why do people name it KKT? Because it is the key matrix recognized by three prominent scholars: Karush, Kuhn and Tucker. Kuhn is famous as a friend of John Nash, who received the Nobel Prize in economics for the game theory and is a model for the main character in the movie *Beautiful Mind*. Tucker is famous as a PhD advisor of John Nash. In fact, they recognized the matrix in the process of deriving necessary conditions for optimality of *general* optimization problems, named *KKT conditions* that you may hear of. The KKT conditions are very important conditions that form the basis of strong duality that we will study in Part II. So we will discuss more on this later.

*A side note:* The KKT conditions were publicized in a conference paper by Kuhn and Tucker in 1951 ([Kuhn and Tucker, 2014](#)). But later it was revealed

that the same conditions were already derived in the master thesis by Karush in 1939 ([Karush, 1939](#)).

**Proof:** (1.167)  $\implies \|Ax - b\|^2 \geq \|Ax^* - b\|^2, \forall x : Cx - e = 0$  As mentioned earlier, the proof is straightforward. Consider

$$\begin{aligned}\|Ax - b\|^2 &= \|(Ax - Ax^*) + (Ax^* - b)\|^2 \\ &= \|Ax - Ax^*\|^2 + \|Ax^* - b\|^2 + 2(Ax - Ax^*)^T(Ax^* - b) \\ &\stackrel{(a)}{=} \|Ax - Ax^*\|^2 + \|Ax^* - b\|^2 \\ &\geq \|Ax^* - b\|^2.\end{aligned}$$

The only thing that remains to complete the proof is to show the step (a) in the above. See below for the proof:

$$\begin{aligned}2(Ax - Ax^*)^T(Ax^* - b) &= 2(x - x^*)^T A^T (Ax^* - b) \\ &\stackrel{(b)}{=} -(x - x^*)^T C^T z \\ &= -(Cx - Cx^*)^T z \\ &\stackrel{(c)}{=} -(e - e)^T z = 0\end{aligned}$$

where (b) comes from the fact that  $2A^T Ax^* - 2A^T b = -C^T z$ , which is the first component in (1.167); and (c) is due to  $Cx^* = e$ , which is the second component in (1.167).

**Proof of (1.168)** The proof idea is *by contradiction*. Suppose

$$\begin{bmatrix} 2A^T A & C^T \\ C & 0 \end{bmatrix} \text{ is not invertible.} \quad (1.170)$$

Here not being invertible means that any column in the matrix in (1.170) can be expressed as a linear combination of the other columns in the matrix. This implies that

$$\exists \begin{bmatrix} \bar{x} \\ \bar{z} \end{bmatrix} \neq 0 : \begin{bmatrix} 2A^T A & C^T \\ C & 0 \end{bmatrix} \begin{bmatrix} \bar{x} \\ \bar{z} \end{bmatrix} = 0. \quad (1.171)$$

This then gives:

$$2A^T A \bar{x} + C^T \bar{z} = 0. \quad (1.172)$$

Multiplying  $\bar{x}^T$  to both sides from the left, we get:

$$2\bar{x}^T A^T A \bar{x} + \bar{x}^T C^T \bar{z} = 0. \quad (1.173)$$

Since  $C\bar{x} = 0$  due to (1.171),  $\bar{x}^T C^T = 0$ . Applying this to (1.173), we get:  $\|Ax\|^2 = 0$ , which then yields:  $A\bar{x} = 0$ . This together with  $C\bar{x} = 0$  gives:

$$\begin{bmatrix} A \\ C \end{bmatrix} \bar{x} = 0. \quad (1.174)$$

Recall one of our assumptions made earlier (1.165). This implies that all the columns of  $\begin{bmatrix} A \\ C \end{bmatrix}$  are *linearly independent*. So (1.174) must imply that

$$\bar{x} = 0. \quad (1.175)$$

Applying this to (1.172), we get:

$$C^T \bar{z} = 0. \quad (1.176)$$

Again recall the other assumption made earlier (1.164). This implies that all the rows of  $C$  (i.e., all the columns of  $C^T$ ) are *linearly independent*. Hence,

$$\bar{z} = 0. \quad (1.177)$$

This together with (1.175) yields contradiction with (1.171), thus completing the proof (1.168).

**General QP** Recall the general QP taking the following standard form:

$$\begin{aligned} & \min w^T x + x^T Q x : \\ & Ax - b \leq 0 \\ & Cx - e = 0 \end{aligned} \quad (1.178)$$

where  $Q = Q^T \succeq 0$  is a PSD matrix. Now how to solve the general QP? Unfortunately, there is no closed-form solution in general. As mentioned in Section 1.3, *strong duality* provides algorithmic insights. So we will study how to solve the problem later when dealing with strong duality in Part II.

**CVXPY implementation** While we will cover the algorithm in Part II, here we investigate how to use CVXPY for solving QP. CVXPY implementation depends highly on the standard form (1.178). So the key procedure includes: (i) constructing the interested matrices and vectors (i.e.,  $Q, w, A, b, C, e$ ); and (ii) formulating a

problem object using QP-tailored built-in functions properly. For illustrative purpose, we consider a simple example:

$$Q = \begin{bmatrix} 26 & 3 & 10 \\ 3 & 10 & 1 \\ 10 & 1 & 6 \end{bmatrix}, w = \begin{bmatrix} 1 \\ 3 \\ 2 \end{bmatrix},$$

$$A = [1 \ -4 \ 3], b = 10, C = [2 \ -1 \ 2], e = 3.$$

See below a code for implementation:

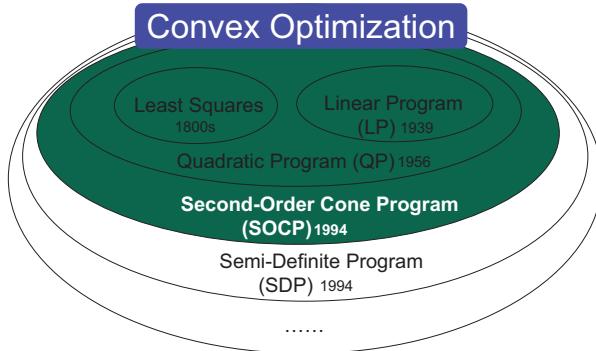
```
import cvxpy as cp
import numpy as np

# optimization variable
x = cp.Variable(3)
# construct (Q,w,A,b,C,e)
Q = np.array([[26, 3, 10],
              [3, 10, 1],
              [10, 1, 6]])
w = np.array([1,3,2])
A = np.array([[1, -4, 3]])
b = np.array([10])
C = np.array([[2, -1, 2]])
e = np.array([3])

# objective function
cost = cp.quad_form(x, Q) + w.T @ x
obj_min = cp.Minimize(cost)
# constraints
constraints = [A @ x <= b, C @ x == e]
# set up a problem
prob = cp.Problem(obj_min,constraints)
# solve the problem
prob.solve()
#print the solution
print('status: ', prob.status)
print('optimal value: ', prob.value)
print('optimal x*: ', x.value)
```

```
status: optimal
optimal value: 10.573694029850747
optimal x*: [-0.27425373 -0.55223881  1.49813433]
```

Here we use a built-in function, `cp.quad_form(x,Q)`, to compute  $x^T Q x$ . `w.T @ x` indicates  $w^T x$ . The syntax for constraints is also simple.



**Figure 1.30.** Hierarchy of convex optimization problems.

**Look ahead** So far, we have studied three instances of convex optimization: LP, LS and QP. In the next section, we will embark on the follow-up instance: Second-Order Cone Program (SOCP). Recall the hierarchy of convex optimization problems in Fig. 1.30.

## 1.12 Second-order Cone Program

---

**Recap** So far we have studied three instances of convex optimization: LP, LS and QP. In the last section, we studied QP, investigating a special case of QP in which there is a closed-form solution: the *equality-constrained LS*. In particular we emphasized that the KKT matrix that appears in the closed-form solution is one of the components that arises in the KKT conditions that we will study in depth in Part II.

**Outline** In this section, we are going to study a follow-up instance that includes LP, LS and QP as special cases: Second-Order Cone Program, SOCP for short. What we are going to do are four folded. First we will study what SOCP is together with the verification of the convexity of the problem. We will then demonstrate that it subsumes LP and QP. Next, we will discuss applications in which SOCP plays a role and therefore one can see the efficacy of the problem. Lastly, we will investigate CVXPY implementation.

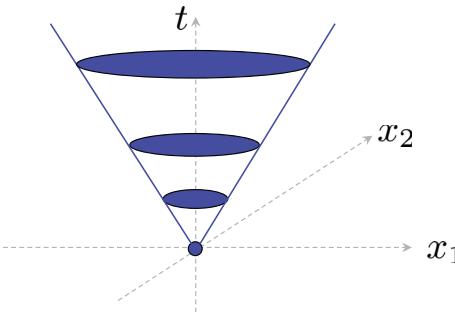
**Second-Order Cone Program (SOCP)** The standard form of SOCP is as follows:

$$\begin{aligned} \min_{x \in \mathbb{R}^d} & w^T x : \\ & \|A_i x - b_i\| \leq c_i^T x + e_i, \quad i \in \{1, \dots, m\}, \\ & Fx = g \end{aligned} \tag{1.179}$$

where  $A_i \in \mathbb{R}^{k_i \times d}$ ,  $b_i \in \mathbb{R}^{k_i}$ ,  $c_i \in \mathbb{R}^d$ ,  $e_i \in \mathbb{R}$ ,  $F \in \mathbb{R}^{p \times d}$ , and  $g \in \mathbb{R}^p$ . The operator  $\|\cdot\|$  indicates the Euclidean norm. By convention, people use the expression  $c_i^T x + e_i$  instead of  $c_i^T x - e_i$  (which is consistent with  $A_i x - b_i$ ). Here the complicated-looking inequality constraint is the one that you have never seen. Let us first verify that the problem belongs to convex optimization. To this end, we need to show that the left-hand-side function in the standard form (in which the right-hand-side is 0 with the “ $\leq$ ” type inequality) is convex:

$$\|A_i x - b_i\| - c_i^T x - e_i.$$

Notice that the latter term  $-c_i^T x - e_i$  in the above is affine and also the inside term of the Euclidean norm is affine. Since convexity preserves under addition and affine transformation, it suffices to show that  $\|x\|$  is convex. In the one-dimensional case, the function  $\|x\|$  is “V”-shaped, so it is convex. It turns out it is the case for an arbitrary dimension. Please check this by yourself.



**Figure 1.31.** Illustration of the second-order cone:  $\{(x, t) \in \mathbb{R}^{d+1} : \|x\| \leq t\}$  and  $d = 2$ .

**Why do we call SOCP?** As you may guess, the naming comes from the never-seen inequality constraint:

$$\|A_i x - b_i\| \leq c_i^T x + e_i. \quad (1.180)$$

To see the rationale behind the naming, let us consider a very simple setting which can give a hint:  $A_i = I$ ,  $b_i = 0$ ,  $c_i = 0$ ,  $e_i = t$ . In this case, the constraint is simplified as:  $\|x\| \leq t$ . Now consider a set of  $(x, t) \in \mathbb{R}^{d+1}$  subject to the constraint (1.180).

$$\mathcal{C} := \{(x, t) \in \mathbb{R}^{d+1} : \|x\| \leq t\}. \quad (1.181)$$

Take a look at the shape of the set, illustrated in Fig. 1.31. It looks like an ice-cream cone. Also the norm that appears in the set is the Euclidean norm, which is the  $\ell_2$  norm. Hence, it is called the *second-order cone (SOC)*. Another names are quadratic cone, ice-cream cone or Lorentz cone.

Since the constraint (1.181) is a special case of the original constraint (1.180), you may still wonder why the problem (1.179) is called SOCP. Here a key observation is that the set of affine transformation of  $x$  is an SOC:

$$(A_i x - b_i, c_i^T x + e_i) \in \mathcal{C}.$$

Since convexity preserves under affine transformation, one can also view the original constraint (1.180) as an SOC upto affine transformation. So one can interpret the problem (1.179) as an SOC-constraint-based Program, which can be simply called SOCP.

**Subsumes QP: QP → SOCP** Let us show that the problem (1.179) includes LP and QP as special cases. One can immediately see the inclusion of LP by setting:

$$A_i = 0 \quad \forall i \in \{1, \dots, m\}$$

in the original problem.

The proof of the inclusion of QP is slightly involved. To this end, we will show that any QP can be cast into the form of SOCP. So let us start with the standard form of QP:

$$\begin{aligned} \min w^T x + x^T Qx : \\ Ax - b \leq 0 \\ Cx - e = 0 \end{aligned} \tag{1.182}$$

where  $Q \in \mathbf{R}^{d \times d} \succeq 0$ ,  $A \in \mathbf{R}^{m \times d}$  and  $C \in \mathbf{R}^{p \times d}$ . Here what is annoying is the quadratic term  $x^T Qx$  that appears in the objective function. In an effort to translate the annoying term into an affine term, let us first manipulate the matrix  $Q$  via eigenvalue decomposition (EVD). Since  $Q$  is *symmetric*, one can apply EVD to obtain:

$$Q = U\Lambda U^T$$

where  $U \in \mathbf{R}^{d \times d}$  is a unitary matrix (i.e.,  $U^T U = I$ ) and  $\Lambda$  is a diagonal matrix:  $\Lambda = \text{diag}(\lambda_1, \dots, \lambda_d)$  where  $\lambda_i$  indicates the  $i$ th eigenvalue of  $Q$ . Now we define  $y := Q^{1/2}x$  where

$$Q^{1/2} := U\Lambda^{1/2}U^T$$

where  $\Lambda^{1/2} := \text{diag}(\sqrt{\lambda_1}, \dots, \sqrt{\lambda_d})$ . This definition is valid because  $\lambda_i$ 's are non-negative due to  $Q \succeq 0$ . This then yields:  $y^T y = x^T Qx$ . Introducing this new variable into (1.182), we get:

$$\begin{aligned} \min_{x, y} w^T x + y^T y : \\ Ax - b \leq 0 \\ Cx - e = 0 \\ y = Q^{1/2}x. \end{aligned} \tag{1.183}$$

While the newly introduced constraint  $y = Q^{1/2}x$  is okay as it is *affine*, the quadratic term  $y^T y$  in the objective is still problematic. To translate this into an affine term, we introduce a new variable, say  $t$ , such that

$$t \geq y^T y. \tag{1.184}$$

Here the key observation is that minimizing  $t$  is equivalent to minimizing  $y^T y$ . Why? By minimizing  $t$ , one can set the upper bound of  $y^T y$  smaller, hence one can

reduce  $y^T y$  upto the limit. Similarly by minimizing  $y^T y$ , we can choose  $t$  (our optimization variable) so that it exactly touches upon the minimized  $y^T y$ . Therefore, we can replace  $y^T y$  in the objective with  $t$  while adding the constraint (1.184), thus obtaining:

$$\begin{aligned} \min_{x, y, t} \quad & w^T x + t : \\ Ax - b \leq 0 \\ Cx - e = 0 \\ y = Q^{1/2}x \\ y^T y \leq t. \end{aligned} \tag{1.185}$$

**Is  $y^T y \leq t$  an SOC constraint?** Is the newly introduced constraint  $y^T y \leq t$  an SOC? At first glance, it looks not the case, as it can be represented as:  $\|y\| \leq \sqrt{t}$ . Notice that  $\sqrt{t}$  is *not affine* in  $t$ . But it turns out it can be the case with some modification. To see this, let us first manipulate it as:  $4\|y\|^2 \leq 4t$ . An interesting trick is to represent  $4t$  as  $(t+1)^2 - (t-1)^2$ . This then yields:

$$4\|y\|^2 + (t-1)^2 \leq (t+1)^2.$$

Observe in the above that we have *square* exponents in every term. What does this remind you of? It reminds you of the *Gauss's trick* that we have seen multiple times: representing the sum of squares as the Euclidean norm of a stacked matrix. In other words, we employ the trick to obtain:

$$\left\| \begin{bmatrix} 2y \\ t-1 \end{bmatrix} \right\|^2 \leq (t+1)^2.$$

Dropping the square exponents in both sides and then using the definition (1.181) of SOC (together with the non-negativity of  $t$ ), we see that the set of  $\left( \begin{bmatrix} 2y \\ t-1 \end{bmatrix}, t+1 \right)$  (affine transformation of the variables) is an SOC:

$$\left( \begin{bmatrix} 2y \\ t-1 \end{bmatrix}, t+1 \right) \in \mathcal{C}.$$

Hence, we obtain the following SOCP (from QP):

$$\begin{aligned} \min_{x, y, t} \quad & w^T x + t : \\ Ax - b \leq 0 \end{aligned}$$

$$Cx - e = 0$$

$$y = Q^{1/2}x \text{ (affine)}$$

$$\left\| \begin{bmatrix} 2y \\ t - 1 \end{bmatrix} \right\| \leq t + 1 \text{ (SOC).} \quad (1.186)$$

**Applications** You may wonder why we care about SOCP. One obvious reason is that it has many applications like LP and LS. In particular, here we highlight two specific settings in which the problem is instrumental.

The first setting represents *practically-relevant scenarios* in which there is *uncertainty* in data and/or parameters. For example, in the legitimate-vs-spam email classification, data points can be viewed as sort of random quantities. It turns out that taking this *probabilistic* aspect into account, one can modify the original LP (that we formulated in Section 1.5) into an SOCP. In fact, such a modified LP is categorized into a broader class of LPs, called *robust LP*, which covers all the probabilistic variations of LPs (Ben-Tal and Nemirovski, 1998; El Ghaoui and Lebret, 1997).

Another example is the LS problem with an *uncertain* matrix  $A$ . For instance, in the CT application that we investigated in Section 1.10, the matrix  $A$  contains the length information of a beam trajectory that traverses small grids. Since the length is a *measured* quantity, it may contain some *measurement noise*, thus incurring some uncertainty. Taking this aspect, one can modify the original LS into an SOCP.

The second setting concerns scenarios in which optimization problems are formulated with Euclidean norms. Examples include: (i) distance-minimizing location planning in which one wants to locate a warehouse so as to serve many service locations while minimizing the transportation cost, which is usually proportional to the Euclidean distance (Drezner and Hamacher, 2004); (ii) image denoising wherein the task is to remove the noise effect on the edges of an image while incorporating a sort of regularization term which involves an Euclidean norm; and (iii) penalized LS in which one wants to minimize a noise effect while adding an Euclidean-norm-associated term (in the object function) for the purpose of penalizing the noise effect.

Here we will not cover all of the above applications due to the interest of focus. Instead we are going to cover one application: robust LP.

### An example of robust LP: Chance Program (CP) (Geletu et al., 2013)

The application that we would like to put an emphasis on is a prominent example of robust LP, named *Chance Program (CP)*. For illustrative purpose, let us consider

a simple LP containing only one inequality constraint:

$$\min_{x \in \mathbf{R}^d} w^T x : \quad \textcolor{red}{a}^T x \leq b \quad (1.187)$$

where  $w, a \in \mathbf{R}^d$  and  $b \in \mathbf{R}$ . In the legitimate-vs-spam email classification, here  $a$ , marked in red, indicates a data point. As mentioned earlier, such a data point can be viewed as a *random* quantity. So in this case,  $a$  can be modeled as a *random vector*.

In an effort to deal with *uncertainty*, one may want to instead consider a *probabilistic* constraint that can be stated as:

$$\mathbb{P}(\textcolor{red}{a}^T x \leq b) \geq 1 - \epsilon \quad (1.188)$$

for some small  $\epsilon > 0$ . Actually one can interpret the margin-based linear classifier as *another probabilistic* approach, since outliers are dealt with *margins*. The approach considered herein is a more direct approach that employs the *probability* directly in the constraint. The inequality (1.188) means that the constraint holds with high probability, e.g., with probability  $1 - \epsilon$ . Now the question is: How to compute  $\mathbb{P}(a^T x \leq b)$ ?

**Gaussian approximation for  $\mathbb{P}(a^T x \leq b)$**  In fact, the exact computation is almost impossible in reality as we have no idea of the probability distribution that the vector  $a$  is subject to. One way to handle is to instead *approximate* the computation assuming that the vector  $a$  follows a well-known distribution in which the probability calculation is tractable. One such well-known distribution is the *Gaussian* distribution. The Gaussian distribution is not only computationally tractable, but it also well represents many practical settings.

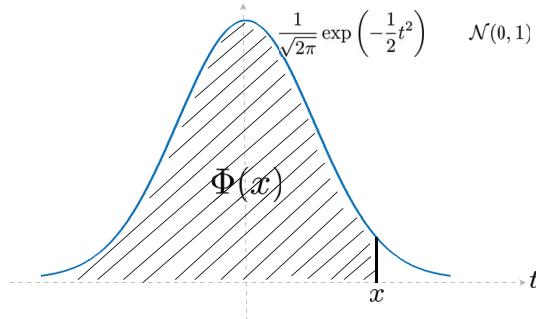
So we will use the Gaussian distribution to approximate the probability computation. Specifically assume that the random vector  $a$  respects the Gaussian distribution like:

$$a \sim \mathcal{N}(\bar{a}, K) \quad (1.189)$$

where  $\bar{a}$  indicates the mean  $\mathbb{E}[a]$  and  $K$  denotes its covariance matrix, defined as  $K := \mathbb{E}[(a - \bar{a})(a - \bar{a})^T]$ . Here the symbol “~” means “is distributed according to”, and  $\mathcal{N}$  denotes the Gaussian (or called Normal) distribution.

Consider a linear combination of  $a$ ,  $a^T x$ , which is of our interest. Under the Gaussian assumption (1.189),  $a^T x$  is also Gaussian:

$$a^T x \sim \mathcal{N}(\bar{a}^T x, x^T K x).$$



**Figure 1.32.** Cumulative density function (CDF)  $\Phi(x)$  of the standard Gaussian distribution  $\mathcal{N}(0, 1)$ .

Why? Check in Prob 4.3. Using this, we can compute:

$$\begin{aligned}\mathbb{P}\left(a^T x \leq b\right) &= \mathbb{P}\left(\frac{a^T x - \bar{a}^T x}{\sqrt{x^T K x}} \leq \frac{b - \bar{a}^T x}{\sqrt{x^T K x}}\right) \\ &= \Phi\left(\frac{b - \bar{a}^T x}{\sqrt{x^T K x}}\right)\end{aligned}\tag{1.190}$$

where  $\Phi(\cdot)$  indicates the cumulative distribution function (CDF) of the *standard* Gaussian distribution (i.e., with mean zero and variance 1):  $\Phi(x) := \mathbb{P}(t \leq x)$  where  $t \sim \mathcal{N}(0, 1)$ ; also see Fig. 1.32 for the illustration of the CDF.

Applying (1.190) into (1.188), we get:

$$\Phi\left(\frac{b - \bar{a}^T x}{\sqrt{x^T K x}}\right) \geq 1 - \epsilon.$$

Since  $\Phi(\cdot)$  is a non-decreasing one-to-one mapping function (again see Fig. 1.32), we can invert the function to get:

$$\frac{b - \bar{a}^T x}{\sqrt{x^T K x}} \geq \Phi^{-1}(1 - \epsilon),$$

which in turns yields:

$$\Phi^{-1}(1 - \epsilon) \sqrt{x^T K x} \leq b - \bar{a}^T x.\tag{1.191}$$

**CP → SOCP** Applying (1.191) to (1.187), we obtain:

$$\min_{x \in \mathbb{R}^d} w^T x : \Phi^{-1}(1 - \epsilon) \sqrt{x^T K x} \leq b - \bar{a}^T x.\tag{1.192}$$

Now the question is: Is the constraint in the above an SOC? To figure this out, let us simplify the constraint by letting  $y := K^{1/2}x$ . Since  $K$  is PSD, one can define  $K^{1/2}$ . We then get:

$$\|y\| \leq \frac{b - \bar{a}^T x}{\Phi^{-1}(1 - \epsilon)}.$$

Here the inequality preserves since  $\Phi^{-1}(1 - \epsilon) > 0$  (due to  $1 - \epsilon > \frac{1}{2}$ ). Notice that the set of affine transformation of the optimization variables is an SOC:

$$\left( y, \frac{b - \bar{a}^T x}{\Phi^{-1}(1 - \epsilon)} \right) \in \mathcal{C}.$$

Hence, we get the following SOCP (from CP):

$$\begin{aligned} \min_{x,y} \quad & w^T x : \\ & y = K^{1/2}x \text{ (affine)} \\ & \|y\| \leq \frac{b - \bar{a}^T x}{\Phi^{-1}(1 - \epsilon)} \text{ (SOC).} \end{aligned} \tag{1.193}$$

*A side note:* One can use the same idea to cover a more general scenario in which there are *multiple* data points. In such a general case, we are able to end up with an SOCP yet with *multiple* SOC constraints. Please check.

**How to solve SOCP?** Like QP, there is no closed-form solution for SOCP in general. So as mentioned earlier, we should rely on strong duality to gain algorithmic insights. Hence, we will study it in depth in Part II.

**CVXPY implementation** Lastly we present how to write a CVXPY script for solving SOCP with the standard form:

$$\begin{aligned} \min_{x \in \mathbf{R}^d} \quad & w^T x : \\ & \|A_i x - b_i\| \leq c_i^T x + e_i, \quad i \in \{1, \dots, m\}, \\ & Fx = g \end{aligned} \tag{1.194}$$

where  $A_i \in \mathbf{R}^{k_i \times d}$ ,  $b_i \in \mathbf{R}^{k_i}$ ,  $c_i \in \mathbf{R}^d$ ,  $e_i \in \mathbf{R}$ ,  $F \in \mathbf{R}^{p \times d}$ , and  $g \in \mathbf{R}^p$ . For instance, we consider a simple case in which  $m = 1$  and the corresponding matrices and

vectors read:

$$w = \begin{bmatrix} 1 \\ 3 \\ 2 \end{bmatrix}, A_1 = \begin{bmatrix} 10 & 2 & -3 \\ 3 & 5 & 1 \\ -1 & 1 & 7 \end{bmatrix}, b_1 = \begin{bmatrix} 2 \\ 4 \\ 7 \end{bmatrix}$$

$$c_1 = \begin{bmatrix} 3 \\ 3 \\ 1 \end{bmatrix}, e_1 = 4, F = [2 \quad -1 \quad 2], g = 3.$$

Here is a code for implementation:

```
import cvxpy as cp
import numpy as np

# optimization variable
x = cp.Variable(3)
# construct (w,A_1,b_1,c_1,e_1,F,g)
w = np.array([1,3,2])
A1 = np.array([[10, 2, -3],
              [3, 5, 1],
              [-1, 1, 7]])
b1 = np.array([2,4,7])
c1 = np.array([3,3,1])
e1 = np.array([4])
F = np.array([[2, -1, 2]])
g = np.array([3])

# objective function
obj_min = cp.Minimize(w.T @ x)
# constraints
soc_constraints = [cp.SOC(c1.T @ x + e1, A1 @ x - b1)]
constraints = soc_constraints + [F @ x == g]
# set up a problem
prob = cp.Problem(obj_min,constraints)
# solve the problem
prob.solve()
#print the solution
print('status: ', prob.status)
print('optimal value: ', prob.value)
print('optimal x*: ', x.value)
```

status: optimal  
optimal value: 0.5920529977190627  
optimal x\*: [ 0.55721217 -0.46268371 0.71144597]

Here we can easily construct the SOC constraint with the built-in function:  
`cp.SOC(c1.T @ w + e1, A1 @ x - b1).`

**Look ahead** So far, we have studied four instances of convex optimization: LP, LS, QP and SOCP. We will embark on the final instance (from this book's point of view) that subsumes all of the prior problems as special cases: Semi-Definite Program (SDP).

## 1.13 Semi-definite Program (SDP)

**Recap** In the last section, we studied SOCP and figured out that the problem is instrumental in very practical contexts in which there is *uncertainty* in data and/or parameters.

**Outline** In this section, we are going to study another instance that includes all of the prior problems as special cases: Semi-Definite Program (SDP). First we will study what SDP is and show that the feasible set in the problem is convex, thus proving the convexity of the problem. Next, we will demonstrate that it indeed subsumes LP, LS, QP and SOCP. Here we will put a particular emphasis on one key lemma that plays a crucial role in translating the prior problems into an SDP: *Schur complement lemma* (Zhang, 2006). We will also prove the lemma accordingly.

**Semi-definite program** The standard form of SDP reads:

$$\begin{aligned} \min_{x \in \mathbb{R}^d} & w^T x : \\ & G + x_1 F_1 + \cdots + x_d F_d \succeq 0 \\ & Cx = e \end{aligned} \tag{1.195}$$

where  $w \in \mathbb{R}^d$ ,  $C \in \mathbb{R}^{p \times d}$ ,  $e \in \mathbb{R}^p$  and  $G, F_i \in \mathbb{R}^{m \times m}$  are *symmetric* matrices. Here the inequality involves a bunch of *matrices* which are related with the components of  $x := (x_1, x_2, \dots, x_d)$  in a *linear* manner. Hence, it is called the *Linear Matrix Inequality (LMI)*.

**Proof of convexity** To prove the convexity of SDP, we need to demonstrate that the following set induced by the inequality constraint is convex:

$$\mathcal{S} := \{x : G + x_1 F_1 + \cdots + x_d F_d \succeq 0\}.$$

The proof is straightforward. Suppose  $x, y \in \mathcal{S}$ . Fix  $\lambda \in [0, 1]$ . Let us check if a convex combination  $\lambda x + (1 - \lambda)y$  is in the set  $\mathcal{S}$ . So we consider:

$$\begin{aligned} & G + (\lambda x_1 + (1 - \lambda)y_1) F_1 + \cdots + (\lambda x_d + (1 - \lambda)y_d) F_d \\ &= \lambda \underbrace{[G + x_1 F_1 + \cdots + x_d F_d]}_{\stackrel{(a)}{\succeq} 0} + (1 - \lambda) \underbrace{[G + y_1 F_1 + \cdots + y_d F_d]}_{\stackrel{(b)}{\succeq} 0} \\ &\stackrel{(c)}{\succeq} 0 \end{aligned}$$

where (a) and (b) come from the hypothesis  $x, y \in \mathcal{S}$ ; and (c) follows from the fact that a convex combination of two PSD matrices is also PSD. Why? Please see

below for the proof. This implies that the convex combination is also in the set:  $\lambda x + (1 - \lambda)y \in \mathcal{S}$ . Hence, this proves the convexity of  $\mathcal{S}$ , thereby showing the convexity of the problem (1.195).

*Proof of the step (c):* For notational simplicity, denote the two interested matrices by  $A, B \in \mathbf{R}^{m \times m} \succeq 0$ . Let  $v \in \mathbf{R}^m$ . Multiplying this and its transpose to the interested linear component on right and left, we get:

$$\begin{aligned} v^T(\lambda A + (1 - \lambda)B)v \\ = \lambda v^T A v + (1 - \lambda)v^T B v \\ \geq 0 \end{aligned} \tag{1.196}$$

where the inequality is due to the hypothesis of  $A, B \succeq 0$  and  $\lambda \in [0, 1]$ . This completes the proof.

**Subsumes LP, LS and QP** It is trivial to prove the inclusion of LP. Setting  $G$  and  $F_i$ 's as *diagonal* matrices in the problem (1.195), one can reduce the problem into an LP:

$$[G]_{ii} + [F_1]_{ii}x_1 + \cdots + [F_d]_{ii}x_d \geq 0 \quad i \in \{1, 2, \dots, m\} \tag{1.197}$$

where  $[G]_{ii}$  (or  $[F_k]_{ii}$ ) indicates the  $(i, i)$  entry of  $G$  (or  $F_k$ ),  $k \in \{1, 2, \dots, d\}$ . Here we use the fact that a diagonal PSD should have non-negative elements.

As for LS and QP, showing inclusion is almost equally difficult to showing the inclusion of SOCP. Since SOCP is shown to subsume LS and QP, we will focus on proving the inclusion of SOCP.

**Inclusion of SOCP** We will demonstrate that SOCP can be cast into the form of SDP. So let us start with the standard form of SOCP:

$$\begin{aligned} \min_{x \in \mathbf{R}^d} w^T x : \\ \|A_i x - b_i\| \leq c_i^T x + e_i, \quad i \in \{1, \dots, m\}, \\ Fx = g \end{aligned} \tag{1.198}$$

where  $w \in \mathbf{R}^d$ ,  $A_i \in \mathbf{R}^{k_i \times d}$ ,  $b_i \in \mathbf{R}^{k_i}$ ,  $c_i \in \mathbf{R}^d$ ,  $e_i \in \mathbf{R}$ ,  $F \in \mathbf{R}^{p \times d}$ , and  $g \in \mathbf{R}^p$ .

Manipulating the SOC constraint in (1.198), we get:

$$(c_i^T x + e_i)^2 \geq \|A_i x - b_i\|^2. \tag{1.199}$$

Notice that  $c_i^T x + e_i$  is non-negative for a feasible  $x$ . This is due to the inequality constraint in (1.198); otherwise  $x$  is not feasible. Also, without loss of generality, we can assume that  $c_i^T x + e_i > 0$ . Otherwise, the constraint becomes  $\|A_i x - b_i\| = 0$ .

Then, it becomes an equality constraint, so it can be merged into  $Fx = g$ . Hence, one can divide  $c_i^T x + e_i$  on both sides to get:

$$c_i^T x + e_i \geq \frac{\|A_i x - b_i\|^2}{c_i^T x + e_i}. \quad (1.200)$$

An alternative yet insightful expression of (1.200) reads:

$$(c_i^T x + e_i) - (A_i x - b_i)^T \{(c_i^T x + e_i) I_{k_i \times k_i}\}^{-1} (A_i x - b_i) \geq 0 \quad (1.201)$$

where  $I_{k_i \times k_i}$  denotes the  $k_i$ -by- $k_i$  identity matrix. Here a key observation is that the left-hand-side in (1.201) is the very famous *Schur complement* of the matrix  $(c_i^T x + e_i) I_{k_i \times k_i}$ . See below for the definition of the Schur complement.

**Definition: (Schur complement)** Suppose that  $p$  and  $q$  are non-negative integers, and  $A \in \mathbb{R}^{p \times p}$ ,  $B \in \mathbb{R}^{p \times q}$ ,  $C \in \mathbb{R}^{q \times q}$ . Let

$$X := \begin{bmatrix} A & B \\ B^T & C \end{bmatrix} \in \mathbb{R}^{(p+q) \times (p+q)}. \quad (1.202)$$

If  $A$  is invertible, then the Schur complement  $S$  of the block  $A$  in matrix  $X$  is defined as:

$$S := C - B^T A^{-1} B. \quad (1.203)$$

We see the following mapping:  $A = (c_i^T x + e_i) I_{k_i \times k_i}$  (marked in blue),  $B = A_i x - b_i$  (marked in red) and  $C = c_i^T x + e_i$  (marked in green).

In fact, there is a very famous lemma concerning Schur complement, which plays a key role in translating (1.201) into the standard form of inequality constraints that appear in SDP. That is, *Schur complement lemma*, formally stated below.

**Schur complement lemma:** Suppose  $A \in \mathbb{R}^{p \times p}$  is positive definite, i.e.,  $v^T A v > 0$  for all  $v \in \mathbb{R}^p$ . It is simply denoted by  $A \succ 0$ . Also suppose  $C$  is symmetric. Then,

$$X := \begin{bmatrix} A & B \\ B^T & C \end{bmatrix} \succeq 0 \iff S := C - B^T A^{-1} B \succeq 0. \quad (1.204)$$

**Proof:** We will present it at the end of this section. ■

From the mapping  $A = (c_i^T x + e_i) I_{k_i \times k_i}$  (marked in blue),  $B = A_i x - b_i$  (marked in red) and  $C = c_i^T x + e_i$  (marked in green), we can write down the SOC constraint (1.201) as an LMI of our desired form:

$$F_i(x) := \begin{bmatrix} (c_i^T x + e_i) I_{k_i \times k_i} & A_i x - b_i \\ (A_i x - b_i)^T & c_i^T x + e_i \end{bmatrix} \succeq 0. \quad (1.205)$$

Note that the matrix  $F_i(x) \in \mathbf{R}^{(k_i+1) \times (k_i+1)}$  is symmetric and its entries are *affine* in  $x$ . The question of interest is then: Is  $F_i(x) \succeq 0$  an LMI? It turns out the answer is yes. In fact, one can readily prove it by showing that all the matrices associated with  $x_i$ 's in (1.205) are symmetric. See below for the proof.

**Proof:**  $F(x) := \begin{bmatrix} (c^T x + e)I_{k \times k} & Ax - b \\ (Ax - b)^T & c^T x + e \end{bmatrix} \succeq 0$  is an LMI. For notational simplicity, we drop the subscript index  $i$  in every place. Also we use a different notation, say  $k$ , to indicate the row dimension of the matrix  $A$ . We first write the interested matrix  $F(x)$  as the sum of two separated matrices:

$$\begin{bmatrix} (c^T x + e)I_{k \times k} & Ax - b \\ (Ax - b)^T & c^T x + e \end{bmatrix} = \begin{bmatrix} eI_{k \times k} & -b \\ -b^T & e \end{bmatrix} + \begin{bmatrix} c^T x I_{k \times k} & Ax \\ x^T A^T & c^T x \end{bmatrix}. \quad (1.206)$$

Here the first matrix in the RHS is symmetric. Let  $A = [a_1 \ a_2 \ \dots \ a_d] \in \mathbf{R}^{k \times d}$  and  $c = [c_1, c_2, \dots, c_d]^T \in \mathbf{R}^d$  where  $a_i \in \mathbf{R}^k$  and  $c_i \in \mathbf{R}$ . Using these notations, one can represent the second matrix in the RHS as:

$$\begin{aligned} \begin{bmatrix} c^T x I_{k \times k} & Ax \\ x^T A^T & c^T x \end{bmatrix} &= \begin{bmatrix} \sum_{i=1}^d c_i x_i I_{k \times k} & \sum_{i=1}^d a_i x_i \\ \sum_{i=1}^d a_i^T x_i & \sum_{i=1}^d c_i x_i \end{bmatrix} \\ &= \sum_{i=1}^d x_i \begin{bmatrix} c_i I_{k \times k} & a_i \\ a_i^T & c_i \end{bmatrix}. \end{aligned}$$

Notice that the matrices  $\begin{bmatrix} c_i I_{k \times k} & a_i \\ a_i^T & c_i \end{bmatrix}$ 's are symmetric for all  $i$ . This together with the fact that the first matrix in the RHS of (1.206) is symmetric argues that  $F(x) \succeq 0$  is an LMI, thus completing the proof.

**SOCP → SDP** We have figured out that  $F_i(x) \succeq 0$  is an LMI for all  $i$ . So we have *multiple* LMIs, while we need a *single* LMI to meet the standard form of SDP. Here the good news is that such multiple LMIs can be merged into a single LMI. The trick is the following:

$$F_1(x), \dots, F_m(x) \succeq 0 \iff F(x) := \begin{bmatrix} F_1(x) & 0 & \cdots & 0 \\ 0 & F_2(x) & \cdots & \vdots \\ \vdots & \vdots & \ddots & 0 \\ 0 & \cdots & 0 & F_m(x) \end{bmatrix} \succeq 0. \quad (1.207)$$

The proof of (1.207) is straightforward. It requires only the definition of PSD – please exercise by yourself. Using this, one can rewrite the problem (1.198) as:

$$\begin{aligned} \min_{x \in \mathbb{R}^d} w^T x : \\ F(x) \succeq 0 \\ Fx = g. \end{aligned} \tag{1.208}$$

Hence, we show that SOCP can be translated into SDP, thus proving the inclusion of SOCP.

**Recall “Schur complement lemma”** Suppose  $A \succ 0$  and  $C$  is symmetric. Then,

$$X := \begin{bmatrix} A & B \\ B^T & C \end{bmatrix} \succeq 0 \iff S := C - B^T A^{-1} B \succeq 0. \tag{1.209}$$

The proof requires the implication of two directions. Let us do it one by one.

**Proof of the forward direction** Notice that the starting point is  $X \succeq 0$ . This together with the definition of PSD motivates us to ponder on the following function:

$$f(x, y) = [x^T \ y^T] \begin{bmatrix} A & B \\ B^T & C \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \tag{1.210}$$

where  $x \in \mathbb{R}^p$  and  $y \in \mathbb{R}^q$ . A key observation w.r.t. the function  $f(x, y)$  allows us to see the relationship with the complement  $S$  of our interest, thereby proving the forward direction. To figure out what it means, let us first rewrite (1.210) as:

$$f(x, y) = x^T Ax + 2x^T By + y^T Cy. \tag{1.211}$$

Here the key observation is that  $f(x, y)$  is *strictly convex in  $x$* . Why? Because of  $A \succ 0$  (from the given hypothesis) as well as the 2nd order condition of convexity. Hence,  $f(x, y)$  has the minimum and it is achieved at the point that satisfies  $\nabla_x f(x^*, y) = 0$ . A straightforward calculation then gives:  $x^* = -A^{-1}By$ .

In an effort to relate this key observation to the complement  $S$  of our interest, let us define the minimum as:

$$g(y) := \min_x f(x, y).$$

Plugging  $x^* = -A^{-1}By$  into the above, we can obtain:

$$\begin{aligned} g(y) &= f(x^*, y) \\ &= (A^{-1}By)^T A(A^{-1}By) - 2(A^{-1}By)^T By + y^T Cy \\ &= y^T(C - B^T A^{-1}B)y \\ &= y^T S y \end{aligned} \tag{1.212}$$

where the second last step comes from  $(A^{-1})^T = A^{-1}$  ( $A$  is symmetric and invertible).

This together with (1.210) enables us to easily prove the forward direction. Suppose that the interested matrix  $X$  is PSD. Then, the formula (1.210) implies that  $f(x, y) \geq 0$  for all  $x, y$ . Hence, one can also say that for a particular  $x$ , say  $x^*$ , the function is non-negative:  $f(x^*, y) \geq 0 \quad \forall y$ . This together with  $g(y) = f(x^*, y) = y^T S y$  (see (1.212)) yields:  $y^T S y \geq 0$  for all  $y$ . This implies  $S \succeq 0$ , thus completing the proof.

**Proof of the reverse direction (converse)** Given the above formulas (1.210) and (1.212), the converse proof is also straightforward. Suppose  $S \succeq 0$ . Then, by the definition of PSD,  $0 \leq g(y)$  for all  $y$  (see (1.212)). This together with the definition  $g(y) := \min_x f(x, y)$  then gives:

$$0 \leq g(y) \leq f(x, y) \quad \forall x, y. \tag{1.213}$$

This implies  $X \succeq 0$  (see (1.210)), thus completing the proof.

**Look ahead** One natural question is: Why do we care about SDP? One obvious reason is that SDP has many applications. In particular, SDP plays a crucial role in *approximating* difficult non-convex optimization problems via a famous technique, called *SDP relaxation*. In the next section, we will explore SDP relaxation in depth.

## 1.14 SDP Relaxation

---

**Recap** In the previous section, we have studied what SDP is, and showed in particular that it subsumes SOCP using a variety of interesting techniques. One key lemma that we put a particular emphasis on was: Schur complement lemma. We also proved the lemma. At the end, we claimed that SDP has many interesting applications.

**Outline** In this section, we are going to discuss such applications. Specifically what we are going to do are four folded. First we will discuss two specific settings in which SDP plays a powerful role. We will then focus on one particular technique that is very instrumental in addressing many difficult non-convex problems. The technique is *SDP relaxation*. Next, we will study SDP relaxation in depth in the context of one certain problem, called the MAXCUT problem (Karp, 1972). Lastly we will present how to do CVXPY implementation for SDP.

**Two settings of SDP's interest** The first is the setting which concerns very difficult non-convex problems. As mentioned earlier, SDP relaxation plays a role in tacking the difficult problems. In fact, it serves to *approximate* them.

The second is the problem context in which the maximum eigenvalue of a matrix or the nuclear norm are interested entities that we wish to minimize. Here the nuclear norm, denoted by  $\|A\|_*$ , is defined as:  $\|A\|_* := \sum_i \sigma_i(A)$  where  $\sigma_i(A)$  indicates the  $i$ th singular value of  $A$ . One of the recent popular applications where such problems arise is: *matrix completion* in which one wishes to identify missing entries of a matrix only from partially revealed entries (Candès and Recht, 2009).

Here we will discuss only one thing in depth: SDP relaxation. We will study SDP relaxation in the context of one specific yet famous problem: the MAXCUT problem.

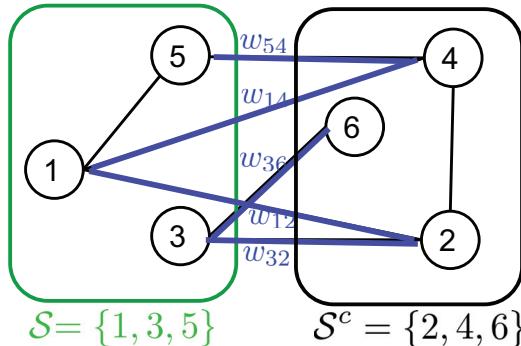
*A side note:* For those who forgot about the concept of singular values, let us leave some details. For a matrix  $A \in \mathbf{R}^{m \times n}$ , a non-negative real number  $\sigma$  is a singular value of  $A$  if and only if there exist vectors  $u \in \mathbf{R}^m$  and  $v \in \mathbf{R}^n$  such that  $Av = \sigma u$  and  $A^T u = \sigma v$ . A way to find the singular values and the corresponding vectors are as follows. Let  $r = \min(m, n)$ . We consider  $A^T A \in \mathbf{R}^{n \times n}$ . Since  $A^T A \succeq 0$ , we can obtain the eigenvalue decomposition as:

$$A^T A = V \Sigma^2 V^T \quad (1.214)$$

where  $V \in \mathbf{R}^{n \times r}$  is a unitary matrix and  $\Sigma := \text{diag}(\sigma_1, \dots, \sigma_r) \in \mathbf{R}^{r \times r}$ . For  $AA^T \in \mathbf{R}^{m \times m}$ , we can do the same thing to get:

$$AA^T = U \Sigma^2 U^T \quad (1.215)$$

where  $U \in \mathbf{R}^{m \times r}$  is a unitary matrix.



**Figure 1.33.** MAXCUT problem: Finding a set that maximizes a cut. In this example, the set  $S = \{1, 3, 5\}$  and the cut w.r.t. the set  $S$  is  $w_{54} + w_{14} + w_{36} + w_{12} + w_{32}$ .

**MAXCUT problem (Karp, 1972)** The goal of the problem is to find a set that maximizes a cut. To understand what it means, we need to know about the concepts of three things: (i) set; (ii) weight; and (iii) cut. The context in which the problem is defined is a graph  $\mathcal{G}$  which consists of a vertex set  $\mathcal{V}$  and an edge set  $\mathcal{E}$ . The problem is concerned about a *undirected* graph in which each edge does not have a direction, meaning that one edge in  $\mathcal{E}$ , say  $(1, 2)$ , is the same as its counterpart  $(2, 1)$ . For the example in Fig. 1.33, the edge set reads:

$$\mathcal{E} = \{(1, 2), (1, 4), (1, 5), (2, 3), (2, 4), (3, 6), (4, 5)\}.$$

Here what it means by a set  $S$  is a subset of the vertex set  $\mathcal{V}$ . For example, in Fig. 1.33, the set is  $S = \{1, 3, 5\} \subset \mathcal{V}$ . The weight is a real value that is associated with an edge. It is denoted by  $w_{ij}$  for an edge  $(i, j) \in \mathcal{E}$ . The cut is defined as the aggregation of all the weights of the edges that come across the set  $S$  and its complement  $S^c$ . In the example of Fig. 1.33, the crossing edges are:  $\{(5, 4), (1, 4), (3, 6), (1, 2), (3, 2)\}$ . Hence the cut w.r.t. the set  $S$  is:

$$\text{Cut}(S) = w_{54} + w_{14} + w_{36} + w_{12} + w_{32}. \quad (1.216)$$

**Optimization for MAXCUT** To formulate an optimization problem for MAXCUT, we first need to come up with a proper optimization variable. Obviously the optimization variable should serve to make a choice of a set  $S$ . Hence, we consider the following variable  $x_i$  such that it indicates whether node  $i$  is in the set  $S$ :

$$x_i = \begin{cases} +1, & \text{if } x_i \in S; \\ -1, & \text{otherwise.} \end{cases} \quad (1.217)$$

Here a key observation is that when  $x_i \neq x_j$ , the edge  $(i, j)$  (if exists) crosses the two sets  $S$  and  $S^c$ , and hence, this should contribute to  $\text{Cut}(S)$  by the amount of

$w_{ij}$ . For  $(i, j) \notin \mathcal{E}$ , we set  $w_{ij} = 0$ . On the other hand, when  $x_i = x_j$  (meaning the edge  $(i, j)$  does not cross the sets), there should be no contribution to the cut. This motivates us to formulate the following optimization:

$$\max_{x_i} \sum_{i=1}^d \sum_{j=1}^d \frac{1}{2} w_{ij} (1 - x_i x_j) : \\ x_i^2 = 1, \quad i \in \{1, \dots, d\} \quad (1.218)$$

where  $d$  indicates the size of the vertex set. Notice in the objective function that we get  $w_{ij}$  whenever  $x_i \neq x_j$ ; 0 otherwise. The constraint  $x_i^2 = 1$  respects the fact that  $x_i$  is only either +1 or -1.

**A translation technique: Lifting** Observe in the objective function (1.218) that we have a *quadratic* term like  $x_i x_j$ . Also we have a *quadratic* equality-constraint. So these do not match the standard form of any convex instance that we have studied thus far.

In an effort to translate such undesirable terms into favourable terms (e.g., *affine* terms), we introduce a well-known technique, called *lifting*. Here the lifting means raising a space that the optimization variable lives in. In the considered example, the optimization variables  $x_i$ 's can be represented as a vector, like:  $x := [x_1, \dots, x_d]^T$ . So the lifting in this context is to convert the vector  $x$  into a higher dimensional entity, say a matrix. For instance, one may introduce a new matrix, say  $X$ , such that its  $(i, j)$ -entry  $[X]_{ij}$  is defined as:

$$X_{ij} := x_i x_j. \quad (1.219)$$

We can then represent  $X$  in a very succinct way:

$$X = \begin{bmatrix} X_{11} & X_{12} & \cdots & X_{1d} \\ X_{21} & X_{22} & \cdots & X_{2d} \\ \vdots & \vdots & \ddots & \vdots \\ X_{d1} & X_{d2} & \cdots & X_{dd} \end{bmatrix} = \begin{bmatrix} x_1 x_1 & x_1 x_2 & \cdots & x_1 x_d \\ x_2 x_1 & x_2 x_2 & \cdots & x_2 x_d \\ \vdots & \vdots & \ddots & \vdots \\ x_d x_1 & x_d x_2 & \cdots & x_d x_d \end{bmatrix} \quad (1.220)$$

$$= \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_d \end{bmatrix} \begin{bmatrix} x_1 & x_2 & \cdots & x_d \end{bmatrix} = x x^T.$$

In fact, there is one thing that we need to worry about whenever doing change of variables. That is, the *constraint* that is newly imposed by the introduction of a new variable. Here a new constraint kicks in. To figure this out, notice that the matrix

$X$  is of rank 1. Why? Remember the eigenvalue decomposition for a symmetric matrix:  $X = U\Lambda U^T$ . From  $X = xx^T$ , we see that  $X$  has only one eigenvector  $x$ . Its eigenvalue reads  $x^T x$ , since

$$X\mathbf{x} = (xx^T)\mathbf{x} = x(x^T x) = (x^T x)\mathbf{x}. \quad (1.221)$$

Obviously the eigenvalue  $x^T x$  is non-negative. So the change of variable induces the following constraints:

$$X_{ii} = 1, \quad X \succeq 0, \quad \text{rank}(X) = 1. \quad (1.222)$$

Actually the above also implies that  $X_{ii} = 1$  and  $X = xx^T$ . Why? Think about it. Hence, with a new matrix variable  $X$ , the problem (1.218) can be rewritten as:

$$\begin{aligned} p^* := \max_X & \sum_{i=1}^d \sum_{j=1}^d \frac{1}{2} w_{ij} (1 - X_{ij}) : \\ & X_{ii} = 1, \quad i \in \{1, \dots, d\} \quad (\text{affine}) \\ & X \succeq 0 \quad (\text{LMI}) \\ & \text{rank}(X) = 1 \quad (\text{rank constraint}). \end{aligned} \quad (1.223)$$

Notice that  $X \succeq 0$  is an LMI. Why? For example, consider the  $d = 2$  case in which

$$X = \begin{bmatrix} X_{11} & X_{12} \\ X_{12} & X_{22} \end{bmatrix} = X_{11} \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} + X_{12} \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} + X_{22} \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}.$$

It is affine in  $X_{ij}$ 's and the associated matrices are all symmetric.

**SDP relaxation** Notice in (1.223) that the objective function is affine in  $X_{ij}$ , the first equality constraint is affine, and the second inequality constraint is an LMI. However, it contains an undesirable constraint:  $\text{rank}(X) = 1$  (rank constraint). So it is not an SDP.

This is where a technique, called SDP relaxation, comes in. The idea of SDP relaxation is simply to ignore the rank constraint. By ignoring the constraint, the search space in the optimization problem becomes expanded and hence it is indeed *relaxation*. Applying the technique, we get:

$$\begin{aligned} p_{\text{SDP}}^* := \max_X & \sum_{i=1}^d \sum_{j=1}^d \frac{1}{2} w_{ij} (1 - X_{ij}) : \\ & X_{ii} = 1, \quad i \in \{1, \dots, d\} \quad (\text{affine}) \\ & X \succeq 0 \quad (\text{LMI}). \end{aligned} \quad (1.224)$$

Obviously  $p_{\text{SDP}}^* \geq p^*$ , since it is relaxation for the *maximization* problem.

Interestingly, in many cases, the gap between  $p_{\text{SDP}}^*$  and  $p^*$  is not so large. In some cases, the gap can be large. But it was shown by Nesterov (a Russian mathematician who has been playing an important role in the convex optimization field) that the gap is not arbitrarily large (Nesterov, 1998). The worst-case bound was shown to be:

$$\frac{p_{\text{SDP}}^* - p^*}{p_{\text{SDP}}^*} \leq \frac{\pi}{2} - 1 \approx 0.571.$$

The proofs of these are out of the scope of this book.

**How to convert  $X_{\text{SDP}}^*$  into an interested vector?** You may wonder how to convert  $X_{\text{SDP}}^*$  (obtained from (1.224)) into an associated vector of the problem's interest that serves to find a set. This is because  $X_{\text{SDP}}^*$  may not be of the following desired form:  $X_{\text{SDP}}^* = xx^T$ . In such an undesirable yet frequently-occurring case, one way to go is to apply a very well-known technique in statistics: *Principal Component Analysis (PCA)* (Pearson, 1901). The way it works is as follows. We first do eigenvalue decomposition to get:

$$X_{\text{SDP}}^* = U \text{diag}(\lambda_1, \lambda_2, \dots, \lambda_d) U^T \quad (1.225)$$

where  $\lambda_1 \geq \lambda_2 \geq \dots \geq \lambda_d \geq 0$ . Here the number of non-zero eigenvalues determines the rank of the matrix. So the idea is to take only the first (*principal*) largest eigenvalue  $\lambda_1$  while ignoring others to approximate it as:

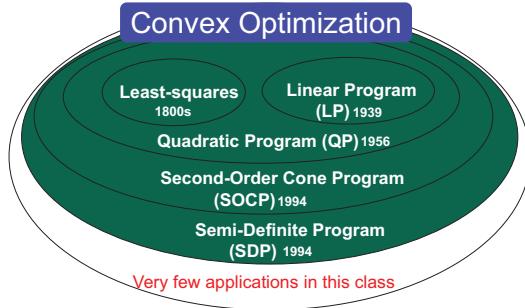
$$\tilde{X}_{\text{SDP}}^* := U \text{diag}(\lambda_1, 0, \dots, 0) U^T. \quad (1.226)$$

This way, we can ensure  $\text{rank}(\tilde{X}_{\text{SDP}}^*) = 1$ , enabling us to obtain the interested vector.

*A note:* Notice that the solution  $\tilde{X}_{\text{SDP}}^*$  is an *approximated* one obtained from sort of a heuristic. So it has nothing to do with respecting the constraints in the optimization problem (1.224). As you may easily image, the approximated solution  $\tilde{X}_{\text{SDP}}^*$  readily violates a constraint like  $X_{ii} = 1$ .

**How to solve general SDP?** Like QP and SOCP, there is no closed-form solution for SDP in general. So as mentioned earlier, we should rely on strong duality and KKT conditions to gain algorithmic insights. Hence, later in Part II, we will study the content in depth.

**CVXPY implementation** Lastly we investigate how to write a CVXPY script for solving SDP. For ease of implementation, we often rely upon another standard form



**Figure 1.34.** Hierarchy of convex optimization problems.

that incorporates the trace operation and a PSD matrix:

$$\begin{aligned} \min_{X \in \mathbf{R}^{n \times n}} & \text{trace}(WX) : \\ & X \succeq 0 \\ & \text{trace}(A_i X) = b_i, \quad i \in \{1, \dots, p\} \end{aligned} \tag{1.227}$$

where  $W, A_i \in \mathbf{R}^{m \times m}$  are *symmetric* matrices and  $b_i \in \mathbf{R}$ . Here the trace operation is defined for a square matrix, say  $A \in \mathbf{R}^{n \times n}$ :

$$\text{trace}(A) := \sum_{i=1}^n A_{ii}, \tag{1.228}$$

where  $A_{ii}$  indicates the  $i$ th diagonal entry of  $A$ . Actually one can represent the objective  $\text{trace}(WX)$  as a linear combination of the optimization variables  $X_{ij}$ 's (the entries of  $X$ ). Similarly for  $\text{trace}(A_i X)$ . Check in Prob 4.9(f). As shown earlier,  $X \succeq 0$  is an LMI.

For code implementation, we consider the following example in which  $n = 3$ ,  $p = 1$  and the corresponding matrices and parameters read:

$$W = \begin{bmatrix} 34 & -2 & -6 \\ -2 & 18 & 15 \\ -6 & 15 & 30 \end{bmatrix}, A_1 = \begin{bmatrix} 19 & 2 & 0 \\ 2 & 6 & 1 \\ 0 & 1 & 22 \end{bmatrix}, b_1 = 10.$$

A code is given by:

```
import cvxpy as cp
import numpy as np

# optimization variable
X = cp.Variable((3,3), symmetric=True)
```

```
# construct (W,A_1,b_1)
W = np.array([[34, -2, -6],
              [-2, 18, 15],
              [-6, 15, 30]])
A1 = np.array([[19, 2, 0],
               [2, 6, 1],
               [0, 1, 22]])
b1 = np.array([10])

# objective function
cost = cp.trace(W @ X)
obj_min = cp.Minimize(cost)
# constraints
ineq_constraints = [X >> 0]
constraints = ineq_constraints + [cp.trace(A1 @ X) == b1]
# set up a problem
prob = cp.Problem(obj_min,constraints)
# solve the problem
prob.solve()
#print the solution
print('status: ', prob.status)
print('optimal value: ', prob.value)
print('optimal X*: ', X.value)
```

```
status: optimal
optimal value: 6.795868612485915
optimal X*: [[ 0.00610944 -0.0485969  0.04867967]
[-0.0485969  0.38656275 -0.38722108]
[ 0.04867967 -0.38722108  0.38788063]]
```

Here we use `cp.trace()` for the trace operation; and use the symbol “ $\gg$ ” to implement the matrix inequality.

**Look ahead** So far, we have studied five instances of convex optimization: LP, LS, QP, SOCP and SDP. In fact, we have more instances which are convex but not belonging to the prior problems. However, there is very little application that such an instance plays a role in. Hence, we stop here. Instead we will focus on studying algorithms for general QP, SOCP and SDP, which we have deferred. So we will embark on Part II to start investigating strong duality.

## Problem Set 4

---

**Prob 4.1 (Invertibility of the KKT matrix)** Consider the equality-constrained least-squares problem:

$$\min \|Ax - b\|^2 : Cx = e \quad (1.229)$$

where  $A \in \mathbf{R}^{m \times d}$  and  $C \in \mathbf{R}^{p \times d}$ . Assume that  $m \geq d$  and  $p \leq d$ . In Section 1.11, we showed that if

$$\text{rank}(C) = p \text{ and } \text{rank} \left( \begin{bmatrix} A \\ C \end{bmatrix} \right) = d, \quad (1.230)$$

then the KKT matrix

$$\begin{bmatrix} 2A^T A & C^T \\ C & 0 \end{bmatrix}$$

is invertible. It turns out the other way around also holds: if the KKT matrix is invertible, then (1.230) holds. Prove the converse.

**Prob 4.2 (KKT equations)** In this problem, you are guided to do the direct derivation of the formula (1.166), which we omitted in Section 1.11. For ease of illustration, let us echo the formula:

$$x^* = d\text{-Components} \left\{ \begin{bmatrix} 2A^T A & C^T \\ C & 0 \end{bmatrix}^{-1} \begin{bmatrix} 2A^T b \\ e \end{bmatrix} \right\} \quad (1.231)$$

where  $d\text{-Components}(\cdot)$  is an operator that takes the first  $d$  components of  $(\cdot)$ , and  $(A, C, b, e)$  are the matrices and vectors associated with the constrained least-squared problem (1.229) in Prob 4.1.

The direct derivation is based on the optimality condition that we did not derive yet, but which can be understood in Part II. In this problem, we will proceed by simply adopting the optimality condition:

$$\nabla f(x^*)^T (x - x^*) \geq 0, \quad \forall x : Cx = e. \quad (1.232)$$

(a) Let  $v = x - x^*$ . Show that the condition (1.232) is equivalent to:

$$\nabla f(x^*)^T v \geq 0, \quad \forall v : Cv = 0. \quad (1.233)$$

(b) For notational simplicity, let us use  $w$  instead of  $\nabla f(x^*)$ . Then, the condition (1.233) reads:

$$w^T v \geq 0 \quad \forall v : Cv = 0. \quad (1.234)$$

Suppose that  $d = 2, p = 1$  and  $C = [1 \ -1]$ . Prove Fact 1 below:

$$\text{Fact 1: (1.234)} \implies w^T v = 0 \quad \forall v : Cv = 0. \quad (1.235)$$

*Hint: You may want to use the proof-by-contradiction: if  $w^T v \neq 0$ , then there exists  $v$  such that  $w^T v < 0$  and  $Cv = 0$ .*

- (c) Suppose that  $d = 4, p = 1$  and  $C = [1 \ -1 \ 0 \ 0]$ . Prove Fact 2 below:

$$\text{Fact 2: (1.235)} \implies w \in \text{range}(C^T). \quad (1.236)$$

- (d) Prove Fact 1 for arbitrary  $(d, p, C)$ .
- (e) Prove Fact 2 for arbitrary  $(d, p, C)$ .
- (f) Using the above proofs, show that there exists  $z \in \mathbf{R}^p$  such that

$$2(A^T A)x^* - 2A^T b + C^T z = 0. \quad (1.237)$$

**Prob 4.3 (Gaussian distribution)** Let  $X$  and  $Y$  be independent continuous random variables with  $f_X(\cdot)$  and  $f_Y(\cdot)$  respectively. Let  $Z = X + Y$ .

- (a) Express the probability density function (pdf)  $f_Z(\cdot)$  in terms of  $f_X(\cdot)$  and  $f_Y(\cdot)$ .
- (b) Compute the two-sided Laplace transform of  $f_Z(\cdot)$  defined as  $F_Z(s) = \int_{-\infty}^{+\infty} e^{-sa} f_Z(a) da$ .
- (c) Suppose that  $X \sim \mathcal{N}(0, \sigma_X^2)$  and  $Y \sim \mathcal{N}(0, \sigma_Y^2)$ . Compute the Laplace transforms of  $f_X(\cdot)$  and  $f_Z(\cdot)$ :  $F_X(s)$  and  $F_Z(s)$ .
- (d) Using the *uniqueness theorem* of Laplace transform (i.e., the transform is one-to-one mapping) argue that the inverse Laplace transform of  $e^{\frac{\sigma^2 s^2}{2}}$  is the Gaussian density with mean 0 and variance  $\sigma^2$ .
- (e) Consider  $X$  and  $Y$  in part (c). Let  $W = aX + bY$  where  $a, b \neq 0$ . Show that  $W$  is Gaussian.

#### Prob 4.4 (Basics)

- (a) Consider a function  $f : \mathbf{R}^d \rightarrow \mathbf{R} : f(x) = \|x\|$ . Show that the function  $f$  is convex.
- (b) Show that  $F_1, F_2 \succeq 0$  if and only if

$$\begin{bmatrix} F_1 & 0 \\ 0 & F_2 \end{bmatrix} \succeq 0.$$

- (c) Suppose that  $F_1, F_2 \succeq 0$ . Show that  $\forall \lambda \in [0, 1]$ ,

$$\lambda F_1 + (1 - \lambda) F_2 \succeq 0.$$

### Prob 4.5 (Linear Matrix Inequality)

- (a) State the definition of a linear matrix inequality (LMI).
- (b) Let  $X \in \mathbf{R}^{2 \times 2}$  be symmetric. Show that  $X \succeq 0$  is an LMI.
- (c) Let  $x, c \in \mathbf{R}^d$ . Show that

$$\begin{bmatrix} (c^T x)I & 0 \\ 0 & c^T x \end{bmatrix} \succeq 0$$

is an LMI.

- (d) Let  $A \in \mathbf{R}^{2 \times 2}$  and  $x \in \mathbf{R}^2$ . Show that

$$\begin{bmatrix} 0 & Ax \\ x^T A^T & 0 \end{bmatrix} \succeq 0$$

is an LMI.

- (e) Let  $A \in \mathbf{R}^{m \times d}$ ,  $x, c \in \mathbf{R}^d$ ,  $b \in \mathbf{R}^m$  and  $e \in \mathbf{R}$ . Show that

$$\begin{bmatrix} (c^T x + e)I & Ax - b \\ (Ax - b)^T & c^T x + e \end{bmatrix} \succeq 0$$

is an LMI.

- (f) Represent the inequality

$$\|Ax - b\| \leq \gamma$$

with  $\gamma > 0$  and a variable  $x$ , as an LMI.

**Prob 4.6 (Robust LS)** Consider an LS problem where  $A$  has some *uncertainty*:

$$A = A_0 + A_1 \delta \quad (1.238)$$

where  $A_0, A_1 \in \mathbf{R}^{m \times d}$  ( $m \geq d$ ) and  $\delta$  is a random variable with zero mean and  $\sigma^2$  variance. The objective function is now a random variable, as it depends on  $\delta$ . We aim at minimizing the *expectation* of such a random function, which can be formulated as:

$$\min_x \mathbb{E} [\|Ax - b\|^2] \quad (1.239)$$

where  $b \in \mathbf{R}^m$  and the expectation is taken over  $\delta$ .

- (a) Show that the problem is *convex*.
- (b) To which class does it belong to? Also cast it into the class.

## Prob 4.7 (Schur complement lemma)

(a) Consider a set:

$$\mathcal{S} = \left\{ x \in \mathbf{R}^6 : x_6 \geq 0, x_4 \geq \frac{x_5^2}{x_6}, x_2 \geq \frac{x_3^2}{x_4 - \frac{x_5^2}{x_6}} \right\}. \quad (1.240)$$

Is the set  $\mathcal{S}$  convex? If so, prove it; otherwise disprove it.

(b) Consider a function  $f : \mathbf{R}^6 \rightarrow \mathbf{R}$ :

$$f(x) = \frac{x_1^2}{x_2 - \frac{x_3^2}{x_4 - \frac{x_5^2}{x_6}}} \quad (1.241)$$

where  $x$  is defined on the set  $\mathcal{S}$  in part (a). Is  $f(x)$  convex in  $x$ ? If so, prove it; otherwise disprove it.

**Prob 4.8 (Generalized Schur complement lemma)** Let  $A \in \mathbf{R}^{n \times n}$  and  $C \in \mathbf{R}^{m \times m}$ . Suppose  $A \succeq 0$  and  $C$  is symmetric. Suppose the eigenvalue decomposition of  $A$  reads:

$$A = U \Sigma U^T \quad (1.242)$$

where  $U \in \mathbf{R}^{n \times n}$  is a unitary matrix, i.e.,  $U^T U = I$  and  $\Sigma := \text{diag}(\lambda_1, \dots, \lambda_n)$ . Define the *pseudo-inverse* of  $A$  as:  $A^\dagger := U \Sigma^{-1} U^T$  where  $\Sigma^{-1}$  is a diagonal matrix whose elements respect:

$$[\Sigma^{-1}]_{ii} := \begin{cases} \lambda_i^{-1} & \text{if } \lambda_i \neq 0; \\ 0 & \text{if } \lambda_i = 0. \end{cases} \quad (1.243)$$

Prove that

$$X := \begin{bmatrix} A & B \\ B^T & C \end{bmatrix} \succeq 0 \iff \quad (1.244)$$

$$S := C - B^T A^\dagger B \succeq 0, \text{ and } Bv \in \text{range}(A) \ \forall v \in \mathbf{R}^m.$$

**Prob 4.9 (Basics on traces)** Consider a square matrix  $A \in \mathbf{R}^{n \times n}$ . Denote the trace of a square matrix by:

$$\text{trace}(A) := \sum_{i=1}^n A_{ii} \quad (1.245)$$

where  $A_{ii}$  indicates the  $i$ th diagonal entry of  $A$ .

(a) Suppose  $A \succeq 0$ . Show that  $A$  can be represented as

$$A = XX^T \quad (1.246)$$

for some  $X \in \mathbf{R}^{n \times n}$ .

(b) Show that  $AA^T \succeq 0$ .

(c) Consider  $A, B \in \mathbf{R}^{n \times n}$ . Show that for  $A \succeq 0$  and  $B \succeq 0$ ,

$$\text{trace}(AB) \geq 0. \quad (1.247)$$

(d) Show that for  $A \in \mathbf{R}^{n \times m}$  and  $B \in \mathbf{R}^{m \times n}$ ,

$$\text{trace}(AB) = \text{trace}(BA). \quad (1.248)$$

(e) Show that for  $A \in \mathbf{R}^{n \times n}$ ,

$$\text{trace}(A) = \sum_{i=1}^n \lambda_i(A) \quad (1.249)$$

where  $\text{trace}(A) := \sum_{i=1}^n A_{ii}$  and  $\lambda_i(A)$  indicates the  $i$ th eigenvalue of  $A$ .

(f) Let  $X, Z \in \mathbf{R}^{n \times n}$  be symmetric matrices. Show that

$$\sum_{i=1}^n \sum_{j=1}^n Z_{ij} X_{ij} = \text{trace}(ZX). \quad (1.250)$$

**Prob 4.10 (A lemma)** Suppose  $X \in \mathbf{R}^{m \times n}$  and  $t \in \mathbf{R}$ . Show that

$$\|X\|_* \leq t \iff$$

$$\exists Y \in \mathbf{R}^{m \times m}, Z \in \mathbf{R}^{n \times n} : \begin{bmatrix} Y & X \\ X^T & Z \end{bmatrix} \succeq 0, \text{ trace}(Y) + \text{trace}(Z) \leq 2t. \quad (1.251)$$

Here  $\|X\|_*$  indicates the nuclear norm:

$$\|X\|_* := \sum_{i=1}^{\min\{m,n\}} \sigma_i(X) \quad (1.252)$$

where  $\sigma_i(X)$  denotes the  $i$ th singular value of  $X$ .

**Prob 4.11 (Matrix completion)** Let  $M \in \mathbf{R}^{m \times n}$  where  $m \geq n$ . Let  $\Omega$  be the set of  $(i, j)$ 's such that the  $(i, j)$  entries of  $M$  are revealed (observed). Suppose that the revealed entries are  $b_{ij}$ 's:

$$M_{ij} = b_{ij}, \quad (i, j) \in \Omega. \quad (1.253)$$

The problem of *matrix completion* is to reconstruct non-revealed missing entries of  $M$  from the revealed ones.

- (a) Suppose that  $m = n = 3$  and the revealed entries are:

$$M = \begin{bmatrix} 1 & * & 2 \\ * & 9 & * \\ 2 & * & * \end{bmatrix} \quad (1.254)$$

where  $(*)$  denotes a missing value. Suppose that  $\text{rank}(M) = 1$ , i.e.,  $M$  is of the form:  $M = axx^T$  where  $a \in \mathbf{R}$ ,  $x \in \mathbf{R}^3$ . Is matrix completion possible? If so, perform matrix completion to find the missing entries of  $M$ . If not, explain why. What if  $\text{rank}(M) = 2$ ?

- (b) It has been shown in the literature that one can perform matrix completion by solving an optimization problem that minimizes the rank of the interested matrix  $M$ , as long as the number of revealed entries is sufficiently large. As a surrogate of the rank, one may often use the nuclear norm:

$$\|M\|_* := \sum_{i=1}^n \sigma_i(M) \quad (1.255)$$

where  $\sigma_i(M)$  denotes the  $i$ th singular value of  $M$ . So the nuclear-norm-based optimization problem reads:

$$\min_M \|M\|_* : M_{ij} = b_{ij} \quad (i, j) \in \Omega. \quad (1.256)$$

Suppose  $M \succeq 0$ . Translate (1.256) into an SDP.

*Hint: You may want to use a trace trick that you proved in Prob 4.9(e).*

- (c) Again consider the optimization problem (1.256). Now suppose  $M$  is a general matrix, neither symmetric nor PSD. Show even in this case that (1.256) can be translated into an SDP.

*Hint: You may want to use the lemma proved in Prob 4.10.*

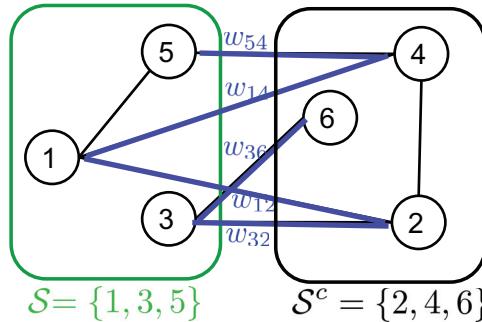
**Prob 4.12 (SDP relaxation)** Consider the MAXCUT problem that we studied in Section 1.14. See Fig. 1.35. The weights  $w_{ij}$ 's associated with edges are given as:

$$(w_{12}, w_{14}, w_{15}) = (1, 2, 6);$$

$$(w_{23}, w_{24}) = (2, 5);$$

$$w_{36} = 1;$$

$$w_{45} = 1.5.$$



**Figure 1.35.** MAXCUT problem: Finding a set that maximizes a cut. In this example, the set  $\mathcal{S} = \{1, 3, 5\}$  and the cut w.r.t. the set  $\mathcal{S}$  is  $w_{54} + w_{14} + w_{36} + w_{12} + w_{32}$ .

Let  $x_i$  denote whether node  $i$  is in the set  $\mathcal{S}$ :

$$x_i = \begin{cases} +1, & x \in \mathcal{S}; \\ -1, & \text{otherwise.} \end{cases} \quad (1.257)$$

- (a) Formulate an optimization problem that intends to find a set that maximizes a cut. Derive the optimal value  $p^*$  and the optimal solution  $x^*$ . You can do this by hand or by computer.
- (b) Formulate an SDP relaxation problem. Solve this problem (derive  $p_{\text{SDP}}^*$ ) using CVXPY. Also write a script for CVXPY implementation. Is  $p_{\text{SDP}}^* = p^*$ ?

**Prob 4.13 (SOCP and/or SDP)** Consider an optimization problem:

$$\begin{aligned} \min w^T x : \\ a^T x \leq b, \quad \forall a \in \mathbf{R}^d : \|a - \bar{a}\|^2 \leq t \end{aligned} \quad (1.258)$$

where  $\bar{a} \in \mathbf{R}^d$ ,  $b \in \mathbf{R}$  and  $t \geq 0$ . Which class does this problem belong to among all the instances that you learned so far? Also cast it into the class.

**Prob 4.14 (True or False?)**

- (a) Consider a Chance Program (CP):

$$\min_{x \in \mathbf{R}^d} w^T x : \mathbb{P}(a^T x \leq b) \geq 1 - \epsilon \quad (1.259)$$

where  $w \in \mathbf{R}^d$ ,  $\epsilon \in (0, 1)$  and  $a$  is a random vector with mean  $\mathbb{E}[a] = \bar{a}$  and covariance matrix  $K = \mathbb{E}[(a - \bar{a})(a - \bar{a})^T]$ . Consider a random variable:

$$Y := \frac{a^T x - \bar{a}^T x}{\sqrt{x^T K x}}. \quad (1.260)$$

Suppose that the cumulative density function (CDF) of  $Y$  and its inverse function are given, i.e., numerical values of CDF evaluated at any point  $Y = y$  (and also their inverse function values) are known. Then, the CP can always be cast into an SOCP.

- (b) Let  $A \in \mathbf{R}^{n \times n}$ . Then, the eigenvalue decomposition of  $A$  reads:

$$A = U\Sigma U^T \quad (1.261)$$

where  $U \in \mathbf{R}^{n \times n}$  is a unitary matrix and  $\Sigma := \text{diag}(\lambda_1, \dots, \lambda_n)$ .

- (c) Consider the MAXCUT problem that we studied in Section 1.14. Using the lifting technique, we formulated the problem as:

$$p^* := \max_X \sum_{(i,j) \in \mathcal{E}} \frac{1}{2} w_{ij}(1 - X_{ij}) :$$

$$X_{ii} = 1, \quad i \in \{1, 2, \dots, d\}, \quad (1.262)$$

$$X \succeq 0,$$

$$\text{rank}(X) = 1$$

where  $w_{ij}$  indicates a weight associated with  $(i, j) \in \mathcal{E}$ ;  $\mathcal{E}$  denotes an edge set in a graph given in the problem; and  $d$  is the number of nodes in the graph. Let  $p_{\text{SDP}}^*$  be the optimal value of an approximated optimization due to SDP relaxation. Then, there exists a rank-1 matrix  $X$  that achieves  $p_{\text{SDP}}^*$ .

- (d) Suppose that  $F(x) \in \mathbf{R}^{m \times m}$  is symmetric and affine in  $x \in \mathbf{R}^d$ . Then,  $F(x) \succeq 0$  is a linear matrix inequality.
- (e) Consider the MAXCUT problem that we studied in Section 1.14. Using the lifting technique, we formulated the problem as:

$$p^* := \max_X \sum_{(i,j) \in \mathcal{E}} \frac{1}{2} w_{ij}(1 - X_{ij}) :$$

$$X_{ii} = 1, \quad i \in \{1, 2, \dots, d\}, \quad (1.263)$$

$$X \succeq 0,$$

$$\text{rank}(X) = 1$$

where  $w_{ij}$  indicates a weight associated with  $(i, j) \in \mathcal{E}$ ,  $\mathcal{E}$  denotes an edge set in a graph given in the problem, and  $d$  is the number of nodes in the graph. Let  $p_{\text{SDP}}^*$  be the optimal value of an approximated optimization due to SDP relaxation. Since the search space in the relaxed optimization is bigger and also the optimization is about maximization,  $p_{\text{SDP}}^* > p^*$ .

## Chapter 2

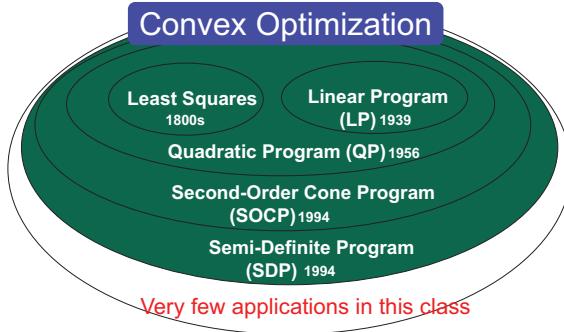
# Duality

### 2.1 Strong Duality

---

**Recap** We have thus far studied several instances of convex optimization problems: LP, Least Squares, QP, SOCP and SDP. Actually we have one more well-known instance which is convex but not belonging to the prior classes. That is, *cone programm* (CP for short). In fact, understanding CP requires lots of mathematical concepts, definitions and techniques, although there are few applications. Hence, we will not go further along this direction. Instead we will focus on studying what we have missed so far. That is, generic algorithms that can be applied to general QP, SOCP and SDP. The reason that we have deferred the content is that algorithms for the general settings are based on *strong duality* and *KKT conditions* that we are supposed to cover in Part II. So from now on, we will move onto Part II to start investigating the contents.

**Outline** In this section, we are going to cover four stuffs. Strong duality is based on the concepts of primal and dual problems. So we will first study what the primal and dual problems are. We will then study what it means by strong duality. Next, we will figure out the KKT conditions and the intimate connection with strong duality. Finally we will understand why they give insights into the design of algorithms. In the next section, we will study an algorithm inspired by them.



**Figure 2.1.** Hierarchy of convex optimization problems.

**Primal & dual problems** Let us start by recalling the standard form of convex optimization:

$$\begin{aligned} \min f(x) : f_i(x) \leq 0, i \in \{1, \dots, m\}, \\ Ax - b = 0 \end{aligned} \tag{2.1}$$

where  $f(x)$  and  $f_i(x)$ 's are *convex* functions,  $A \in \mathbf{R}^{p \times d}$  and  $p < d$ . Without loss of generality, assume that  $\text{rank}(A) = p$ ; otherwise, one can remove dependent rows in  $A$  to make it full-ranked. The primal problem is defined as a problem that we start with, and hence the above is the primal problem.

There is another problem which is intimately related to the primal problem, called the *dual* problem. But to explain what it means, we need to first know about a function, called the *Lagrange function*. The Lagrange function is denoted by  $\mathcal{L}(x, \lambda, v)$ . It takes three arguments. The first is the interested optimization variable  $x$ . The second argument  $\lambda$  is a real-valued vector of size  $m$ , which coincides with the number of inequality constraints:  $\lambda := [\lambda_1, \dots, \lambda_m]^T$ . The last argument  $v$  (pronounced as "nu") is also a real-valued vector yet of different size  $p$ , which is the number of equality constraints:  $v := [v_1, \dots, v_p]^T$ . The Lagrange function is defined as:

$$\mathcal{L}(x, \lambda, v) := f(x) + \sum_{i=1}^m \lambda_i f_i(x) + v^T (Ax - b). \tag{2.2}$$

Notice in the second summation term that  $f_i(x)$  (that appears in the  $i$ th inequality constraint) is multiplied by with  $\lambda_i$ . Similarly the  $i$ th equality-constraint function is multiplied by with  $v_i$  to form the last term  $v^T (Ax - b)$ . Hence,  $\lambda_i$ 's and  $v_i$ 's are called *Lagrange multipliers*.

Are we now ready to define the dual problem? No. To explain what the dual problem is, we need to know about one more function, called the *Lagrange dual function*, or simply the *dual function*. Let us just use the simpler version: the dual function. It is denoted by  $g(\lambda, v)$  and defined as:

$$\begin{aligned} g(\lambda, v) &:= \min_{x \in \mathcal{X}} \mathcal{L}(x, \lambda, v) \\ &= \min_{x \in \mathcal{X}} f(x) + \sum_{i=1}^m \lambda_i f_i(x) + v^T(Ax - b). \end{aligned} \tag{2.3}$$

Two things to note. The first and very important one is that the minimization here is over the *entire space* that  $x$  lies in w.r.t.  $f(x)$  and  $f_i(x)$ 's:  $\mathcal{X} := \mathbf{dom}f \cap \mathbf{dom}f_1 \cap \dots \cap \mathbf{dom}f_m$ . Notice that the search space is *not limited to the feasible set* induced by inequality and equality constraints in the primal problem. The second thing to note is that in general  $\mathcal{L}(x, \lambda, v)$  is not necessarily convex in  $x$ . On the other hand, when  $\lambda_i \geq 0 \forall i$ ,  $\mathcal{L}(x, \lambda, v)$  is simply a summation of convex and affine functions. So in this case, the function is convex. However,  $\lambda_i$ 's could be negative, as there is no sign constraint on  $\lambda$  in defining  $\mathcal{L}(x, \lambda, v)$ . In such a case,  $g(\lambda, v)$  could be  $-\infty$ . For instance, think about a situation in which  $\lambda_1 = -1$ ,  $f_1(x) = e^x$  (convex) and  $\mathcal{X} \in \mathbf{R}$ . In this case, taking  $x = +\infty$  yields  $g(\lambda, v) = -\infty$ .

We are now ready to define the dual problem of our primary interest. Observe in (2.3) that  $g(\lambda, v)$  is a *pointwise minimum* of affine functions (in  $(\lambda, v)$ ) over all  $x$ 's in  $\mathcal{X}$ . Hence, it is *concave* in  $(\lambda, v)$ . Why? Think about what we proved in Prob 1.6(c): the *maximum* of convex functions is *convex*. More generally, one can prove that the maximum of *affine* functions is convex (similarly the minimum of affine functions is *concave*). Someone may still wonder about the above case (2.3) in which the minimum is taken over potentially infinitely many candidates of  $x$  not over only a few candidates. Even in this case, one can readily prove the claim still holds. The proof is not that difficult – think about it. Hence, the maximum is always attained. The dual problem is an optimization problem that intends to find the maximum. So it is formulated as:

$$(\text{Dual problem}): \max_{\lambda, v} g(\lambda, v) : \lambda \geq 0. \tag{2.4}$$

Notice that there is a constraint on  $\lambda$  ( $\lambda \geq 0$ ) while there is none for  $v$ . This together with the definition (2.3) of the dual function gives the following equivalent expression:

$$\max_{\lambda, v} \min_{x \in \mathcal{X}} f(x) + \sum_{i=1}^m \lambda_i f_i(x) + v^T(Ax - b) : \lambda \geq 0. \tag{2.5}$$

**What strong duality means?** Here is a summary of the primal and dual problems:

$$(\text{Primal}): p^* := \min f(x) : f_i(x) \leq 0, i \in \{1, \dots, m\}, Ax - b = 0;$$

$$(\text{Dual}): d^* := \max_{\lambda, v} \min_{x \in \mathcal{X}} f(x) + \sum_{i=1}^m \lambda_i f_i(x) + v^T (Ax - b) : \lambda \geq 0.$$

We denote by  $p^*$  (or  $d^*$ ) the optimal value for the primal (or dual) problem.

Using these, we can now state what strong duality is. What it means is that the optimal values of the two problems are equal:

$$(\text{Strong duality}): p^* = d^*. \quad (2.6)$$

It has been shown that in general, the optimal values are different, i.e., strong duality does not hold. But interestingly strong duality (2.6) does hold for *convex* optimization of our interest, under a very mild condition.<sup>1</sup> We call this the *strong duality theorem*.

Now you may wonder why the strong duality theorem matters in the design of algorithms. The reason is that when strong duality holds, one can derive necessary and sufficient conditions (in order for strong duality to hold), which provide algorithmic insights. So for the rest of this section, we will derive such conditions. For upcoming sections, we will understand why the conditions shed lights as to how to design algorithms. We will then prove the strong duality theorem.

**Necessary conditions for strong duality to hold** Let us first focus on the derivation of necessary conditions. Suppose that strong duality holds  $p^* = d^*$ , and  $x^*$  and  $(\lambda^*, v^*)$  are the optimal solutions of the primal and dual problems, respectively.

Since  $f(x^*) = p^* = d^* = g(\lambda^*, v^*)$  under the hypothesis, we get:

$$\begin{aligned} f(x^*) &= g(\lambda^*, v^*) \\ &\stackrel{(a)}{=} \min_{x \in \mathcal{X}} f(x) + \sum_{i=1}^m \lambda_i^* f_i(x) + v^{*T} (Ax - b) \end{aligned}$$

1. The mild condition says that there exists  $x$  such that strict inequality holds  $f_i(x) < 0 \forall i$  subject to  $Ax = b$ . The condition holds for almost all the problem instances that arise in reality. So one can say that strong duality *usually* holds for convex optimization. We will later discuss on this in detail.

$$\begin{aligned}
&\stackrel{(b)}{\leq} f(x^*) + \sum_{i=1}^m \lambda_i^* f_i(x^*) + v^{*T} (Ax^* - b) \\
&\stackrel{(c)}{\leq} f(x^*)
\end{aligned} \tag{2.7}$$

where (a) is due to the definitions of the dual function (2.3) and the Lagrange function (2.2); (b) comes from the fact that  $x^*$  is a particular choice in view of the minimization problem in step (a); and (c) follows from the fact that  $\lambda_i^* \geq 0$ ,  $f_i(x^*) \leq 0 \forall i$ , and  $Ax^* - b = 0$  (since  $(x^*, \lambda^*)$  must be feasible points).

In the above, the left hand side and the right hand side are the same as  $f(x^*)$ , suggesting that the two inequalities in steps (b) and (c) are tight. From the tightness of the inequality (b), we see that  $x^*$  indeed minimizes  $\mathcal{L}(x, \lambda^*, v^*)$  over  $x$ . Since it is unconstrained, the optimality condition is that its gradient at  $x^*$  is zero:

$$\nabla_x \mathcal{L}(x^*, \lambda^*, v^*) = 0. \tag{2.8}$$

On the other hand, the tightness of the second inequality (c) implies  $\sum_{i=1}^m \lambda_i^* f_i(x^*) = 0$ , which in turn yields:

$$\lambda_i^* f_i(x^*) = 0 \quad \forall i. \tag{2.9}$$

This is because  $\lambda_i^* \geq 0$ ,  $f_i(x^*) \leq 0 \forall i$  for feasible points  $(x^*, \lambda^*)$ . There is a name for this condition. It is called the *complementary slackness* condition. Why do we have the naming? The term  $\lambda^* f_i(x^*)$  captures sort of *slack (gap)* between  $d^* := g(\lambda^*, v^*)$  and  $p^* := f(x^*)$ ; see the 1st, 3rd and 4th line in (2.7). The condition (2.9) implies:  $\forall i$ ,

$$\begin{aligned}
f_i(x^*) < 0 &\implies \lambda_i^* = 0; \\
\lambda_i^* > 0 &\implies f_i(x^*) = 0.
\end{aligned}$$

This says that whenever one of the inequality constraints is strict, the other inequality must be tight, i.e., both are sort of *complementary* in view of ensuring the equality.

The conditions (2.8) and (2.9) together with the constraints in the primal and dual problems then constitute the following necessary conditions for strong duality to hold:

$$\nabla_x \mathcal{L}(x^*, \lambda^*, v^*) = 0; \tag{2.10}$$

$$\lambda_i^* f_i(x^*) = 0 \quad \forall i; \tag{2.11}$$

$$f_i(x^*) \leq 0 \quad \forall i; \quad (2.12)$$

$$Ax^* - b = 0; \quad (2.13)$$

$$\lambda^* \geq 0 \quad (2.14)$$

where (2.12) and (2.13) come from the primal problem and (2.14) is from the dual problem.

In fact, these conditions (2.10)~(2.14) coincide with the ones that we mentioned in Section 1.11 while deriving the closed-form solution for the equality-constrained least squares problem. These are the *KKT conditions*! Remember that the KKT conditions are not limited to convex optimization, but are intended for general convex & non-convex optimization problems. These are necessary conditions for a solution to be optimal for a general optimization problem.

**KKT conditions are also sufficient for strong duality to hold** Interestingly the KKT conditions are also sufficient for strong duality to hold:

$$\text{KKT conditions} \implies p^* = f(x^*) = g(\lambda^*, \nu^*) = d^* \quad (2.15)$$

where  $(x^*, \lambda^*, \nu^*)$  are the points that respect the KKT conditions.

We will prove the following two:  $p^* \leq d^*$  and  $p^* \geq d^*$ . Focus on the former. To this end, we will first show that  $f(x^*) = g(\lambda^*, \nu^*)$ . Recall the definition of the Lagrange function (2.2) to obtain:

$$\begin{aligned} \mathcal{L}(x^*, \lambda^*, \nu^*) &:= f(x^*) + \sum_{i=1}^m \lambda_i^* f_i(x^*) + \nu^{*T} (Ax^* - b) \\ &= f(x^*) \end{aligned} \quad (2.16)$$

where the second equality follows from (2.11) and (2.13). On the other hand, from the definition of the dual function (2.3), we get:

$$\begin{aligned} g(\lambda^*, \nu^*) &:= \min_{x \in \mathcal{X}} \mathcal{L}(x, \lambda^*, \nu^*) \\ &= \min_{x \in \mathcal{X}} f(x) + \sum_{i=1}^m \lambda_i^* f_i(x) + \nu^{*T} (Ax - b) \\ &\stackrel{(a)}{=} f(x^*) + \sum_{i=1}^m \lambda_i^* f_i(x^*) + \nu^{*T} (Ax^* - b) \\ &= \mathcal{L}(x^*, \lambda^*, \nu^*) \end{aligned} \quad (2.17)$$

where (a) comes from the fact that the condition (2.10) in the unconstrained convex minimization suggests that  $x^*$  is the minimizer of  $\mathcal{L}(x, \lambda^*, v^*)$ . This together with (2.16) gives:

$$f(x^*) = g(\lambda^*, v^*). \quad (2.18)$$

Since  $p^*$  is the optimal value in the primal *minimization* problem,  $p^* \leq f(x^*)$ . Also  $d^* \geq g(\lambda^*, v^*)$  as it is the optimal value in the dual *maximization* problem. These together with (2.18) yield:

$$p^* \leq d^*. \quad (2.19)$$

Now we will below prove that

$$p^* \geq d^* \quad (2.20)$$

to complete the proof of sufficiency of the KKT conditions for ensuring strong duality. To prove (2.20), consider a primal optimal point, say  $\tilde{x}$ , that achieves  $p^*$ . Also consider a dual optimal point, say  $(\tilde{\lambda}, \tilde{v})$ , that achieves  $d^*$ . Then,

$$\begin{aligned} p^* &= f(\tilde{x}) \\ &\stackrel{(a)}{\geq} f(\tilde{x}) + \sum_{i=1}^m \tilde{\lambda}_i f_i(\tilde{x}) + \tilde{v}^T (A\tilde{x} - b) \\ &\geq \min_{x \in \mathcal{X}} f(x) + \sum_{i=1}^m \tilde{\lambda}_i f_i(x) + \tilde{v}^T (Ax - b) \\ &\stackrel{(b)}{=} g(\tilde{\lambda}, \tilde{v}) \\ &= d^* \end{aligned} \quad (2.21)$$

where (a) follows from the fact that  $\tilde{\lambda}_i \geq 0, f_i(\tilde{x}) \leq 0 \ \forall i$  and  $A\tilde{x} - b = 0$  (since  $(\tilde{x}, \tilde{\lambda}, \tilde{v})$  must be feasible points); and (b) is due to the definition (2.3) of the dual function.

Actually the relationship between  $p^*$  and  $d^*$  stated in (2.20) is called *weak duality*. It turns out weak duality holds for *any* optimization problem (including non-convex optimization), and this will be explored further in a later section.

**Look ahead** What can we do with the KKT conditions in the design of algorithms? In the next section, we will study details on this, and will demonstrate that the conditions indeed play a crucial role in gaining algorithmic insights.

## 2.2 Interior Point Method

---

**Recap** In the previous section, we embarked on Part II and started investigating strong duality and KKT conditions which we claimed several times that they provide a detailed guideline as to how to design algorithms. Strong duality relies on the concept of primal and dual problems:

$$\text{(Primal): } p^* := \min f(x) : f_i(x) \leq 0, i \in \{1, \dots, m\}, Ax - b = 0;$$

$$\text{(Dual): } d^* := \max_{\lambda, v} \min_{x \in \mathcal{X}} f(x) + \sum_{i=1}^m \lambda_i f_i(x) + v^T (Ax - b) : \lambda \geq 0$$

where  $\mathcal{X} := \mathbf{dom}f \cap \mathbf{dom}f_1 \cap \dots \cap \mathbf{dom}f_m$ . Using these, we figured out what strong duality means:

$$\text{(Strong duality): } p^* = d^*. \quad (2.22)$$

We then stated (yet without proof) that strong duality (2.22) holds for *convex* optimization, under a mild condition. Next we derived necessary and sufficient conditions (KKT conditions) in order for strong duality to hold under a feasible point of  $(x^*, \lambda^*, v^*)$ :

$$\nabla_x \mathcal{L}(x^*, \lambda^*, v^*) = 0; \quad (2.23)$$

$$\lambda_i^* f_i(x^*) = 0 \quad \forall i; \quad (2.24)$$

$$f_i(x^*) \leq 0 \quad \forall i; \quad (2.25)$$

$$Ax^* - b = 0; \quad (2.26)$$

$$\lambda^* \geq 0. \quad (2.27)$$

Lastly we claimed that the KKT conditions give algorithmic insights.

**Outline** In this section, we are going to study details as to why that is the case. We will support the claim in the context of the following three problem settings. The first is a somewhat special yet prominent problem setting where we already saw the KKT conditions (in Section 1.11): the equality-constrained least squares problem. In this case, we will demonstrate that the KKT conditions indeed lead to the closed-form solution that we saw. The second is a broader setting which however has still only the equality constraints. In this setting, we will show that the KKT conditions can be solved via gradient descent that we studied in Section 1.3. The last is a general setting which contains inequality constraints as well. We will introduce one very powerful algorithm, called the *interior point method* (Wright, 2005), which

can approximately implement the KKT conditions and therefore can approach the optimal value with a reasonably small performance gap to the optimality.

**Equality-constrained least squares** Recall the equality-constrained least squares problem:

$$\min \|Ax - b\|^2 : Cx - e = 0.$$

Let us verify that the KKT conditions (2.23)~(2.27) lead to the closed-form solution  $x^*$  that respects

$$\begin{bmatrix} 2A^T A & C^T \\ C & 0 \end{bmatrix} \begin{bmatrix} x^* \\ z \end{bmatrix} = \begin{bmatrix} 2A^T b \\ e \end{bmatrix} \quad (2.28)$$

for some  $z$ . Let  $\mathcal{L}(x, v)$  be the Lagrange function:

$$\mathcal{L}(x, v) = \|Ax - b\|^2 + v^T(Cx - e).$$

We first simplify the KKT conditions, tailoring them for this equality-constrained setting:

$$\nabla_x \mathcal{L}(x^*, v^*) = 0; \quad (2.29)$$

$$Cx^* - e = 0. \quad (2.30)$$

Taking a derivative of the Lagrange function w.r.t.  $x$  and setting it to 0, (2.29) reads:

$$\nabla_x \mathcal{L}(x^*, v^*) = 2A^T Ax^* - 2A^T b + C^T v^* = 0. \quad (2.31)$$

This together with the equality constraint yields:

$$2A^T Ax^* - 2A^T b + C^T v^* = 0; \quad (2.32)$$

$$Cx^* - e = 0. \quad (2.33)$$

Compared to the setting investigated in Section 1.11, the only distinction here is that we used a different notation  $v^*$  instead of  $z$ .

**Equality-constrained convex optimization** What about for general convex optimization problems? It turns out that solving the KKT conditions, one can develop some algorithms. In particular, for *equality*-constrained optimization problems, one can come up with a simple algorithm.

So let us first consider the equality-constrained setting:

$$\min f(x) : Ax - b = 0. \quad (2.34)$$

The Lagrange function is defined as:  $\mathcal{L}(x, v) = f(x) + v^T(Ax - b)$ . Under this setting, the KKT conditions (2.23)~(2.27) read:

$$\nabla_x \mathcal{L}(x^*, v^*) = 0; \quad (2.35)$$

$$Ax^* - b = 0. \quad (2.36)$$

Here the second condition can be rewritten as:

$$Ax^* - b = 0 \iff \nabla_v \mathcal{L}(x^*, v^*) = 0. \quad (2.37)$$

Since strong duality holds  $p^* = d^*$  in the convex optimization setting (due to the *strong duality theorem*), it suffices to develop an algorithm that achieves the optimal value in the *dual* problem. So we focus on:

$$\begin{aligned} d^* &:= \max_v g(v) \\ &= \max_v \min_{x \in \mathcal{X}} \mathcal{L}(x, v). \end{aligned}$$

Here one can make the following observations: (i)  $\mathcal{L}(x, v)$  is *convex* in  $x$ ; (ii)  $\min_{x \in \mathcal{X}} \mathcal{L}(x, v)$  is *concave* in  $v$ ; (iii)  $\min_{x \in \mathcal{X}} \mathcal{L}(x, v)$  is *unconstrained* (w.r.t.  $x$ ); and (iv)  $\max_v \min_{x \in \mathcal{X}} \mathcal{L}(x, v)$  is *unconstrained* (w.r.t.  $v$ ). Remember in Section 1.3 that the optimal condition for *unconstrained* convex minimization (or maximization) is that the gradient evaluated at the optimal point must be 0. More specifically, the optimality condition for the inner minimization problem is: given a  $v$ ,

$$\nabla_x \mathcal{L}(x^*(v), v) = 0 \quad (2.38)$$

where  $x^*(v) := \arg \min_{x \in \mathcal{X}} \mathcal{L}(x, v)$ . The optimality condition for the outer maximization problem is:

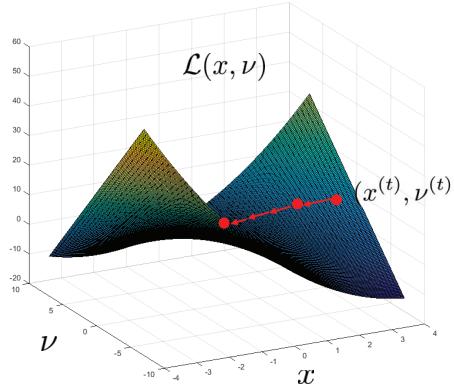
$$\nabla_v \mathcal{L}(x^*(v^*), v^*) = 0 \quad (2.39)$$

where  $\mathcal{L}(x^*(v), v) = \min_{x \in \mathcal{X}} \mathcal{L}(x, v)$ . Letting  $x^* := x^*(v^*)$ , the two conditions (2.38), (2.39) yield:

$$\nabla_x \mathcal{L}(x^*, v^*) = 0;$$

$$\nabla_v \mathcal{L}(x^*, v^*) = 0.$$

This then naturally motivates us to find a point  $(x^*, v^*)$  such that the two gradients are zeros.



**Figure 2.2.** Alternating gradient descent for equality-constrained optimization.

**Gradient descent** In Section 1.3, we studied one popular algorithm which allows us to find a stationary point where its gradient is 0. That was: *gradient descent*. So we can use the same algorithm. The only distinction here is that we have two points  $(x, v)$  to optimize over and so we have two corresponding gradients to compute. Below is how a modified algorithm works.

Let  $(x^{(t)}, v^{(t)})$  be the estimates at the  $t$ th iteration; see Fig. 2.2. First we compute a gradient w.r.t.  $x$  at the point:  $\nabla_x \mathcal{L}(x^{(t)}, v^{(t)})$ . Since  $\mathcal{L}(x, v)$  is convex in  $x$  (see Fig. 2.2 as well), we should move the point to the opposite direction (in reference to the gradient) so as to approach the optimal solution. So we update  $x^{(t)}$  as:

$$x^{(t+1)} \leftarrow x^{(t)} - \alpha^{(t)} \nabla_x \mathcal{L}(x^{(t)}, v^{(t)})$$

where  $\alpha^{(t)} > 0$  indicates the learning rate, which is usually set as a decaying function like  $\alpha^{(t)} = \frac{1}{2^t}$ .

Next, we compute a gradient w.r.t.  $v$ :  $\nabla_v \mathcal{L}(x^{(t)}, v^{(t)})$ . Notice that  $\min_x \mathcal{L}(x, v) (=: \mathcal{L}(x^*(v), v))$  is concave in  $v$ ; see the bottom part of the curve in Fig. 2.2 where the minimum of  $\mathcal{L}(x, v)$  is attained over  $x$ . So we should move the point to the *same* direction (in reference to the gradient) so as to approach the optimal solution. So we update  $v^{(t)}$  as:

$$v^{(t+1)} \leftarrow v^{(t)} + \beta^{(t)} \nabla_v \mathcal{L}(x^{(t)}, v^{(t)})$$

where  $\beta^{(t)} > 0$  indicates another learning rate, which is not necessarily the same as  $\alpha^{(t)}$ . Precisely speaking, this algorithm is called gradient *ascent* instead of gradient *descent*, although many people usually call it gradient descent nonetheless. So the above entire procedure is often called *alternating gradient descent*.

We repeat the above procedures until  $(x^{(t)}, v^{(t)})$  converges. It turns out: as  $t \rightarrow \infty$ , it actually converges:

$$(x^{(t)}, v^{(t)}) \longrightarrow (x^*, v^*), \quad (2.40)$$

as long as the learning rates  $(\alpha^{(t)}, \beta^{(t)})$  are properly chosen (like decaying functions). As in Section 1.3, we will not touch upon the convergence proof.

**Interior point method (Wright, 2005)** What about for general convex optimization settings which also involve *inequality* constraints? It turns out this is a bit challenging case. It is not that simple to solve the KKT conditions (2.23)~(2.27) directly. Instead there are algorithms which can *approximate* the KKT conditions. One such very popular algorithm is the *interior point method*.

The idea of the method is to take the following two steps:

1. *Approximate* the primal problem into an *equality*-constrained optimization.
2. Apply equality-constraint-tailored algorithms (like alternating gradient descent explained earlier) to the approximated optimization.

Since the method is based on an *approximation* trick, one may wonder how the performance of such an approach is far from optimality. It turns out that with a proper approximation trick (that we will investigate soon), we can achieve the optimal solution with a small gap to the optimality. To see this, let us first investigate what the approximation trick is.

**Approximation trick** Recall the standard form of general convex optimization including inequality constraints:

$$\begin{aligned} \min f(x) : & f_i(x) \leq 0, i \in \{1, \dots, m\}, \\ & Ax - b = 0 \end{aligned} \quad (2.41)$$

where  $f(x)$  and  $f_i(x)$ 's are convex functions,  $A \in \mathbf{R}^{p \times d}$  and  $p < d$ .

How to handle the inequality constraints? What we wish to do is to merge them with the objective function  $f(x)$  so that we have only equality constraints. To this end, we can set up a specific goal as:

$$\begin{aligned} f_i(x) \leq 0 & \longrightarrow \text{the merged objective function} = f(x); \\ f_i(x) > 0 & \longrightarrow \text{the reformulated optimization is infeasible.} \end{aligned}$$

In an effort to implement the goal, we introduce a function, called the *barrier function*, defined as:

$$I_-(z) = \begin{cases} 0, & z \leq 0; \\ \infty, & z > 0. \end{cases} \quad (2.42)$$

Now inputting  $f_i(x)$  to the barrier function as an argument, we get:

$$I_-(f_i(x)) = \begin{cases} 0, & f_i(x) \leq 0; \\ \infty, & f_i(x) > 0. \end{cases}$$

This then motivates the following natural idea: Adding  $I_-(f_i(x))$  to the objective function  $f(x)$ . This leads to the following reformulated problem:

$$\min f(x) + \sum_{i=1}^m I_-(f_i(x)) : Ax - b = 0. \quad (2.43)$$

Notice that when  $f_i(x) \leq 0 \forall i$ , the objective function is unchanged; on the other hand, whenever  $f_i(x) > 0$  for some  $i$ ,  $f(x)$  takes infinity, making the problem infeasible.

**A surrogate of the barrier function** In the reformulated problem (2.43), however, one critical issue arises. The issue comes from the fact that  $I_-(\cdot)$  is *not differentiable*. Notice that the first KKT condition (2.35) includes the *gradient* term. So it requires *differentiability* of the barrier functions as they appear in the Lagrange function. However, since  $I_-$  is not differentiable, we cannot implement the KKT conditions.

To resolve the critical issue, we consider a *surrogate* of the barrier function, which is differentiable and well approximates the barrier function. One very well-known surrogate is the *logarithmic* barrier:

$$\text{LB}(z) := -\mu \log(-z), \quad \mu > 0. \quad (2.44)$$

The function is indeed differentiable, and it well approximates the barrier function for a small  $\mu$ . See Fig. 2.3. Moreover, it is *convex* in  $z$ , and hence, we can maintain the objective function as a convex one.

**Approximated convex optimization** Replacing the barrier function with the logarithmic barrier in (2.43), we can approximate (2.43) as:

$$\min f(x) - \mu \sum_{i=1}^m \log(-f_i(x)) : Ax - b = 0. \quad (2.45)$$

There is one caveat here. That is, the search space of  $x$  should be:

$$\{x : f_1(x) < 0, \dots, f_m(x) < 0, \quad Ax - b = 0\}. \quad (2.46)$$

This is because the equality  $f_i(x) = 0$  for some  $i$  makes the logarithmic barrier function blow up. So we assume that the set in (2.46) is not empty. Actually this

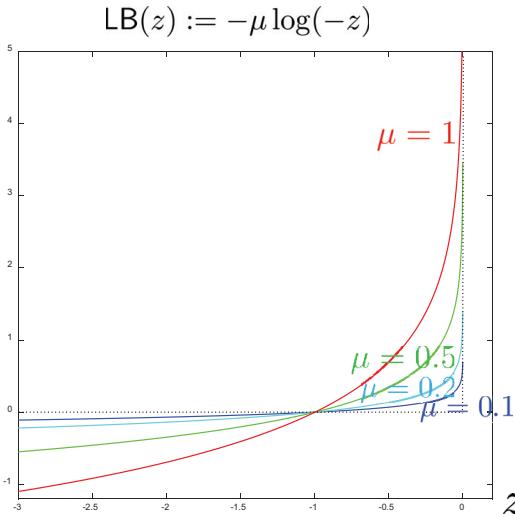


Figure 2.3. Logarithmic barrier functions for different control parameters  $\mu$ .

suggested the naming of “interior point method”, since the method searches over interior points.

Since the approximated optimization (2.45) contains only the *equality* constraint, we can apply exactly the same approach that we took for the earlier equality-constrained setting. In other words, we first compute the Lagrange function:

$$\mathcal{L}(x, v) = f(x) - \mu \sum_{i=1}^m \log(-f_i(x)) + v^T(Ax - b). \quad (2.47)$$

We then try to find a stationary point  $(x^*, v^*)$  that satisfies the KKT conditions:

$$\begin{aligned} \nabla_x \mathcal{L}(x^*, v^*) &= 0; \\ \nabla_v \mathcal{L}(x^*, v^*) &= 0. \end{aligned} \quad (2.48)$$

Again one can use alternating gradient descent to solve this.

**Performance gap to the optimality** Once we employ the interior point method that is based on the *approximated* optimization (2.45), one natural question that arises is: How far is the performance of the approximation approach from optimality?

To figure this out, we first consider the stationary point  $(x^*, v^*)$  that respects the KKT conditions (2.48). At this point, we obtain  $f(x^*)$  and obviously  $f(x^*) \geq p^*$ , since  $x^*$  (that satisfies (2.48) intended for the approximated optimization) is not necessarily the optimal solution of the *original* non-approximated optimization.

So the performance gap can be quantified as  $f(x^*) - p^*$ . The gap depends obviously on  $\mu$ , which is a control parameter adjusting the closeness to the barrier function. Let us figure out how it varies over  $\mu$ . Remember that the smaller  $\mu$ , the more precise the approximation is. Hence, one can expect that the smaller  $\mu$ , the smaller gap. It turns out it is the case:

$$f(x^*) - p^* \leq m\mu. \quad (2.49)$$

Observe that the gap is at most  $m\mu$ , so we approach the optimality for a small value of  $\mu$ .

*A special note on the choice of  $\mu$ :* One may want to set  $\mu$  arbitrarily small to ensure almost the optimal performance. In practice, however, this is not suggested. The reason is that the KKT conditions (2.48) are implemented via an algorithm (like alternating gradient descent) whose convergence speed is significantly affected by  $\mu$ . The smaller  $\mu$ , the slower speed. Hence, in practice, the choice should be carefully made taking the tradeoff into consideration.

**Proof of  $f(x^*) - p^* \leq m\mu$**  Here  $x^*$  together with  $v^*$  denotes the stationary point of the approximated optimization (not necessarily the optimal solution to the original optimization):

$$\begin{aligned} & \nabla_x \mathcal{L}_{\text{app}}(x^*, v^*) \\ &= \nabla_x \left( f(x) - \mu \sum_{i=1}^m \log(-f_i(x)) + v^T(Ax - b) \right) = 0 \end{aligned} \quad (2.50)$$

where  $\mathcal{L}_{\text{app}}(x, v)$  denotes the Lagrange function of the approximated optimization.

Starting with strong duality, we get:

$$\begin{aligned} p^* &= d^* \\ &= \max_{\lambda \geq 0, v} g(\lambda, v) \\ &\stackrel{(a)}{\geq} g(\lambda^*, v^*) \\ &\stackrel{(b)}{=} \min_{x \in \mathcal{X}} f(x) + \sum_{i=1}^m \lambda_i^* f_i(x) + v^{*T}(Ax - b) \end{aligned} \quad (2.51)$$

where (a) follows from the fact that  $v^*$  is a feasible point that satisfies (2.50) (not necessarily the one that maximizes  $g(\lambda, v)$ ), and  $\lambda^*$  is another particular feasible

point which will be detailed soon; and (b) is due to the definition of the dual function.

Remember that  $(x^*, v^*)$  is a point that satisfies (2.50), and hence:

$$\nabla_x \mathcal{L}_{\text{app}}(x^*, v^*) = \nabla f(x^*) + \sum_{i=1}^m \frac{-\mu}{f_i(x^*)} \nabla f_i(x^*) + A^T v^* = 0. \quad (2.52)$$

From this, we see that the particular point  $\lambda_i^*$  can be chosen as:

$$\lambda_i^* = \frac{-\mu}{f_i(x^*)}. \quad (2.53)$$

Under this particular choice (2.53), the condition (2.52) implies that  $x^*$  is a minimizer of the optimization in step (b) in (2.51) under  $(-\mu/f(x^*), v^*)$ :

$$\min_{x \in \mathcal{X}} \mathcal{L}(x, -\mu/f(x^*), v^*) = \mathcal{L}(x^*, -\mu/f(x^*), v^*) \quad (2.54)$$

where  $\mathcal{L}(\cdot, \cdot, \cdot)$  indicates the Lagrange function of the *original* optimization. Hence, applying this to (2.51), we get:

$$\begin{aligned} p^* &\geq \min_{x \in \mathcal{X}} f(x) + \sum_{i=1}^m \lambda_i^* f_i(x) + v^{*T} (Ax - b) \\ &= f(x^*) + \sum_{i=1}^m \frac{-\mu}{f_i(x^*)} f_i(x^*) + v^{*T} (Ax^* - b) \\ &\stackrel{(c)}{=} f(x^*) - m\mu \end{aligned} \quad (2.55)$$

where (c) comes from  $Ax^* - b = 0$  (since  $x^*$  must be a feasible point). This then yields the upper bound of the gap to complete the proof:

$$f(x^*) - p^* \leq m\mu.$$

**Look ahead** So far we have studied what strong duality is, and derived the KKT conditions, which are necessary and sufficient conditions for strong duality to hold in convex optimization. We also demonstrated that the KKT conditions provide detailed guidelines as to how to design algorithms. In the next section, we will prove the *strong duality theorem* which we only stated without proving.

## Problem Set 5

---

**Prob 5.1 (Lagrange function & dual function)** Consider a general optimization problem (not necessarily convex):

$$\begin{aligned} \min_{x \in \mathbb{R}^d} f(x) : f_i(x) \leq 0, \quad i \in \{1, \dots, m\}, \\ Ax - b = 0 \end{aligned} \tag{2.56}$$

where  $A \in \mathbb{R}^{p \times d}$ ,  $b \in \mathbb{R}^p$  and  $p < d$ . Assume that  $f : \mathbb{R}^d \rightarrow \mathbb{R}$  and  $f_i : \mathbb{R}^d \rightarrow \mathbb{R}$  are differentiable.

- (a) State the Lagrange function  $\mathcal{L}(x, \lambda, v)$ , the dual function  $g(\lambda, v)$ , and the dual problem.

- (b) Prove that  $g(\lambda, v)$  is concave in  $(\lambda, v)$ .

*Note: Please do not use any properties that we learned in Sections and/or Problem Sets (without derivation). Use only the definition of concavity.*

- (c) Let  $p^*$  (or  $d^*$ ) be the optimal value of the primal (or dual) problem. Show that

$$p^* \geq d^*. \tag{2.57}$$

**Prob 5.2 (KKT conditions for Quadratic Program)** Consider an equality-constrained least-squares problem:

$$\min \|Ax - b\|^2 : Cx - e = 0 \tag{2.58}$$

where  $A \in \mathbb{R}^{m \times d}$  and  $C \in \mathbb{R}^{p \times d}$ . Assume that  $m \geq d$  and  $p < d$ .

- (a) State the KKT matrix. Also prove that the following condition is necessary and sufficient for the KKT matrix to be invertible:

$$\text{rank}(C) = p, \quad \text{rank}\left(\begin{bmatrix} A \\ C \end{bmatrix}\right) = d. \tag{2.59}$$

- (b) State the KKT conditions. Suppose that the KKT matrix is invertible and there exist  $x^* \in \mathbb{R}^d$  and  $z \in \mathbb{R}^p$  such that the KKT conditions are satisfied. Show that:

$$\|Ax - b\|^2 \geq \|Ax^* - b\|^2 \quad \forall x : Cx - e = 0. \tag{2.60}$$

**Prob 5.3 (Performance of the interior point method)** Consider a convex optimization problem:

$$p^* := \max_{\lambda, v} g(\lambda, v) : \lambda \geq 0 \tag{2.61}$$

where  $\lambda \in \mathbf{R}^m$  and  $v \in \mathbf{R}^p$  are optimization variables, and  $g(\lambda, v)$  is a concave function. Using the logarithmic barrier function:

$$\text{LB}(z) := -\mu \log(-z) \text{ for } \mu > 0,$$

we formulate an approximated optimization problem:

$$\max_{\lambda, v} g(\lambda, v) + \mu \sum_{i=1}^m \log \lambda_i. \quad (2.62)$$

- (a) Suppose  $(\tilde{\lambda}, \tilde{v})$  is a feasible point that satisfies the KKT conditions w.r.t. the *approximated* problem (2.62). State the KKT conditions.
- (b) State the Lagrange function  $\mathcal{L}_{\text{dual}}(\lambda, v, \lambda_{\text{dual}})$  of the primal problem (2.61). Also state the dual function  $g_{\text{dual}}(\lambda_{\text{dual}})$  of  $\mathcal{L}_{\text{dual}}(\lambda, v, \lambda_{\text{dual}})$ . Here  $\lambda_{\text{dual}}$  denotes a Lagrange multiplier w.r.t. (2.61).
- (c) Suppose  $(\lambda^*, v^*)$  is a feasible point that satisfies the KKT conditions w.r.t. the *primal* problem (2.61). State the KKT conditions.
- (d) Show that:

$$p^* - g(\tilde{\lambda}, \tilde{v}) \leq m\mu. \quad (2.63)$$

#### Prob 5.4 (True or False?)

- (a) Consider primal and dual problems with the optimal values  $p^*$  and  $d^*$ , respectively. Then, the KKT conditions are necessary and sufficient conditions for  $p^* = d^*$  to hold.
- (b) Consider a convex optimization problem:

$$p^* := \max_{\lambda, v} g(\lambda, v) : \lambda \geq 0 \quad (2.64)$$

where  $\lambda \in \mathbf{R}^m$  and  $v \in \mathbf{R}^p$  are optimization variables, and  $g(\lambda, v)$  is a concave function. Consider another optimization problem:

$$\max_{\lambda, v} g(\lambda, v) + \mu \sum_{i=1}^m \log \lambda_i \quad (2.65)$$

where  $\mu > 0$ . Let  $\tilde{\lambda}$  be a feasible point, i.e.,  $\tilde{\lambda} \geq 0$ . Then,

$$p^* - g(\tilde{\lambda}, \tilde{v}) \leq m\mu. \quad (2.66)$$

as long as  $\tilde{v}$  is a feasible point that satisfies the KKT conditions w.r.t. the approximated optimization (2.65).

(c) Consider a convex optimization problem:

$$\max_{x \in \mathbf{R}^d} f(x) : Ax - b = 0 \quad (2.67)$$

where  $A \in \mathbf{R}^{p \times d}$  and  $b \in \mathbf{R}^p$ . Let  $\mathcal{L}(x, v)$  be the Lagrange function where  $v \in \mathbf{R}^p$  denotes the Lagrange multiplier. Suppose  $\tilde{x}$  is a stationary point, i.e.,  $\nabla_x \mathcal{L}(\tilde{x}, v) = 0$ . Then,  $\mathcal{L}(\tilde{x}, v)$  is always concave in  $v$ .

(d) Consider a convex optimization problem:

$$\begin{aligned} p := \min_{x \in \mathbf{R}^d} f(x) : f_i(x) \leq 0 & \quad i \in \{1, \dots, m\}, \\ h_i(x) = 0 & \quad i \in \{1, \dots, p\}. \end{aligned} \quad (2.68)$$

Let  $x^*$  be a feasible point that achieves the optimal value  $p^*$ . Suppose that there exists a feasible point  $(\tilde{\lambda}, \tilde{v})$  w.r.t. the dual problem such that the KKT conditions are satisfied:

$$\begin{aligned} \nabla_x \mathcal{L}(x^*, \tilde{\lambda}, \tilde{v}) &= 0; \\ \tilde{\lambda}_i f_i(x^*) &= 0 \quad \forall i; \\ f_i(x^*) &\leq 0 \quad \forall i; \\ h_i(x^*) &= 0 \quad \forall i; \\ \tilde{\lambda}_i &\geq 0 \quad \forall i \end{aligned} \quad (2.69)$$

where  $\mathcal{L}(x, \lambda, v)$  indicates the Lagrange function. Then, the feasible point  $(\tilde{\lambda}, \tilde{v})$  also achieves the optimal value  $d^*$  of the dual problem.

## 2.3 Proof of Strong Duality Theorem (1/2)

---

**Recap** During the past sections, we have investigated strong duality. In order to understand what it means, we studied the concept of primal and dual problems:

$$\text{(Primal): } p^* := \min f(x) : f_i(x) \leq 0, i \in \{1, \dots, m\}, Ax - b = 0;$$

$$\text{(Dual): } d^* := \max_{\lambda, v} g(\lambda, v) : \lambda \geq 0.$$

Using these, we stated:

$$\text{(Strong duality): } p^* = d^*. \quad (2.70)$$

We then argued that strong duality (2.70) holds for *convex* optimization under a mild condition. The mild condition was:

$$\exists x : f_1(x) < 0, \dots, f_m(x) < 0, Ax - b = 0. \quad (2.71)$$

Next we derived necessary and sufficient conditions (KKT conditions) in order for strong duality to hold. In the last section, we found that the KKT conditions indeed give algorithmic insights.

**Outline** In this section, we are going to move onto the strong duality theorem that we deferred the proof of. Actually the proof is not that easy. Not only the proof takes many non-trivial steps together with a bunch of ideas, but it also requires some important theorem that we did not dig into. So we will prove it step-by-step so that you can easily grasp how the proof goes on. Specifically we will investigate from simple to general cases: (i) *unconstrained* case; (ii) *equality*-constrained case; (iii) *inequality*-constrained case; and (iv) *general* case (including both equality-&-inequality constraints). In this section, we will cover the 1st and 2nd cases. In the next section, we will prove the 3rd and last cases to complete. Throughout the proof, we will assume that a minimum is attained, i.e.,  $p^*$  is finite. Otherwise,  $p^* = -\infty$  or  $+\infty$ . This is definitely not an interested scenario.

**Unconstrained optimization** This is a trivial case. In this case, the primal and dual problems read:

$$\begin{aligned} p^* &:= \min f(x); \\ d^* &:= \max g. \end{aligned}$$

Since  $d^*$  is simply  $g$ , we get:

$$\begin{aligned} d^* &= g : \stackrel{(a)}{=} \min_{x \in \mathcal{X}} \mathcal{L}(x) \\ &= \min_{x \in \mathcal{X}} f(x) = p^*, \end{aligned}$$

where (a) is due to the definition of the dual function.

**Equality-constrained optimization** Consider:

$$p^* := \min f(x) : Ax - b = 0;$$

$$d^* := \max_v g(v)$$

where  $A \in \mathbf{R}^{p \times d}$  and  $p < d$ . Without loss of generality, assume that  $\text{rank}(A) = \min(p, d) = p$ . Why? Otherwise, one can remove dependent rows in  $A$  to make it full-ranked. Remember that  $p \geq d$  is not of our interest, since in the case  $x^*$  is solely decided by the equality constraint, having nothing to do with the objective function. We will prove strong duality by showing the following two:

$$p^* \geq d^*; \tag{2.72}$$

$$p^* \leq d^*. \tag{2.73}$$

In fact, (2.72) is what we proved earlier in Section 2.1 for a more general context. In the sequel, we will repeat the proof for those who do not remember details.

**Review of the proof of (2.72):  $p^* \geq d^*$**  Suppose that a feasible point in the primal problem, say  $x^*$ , achieves  $p^*$ ; similarly, another feasible point in the dual problem, say  $v^*$ , achieves  $d^*$ . Using the fact that  $(x^*, v^*)$  are the minimizer and maximizer of the primal and dual problems respectively, we get:

$$\begin{aligned} p^* &= f(x^*) \\ &\stackrel{(a)}{=} f(x^*) + v^{*T} (Ax^* - b) \\ &\geq \min_{x \in \mathcal{X}} f(x) + v^{*T} (Ax - b) \\ &\stackrel{(b)}{=} g(v^*) \\ &= d^* \end{aligned}$$

where (a) follows from  $Ax^* - b = 0$  for a feasible point  $x^*$ ; and (b) comes from the definition of the dual function.

**Proof of (2.73):  $p^* \leq d^*$**  The proof of this is not that straightforward. It relies upon some trick which is based on a smartly-manipulated set (that you will see soon), as well as a well-known theorem, concerning the role of a hyperplane<sup>2</sup> when there are two disjoint convex sets. As of now, you may have no idea of what we are talking about. Don't worry. This will be clearer soon.

Let us start by defining the smartly-manipulated set:

$$\mathcal{S} := \{(v, t) \in \mathbf{R}^{p+1} : \exists x \text{ such that } f(x) \leq t, Ax - b = v\}. \quad (2.74)$$

*Three key properties of the set  $\mathcal{S}$ :* We point out three key properties of  $\mathcal{S}$ , which will play a crucial role in proving (2.73). The first is that the set  $\mathcal{S}$  contains the optimal point  $p^*$  of the primal problem when  $v = 0$ , i.e.,  $(0, p^*) \in \mathcal{S}$ . Also the point  $p^*$  is the *minimum* when  $v = 0$ , i.e.,  $(0, p^*)$  is on the *boundary* of the set. Why? Suppose that  $p^*$  is not the minimum, i.e.,  $(0, p^*)$  is strictly inside  $\mathcal{S}$ . Then, there exists some arbitrarily small  $\epsilon > 0$  such that another point  $(0, p^* - \epsilon)$  is in  $\mathcal{S}$ , which contradicts with the fact that  $p^*$  is the optimal value.

The second property is that:

$$(v, t) \in \mathcal{S} \implies (v, t') \in \mathcal{S}, \quad \forall t' \geq t. \quad (2.75)$$

This is obvious, since  $f(x) \leq t$  implies that  $f(x) \leq t'$  for  $t' \geq t$ . For instance, any point  $(0, t') \in \mathcal{S}$  whenever  $t' \geq p^*$ . See a blue line in Fig. 2.4 for illustration.

The last property that we would like to emphasize is that the set  $\mathcal{S}$  is convex. The proof of this is straightforward. Suppose  $(v_1, t_1), (v_2, t_2) \in \mathcal{S}$ . Then, this together with the definition (2.74) of the set  $\mathcal{S}$  yields: there exist some points, say  $x_1$  and  $x_2$ ,



**Figure 2.4.** The set  $\mathcal{S} := \{(v, t) \in \mathbf{R}^{p+1} : \exists x \text{ such that } f(x) \leq t, Ax - b = v\}$  contains the point  $(0, p^*)$  as well as any point  $(0, t')$  where  $t' \geq p^*$ . This figure is a simplified version when the dimension of  $v$  is 1.

2. For those who do not remember the definition of the hyperplane, here we echo. A hyperplane is a linear subspace whose dimension is one less than that of its ambient space.

such that

$$f(x_1) \leq t_1, \quad Ax_1 - b = v_1;$$

$$f(x_2) \leq t_2, \quad Ax_2 - b = v_2.$$

Applying an  $\lambda$ -weighted convex combination to the above, we get: for  $\lambda \in [0, 1]$ ,

$$\lambda v_1 + (1 - \lambda)v_2 = A(\lambda x_1 + (1 - \lambda)x_2) - b;$$

$$\lambda t_1 + (1 - \lambda)t_2 \geq \lambda f(x_1) + (1 - \lambda)f(x_2)$$

$$\stackrel{(a)}{\geq} f(\lambda x_1 + (1 - \lambda)x_2)$$

where (a) follows from the convexity of  $f(x)$ . This implies that there exists  $x = \lambda x_1 + (1 - \lambda)x_2$  such that:

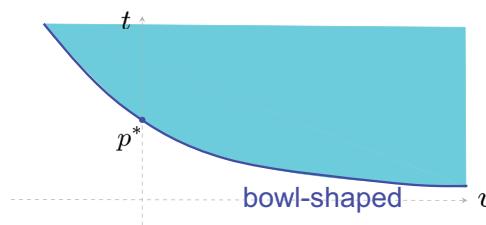
$$f(x) \leq \lambda t_1 + (1 - \lambda)t_2;$$

$$Ax - b = \lambda v_1 + (1 - \lambda)v_2,$$

which in turn yields that  $\lambda(v_1, t_1) + (1 - \lambda)(v_2, t_2) \in \mathcal{S}$ , thus proving the convexity of  $\mathcal{S}$ .

*How the set  $\mathcal{S}$  looks like and the use of the separating hyperplane theorem:* Using the second and third properties mentioned above, one can imagine how the set looks like. Since it is convex and also any point *above the boundary* is in the set, the boundary of the set would be *bowl-shaped*, as illustrated in Fig. 2.5.

We are now ready to introduce a well-known theorem regarding a hyperplane, so called the *separating hyperplane theorem*. The theorem says: If there are two disjoint convex sets, then there exists a hyperplane which separates the two convex sets. Intuitively this makes sense. Why? Think about two disjoint circles in a 2-dimensional space, which are obviously convex. Then, there must be a line somewhere in between the two circles, which separates the two. Actually the proof of this



**Figure 2.5.** The boundary (marked in the blue curve) of the convex set  $\mathcal{S} := \{(v, t) \in \mathbb{R}^{p+1} : \exists x \text{ such that } f(x) \leq t, Ax - b = v\}$  is of a *bowl shape*.

trivially-looking theorem is non-trivial. Here we will not cover the proof, but you will have a chance to prove it in Prob 6.1.

Now you may wonder why the theorem kicks in. The reason is that the theorem allows us to come up with a hyperplane which passes through the boundary point  $(0, p^*)$  while separating the set  $\mathcal{S}$  from another disjoint convex set, and this will help us to prove (2.73) in the end. Why does the theorem ensure the existence of such a hyperplane? To see this, consider another set, say  $\mathcal{S}'$ , defined as:

$$\mathcal{S}' := \{(0, s) \in \mathbf{R}^{p+1} : s < p^*\}. \quad (2.76)$$

Obviously this is convex (as it is just a line) and disjoint with  $\mathcal{S}$ . Now using the separating hyperplane theorem, we can say that there exists a hyperplane which separates  $\mathcal{S}$  from  $\mathcal{S}'$  while passing through the boundary point  $(0, p^*)$ . Let  $\begin{bmatrix} v \\ \mu \end{bmatrix} \in \mathbf{R}^{p+1}$  be the support vector of the hyperplane, being perpendicular to its tangent vector. Then, the hyperplane is represented as:

$$\begin{bmatrix} v \\ \mu \end{bmatrix}^T \left( \begin{bmatrix} v \\ t \end{bmatrix} - \begin{bmatrix} 0 \\ p^* \end{bmatrix} \right) = 0. \quad (2.77)$$

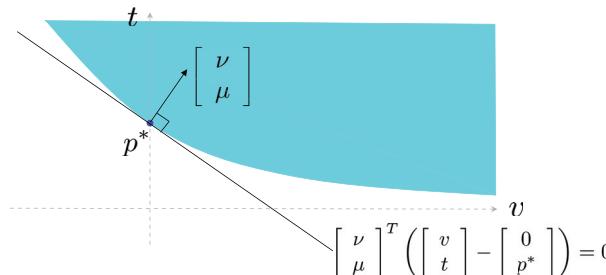
Why? Notice that the slope is the support vector and this plane passes through the point  $(0, p^*)$ . The separating hyperplane theorem says that whenever  $(v, t) \in \mathcal{S}$ , it always lies in the right-hand-side space in reference to the hyperplane, i.e.,

$$(v, t) \in \mathcal{S} \implies \begin{bmatrix} v \\ \mu \end{bmatrix}^T \left( \begin{bmatrix} v \\ t \end{bmatrix} - \begin{bmatrix} 0 \\ p^* \end{bmatrix} \right) \geq 0. \quad (2.78)$$

See Fig. 2.6 to help your understanding.

*Last step of the proof:* Using (2.78), we see:

$$\mu p^* \leq v^T v + \mu t \quad \forall (v, t) \in \mathcal{S}. \quad (2.79)$$



**Figure 2.6.** There exists a hyperplane that passes through  $(0, p^*)$  in the set  $\mathcal{S}$  while separating  $\mathcal{S}$  from another disjoint convex set.

One important thing to notice here is that  $\mu \geq 0$ . To prove this, suppose  $\mu < 0$ . Then, one can make  $t \rightarrow \infty$ . Observe that such  $(v, t) = (v, \infty)$  is still in  $\mathcal{S}$  due to the second property (2.75) of  $\mathcal{S}$ . But this yields a contradiction with (2.79) as:

$$\mu p^* \leq v^T v + \mu t = -\infty.$$

Notice that for finite  $p^*$  (which we assumed),  $\mu p^*$  is also finite, which can never go below  $-\infty$ . Another thing to notice is that:

$$\mu \neq 0. \quad (2.80)$$

We will prove this soon. This together with  $\mu \geq 0$  gives:  $\mu > 0$ . This then enables us to divide both sides in (2.79) by  $\mu > 0$ , thus obtaining:

$$p^* \leq \left(\frac{v}{\mu}\right)^T v + t \quad \forall (v, t) \in \mathcal{S}. \quad (2.81)$$

Recall the definition of the interested set:  $\mathcal{S} := \{(v, t) \in \mathbf{R}^{p+1} : \exists x \text{ such that } f(x) \leq t, Ax - b = v\}$ . The fact that  $(v, t) \in \mathcal{S}$  means that there exists  $x$  such that  $f(x) \leq t$  and  $Ax - b = v$ . Applying such  $x$ 's to the above (2.81), we obtain:

$$\begin{aligned} p^* &\leq \min_{x: f(x) \leq t} t + \left(\frac{v}{\mu}\right)^T (Ax - b) \\ &\stackrel{(a)}{=} \min_{x: f(x) \leq t} f(x) + \left(\frac{v}{\mu}\right)^T (Ax - b) \end{aligned} \quad (2.82)$$

where (a) follows from the fact that minimizing  $t$  is equivalent to minimizing  $f(x)$ .

Notice in the above that  $f(x) \leq t$  becomes unconstrained by taking  $t \rightarrow \infty$ . Hence, taking  $t \rightarrow \infty$ , we get:

$$\begin{aligned} p^* &\leq \inf_x f(x) + \left(\frac{v}{\mu}\right)^T (Ax - b) \\ &\stackrel{(a)}{=} g\left(\frac{v}{\mu}\right) \\ &\stackrel{(b)}{\leq} d^*, \end{aligned}$$

where (a) is due to the definition of the dual function; and (b) comes from the definition of  $d^*$ . This completes the proof of (2.73).

**Proof of (2.80):  $\mu \neq 0$**  The proof idea is by contradiction. Suppose  $\mu = 0$ . Then, (2.79) implies:

$$\nu^T v \geq 0 \quad \forall(v, t) \in \mathcal{S}. \quad (2.83)$$

Two things to note. The first is that  $v \neq 0$ . This is obvious. Otherwise  $(v, \mu) = 0$ , and this implies that there does not exist a hyperplane that passes through  $(0, p^*)$  in  $\mathcal{S}$  while separating  $\mathcal{S}$  from another disjoint convex set. This then violates the separating hyperplane theorem. The second is that  $v$  can be chosen arbitrarily as long as there exists  $x$  such that  $Ax - b = v$  and  $f(x) \leq t$ . Due to the second property of  $\mathcal{S}$  mentioned earlier,  $(v, \infty) \in \mathcal{S}$  whenever  $(v, t) \in \mathcal{S}$ . Hence, the only thing that we need to worry about is whether there exists  $x$  satisfying  $Ax - b = v$ . Now remember what we assumed in the beginning:  $A$  has full rank ( $\text{rank}(A) = p$ ). This suggests that we can choose  $x$  such that  $Ax - b$  points to an arbitrary direction. Hence, there exists some point, say  $x'$ , such that the direction of  $Ax' - b = v$  is somewhat *opposite* to  $v$  so that:

$$\nu^T v < 0.$$

This contradicts with (2.83), thus completing the proof of (2.80).

**Look ahead** It turns out that using the techniques employed so far, one can prove  $p^* \leq d^*$  for the inequality-constrained case and general case. We will cover the two remaining cases to complete the proof in the next section.

## 2.4 Proof of Strong Duality Theorem (2/2)

**Recap** In the last section, we proved the strong duality theorem for two simple cases: (i) unconstrained optimization; and (ii) equality-constrained optimization. The key trick was to introduce the following set:

$$\mathcal{S} := \{(v, t) \in \mathbf{R}^{p+1} : \exists x \text{ such that } Ax - b = v, f(x) \leq t\} \quad (2.84)$$

where  $v \in \mathbf{R}^p$  and  $t \in \mathbf{R}$ . By using the key properties of  $\mathcal{S}$  (e.g., the convexity of  $\mathcal{S}$  and  $(0, p^*) \in \mathcal{S}$ ) together with the separating hyperplane theorem, we proved  $p^* \leq d^*$  where  $p^*$  and  $d^*$  are the optimal solutions of the primal and dual problems respectively. Combining this with the already-proven fact  $p^* \geq d^*$ , we completed the proof.

**Outline** In this section, we are going to cover the two remaining cases (inequality-constrained and general optimization) to complete the proof. Again we assume that  $p^*$  is finite. Actually in a more general setting that includes *inequality* constraints, the statement of the strong duality theorem should be made carefully. The precise statement reads:  $p^* = d^*$  holds under a *mild condition*:

$$\exists x \text{ such that strict inequalities hold i.e., } f_1(x) < 0, \dots, f_m(x) < 0 \quad (2.85)$$

where  $f_i(x)$ 's indicate the LHS functions in the standard form of inequality constraints. This condition is called *Slater's condition*, since it was found by a mathematician Morton L. Slater ([Slater, 2014](#)). It serves as a sufficient condition for strong duality to hold. It is considered to be *mild*, as the condition often holds in practice.

**Inequality-constrained optimization** For illustrative purpose, we first consider a simple case having only one inequality constraint:

$$p^* := \min f(x) : f_1(x) \leq 0;$$

$$d^* := \max_{\lambda \geq 0} g(\lambda).$$

It turns out one can readily extend this to a general case. So let us focus on this simple setting for now. Like the equality-constrained case, one can easily show that  $p^* \geq d^*$ . The proof is almost same. Please check this by yourself. So it suffices to prove the other way around:

$$p^* \leq d^*. \quad (2.86)$$

**Proof of (2.86):  $p^* \leq d^*$**  Like the equality-constrained case, we introduce a smartly-manipulated set that plays an important role in the proof. The set is defined similarly to (2.84):

$$\mathcal{S} = \{(u, t) \in \mathbf{R}^2 : \exists x \text{ such that } f_1(x) \leq u, f(x) \leq t\} \quad (2.87)$$

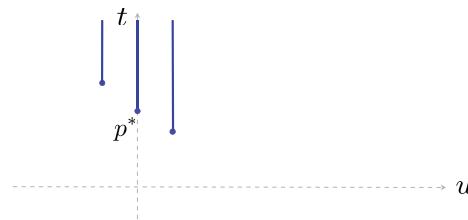
where  $u \in \mathbf{R}$ . One distinction is that we now have  $u$  (instead of  $v$ ) which acts as an upper bound on  $f_1(x)$ . Similar to (2.84), the newly-defined set  $\mathcal{S}$  (2.87) also exhibits three properties:

- (i) It contains the boundary point  $(0, p^*)$ ;
- (ii)  $(u, t) \in \mathcal{S} \Rightarrow (u', t') \in \mathcal{S}$  whenever  $u' \geq u$  and  $t' \geq t$ ;
- (iii)  $\mathcal{S}$  is convex.

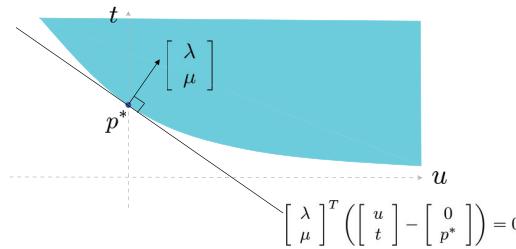
The first and second are obvious. The third property makes an intuitive sense if we think about a picture, as illustrated in Fig. 2.7. Consider a case in which  $u > 0$ . Actually this is a *more relaxed scenario* relative to  $u = 0$ . This is because the constraint  $f_1(x) \leq u$  has a larger search space compared to  $f_1(x) \leq 0$ . Hence, the minimum  $t \in \mathcal{S}$  would be *smaller than or equal to  $p^*$* . On the other hand, when  $u < 0$ , the constraint  $f_1(x) \leq u$  yields a shrunked search space; and therefore the minimum  $t \in \mathcal{S}$  would be *larger than or equal to  $p^*$* . With this argument, one can image a shape of the set  $\mathcal{S}$  like the one in Fig. 2.8 (a cyan-colored region). So one can conjecture that the set  $\mathcal{S}$  is convex. It turns out it is indeed the case. The proof is almost the same as in the equality-constrained case.

*Proof of the convexity of  $\mathcal{S}$ :* Suppose  $(u_1, t_1), (u_2, t_2) \in \mathcal{S}$ . Then, this together with the definition (2.87) of the set  $\mathcal{S}$  yields: there exist some points, say  $x_1$  and  $x_2$ , such that

$$\begin{aligned} f_1(x_1) &\leq u_1, & f(x_1) &\leq t_1; \\ f_1(x_2) &\leq u_2, & f(x_2) &\leq t_2. \end{aligned}$$



**Figure 2.7.** The set  $\mathcal{S} = \{(u, t) \in \mathbf{R}^2 : \exists x \text{ such that } f_1(x) \leq u, f(x) \leq t\}$  contains the point  $(0, p^*)$  as well as any point  $(0, t')$  where  $t' \geq p^*$ . Also for  $u > 0$ , the minimum  $t$  in the set  $\mathcal{S}$  is smaller than or equal to  $p^*$ ; when  $u < 0$ , the minimum  $t$  is larger than or equal to  $p^*$ .



**Figure 2.8.** There exists a hyperplane (a line in this example) that passes through  $(0, p^*)$  in the set  $\mathcal{S} = \{(u, t) \in \mathbb{R}^2 : \exists x \text{ such that } f_1(x) \leq u, f(x) \leq t\}$ .

Applying an  $\lambda$ -weighted convex combination to the above, we get: for  $\lambda \in [0, 1]$ ,

$$\begin{aligned} \lambda u_1 + (1 - \lambda)u_2 &\geq \lambda f_1(x_1) + (1 - \lambda)f_1(x_2) \\ &\stackrel{(a)}{\geq} f_1(\lambda x_1 + (1 - \lambda)x_2) \end{aligned}$$

where (a) follows from the convexity of  $f_1(x)$ . Similarly, using the convexity of  $f(x)$ , we get:

$$\lambda t_1 + (1 - \lambda)t_2 \geq f(\lambda x_1 + (1 - \lambda)x_2).$$

Hence,  $\lambda(u_1, t_1) + (1 - \lambda)(u_2, t_2) \in \mathcal{S}$ . This completes the proof. ■

Now using the *separating hyperplane theorem*, we argue that there exists  $\begin{bmatrix} \lambda \\ \mu \end{bmatrix} \in \mathbb{R}^2 \neq 0$  s.t.

$$(u, t) \in \mathcal{S} \implies \begin{bmatrix} \lambda \\ \mu \end{bmatrix}^T \left( \begin{bmatrix} u \\ t \end{bmatrix} - \begin{bmatrix} 0 \\ p^* \end{bmatrix} \right) \geq 0. \quad (2.88)$$

Looking at the hyperplane in Fig. 2.8, we see that the support vector w.r.t. the hyperplane has a *positive* direction. Hence, one may conjecture that

$$\lambda \geq 0, \quad \mu \geq 0. \quad (2.89)$$

It turns out this is indeed the case.

*Proof of (2.89):* Notice that (2.88) gives:

$$\mu p^* \leq \lambda u + \mu t \quad \forall (u, t) \in \mathcal{S}. \quad (2.90)$$

Let us first prove  $\mu \geq 0$ . Suppose  $\mu < 0$ . We then make  $t \rightarrow \infty$ . Here such  $(u, \infty)$  is still in  $\mathcal{S}$  due to the second property of  $\mathcal{S}$ . But this yields a contradiction:

$$\mu p^* \leq \lambda u + \mu t = -\infty. \quad (2.91)$$

Similarly, one can prove  $\lambda \geq 0$ . ■

Let us go back to the main stream of the proof. Like the equality-constrained case, it turns out:

$$\mu \neq 0 \quad (2.92)$$

where the proof will be given soon. This together with (2.89) gives  $\mu > 0$ . Dividing both sides in (2.90) by  $\mu > 0$ , we obtain:

$$p^* \leq t + \frac{\lambda}{\mu} u \quad \forall (u, t) \in \mathcal{S}. \quad (2.93)$$

Recall the definition of the interested set:

$$\mathcal{S} := \{(u, t) \in \mathbf{R}^2 : \exists x \text{ such that } f_1(x) \leq u, f(x) \leq t\}. \quad (2.94)$$

The fact  $(u, t) \in \mathcal{S}$  means that there exists  $x$  such that  $f_1(x) \leq u$  and  $f(x) \leq t$ . By applying such  $x$ 's to (2.93), we obtain:

$$\begin{aligned} p^* &\leq \min_{x: f_1(x) \leq u, f(x) \leq t} t + \frac{\lambda}{\mu} u \\ &\stackrel{(a)}{=} \min_{x: f_1(x) \leq u, f(x) \leq t} f(x) + \frac{\lambda}{\mu} f_1(x) \end{aligned} \quad (2.95)$$

where (a) follows from the fact that minimizing  $t$  and  $u$  are equivalent to minimizing  $f(x)$  and  $f_1(x)$ , respectively. Notice in the above that  $f_1(x) \leq u$  and  $f(x) \leq t$  become unconstrained as  $(u, t) \rightarrow (\infty, \infty)$ . Hence, we get:

$$\begin{aligned} p^* &\leq \min_x f(x) + \frac{\lambda}{\mu} f_1(x) \\ &= g\left(\frac{\lambda}{\mu}\right) \\ &\leq d^*. \end{aligned}$$

This completes the proof.

**Proof of (2.92):  $\mu \neq 0$**  The proof is by contradiction. Suppose  $\mu = 0$ . Then, (2.90) implies that:

$$\lambda u \geq 0 \quad \forall (u, t) \in \mathcal{S}. \quad (2.96)$$

Here one thing to notice is that  $\lambda$  (which was claimed to be non-negative in (2.89)) is strictly positive:

$$\lambda > 0. \quad (2.97)$$

Why? Otherwise,  $(\lambda, \mu) = 0$  and this contradicts with the *separating hyperplane theorem*. Next, consider  $f_1(x)$ . Actually we have never used the mild condition thus far:  $\exists x$  such that  $f_1(x) < 0$ . This is where the mild condition kicks in. Due to the condition, there exists a point, say  $\bar{x}$ , such that:

$$f_1(\bar{x}) < 0. \quad (2.98)$$

Given  $\bar{x}$ , pick up  $(u, t)$  such that  $f_1(\bar{x}) \leq u < 0$  and  $f(\bar{x}) \leq t$ . Applying this to the definition (2.87) of the set  $\mathcal{S}$ , we see that  $(u, t) \in \mathcal{S}$ . But for such  $u$ ,

$$\lambda u < 0 \quad (2.99)$$

due to (2.97) and  $u < 0$ . This contradicts with (2.96), thus completing the proof.

**Multiple-inequality-constrained optimization** Next consider a multiple-inequality-constrained convex optimization:

$$p^* := \min f(x) : f_i(x) \leq 0 \quad i \in \{1, \dots, m\},$$

$$d^* := \max_{\lambda \geq 0} g(\lambda).$$

The proof of  $p^* \leq d^*$  is almost the same as in the single-inequality case. The only distinction here lies in the definition of the smartly-manipulated set:

$$\mathcal{S} = \left\{ (u, t) \in \mathbf{R}^{m+1} : \exists x \text{ such that } f_i(x) \leq u_i \quad i \in \{1, \dots, m\} \right. \\ \left. \text{and } f(x) \leq t \right\} \quad (2.100)$$

where  $u \in \mathbf{R}^m$ . Like the single-inequality case, one can readily show the following to prove  $p^* \leq d^*$ :

1.  $\mathcal{S}$  is convex;
2. There exists a hyperplane  $\begin{bmatrix} \lambda \\ \mu \end{bmatrix} \in \mathbf{R}^{m+1} \neq 0$  such that

$$(u, t) \in \mathcal{S} \implies \begin{bmatrix} \lambda \\ \mu \end{bmatrix}^T \left( \begin{bmatrix} u \\ t \end{bmatrix} - \begin{bmatrix} 0 \\ p^* \end{bmatrix} \right) \geq 0; \quad (2.101)$$

3.  $\lambda \geq 0, \mu > 0$ ;
4.  $p^* \leq t + \frac{\lambda^T u}{\mu} \quad \forall (u, t) \in \mathcal{S}$ ;
5.  $p^* \leq \min_x f(x) + \sum_{i=1}^m \frac{\lambda_i}{\mu} f_i(x) = g\left(\frac{\lambda}{\mu}\right) \leq d^*$ .

**General optimization** Lastly consider a general convex optimization problem which has now both equality-&inequality constraints:

$$p^* := \min f(x) : f_i(x) \leq 0 \quad i = 1, \dots, m,$$

$$Ax - b = 0;$$

$$d^* := \max_{\lambda \geq 0, v} g(\lambda, v)$$

where  $A \in \mathbf{R}^{p \times d}$ ,  $p < d$  and  $\text{rank}(A) = p$ . Again the key distinction is in the definition of the smartly-manipulated set:

$$\begin{aligned} \mathcal{S} = \{ (u, v, t) \in \mathbf{R}^{m+p+1} : \exists x \text{ s.t. } & f_i(x) \leq u_i, i \in \{1, \dots, m\}, \\ & Ax - b = v, f(x) \leq t \} \end{aligned} \quad (2.102)$$

where  $u \in \mathbf{R}^m$  and  $v \in \mathbf{R}^p$ . The procedure would look very much complicated. But the key procedure follows simply a combination of the ideas that we studied while tackling the above simpler cases. Concretely, one can show the following to prove  $p^* \leq d^*$ :

1.  $\mathcal{S}$  is convex;

2. There exists a hyperplane  $\begin{bmatrix} \lambda \\ v \\ \mu \end{bmatrix} \in \mathbf{R}^{m+p+1} \neq 0$  such that

$$(u, v, t) \in \mathcal{S} \implies \begin{bmatrix} \lambda \\ v \\ \mu \end{bmatrix}^T \left( \begin{bmatrix} u \\ v \\ t \end{bmatrix} - \begin{bmatrix} 0 \\ 0 \\ p^* \end{bmatrix} \right) \geq 0; \quad (2.103)$$

3.  $\lambda \geq 0$ ,  $\mu > 0$ ;

4.  $p^* \leq t + \frac{\lambda^T u}{\mu} + \frac{v^T v}{\mu} \quad \forall (u, v, t) \in \mathcal{S}$ ;

5.  $p^* \leq \min_x f(x) + \sum_{i=1}^m \frac{\lambda_i}{\mu} f_i(x) + \frac{v^T (Ax - b)}{\mu} = g\left(\frac{\lambda}{\mu}, \frac{v}{\mu}\right) \leq d^*$ .

You will also have a chance to dig into details of the proof in Prob 6.3.

**Look ahead** In Part I, we studied a variety of convex optimization problems: all the problems in Fig. 2.9. However, we did not learn how to design generic algorithms that can be applied to arbitrary scenarios. To this end, during the past sections, we learned about strong duality and KKT conditions. We then studied a generic algorithm building upon them: the *interior point method*. And in this and last sections, we proved the *strong duality theorem* that we deferred proving earlier. So we are essentially done with the convex optimization story.

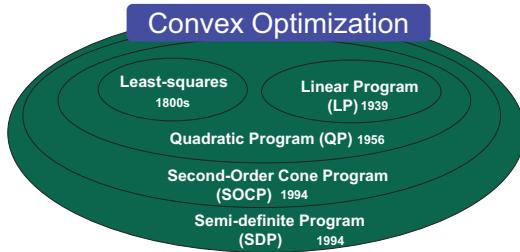


Figure 2.9. Convex optimization instances that we explored in Part I.

What is next? We may want to ask some interesting questions that can spark future studies. One very natural follow-up question is: What about for *non-convex* optimization? Can the techniques that we have learned w.r.t. convex optimization problems help address the general case? Fortunately, it is indeed the case. It turns out those techniques can help *approximating* optimal solutions in general problems. In order to understand what it means, we need to study another important theory, so called *weak duality*, which forms the content of the next section.

## Problem Set 6

---

**Prob 6.1 (Separating hyperplane theorem)** Suppose that  $\mathcal{S}$  and  $\mathcal{S}'$  are two disjoint non-empty convex sets which do not intersect, i.e.,  $\mathcal{S} \cap \mathcal{S}' = \emptyset$ . The *separating hyperplane theorem* that we mentioned in Section 2.3 says: There exist  $a \neq 0 \in \mathbf{R}^d$  and  $b \in \mathbf{R}$  such that

$$\begin{aligned} x \in \mathcal{S} &\implies a^T x - b \geq 0; \\ x \in \mathcal{S}' &\implies a^T x - b \leq 0. \end{aligned} \tag{2.104}$$

This problem explores the proof of this theorem. Define the Euclidean distance between the two sets  $\mathcal{S}$  and  $\mathcal{S}'$  as:

$$\|\mathcal{S} - \mathcal{S}'\| := \min\{\|s - s'\| : s \in \mathcal{S}, s' \in \mathcal{S}'\}. \tag{2.105}$$

- (a) State the definitions of the *closed* set, the *open* set, and the *bounded* set.
- (b) Suppose:

$$\mathcal{S} \text{ and } \mathcal{S}' \text{ are } \textit{closed} \text{ and one set, say } \mathcal{S}, \text{ is } \textit{bounded}. \tag{2.106}$$

Show that  $\|\mathcal{S} - \mathcal{S}'\|$  is positive and there exist points  $s \in \mathcal{S}$  and  $s' \in \mathcal{S}'$  that minimize  $\|\mathcal{S} - \mathcal{S}'\|$ , i.e.,  $\|s - s'\| = \|\mathcal{S} - \mathcal{S}'\|$ .

- (c) Suppose (2.106) holds. Prove the separating hyperplane theorem.
- (d) Consider the set  $\mathcal{D} := \{s - s' : s \in \mathcal{S}, s' \in \mathcal{S}'\}$ . Show that  $\mathcal{D}$  is convex and does not contain the origin.
- (e) Suppose (2.106) does not necessarily hold. Prove the separating hyperplane theorem.

**Prob 6.2 (Proof of the strong duality theorem: Exercise 1)** Consider a convex optimization problem:

$$p^* := \min_{x \in \mathbf{R}^d} f(x) : f_1(x) \leq 0; \quad a^T x - b = 0$$

where  $a \in \mathbf{R}^d$  and  $b \in \mathbf{R}$ . The dual problem is then:

$$d^* := \max_{\lambda \geq 0, v} g(\lambda, v)$$

where  $g(\lambda, v)$  indicates the dual function. Define a set:

$$\mathcal{S} = \{(u, v, t) \in \mathbf{R}^3 : \exists x \text{ s.t. } f_1(x) \leq u, a^T x - b = v, f(x) \leq t\}.$$

Assume that  $p^*$  is finite and there exists  $\bar{x} \in \mathbf{R}^d$  such that

$$f_1(\bar{x}) < 0, \quad a^T \bar{x} - b = 0. \tag{2.107}$$

- (a) Prove that  $p^* \geq d^*$ .
- (b) Prove that  $\mathcal{S}$  is convex.
- (c) State the separating hyperplane theorem. Use the theorem to prove that there exists  $(\lambda, v, \mu) \neq 0$  such that

$$\mu p^* \leq \lambda u + v v + \mu t \quad \forall (u, v, t) \in \mathcal{S}. \quad (2.108)$$

- (d) For  $(\lambda, \mu)$  in part (c), show that

$$\lambda \geq 0, \mu > 0. \quad (2.109)$$

- (e) Prove that  $p^* \leq d^*$ .

**Prob 6.3 (Proof of the strong duality theorem: Exercise 2)** Consider a convex optimization problem:

$$p^* := \min_{x \in \mathbb{R}^d} f(x) : f_i(x) \leq 0 \quad i \in \{1, \dots, m\}; \quad Ax - b = 0$$

where  $A \in \mathbb{R}^{p \times d}$ ,  $b \in \mathbb{R}^p$ ,  $p < d$  and  $\text{rank}(A) = p$ . The dual problem is then:

$$d^* := \max_{\lambda \geq 0, v} g(\lambda, v)$$

where  $g(\lambda, v)$  indicates the dual function. Define a set:

$$\mathcal{S} = \{(u, v, t) \in \mathbb{R}^{m+p+1} : \exists x \text{ s.t. } f_i(x) \leq u_i, \forall i, Ax - b = v, f(x) \leq t\}$$

where  $u \in \mathbb{R}^m$  and  $v \in \mathbb{R}^p$ . Assume that  $p^*$  is finite and Slater's condition is satisfied, i.e., there exists  $\bar{x} \in \mathbb{R}^d$  such that

$$f_1(\bar{x}) < 0, \dots, f_m(\bar{x}) < 0, A\bar{x} - b = 0. \quad (2.110)$$

- (a) Prove that  $p^* \geq d^*$ .
- (b) Prove that  $\mathcal{S}$  is convex.
- (c) Using the separating hyperplane theorem, show that there exists  $(\lambda, v, \mu) \neq 0$  such that

$$\mu p^* \leq \lambda^T u + v^T v + \mu t \quad \forall (u, v, t) \in \mathcal{S}. \quad (2.111)$$

- (d) For  $\mu$  in part (c), show that  $\mu > 0$ .
- (e) Prove that  $p^* \leq d^*$ .

**Prob 6.4 (Trace tricks)** Consider a square matrix  $A \in \mathbf{R}^{n \times n}$ . Denote the trace of a square matrix by:

$$\text{trace}(A) := \sum_{i=1}^n A_{ii} \quad (2.112)$$

where  $A_{ii}$  indicates the  $i$ th diagonal entry of  $A$ .

- (a) Suppose  $A \succeq 0$ . Show that  $A$  can be represented as

$$A = XX^T \quad (2.113)$$

for some  $X \in \mathbf{R}^{n \times n}$ .

- (b) Show that  $AA^T \succeq 0$ .

- (c) Consider  $A, B \in \mathbf{R}^{n \times n}$ . Suppose  $A \succeq 0$  and  $B \succeq 0$ . Show that

$$\text{trace}(AB) \geq 0. \quad (2.114)$$

- (d) Let  $X, Z \in \mathbf{R}^{n \times n}$  be symmetric matrices. Show that

$$\sum_{i=1}^n \sum_{j=1}^n Z_{ij} X_{ij} = \text{trace}(ZX). \quad (2.115)$$

**Prob 6.5 (KKT conditions for SDP)** Consider an optimization problem:

$$\begin{aligned} p^* := \min_X \text{trace}(WX) : \\ X \succeq 0, \quad X_{ii} = 1, \quad i \in \{1, \dots, d\} \end{aligned} \quad (2.116)$$

where  $W \in \mathbf{R}^{d \times d}$ ,  $X \in \mathbf{R}^{d \times d}$  and  $X_{ii}$  indicates the  $i$ th diagonal entry of  $X$ . Let  $Z \in \mathbf{R}^{d \times d}$  be symmetric and  $v \in \mathbf{R}^d$ . Define the Lagrange function as:

$$\mathcal{L}(X, Z, v) := \text{trace}(WX) + \sum_{i=1}^d \sum_{j=1}^d Z_{ij}(-X_{ij}) + \sum_{i=1}^d v_i(1 - X_{ii}). \quad (2.117)$$

- (a) Is the problem (2.116) convex? If so, specify the class of the problem.  
 (b) Show that

$$\mathcal{L}(X, Z, v) = \text{trace}((W - Z - D_v)X) + v^T 1 \quad (2.118)$$

where  $D_v := \text{diag}(v_1, \dots, v_d)$ .

- (c) Derive the dual function. Also state the dual problem with a constraint  $Z \succeq 0$ .

- (d) Let  $d^*$  be the optimal value of the dual problem that you formulated in part (c). Derive necessary conditions for strong duality to hold (i.e.,  $p^* = d^*$ ) in this problem context.
- (e) Are the necessary conditions also sufficient for  $p^* = d^*$ ? If so, prove the sufficiency. Otherwise, are there any further necessary conditions that are also sufficient for  $p^* = d^*$ ?

**Prob 6.6 (Strong duality theorem for SDP)** Consider the same optimization problem (2.116) as in Prob 6.5:

$$\begin{aligned} p^* := \min_X \text{trace}(WX) : \\ X \succeq 0, X_{ii} = 1, i \in \{1, \dots, d\} \end{aligned} \quad (2.119)$$

where  $W \in \mathbf{R}^{d \times d}$ ,  $X \in \mathbf{R}^{d \times d}$  and  $X_{ii}$  indicates the  $i$ th diagonal entry of  $X$ .

The dual problem is then:

$$d^* := \max_{Z \succeq 0} g(Z, v)$$

where  $g(Z, v)$  indicates the dual function,  $Z \in \mathbf{R}^{d \times d}$  is a Lagrange multiplier for the inequality constraint, and  $v \in \mathbf{R}^d$  is the equality constraint counterpart.

Define a set:

$$\mathcal{S} = \{(u, v, t) \in \mathbf{R}^{m+p+1} : \exists x \text{ s.t. } f_i(x) \leq u_i, \forall i, Ax - b = v, f(x) \leq t\}.$$

Assume that  $p^*$  is finite and there exists  $\bar{x} \in \mathbf{R}^d$  such that

$$f_1(\bar{x}) < 0, \dots, f_m(\bar{x}) < 0, A\bar{x} - b = 0. \quad (2.120)$$

- (a) Prove that  $p^* \geq d^*$ .
- (b) Prove that  $\mathcal{S}$  is convex.
- (c) Using the separating hyperplane theorem, show that there exists  $(\lambda, v, \mu) \neq 0$  such that

$$\mu p^* \leq \lambda^T u + v^T v + \mu t \quad \forall (u, v, t) \in \mathcal{S}. \quad (2.121)$$

- (d) For  $\mu$  in part (c), show that  $\mu > 0$ .
- (e) Prove that  $p^* \leq d^*$ .

**Prob 6.7 (True or False?)**

- (a) Consider a convex optimization problem:

$$\min_x f(x) : f_1(x) \leq 0 \quad (2.122)$$

where  $f_1(x)$  is a convex function. Let

$$\mathcal{S} = \{(u, t) \in \mathbf{R}^2 : \exists x \text{ such that } f_1(x) \leq u, f(x) \leq t\}. \quad (2.123)$$

Then, there exists  $(0, t) \in \mathcal{S}$ .

- (b) Suppose that  $\mathcal{S}$  and  $\mathcal{S}'$  are two disjoint non-empty convex sets which do not intersect, i.e.,  $\mathcal{S} \cap \mathcal{S}' = \emptyset$ . Then, there exists the unique pair of  $a \neq 0 \in \mathbf{R}^d$  and  $b \in \mathbf{R}$  such that

$$\begin{aligned} x \in \mathcal{S} &\implies a^T x - b \geq 0; \\ x \in \mathcal{S}' &\implies a^T x - b \leq 0. \end{aligned} \quad (2.124)$$

- (c) Suppose that  $\mathcal{S}$  and  $\mathcal{S}'$  are two disjoint non-empty convex sets which do not intersect, i.e.,  $\mathcal{S} \cap \mathcal{S}' = \emptyset$ . Suppose there exists the *unique* pair of  $a \neq 0 \in \mathbf{R}^d$  and  $b \in \mathbf{R}$  such that

$$\begin{aligned} x \in \mathcal{S} &\implies a^T x - b \geq 0; \\ x \in \mathcal{S}' &\implies a^T x - b \leq 0. \end{aligned} \quad (2.125)$$

Then, at least one among  $\mathcal{S}$  and  $\mathcal{S}'$  must be open.

## 2.5 Weak Duality

**Recap** In Part I, we investigated many instances of convex optimization problems, ranging from LP, Least Squares, QP, SOCP and all the way up to SDP. See Fig. 2.10. But in Part I, the algorithm part was not complete. We studied only two algorithms: the simplex algorithm and gradient descent. These can be applied only to specific problem settings: LP and some classes of QP. In other words, we did not learn how to design *generic* algorithms intended for general convex optimization. To this end, during the past sections, we learned about strong duality and KKT conditions. We then studied a generic algorithm inspired by them. That was, the *interior point method*. Over the last two sections, we proved the *strong duality theorem* which forms the basis of the interior point method. With all of these, we could finally end the convex optimization story.

A natural follow-up question that one can think of is: What if optimization problems of interest are *non-convex*? Can the techniques that we have studied so far help saying something about non-convex optimization? It turns out the answer is yes. Interestingly it has been shown that the techniques can serve to *approximate* optimal solutions of such non-convex optimization. In fact, in order to understand what it means, we need to study another important theory. That is, *weak duality*.

**Outline** In this section, we are going to explore weak duality in depth. Specifically what we are going to do are three folded. First of all, we will figure out what weak duality means. Like strong duality, there is a relevant important theorem, named the *weak duality theorem*. So in the first part, we will prove the theorem. Next we will investigate how weak duality can serve the claimed role: help approximating non-convex optimization problems. Lastly we will discuss how good the approximated solution is. To this end, we will figure out a gap to the optimality of the original optimization.

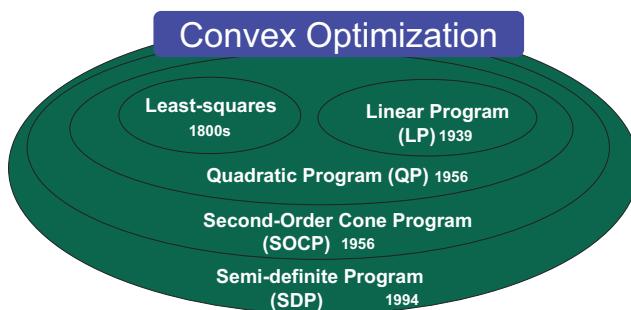


Figure 2.10. Convex optimization problems that we explored in Part I.

**Primal & dual problems** Like strong duality, weak duality is stated in the context of primal and dual problems. To define a primal problem, let us first recall the standard form of general optimization problems that we introduced in Section 1.2:

$$\begin{aligned} \min_{x \in \mathbb{R}^d} f(x) : f_i(x) \leq 0, \quad i \in \{1, \dots, m\}; \\ h_i(x) = 0, \quad i \in \{1, \dots, p\} \end{aligned} \tag{2.126}$$

where  $(f(x), f_i(x), h_i(x))$  are arbitrary scalar functions  $(\mathbb{R}^d \rightarrow \mathbb{R})$ , not necessarily *convex* or *affine* functions. Since we start with the above problem, we can say that the problem is *primal*.

In order to figure out the dual problem, we need to come up with the Lagrange and dual functions. First consider the Lagrange function:

$$\mathcal{L}(x, \lambda, \nu) := f(x) + \sum_{i=1}^m \lambda_i f_i(x) + \sum_{i=1}^p \nu_i h_i(x) \tag{2.127}$$

where  $\lambda = [\lambda_1, \dots, \lambda_m]^T \in \mathbb{R}^m$  and  $\nu = [\nu_1, \dots, \nu_p]^T \in \mathbb{R}^p$  are Lagrange multipliers. The dual function is then defined as:

$$g(\lambda, \nu) := \min_{x \in \mathcal{X}} \mathcal{L}(x, \lambda, \nu). \tag{2.128}$$

Here  $\mathcal{X}$  indicates the entire space that the optimization variable  $x$  lies in:

$$\mathcal{X} := \text{dom } f \cap \text{dom } f_1 \cap \dots \cap \text{dom } f_m \cap \text{dom } h_1 \cap \dots \cap \text{dom } h_p.$$

Using this dual function, we can formulate the dual problem as:

$$(\text{Dual problem}): \max_{\lambda, \nu} g(\lambda, \nu) : \lambda \geq 0. \tag{2.129}$$

**What does weak duality mean?** Like strong duality, we can state weak duality using the optimal values  $(p^*, d^*)$  of the primal and dual problems:

$$(\text{Primal}): p^* := \min f(x) : f_i(x) \leq 0, \quad 1 \leq i \leq m, \quad h_i(x) = 0, \quad 1 \leq i \leq p;$$

$$(\text{Dual}): d^* := \max_{\lambda, \nu} g(\lambda, \nu) : \lambda \geq 0.$$

What weak duality means is that the optimal values of the two problems respect:

$$(\text{Weak duality}): p^* \geq d^*. \tag{2.130}$$

As you may notice, we already saw weak duality (2.130) before. Actually we saw twice; one in Section 2.1, and the other in Section 2.3. But we proved this is the case only for a specific context: the *convex optimization* context. Here the most

critical point that we would like to emphasize is that weak duality (2.130) holds for *any* optimization problems which include non-convex optimization, i.e., no matter what the function types of  $(f(x), f_i(x), h_i(x))$  are. We call this the *weak duality theorem*.

**Proof of the weak duality theorem (2.130):**  $p^* \geq d^*$  The proof is almost the same as before. The proof is not tailored for the convexity condition, i.e., it can carry over to non-convex optimization as well. Let us verify that it is indeed the case.

Suppose that a feasible point in the primal problem, say  $x^*$ , achieves  $p^*$ ; similarly, another feasible point in the dual problem, say  $(\lambda^*, v^*)$ , achieves  $d^*$ . Using the fact that  $x^*$  and  $(\lambda^*, v^*)$  are the minimizer and maximizer of the primal and dual problems respectively, we get:

$$\begin{aligned}
 p^* &= f(x^*) \\
 &\stackrel{(a)}{\geq} f(x^*) + \sum_{i=1}^m \lambda_i^* f_i(x^*) + \sum_{i=1}^p v_i^* h_i(x^*) \\
 &\geq \min_{x \in \mathcal{X}} f(x) + \sum_{i=1}^m \lambda_i^* f_i(x) + \sum_{i=1}^p v_i^* h_i(x) \\
 &\stackrel{(b)}{=} g(\lambda^*, v^*) \\
 &= d^*
 \end{aligned} \tag{2.131}$$

where (a) follows from the fact that  $f_i(x^*) \leq 0$ ,  $\lambda_i^* \geq 0$  and  $h_i(x^*) = 0$  for a feasible point  $(x^*, \lambda^*, v^*)$ ; and (b) is due to the definition of the dual function.

Take a careful look at the procedures in (2.131). We never used anything about function types of  $(f(x), f_i(x), h_i(x))$ . Hence, weak duality holds for any optimization.

**Why does weak duality matter?** You may wonder why we brought up the weak duality theorem. As claimed earlier, it can serve to *approximate* the primal non-convex optimization problem. To figure out what it means, let us first point out one critical fact about the dual problem. The critical fact is that the dual problem is *always convex* no matter what the optimization type is, and therefore it is tractable (i.e., solvable). The convexity of the problem then allows us to simply focus on the tractable dual problem. It turns out by solving the tractable dual problem, one can obtain an *approximated* solution of the original non-convex primal problem with a gap of  $p^* - d^*$ .

For the rest of this section, we will provide details on the above. First we will prove that the dual problem is indeed convex. We will then figure out how the gap  $p^* - d^*$  comes up.

**Proof of the convexity of the dual problem** Let us prove that the dual problem is always convex no matter what the function types of  $(f(x), f_i(x), h_i(x))$  are. Let us start by considering the dual function:

$$g(\lambda, \nu) = \min_{x \in \mathcal{X}} f(x) + \sum_{i=1}^m \lambda_i f_i(x) + \sum_{i=1}^p \nu_i h_i(x).$$

Here one can make two key observations. The first is that for a particular value of  $x$ ,

$$f(x) + \sum_{i=1}^m \lambda_i f_i(x) + \sum_{i=1}^p \nu_i h_i(x)$$

is *affine* in  $(\lambda, \nu)$ , for any types of the functions  $(f(x), f_i(x), h_i(x))$ . The second observation is that taking the *minimum* of any affine functions, we obtain a *concave* function. Why? This is what we checked before. Hence,

$$g(\lambda, \nu) \text{ is always } \text{concave} \text{ in } (\lambda, \nu),$$

no matter what the function types of  $(f(x), f_i(x), h_i(x))$  are. Therefore, the dual problem that maximizes the concave function is *convex optimization*.

**How to solve the dual problem?** As mentioned earlier, weak duality together with the convexity of the dual problem motivates us to focus on solving the dual problem to obtain an approximate solution. So let us first discuss how to solve the dual problem:

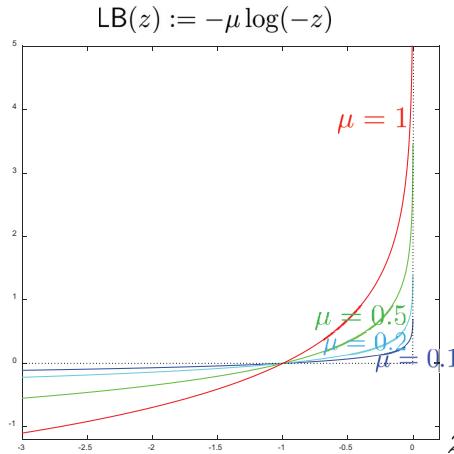
$$(\text{Dual}) \quad d^* := \max_{\lambda, \nu} g(\lambda, \nu) : \lambda \geq 0. \quad (2.132)$$

Notice that the dual problem contains an *inequality* constraint. So one cannot rely simply upon gradient descent. We should instead employ another more sophisticated algorithm. One such algorithm that we studied in Section 2.2 is: the *interior point method*. Remember that it takes the following two procedures:

1. We first approximate the problem into an unconstrained problem via the logarithmic barrier function:

$$\text{LB}(z) := -\mu \log(-z), \text{ for some } \mu > 0. \quad (2.133)$$

But we should employ a slight variant of  $\text{LB}(z)$  in approximation. Since the interested dual problem is about *maximization* (instead of minimization), we



**Figure 2.11.** Shape of the logarithmic barrier function for different  $\mu$ 's.

should employ the *minus* version of  $\text{LB}(z)$  so that it takes  $-\infty$  (instead of  $+\infty$ ) when  $z \rightarrow 0$ :

$$-\text{LB}(z) = +\mu \log(-z), \text{ for some } \mu > 0.$$

Applying this to the dual problem (2.132), we can obtain an approximated unconstrained optimization:

$$\text{(Approximated optimization)} \quad \max_{\lambda, v} g(\lambda, v) + \mu \sum_{i=1}^m \log \lambda_i. \quad (2.134)$$

Notice that  $\log \lambda_i$  comes from  $\log(-(-\lambda_i))$  in the standard form  $-\lambda \leq 0$  of the inequality constraint.

2. The next thing to do is to apply *alternating gradient descent* into the approximated optimization (2.134). The Lagrange function of the approximated optimization reads:

$$\mathcal{L}_{\text{app}}(\lambda, v) = g(\lambda, v) + \mu \sum_{i=1}^m \log \lambda_i. \quad (2.135)$$

Alternating gradient descent then allows us to find a stationary point, say  $(\tilde{\lambda}, \tilde{v})$ :

$$\nabla_\lambda \mathcal{L}_{\text{app}}(\tilde{\lambda}, \tilde{v}) = 0; \quad \nabla_v \mathcal{L}_{\text{app}}(\tilde{\lambda}, \tilde{v}) = 0. \quad (2.136)$$

**Performance of the interior point method** Remember that the performance of the interior point method depends on the control parameter  $\mu$  that appears in the logarithmic barrier (2.133). To see how it depends, consider the stationary point

$(\tilde{\lambda}, \tilde{v})$ . Then, the dual function evaluated at the point should be larger than or equal to  $d^*$ , as the dual problem is about *maximization*. Hence, we get:

$$d^* \geq g(\tilde{\lambda}, \tilde{v}).$$

Also remember what we derived w.r.t. the gap in Section 2.2. Therein, we considered the *minimization* problem. In the context, what we showed is that the gap to the optimality is upper-bounded by  $m\mu$ . Actually we can apply the same trick to obtain the same bound even w.r.t. the *maximization* problem of this section's interest. In other words, we can derive:

$$d^* - g(\tilde{\lambda}, \tilde{v}) \leq m\mu. \quad (2.137)$$

If you are not crystal clear about this, you may want to derive it, by consulting with Prob 5.3. From (2.137), we can now say that for a sufficient small value of  $\mu$ ,

$$g(\tilde{\lambda}, \tilde{v}) \approx d^*.$$

**Performance gap to optimality  $p^*$**  So under a choice of sufficiently small  $\mu$ , the gap to the optimal value  $p^*$  in the primal problem would be:

$$\text{Gap} \approx p^* - d^*.$$

There is a terminology which indicates the gap. Since the gap is w.r.t. the *dual* problem, it is called the *duality gap*.

Let us introduce another naming which refers to the approximation technique based on weak duality. We call the technique *Lagrange relaxation*. Why? Notice that the dual problem yields a *smaller* solution ( $d^* \leq p^*$ ), and the primal problem is about *minimization*. Hence, we can interpret the dual problem as the one with more *relaxed* constraints. Also it is about duality, so it employs the *Lagrange* function in the process. That's why we call it *Lagrange relaxation*.

Whenever we study a relaxation technique, we need to worry about how good the relaxation technique is. So one can ask: How far is the bound due to Lagrange relaxation from the optimal value  $p^*$ ?

**How good is Lagrange relaxation?** Here we intend to address the question by comparing to other relaxation techniques that we investigated earlier. The first relaxation technique is the one that we studied in Section 1.5 in the context of LP. That was, LP relaxation. The second is the one that we studied in Section 1.14 for a different problem context. That was, SDP relaxation. So we are interested particularly in how good Lagrange relaxation is relative to LP and SDP relaxation techniques.

**Lagrange relaxation vs. LP relaxation** First let us compare to LP relaxation. Actually such relaxation arose in a very difficult problem class that we studied before: the *Boolean problem*. So let us make a comparison under the Boolean problem. It turns out that for such a problem:

Lagrange relaxation has *the same performance as* that of LP relaxation.

**Lagrange relaxation vs. SDP relaxation** What about for SDP relaxation? It turns out that in general,

Lagrange relaxation is *at least as good as* SDP relaxation.

Another interesting result comes in for a variety of important and classical problem instances including the MAXCUT problem that we studied in Section 1.14. It turns out that for such problem instances:

Lagrange relaxation and SDP relaxation yield *the same performance*.

**Look ahead** During upcoming sections, we will study how to implement Lagrange relaxation for the interested Boolean and MAXCUT problems.

## 2.6 Lagrange Relaxation for Boolean Problems

---

**Recap** In the previous section, we switched gears to move onto *non-convex* optimization. We emphasized that the techniques that we have learned so far play a crucial role in *approximating* non-convex optimization. To understand how the approximation is implemented, we needed to study another important theory regarding duality. That is, the *weak duality theorem*:

$$p^* \geq d^* \text{ holds for any optimization problem.} \quad (2.138)$$

Here  $p^*$  and  $d^*$  indicate the optimal values of the primal and dual problems, respectively. One important thing that we proved next is that the dual problem is *always convex* no matter what the types of the functions that appear in the optimization problem are. This motivates us to focus on solving the *solvable* dual problem, and one can solve it using the interior point method. Assuming that we use a sufficiently small value of the control parameter  $\mu$  that appears in the logarithmic barrier employed for the interior point method, we can achieve  $d^*$  approximately, which in turn ensures the performance gap to the optimality to be roughly  $p^* - d^*$ . This gap is called the *duality gap*, and the approximation technique that leads to the gap is called *Lagrange relaxation*. At the end, we mentioned that Lagrange relaxation plays a significant role especially for two interested non-convex problems that we investigated earlier: Boolean and MAXCUT problems.

**Outline** In this section, we are going to investigate how to implement Lagrange relaxation for one such problem: the Boolean problem. Specifically what we are going to do are four folded. First of all, we will review the Boolean problem. Next we will express the Boolean problem in the standard form. We will then employ Lagrange relaxation to derive the dual problem. Finally we will show that the translated dual problem belongs to one convex instance that we studied in Part I: *SDP*.

**Review of Boolean problems** Remember that we discussed the Boolean problems in the context of LP. So the problems are basically on top of LP, but one special constraint is added. That is, the optimization variable  $x$  takes *binary* values. So the problem respects the following form:

$$\begin{aligned} p^* := \min \quad & w^T x : \\ & Ax - b \leq 0, \quad Cx - e = 0, \\ & x_i \in \{0, 1\}, \quad i \in \{1, \dots, d\}. \end{aligned} \quad (2.139)$$

Here the last additional constraint says that the optimization variable is constrained to be *boolean*.

**Standard form** Before studying how to implement Lagrange relaxation, let us convert the above into the standard form that contains only canonical types of constraints: inequality and/or equality constraints. Notice that the boolean constraint above (marked in red) is neither of inequality nor of equality type. But the translation to an equality constraint type is easy. The boolean constraint can be equivalently expressed as follows:

$$x_i(x_i - 1) = 0 \quad i \in \{1, \dots, d\}. \quad (2.140)$$

We now have two types of equality constraints: one is the affine constraint  $Cx - e = 0$ ; and the other is the above (2.140). In an effort to exhibit only one type of equality constraints, let us convert the affine constraint  $Cx - e = 0$  into inequality constraints. This way, we can merge them with  $Ax - b \leq 0$ , making notations simpler. We have already known how to convert the affine constraint. That is to represent it with two inequality constraint:  $Cx - e \leq 0$  and  $Cx - e \geq 0$ . Taking the conversion, we can then simplify (2.139) as:

$$\begin{aligned} p^* &:= \min w^T x : \\ Ax - b &\leq 0, \\ x_i(x_i - 1) &= 0, \quad i \in \{1, \dots, d\}. \end{aligned} \quad (2.141)$$

where  $(A, b)$  are not the same as those in the original optimization (2.139), but being the merged version with  $Cx - e \leq 0$  and  $Cx - e \geq 0$ .

**Lagrange function** Let us think about how to implement Lagrange relaxation. To this end, first consider the Lagrange function:

$$\begin{aligned} \mathcal{L}(x, \lambda, v) &= w^T x + \lambda^T (Ax - b) + \sum_{i=1}^d v_i x_i(x_i - 1) \\ &= (w + A^T \lambda - v)^T x - \lambda^T b + \sum_{i=1}^d v_i x_i^2 \\ &= (w + A^T \lambda - v)^T x - \lambda^T b + x^T \text{diag}(v_1, \dots, v_d) x \\ &= (w + A^T \lambda - v)^T x - \lambda^T b + x^T D_v x \end{aligned} \quad (2.142)$$

where the last equality follows from the definition of  $D_v := \text{diag}(v_1, \dots, v_d)$ .

**Dual function** Next consider the dual function:

$$g(\lambda, v) = \min_{x \in \mathcal{X}} x^T D_v x + (w + A^T \lambda - v)^T x - \lambda^T b \quad (2.143)$$

where  $\mathcal{X}$  denotes the entire space ( $\mathbf{R}^d$  in this example). Here  $g(\lambda, v)$  looks like a QP, as it contains a *quadratic* term  $x^T D_v x$  and  $D_v$  is *symmetric*. But it is not clear whether the associated optimization is indeed a QP. Why? The reason is that  $D_v$  is not necessarily positive semi-definite (PSD). Notice that  $D_v$  is what we can optimize over in view of the *dual problem* taking  $v$  (together with  $\lambda$ ) as an optimization variable.

Depending on what to choose for  $D_v$ , we can think of two cases. The first is sort of an easy case where the problem is indeed a QP and therefore the solution can be readily derived. In the easy case, the symmetric matrix  $D_v$  respects:

$$\text{Case I: } D_v \succeq 0. \quad (2.144)$$

In this case,  $D_v$  is indeed PSD, so the optimization problem in (2.143) becomes an *unconstrained QP*. So one can solve it by finding the stationary point  $x^*$  that satisfies:

$$\nabla_x \mathcal{L}(x^*, \lambda, v) = 2D_v x^* + w + A^T \lambda - v = 0. \quad (2.145)$$

But there is a caveat here. The caveat is that there may be no solution  $x^*$  that satisfies (2.145). To see this clearly, let us consider two subcases depending on whether the solution exists.

The first is the no-solution case:

$$\text{Case I-1: } D_v \succeq 0, \quad w + A^T \lambda - v \notin \text{range}(D_v). \quad (2.146)$$

In this case, there is no stationary point  $x^*$ . This implies that the Lagrange function  $\mathcal{L}(x, \lambda, v)$  can decrease arbitrarily small, being all the way down to  $-\infty$ . What people in the literature say about this behaviour is that  $\mathcal{L}(x, \lambda, v)$  is *unbounded below*. So in this case, the dual function  $g(\lambda, v)$  is obviously  $-\infty$ . This is definitely not an interested case. Why? The dual problem (that we will formulate soon) is about *maximization*, so  $g(\lambda, v) = -\infty$  is definitely not the one that we want to achieve. The second is the case in which there is indeed a solution:

$$\text{Case I-2: } D_v \succeq 0, \quad w + A^T \lambda - v \in \text{range}(D_v). \quad (2.147)$$

This is definitely of our interest. In this case, we can solve such  $x^*$  by resorting to the generalized definition of  $D_v^{-1}$ :

$$[D_v^{-1}]_{ii} := \begin{cases} v_i^{-1} & \text{if } v_i \neq 0; \\ 0 & \text{if } v_i = 0. \end{cases} \quad (2.148)$$

This generalized definition is needed because  $v_i = 0$  makes  $v_i^{-1}$  blow up. In the case of  $v_i = 0$ , the corresponding  $x_i$  can be anything, so  $x_i = 0$  could be a feasible

point. Hence, one can set the minimizer  $x^*$  simply as:

$$x^* = -\frac{1}{2}D_v^{-1}(w + A^T\lambda - v). \quad (2.149)$$

Plugging this into the objective function in (2.143), we can obtain the dual function as:

$$\begin{aligned} g(\lambda, v) &= \mathcal{L}(x^*, \lambda, v) \\ &= \frac{1}{4}(w + A^T\lambda - v)^T(D_v^{-1})^T D_v D_v^{-1}(w + A^T\lambda - v) \\ &\quad - \frac{1}{2}(w + A^T\lambda - v)^T D_v^{-1}(w + A^T\lambda - v) - \lambda^T b \\ &= -\frac{1}{4}(w + A^T\lambda - v)^T D_v^{-1}(w + A^T\lambda - v) - \lambda^T b. \end{aligned} \quad (2.150)$$

On the other hand, one may want to consider the other case:

$$\text{Case II: } D_v \not\succeq 0. \quad (2.151)$$

Being  $D_v \not\succeq 0$  means that there exists  $v_i < 0$  for some  $i$ . This is obviously not an interested case. Why? Recall the expression (2.142) of the Lagrange function:

$$\mathcal{L}(x, \lambda, v) = (w + A^T\lambda - v)^T x - \lambda^T b + x^T D_v x. \quad (2.152)$$

By setting  $x_i = \infty$  for such  $i$ , we get  $\mathcal{L}(x, \lambda, v) = -\infty$ . This then leads to  $g(\lambda, v) = -\infty$ . Again this is not what we want to achieve because the dual problem is about maximization.

**Dual problem** In summary, what we are interested in is only Case I-2 (2.147) wherein  $g(\lambda, v)$  is finite. Focusing on this case, we obtain the dual problem as:

$$\begin{aligned} \max_{\lambda \geq 0, v \geq 0} \quad & -\frac{1}{4}(w + A^T\lambda - v)^T D_v^{-1}(w + A^T\lambda - v) - \lambda^T b : \\ & w + A^T\lambda - v \in \text{range}(D_v) \end{aligned} \quad (2.153)$$

where the constraint  $v \geq 0$  comes from  $D_v \succeq 0$  (2.144) and the dual function  $g(\lambda, v)$  is due to (2.150). As mentioned earlier, the definition of  $D_v^{-1}$  is subject to (2.148).

Notice in the above that the first term in the objective function (marked in red) is not compatible with the standard form of any convex optimization problem that we studied earlier. But interestingly it turns out we can convert it into a form that we are familiar with: SDP. So for the rest of this section, we will translate (2.153) into an SDP.

We first introduce a new optimization variable, say  $t$ , such that

$$t \leq -\frac{1}{4}(w + A^T \lambda - v)^T D_v^{-1} (w + A^T \lambda - v).$$

A key observation here is that by maximizing  $t$ , one should make the RHS larger, and also by maximizing the RHS, one can set  $t$  larger. So this implies that maximizing  $t$  is equivalent to maximizing the RHS. Hence, with this new variable  $t$ , one can write (2.153) as:

$$\begin{aligned} \max_{\lambda \geq 0, v \geq 0} g(\lambda, v) &= \max_{\lambda \geq 0, v \geq 0, t} t - \lambda^T b : \\ w + A^T \lambda - v &\in \text{range}(D_v); \\ t &\leq -\frac{1}{4}(w + A^T \lambda - v)^T D_v^{-1} (w + A^T \lambda - v). \end{aligned} \quad (2.154)$$

The newly-introduced inequality constraint here can be alternatively written as:

$$-t - (w + A^T \lambda - v)^T (4D_v)^{-1} (w + A^T \lambda - v) \geq 0. \quad (2.155)$$

This is the one that you should be familiar with. Why? This reminds you of the Schur complement lemma. More precisely, the *generalized* Schur complement lemma. Notice that the interested matrix  $D_v$  is PSD, not necessarily PD.

**Generalized Schur complement lemma:** Suppose  $A \succeq 0$ . Then,

$$X = \begin{bmatrix} A & B \\ B^T & C \end{bmatrix} \succeq 0 \iff S := C - B^T A^\dagger B \succeq 0; \quad Bv \in \text{range}(A) \quad \forall v \quad (2.156)$$

where  $A^\dagger := U\Sigma^{-1}U^T$  when  $A = U\Sigma U^T$ . Here  $\Sigma^{-1}$  is subject to the definition of (2.148). ■

As per the above generalized lemma, as long as we define  $D_v^{-1}$  as the one in (2.148), the two constraints in (2.154) are equivalent to:

$$\begin{bmatrix} 4D_v & w + A^T \lambda - v \\ (w + A^T \lambda - v)^T & -t \end{bmatrix} \succeq 0. \quad (2.157)$$

Taking this conversion, we can write (2.154) as:

$$\begin{aligned} d^* := \max_{\lambda, v, t} t - \lambda^T b : \\ \begin{bmatrix} 4D_v & w + A^T \lambda - v \\ (w + A^T \lambda - v)^T & -t \end{bmatrix} \succeq 0, \\ \lambda \geq 0, v \geq 0. \end{aligned} \quad (2.158)$$

Notice that all the inequalities are the ones that we are familiar with. These are all LMIs. So this problem belongs to an SDP.

**Look ahead** In the next section, we will study how to implement Lagrange relaxation for another interested problem: the MAXCUT problem.

## 2.7 Lagrange Relaxation for the MAXCUT Problem

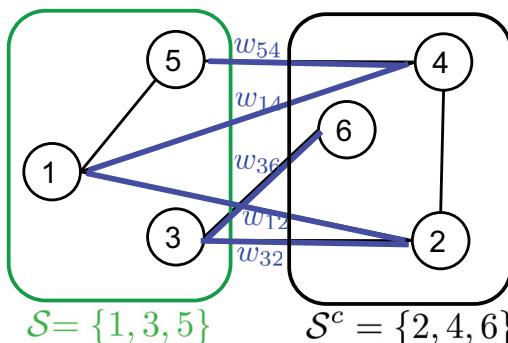
**Recap** In the previous section, we studied how to implement Lagrange relaxation in the context of the Boolean problem. We derived the dual function in closed form. We then employed the generalized Schur complement lemma to translate the dual problem into a tractable SDP. In an effort to say a few words about the performance of Lagrange relaxation, we also discussed in brief the comparison to LP relaxation.

**Outline** In this section, we will do the same thing yet w.r.t. a different problem: the MAXCUT problem, explored in Section 1.14. Specifically we are going to cover the following five stuffs. First off, we will review what the MAXCUT problem is. We will then express the MAXCUT problem in the standard form. Next we will derive the dual function in closed form, by employing the technique that we learned in the prior section. In the fourth part, we will show that the dual problem can be formulated as an SDP. Lastly we will discuss comparison to SDP relaxation that we employed for the purpose of approximating the MAXCUT problem in Section 1.14.

**Review of the MAXCUT problem** Let us start by reviewing the MAXCUT problem. The goal of the problem is to find a set that maximizes a cut. See Fig. 2.12 to refresh your memory.

To formulate an optimization problem, we introduced an optimization variable  $x_i$  that indicates whether node  $i$  is in a candidate set  $\mathcal{S}$ :

$$x_i = \begin{cases} +1, & x \in \mathcal{S}; \\ -1, & \text{otherwise.} \end{cases} \quad (2.159)$$



**Figure 2.12.** The MAXCUT problem in which the goal is to find a set that maximizes a cut. In this example, the set  $\mathcal{S} = \{1, 3, 5\}$  and the cut w.r.t. the set  $\mathcal{S}$  is  $w_{54} + w_{14} + w_{36} + w_{12} + w_{32}$ . Here  $w_{ij}$  denotes a weight associated with an edge  $(i, j) \in \mathcal{E}$ .

We then made a key observation. When  $x_i \neq x_j$ , the edge  $(i, j)$  comes across the two sets  $\mathcal{S}$  and  $\mathcal{S}^c$ , and hence, this should contribute to a cut by the amount of  $w_{ij}$ . On the other hand, when  $x_i = x_j$ , there should be no contribution to the cut. This yielded the following optimization:

$$\begin{aligned} \max_{x_i} \sum_{i=1}^d \sum_{j=1}^d \frac{1}{2} w_{ij} (1 - x_i x_j) : \\ x_i^2 = 1, \quad i \in \{1, \dots, d\} \end{aligned} \quad (2.160)$$

where  $d$  denotes the number of nodes in the graph and the constraint  $x_i^2 = 1$  respects the fact that  $x_i$  can be only either  $+1$  or  $-1$ . Here we allow for double counting, as it does not alter the optimal solution.

**Standard form** For simplification, let us multiply the objective function by 2. Since the standard form is about *minimization*, we also flip the sign in the objective to convert the problem into:

$$\begin{aligned} \text{(Primal): } p^* := \min_{x_i} \sum_{i=1}^d \sum_{j=1}^d w_{ij} (x_i x_j - 1) : \\ x_i^2 = 1, \quad i \in \{1, \dots, d\}. \end{aligned} \quad (2.161)$$

Say that this is the primal problem that we start with.

**Lagrange function** How to implement Lagrange relaxation for the primal problem (2.161)? To this end, first consider the Lagrange function:

$$\begin{aligned} \mathcal{L}(x, v) &= \sum_{i=1}^d \sum_{j=1}^d w_{ij} (x_i x_j - 1) + \sum_{i=1}^d v_i (1 - x_i^2) \\ &= - \sum_{i=1}^d \sum_{j=1}^d w_{ij} + v^T 1 + \sum_{i=1}^d \sum_{j=1}^d w_{ij} x_i x_j - \sum_{i=1}^d v_i x_i^2 \end{aligned} \quad (2.162)$$

where  $v \in \mathbf{R}^d$  denotes a Lagrange multiplier associated with the equality constraints.

Notice that the Lagrange function (2.162) contains some complicated-looking terms, which are *summation* terms. One way to succinctly represent the dirty terms is to rely upon *matrix* and *vector* notations. To apply this way, consider a matrix,

say  $W$ , which is defined as:

$$W := \begin{bmatrix} w_{11} & w_{12} & \cdots & w_{1d} \\ w_{21} & w_{22} & \cdots & w_{2d} \\ \vdots & \vdots & \ddots & \vdots \\ w_{d1} & w_{d2} & \cdots & w_{dd} \end{bmatrix} \in \mathbf{R}^{d \times d}. \quad (2.163)$$

Note that the matrix  $W$  is *symmetric*, i.e.,  $W = W^T$ , as we consider the *undirected* graph for the MAXCUT problem. We also set  $w_{ij} = 0$  when  $(i, j) \notin \mathcal{E}$ . So the diagonal entries must be zero:  $w_{ii} = 0$ .

With this  $W$  notation, we can represent the summation terms in (2.162) as:

$$\begin{aligned} \sum_{i=1}^d \sum_{j=1}^d w_{ij} &= [1, 1, \dots, 1] \begin{bmatrix} w_{11} & w_{12} & \cdots & w_{1d} \\ w_{21} & w_{22} & \cdots & w_{2d} \\ \vdots & \vdots & \ddots & \vdots \\ w_{d1} & w_{d2} & \cdots & w_{dd} \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix} \\ &= 1^T W 1; \\ \sum_{i=1}^d \sum_{j=1}^d w_{ij} x_i x_j &= \sum_{i=1}^d x_i \sum_{j=1}^d w_{ij} x_j \\ &= [x_1, x_2, \dots, x_d] \begin{bmatrix} w_{11} & w_{12} & \cdots & w_{1d} \\ w_{21} & w_{22} & \cdots & w_{2d} \\ \vdots & \vdots & \ddots & \vdots \\ w_{d1} & w_{d2} & \cdots & w_{dd} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_d \end{bmatrix}. \end{aligned}$$

Applying the above into (2.162), we then get:

$$\mathcal{L}(x, v) = -1^T W 1 + v^T 1 + x^T W x - x^T D_v x \quad (2.164)$$

where  $D_v := \text{diag}(v_1, v_2, \dots, v_d)$ .

**Dual function** Next consider the dual function:

$$g(v) = \min_{x \in \mathcal{X}} x^T (W - D_v)x - 1^T W 1 + v^T 1 \quad (2.165)$$

where  $\mathcal{X}$  indicates the entire space. Here  $g(v)$  looks like a *QP*, as it contains a *quadratic* term  $x^T (W - D_v)x$  and  $W - D_v$  is *symmetric*. But as we encountered in Section 2.6, it is not clear whether the associated optimization problem is indeed a QP. The reason is that  $W - D_v$  is not necessarily positive semi-definite (PSD). So let us consider two cases depending on whether  $W - D_v$  is PSD.

The first case is:

$$\text{Case I: } W - D_v \succeq 0 \text{ (PSD).} \quad (2.166)$$

In this case, the optimization problem in (2.165) is an *unconstrained QP*. So one can solve it by finding  $x^*$  such that

$$\nabla_x \mathcal{L}(x^*, v) = 2(W - D_v)x^* = 0 \quad (2.167)$$

if such  $x^*$  exists. Notice that there always exists such  $x^*$  satisfying the optimality condition. Hence, by putting  $x^*$  into  $\mathcal{L}(x, v)$ , we obtain the optimal value as:

$$\begin{aligned} g(v) &= \mathcal{L}(x^*, v) = x^{*T}(W - D_v)x^* - 1^T W 1 + v^T 1 \\ &= -1^T W 1 + v^T 1 \end{aligned} \quad (2.168)$$

where the last equality comes from (2.167).

On the other hand, the second case is:

$$\text{Case II: } W - D_v \not\succeq 0 \text{ (not PSD).} \quad (2.169)$$

This non-PSD condition implies that there exists  $x \in \mathbf{R}^d$  such that

$$x^T(W - D_v)x < 0.$$

Here by scaling up such  $x$  infinitely, we get  $\mathcal{L}(x, v) = -\infty$ , which in turn leads to  $g(v) = -\infty$ . Obviously this is not what we want to achieve, so we ignore the second case.

**Dual problem** Focusing on the first case (2.166) in which  $g(v)$  is finite and we have the PSD constraint of  $W - D_v \succeq 0$ , we can obtain the dual problem as:

$$\begin{aligned} d^* &:= \max_v g(v) \\ &= \max_v -1^T W 1 + v^T 1 : \\ &\quad W - D_v \succeq 0 \end{aligned} \quad (2.170)$$

where  $g(v)$  is due to (2.168). Notice that this problem is the one that we are familiar with. That is, an SDP.

**Comparison to SDP relaxation** Let us say a few words about the performance of Lagrange relaxation relative to another relaxation technique that we applied to the MAXCUT problem before (Section 1.14): SDP relaxation. To make a concrete

comparison, let us review what the optimization problem due to SDP relaxation was. To this end, first recall the primal problem (2.161) that we started with.

$$\begin{aligned} p^* := \min_x \sum_{i=1}^d \sum_{j=1}^d w_{ij} x_i x_j - 1^T W 1 : \\ x_i^2 = 1, \quad i \in \{1, \dots, d\}. \end{aligned} \tag{2.171}$$

Here we represent the problem using matrix-vector notations, in an effort to ease comparison with the Lagrange-relaxed problem (2.170), represented with such notations.

In Section 1.14, we employed a technique, called the *lifting*, to enable SDP relaxation. The idea of lifting is to raise a vector space that the optimization variable lives in, into a *matrix* space. In order to apply this idea, we introduced a new matrix  $X$  such that its  $(i, j)$ -entry  $[X]_{ij}$  is defined as:

$$X_{ij} := x_i x_j. \tag{2.172}$$

A more succinct way to represent this was:  $X = \mathbf{x}\mathbf{x}^T$ . This then yielded the following constraints:

$$X_{ii} = 1, \quad X \succeq 0, \quad \text{rank}(X) = 1. \tag{2.173}$$

Dropping the last *rank* constraint was essentially the key idea of SDP relaxation. So the SDP-relaxed problem was:

$$\begin{aligned} p_{\text{SDP}}^* := \min_X \sum_{i=1}^d \sum_{j=1}^d w_{ij} X_{ij} - 1^T W 1 : \\ X_{ii} = 1, \quad i \in \{1, \dots, d\}, \\ X \succeq 0. \end{aligned} \tag{2.174}$$

Looking at (2.170) and (2.174), a comparison seems not that straightforward. The problem (2.170) is about *maximization*, while the problem (2.174) is about *minimization*. In order to ease comparison, we can apply the strong duality theorem to obtain the dual problem of (2.174). This would be then definitely about *maximization*. However, there is an issue in formulating the dual problem of (2.174). The issue comes from the inequality constraint form in (2.174). Why? The inequality is now w.r.t. a symmetric *matrix*, not a scalar or a vector. Actually we never formulated a Lagrange function w.r.t. an optimization problem which involves such *matrix-associated inequality*. So we do not know what is a proper *Lagrange multiplier*

for the problem. But it turns out that there is a proper way of defining a Lagrange function in the matrix-inequality context. Interestingly, by applying the trick, one can show that the dual problem of (2.174) is exactly the same as that (2.170) of Lagrange relaxation. This implies that the two relaxation techniques yield the same performance. We will not delve into details here. But you will have a chance to check this in Prob 7.4 (with the help of Prob. 7.5).

**Summary of Part I and Part II** We have thus far studied lots of stuffs for both convex and non-convex optimization problems. In Part I, we have investigated many instances of convex optimization problems together with some algorithms (like the simplex algorithm and gradient descent) that can be applied to certain settings. One critical thing that we missed is about the development of *generic algorithms* that can be applied to arbitrary settings under the class.

To fill up the missing part, in Part II, we studied an important concept about duality: *strong duality*. We studied what it means and then figured out that it provides algorithmic insights into convex optimization. It leads to a famous algorithm, called the interior point method. With strong duality, we could complete the convex optimization story.

We then moved onto *non-convex* optimization problems. What we could figure out is that another important theory regarding duality helps approximating optimal solutions of non-convex optimization. That is, the *weak duality theorem*. We also studied an approximation technique based on weak duality (called Lagrange relaxation) that offers good performances for some class of difficult problems including the Boolean and MAXCUT problems. All of the above form the contents of Part I and Part II.

**Outline of Part III** A natural follow-up question arises. What can do we further with the techniques that we have learned so far? It turns out we can do something crucial in a wide variety of research fields. One such field that is quite trending during the past decades is: *Machine learning*.

So in Part III, we are going to study how the optimization techniques that we learned can play a role in the trending field. In machine learning, there are two representative learning methods: (i) *supervised learning* in which training data contain *label* information (like the identity of emails among spam vs. legitimate emails); and (ii) *unsupervised learning* in which such label information is not available. Actually supervised learning is the one that we already investigated in Part I, specifically in the context of spam filter design. In supervised learning, we will explore further, particularly the role of convex optimization, in the design of potentially better classifiers that yield sometimes the optimal performance in a certain sense. In unsupervised learning, we will study one very popular technique based on Generative

Adversarial Networks (GANs). In this context, we will figure out a powerful role of strong duality that we learned in Part II. In addition to these popular methodologies, we will investigate the role of the regularization technique and KKT conditions in the design of trending machine learning classifiers, called *fair classifiers*, which ensure *fairness* for disadvantageous against advantageous groups.

## Problem Set 7

---

**Prob 7.1 (The dual problem of LP relaxation)** Consider an LP which is relaxed from a Boolean problem:

$$\begin{aligned} \min_{x \in \mathbf{R}^d} w^T x : \\ Ax - b \leq 0, \\ 0 \leq x \leq 1 \end{aligned} \tag{2.175}$$

where  $A \in \mathbf{R}^{m \times d}$  and  $b \in \mathbf{R}^m$ . Show that the dual problem can be represented as:

$$\begin{aligned} \max_{\lambda, v, t} t - \lambda^T b : \\ w + A^T \lambda + v \geq 0, \\ -t - v^T 1 \geq 0, \\ \lambda \geq 0, v \geq 0 \end{aligned} \tag{2.176}$$

where  $\lambda \in \mathbf{R}^m$ ,  $v \in \mathbf{R}^d$  and  $t \in \mathbf{R}$ .

**Prob 7.2 (Lagrange relaxation vs. LP relaxation)** Consider the Lagrange-relaxed problem of a Boolean problem that we formulated in Section 2.6:

$$\begin{aligned} d^* := \max_{\lambda, v, t} t - \lambda^T b : \\ \begin{bmatrix} 4D_v & w + A^T \lambda - v \\ (w + A^T \lambda - v)^T & -t \end{bmatrix} \succeq 0, \\ \lambda \geq 0, v \geq 0 \end{aligned} \tag{2.177}$$

where  $D_v := \text{diag}(v_1, \dots, v_d)$ ,  $\lambda \in \mathbf{R}^m$ ,  $A \in \mathbf{R}^{m \times d}$ ,  $b \in \mathbf{R}^m$  and  $t \in \mathbf{R}$ . Also consider the dual problem of the LP-relaxed problem that we claimed in Section 2.6 and you proved in Prob 7.1:

$$\begin{aligned} d_{\text{LP}}^* := \max_{\lambda, v, t} t - \lambda^T b : \\ w + A^T \lambda + v \geq 0, \\ -t - v^T 1 \geq 0, \\ \lambda \geq 0, v \geq 0. \end{aligned} \tag{2.178}$$

Show that  $d_{\text{LP}}^* \leq d^*$ .

**Prob 7.3 (Lagrange relaxation)** Consider a Boolean problem:

$$\begin{aligned} p^* := \min w^T x : \\ Ax - b \leq 0, \\ x_i(x_i - 1) = 0, \quad i \in \{1, \dots, d\} \end{aligned} \tag{2.179}$$

where  $x, w \in \mathbf{R}^d$ ;  $A \in \mathbf{R}^{m \times d}$ ; and  $b \in \mathbf{R}^m$ . Also consider an LP relaxation of the problem (2.179):

$$\begin{aligned} p_{\text{LP}}^* := \min w^T x : \\ Ax - b \leq 0, \\ 0 \leq x \leq 1. \end{aligned} \tag{2.180}$$

- (a) Derive a Lagrange-relaxed optimization problem of (2.179) as an SDP. Also explain why it is an SDP.
- (b) Derive the dual problem of (2.180) as an SDP. Also explain why it is an SDP.
- (c) Let  $d^*$  be the optimal value of the Lagrange-relaxed optimization problem derived in part (a). Let  $d_{\text{LP}}^*$  be the optimal value of the dual problem derived in part (b). A student claims that there exists a case in which  $d_{\text{LP}}^* < d^*$ . Prove or disprove this claim.

**Prob 7.4 (Lagrange relaxation vs. SDP relaxation)** Consider the optimization problem of the MAXCUT problem that we studied in Section 2.7.

$$p^* := \min_{x=(x_1, \dots, x_d)} \sum_{i=1}^d \sum_{j=1}^d w_{ij}(1 - x_i x_j) : x_i^2 = 1 \tag{2.181}$$

where  $w_{ij} \in \mathbf{R}$  denotes a weight associated with an edge  $(i, j)$  in an undirected graph  $\mathcal{G}$ .

- (a) Using Lagrange relaxation, derive the dual problem of (2.181) as an SDP. Also explain why it is an SDP.
- (b) Using SDP relaxation together with the lifting technique, show that the optimization (2.181) can be approximated as:

$$p_{\text{SDP}}^* := \min_X \sum_{i=1}^d \sum_{j=1}^d w_{ij}(X_{ij} - 1) : X_{ii} = 1, X \succeq 0. \tag{2.182}$$

- (c) Let  $d^*$  be the optimal value of the dual problem derived in part (a). Let  $d_{\text{SDP}}^*$  be the optimal value of the dual problem w.r.t. the SDP-relaxed optimization (2.182). A student claims that  $d_{\text{SDP}}^* = d^*$ . Prove or disprove this claim.

**Prob 7.5 (Weak duality for SDP)** Consider an optimization problem:

$$\begin{aligned} p^* := \max_{x \in \mathbf{R}^d} f(x) : \\ G + x_1 F_1 + \cdots + x_d F_d \succeq 0 \end{aligned} \tag{2.183}$$

where  $G \in \mathbf{R}^{m \times m}$  and  $F_i \in \mathbf{R}^{m \times m}$  are symmetric. Assume that  $f(x)$  is finite.

- (a) Propose a proper way of defining the Lagrange function, the dual function and the dual problem that leads to the weak duality theorem.
- (b) Employing the way proposed in part (a), derive the dual problem of (2.183).
- (c) Let  $d^*$  be the optimal value of the dual problem derived in part (b). State the weak duality theorem that shows the relationship between  $p^*$  and  $d^*$ . Also prove the theorem.

**Prob 7.6 (Weak duality for SDP)** Consider an optimization problem:

$$\begin{aligned} p^* := \min f(x) : \\ G + x_1 F_1 + \cdots + x_d F_d \succeq 0 \end{aligned} \tag{2.184}$$

where  $x := [x_1, \dots, x_d]^T \in \mathbf{R}^d$ , and  $G \in \mathbf{R}^{m \times m}$  and  $F_i \in \mathbf{R}^{m \times m}$  are symmetric. Assume that  $f(x)$  is finite. Let  $Z \in \mathbf{R}^{m \times m}$  be symmetric. Define the Lagrange function as:

$$\mathcal{L}(x, Z) := f(x) - \text{trace}(Z(G + x_1 F_1 + \cdots + x_d F_d)). \tag{2.185}$$

- (a) Derive the dual problem of (2.184).
- (b) Prove that  $p^* \geq d^*$ .

**Prob 7.7 (True or False?)**

- (a) Consider an optimization problem:

$$\begin{aligned} \min f(x) : \\ G + x_1 F_1 + \cdots + x_d F_d \succeq 0 \end{aligned} \tag{2.186}$$

where  $x \in \mathbf{R}^d$ , and  $G \in \mathbf{R}^{m \times m}$  and  $F_i \in \mathbf{R}^{m \times m}$  are symmetric. Then, one can derive the dual problem only when  $f(x)$  is finite  $\forall x$ .

## Chapter 3

# Machine Learning Applications

### 3.1 Supervised Learning and Optimization

---

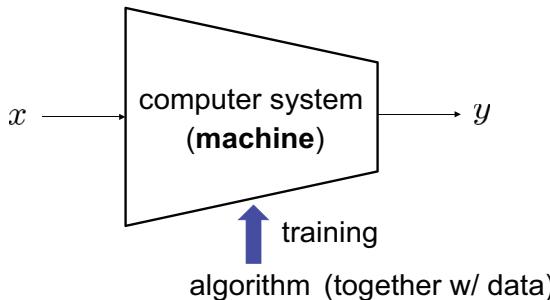
**Recap** In Part II, we focused on the study of the two important theorems: (i) strong duality theorem; (ii) weak duality theorem. The strong duality theorem provided algorithmic insights, thus leading to the interior point method that can be applied to generic settings that we have investigated in Part I. The weak duality theorem helped us to approximate optimal solutions for non-convex optimization problems, which are intractable in general.

At the end of Part II, we emphasized that what we have learned so far are instrumental in addressing important issues that arise in a trending research field: *Machine learning*. So the goal of Part III is to support this claim.

**Outline** In this section, we will start investigating the field of machine learning and the role of optimization therein. What we are going to do are three folded. First of all, we will study what *machine learning* is and what the mission of the field is. We will then explore one very popular & traditional way to achieve the mission:

*Supervised learning.*

Lastly we will figure out how optimization techniques are related to supervised learning.



**Figure 3.1.** Machine learning is the study of algorithms which provide a set of instructions to a computer system so that it can perform a specific task of interest. Let  $x$  be an input indicating information employed to perform a task. Let  $y$  be an output to denote a task result.

**Machine learning** Machine learning is about an algorithm which is defined to be a set of instructions that a computer system can execute. Formally speaking, machine learning is the study of *algorithms* with which one can train a computer system so that it can perform a specific task of interest. See Fig. 3.1 for pictorial illustration.

The entity that we are interested in building up is a computer system, which is definitely a *machine*. Since it is a system (i.e., a function), it has input and output. The input, usually denoted by  $x$ , indicates information employed to perform a task of interest. The output, usually denoted by  $y$ , denotes a task result. For instance, if a task is legitimate-vs-spam email filtering that we studied in Part I, then  $x$  could be *features* (e.g., frequencies of some keywords that appear in an email), and  $y$  is an email entity, e.g.,  $y = +1$  indicates a legitimate email while  $y = -1$  denotes a spam email. Or if an interested task is cat-vs-dog classification,  $x$  could be *image-pixel values* and  $y$  is a binary value indicating whether the fed image is a cat (say  $y = 1$ ) or a dog ( $y = 0$ ).

Machine learning is about designing *algorithms*, wherein the main role is to train the computer system so that it can perform the task well. In the process of designing algorithms, we use something, called *data*.

**A remark on the naming** One can easily see the rationale behind the naming via changing a viewpoint. From a *machine's perspective*, a *machine learns* the task from data. Hence, it is called *machine learning*. This naming was coined in 1959 by Arthur Lee Samuel (Samuel, 1967). See Fig. 3.2.

Arthur Samuel is one of the pioneers in *Artificial Intelligence* (AI) which includes machine learning as a sub-field. The AI field is the study of creating *intelligence* by *machines*, unlike the *natural intelligence* displayed by intelligent beings like humans and animals.



Arthur Samuel '59



checkers

**Figure 3.2.** Arthur Lee Samuel is an American pioneer in the field of artificial intelligence. One of his prominent achievements in early days is to develop computer checkers which later formed the basis of AlphaGo.

One of his achievements in early days is to develop a human-like computer player for a board game, called *checkers*; see the right figure in Fig. 3.2. He proposed many algorithms and ideas while developing computer checkers. Those algorithms could form the basis of *AlphaGo* (Silver *et al.*, 2016), a computer program for the board game Go which defeated one of the 9-dan professional players, Lee Sedol, with 4 wins out of 5 games in 2016 (News, 2016).

**Mission of machine learning** The end-mission of machine learning is *achieving artificial intelligence (AI)*. So it can be viewed as one methodology for AI. In light of the block diagram in Fig. 3.1, one can say that the goal of ML is to design an algorithm so that the trained machine *behaves like intelligent beings*.

**Supervised learning** There are some methodologies which help us to achieve the goal of ML. One specific yet very popular method is the one called:

*Supervised learning.*

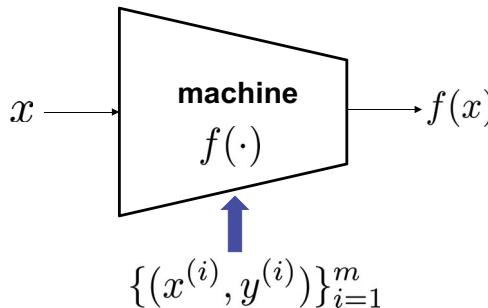
What supervised learning means is *learning* a function  $f(\cdot)$  (indicating a functional of the machine) with the help of a *supervisor*. See Fig. 3.3.

What the supervisor means in this context is the one who provides input-output paired samples. Obviously the input-output samples form *data* employed for training the machine, usually denoted by:

$$\{(x^{(i)}, y^{(i)})\}_{i=1}^m \quad (3.1)$$

where  $(x^{(i)}, y^{(i)})$  indicates the  $i$ th input-output sample (or called a *training sample* or an *example*) and  $m$  denotes the number of samples. Using this notation (3.1), one can say that supervised learning is to:

$$\text{Estimate } f(\cdot) \text{ using the training samples } \{(x^{(i)}, y^{(i)})\}_{i=1}^m. \quad (3.2)$$



**Figure 3.3.** Supervised learning: A methodology for designing a computer system  $f(\cdot)$  with the help of a supervisor which offers input-output pair samples, called a training dataset  $\{(x^{(i)}, y^{(i)})\}_{i=1}^m$ .

**Optimization** A common way to estimate  $f(\cdot)$  is via *optimization*. This is exactly how the optimization techniques that we learned are related to supervised learning.

In view of the goal (3.2) of supervised learning, what we want is:

$$y^{(i)} \approx f(x^{(i)}), \quad \forall i \in \{1, \dots, m\}.$$

Then, a natural question arises. How to quantify closeness between the two quantities:  $y^{(i)}$  and  $f(x^{(i)})$ ? One very common way that has been used in the field is to employ a function, called a *loss* function, usually denoted by:

$$\ell(y^{(i)}, f(x^{(i)})). \quad (3.3)$$

One obvious property that the loss function  $\ell(\cdot, \cdot)$  should have is that it should be small when the two arguments are close, while being zero when the two are identical. One prominent loss function that you saw earlier is: the squared error loss, introduced by the father of optimization, Gauss:  $\|y^{(i)} - f(x^{(i)})\|^2$ .

Using the loss function (3.3), one can then formulate an optimization problem as follows:

$$\min_{f(\cdot)} \sum_{i=1}^m \ell(y^{(i)}, f(x^{(i)})). \quad (3.4)$$

In fact, this is not of the conventional optimization problem structure that we are familiar with. In (3.4), the quantity that we optimize over is *the function  $f(\cdot)$* , marked in red.

We never saw this type of optimization, called *function optimization*. How to deal with such function optimization? There is one typical approach in the field. The approach is to specify a function class (e.g., linear or quadratic), represent the function with *parameters* (or called *weights*), denoted by  $w$ , and then consider the



Frank Rosenblatt '57

**Figure 3.4.** Frank Rosenblatt (1928–1971) is an American psychologist notable as the inventor of Perceptron. One sad story is that he died in 1971 on his 43rd birthday, in a boating accident.

weights as an optimization variable. Taking this approach, one can translate the problem (3.4) into:

$$\min_{\mathbf{w}} \sum_{i=1}^m \ell(y^{(i)}, f_{\mathbf{w}}(\mathbf{x}^{(i)})) \quad (3.5)$$

where  $f_{\mathbf{w}}(\mathbf{x}^{(i)})$  denotes the function  $f(\mathbf{x}^{(i)})$ , parameterized by  $\mathbf{w}$ .

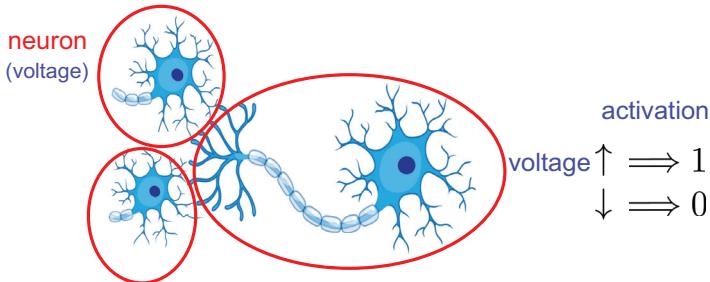
Now how is the optimization problem (3.5) related to convex optimization that we have thus far learned about? To see this, we need to check whether the objective function  $\ell(y^{(i)}, f_{\mathbf{w}}(\mathbf{x}^{(i)}))$  is convex in the optimization variable  $\mathbf{w}$ . Obviously the convexity depends on how we define the two functions: (i)  $f_{\mathbf{w}}(\mathbf{x}^{(i)})$  w.r.t.  $\mathbf{w}$ ; and (ii) the loss function  $\ell(\cdot, \cdot)$ . In machine learning, lots of works have been done for the choice of the functions.

**Introduction of neural networks** Around at the same time when the ML field was founded, one architecture was suggested for the first function  $f_{\mathbf{w}}(\cdot)$  in the context of simple binary classifiers in which  $y$  takes one among the two options only. The architecture is called:

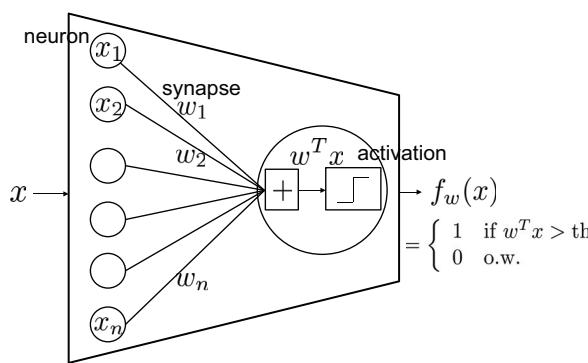
*Perceptron,*

and was invented in 1957 by one of the pioneers in the AI field, named Frank Rosenblatt (Rosenblatt, 1958). See Fig. 3.4. Interestingly, Frank Rosenblatt was a *psychologist*. So he was interested in how brains of intelligent beings work and his study on brains led him to come up with Perceptron which is inspired by the brain structure and therefore gave significant insights into *neural networks*.

**How brains work** Here are details on how the brain structure inspired the architecture of Perceptron. Inside a brain, there are many *electrically excitable cells*, called *neurons*; see Fig. 3.5. Here a red-circled one indicates a neuron. So the figure shows



**Figure 3.5.** Neurons are electrically excitable cells and are connected through synapses.



**Figure 3.6.** The architecture of Perceptron.

three neurons in total. There are three major properties about neurons that led to the architecture of Perceptron.

The first is that a neuron is an *electrical* quantity, so it has a *voltage*. The second property is that neurons are connected with each other through mediums, called *synapses*. So the main role of synapses is to deliver electrical voltage signals across neurons. Depending on the connectivity strength level of a synapse, a voltage signal from one neuron to another can increase or decrease. The last is that a neuron takes a particular action, called *activation*. Depending on its voltage level, it generates an all-or-nothing pulse signal. For instance, if its voltage level is above a certain threshold, then it generates an impulse signal with a certain magnitude, say 1; otherwise, it produces nothing.

**Perceptron** The above three properties about neurons led Frank Rosenblatt to propose the architecture of Perceptron, as illustrated in Fig. 3.6.

Let  $x$  be an  $n$ -dimensional real-valued signal:  $x := [x_1, x_2, \dots, x_n]^T$ . Suppose each component  $x_i$  is distributed to each neuron, and let us interpret  $x_i$  as a voltage level of the  $i$ th neuron. The voltage signal  $x_i$  is then delivered through a synapse to another neuron (placed on the right in the figure, indicated by a big circle).

Remember that the voltage level can increase or decrease depending on the connectivity strength of a synapse. To capture this, a weight, say  $w_i$ , is multiplied to  $x_i$  so  $w_i x_i$  is a delivered voltage signal at the terminal neuron. Based on another observation that people made on neurons that the voltage level at the terminal neuron increases with more connected neurons, Rosenblatt introduced an adder which simply aggregates all the voltage signals coming from many neurons, so he modeled the voltage signal at the terminal neuron as:

$$w_1 x_1 + w_2 x_2 + \cdots + w_n x_n = w^T x. \quad (3.6)$$

Lastly in an effort to mimic the *activation*, he modeled the output signal as

$$f_w(x) = \begin{cases} 1 & \text{if } w^T x > \text{th}, \\ 0 & \text{o.w.} \end{cases} \quad (3.7)$$

where “th” indicates a certain threshold level. It can also be simply denoted as

$$f_w(x) = \mathbf{1}\{w^T x > \text{th}\}. \quad (3.8)$$

**Activation functions** Taking the Perceptron architecture in Fig. 3.6, one can formulate the optimization problem (3.5) as:

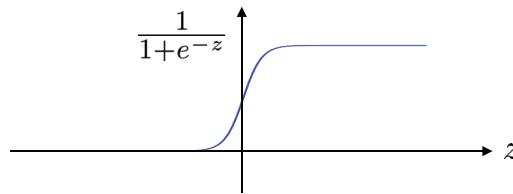
$$\min_w \sum_{i=1}^m \ell\left(y^{(i)}, \mathbf{1}\{w^T x^{(i)} > \text{th}\}\right). \quad (3.9)$$

This is an initial optimization problem that people developed, inspired by Perceptron. However, people immediately figured out there is an issue in solving this optimization. The issue comes from the fact that the objective function contains an indicator function, so it is *not differentiable*. Why is the non-differentiability problematic? Remember the algorithms that we learned in the past: gradient descent and the interior point method. All of them involve *derivatives* operations. So the non-differentiability of the objective function does not allow us to enjoy the algorithms.

What can we do? One typical way that people have taken in the field is to *approximate* the activation function. There are many ways for approximation. From below, we will investigate one popular approach.

**Logistic regression** The popular approximation approach is to take sort of a *smooth* transition from 0 to 1 for the abrupt indicator function:

$$f_w(x) = \frac{1}{1 + e^{-w^T x}}. \quad (3.10)$$



**Figure 3.7.** Logistic function:  $\sigma(z) = \frac{1}{1+e^{-z}}$ .

Notice that  $f_w(x) \approx 0$  when  $w^T x$  is very small; it then grows exponentially with an increase in  $w^T x$ ; later grows logarithmically; and finally saturates as 1 when  $w^T x$  is very large. See Fig. 3.7. Actually the function (3.10) is a very popular one used in statistics, called the *logistic*<sup>1</sup> function (Garnier and Quetelet, 1838). There is another name: the *sigmoid*<sup>2</sup> function.

There are two good things about the logistic function. First it is differentiable. Second, it can play a role as the *probability* for the output in the binary classifier, e.g.,  $\mathbb{P}(y=1)$  where  $y$  denotes the ground-truth label in the binary classifier. So it is very much interpretable.

For this function, people came up with a loss function, which turns out to be *optimal in some sense* and expressed as:

$$\ell(y, \hat{y}) = -y \log \hat{y} - (1 - y) \log(1 - \hat{y}). \quad (3.11)$$

This function is called *cross entropy loss* (Cover, 1999) and the rationale behind the naming will be explained later. Taking the logistic function together with cross entropy loss, we can then formulate the problem (3.5) as:

$$\min_w \sum_{i=1}^m -y^{(i)} \log \frac{1}{1 + e^{-w^T x^{(i)}}} - (1 - y^{(i)}) \log \frac{e^{-w^T x^{(i)}}}{1 + e^{-w^T x^{(i)}}}. \quad (3.12)$$

It turns out this optimization problem is *convex* and provides the *optimal* performance in some sense. The name of this classifier is *logistic regression*.

**Look ahead** In the next section, we will show that logistic regression is indeed a *convex* classifier. We will also study in what sense logistic regression is *optimal* and will prove the optimality.

1. The word *logistic* comes from a Greek word which means a slow growth, like a logarithmic growth.
2. Sigmoid means resembling the lower-case Greek letter sigma, S-shaped.

## 3.2 Logistic Regression

---

**Recap** In the last section, we embarked on Part III which focuses on machine learning applications. We emphasized that the optimization techniques play a significant role in the field. Specifically we focused on one machine learning methodology: supervised learning, which serves as a powerful and classical methodology in achieving the goal of ML. Recall that the goal of supervised learning is to estimate a function  $f_w(\cdot)$  of a machine using input-output samples:  $\{(x^{(i)}, y^{(i)})\}_{i=1}^m$  where  $m$  denotes the number of training samples. Here  $w$  indicates a collection of parameters which serve to implement the function  $f_w(\cdot)$ . For the function  $f_w(\cdot)$ , we studied one specific yet historical architecture, inspired by brains' neural networks: *Perceptron*. It first takes a linear operation with an input, say  $x$ , to compute  $w^T x$ . It then passes it to an activation function to yield an output, say  $\hat{y} := f_w(x)$ .

Next, we formulated an optimization problem accordingly:

$$\min_w \sum_{i=1}^m \ell(y^{(i)}, f_w(x^{(i)})) \quad (3.13)$$

where  $\ell(y^{(i)}, f_w(x^{(i)}))$  indicates a loss function which quantifies closeness between the ground truth label  $y^{(i)}$  and its estimate  $f_w(x^{(i)})$ . Since the stair-shaped activation function that Frank Rosenblatts introduced initially is not differentiable and thus inapplicable to algorithms like gradient descent, we approximated the activation via the logistic function  $f_w(x) = \frac{1}{1+e^{-w^T x}}$ . Taking this together with a loss function, called *cross entropy loss*,

$$\ell_{\text{CE}}(y, \hat{y}) = -y \log \hat{y} - (1-y) \log(1-\hat{y}), \quad (3.14)$$

we obtained a predictor, named *logistic regression*. We then claimed that logistic regression is formulated as a *convex* optimization problem. We also claimed that the predictor based on cross entropy loss (3.14) yields the *optimal* performance in *some sense*.

**Outline** In this section, we will prove this claim. What we are going to do are three folded. First, we will show that logistic regression is indeed a *convex* predictor. Next, we will investigate *in what sense* the predictor is *optimal*. We will then prove the optimality. In addition, we will discuss how to solve logistic regression.

**Objective function in logistic regression** Taking the logistic function and cross entropy loss (3.14), we obtain:

$$\min_w \sum_{i=1}^m \ell_{\text{CE}}(y^{(i)}, f_w(x^{(i)}))$$

$$\begin{aligned}
&= \min_w \sum_{i=1}^m \ell_{\text{CE}} \left( y^{(i)}, \frac{1}{1 + e^{-w^T x^{(i)}}} \right) \\
&= \min_w \sum_{i=1}^m -y^{(i)} \log \frac{1}{1 + e^{-w^T x^{(i)}}} - (1 - y^{(i)}) \log \frac{e^{-w^T x^{(i)}}}{1 + e^{-w^T x^{(i)}}}. \quad (3.15)
\end{aligned}$$

**Proof of convexity** Let us first prove that the optimization problem (3.15) is indeed convex. Since convexity preserves under addition, it suffices to prove:

(i)  $-\log \frac{1}{1 + e^{-w^T x}}$  is convex in  $w$ ;

(ii)  $\log \frac{e^{-w^T x}}{1 + e^{-w^T x}}$  is convex in  $w$ .

Since the 2nd function can be represented as the sum of an affine function and the 1st function:

$$-\log \frac{e^{-w^T x}}{1 + e^{-w^T x}} = w^T x - \log \frac{1}{1 + e^{-w^T x}},$$

it suffices to prove the convexity of the 1st function.

Notice that the 1st function can be rewritten as:

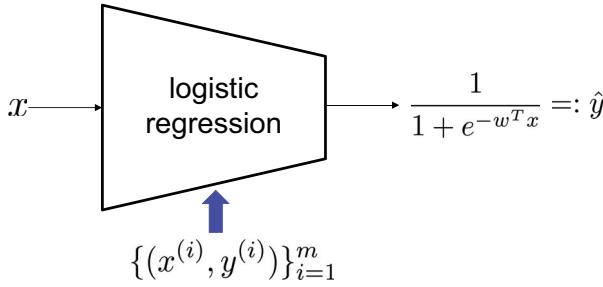
$$-\log \frac{1}{1 + e^{-w^T x}} = \log(1 + e^{-w^T x}). \quad (3.16)$$

Taking a derivative of the RHS formula in (3.16) w.r.t.  $w$ , we get:

$$\nabla_w \log(1 + e^{-w^T x}) = \frac{-xe^{-w^T x}}{1 + e^{-w^T x}}.$$

This is due to a chain rule of derivatives. Taking another derivative of the above, we obtain a Hessian as follows:

$$\begin{aligned}
\nabla_w^2 \log(1 + e^{-w^T x}) &= \nabla_w \left( \frac{-xe^{-w^T x}}{1 + e^{-w^T x}} \right) \\
&\stackrel{(a)}{=} \frac{xx^T e^{-w^T x}(1 + e^{-w^T x}) - xx^T e^{-w^T x} e^{-w^T x}}{(1 + e^{-w^T x})^2} \\
&= \frac{xx^T e^{-w^T x}}{(1 + e^{-w^T x})^2} \\
&\stackrel{(b)}{\succeq} 0
\end{aligned} \quad (3.17)$$



**Figure 3.8.** Logistic regression.

where (a) is due to the derivative rule of a quotient of two functions and (b) follows from the fact that the eigenvalue of  $xx^T$  is  $x^T x \geq 0$  (why?). As the Hessian is PSD, we prove the convexity.

**In what sense logistic regression is optimal?** Notice that the range of the output  $\hat{y}$  is in between 0 and 1:

$$0 \leq \hat{y} \leq 1.$$

Hence, one can interpret this as a *probability* quantity. The optimality of a predictor can be defined under the following assumption inspired by the probabilistic interpretation:

$$\text{Assumption : } \hat{y} = \mathbb{P}(y = 1|x). \quad (3.18)$$

To understand what it means in detail, consider the *likelihood* of the ground-truth predictor:

$$\mathbb{P}\left(\{y^{(i)}\}_{i=1}^m \mid \{x^{(i)}\}_{i=1}^m\right). \quad (3.19)$$

Notice that the output  $\hat{y}$  is a function of weights  $w$ . Hence, we see that assuming (3.18), the likelihood (3.19) is also a function of  $w$ .

We are now ready to define the optimal  $w$ . The optimal weight, say  $w^*$ , is defined as the one that *maximizes the likelihood* (3.19):

$$w^* := \arg \max_w \mathbb{P}\left(\{y^{(i)}\}_{i=1}^m \mid \{x^{(i)}\}_{i=1}^m\right). \quad (3.20)$$

Of course, there are other ways to define the optimality. Here, we employ the maximum likelihood principle, the most popular choice. This is exactly where the definition of the *optimal loss function*, say  $\ell^*(\cdot, \cdot)$  kicks in. We say that  $\ell^*(\cdot, \cdot)$  is defined

as the one that satisfies:

$$\arg \min_w \sum_{i=1}^m \ell^* \left( y^{(i)}, \hat{y}^{(i)} \right) = \arg \max_w \mathbb{P} \left( \{y^{(i)}\}_{i=1}^m | \{x^{(i)}\}_{i=1}^m \right). \quad (3.21)$$

It turns out the condition (3.21) would give us the optimal loss function  $\ell^*(\cdot, \cdot)$  that yields a well-known classifier: logistic regression, in which the loss function reads:

$$\ell^*(y, \hat{y}) = \ell_{\text{CE}}(y, \hat{y}) = -y \log \hat{y} - (1 - y) \log(1 - \hat{y}). \quad (3.22)$$

Now let us prove (3.22).

**Proof of  $\ell^*(\cdot, \cdot) = \ell_{\text{CE}}(\cdot, \cdot)$**  Usually samples are obtained from different data  $x^{(i)}$ 's. Hence, it is reasonable to assume that such samples are independent with each other:

$$\{(x^{(i)}, y^{(i)})\}_{i=1}^m \text{ are independent over } i. \quad (3.23)$$

Under this assumption, we can rewrite the likelihood (3.19) as:

$$\begin{aligned} \mathbb{P} \left( \{y^{(i)}\}_{i=1}^m | \{x^{(i)}\}_{i=1}^m \right) &\stackrel{(a)}{=} \frac{\mathbb{P} \left( \{(x^{(i)}, y^{(i)})\}_{i=1}^m \right)}{\mathbb{P} \left( \{x^{(i)}\}_{i=1}^m \right)} \\ &\stackrel{(b)}{=} \frac{\prod_{i=1}^m \mathbb{P}(x^{(i)}, y^{(i)})}{\prod_{i=1}^m \mathbb{P}(x^{(i)})} \\ &\stackrel{(c)}{=} \prod_{i=1}^m \mathbb{P} \left( y^{(i)} | x^{(i)} \right) \end{aligned} \quad (3.24)$$

where (a) and (c) are due to the definition of conditional probability; and (b) comes from the independence assumption (3.23). Here  $\mathbb{P}(x^{(i)}, y^{(i)})$  denotes the probability distribution of the input-output pair of the system:

$$\mathbb{P}(x^{(i)}, y^{(i)}) := \mathbb{P}(X = x^{(i)}, Y = y^{(i)}) \quad (3.25)$$

where  $X$  and  $Y$  indicate random variables of the input and the output, respectively.

Recall the probability-interpretation-related assumption (3.18) made with regard to  $\hat{y}$ :

$$\hat{y} = \mathbb{P}(y = 1 | x).$$

This implies that:

$$\begin{aligned} y = 1 : \quad \mathbb{P}(y|x) &= \hat{y}; \\ y = 0 : \quad \mathbb{P}(y|x) &= 1 - \hat{y}. \end{aligned}$$

Hence, one can represent  $\mathbb{P}(y|x)$  as:

$$\mathbb{P}(y|x) = \hat{y}^y (1 - \hat{y})^{1-y}.$$

Now using the notations of  $(x^{(i)}, y^{(i)})$  and  $\hat{y}^{(i)}$ , we obtain:

$$\mathbb{P}\left(y^{(i)}|x^{(i)}\right) = (\hat{y}^{(i)})^{y^{(i)}} (1 - \hat{y}^{(i)})^{1-y^{(i)}}.$$

Plugging this into (3.24), we get:

$$\begin{aligned} &\mathbb{P}\left(\{y^{(i)}\}_{i=1}^m | \{x^{(i)}\}_{i=1}^m\right) \\ &= \prod_{i=1}^m \mathbb{P}(x^{(i)}) \prod_{i=1}^m (\hat{y}^{(i)})^{y^{(i)}} (1 - \hat{y}^{(i)})^{1-y^{(i)}}. \end{aligned} \tag{3.26}$$

This together with (3.20) yields:

$$\begin{aligned} w^* &:= \arg \max_w \prod_{i=1}^m (\hat{y}^{(i)})^{y^{(i)}} (1 - \hat{y}^{(i)})^{1-y^{(i)}} \\ &\stackrel{(a)}{=} \arg \max_w \sum_{i=1}^m y^{(i)} \log \hat{y}^{(i)} + (1 - y^{(i)}) \log(1 - \hat{y}^{(i)}) \\ &\stackrel{(b)}{=} \arg \min_{\textcolor{red}{w}} \sum_{i=1}^m -y^{(i)} \log \hat{y}^{(i)} - (1 - y^{(i)}) \log(1 - \hat{y}^{(i)}) \end{aligned} \tag{3.27}$$

where (a) comes from the fact that  $\log(\cdot)$  is a non-decreasing function and  $\prod_{i=1}^m (\hat{y}^{(i)})^{y^{(i)}} (1 - \hat{y}^{(i)})^{1-y^{(i)}}$  is positive; and (b) is due to changing a sign of the objective while replacing max with min.

Note that the term inside the summation in the last equality in (3.27) respects the formula of *cross entropy loss*:

$$\ell_{\text{CE}}(y, \hat{y}) := -y \log \hat{y} - (1 - y) \log(1 - \hat{y}). \tag{3.28}$$

Hence, the optimal loss function that yields the maximum likelihood solution is:

$$\ell^*(\cdot, \cdot) = \ell_{\text{CE}}(\cdot, \cdot).$$

**Remarks on cross entropy loss (3.28)** Before moving onto the next topic about how to solve the optimization problem (3.15), let us say a few words about why the loss function (3.28) is called *cross entropy loss*. This naming comes from the definition of *cross entropy*, which is a measure used in the field of *information theory*. The cross entropy is defined w.r.t. two random variables. For simplicity, let us consider two binary random variables, say  $X \sim \text{Bern}(p)$  and  $Y \sim \text{Bern}(q)$  where  $X \sim \text{Bern}(p)$  indicates a binary random variable with  $p = \mathbb{P}(X = 1)$ . For such two random variables, the cross entropy is defined as (Cover, 1999):

$$H(p, q) := -p \log q - (1 - p) \log(1 - q). \quad (3.29)$$

Notice that the formula of (3.28) is exactly the same as the term inside summation in (3.27), except for having different notations. Hence, it is called *cross entropy loss*.

Here you may wonder why  $H(p, q)$  in (3.29) is called *cross entropy*. The rationale comes from the following fact:

$$H(p, q) \geq H(p) := -p \log p - (1 - p) \log(1 - p) \quad (3.30)$$

where  $H(p)$  is a very-well known quantity in information theory, named *entropy* or *Shannon entropy* (Shannon, 2001). One can actually prove the inequality in (3.30) using Jensen's inequality. Also one can verify that the equality holds when  $p = q$ . We will not prove this here. But you will have a chance to check this in Prob 8.3(b). So from this, one can interpret  $H(p, q)$  as an *entropic*-measure of discrepancy across distributions. Hence, it is called *cross entropy*.

**How to solve logistic regression (3.15)?** Let  $J(w)$  be the objective function normalized by the number  $m$  of samples:

$$J(w) := \frac{1}{m} \sum_{i=1}^m -y^{(i)} \log \hat{y}^{(i)} - (1 - y^{(i)}) \log(1 - \hat{y}^{(i)}). \quad (3.31)$$

Remember in the beginning of this section that we proved the convexity of the objective function. Also the optimization problem (3.15) is unconstrained. Hence, we can use gradient descent to find the optimal solution. Specifically one can take the following update rule:

$$w^{(t+1)} \leftarrow w^{(t)} - \alpha^{(t)} \nabla J(w^{(t)})$$

where  $w^{(t)}$  denotes the estimate of weights at the  $t$ th iteration, and  $\alpha^{(t)}$  indicates the learning rate. The gradient of the normalized objective function can be computed as:

$$\nabla J(w) = \frac{1}{m} \sum_{i=1}^m -y^{(i)} \frac{\nabla \hat{y}^{(i)}}{\hat{y}^{(i)}} + (1 - y^{(i)}) \frac{\nabla \hat{y}^{(i)}}{1 - \hat{y}^{(i)}}. \quad (3.32)$$

Here the gradient of  $y^{(i)}$  (marked in red) can be expressed as:

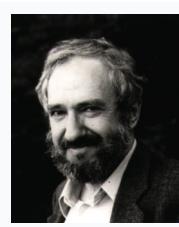
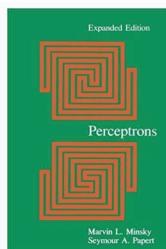
$$\nabla \hat{y}^{(i)} = \frac{x^{(i)} e^{-w^T x^{(i)}}}{(1 + e^{-w^T x^{(i)}})^2} = x^{(i)} \hat{y}^{(i)} (1 - \hat{y}^{(i)})$$

Plugging this to (3.32), we get:

$$\begin{aligned} \nabla J(w) &= \frac{1}{m} \sum_{i=1}^m x^{(i)} \left\{ -y^{(i)} (1 - \hat{y}^{(i)}) + (1 - y^{(i)}) \hat{y}^{(i)} \right\} \\ &= \frac{1}{m} \sum_{i=1}^m x^{(i)} \left\{ \hat{y}^{(i)} - y^{(i)} \right\}. \end{aligned}$$

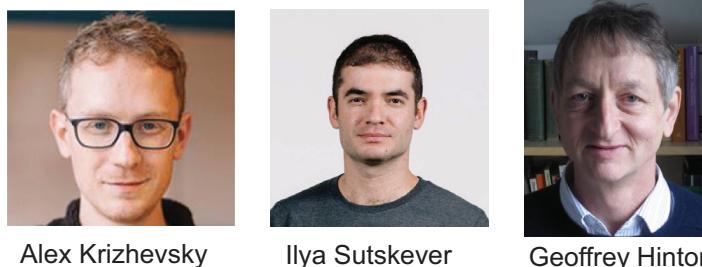
Notice that  $\nabla J(w)$  is simple to calculate.

**AI boomed in the 1960s but ...** As we verified above, logistic regression is the best binary classifier in a sense of maximizing the likelihood of training data, assuming that the overall architecture is based on Perceptron. But as you can see, the Perceptron architecture is somewhat restricted. So you may guess that the performance of logistic regression based on the restricted architecture may not be that good in many applications. It turns out this is the case. Actually it was already verified in 1969 by two pioneers in the AI field, named Marvin Minsky and Seymour Papert; see Fig. 3.9 for their portraits.



Marvin Minsky Seymour Papert '69

**Figure 3.9.** Two AI pioneers, Minsky & Papert, demonstrated limitations of the architecture of Perceptron in a book titled “Perceptrons”. Unfortunately, this led to a very long depression period in the AI field, called the *AI winter*.



**Figure 3.10.** A giant in the AI field, Geoffrey Hinton, together with his PhD students, Alex Krizhevsky and Ilya Sutskever, developed a deep neural network, named AlexNet, intended for image classification (Krizhevsky *et al.*, 2012). AlexNet achieved almost *human-level recognition performances*, which were never attained earlier. This won them the ImageNet competition in 2012. More importantly, this recognition anchored the start of *deep learning revolution*.

They published a book, titled “Perceptrons” (Minsky and Papert, 1969), where they demonstrated that the Perceptron architecture cannot implement even very simple functions, like XOR. Their results made many people at that time disappointed. This finally led to a very long depression period of the AI field, called the AI winter.

**AI revived in 2012** The AI winter had continued until recently. However, a big event happened in 2012, enabling the AI field to revive. The big event was the winning of ImageNet recognition competition by the following three people: Geoffrey Hinton (a very well-known figure in the AI field, known as the Godfather of deep learning) and his two PhD students (Alex Krizhevsky and Ilya Sutskever); see Fig. 3.10.

They built a Perceptron-like neural network but which consists of many layers, called a deep neural network (DNN) (Krizhevsky *et al.*, 2012). They then showed that their DNN, which they named AlexNet, could achieve almost *human-level recognition performances*, which were astonishing at the time. This enabled *deep learning revolution* in the AI field.

**Look ahead** For a couple of next sections, we will investigate deep neural networks (DNNs) and the role of optimization in that context. Specifically we will cover the following four stuffs. First we will study the architecture of DNNs. We will then formulate a corresponding optimization problem. Next, we will explore an efficient way of solving the problem. We will also discuss why DNNs offer great performances yet in a brief manner. Lastly, we will investigate how to implement DNNs using one of the most popular deep learning software frameworks, TensorFlow.

### 3.3 Deep Learning I

**Recap** During the past two sections, we have studied one of application topics of optimization: supervised learning. The goal of supervised learning is to learn a function of a machine using training samples. We considered a specific yet brain-inspired architecture for the function structure: *Perceptron*; see Fig. 3.11. Taking the logistic function together with cross entropy loss, we obtained logistic regression. We then proved that logistic regression is *optimal* in a sense of maximizing the likelihood of training samples.

At the end of the last section, we mentioned briefly about how the AI field has evolved with research on *neural networks*. While one of the first neural networks, Perceptron, enabled the AI revolution in the 1960s, the boom ended shortly after publication of a book by Minsky & Papert, titled “*Perceptrons*”. The book criticized limitations of the Perceptron architecture, and this freezed the passion of many people working on the AI field, thus leading to the AI winter.

The AI field boomed again in 2012 by Hinton & his group members. They developed a neural network (which they called AlexNet) that demonstrated human-level performances of image recognition, thus making people excited about the field again. AlexNet is based on a *deep neural network* architecture (DNN for short).

**Outline** In this section, we are going to start investigating deep neural networks (DNNs). Specifically we will cover the following four stuffs. First, we will study what DNNs mean and the network architecture. We will then investigate how DNNs were proposed, i.e., who the inventor was, as well as, what the motivation was. Next, we will discuss why DNNs were recently appreciated, in other words, why they were not appreciated during the AI winter. Finally we will formulate a

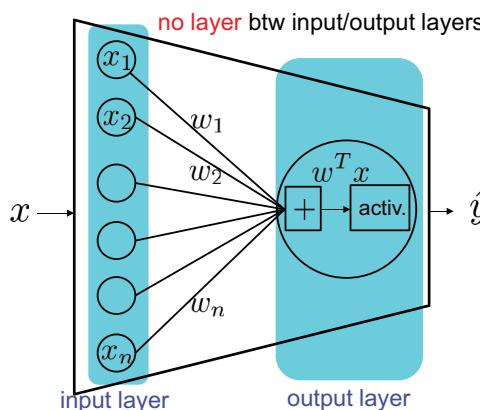


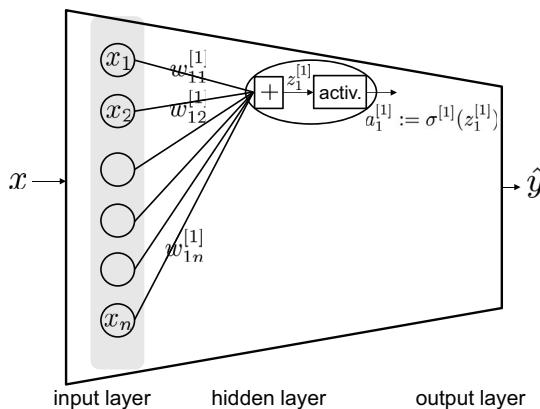
Figure 3.11. Input and output layers in Perceptron.

corresponding optimization problem to start talking about connection to optimization of this book's focus.

**Terminologies** Recall the Perceptron architecture in Fig. 3.11. Before defining the deep neural network (DNN), we need to introduce a couple of terminologies. The first is the *input layer*. We say that a collection of neurons which take input  $x$  is the input layer. Similarly, the *output layer* is defined as a collection of the output neuron(s). A *shallow* neural network is defined as a network which consists of only input and output layers, i.e., there is no intermediate layer between the two, like Perceptron in Fig. 3.11.

**Definition of deep neural networks (DNNs)** We say that a neural network is *deep* if it has at least one intermediate layer between input and output layers. Such in-between-placed layer is called a *hidden layer*. So a *deep neural network* is defined as a network which contains hidden layer(s).

**Two-layer DNN architecture** Here are details on how the DNN looks like. For illustrative purpose, let us explain the architecture with a simple setting in which there is only one hidden layer, named a 2-layer neural network in the field<sup>3</sup>; also see Fig. 3.12.



**Figure 3.12.** The operation at a neuron in a hidden layer is exactly the same as that in Perceptron.

3. Someone may argue that this is a *3-layer* neural network as it has input/hidden/output layers. But the convention in the deep learning field, adopted by many of the pioneers in the field, is that the number of layers is counted as the total number of layers minus 1. The reason is that the input layer is not counted in computing the total number, as it does not have an activation function. Personally this way of defining a network is confusing. Nonetheless, we will adopt this convention as it has already been so widely used.

Let us consider an operation at the first neuron in the hidden layer. The operation is exactly the same as the operation that we saw earlier in Perceptron. First it takes a linear combination to yield:

$$z_1^{[1]} := w_{11}^{[1]}x_1 + w_{12}^{[1]}x_2 + \cdots + w_{1n}^{[1]}x_n \quad (3.33)$$

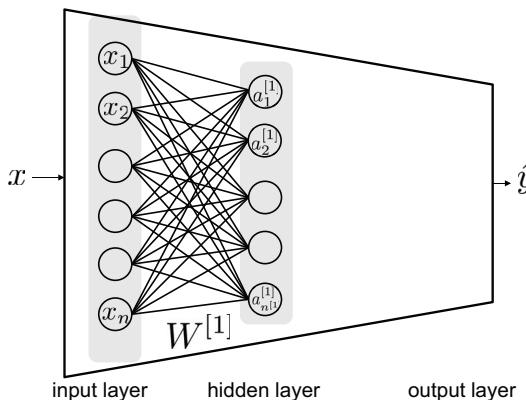
where  $w_{1j}^{[1]}$  indicates a weight associated with  $x_j$  and the 1st neuron in the (1st) hidden layer. Here the upper-script  $(\cdot)^{[1]}$  denotes the index of hidden layers. In general, a bias term, say  $b_1^{[1]}$ , can be added into (3.33). But for illustrative simplicity, we will drop all the bias terms throughout. Next the linear combination is passed onto an activation function, so we get:

$$a_1^{[1]} := \sigma^{[1]}(z_1^{[1]}) \quad (3.34)$$

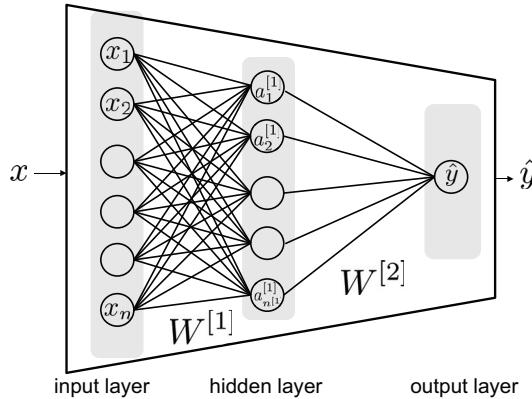
where  $\sigma^{[1]}(\cdot)$  indicates an activation function employed in the 1st hidden layer. Usually we are allowed to use different activation functions across different layers, while the same activation function applies within the same layer by convention. Applying the same operation to the other neurons in the hidden layer, we obtain a picture like the one in Fig. 3.13.

For notational simplicity, we introduce a matrix, say  $W^{[1]}$ , which aggregates all the weight components associated with the input layer and the hidden layer:

$$W^{[1]} := \begin{bmatrix} w_{11}^{[1]} & w_{12}^{[1]} & \cdots & w_{1n}^{[1]} \\ w_{21}^{[1]} & w_{22}^{[1]} & \cdots & w_{2n}^{[1]} \\ \vdots & \vdots & \ddots & \vdots \\ w_{n^{[1]}1}^{[1]} & w_{n^{[1]}2}^{[1]} & \cdots & w_{n^{[1]}n}^{[1]} \end{bmatrix} \in \mathbf{R}^{n^{[1]} \times n} \quad (3.35)$$



**Figure 3.13.** The architecture of a hidden layer in a 2-layer DNN.



**Figure 3.14.** The architecture of a 2-layer DNN.

where \$n^{[1]}\$ denotes the number of neurons in the (1st) hidden layer. Using this matrix notation, we can then represent the output of the hidden layer as:

$$a^{[1]} = \sigma^{[1]}(W^{[1]}x) \in \mathbf{R}^{n^{[1]}} \quad (3.36)$$

where \$\sigma^{[1]}(\cdot)\$ indicates a *component-wise* function.

Applying the same operation into the one between the hidden and output layers, we obtain a picture like the one in Fig. 3.14. Using another matrix notation, say \$W^{[2]} \in \mathbf{R}^{1 \times n^{[1]}}\$, we can represent the output in the output layer as:

$$\hat{y} = \sigma^{[2]}(W^{[2]}a^{[1]}) \in \mathbf{R} \quad (3.37)$$

where \$\sigma^{[2]}\$ indicates an activation function at the output layer, which can possibly be different from \$\sigma^{[1]}(\cdot)\$, as mentioned earlier.

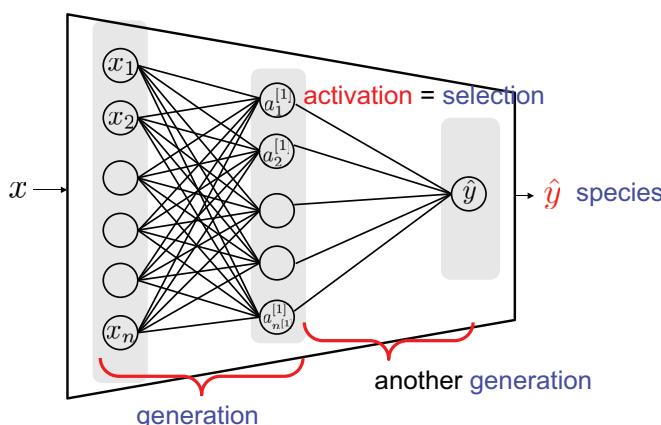
**General DNN architecture** In general, an \$(L + 1)\$-layer DNN (an \$L\$-hidden-layer DNN) can be expressed as:

$$\begin{aligned} a^{[1]} &= \sigma^{[1]}(W^{[1]}x) \in \mathbf{R}^{n^{[1]}}, \\ a^{[2]} &= \sigma^{[2]}(W^{[2]}a^{[1]}) \in \mathbf{R}^{n^{[2]}}, \\ &\vdots \\ a^{[L]} &= \sigma^{[L]}(W^{[L]}a^{[L-1]}) \in \mathbf{R}^{n^{[L]}}, \\ \hat{y} &= \sigma^{[L+1]}(W^{[L+1]}a^{[L]}) \in \mathbf{R} \end{aligned} \quad (3.38)$$

where  $a^{[i]} \in \mathbf{R}^{n^{[i]}}$  indicates the output of the  $i$ th hidden layer;  $\sigma^{[i]}(\cdot)$  denotes the component-wise activation function at the  $i$ th hidden layer; and  $W^{[i]} \in \mathbf{R}^{n^{[i]} \times n^{[i-1]}}$  denotes the weight matrix associated with the  $i$ th hidden layer and  $(i-1)$ th hidden layer. For notational consistency, one can define the 0th hidden layer as the input layer;  $(L+1)$ th hidden layer as the output layer, and hence,  $n^{[0]} := n$  and  $n^{[L+1]} := 1$ .

**How DNNs were proposed** We explain how the DNN expressed in (3.38) was developed. The first DNN was proposed in 1965 by an Ukrainian mathematician, named Alexey Ivakhnenko ([Ivakhnenko, 1971](#)); see the first left picture in Fig. 3.16. He noticed that the Perceptron architecture is too simple to represent a somewhat complex system. So he believed that a proper architecture should incorporate much more neurons as well as capture much higher connectivities across neurons. Obviously the most complex structure is the one in which each neuron is connected with all of the other neurons. But it was not that clear to him as to whether such complex structure is indeed the case in biological networks for brains of intelligent beings.

He was trying to gain some insights into this from another field: *evolution* in biology. In particular, he was inspired by *genetic natural selection in evolution*. What he was inspired is that a *complex species* is a consequence of evolution through many *generations* by *natural genetic selection*. He then made an analogy between such evolution process and the process of a complex system (machine) of interest. Specifically he came up with operations/entities in a complex system which correspond to *species*, *generation* and *selection* that appear in the evolution process. See Fig. 3.15.



**Figure 3.15.** Analogy between the evolution process (by genetic natural selection in biology) and the process of a complex system (machine).

First of all, the *species* was mapped to the output  $\hat{y}$  in an interested system. The *generation* was interpreted as the process that occurs in between two consecutive layers in the system. So the process with two generations yields a 2-layer neural network. Lastly the *selection* was captured by the *activation* process in the system. These analogies naturally led to the DNN architecture illustrated in Fig. 3.15.

**Some pioneering efforts on DNNs** Initially, the DNN architecture in Fig. 3.15 was investigated in depth by only a few people in the field. One of the reasons was that there was no theoretical basis which supports that the architecture can represent any arbitrary complex functional of a system. The architecture was based solely on the hypothesis. There was no proof. Even worse, it was not that simple to do *experimental verification* because the technology of the day was so immature that the time required to train a DNN was very long from days to weeks. Nonetheless, there were some people who studied this architecture in depth. Here we list three of them below.

The first is obviously the inventor of the DNN architecture: Alexey Ivakhnenko. One of his great achievements in this field was to propose a 7-hidden-layer DNN in 1971. The second pioneer is a Japanese computer scientist, named Kunihiko Fukushima; see the middle picture in Fig. 3.16. He developed a specially-structured DNN intended for *pattern recognition* in computer vision in 1980 ([Fukushima, 1980](#)). That was actually the first *convolutional neural network* (CNN), which is now known as the most famous and widely-used DNN in the computer vision field. The third pioneer is a French computer scientist, which is now very famous in the deep learning field and also a winner of the 2018 Turing Award (considered as the Nobel Prize in computer science). In 1989, he trained a CNN for the purpose of recognizing handwritten ZIP codes on mails ([LeCun et al., 1989](#)). This development played a role to vitalize the deep learning field because the trained CNN worked very well and so was commercialized. Nonetheless, it was not enough to enable the deep learning revolution. One of the reasons was that the training time required 3 days with the technology of the day.



Alexey Ivakhnenko



Kunihiko Fukushima



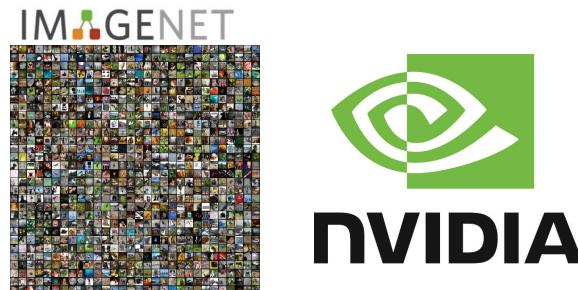
Yann LeCun

**Figure 3.16.** Three deep learning pioneers in early days.

**Not appreciated much in early days** There were more critical reasons as to why the DNN architecture was not appreciated much at that time. These are two folded. The first reason is concerning *performances*. The DNN-based algorithms at the time were easily outperformed by much simpler approaches. One of the simpler approaches was Support Vector Machines (SVMs), which is sort of a variant of the margin-based LP classifier that we learned in Part I. Remember that the margin-based LP classifier is very simple and runs very fast. On the other hand, the DNN architecture was relatively much more complex, yet even worse, the performance was not better. The second reason is about model complexity. The DNN model was so complex considering the technology of the day, so it required very long training time, from days to weeks.

**Why appreciated nowadays?** But as many of you know, the DNN is greatly appreciated nowadays. Why is that? As mentioned earlier, this is mainly due to the recent big event by Hinton<sup>4</sup> and his PhD students who demonstrated that DNNs can achieve human-level recognition performances. Then, how that happened? There are two technology breakthroughs that enabled the deep learning revolution; see Fig. 3.17.

The first breakthrough is the *advent of big data*. Nowadays we are living in the big data era. There are tons of data that are floating in the cyber-world. So it is possible to gather lots of training data. One such huge dataset gathered for the purpose of image cognition was ImageNet ([Russakovsky et al., 2015](#)). The dataset was created in 2009 by a computer-vision team at Stanford, led by Professor Fei-Fei Li. It turned out this dataset played a crucial role for Hinton's team to demonstrate the power of DNN by offering a sufficiently large number of training samples enough to learn a complex model.



**Figure 3.17.** Two technology breakthroughs that enabled the deep learning revolution.

4. Geoffrey Hinton is also a co-winner of the 2018 Turing Award with Yann LeCun and another giant in the field, named Yoshua Bengio.

The second breakthrough is the supply of very fast and not-so-expensive Graphic Processing Units (GPUs). The major company that provided such GPUs is NVIDIA. GPUs offered great computational power to reduce training time of DNNs significantly.

**An optimization problem** Now let us connect the DNN architecture to optimization of this book's interest. For illustrative purpose, let us formulate an optimization problem for the simple two-layer DNN in Fig. 3.14. The optimization problem based on the DNN reads:

$$\min_w \sum_{i=1}^m \frac{1}{m} \ell(y^{(i)}, \hat{y}^{(i)}) \quad (3.39)$$

where:

$$\begin{aligned}\hat{y}^{(i)} &= \sigma^{[2]} \left( W^{[2]} a^{[1],(i)} \right), \\ a^{[1],(i)} &= \sigma^{[1]} \left( W^{[1]} x^{(i)} \right), \\ w &= (W^{[1]}, W^{[2]}).\end{aligned}$$

**Optimal loss function** Suppose we use the logistic function for

$$\sigma^{[2]}(z) = \sigma(z) := \frac{1}{1 + e^{-z}}.$$

Then, using exactly the same argument that we made in Section 3.2, one can show that the optimal loss is *cross entropy loss*:

$$\ell^*(y, \hat{y}) = \ell_{\text{CE}}(y, \hat{y}) = -y \log y - (1 - y) \log(1 - \hat{y}). \quad (3.40)$$

**Is it convex?** Pugging cross entropy loss into (3.39), we obtain:

$$\arg \min_w \frac{1}{m} \sum_{i=1}^m -y^{(i)} \log \hat{y}^{(i)} - (1 - y^{(i)}) \log(1 - \hat{y}^{(i)}) \quad (3.41)$$

where  $\hat{y}^{(i)} = \sigma(W^{[2]} a^{[1],(i)})$ . Now a natural question arises. How to solve the problem? We are familiar with solving only *convex* optimization problems. So the question of our interest is: Is the objective function *convex*? Obviously it depends on how to choose an activation function in the hidden layer:  $\sigma^{[1]}(\cdot)$ .

**Look ahead** There is a very well-known and powerful activation function for  $\sigma^{[1]}(\cdot)$ . Unfortunately, under the choice of the function, the optimization problem (3.41) was shown to be *non-convex*. But there is a good news. That is, we have a way to handle such a non-convex problem. In the next section, we will study details on the way.

## 3.4 Deep Learning II

---

**Recap** In the previous section, we learned about the architecture of DNNs which have been shown to be quite powerful in recent years. We also discussed a brief history of DNNs; who the inventor was; what the motivation was; why they were not appreciated until recently; and what led to the deep learning revolution. We then formulated an optimization problem for DNNs to start talking about connection to optimization topics of this book's interest. Here is the optimization problem intended for a 2-layer DNN:

$$\arg \min_{w=(W^{[1]}, W^{[2]})} \frac{1}{m} \sum_{i=1}^m -y^{(i)} \log \hat{y}^{(i)} - (1 - y^{(i)}) \log(1 - \hat{y}^{(i)}) \quad (3.42)$$

where

$$\hat{y}^{(i)} = \sigma\left(W^{[2]} a^{[1],(i)}\right); \quad a^{[1],(i)} = \sigma^{[1]}\left(W^{[1]} x^{(i)}\right).$$

Here  $\sigma(\cdot)$  indicates the component-wise logistic function defined as:

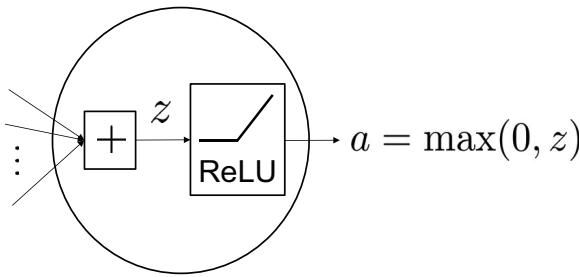
$$\sigma(z) := \frac{1}{1 + e^{-z}}. \quad (3.43)$$

On the other hand,  $\sigma^{[1]}(\cdot)$  denotes a possibly-different activation function used at the hidden layer.

At the end of the last section, we claimed that for a widely-used  $\sigma^{[1]}(\cdot)$ , the objective function in (3.42) is *non-convex*, which is intractable in general. We also claimed that there is a proper way to address such a non-convex optimization problem.

**Outline** In this section, we are going to support these claims. Specifically what we are going to do are four folded. First of all, we will study what the widely-used activation function is. We will then check that the objective function is not convex. Next we will investigate what the proper way is. Finally we will discuss how to solve the optimization problem in detail.

**Widely-used activation function** Consider an operation that occurs at one neuron in the hidden layer; see Fig. 3.18. As mentioned earlier, the DNN architecture takes the *Perception architecture* as the basic operation unit. So the basic operation consists of two procedures. First we perform a linear operation by aggregating weighted signals coming from neurons in the preceding layer, thus yielding an output, say  $z$ . The output  $z$  is then passed onto an activation function, so we get an output, say  $a$ .



**Figure 3.18.** Rectified Linear Unit (ReLU).

The activation function which has been widely used during recent years is a function named the *Rectified Linear Unit*, simply called ReLU. The functional is very simple; it simply bypasses the input if it is non-negative; yields 0 otherwise:

$$a = \begin{cases} z, & \text{if } z \geq 0; \\ 0, & \text{if } z < 0. \end{cases} \quad (3.44)$$

So it can be represented as:

$$a = \max(0, z). \quad (3.45)$$

**A brief history of ReLU** In fact, the ReLU function was introduced in very early days in a variety of fields, not limited to the deep learning field. It appeared even in Fukushima's 1980 paper on convolutional neural networks (CNNs) ([Fukushima, 1980](#)).

But the function was not frequently used in DNNs until recently. Instead more interpretable activation functions like the logistic function were widely used. Another popular activation function was a shifted version of the logistic function, called the  $\tanh$  function:

$$\begin{aligned} \tanh(z) &:= \frac{e^z - e^{-z}}{e^z + e^{-z}} \\ &= \frac{1}{1 + e^{-2z}} - \frac{e^{-2z}}{1 + e^{-2z}} \\ &= \sigma(2z) - (1 - \sigma(2z)) \\ &= 2\sigma(2z) - 1. \end{aligned} \quad (3.46)$$

Note that  $\tanh(z) = 2\sigma(2z) - 1$ , so the range of the function is shifted from  $(0, 1)$  to  $(-1, 1)$ .

A common rule of thumb that had been applied in the deep learning field until recently was to use the logistic function only at the output layer while taking the



Yoshua Bengio



Seppo Linnainmaa

**Figure 3.19.** Yoshua Bengio (left) is one of the giants in the deep learning field, and also a co-recipient of the 2018 Turing Award. One of his main achievements is to demonstrate the power of ReLU activation function, thus popularizing the use of the function in the deep learning society. Seppo Linnainmaa (right) is the inventor of *backpropagation* which serves as an efficient way of computing gradients in DNNs.

tanh function at all of the other neurons placed in hidden layers. Many empirical results have demonstrated that the rule of thumb always yields a better or equal performance, as compared to the other alternative which takes the logistic function at all places. There is no theoretical justification on this. But it looks more or less making-sense. The reason is that taking the tanh function at hidden neurons broadens the output range, thus yielding a more degree of freedom relative to the one by the logistic function.

**ReLU became prevalent since 2011** A recent big wave arose in the domain of activation functions. In 2011, one of the deep learning heroes, named Yoshua Bengio (see the left picture in Fig. 3.19), together with his group members, Xavier Glorot (PhD student) and Antoine Bordes (postdoc), demonstrated via extensive simulation results that ReLU enables *faster and more effective training* of DNNs, compared to the logistic and/or tanh functions (Glorot *et al.*, 2011). This also was empirically confirmed by numerous practitioners on many datasets. Hence, ReLU now acts as a default activation function in hidden layers.

They also provided some intuitions as to why that is the case. One intuition is that ReLU better mimicks how brains of intelligent beings work. A report by neuroscientists says that *only a few percentages* of the neurons in human brains are *activated* even during active brain activities. This is somewhat consistent with a consequence of taking ReLU, as that way leads many of the neurons to be simply set to 0 when their values take negative.

Another explanation concerns a technical operation, being tailored for a particular yet popular learning algorithm. One of the popular training algorithms employed in the field is based on computation of *gradients* of the objective function w.r.t. weights (optimization variables). Dynamics of gradients for the logistic or

tanh functions are somewhat limited. They are close to 0 for a large or small value of  $z$ . Why? Think about the shape of the function. It takes somewhat a meaningful gradient only when  $z$  is in a narrow range. On the other hand, the gradient of ReLU does not vanish even when  $z$  is very large. Notice that the gradient of ReLU reads:

$$\frac{d\text{ReLU}(z)}{dz} = \begin{cases} 1, & \text{if } z > 0; \\ 0, & \text{if } z < 0; \\ \text{undefined,} & \text{if } z = 0. \end{cases} \quad (3.47)$$

Note that the gradient takes 1 even when  $z$  is very large. So they believed that this *non-vanishing gradient effect* yields a better training.

**Remark on ReLU** As you may see from (3.47), there is an issue in computing the gradient of ReLU. The issue is that the function is *not differentiable*. The gradient is undefined at  $z = 0$ . But this is not a big deal in practice. In reality, the event  $z = 0$  rarely happens. Actually the exactly-zero-event never happened. It has a *measure-zero-event*. So there is no problem to use in practice, although it is mathematically problematic.

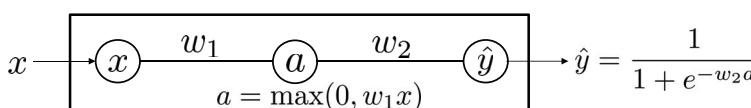
**Convex vs. non-convex?** Let us go back to the optimization problem. When taking the ReLU activation function, we obtain:

$$\arg \min_{w=(W^{[1]}, W^{[2]})} \frac{1}{m} \sum_{i=1}^m -y^{(i)} \log \hat{y}^{(i)} - (1 - y^{(i)}) \log(1 - \hat{y}^{(i)}) \quad (3.48)$$

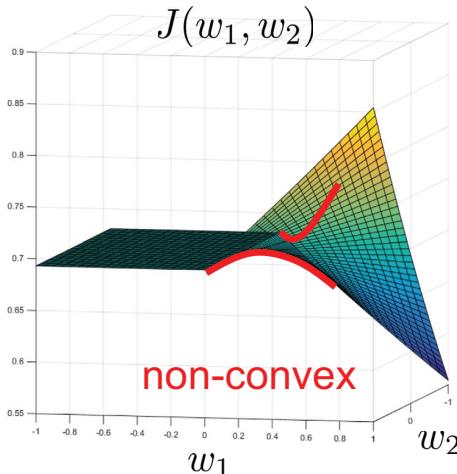
where

$$\hat{y}^{(i)} = \sigma \left( W^{[2]} \max(0, W^{[1]} x^{(i)}) \right).$$

The question of this book's interest is: Is the objective function *convex*? As claimed earlier, the answer is no. The objective function is *non-convex* in general. The proof of this is a bit involved, as it may include a complicated Hessian calculation. So this may distract us from the main stream of the contents of this section. So we omit the proof here. But you will have a chance to prove this in Prob 8.5 for a simple setting, illustrated in Fig. 3.20.



**Figure 3.20.** A simple 2-layer DNN in which an associated objective function is non-convex.



**Figure 3.21.** Shape of the objective function of the simple 2-layer DNN in Fig. 3.20.

If you are not interested in the proof, you may want to be convinced about non-convexity from the following numerical plot in Fig. 3.21. Notice in the figure that the objective function in this simple setting is indeed non-convex.

**A way to handle such a non-convex problem** As mentioned earlier, there is a proper way to handle such a non-convex optimization problem. The way is inspired by the observation made by numerous practitioners working on the field. Many *experimental* results by them revealed that in most cases:

$$\text{Any local minimum is the global minimum.} \quad (3.49)$$

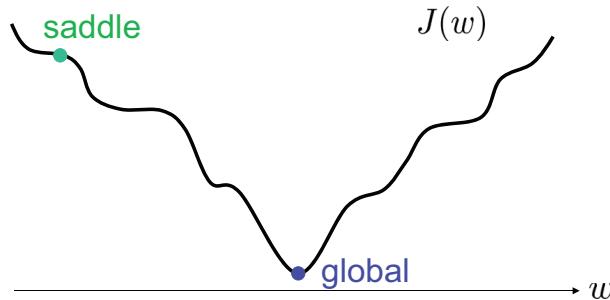
What this means is that in many of the practical settings, there is *no spurious local minimum*. As you may conjecture, this is not a mathematically correct statement. Actually this was proven to be *mathematically wrong*, meaning that there are counter-examples in which there are *spurious local minima*. But it was also empirically shown that those counter-examples rarely happen in many of the working DNNs. In fact, we are still very much lacking in our understanding on this. In other words, currently we have no idea what is the necessary/sufficient condition for (3.49) to hold.

Nonetheless, in many realistic scenarios, (3.49) was observed. So many people believe that in most interested cases, the landscape of the objective function for an DNN-based optimization problem looks like the one in Fig. 3.22.

Note in Fig. 3.22 that there is no spurious local minimum. We have only the global minimum or saddle points.<sup>5</sup> This observation made through many

---

5. A saddle point is defined as a point in which its derivative is zero while having neither a maximum nor a minimum value.



**Figure 3.22.** Landscape of the objective function in DNN optimization.

experimental results suggested a good guideline in practice. That is, simply to find any minimum and then take it as a solution, no matter what the type of an optimization problem is. The question of interest is then: How to find a minimum? One very popular way is to apply *gradient descent* which we are familiar with. Of course, the gradient descent algorithm may lead us to get stuck in some saddle point which we do not want to arrive at. But the good news in practice is that it is extremely rare to get stuck in a saddle point when there are minima. Actually it is even difficult to arrive at a saddle point even if we wish to do so. Hence, a general rule of thumb is to simply apply gradient descent no matter what.

**Gradient descent** What is the gradient descent algorithm in the interested optimization problem below?

$$\arg \min_{w=(W^{[1]}, W^{[2]})} \underbrace{\frac{1}{m} \sum_{i=1}^m -y^{(i)} \log \hat{y}^{(i)} - (1 - y^{(i)}) \log(1 - \hat{y}^{(i)})}_{=:J(w)}. \quad (3.50)$$

Here  $\hat{y}^{(i)} = \sigma(W^{[2]} \max(0, W^{[1]}x^{(i)}))$ .

The algorithm is to iterate the following procedure: for the  $t$ th iteration, the new  $(t+1)$ th estimate for weights takes:

$$w^{(t+1)} \leftarrow w^{(t)} - \alpha^{(t)} \nabla_w J(w^{(t)})$$

where  $\alpha^{(t)}$  indicates the learning rate. Since  $w$  is a collection of  $(W^{[1]}, W^{[2]})$ , the detailed procedure is:

$$W^{[2],(t+1)} \leftarrow W^{[2],(t)} - \alpha^{(t)} \nabla_{W^{[2]}} J(w^{(t)});$$

$$W^{[1],(t+1)} \leftarrow W^{[1],(t)} - \alpha^{(t)} \nabla_{W^{[1]}} J(w^{(t)}).$$

As you can see here, there are multiple weight update procedures – in this case, two procedures. Actually these multiple procedures yielded a critical concern in early days of the DNN research. The reason is that it requires computationally heavy calculations. Especially when an DNN has many layers, it raises a critical computational concern. So some people tried to address this problem in early days to come up with an efficient way of computing such many gradients, called:

### *Backpropagation.*

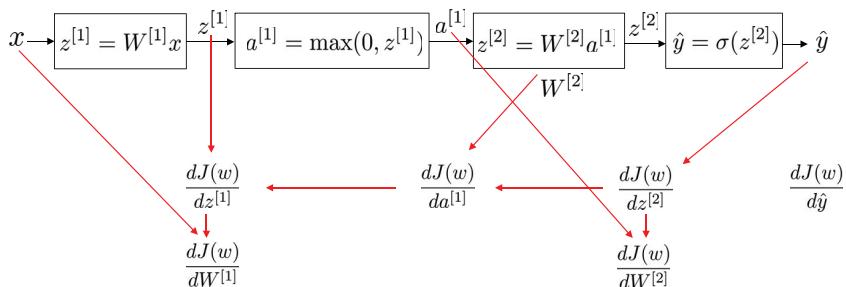
**Backpropagation** The same backpropagation method was independently developed by a bunch of research groups including Hinton's development in 1986 together with his colleagues, David Rumelhart & Ronald Williams ([Rumelhart et al., 1986](#)). But the first invention was earlier. It was around when the book of *Perceptrons* was published – that was 1970. In 1970, a Finnish mathematician (as well as a computer scientist), named Seppo Linnainmaa (see the right picture in Fig. 3.19), invented the method ([Linnainmaa, 1970](#)).

The idea is very natural although it involves some complicatedly-looking math equations. Perhaps this may be one of the reasons that there were several independent yet same inventions. The idea is to:

Successively compute gradients in a *backward* manner  
by using a *chain rule* for derivatives.

For illustrative purpose, let us first explain how it works for a simple single-example setting ( $m = 1$ ). We will then extend it to the general case.

**Backpropagation in action:  $m = 1$**  The illustration of the method can be streamlined with the help of some picture which visualizes paths of signals. One such path is the *forward* path; see the top row in Fig. 3.23. The input signal  $x$  passes through the hidden layer to yield  $z^{[1]}$  and then  $a^{[1]}$ . Similarly we get  $z^{[2]}$  and then  $\hat{y} := a^{[2]}$ .



**Figure 3.23.** Illustration of *backpropagation*:  $m = 1$ .

The *backpropagation* starts from *backward*. Consider the gradient of the objective function  $J(w)$  w.r.t. the *last* output signal  $\hat{y}$ :  $\frac{dJ(w)}{d\hat{y}}$ . To ease illustration, we will use the notation of  $\frac{d}{d\hat{y}}$  instead of  $\nabla_{\hat{y}}$ . The reason is that backpropagation is based on the *chain rule* for derivatives, so the notation  $\frac{d}{d\hat{y}}$  helps us to better understand how it works, relative to  $\nabla_{\hat{y}}$ . Since  $J(w)$  is simply cross entropy loss for  $m = 1$ , the gradient reads:

$$\frac{dJ(w)}{d\hat{y}} = -\frac{y}{\hat{y}} + \frac{1-y}{1-\hat{y}}. \quad (3.51)$$

One can easily compute this, since  $y$  is given in the problem and  $\hat{y}$  is available once we compute the forward path.

Next, we consider the gradient of  $J(w)$  w.r.t. the *second last* signal  $z^{[2]}$ :  $\frac{dJ(w)}{dz^{[2]}}$ . This is where the idea of the *chain rule* kicks in. Using the chain rule, we get:

$$\begin{aligned} \frac{dJ(w)}{dz^{[2]}} &= \frac{dJ(w)}{d\hat{y}} \frac{d\hat{y}}{dz^{[2]}} \\ &\stackrel{(a)}{=} \left( -\frac{y}{\hat{y}} + \frac{1-y}{1-\hat{y}} \right) \hat{y}(1-\hat{y}) \\ &= \hat{y} - y \end{aligned} \quad (3.52)$$

where (a) follows from (3.51) (already computed earlier) as well as the fact that the gradient of the logistic function can simply be expressed as:

$$\begin{aligned} \frac{d\sigma(z)}{dz} &= \frac{d}{dz} \left( \frac{1}{1+e^{-z}} \right) = \frac{e^{-z}}{(1+e^{-z})^2} \\ &= \frac{1}{1+e^{-z}} \cdot \frac{e^{-z}}{1+e^{-z}} \\ &= \sigma(z)(1-\sigma(z)). \end{aligned} \quad (3.53)$$

From (3.52), we can compute one of the interested gradients:  $\frac{dJ(w)}{dW^{[2]}}$ . Again using the chain rule, we get:

$$\begin{aligned} \frac{dJ(w)}{dW^{[2]}} &= \frac{dJ(w)}{dz^{[2]}} \frac{dz^{[2]}}{dW^{[2]}} \\ &\stackrel{(a)}{=} \frac{dJ(w)}{dz^{[2]}} a^{[1]T} \end{aligned} \quad (3.54)$$

where (a) comes from  $z^{[2]} = W^{[2]}a^{[1]}$  (why taking a transpose to get  $a^{[1]T}$ ?). Notice that this can be computed from  $\frac{dJ(w)}{dz^{[2]}}$  (already computed from (3.52)) and

the knowledge of  $a^{[1]}$ . To indicate how it can be computed, we draw red-lined flows in Fig. 3.23.

Now you may grab the idea of how backpropagation works. We compute gradients w.r.t. from the last output signal ( $j$ ) to the inner signals ( $z^{[2]}, W^{[2]}$ ), all the way *back* to ( $z^{[1]}, W^{[1]}$ ). For those who did not get this yet, let us repeat.

We next consider the gradient of  $J(w)$  w.r.t. the *third last* signal  $a^{[1]}$ :  $\frac{dJ(w)}{da^{[1]}}$ . Again using the chain rule, we obtain:

$$\begin{aligned} \frac{dJ(w)}{da^{[1]}} &= \frac{dJ(w)}{dz^{[2]}} \frac{dz^{[2]}}{da^{[1]}} \\ &\stackrel{(a)}{=} W^{[2]T} \frac{dJ(w)}{dz^{[2]}} \end{aligned} \quad (3.55)$$

where (a) is due to  $z^{[2]} = W^{[2]}a^{[1]}$ . Here you may be very confused about how the last equality comes up. Why do we take a transpose for  $W^{[2]}$ ? Why do we first have  $W^{[2]T}$ , followed by  $\frac{dJ(w)}{dz^{[2]}}$ ? Why not the other way around? There is a rule of thumb for this computation. First of all, you need to check what the dimension of the final result is. In this case, the final result is  $\frac{dJ(w)}{da^{[1]}}$ . The dimension should be exactly the same as that of  $a^{[1]}$ , so  $\frac{dJ(w)}{da^{[1]}} \in \mathbf{R}^{n^{[1]}}$ . Next think about the dimension of  $\frac{dJ(w)}{dz^{[2]}}$ . It should be  $\frac{dJ(w)}{dz^{[2]}} \in \mathbf{R}^{n^{[2]}}$ . Why? This suggests that  $\frac{dJ(w)}{dz^{[2]}}$  should come *after*  $W^{[2]T}$ . Otherwise, dimensions do not match – a syntax error occurs. Now why are we taking a transpose for  $W^{[2]}$ ? Again this is due to dimension matching. With the transpose, we can make sure that the dimension of the end result is  $n^{[1]} \times 1$ , which is what we want.

Again the key observation in (3.55) is that  $\frac{dJ(w)}{da^{[1]}}$  can be computed from  $\frac{dJ(w)}{dz^{[2]}}$  (which we already obtained from (3.52)) and the knowledge of  $W^{[2]}$ . See the knowledge path marked with red lines in Fig. 3.23.

We can next do the same thing for  $\frac{dJ(w)}{dz^{[1]}}$  and  $\frac{dJ(w)}{dW^{[1]}}$ . Using the chain rule, we get:

$$\begin{aligned} \frac{dJ(w)}{dz^{[1]}} &= \frac{dJ(w)}{da^{[1]}} \frac{da^{[1]}}{dz^{[1]}} \\ &\stackrel{(a)}{=} \frac{dJ(w)}{da^{[1]}} \cdot \mathbf{*} \mathbf{1}\{z^{[1]} \geq 0\} \end{aligned} \quad (3.56)$$

where (a) follows from (3.47):  $\frac{d\text{ReLU}(z)}{dz} = \mathbf{1}\{z \geq 0\}$ . Actually this is not quite correct mathematically, since the ReLU function is not differentiable at  $z = 0$ . But since it is okay to ignore the rare event in practice, we simply assume that the gradient is 1 at  $z = 0$ . Here the symbol  $\cdot \mathbf{*}$  indicates the component-wise multiplication (MATLAB notation), not the normal multiplication. You can also easily think that

it should be the component-wise multiplication, since otherwise dimensions do not match. Next we get:

$$\begin{aligned} \frac{dJ(w)}{dW^{[1]}} &= \frac{dJ(w)}{dz^{[1]}} \frac{dz^{[1]}}{dW^{[1]}} \\ &\stackrel{(a)}{=} \frac{dJ(w)}{dz^{[1]}} x^T \end{aligned} \quad (3.57)$$

where (a) is due to  $z^{[1]} = W^{[1]}x$ .

Here is a summary of all the important gradients that we derived in a *backward* manner:

$$\frac{dJ(w)}{dz^{[2]}} = \hat{y} - y; \quad (3.58)$$

$$\frac{dJ(w)}{dW^{[2]}} = \frac{dJ(w)}{dz^{[2]}} a^{[1]T}; \quad (3.59)$$

$$\frac{dJ(w)}{da^{[1]}} = W^{[2]T} \frac{dJ(w)}{dz^{[2]}}; \quad (3.60)$$

$$\frac{dJ(w)}{dz^{[1]}} = \frac{dJ(w)}{da^{[1]}} \cdot * \mathbf{1}\{z^{[1]} \geq 0\}; \quad (3.61)$$

$$\frac{dJ(w)}{dW^{[1]}} = \frac{dJ(w)}{dz^{[1]}} x^T. \quad (3.62)$$

To run the gradient descent algorithm, what we need to use are (3.59) and (3.62). But the other gradients are still important because they serve as *bridges* to compute the interested gradients ((3.59) and (3.62)) in the end.

**Backpropagation for general  $m$**  What about for the general  $m$  case? The idea is exactly the same. The only distinction is that we need to incorporate all the examples in computing gradients. It turns out *matrix* notations help us to derive such generalized gradients. Let

$$\begin{aligned} Y &:= [y^{(1)} \quad y^{(2)} \quad \dots \quad y^{(m)}] \in \mathbf{R}^{1 \times m}; \\ \hat{Y} &:= [\hat{y}^{(1)} \quad \hat{y}^{(2)} \quad \dots \quad \hat{y}^{(m)}] \in \mathbf{R}^{1 \times m}; \\ A^{[1]} &:= [a^{[1],(1)} \quad a^{[1],(2)} \quad \dots \quad a^{[1],(m)}] \in \mathbf{R}^{n^{[1]} \times m}; \\ Z^{[1]} &:= [z^{[1],(1)} \quad z^{[1],(2)} \quad \dots \quad z^{[1],(m)}] \in \mathbf{R}^{n^{[1]} \times m}; \\ Z^{[2]} &:= [z^{[2],(1)} \quad z^{[2],(2)} \quad \dots \quad z^{[2],(m)}] \in \mathbf{R}^{n^{[2]} \times m}; \\ X &:= [x^{(1)} \quad x^{(2)} \quad \dots \quad x^{(m)}] \in \mathbf{R}^{n \times m}. \end{aligned} \quad (3.63)$$

Using these matrix notations, one can readily show that the important corresponding gradients for the general  $m$  case read:

$$\frac{dJ(w)}{dZ^{[2]}} = \hat{Y} - Y; \quad (3.64)$$

$$\frac{dJ(w)}{dW^{[2]}} = \frac{dJ(w)}{dZ^{[2]}} A^{[1]T}; \quad (3.65)$$

$$\frac{dJ(w)}{dA^{[1]}} = W^{[2]T} \frac{dJ(w)}{dZ^{[2]}}, \quad (3.66)$$

$$\frac{dJ(w)}{dA^{[1]}} = \frac{dJ(w)}{dA^{[1]}} * \mathbf{1}\{Z^{[1]} \geq 0\}; \quad (3.67)$$

$$\frac{dJ(w)}{dW^{[1]}} = \frac{dJ(w)}{dZ^{[1]}} X^T. \quad (3.68)$$

Note that these are exactly the same as those in the  $m = 1$  case except one thing. That is, we have now all the capital letters.

**Look ahead** So far we have formulated an DNN-based optimization problem for supervised learning, and found that cross entropy serves as a key component in the design of the optimal loss function. We also learned how to solve the problem via a famous and efficient method, called backpropagation. There would be programming implementation of the algorithm. In the next section, we will study such implementation details in the context of a simple classifier via TensorFlow.

## 3.5 Deep Learning: TensorFlow Implementation

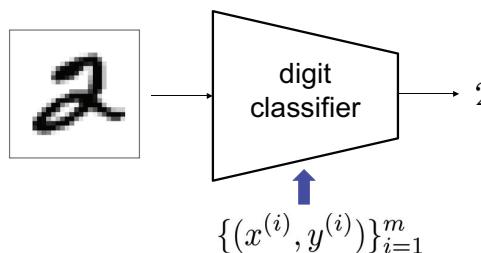
**Recap** We have thus far formulated an DNN-based optimization problem in the context of supervised learning:

$$\min_w \sum_{i=1}^m \ell_{\text{CE}}(y^{(i)}, \hat{y}^{(i)}) \quad (3.69)$$

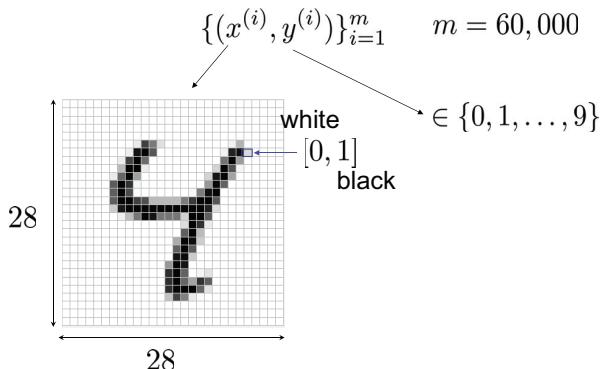
where  $y^{(i)}$  denotes the label of the  $i$ th example;  $\hat{y}$  indicates the predictor output of the DNN (often the output of the logistic function); and  $\ell_{\text{CE}}(y, \hat{y}) = -y \log \hat{y} - (1 - y) \log(1 - \hat{y})$ . We proved that cross entropy loss  $\ell_{\text{CE}}(\cdot, \cdot)$  is the optimal loss function in a sense of maximizing the likelihood. We have also learned that in many of the interested settings, optimization problems for DNNs have no spurious local minima, although the problems are highly non-convex. This motivated the use of gradient descent for such problems. Lastly we studied an efficient way of computing gradients: *backpropagation*, or simply called *backprop*.

**Outline** In this section, we will study how to implement the algorithm via a software tool in the context of a simple classifier. We will first investigate what that simple classifier setting of our focus is. We will then study four implementation details w.r.t. the classifier: (i) dataset that we will use for training and testing; (ii) an DNN model & ReLU activation; (iii) Softmax: a natural extension of a logistic activation for multiple (more than two) classes; and (iv) Adam: an advanced version of gradient descent that is widely used in practice ([Kingma and Ba, 2014](#)). Lastly we will learn how to do programming for the classifier via one prominent deep learning framework: TensorFlow. More specifically, we will employ a higher-level programming language, Keras, which is fully integrated with TensorFlow.

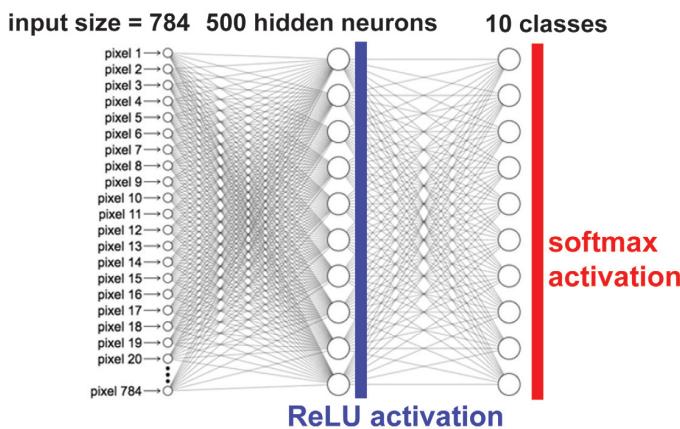
**Handwritten digit classification** The simple classifier that we will focus on for implementation exercise is a handwritten digit classifier wherein the task is to figure out a digit from a handwritten image; see Fig. 3.24. The figure illustrates an instance in which an image of digit 2 is correctly recognized.



**Figure 3.24.** Handwritten digit classification.



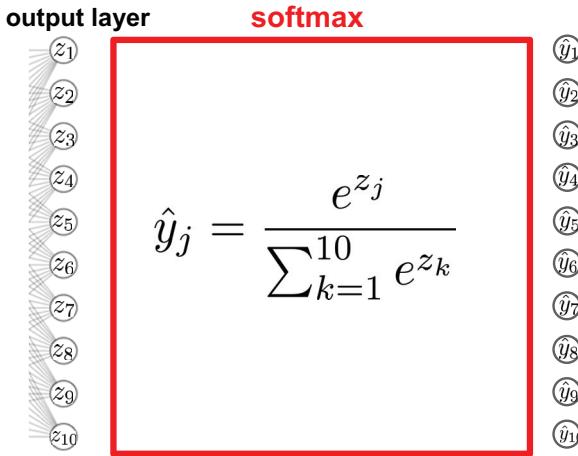
**Figure 3.25.** MNIST dataset: An input image is of 28-by-28 pixels, each indicating an intensity from 0 (white) to 1 (black); and each label with size 1 takes one of the 10 classes from 0 to 9.



**Figure 3.26.** A two-layer fully-connected neural network where input size is  $28 \times 28 = 784$ , the number of hidden neurons is 500 and the number of classes is 10. We employ ReLU activation for the hidden layer, and softmax activation for the output layer; see Fig. 3.27 for details.

For training a model, we employ a popular dataset, named the MNIST (Modified National Institute of Standards and Technology) dataset. It was created by re-mixing the examples from NIST's original dataset. Hence, the naming was suggested. It was prepared by one of the deep learning pioneers, Yann LeCun. It contains  $m = 60,000$  training images and  $m_{\text{test}} = 10,000$  testing images. Each image, say  $x^{(i)}$ , consists of  $28 \times 28$  pixels, each indicating a gray-scale level ranging from 0 (white) to 1 (black). It also comes with a corresponding label, say  $y^{(i)}$ , that takes one of the 10 classes  $y^{(i)} \in \{0, 1, \dots, 9\}$ . See Fig. 3.25.

**A deep neural network model** As a model, we employ a simple two-layer DNN, illustrated in Fig. 3.26. As mentioned earlier, the rule of thumb for the



**Figure 3.27.** Softmax activation for output layer. This is a natural extension of logistic activation intended for the 2-class case.

choice of the activation function in a hidden layer is to use ReLU:  $\text{ReLU}(x) = \max(0, x)$ . So we adopt this here.

**Softmax activation for output layer** So far we have considered *binary* classifiers and hence employed the corresponding activation function in the output layer: logistic function. In our digit classifier, however, there are 10 classes in total. So the logistic function is not directly applicable. One natural extension of the logistic function for a general classifier with more than two classes is to use a generalized version, called softmax. See Fig. 3.27 for its operation.

Let  $z$  be the output of the last layer in a neural network prior to activation:

$$z := [z_1, z_2, \dots, z_c]^T \in \mathbf{R}^c \quad (3.70)$$

where  $c$  denotes the number of classes. The softmax function is then defined as:

$$\hat{y}_j := [\text{softmax}(z)]_j = \frac{e^{z_j}}{\sum_{k=1}^c e^{z_k}} \quad j \in \{1, 2, \dots, c\}. \quad (3.71)$$

Note that this is a natural extension of the logistic function: for  $c = 2$ ,

$$\begin{aligned} \hat{y}_1 := [\text{softmax}(z)]_1 &= \frac{e^{z_1}}{e^{z_1} + e^{z_2}} \\ &= \frac{1}{1 + e^{-(z_1 - z_2)}} \\ &= \sigma(z_1 - z_2) \end{aligned} \quad (3.72)$$

where  $\sigma(\cdot)$  is the logistic function. Viewing  $z_1 - z_2$  as the binary classifier output  $\hat{y}$ , this coincides exactly with the logistic function.

Here  $\hat{y}_i$  can be interpreted as the probability that the  $i$ th example belongs to class  $i$ . Hence, like the binary classifier, one may want to assume:

$$\hat{y}_i = \mathbb{P}(y = [0, \dots, \underbrace{1}_{i\text{th position}}, \dots, 0]^T | x), i \in \{1, \dots, c\}. \quad (3.73)$$

As you may expect, under this assumption, one can verify that the optimal loss function (in a sense of maximizing likelihood) is again cross entropy loss:

$$\ell^*(y, \hat{y}) = \ell_{\text{CE}}(y, \hat{y}) = \sum_{j=1}^c -y_j \log \hat{y}_j$$

where  $y$  indicates a label of one-hot vector type. For instance, in the case of label = 2 with  $c = 10$ ,  $y$  takes:

$$y = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \begin{array}{l} 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \end{array}$$

The proof of this is almost the same as that in the binary classifier. So we will omit the proof. Instead you will have a chance to prove it in Prob 8.2.

Due to the above rationales, the softmax activation has been widely used for many classifiers in the field. Hence, we will also use the conventional function in our digit classifier.

**Adam optimizer** Let us discuss a specific algorithm that we will employ in our setting. As mentioned earlier, we will use an advanced version of gradient descent, called the Adam optimizer. To see how the optimizer operates, let us first recall the vanilla gradient descent:

$$w^{(t+1)} \leftarrow w^{(t)} - \alpha \nabla J(w^{(t)})$$

where  $w^{(t)}$  indicates the estimated weight in the  $t$ th iteration, and  $\alpha$  denotes the learning rate. Notice that the weight update relies only on the *current* gradient, reflected in  $\nabla J(w^{(t)})$ . Hence, in case  $\nabla J(w^{(t)})$  fluctuates too much over iterations, the weight update oscillates significantly, thereby bringing about unstable training.

To address this, people often use a variant algorithm that exploits *past* gradients for the purpose of stabilization. That is, the Adam optimizer.

Here is how Adam works. The weight update takes the following formula instead:

$$w^{(t+1)} = w^{(t)} + \alpha \frac{m^{(t)}}{\sqrt{s^{(t)}} + \epsilon} \quad (3.74)$$

where  $m^{(t)}$  indicates a weighted average of the current and past gradients:

$$m^{(t)} = \frac{1}{1 - \beta_1^t} \left( \beta_1 m^{(t-1)} - (1 - \beta_1) \nabla J(w^{(t)}) \right). \quad (3.75)$$

Here  $\beta_1 \in [0, 1]$  is a hyperparameter that captures the weight of past gradients, and hence it is called the *momentum*. So the notation  $m$  stands for momentum. The factor  $\frac{1}{1 - \beta_1^t}$  is applied in front, in an effort to stabilize training in initial iterations (small  $t$ ). Check the detailed rationale behind this in Prob 8.9.

$s^{(t)}$  is a normalization factor that makes the effect of  $\nabla J(w^{(t)})$  almost constant over  $t$ . In case  $\nabla J(w^{(t)})$  is too big or too small, we may have significantly different scalings in magnitude. Similar to  $m^{(t)}$ ,  $s^{(t)}$  is defined as a weighted average of the current and past values:

$$s^{(t)} = \frac{1}{1 - \beta_2^t} \left( \beta_2 s^{(t-1)} - (1 - \beta_2) (\nabla J(w^{(t)}))^2 \right) \quad (3.76)$$

where  $\beta_2 \in [0, 1]$  denotes another hyperparameter that captures the weight of past values, and  $s$  stands for *square*.

Notice that the dimensions of  $w^{(t)}$ ,  $m^{(t)}$  and  $s^{(t)}$  are the same. So all the operations that appear in the above (including division in (3.74) and square in (3.76)) are *component-wise*. In (3.74),  $\epsilon$  is a tiny value introduced to avoid division by 0 in practice (usually  $10^{-8}$ ).

**TensorFlow: MNIST data loading** Let us study how to do TensorFlow programming for implementing the simple digit classifier that we have discussed so far. First, MNIST data loading. MNIST is a very famous dataset, so it is offered by a sub-library: tensorflow.keras.datasets. Even more, train and test datasets are already therein with a proper split ratio. So we do not need to worry about how to split them. The only script that we should write for importing MNIST is:

```
from tensorflow.keras.datasets import mnist
(X_train, y_train), (X_test, y_test) = mnist.load_data()
X_train = X_train/255.
X_test = X_test/255.
```

Here we divide the input ( $X_{\text{train}}$  or  $X_{\text{test}}$ ) by its maximum value 255 for the purpose of normalization. This procedure is often done as a part of data preprocessing.

**TensorFlow: 2-layer DNN** In order to implement the simple DNN, illustrated in Fig. 3.26, we rely upon two major packages:

- (i) tensorflow.keras.models;
- (ii) tensorflow.keras.layers.

The `models` package contains several functionalities regarding a neural network itself. One major module is `Sequential` which is a neural network entity and hence can be described as a linear stack of layers. The `layers` package includes many elements that constitute a neural network. Examples include fully-connected dense layers and activation functions. These two allow us to readily construct a model illustrated in Fig. 3.26.

```
from tensorflow.keras.models import Sequential  
from tensorflow.keras.layers import Dense, Flatten  
  
model = Sequential()  
model.add(Flatten(input_shape=(28,28)))  
model.add(Dense(500, activation='relu'))  
model.add(Dense(10, activation='softmax'))
```

Here `Flatten` is an entity that indicates a vector expanded from a higher dimensional one, like a 2D matrix. In this example, a digit image of size 28-by-28 is flattened into a vector of size 784( $= 28 \times 28$ ). `add()` is a method for attaching an interested layer to the last part in the sequential model. `Dense` refers to a fully-connected layer. The input size is automatically determined by the last part that it will be attached to in the sequential model. So the only thing to specify is the number of output neurons. In this example, 500 refers to the number of hidden neurons. We can also set an activation function with another argument, like `activation='relu'`. The output layer comes with 10 neurons (coinciding with the number of classes) and softmax activation.

**TensorFlow: Training a model** For training, we need to first set up an algorithm (optimizer) to be employed. Here we use the Adam optimizer. As mentioned earlier, Adam has three key hyperparameters: (i) the learning rate  $\alpha$ ; (ii)  $\beta_1$  (capturing the weight of past gradients); and (iii)  $\beta_2$  (indicating the weight of the square of past gradients). The default choice reads:  $(\alpha, \beta_1, \beta_2) = (0.001, 0.9, 0.999)$ . So these values would be set if nothing is specified.

We also need to specify a loss function. Here we employ the optimal loss function: cross entropy loss. A performance metric that we will look at during training and testing can also be specified. One metric frequently employed is accuracy. One can set all of these via another method compile.

```
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['acc'])
```

Here the option optimizer='adam' sets the default choice of the learning rate and betas. For a manual choice, we first define:

```
opt= tensorflow.keras.optimizers.Adam(
    learning_rate=0.01,
    beta_1 = 0.92,
    beta_2 = 0.992)
```

We then replace the above option with optimizer=opt. The option loss='sparse\_categorical\_crossentropy' means the use of cross entropy loss.

Now we can bring this to train the model on MNIST data. During training, we often employ a part of the entire examples to compute a gradient of a loss function. The part is called *batch*. Two more terminologies. One is the *step* which refers to a loss computation procedure spanning the examples only in a single batch. The other is the *epoch* which refers to the entire procedure associated with all the examples. In our experiment, we use the batch size of 64 and the number 20 of epochs.

```
model.fit(X_train, y_train, batch_size=64, epochs=20)
```

**TensorFlow: Testing the trained model** For testing, we first need to make a prediction from the model output. To this end, we use the predict() function as follows:

```
model.predict(X_test).argmax(1)
```

Here argmax(1) returns the class w.r.t. the highest softmax output among the 10 classes. In order to evaluate the test accuracy, we use the evaluate() function:

```
model.evaluate(X_test, y_test)
```

**Look ahead** This is the end of the supervised learning part. There may be more contents that may be of your interest. But we stop here due to the interest of other topics. Obviously we cannot cover all the contents. If you are interested in more on supervised learning, we recommend you to take many useful *deep learning* courses offered online, e.g., Coursera.

In the next section, we will move onto another application of optimization: *unsupervised learning*. Specifically we will focus on one of popular machine learning frameworks for unsupervised learning, called *Generative Adversarial Networks* (GANs for short) ([Goodfellow et al., 2014](#)). It turns out the duality theorems that we learned in Part II play a crucial role to understand the GANs. We will cover details from the next section onwards.

## Problem Set 8

---

**Prob 8.1 (Logistic regression)** Suppose that  $\{(x^{(i)}, y^{(i)})\}_{i=1}^m$  are independent across all the examples. Show that

$$\mathbb{P}(y^{(1)}, \dots, y^{(m)} | x^{(1)}, \dots, x^{(m)}) = \prod_{i=1}^m \mathbb{P}(y^{(i)} | x^{(i)}). \quad (3.77)$$

In Section 3.2, we proved under the Perceptron architecture that logistic regression is optimal in a sense of maximizing:

$$\mathbb{P}\left(\{(x^{(i)}, y^{(i)})\}_{i=1}^m\right).$$

Prove that logistic regression is optimal also in a sense of maximizing the following *conditional* probability:

$$\mathbb{P}(y^{(1)}, \dots, y^{(m)} | x^{(1)}, \dots, x^{(m)}). \quad (3.78)$$

**Prob 8.2 (Multiclass classifier & softmax)** This problem explores a general setting in which the number of classes is arbitrary, say  $c$ . Let

$$z := [z_1, z_2, \dots, z_c]^T \in \mathbf{R}^c \quad (3.79)$$

be the output of a neural network model prior to activation. In an attempt to make those real values  $z_j$ 's being interpreted as *probability* quantities that lie in between 0 and 1, people usually employ the following activation function, called softmax:

$$\hat{y}_j := [\text{softmax}(z)]_j = \frac{e^{z_j}}{\sum_{k=1}^c e^{z_k}} \quad j \in \{1, 2, \dots, c\}. \quad (3.80)$$

Note that this is a natural extension of the logistic function: for  $c = 2$ ,

$$\begin{aligned} \hat{y}_1 &:= [\text{softmax}(z)]_1 = \frac{e^{z_1}}{e^{z_1} + e^{z_2}} \\ &= \frac{1}{1 + e^{-(z_1 - z_2)}} \\ &= \sigma(z_1 - z_2) \end{aligned} \quad (3.81)$$

where  $\sigma(\cdot)$  is the logistic function. Viewing  $z_1 - z_2$  as the binary classifier output  $\hat{y}$ , this coincides with logistic regression.

Let  $y \in \{[1, 0, \dots, 0]^T, [0, 1, 0, \dots, 0]^T, \dots, [0, \dots, 0, 1]^T\}$  be a label of one-hot-vector type. Here  $\hat{y}_i$  can be interpreted as the probability that the  $i$ th example

belongs to class  $i$ . Hence, let us assume that

$$\hat{y}_i = \mathbb{P}(y = [0, \dots, \underbrace{1}_{i\text{th position}}, \dots, 0]^T | x), \quad i \in \{1, \dots, c\}. \quad (3.82)$$

We also assume that training examples  $\{(x^{(i)}, y^{(i)})\}_{i=1}^m$  are independent over  $i$ .

- (a) Derive the likelihood of training examples:

$$\mathbb{P}\left(y^{(1)}, \dots, y^{(m)} | x^{(1)}, \dots, x^{(m)}\right). \quad (3.83)$$

This should be expressed in terms of  $y^{(i)}$ 's and  $\hat{y}^{(i)}$ 's.

- (b) Derive the optimal loss function that maximizes the likelihood (3.83).  
(c) What is the name of the optimal loss function derived in part (b)? What is the rationale behind the naming?

### Prob 8.3 (Jensen's inequality & cross entropy)

- (a) Suppose that a function  $f$  is concave and  $X$  is a discrete random variable. Show that

$$\mathbb{E}[f(X)] \leq f(\mathbb{E}[X]).$$

Also identify conditions under which the equality holds.

- (b) In Section 3.2, we defined cross entropy only for two *binary* random variables. Actually it can be defined for any two arbitrary distributions, say  $p$  and  $q$ , as:

$$H(p, q) := - \sum_{x \in \mathcal{X}} p(x) \log q(x) = \mathbb{E}_p \left[ \log \frac{1}{q(X)} \right] \quad (3.84)$$

where  $X \in \mathcal{X}$  is a discrete random variable. Show that

$$H(p, q) \geq H(p) := - \sum_{x \in \mathcal{X}} p(x) \log p(x) = \mathbb{E} \left[ \log \frac{1}{p(X)} \right] \quad (3.85)$$

where  $H(p)$  is known as the Shannon entropy. Also identify conditions under which the equality in (3.85) holds.

**Prob 8.4 (Kullback-Leibler divergence)** In statistics, information theory and machine learning, there is a very well-known divergence measure, called Kullback-Leibler divergence, defined as: for two distributions  $p$  and  $q$ ,

$$\text{KLD}(p, q) := \sum_{x \in \mathcal{X}} p(x) \log \frac{p(x)}{q(x)} = \mathbb{E}_p \left[ \log \frac{p(X)}{q(X)} \right] \quad (3.86)$$

where  $X \in \mathcal{X}$  is a discrete random variable. Show that

$$\text{KLD}(p, q) \geq 0. \quad (3.87)$$

Also identify conditions under which the equality in (3.87) holds.

**Prob 8.5 (Non-convexity of a 2-layer DNN)** Consider a 2-layer DNN with one input neuron, one hidden neuron and one output neuron. We employ ReLU activation for the hidden neuron while using the logistic function for the output neuron:

$$\hat{y} = \frac{1}{1 + e^{-w_2 \max(0, w_1 x)}} \quad (3.88)$$

where  $x \in \mathbf{R}$  indicates an input;  $w_1 \in \mathbf{R}$  and  $w_2 \in \mathbf{R}$  denote the weights for hidden and output layers, respectively. Let  $y \in \{0, 1\}$  be a label. Consider cross entropy loss for an objective function:

$$J(w_1, w_2) = -y \log \hat{y} - (1 - y) \log(1 - \hat{y}). \quad (3.89)$$

- (a) Consider a case in which  $(w_1, w_2) = (t, t)$  where  $t \in \mathbf{R}$ . Derive  $\nabla_t J(w_1, w_2)$ . Compute  $\nabla_t J(w_1, w_2)|_{t=0}$ .
- (b) Still consider the case in part (a). Derive  $\nabla_t^2 J(w_1, w_2)$ .
- (c) Now consider a different case in which  $(w_1, w_2) = (t, -t)$  where  $t \in \mathbf{R}$ . Derive  $\nabla_t J(w_1, w_2)$ . Compute  $\nabla_t J(w_1, w_2)|_{t=0}$ .
- (d) Consider the case in part (c). Derive  $\nabla_t^2 J(w_1, w_2)$ .
- (e) Show that  $J(w_1, w_2)$  is non-convex in  $(w_1, w_2)$ .

**Prob 8.6 (Backpropagation: 2-layer DNN with bias terms)** In Section 3.3, we considered an DNN architecture in which a linear operation that occurs at each neuron does not allow for having a bias term:

$$z^{[i]} = W^{[i]} a^{[i-1]} \quad (3.90)$$

where  $z^{[i]} \in \mathbf{R}^{n^{[i]}}$  and  $a^{[i]} \in \mathbf{R}^{n^{[i]}}$  indicate the pre-activation and post-activation outputs of the  $i$ th hidden layer, respectively; and  $W^{[i]} \in \mathbf{R}^{n^{[i]} \times n^{[i-1]}}$  denotes a weight matrix between the  $(i-1)$ th and  $i$ th hidden layers.

In this problem, we explore a slightly more general DNN architecture which allows for having a bias term in the linear operation:

$$z^{[i]} = W^{[i]} a^{[i-1]} + b^{[i]} \quad (3.91)$$

where  $b^{[i]} \in \mathbf{R}^{n^{[i]}}$ . One special structural assumption on  $b^{[i]}$  is that all the components in  $b^{[i]}$  are *identical*:

$$b^{[i]} = \begin{bmatrix} b_i \\ b_i \\ \vdots \\ b_i \end{bmatrix}$$

where  $b_i \in \mathbf{R}$ .

Consider a 2-layer DNN with such bias terms. We assume that the hidden layer activation is ReLU and the output layer activation is logistic. Let  $\{(x^{(i)}, y^{(i)})\}_{i=1}^m$  be training examples. Consider cross entropy loss for an objective function:

$$J(w) = \sum_{i=1}^m -y^{(i)} \log \hat{y}^{(i)} - (1 - y^{(i)}) \log(1 - \hat{y}^{(i)}) \quad (3.92)$$

where  $w := (W^{[2]}, W^{[1]})$ .

(a) Suppose  $m = 1$ . Show that

$$\frac{dJ(w)}{dz^{[2]}} = \hat{y} - y; \quad (3.93)$$

$$\frac{dJ(w)}{dW^{[2]}} = \frac{dJ(w)}{dz^{[2]}} a^{[1]T}; \quad (3.94)$$

$$\frac{dJ(w)}{db^{[2]}} = \frac{dJ(w)}{dz^{[2]}}; \quad (3.95)$$

$$\frac{dJ(w)}{da^{[1]}} = W^{[2]T} \frac{dJ(w)}{dz^{[2]}}; \quad (3.96)$$

$$\frac{dJ(w)}{dz^{[1]}} = \frac{dJ(w)}{da^{[1]}} * \mathbf{1}\{z^{[1]} \geq 0\}; \quad (3.97)$$

$$\frac{dJ(w)}{dW^{[1]}} = \frac{dJ(w)}{dz^{[1]}} x^T; \quad (3.98)$$

$$\frac{dJ(w)}{db^{[1]}} = \frac{dJ(w)}{dz^{[1]}}. \quad (3.99)$$

(b) Consider the general  $m$  case. Let

$$\begin{aligned} Y &:= [y^{(1)} \quad y^{(2)} \quad \dots \quad y^{(m)}] \in \mathbf{R}^{1 \times m}; \\ \hat{Y} &:= [\hat{y}^{(1)} \quad \hat{y}^{(2)} \quad \dots \quad \hat{y}^{(m)}] \in \mathbf{R}^{1 \times m}; \\ A^{[1]} &:= [a^{[1],(1)} \quad a^{[1],(2)} \quad \dots \quad a^{[1],(m)}] \in \mathbf{R}^{n^{[1]} \times m}; \\ Z^{[1]} &:= [z^{[1],(1)} \quad z^{[1],(2)} \quad \dots \quad z^{[1],(m)}] \in \mathbf{R}^{n^{[1]} \times m}; \\ Z^{[2]} &:= [z^{[2],(1)} \quad z^{[2],(2)} \quad \dots \quad z^{[2],(m)}] \in \mathbf{R}^{n^{[2]} \times m}; \\ X &:= [x^{(1)} \quad x^{(2)} \quad \dots \quad x^{(m)}] \in \mathbf{R}^{n \times m}. \end{aligned} \tag{3.100}$$

Show that

$$\frac{dJ(w)}{dZ^{[2]}} = \hat{Y} - Y; \tag{3.101}$$

$$\frac{dJ(w)}{dW^{[2]}} = \frac{dJ(w)}{dZ^{[2]}} A^{[1]T}; \tag{3.102}$$

$$\frac{dJ(w)}{db^{[2]}} = \sum_{i=1}^m \left[ \frac{dJ(w)}{dZ^{[2]}} \right]_i; \tag{3.103}$$

$$\frac{dJ(w)}{dA^{[1]}} = W^{[2]T} \frac{dJ(w)}{dZ^{[2]}}; \tag{3.104}$$

$$\frac{dJ(w)}{dZ^{[1]}} = \frac{dJ(w)}{dA^{[1]}} \cdot \mathbf{1}\{Z^{[1]} \geq 0\}; \tag{3.105}$$

$$\frac{dJ(w)}{dW^{[1]}} = \frac{dJ(w)}{dZ^{[1]}} X^T; \tag{3.106}$$

$$\frac{dJ(w)}{db^{[1]}} = \sum_{i=1}^m \left[ \frac{dJ(w)}{dZ^{[1]}} \right]_i \tag{3.107}$$

where  $\left[ \frac{dJ(w)}{dZ^{[j]}} \right]_i$  indicates the  $i$ th column component of  $\frac{dJ(w)}{dZ^{[j]}}$  for  $j \in \{1, 2\}$ .

**Prob 8.7 (Backpropagation: 3-layer DNN)** Consider a 3-layer DNN such that:

$$\begin{aligned} z^{[i]} &= W^{[i]} a^{[i-1]} \in \mathbf{R}^{n^{[i]}} \quad i \in \{1, 2, 3\}; \\ a^{[i]} &= \max(0, z^{[i]}) \in \mathbf{R}^{n^{[i]}} \quad i \in \{1, 2\}; \end{aligned} \tag{3.108}$$

$$\hat{y} := a^{[3]} = \frac{1}{1 + e^{-z^{[3]}}} \in \mathbf{R}$$

where  $a^{[0]}$  is defined as the input  $x \in \mathbf{R}^n$ , and  $W^{[i]} \in \mathbf{R}^{n^{[i]} \times n^{[i-1]}}$  denotes a weight matrix associated with the  $(i-1)$ th and  $i$ th hidden layers. Let  $\{(x^{(i)}, y^{(i)})\}_{i=1}^m$  be training examples. Consider cross entropy loss for an objective function:

$$J(w) = \sum_{i=1}^m -y^{(i)} \log \hat{y}^{(i)} - (1 - y^{(i)}) \log(1 - \hat{y}^{(i)}) \quad (3.109)$$

where  $w := (W^{[3]}, W^{[2]}, W^{[1]})$ . Describe the detailed procedure of *backpropagation*, i.e., draw the forward and backward paths and list all the key gradient-related equations with a correct order.

**Prob 8.8 (Implementing the XOR function)** Consider a 2-layer DNN with two input neurons, two hidden neurons and one output neuron. We employ ReLU activation for the hidden layer while taking logistic activation for the output neuron. We allow for having bias terms in all of the layers:

$$\hat{y}^{(i)} = \sigma \left( W^{[2]} \max(0, W^{[1]}x^{(i)} + b^{[1]}) + b^{[2]} \right) \quad (3.110)$$

where  $x^{(i)} \in \{0, 1\}^2$  indicates the  $i$ th input example;  $W^{[1]} \in \mathbf{R}^{2 \times 2}$  and  $W^{[2]} \in \mathbf{R}^{1 \times 2}$  denote the weight matrices for hidden and output layers, respectively; and  $b^{[1]} \in \mathbf{R}^2$  and  $b^{[2]} \in \mathbf{R}$  indicate the bias terms at hidden and output layers, respectively. Let  $y^{(i)} \in \{0, 1\}$  be the  $i$ th label. Let training examples be:

$$(x^{(i)}, y^{(i)}) = \begin{cases} ((0, 0), 0), & i = 1; \\ ((0, 1), 1), & i = 2; \\ ((1, 0), 1), & i = 3; \\ ((1, 1), 0), & i = 4. \end{cases}$$

Consider cross entropy loss for an objective function:

$$J(w) = \sum_{i=1}^4 -y^{(i)} \log \hat{y}^{(i)} - (1 - y^{(i)}) \log(1 - \hat{y}^{(i)}) \quad (3.111)$$

where  $w := (W^{[2]}, W^{[1]})$ . Use the same matrix notations as in (3.100).

- (a) Draw the forward path. Implement a Python script for the following function:

```
[Yhat,Z2,A1,Z1] = ForwardPath(W1,b1,W2,b2,X)
```

where:

$$\begin{aligned} & (\text{Yhat}, \text{Z2}, \text{A1}, \text{Z1}, \text{W1}, \text{b1}, \text{W2}, \text{b2}, \text{X}) \\ & := (\hat{Y}, Z^{[2]}, A^{[1]}, Z^{[1]}, W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}, X). \end{aligned}$$

- (b) Draw the backward path for backprop. Implement a Python script for the following function:

$$\begin{aligned} & [\text{dZ2}, \text{dW2}, \text{db2}, \text{dA1}, \text{dZ1}, \text{dW1}, \text{db1}] \\ & = \text{BackwardPath}(\text{Y}, \text{Yhat}, \text{W1}, \text{b1}, \text{W2}, \text{b2}, \text{A1}, \text{Z1}, \text{X}) \end{aligned}$$

where:

$$\begin{aligned} & (\text{dZ2}, \text{dW2}, \text{db2}, \text{dA1}, \text{dZ1}, \text{dW1}, \text{db1}) \\ & := \left( \frac{\partial J(w)}{\partial Z^{[2]}}, \frac{\partial J(w)}{\partial W^{[2]}}, \frac{\partial J(w)}{\partial b^{[2]}}, \frac{\partial J(w)}{\partial A^{[1]}}, \frac{\partial J(w)}{\partial Z^{[1]}}, \frac{\partial J(w)}{\partial W^{[1]}}, \frac{\partial J(w)}{\partial b^{[1]}} \right). \end{aligned}$$

- (c) Using the already implemented functions in parts (a) and (b), implement a Python script for training the DNN via gradient descent.  
(d) Consider the following weight initialization:

```
import random
from numpy.random import randn

random.seed(0)

W1 = randn(n1,n)
W2 = randn(1,n1)
b1 = randn(n1,1)
b2 = randn(1,1)
```

where  $(n1, n) := (n^{[1]}, n) = (2, 2)$ . Set the learning rate as 0.1. Fix the number of iterations (also called epoches) as 10000. Run the code implemented in part (c) together with the above initialization to compute  $(W1, b1, W2, b2)$ .

- (e) For an input  $x \in \{(0, 0), (0, 1), (1, 0), (1, 1)\}$ , use  $(W1, b1, W2, b2)$  computed in part (d) to obtain  $\hat{y}$ .

**Prob 8.9 (Optimizers)** Consider the gradient descent algorithm:

$$w^{(t+1)} = w^{(t)} - \alpha \nabla J(w^{(t)})$$

where  $w^{(t)}$  indicates the weights of an interested model at the  $t$ th iteration;  $J(w^{(t)})$  denotes the cost function evaluated at  $w^{(t)}$ ; and  $\alpha$  is the learning rate. Note that only the *current* gradient, reflected in  $\nabla J(w^{(t)})$ , affects the weight update.

- (a) (*Momentum optimizer*) In the literature, there is a prominent variant of gradient descent that takes into account *past* gradients as well. Using such past information, one can damp an oscillating effect in the weight update that may incur instability in training. To capture past gradients and therefore address the oscillation problem, another quantity, often denoted by  $m^{(t)}$ , is usually introduced:

$$m^{(t)} = \beta m^{(t-1)} - (1 - \beta) \nabla J(w^{(t)}) \quad (3.112)$$

where  $\beta$  denotes another hyperparameter that captures the weight of the past gradients, simply called the *momentum*. Here  $m$  stands for the *momentum* vector. The variant of the algorithm (often called the *momentum optimizer*) takes the following update for  $w^{(t+1)}$ :

$$w^{(t+1)} = w^{(t)} + \alpha m^{(t)}. \quad (3.113)$$

Show that

$$w^{(t+1)} = w^{(t)} - \alpha(1 - \beta) \sum_{k=0}^{t-1} \beta^k \nabla J(w^{(t-k)}) + \alpha \beta^t m^{(0)}.$$

- (b) (*Bias correction*) Assuming that  $\nabla J(w^{(t)})$  is the same for all  $t$  and  $m^{(0)} = 0$ , show that

$$w^{(t+1)} = w^{(t)} - \alpha(1 - \beta^t) \nabla J(w^{(t)}).$$

*Note:* For a large value of  $t$ ,  $1 - \beta^t \approx 1$ , so it has almost the same scaling as that in the regular gradient descent. On the other hand, for a small value of  $t$ ,  $1 - \beta^t$  can be small, being far from 1. For instance, when  $\beta = 0.9$  and  $t = 2$ ,  $1 - \beta^t = 0.19$ . This motivates people to rescale the moment  $m^{(t)}$  in (3.112) through division by  $1 - \beta^t$ . So in practice, we use:

$$\hat{m}^{(t)} = \frac{m^{(t)}}{1 - \beta^t}; \quad (3.114)$$

$$w^{(t+1)} = w^{(t)} + \alpha \hat{m}^{(t)}. \quad (3.115)$$

This technique is so called the *bias correction*.

- (c) (*Adam optimizer*) Notice in (3.112) that a very large or very small value of  $\nabla J(w^{(t)})$  affects the weight update in quite a different scaling. In an effort to avoid such a different scaling problem, people in practice often make *normalization* in the weight update (3.115) via a normalization factor, often

denoted by  $\hat{s}^{(t)}$ :

$$w^{(t+1)} = w^{(t)} + \alpha \frac{\hat{m}^{(t)}}{\sqrt{\hat{s}^{(t)}} + \epsilon} \quad (3.116)$$

where the division is component-wise, and

$$\hat{m}^{(t)} = \frac{m^{(t)}}{1 - \beta_1^t}, \quad (3.117)$$

$$m^{(t)} = \beta_1 m^{(t-1)} - (1 - \beta_1) \nabla J(w^{(t)}), \quad (3.118)$$

$$\hat{s}^{(t)} = \frac{s^{(t)}}{1 - \beta_2^t}, \quad (3.119)$$

$$s^{(t)} = \beta_2 s^{(t-1)} + (1 - \beta_2) (\nabla J(w^{(t)}))^2. \quad (3.120)$$

Here  $(\cdot)^2$  indicates a component-wise square;  $\epsilon$  is a tiny value introduced to avoid division by 0 in practice (usually  $10^{-8}$ ); and  $s$  stands for *square*. This optimizer (3.116) is called the Adam optimizer. Explain the rationale behind the division by  $1 - \beta_2^t$  in (3.120).

**Prob 8.10 (TensorFlow implementation of a digit classifier)** Consider a handwritten digit classifier that we learned in Section 3.5. In this problem, you are asked to build a classifier using a two-layer (one hidden-layer) neural network with ReLU activation in the hidden layer and softmax activation in the output layer.

- (a) (*MNIST dataset loading*) Use the following script (or otherwise), load the MNIST dataset:

```
from tensorflow.keras.datasets import mnist
(X_train,y_train),(X_test,y_test)=mnist.load_data()
X_train = X_train/255.
X_test = X_test/255.
```

What are  $m$  (the number of training examples) and  $m_{\text{test}}$ ? What are the shapes of  $X_{\text{train}}$  and  $y_{\text{train}}$ ?

- (b) (*Data visualization*) Upon the code in part (a) being executed, report an output for the following:

```
import matplotlib.pyplot as plt
num_of_images = 60
for index in range(1,num_of_images+1):
    plt.subplot(6,10, index)
    plt.axis('off')
    plt.imshow(X_train[index], cmap = 'gray_r')
```

- (c) (*Model*) Using a skeleton code provided in Section 3.5, write a script for a 2-layer neural network model with 500 hidden units fed by MNIST data.
- (d) (*Training*) Using a skeleton code in Section 3.5, write a script for training the model generated in part (c) with cross entropy loss. Use the Adam optimizer with:

$$\text{learning rate} = 0.001; \quad (\beta_1, \beta_2) = (0.9, 0.999)$$

and the number of epochs is 10. Also plot a training loss as a function of epochs.

- (e) (*Testing*) Using a skeleton code in Section 3.5, write a script for testing the model (trained in part (d)). What is the test accuracy?

### Prob 8.11 (True or False?)

- (a) For two arbitrary distributions, say  $p$  and  $q$ , consider cross entropy  $H(p, q)$ . Then,

$$H(p, q) \geq H(q) \quad (3.121)$$

where  $H(q)$  is the Shannon entropy w.r.t.  $q$ .

- (b) For two arbitrary distributions, say  $p$  and  $q$ , consider cross entropy:

$$H(p, q) := - \sum_{x \in \mathcal{X}} p(x) \log q(x) = \mathbb{E}_p \left[ \log \frac{1}{q(X)} \right] \quad (3.122)$$

where  $X \in \mathcal{X}$  is a discrete random variable. Then,

$$H(p, q) = H(p) := - \sum_{x \in \mathcal{X}} p(x) \log p(x). \quad (3.123)$$

only when  $q = p$ .

- (c) Consider a binary classifier in the supervised learning setup where we are given input-output example pairs  $\{(x^{(i)}, y^{(i)})\}_{i=1}^m$ . Let  $0 \leq \hat{y}^{(i)} \leq 1$  be the classifier output for the  $i$ th example. Let  $w$  be parameters of the classifier. Define:

$$w_{\text{CE}}^* := \arg \min_w \frac{1}{m} \sum_{i=1}^m \ell_{\text{CE}}(y^{(i)}, \hat{y}^{(i)})$$

$$w_{\text{KL}}^* := \arg \min_w \frac{1}{m} \sum_{i=1}^m \text{KLD}(y^{(i)}, \hat{y}^{(i)})$$

where  $\ell_{\text{CE}}(\cdot, \cdot)$  denotes cross entropy loss and  $\text{KLD}(y^{(i)}, \hat{y}^{(i)})$  indicates the KL divergence between two binary random variables with parameters  $y^{(i)}$

and  $\hat{y}^{(i)}$ , respectively. Then,

$$w_{\text{CE}}^* = w_{\text{KL}}^*.$$

- (d) Consider a 2-layer DNN with one input neuron, one hidden neuron and one output neuron. We employ the *linear* activation both for the hidden and output neurons:

$$\hat{y} = w_2 w_1 x \quad (3.124)$$

where  $x \in \mathbf{R}$  indicates an input;  $w_1 \in \mathbf{R}$  and  $w_2 \in \mathbf{R}$  denote the weights for hidden and output layers, respectively. Let  $y \in \{-1, 1\}$  be a label. Consider the squared error loss for an objective function:

$$J(w_1, w_2) = \|y - \hat{y}\|^2. \quad (3.125)$$

Then, the objective function is convex in  $(w_1, w_2)$ .

- (e) One of the reasons that DNNs were not appreciated much during the AI winter is that the DNN model was so complex in view of the technology of the day although it offers better performances relative to simpler approaches.
- (f) Suppose we execute the following code:

```
import numpy as np
a = np.random.randn(4,3,3)
b = np.ones_like(a)
print(b[0].shape)
print(b.shape[0])
```

Then, the two prints yield the same results.

- (g) Suppose that `image` is an MNIST image of numpy array type. Then, one can use the following commands to plot the image:

```
import matplotlib.pyplot as plt
plt.imshow(image.squeeze(), cmap='gray_r')
```

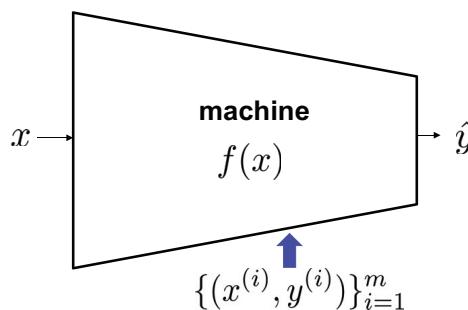
## 3.6 Unsupervised Learning: Generative Modeling

**Recap** During the past five sections, we have studied some basic contents on supervised learning. The goal of supervised learning is to estimate a function  $f(\cdot)$  of an interested computer system (machine) from input-output samples, as illustrated in Fig. 3.28.

In an effort to translate a *function* optimization problem (a natural formulation of supervised learning) into a parameter-based optimization problem that we are familiar with, we expressed the function with parameters (or called weights) assuming a certain architecture of the system.

The certain architecture was: *Perceptron*. Taking the logistic function together with cross entropy loss, we obtained logistic regression. We then proved that logistic regression is optimal in a sense of maximizing the likelihood of training data.

We next considered the Deep Neural Networks (DNNs) architecture for  $f(\cdot)$ , which has been shown to be more expressive. Since there is no theoretical basis on the choice of activation functions in the DNN context, we investigated only a rule-of-thumb which is common to use in the field: Taking ReLU at all hidden neurons while taking the logistic function at the output layer. We have a theoretical justification only on the choice of a loss function: cross entropy loss. We have also learned that in many of the interested settings, optimization problems for DNNs have no spurious local minima, although the problems are highly non-convex. This motivated the use of gradient descent for such problems. We also studied an efficient way of computing gradients: *backpropagation*, or simply called *backprop*. Lastly, we investigated how to implement neural networks via TensorFlow.



**Figure 3.28.** Supervised learning: Learning the function  $f(\cdot)$  of an interested system from data  $\{(x^{(i)}, y^{(i)})\}_{i=1}^m$ .

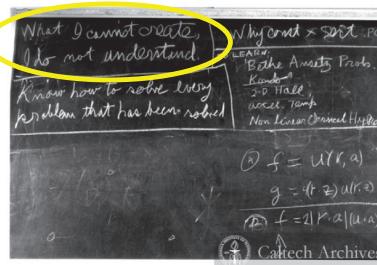
**Outline** What is next? In fact, we face one critical challenge in supervised learning. The challenge is that it is not that easy to collect *labeled* data in many realistic situations. In general, gathering labeled data is very expensive, as it usually requires extensive human-labour-based annotations. So people wish to do something without such labeled data. Then, a natural question arises. What can we do *only* with  $\{x^{(i)}\}_{i=1}^m$ ?

This is where the concept of *unsupervised learning* kicks in. Unsupervised learning is a methodology for learning something about data  $\{x^{(i)}\}_{i=1}^m$ . You may then ask: What is *something*? There are a few candidates for such something to learn in the field. Depending on target candidates, there are different unsupervised learning methods.

In this section, we will start investigating details on these. Specifically we are going to cover the following four stuffs. First of all, we will study what such candidates for something to learn are. We will then investigate what the corresponding unsupervised learning methods are. Next we will focus on arguably the most prominent and fundamental learning method among them: *Generative modeling*. Finally, we will connect this to optimization of this book's interest, by formulating an optimization problem for generative models.

**Candidates for something to learn** There are three candidates for something to learn, from simple to complex. The first candidate, which is perhaps the simplest, is the *basic structure* of data. For instance, when  $\{x_i\}_{i=1}^m$  indicates users/customers data, such basic structures could be *membership* of individuals, community type, gender type, or race type. For products-related data, it could be *abnormal (defect)* vs *normal* information. The second candidate is the one that we learned about in Part I, which is *features*: expressive (and/or compressed) components that well describe characteristics of data. The last is a sort of the most complex yet most fundamental information: the *probability distribution* of data, which allows us to create data as we wish.

**Three unsupervised learning methods** Depending on which candidate we focus on, we have three different unsupervised learning methods. The first is *clustering*, which serves to identify the basic structures of data. You may hear of k-means, k-nearest neighbors, community-detection, or anomaly-detection algorithms. All of these belong to this category. The second is *feature learning* (or called *representation learning*), which allows us to extract some well-representative features. You may hear of autoencoder, matrix factorization, principal component analysis, or dictionary learning, all of which can be categorized into this class. The last is *generative modeling*, which enables us to create arbitrary examples that well mimick characteristics of real data. This is actually the most famous unsupervised learning method,



Richard Feynman '88

**Figure 3.29.** Richard Feynman left a quote on the relationship between *understanding* and *creating* on a blackboard around right before he died in 1988. The quote says, “What I cannot create, I do not understand.” What this quote suggests is that being able to create convincing examples of data is a strong evidence of having understood it.

which has received a particularly significant attention in the field nowadays. So in this book, we are going to focus on this method.

**Why is generative modeling prominent?** Before explaining details on generative modeling, let us say a few words about why generative modeling is most prominent in the field. We list three reasons below which we believe major.

The first reason is somewhat related to a famous quote by Richard Feynman; see Fig. 3.29. Right before he died in 1988, he left an intriguing quote in his blackboard: “*What I cannot create, I do not understand.*” What this quote implies in the context of *unsupervised learning* is that creating convincing examples of data is a *necessary condition* for complete understanding. In this regard, a generative model serves an important role as it enables us to create arbitrary yet plausible examples that mimick real data.

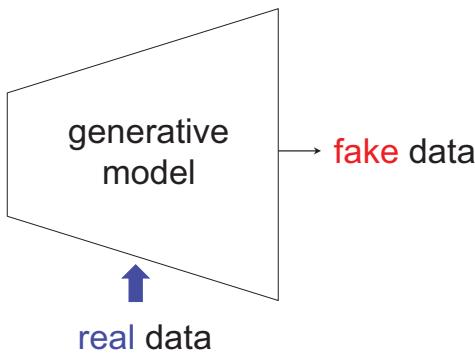
The second reason is related to a recent breakthrough made in the history of the AI field by a research scientist, named Ian Goodfellow; see Fig. 3.30. During his PhD, he could develop a powerful generative model, which he named “*Generative Adversarial Networks (GANs)*” (Goodfellow et al., 2014). The GANs are shown to be extremely instrumental in a wide variety of applications, even not limited to the AI field. Such applications include: image creation, human image synthesis, image inpainting, coloring, super-resolution image synthesis, speech synthesis, style transfer, robot navigation, to name a few. Since it works pretty well, in 2019, the state of California passed a bill that would ban the use of GANs to make fake pornography without the consent of the people depicted. So the GANs have played a crucial role to popularize generative modeling.

The third reason is related to *optimization* of this book’s interest. The GANs borrow very interesting ideas from *optimization*, thus making many optimization experts excited about the generative models. In particular, the duality theorems that



Ian Goodfellow 2014

**Figure 3.30.** Ian Goodfellow, a young figure in the modern AI field. He is best known as the inventor of the Generative Adversarial Networks (GANs), which made a big wave in the history of the AI field.



**Figure 3.31.** A generative model is the one that generates *fake* data which resembles *real* data. Here what *resembling* means in a mathematical language is that it has a *similar distribution*.

we studied in Part II play a crucial role to understand the GANs as well as many GAN variants.

**Generative modeling** Let us dive into details on generative modeling. Generative modeling is a technique for generating *fake* data so that it has a *similar distribution* as that of *real* data. See Fig. 3.31 for pictorial representation. The model parameters are learned via real data so that the learned model outputs fake data that resemble real data. Here an input signal can be either an *arbitrary random* signal or a specifically synthesized signal that forms the skeleton of fake data. The type of the input depends on applications of interest – this will be detailed later on.

**Remarks on generative models** In fact, the problem of designing a generative model is one of the most important problems in *statistics*, so it has been a classical age-old problem in that field. This is because the major goal of the field of statistics is to figure out (or estimate) the probability distribution of data that arise in the real

world (that we call real data), and the generative model plays a role as a underlying framework in achieving the goal. Actually the model can do even more. It provides a concrete function block (called the *generator* in the field) which can create realistic fake data. There is a very popular name in statistics that indicates such a problem, that is the *density estimation problem*. Here the density refers to the probability distribution.

As you may guess from the second reason mentioned above regarding why generative modeling is prominent, this problem was not that popular in the AI field until very recently, precisely 2014 when the GANs were invented.

**How to formulate an optimization problem?** Let us relate generative modeling to optimization of our interest. As mentioned earlier, we can feed some input signal (that we call *fake input*) which one can arbitrarily synthesize. Common ways employed in the field to generate them are to use Gaussian or uniform distributions. Since it is an *input* signal, we may wish to use a conventional “ $x$ ” notation. So let us use  $x \in \mathbf{R}^k$  to denote a fake input where  $k$  indicates the dimension of the signal.

Notice that this has a conflict with real data notation  $\{x^{(i)}\}_{i=1}^m$ . To avoid the conflict, let us use a different notation, say  $\{y^{(i)}\}_{i=1}^m$ , to denote real data. Please don't be confused with labeled data. These are not labels. In fact, the convention in the machine learning field is to use a notation  $z$  to indicate a fake input while maintaining real data notation as  $\{x^{(i)}\}_{i=1}^m$ . This may be another way to go; perhaps this is the way that you should take when writing papers. Anyhow let us take the first unorthodox yet reasonable option for this book.

Let  $\hat{y} \in \mathbf{R}^n$  be a fake output. Considering  $m$  examples, let  $\{(x^{(i)}, \hat{y}^{(i)})\}_{i=1}^m$  be such fake input-output  $m$  pairs and let  $\{y^{(i)}\}_{i=1}^m$  be  $m$  real data examples. See Fig. 3.32.

**Goal** Let  $G(\cdot)$  be a function of the generative model. Then, the goal of the generative model can be stated as: Designing  $G(\cdot)$  such that

$$\{\hat{y}^{(i)}\}_{i=1}^m \approx \{y^{(i)}\}_{i=1}^m \text{ in distribution.}$$

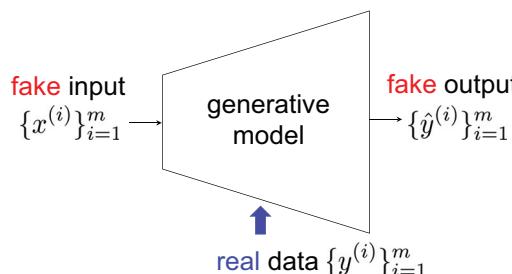


Figure 3.32. Problem formulation for generative modeling.

Here what does it mean by “in distribution”? To make it clear, we need to quantify closeness between two distributions. One natural yet prominent approach employed in the statistics field is to take the following two steps:

1. Compute empirical distributions or estimate distributions from  $\{y^{(i)}\}_{i=1}^m$  and  $\{(x^{(i)}, \hat{y}^{(i)})\}_{i=1}^m$ . Let such distributions be:

$$\mathbb{Q}_Y, \mathbb{Q}_{\hat{Y}}$$

for real and fake data, respectively.

2. Next employ a well-known *divergence measure* in statistics which can serve to quantify the closeness of two distributions. Let  $D(\cdot, \cdot)$  be one such divergence measure. Then, the similarity between  $\mathbb{Q}_Y$  and  $\mathbb{Q}_{\hat{Y}}$  can be quantified as:

$$D(\mathbb{Q}_Y, \mathbb{Q}_{\hat{Y}}).$$

Taking the above natural approach, one can concretely state the goal as: Designing  $G(\cdot)$  such that

$$D(\mathbb{Q}_Y, \mathbb{Q}_{\hat{Y}}) \text{ is minimized.}$$

**Optimization under the approach** Hence, under the approach, one can formulate an optimization problem as:

$$\min_{G(\cdot)} D(\mathbb{Q}_Y, \mathbb{Q}_{\hat{Y}}). \quad (3.126)$$

As you may easily notice, there are some issues in solving the above problem (3.126). There are three major issues.

The first is that it is *function optimization* which we are not familiar with. Notice that the optimization is over the function  $G(\cdot)$ . Second, the objective function  $D(\mathbb{Q}_Y, \mathbb{Q}_{\hat{Y}})$  is a very complicated function of the knob  $G(\cdot)$ . Note that  $\mathbb{Q}_{\hat{Y}}$  is a function of  $G(\cdot)$ , as  $\hat{y} = G(x)$ . So the objective is a twice folded composite function of  $G(\cdot)$ . The last is perhaps the most fundamental issue. It is not clear as to how to choose a divergence measure  $D(\cdot, \cdot)$ .

**Look ahead** It turns out there are some ways to address the above issues. Interestingly, one such way leads to an optimization problem for GANs. So in the next section, we will study what that way is, and then will take the way to derive an optimization problem for GANs.

## 3.7 Generative Adversarial Networks (GANs)

---

**Recap** In the previous section, we started investigating *unsupervised learning*. The goal of unsupervised learning is to learn something about data, which we newly denoted by  $\{y^{(i)}\}_{i=1}^m$ , instead of  $\{x^{(i)}\}_{i=1}^m$ . Depending on target candidates for something to learn, there are a few unsupervised learning methods. Among them, we explored one prominent method, which is *generative modeling*. We formulated an optimization problem for generative modeling:

$$\min_{G(\cdot)} D(Q_Y, Q_{\hat{Y}}) \quad (3.127)$$

where  $Q_Y$  and  $Q_{\hat{Y}}$  indicate the empirical distributions (or the estimates of the true distributions) for real and fake data, respectively;  $G(\cdot)$  denotes the function of a generative model; and  $D(\cdot, \cdot)$  is a divergence measure. We then encountered a couple of issues that arise in the problem: (i) it is a *function optimization* which we are not familiar with; (ii) the objective is a very complicated function of  $G(\cdot)$ ; and (iii) it is not that clear as to how to choose  $D(\cdot, \cdot)$ .

At the end of the last section, we claimed that there are some ways to address such issues, and interestingly, one such way leads to an optimization problem for a recent powerful generative model, named Generative Adversarial Networks (GANs).

**Outline** In this section, we are going to explore details on GANs. What we are going to do are three folded. First we will investigate what that way leading to GANs is. We will then take the way to derive an optimization problem for GANs. Lastly we will demonstrate that GANs indeed address the issues: (i) the GAN optimization problem is tractable; and (ii) the problem can also be expressed as that in (3.127).

**What is the way to address the issues?** Remember one challenge that we faced in the optimization problem (3.127):  $D(Q_Y, Q_{\hat{Y}})$  is a complicated function of  $G(\cdot)$ . To address this, we take an *indirect way* to represent  $D(Q_Y, Q_{\hat{Y}})$ . We first observe how  $D(Q_Y, Q_{\hat{Y}})$  should behave, and then based on the observation, we will come up with an *indirect way* to mimic the behaviour. It turns out the way leads us to explicitly compute  $D(Q_Y, Q_{\hat{Y}})$ . Below are details.

**How  $D(Q_Y, Q_{\hat{Y}})$  should behave?** One observation that we can make is that if one can *easily discriminate* real data  $y$  from fake data  $\hat{y}$ , then the divergence must be *large*; otherwise, it should be small. This naturally motivates us to:

Interpret  $D(Q_Y, Q_{\hat{Y}})$  as *the ability to discriminate*.

We introduce an entity that can play this discriminating role. The entity is the one that Ian Goodfellow, the inventor of GAN, introduced, and he named it:

*Discriminator.*

Goodfellow considered a simple *binary-output* discriminator which takes as an input, either real data  $y$  or fake data  $\hat{y}$ . He then wanted to design  $D(\cdot)$  such that  $D(\cdot)$  well *approximates* the probability that the input  $(\cdot)$  is *real* data:

$$D(\cdot) \approx \mathbb{P}((\cdot) = \text{real data}).$$

Noticing that

$$\mathbb{P}(y = \text{real}) = 1;$$

$$\mathbb{P}(\hat{y} = \text{real}) = 0,$$

he wanted to design  $D(\cdot)$  such that:

$D(y)$  is as large as possible, close to 1;

$D(\hat{y})$  is as small as possible, close to 0.

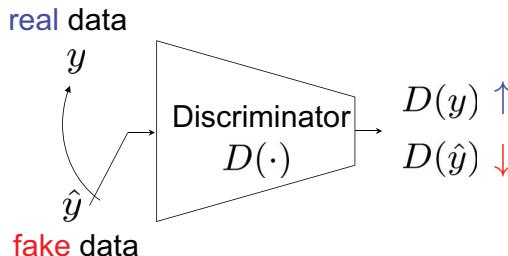
See Fig. 3.33.

**How to quantify the ability to discriminate?** Keeping the picture in Fig. 3.33 in his mind, he wanted to quantify the ability to discriminate. To this end, he first observed that if  $D(\cdot)$  can easily discriminate, then we should have:

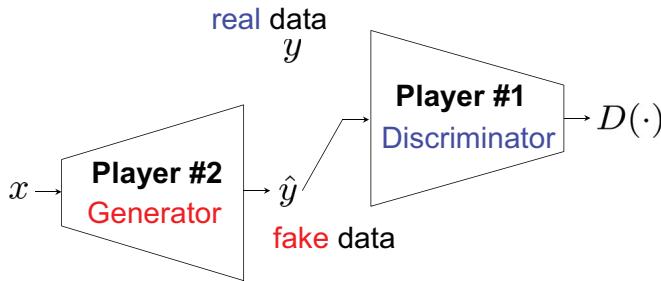
$$D(y) \uparrow; \quad 1 - D(\hat{y}) \uparrow.$$

One naive way to capture the ability is simply *adding* the above two terms. But Goodfellow did not take the naive way. Instead he took the following *logarithmic* summation:

$$\log D(y) + \log(1 - D(\hat{y})). \quad (3.128)$$



**Figure 3.33.** Discriminator wishes to output  $D(\cdot)$  such that  $D(y)$  is as large as possible while  $D(\hat{y})$  is as small as possible.



**Figure 3.34.** A two-player game for GAN: Discriminator  $D(\cdot)$ , wishes to maximize the quantified ability (3.129), while another player, generator  $G(\cdot)$ , wants to minimize (3.129).

In NeurIPS 2016, Goodfellow gave a tutorial on GANs, mentioning that the problem formulation was inspired by a paper published in AISTATS 2010 ([Gutmann and Hyvärinen, 2010](#)). See Eq. (3) in the paper.

Making the particular choice, the ability to discriminate for  $m$  examples can be quantified as:

$$\frac{1}{m} \sum_{i=1}^m \log D(y^{(i)}) + \log(1 - D(\hat{y}^{(i)})). \quad (3.129)$$

**A two-player game** Goodfellow then introduced a *two-player game* in which player 1, discriminator  $D(\cdot)$ , wishes to maximize the quantified ability (3.129), while player 2, generator  $G(\cdot)$ , wants to minimize (3.129). See Fig. 3.34 for illustration.

**Optimization for GANs** The two-player game motivated him to formulate the following min max optimization problem:

$$\min_{G(\cdot)} \max_{D(\cdot)} \frac{1}{m} \sum_{i=1}^m \log D(y^{(i)}) + \log(1 - D(\hat{y}^{(i)})). \quad (3.130)$$

You may wonder why not max min. That may be another way to go, but Goodfellow made the above choice. In fact, there is a reason why the way is taken. This will be clearer soon. Notice that the optimization is over the two *functions* of  $D(\cdot)$  and  $G(\cdot)$ , meaning that it is still a *function* optimization. Luckily the year of 2014 (when the GAN paper was published) was after the starting point of the deep learning revolution, the year of 2012. So Goodfellow was very much aware of the power of neural networks:

*“Deep neural networks can well represent any arbitrary function.”*

This motivated him to parameterize the two functions with DNNs, which in turn led to the following optimization problem:

$$\min_{G(\cdot) \in \mathcal{N}} \max_{D(\cdot) \in \mathcal{N}} \frac{1}{m} \sum_{i=1}^m \log D(y^{(i)}) + \log(1 - D(\hat{y}^{(i)})) \quad (3.131)$$

where  $\mathcal{N}$  denotes a set of DNN-based functions. This is exactly the optimization problem for GANs.

**Related to original optimization?** Remember what we mentioned earlier. The way leading to the GAN optimization is an indirect way of solving the original optimization problem:

$$\min_{G(\cdot)} D(\mathbb{Q}_Y, \mathbb{Q}_{\hat{Y}}). \quad (3.132)$$

Then, a natural question arises. How are the two problems (3.131) and (3.132) related? It turns out these are very much related. This is exactly where the choice of min max (instead of max min) plays the role; the other choice cannot establish a connection. It has been shown that assuming that deep neural networks can represent any arbitrary function, the GAN optimization (3.131) can be translated into the original optimization form (3.132). We will prove this below.

**Simplification & manipulation** Let us start by simplifying the GAN optimization (3.131). Since we assume that  $\mathcal{N}$  can represent any arbitrary function, the problem (3.131) becomes *unconstrained*:

$$\min_{G(\cdot)} \max_{D(\cdot)} \frac{1}{m} \sum_{i=1}^m \log D(y^{(i)}) + \log(1 - D(\hat{y}^{(i)})). \quad (3.133)$$

Notice that the objective is a function of  $D(\cdot)$ , and the two functions  $D(\cdot)$ 's appear but with *different* arguments: one is  $y^{(i)}$ , marked in blue; the other is  $\hat{y}^{(i)}$ , marked in red. So in the current form (3.133), the inner (max) optimization problem is not quite tractable to solve. In an attempt to make it tractable, let us express it in a different manner using the following notations.

Define a random vector  $Y$  which takes one of the  $m$  real examples with probability  $\frac{1}{m}$  (uniform distribution):

$$Y \in \{y^{(1)}, \dots, y^{(m)}\} =: \mathcal{Y}; \quad \mathbb{Q}_Y(y^{(i)}) = \frac{1}{m}, \quad i \in \{1, 2, \dots, m\}$$

where  $\mathbb{Q}_Y$  indicates the probability distribution of  $Y$ . Similarly define  $\hat{Y}$  for fake examples:

$$\hat{Y} \in \{\hat{y}^{(1)}, \dots, \hat{y}^{(m)}\} =: \hat{\mathcal{Y}}; \quad \mathbb{Q}_{\hat{Y}}(\hat{y}^{(i)}) = \frac{1}{m}, \quad i \in \{1, 2, \dots, m\}$$

where  $\mathbb{Q}_{\hat{Y}}$  indicates the probability distribution of  $\hat{Y}$ . Using these notations, one can rewrite the problem (3.133) as:

$$\min_{G(\cdot)} \max_{D(\cdot)} \sum_{i=1}^m \mathbb{Q}_Y(y^{(i)}) \log D(y^{(i)}) + \mathbb{Q}_{\hat{Y}}(\hat{y}^{(i)}) \log(1 - D(\hat{y}^{(i)})). \quad (3.134)$$

Still we have different arguments in the two  $D(\cdot)$  functions.

To address this, let us introduce another notation. Let  $z \in \mathcal{Y} \cup \hat{\mathcal{Y}}$ . Newly define  $\mathbb{Q}_Y(\cdot)$  and  $\mathbb{Q}_{\hat{Y}}(\cdot)$  such that:

$$\mathbb{Q}_Y(z) := 0 \text{ if } z \in \hat{\mathcal{Y}} \setminus \mathcal{Y}; \quad (3.135)$$

$$\mathbb{Q}_{\hat{Y}}(z) := 0 \text{ if } z \in \mathcal{Y} \setminus \hat{\mathcal{Y}}. \quad (3.136)$$

Using the  $z$  notation, one can then rewrite the problem (3.134) as:

$$\min_{G(\cdot)} \max_{D(\cdot)} \sum_{z \in \mathcal{Y} \cup \hat{\mathcal{Y}}} \mathbb{Q}_Y(z) \log D(z) + \mathbb{Q}_{\hat{Y}}(z) \log(1 - D(z)). \quad (3.137)$$

We see that the same arguments appear in the two  $D(\cdot)$  functions.

**Solving the inner optimization problem** We are ready to solve the inner optimization problem in (3.137). Key observations are:  $\log D(z)$  is concave in  $D(\cdot)$ ;  $\log(1 - D(z))$  is concave in  $D(\cdot)$ ; and therefore, the objective function is concave in  $D(\cdot)$ . This implies that the objective has the *unique maximum* in the function space  $D(\cdot)$ . Hence, one can find the maximum by searching for the one in which the derivative is zero. Taking a derivative and setting it to zero, we get:

$$\text{Derivative} = \sum_z \left[ \frac{\mathbb{Q}_Y(z)}{D^*(z)} - \frac{\mathbb{Q}_{\hat{Y}}(z)}{1 - D^*(z)} \right] = 0.$$

Hence, we get:

$$D^*(z) = \frac{\mathbb{Q}_Y(z)}{\mathbb{Q}_Y(z) + \mathbb{Q}_{\hat{Y}}(z)} \quad \forall z \in \mathcal{Y} \cup \hat{\mathcal{Y}}. \quad (3.138)$$

Plugging this into (3.137), we obtain:

$$\min_{G(\cdot)} \sum_{z \in \mathcal{Y} \cup \hat{\mathcal{Y}}} \mathbb{Q}_Y(z) \log \frac{\mathbb{Q}_Y(z)}{\mathbb{Q}_Y(z) + \mathbb{Q}_{\hat{Y}}(z)} + \mathbb{Q}_{\hat{Y}}(z) \log \frac{\mathbb{Q}_{\hat{Y}}(z)}{\mathbb{Q}_Y(z) + \mathbb{Q}_{\hat{Y}}(z)}. \quad (3.139)$$

**Jensen-Shannon divergence** Let us massage the objective function in (3.139) to express it as:

$$\min_{G(\cdot)} \underbrace{\sum_{z \in \mathcal{Y} \cup \hat{\mathcal{Y}}} \mathbb{Q}_Y(z) \log \frac{\mathbb{Q}_Y(z)}{\frac{\mathbb{Q}_Y(z) + \mathbb{Q}_{\hat{Y}}(z)}{2}} + \mathbb{Q}_{\hat{Y}}(z) \log \frac{\mathbb{Q}_{\hat{Y}}(z)}{\frac{\mathbb{Q}_Y(z) + \mathbb{Q}_{\hat{Y}}(z)}{2}}}_{-2 \log 2}. \quad (3.140)$$

The above underbraced term can be expressed with a well-known divergence measure in statistics, called *Jensen-Shannon divergence*<sup>6</sup>: for any two distributions, say  $p$  and  $q$ ,

$$\text{JSD}(p, q) := \frac{1}{2} \sum_z p(z) \log \frac{p(z)}{\frac{p(z) + q(z)}{2}} + \frac{1}{2} \sum_z q(z) \log \frac{q(z)}{\frac{p(z) + q(z)}{2}}. \quad (3.141)$$

This is indeed a valid divergence measure, i.e., it is non-negative, being equal to zero if and only if  $p = q$ . We will not prove this here, but you will have a chance to prove it in Prob 9.1.

**Equivalent form** Using the divergence, one can then rewrite the problem (3.140) as:

$$\min_{G(\cdot)} 2 \text{JSD}(\mathbb{Q}_Y, \mathbb{Q}_{\hat{Y}}) - 2 \log 2. \quad (3.142)$$

Hence, we get:

$$G_{\text{GAN}}^* = \arg \min_{G(\cdot)} \text{JSD}(\mathbb{Q}_Y, \mathbb{Q}_{\hat{Y}}). \quad (3.143)$$

We see that this indeed belongs to the original optimization form (3.132) if one makes a choice as:  $D(\cdot, \cdot) = \text{JSD}(\cdot, \cdot)$ .

**Look ahead** So far we have formulated an optimization problem for GANs and made an interesting connection to the Jensen-Shannon divergence. In the next section, we will study how to solve the GAN optimization (3.131) and implement it via TensorFlow.

---

6. One may guess that this is the divergence that Johan Jensen (the inventor of Jensen's inequality) and Claude Shannon (the father of information theory) developed. But it is not the case. Johan Jensen died in 1925 when Claude Shannon was a child, so there was no collaboration between the two. Actually it was invented much later days in 1991 by a Taiwanese information theorist, named Jianhua Lin (Lin, 1991).

### 3.8 GANs: TensorFlow Implementation

**Recap** In the prior section, we investigated Goodfellow's approach to deal with an optimization problem for generative modeling, which in turn led to GANs. He started with quantifying the ability to discriminate real against fake samples:

$$\frac{1}{m} \sum_{i=1}^m \log D(y^{(i)}) + \log(1 - D(\hat{y}^{(i)})) \quad (3.144)$$

where  $y^{(i)}$  and  $\hat{y}^{(i)} := G(x^{(i)})$  indicate real and fake samples, respectively;  $D(\cdot)$  denotes the output of discriminator; and  $m$  is the number of examples. He then introduced two players: (i) player 1, discriminator, who wishes to maximize the ability; (ii) player 2, generator, who wants to minimize it. This naturally led to the optimization problem for GANs:

$$\min_{G(\cdot) \in \mathcal{N}} \max_{D(\cdot) \in \mathcal{N}} \frac{1}{m} \sum_{i=1}^m \log D(y^{(i)}) + \log(1 - D(G(x^{(i)}))) \quad (3.145)$$

where  $\mathcal{N}$  denotes a class of neural network functions. Lastly we demonstrated that the problem (3.145) can be stated in terms of the Jensen-Shannon divergence, thus making a connection to statistics.

Two natural questions arise. First, how to solve the problem (3.145)? Second, how to do programming via a popular deep learning framework TensorFlow?

**Outline** In this section, we will answer these two questions. Specifically we are going to cover four stuffs. First we will investigate a practical method to solve the problem (3.145). We will then do one case study for the purpose of exercising the method. In particular, we will study how to implement an MNIST style handwritten digit image generator. Next we will explore one important implementation detail: Batch Normalization ([Ioffe and Szegedy, 2015](#)), particularly useful for very deep neural networks. Lastly we will learn how to write a TensorFlow script for software implementation.

**Parameterization** Solving the problem (3.145) starts with parameterizing the two functions  $G(\cdot)$  and  $D(\cdot)$  with neural networks:

$$\min_{\textcolor{red}{w}} \max_{\textcolor{blue}{\theta}} \underbrace{\frac{1}{m} \sum_{i=1}^m \log D_{\theta}(y^{(i)}) + \log(1 - D_{\theta}(G_w(x^{(i)})))}_{=:J(w, \theta)} \quad (3.146)$$

where  $w$  and  $\theta$  indicate parameters for  $G(\cdot)$  and  $D(\cdot)$ , respectively. Now the question of interest is: Is the parameterized problem (3.146) the one that we are familiar

with? In other words, is  $J(w, \theta)$  is *convex* in  $w$ ? Is  $J(w, \theta)$  is *concave* in  $\theta$ ? Unfortunately, it is not the case. In general, the objective is highly non-convex in  $w$  and also highly non-concave in  $\theta$ .

Then, what can we do? Actually there is nothing we can do more beyond what we know. We only know how to find a stationary point via a method like gradient descent. So one practical way that we can take is to simply look for a stationary point, say  $(w^*, \theta^*)$ , such that

$$\nabla_w J(w^*, \theta^*) = 0, \quad \nabla_\theta J(w^*, \theta^*) = 0,$$

while cross-fingering that such a point yields a near optimal performance. It turns out luckily it is often the case in reality, especially when employing neural networks for parameterization. There have been huge efforts by many theorists in figuring out why that is the case, e.g., ([Arora et al., 2017](#)). However, a clear theoretical understanding is still missing despite their serious efforts.

**Alternating gradient descent** One practical method to attempt to find (yet not necessarily guarantee to find) such a stationary point in the context of the min-max optimization problem (3.146) is: *alternating gradient descent*. Actually we saw this in Section 2.2. For those who do not remember, let us explain again how it works yet in the context of GANs.

At the  $t$ th iteration, we first update generator's weight:

$$w^{(t+1)} \leftarrow w^{(t)} - \alpha_1 \nabla_w J(w^{(t)}, \theta^{(t)})$$

where  $w^{(t)}$  and  $\theta^{(t)}$  denote the weights of generator and discriminator at the  $t$ th iteration, respectively; and  $\alpha_1$  is the learning rate for generator. Given  $(w^{(t+1)}, \theta^{(t)})$ , we next update discriminator's weight as per:

$$\theta^{(t+1)} \leftarrow \theta^{(t)} + \alpha_2 \nabla_\theta J(w^{(t+1)}, \theta^{(t)})$$

where  $\alpha_2$  is the learning rate for discriminator. Here one important thing to notice is that we should perform gradient *ascent*, i.e., the direction along which the discriminator's weight is updated should be *aligned* with the gradient, reflected in the plus sign colored in blue. Lastly we repeat the above two until converged.

In practice, we may wish to control the frequency of discriminator weight update relative to that of generator. To this end, we often employ  $k : 1$  alternating gradient descent:

1. Update generator's weight:

$$w^{(t+1)} \leftarrow w^{(t)} - \alpha_1 \nabla_w J(w^{(t)}, \theta^{(t,k)}).$$

2. Update discriminator's weight  $k$  times while fixing  $w^{(t+1)}$ : for  $i=1:k$ ,

$$\theta^{(t \cdot k + i)} \leftarrow \theta^{(t \cdot k + i - 1)} + \alpha_2 \nabla_{\theta} J(w^{(t+1)}, \theta^{(t \cdot k + i - 1)}).$$

3. Repeat the above.

You may wonder why we update discriminator more frequently than generator. Usually more updates in the *inner* optimization yield better performances in practice. Further, we often employ the Adam counterpart of the algorithm together with batches. Details are omitted although we will apply such a practical version for programming assignment in Prob 9.4.

**A practical tip on generator** Before moving onto a case study for implementation, let us say a few words about generator optimization. Given discriminator's parameter  $\theta$ : the generator wishes to minimize:

$$\min_w \frac{1}{m_{\mathcal{B}}} \sum_{i \in \mathcal{B}} \log D_{\theta}(y^{(i)}) + \log(1 - D_{\theta}(G_w(x^{(i)})))$$

where  $\mathcal{B}$  indicates a batch of interest and  $m_{\mathcal{B}}$  is the batch size (the number of examples in the interested batch). Notice that  $\log D_{\theta}(y^{(i)})$  in the above is irrelevant of generator's weight  $w$ . Hence, it suffices to minimize the following:

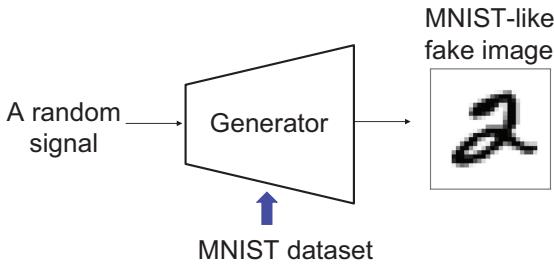
$$\underbrace{\min_w \frac{1}{m_{\mathcal{B}}} \sum_{i \in \mathcal{B}} \log(1 - D_{\theta}(G_w(x^{(i)})))}_{\text{generator loss}}$$

where the underbraced term is called "generator loss". However, in practice, instead of minimizing the generator loss directly, people often rely on the following *proxy*:

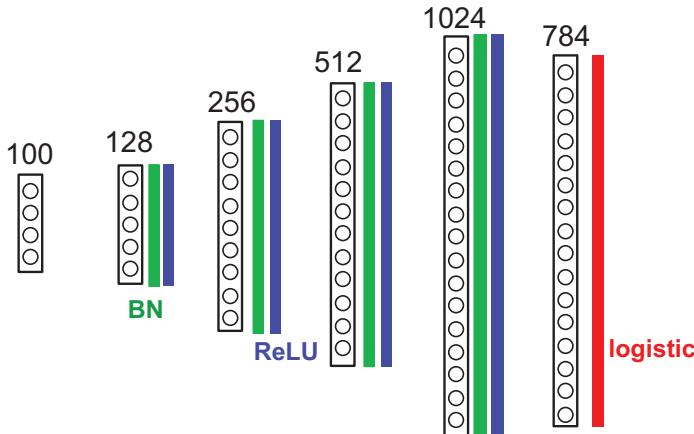
$$\min_w \frac{1}{m_{\mathcal{B}}} \sum_{i \in \mathcal{B}} -\log D_{\theta}(G_w(x^{(i)})). \quad (3.147)$$

You may wonder why. There is a technically detailed rationale behind the use of the proxy for the generator loss. Check this in Prob 9.2.

**Task** Let us discuss one case study for implementation. The task that we will focus on is the one related to the simple digit classifier that we exercised on in Section 3.5. The task is to generate MNIST style handwritten digit images, as illustrated in Fig. 3.35. Here we intend to train generator with MNIST dataset so that it outputs an MNIST style fake image when fed by a random input signal.



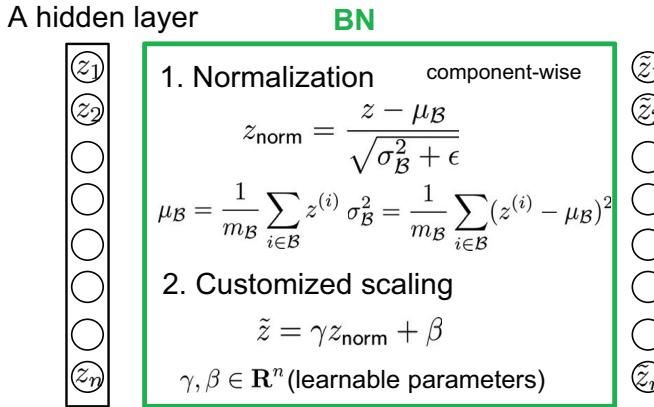
**Figure 3.35.** Generator for MNIST-style handwritten digit images.



**Figure 3.36.** Generator: A 5-layer fully-connected neural network where the input size (the dimension of a latent signal) is 100; the numbers of hidden neurons are 128, 256, 512, 1024; and the output size is 784 ( $= 28 \times 28$ ). We employ ReLU activation for every hidden layer, and logistic activation for the output layer to ensure 0-to-1 output signals. We also use Batch Normalization prior to ReLU at each hidden layer. See Fig. 3.37 for details.

**Model for generator** As a generator model, we employ a 5-layer fully-connected neural network with four hidden layers, as depicted in Fig. 3.36. For activation at each hidden layer, we employ ReLU. Remember that an MNIST image consists of 28-by-28 pixels, each indicating a gray-scaled value that spans from 0 to 1. Hence, for the output layer, we use 784 ( $= 28 \times 28$ ) neurons and logistic activation to ensure the range of  $[0, 1]$ .

The employed network has five layers, so it is deeper than the 2-layer case that we used earlier. In practice, for a somewhat deep neural network, each layer's signals can exhibit quite different scalings. It turns out such dynamically-swung scaling yields a detrimental effect upon training: *unstable* training. So in practice, people often apply an additional procedure (prior to ReLU) so as to control the scaling in our own manner. The procedure is called: Batch Normalization.



**Figure 3.37.** Batch Normalization (BN): First we do zero-centering and normalization with the mean  $\mu_{\mathcal{B}}$  and the variance  $\sigma_{\mathcal{B}}^2$  computed over the examples in an associated batch  $\mathcal{B}$ . Next we do a customized scaling by introducing two new parameters that would also be learned during training:  $\gamma \in \mathbf{R}^n$  and  $\beta \in \mathbf{R}^n$ .

**Batch Normalization (Ioffe and Szegedy, 2015)** Here is how it works; see Fig. 3.37. For illustrative purpose, focus on one particular hidden layer. Let  $z := [z_1, \dots, z_n]^T$  be the output of the considered hidden layer prior to activation. Here  $n$  denotes the number of neurons in the hidden layer.

Batch Normalization (BN for short) consists of two steps. First we do zero-centering and normalization using the mean and variance w.r.t. examples in an associated batch  $\mathcal{B}$ :

$$\mu_{\mathcal{B}} = \frac{1}{m_{\mathcal{B}}} \sum_{i \in \mathcal{B}} z^{(i)}, \quad \sigma_{\mathcal{B}}^2 = \frac{1}{m_{\mathcal{B}}} \sum_{i \in \mathcal{B}} (z^{(i)} - \mu_{\mathcal{B}})^2 \quad (3.148)$$

where  $(\cdot)^2$  indicates a *component-wise* square, and hence  $\sigma_{\mathcal{B}}^2 \in \mathbf{R}^n$ . In other words, we generate the normalized output, say  $z_{\text{norm}}$ , as:

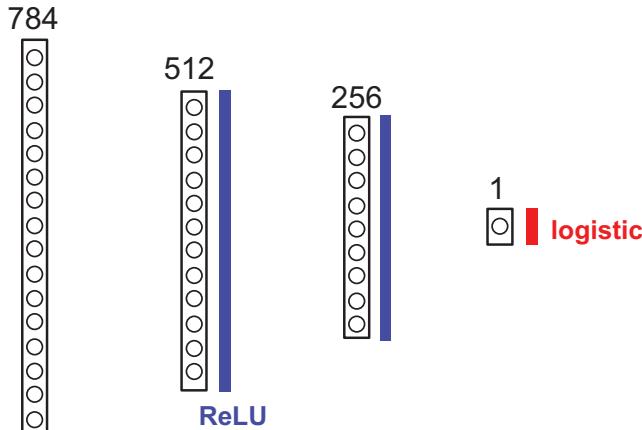
$$z_{\text{norm}} = \frac{z^{(i)} - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \quad (3.149)$$

where division and multiplication are all component-wise. Here  $\epsilon$  is a tiny value introduced to avoid division by 0 (typically  $10^{-5}$ ).

Second, we do a customized scaling as per:

$$\tilde{z}^{(i)} = \gamma z_{\text{norm}}^{(i)} + \beta \quad (3.150)$$

where  $\gamma, \beta \in \mathbf{R}^n$  indicate two new scaling parameters which are learnable via training. Again the operations in (3.150) are all component-wise.



**Figure 3.38.** Discriminator: A 3-layer fully-connected neural network where the input size (the dimension of a flattened vector of a real (or fake) image) is 784 ( $= 28 \times 28$ ); the numbers of hidden neurons are 512, 256; and the output size is 1. We employ ReLU activation for every hidden layer, and logistic activation for the output layer.

BN lets the model learn the optimal scale and mean of the inputs for each hidden layer. This technique is quite instrumental in stabilizing and speeding up training especially for a very deep neural network. This has been verified experimentally by many practitioners, although no clear theoretical justification has been provided thus far.

**Model for discriminator** As a discriminator model, we use a simple 3-layer fully-connected network with two hidden layers; see Fig. 3.38. Here the input size must be the same as that of the flattened real (or fake) image. Again we employ ReLU at hidden layers and logistic activation at the output layer.

**TensorFlow: How to use BN?** Let us talk about how to do TensorFlow programming for implementation. Loading MNIST data is exactly the same as before. So we omit it. Instead let us discuss how to use BN.

As you expect, TensorFlow provides a built-in class for BN:

```
BatchNormalization()
```

This is placed in `tensorflow.keras.layers`. Here is how to use the class in our setting:

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import BatchNormalization
from tensorflow.keras.layers import ReLU

generator = Sequential()
```

```
generator.add(Dense(128,input_dim=latent_dim))
generator.add(BatchNormalization())
generator.add(ReLU())
generator.add(Dense(256))
generator.add(BatchNormalization())
# ...
```

where latent\_dim is the dimension of the latent signal (which we set as 100).

**TensorFlow: Models for generator & discriminator** Using the DNN architectures for generator and discriminator illustrated in Figs. 3.36 and 3.38, we can implement a code as below.

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import BatchNormalization
from tensorflow.keras.layers import ReLU

latent_dim = 100
generator = Sequential()
generator.add(Dense(128, input_dim=latent_dim))
generator.add(BatchNormalization())
generator.add(ReLU())
generator.add(Dense(256))
generator.add(BatchNormalization())
generator.add(ReLU())
generator.add(Dense(512))
generator.add(BatchNormalization())
generator.add(ReLU())
generator.add(Dense(1024))
generator.add(BatchNormalization())
generator.add(ReLU())
generator.add(Dense(28*28, activation='sigmoid'))
```

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import ReLU

discriminator = Sequential()
discriminator.add(Dense(512, input_shape=(784,)))
discriminator.add(ReLU())
discriminator.add(Dense(256))
discriminator.add(ReLU())
discriminator.add(Dense(1, activation='sigmoid'))
```

**TensorFlow: Optimizers for generator & discriminator** We use Adam optimizers with  $\text{lr}=0.0002$  and  $(\text{b1}, \text{b2})=(0.5, 0.999)$ . Since we have two models (generator and discriminator), we employ two optimizers accordingly:

```
from tensorflow.keras.optimizers import Adam
lr = 0.0002
b1 = 0.5
b2 = 0.999 # default choice
optimizer_G = Adam(learning_rate=lr, beta_1=b1)
optimizer_D = Adam(learning_rate=lr, beta_1=b1)
```

**TensorFlow: Generator input** As a generator input, we use a random signal with the *Gaussian* distribution. In particular, we use:

$$x \in \mathbf{R}^{\text{latent\_dim}} \sim \mathcal{N}(\mathbf{0}, \mathbf{I}_{\text{latent\_dim}}).$$

Here is how to generate the Gaussian random signal in TensorFlow:

```
from tensorflow.random import normal
x = normal([batch_size,latent_dim])
```

**TensorFlow: Binary cross entropy loss** Consider the batch version of the GAN optimization (3.146):

$$\min_w \max_{\theta} \frac{1}{m_B} \sum_{i \in \mathcal{B}} \log D_{\theta}(y^{(i)}) + \log(1 - D_{\theta}(G_w(x^{(i)}))). \quad (3.151)$$

Now introduce the ground-truth real-vs-fake indicator vector  $[1, 0]^T$  ( $\text{real}=1$ ,  $\text{fake}=0$ ). Then, the term  $\log D_{\theta}(y^{(i)})$  can be viewed as the minus binary cross entropy between the real/fake indicator vector and its prediction counterpart  $[D_{\theta}(y^{(i)}), 1 - D_{\theta}(y^{(i)})]^T$ :

$$\begin{aligned} \log D_{\theta}(y^{(i)}) &= 1 \cdot \log D_{\theta}(y^{(i)}) + 0 \cdot \log(1 - D_{\theta}(y^{(i)})) \\ &= -\ell_{\text{BCE}}(1, D_{\theta}(y^{(i)})). \end{aligned} \quad (3.152)$$

On the other hand, another term  $\log(1 - D_{\theta}(\hat{y}^{(i)}))$  can be interpreted as the minus binary cross entropy between the fake-vs-real indicator vector ( $\text{fake}=0$ ,  $\text{real}=1$ ) and its prediction counterpart:

$$\begin{aligned} \log(1 - D_{\theta}(\hat{y}^{(i)})) &= 0 \cdot \log D_{\theta}(\hat{y}^{(i)}) + 1 \cdot \log(1 - D_{\theta}(\hat{y}^{(i)})) \\ &= -\ell_{\text{BCE}}(0, D_{\theta}(\hat{y}^{(i)})). \end{aligned} \quad (3.153)$$

From this, we see that cross entropy plays a role in computation of the objective function. TensorFlow offers a built-in class for cross entropy:

`BinaryCrossentropy()`. This is placed in `tensorflow.keras.losses`. Here is how to use it in our setting:

```
from tensorflow.keras.losses import BinaryCrossentropy
CE_loss = BinaryCrossentropy(from_logits=False)
loss = CE_loss(real_fake_indicator, output)
```

where `output` denotes an output of discriminator, and `real_fake_indicator` is real/fake indicator vector (`real=1`, `fake=0`). Here one important thing to notice is that `output` is the result *after* logistic activation; and `real_fake_indicator` is also a *vector* with the same dimension as `output`. The function `BinaryCrossentropy()` automatically detects the number of examples in an associated batch, thus yielding a normalized version (through division by  $m_B$ ).

**TensorFlow: Generator loss** Recall the proxy (3.147) for the generator loss that we will use:

$$\begin{aligned} & \min_w \frac{1}{m_B} \sum_{i \in \mathcal{B}} -\log D_\theta(G_w(x^{(i)})) \\ & \stackrel{(a)}{=} \min_w \frac{1}{m_B} \sum_{i \in \mathcal{B}} \ell_{\text{BCE}}\left(1, D_\theta(G_w(x^{(i)}))\right) \end{aligned} \quad (3.154)$$

where (a) follows from (3.152). Hence, we can use the function `CE_loss` implemented above to easily write a code as below:

```
g_loss = CE_loss(valid, discriminator(gen_imgs))
```

where `gen_imgs` indicate fake images (corresponding to  $G_w(x^{(i)})$ 's) and `valid` denotes an all-1's vector with the same dimension as `gen_imgs`.

**TensorFlow: Discriminator loss** Recall the batch version of the optimization problem:

$$\max_\theta \frac{1}{m_B} \sum_{i \in \mathcal{B}} \log D_\theta(y^{(i)}) + \log(1 - D_\theta(G_w(x^{(i)}))).$$

Taking the minus sign in the objective, we obtain the equivalent minimization optimization:

$$\min_\theta \underbrace{\frac{1}{m_B} \sum_{i \in \mathcal{B}} -\log D_\theta(y^{(i)}) - \log(1 - D_\theta(G_w(x^{(i)})))}_{\text{discriminator loss}}$$



Léon Bottou 2017

**Figure 3.39.** Léon Bottou is the inventor of Wasserstein GANs. He is another big figure in the AI field.

where the discriminator loss is defined as the minus version. Using (3.152) and (3.153), we can implement the discriminator loss as:

```
real_loss = CE_loss(valid, discriminator(real_imgs))
fake_loss = CE_loss(fake, discriminator(gen_imgs))
d_loss = real_loss + fake_loss
```

where `real_imgs` indicate real images (corresponding to  $y^{(i)}$ 's) and `fake` denotes an all-0's vector with the same dimension as `gen_imgs`.

**TensorFlow: Training** Using all of the above, one can implement a code for training. Here we omit details. But you will have a chance to be guided in detail in Prob 9.4.

**Look ahead** We have investigated the GAN optimization problem together with its TensorFlow implementation. While GANs work well in practice, there is one critical issue which we did not delve into. In fact, the issue can be understood from the equivalent form of the GAN optimization that we derived in the previous section:

$$G_{\text{GAN}}^* = \arg \min_{G(\cdot)} \text{JSD}(\mathbb{Q}_Y, \mathbb{Q}_{\hat{Y}}). \quad (3.155)$$

The issue could be figured out by the team of Professor Léon Bottou, a computer scientist as well as a mathematician. See Fig. 3.39 for his portrait. Since he is strong at math and stats, he could understand that the critical issue comes from some undesirable property of JSD. More importantly, he knew how to address the issue. In the course of addressing the issue, he could develop a variant of GAN, which he called ([Arjovsky et al., 2017](#)):

*Wasserstein GAN.*

In the next section, we will figure out what that critical issue is. We will then investigate how Bottou came up with Wasserstein GAN.

## Problem Set 9

---

**Prob 9.1 (Jensen-Shannon divergence)** Recall the Jensen-Shannon divergence that we encountered in Section 3.7. Let  $p$  and  $q$  be two distributions.

(a) Show that

$$\text{JSD}(p, q) = \text{JSD}(q, p). \quad (3.156)$$

(b) Show that

$$\text{JSD}(p, q) \geq 0. \quad (3.157)$$

Also identify conditions under which the equality in (3.157) holds.

**Prob 9.2 (Proxy for generator loss in GAN)** Consider the optimization problem for GAN that we learned in Section 3.7:

$$\min_{G(\cdot) \in \mathcal{N}} \max_{D(\cdot) \in \mathcal{N}} \frac{1}{m} \sum_{i=1}^m \log D(y^{(i)}) + \log(1 - D(\hat{y}^{(i)})) \quad (3.158)$$

where  $\mathcal{N}$  indicates a neural network, and  $y^{(i)}$  and  $\hat{y}^{(i)} := G(x^{(i)})$  denote real and fake samples respectively. Here  $x^{(i)}$  denotes an input to the generator that one can synthesize arbitrarily, and  $m$  is the number of examples. Suppose that the inner optimization is solved to yield  $D^*(\cdot)$ . Then, the optimization problem reduces to:

$$\min_{G(\cdot) \in \mathcal{N}} \frac{1}{m} \sum_{i=1}^m \log D^*(y^{(i)}) + \log(1 - D^*(\hat{y}^{(i)})). \quad (3.159)$$

(a) Show that the optimization problem (3.159) is equivalent to:

$$\min_{G(\cdot) \in \mathcal{N}} \frac{1}{m} \sum_{i=1}^m \log(1 - D^*(\hat{y}^{(i)})). \quad (3.160)$$

(b) Let  $w$  be the weights of the generator model. Show that

$$\frac{d \log(1 - D^*(\hat{y}^{(i)}))}{dw} = \frac{1}{D^*(\hat{y}^{(i)}) - 1} \frac{dD^*(\hat{y}^{(i)})}{d\hat{y}^{(i)}} \frac{d\hat{y}^{(i)}}{dw}, \quad (3.161)$$

$$\frac{d(-\log D^*(\hat{y}^{(i)}))}{dw} = \frac{-1}{D^*(\hat{y}^{(i)})} \frac{dD^*(\hat{y}^{(i)})}{d\hat{y}^{(i)}} \frac{d\hat{y}^{(i)}}{dw}. \quad (3.162)$$

- (c) Suppose that the Discriminator works almost optimally, i.e.,  $D^*(\hat{y}^{(i)})$  is very close to 0. Which is larger in magnitude between (3.161) and (3.162)? Instead of solving (3.160), many researchers including the inventors of GAN prefer to solve the following for  $G(\cdot)$ :

$$\min_{G(\cdot) \in \mathcal{N}} \frac{1}{m} \sum_{i=1}^m -\log D^*(\hat{y}^{(i)}). \quad (3.163)$$

Explain the rationale behind this alternative.

**Prob 9.3 (Batch normalization (Ioffe and Szegedy, 2015))** Consider a deep neural network. Let  $z^{(i)} := [z_1^{(i)}, \dots, z_n^{(i)}]^T$  be the output of a hidden layer prior to activation for the  $i$ th example where  $i \in \{1, 2, \dots, m\}$  and  $m$  is the number of examples. Here  $n$  denotes the number of neurons in the hidden layer.

(a) Let

$$\mu = \frac{1}{m} \sum_{i=1}^m z^{(i)}, \quad \sigma^2 = \frac{1}{m} \sum_{i=1}^m (z^{(i)} - \mu)^2 \quad (3.164)$$

where  $(\cdot)^2$  indicates a *component-wise* square, and hence  $\sigma^2 \in \mathbf{R}^n$ . Consider

$$z_{\text{norm}}^{(i)} = \frac{z^{(i)} - \mu}{\sqrt{\sigma^2 + \epsilon}} \quad (3.165)$$

$$\tilde{z}^{(i)} = \gamma z_{\text{norm}}^{(i)} + \beta \quad (3.166)$$

where  $\gamma, \beta \in \mathbf{R}^n$ . Again the division and multiplication are all component-wise. Here  $\epsilon$  is a tiny value introduced to avoid division by 0 (typically  $10^{-5}$ ). This is called a smoothing term. Assuming that  $\epsilon$  is negligible and  $z^{(i)}$ 's are independent over  $i$ , what are the mean and variance of  $\tilde{z}^{(i)}$ ?

- (b) Many researchers often employ  $\tilde{z}^{(i)}$  instead of  $z^{(i)}$  during training. These operations include zero-centering and normalization (hence it is named *batch normalization*), followed by rescaling and shifting with two new parameters ( $\gamma$  and  $\beta$ ) which are learnable via training. In other words, these operations let the model learn the optimal scale and mean of the inputs for each layer. It turns out this technique plays a significant role in stabilizing and speeding up training especially for a very deep neural network. This has been verified experimentally by many practitioners, but no clear theoretical justification has been provided thus far.

In practice, this operation is done over the current *mini-batch*, so the whole procedure is summarized as follows: for the current mini-batch  $\mathcal{B}$  with the size  $m_{\mathcal{B}}$ ,

$$\begin{aligned}\mu_{\mathcal{B}} &= \frac{1}{m_{\mathcal{B}}} \sum_{i=1}^{m_{\mathcal{B}}} z^{(i)}, \quad \sigma_{\mathcal{B}}^2 = \frac{1}{m_{\mathcal{B}}} \sum_{i=1}^{m_{\mathcal{B}}} (z^{(i)} - \mu_{\mathcal{B}})^2, \\ z_{\text{norm}}^{(i)} &= \frac{z^{(i)} - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}}, \quad \tilde{z}^{(i)} = \gamma z_{\text{norm}}^{(i)} + \beta.\end{aligned}\tag{3.167}$$

At *test* time, there is no mini-batch to compute the empirical mean and standard deviation. What can we do then? Suggest a way to handle this issue and also explain the rationale behind your suggestion. You may want to consult with some well-known literature if you wish.

**Prob 9.4 (TensorFlow implementation of GAN)** Consider the GAN that we learned in Section 3.7. In this problem, you are asked to build a simple GAN that generates MNIST style handwritten digit images. We employ a 5-layer neural network for generator with ReLU in all the hidden layers and logistic activation in the output layer.

- (a) (*MNIST dataset loading*) Use the following script (or otherwise), load the MNIST dataset:

```
from tensorflow.keras.datasets import mnist
(X_train,y_train),(X_test, y_test)=mnist.load_data()
X_train = X_train/255.
X_test = X_test/255.
```

Explain the role of the following script:

```
import numpy as np
def get_batches(data, batch_size):
    batches = []
    for i in range(int(data.shape[0] // batch_size)):
        batch=data[i*batch_size:(i +1)*batch_size]
        batches.append(batch)
    return np.asarray(batches)
```

- (b) (*Data visualization*) Assume that the code in part (a) is executed. Using a skeleton code provided in Prob 8.10(b), write a script that plots 60 images in the first batch of  $X_{\text{train}}$  in one figure. Also plot the figure.

- (c) (*Generator*) Draw a block diagram for generator implemented by the following:

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import BatchNormalization
from tensorflow.keras.layers import ReLU

latent_dim =100
generator=Sequential()
generator.add(Dense(128,input_dim=latent_dim))
generator.add(BatchNormalization())
generator.add(ReLU())
generator.add(Dense(256))
generator.add(BatchNormalization())
generator.add(ReLU())
generator.add(Dense(512))
generator.add(BatchNormalization())
generator.add(ReLU())
generator.add(Dense(1024))
generator.add(BatchNormalization())
generator.add(ReLU())
generator.add(Dense(28*28,activation='sigmoid'))
```

- (d) (*Generator check*) Upon the above codes being executed, report an output for the following:

```
from tensorflow.random import normal
import matplotlib.pyplot as plt

batch_size = 64
x = normal([batch_size,latent_dim])
gen_imgs = generator.predict(x)
gen_imgs = gen_imgs.reshape(-1,28,28)

num_of_images = 60
for index in range(1,num_of_images+1):
    plt.subplot(6,10, index)
    plt.axis('off')
    plt.imshow(gen_imgs[index], cmap = 'gray_r')
```

- (e) (*Discriminator*) Draw a block diagram for discriminator implemented by the following:

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import ReLU
```

```
discriminator=Sequential()
discriminator.add(Dense(512,input_shape=(784,)))
discriminator.add(ReLU())
discriminator.add(Dense(256))
discriminator.add(ReLU())
discriminator.add(Dense(1,activation= 'sigmoid'))
```

(f) (*Training*) Suppose we create the generator and discriminator as follows:

```
from tensorflow.keras.layers import Input
from tensorflow.keras.models import Model
from tensorflow.keras.optimizers import Adam

adam = Adam(learning_rate=0.0002, beta_1=0.5)

# discriminator compile
discriminator.compile(loss='binary_crossentropy',
                      optimizer=adam)
# fix disc's weights while training generator
discriminator.trainable = False

# define GAN with fake input and disc. output
gan_input = Input(shape=(latent_dim,))
x = generator(inputs=gan_input)
output = discriminator(x)
gan = Model(gan_input, output)
gan.compile(loss='binary_crossentropy', optimizer=adam)
```

where generator() and discriminator() are the classes designed in parts (c) and (e), respectively.

Now explain how generator and discriminator are trained in the following code:

```
import numpy as np
from tensorflow.random import normal

EPOCHS = 50
k=2 # k:1 alternating gradient descent
d_losses = []
g_losses = []

for epoch in range(1,EPOCHS + 1):
    # train per each batch
    np.random.shuffle(X_train)
    for i, real_imgs in enumerate(get_batches(X_train, batch_size)):
```

```

#####
# train discriminator
#####
# fake input generation
gen_input = normal([batch_size,latent_dim])
# fake images
gen_imgs = generator.predict(gen_input)
real_imgs = real_imgs.reshape(-1,28*28)
# input for discriminator
d_input = np.concatenate([real_imgs,gen_imgs])
# label for discriminator
# (first half: real (1); second half: fake (0))
d_label = np.zeros(2*batch_size)
d_label[:batch_size] = 1
# train Discriminator
d_loss = discriminator.train_on_batch(d_input, d_label)

#####
# train generator
#####
if i%k: # 1:k alternating gradient descent
    # fake input generation
    g_input = normal([batch_size,latent_dim])
    # label for fake image
    # Generator wants fake images to be treated
    # as real ones
    g_label = np.ones(batch_size)
    # train generator
    g_loss = gan.train_on_batch(g_input, g_label)

d_losses.append(d_loss)
g_losses.append(g_loss)

```

- (g) (*Training check*) For epoch= 10, 30, 50, 70, 90: plot a figure that shows 25 fake images from generator trained in part (f) or by other methods of yours. Also plot the generator loss and discriminator loss as a function of epochs. Include Python scripts as well.

**Prob 9.5 (Minimax theorem)** Let  $f(x, y)$  be a continuous real-valued function defined on  $\mathcal{X} \times \mathcal{Y}$  such that

- (i)  $f(x, y)$  is *convex* in  $x \in \mathcal{X} \forall y \in \mathcal{Y}$ ; and
- (ii)  $f(x, y)$  is *concave* in  $y \in \mathcal{Y} \forall x \in \mathcal{X}$

where  $\mathcal{X}$  and  $\mathcal{Y}$  are convex and compact sets.

*Note:* You do not need to solve the *optional* problems below.

(a) Show that

$$\min_{x \in \mathcal{X}} \max_{y \in \mathcal{Y}} f(x, y) \geq \max_{y \in \mathcal{Y}} \min_{x \in \mathcal{X}} f(x, y). \quad (3.168)$$

Does (3.168) hold also for any arbitrary function  $f(\cdot, \cdot)$ ?

(b) Suppose

$$\alpha \leq \min_{x \in \mathcal{X}} \max_{y \in \mathcal{Y}} f(x, y) \implies \alpha \leq \max_{y \in \mathcal{Y}} \min_{x \in \mathcal{X}} f(x, y). \quad (3.169)$$

Then, argue that (3.169) implies:

$$\min_{x \in \mathcal{X}} \max_{y \in \mathcal{Y}} f(x, y) \leq \max_{y \in \mathcal{Y}} \min_{x \in \mathcal{X}} f(x, y). \quad (3.170)$$

(c) Suppose that  $\alpha \leq \min_{x \in \mathcal{X}} \max_{y \in \mathcal{Y}} f(x, y)$ . Then, show that there are finite  $y_1, \dots, y_n \in \mathcal{Y}$  such that

$$\alpha \leq \min_{x \in \mathcal{X}} \max_{y \in \{y_1, \dots, y_n\}} f(x, y). \quad (3.171)$$

(d) (*Optional*) Suppose that  $\alpha \leq \min_{x \in \mathcal{X}} \max_{y \in \{y_1, y_2\}} f(x, y)$  for any  $y_1, y_2 \in \mathcal{Y}$ . Then, show that there exists  $y_0 \in \mathcal{Y}$  such that

$$\alpha \leq \min_{x \in \mathcal{X}} f(x, y_0). \quad (3.172)$$

(e) (*Optional*) Suppose that  $\alpha \leq \min_{x \in \mathcal{X}} \max_{y \in \{y_1, \dots, y_n\}} f(x, y)$  for any finite  $y_1, \dots, y_n \in \mathcal{Y}$ . Then, show that there exists  $y_0 \in \mathcal{Y}$  such that

$$\alpha \leq \min_{x \in \mathcal{X}} f(x, y_0). \quad (3.173)$$

*Hint:* Use the proof-by-induction and part (d).

*Note:* (3.173) implies that  $\alpha \leq \max_{y \in \mathcal{Y}} \min_{x \in \mathcal{X}} f(x, y)$ . This together with the results in parts (b) and (c) proves (3.170). Combining this with (3.168) proves the *minimax theorem*:

$$\min_{x \in \mathcal{X}} \max_{y \in \mathcal{Y}} f(x, y) = \max_{y \in \mathcal{Y}} \min_{x \in \mathcal{X}} f(x, y). \quad (3.174)$$

**Prob 9.6 (Training instability)** Consider a function:

$$f(x, y) = (2 + \cos x)(2 + \cos y) \quad (3.175)$$

where  $x, y \in \mathbf{R}$ .

- (a) Solve the following optimization (i.e., find the optimal solution as well as the points that achieve it):

$$\min_x \max_y f(x, y). \quad (3.176)$$

- (b) Solve the reverse version of the optimization:

$$\max_y \min_x f(x, y). \quad (3.177)$$

- (c) Suppose that we perform 1 : 1 alternating gradient descent for  $f(x, y)$  with an initial point  $(x^{(0)}, y^{(0)}) = (\pi + 0.1, -0.1)$ . Plot  $f(x^{(t)}, y^{(t)})$  as a function of  $t$  where  $(x^{(t)}, y^{(t)})$  denotes the estimate at the  $t$ th iteration. What are the limiting values of  $(x^{(t)}, y^{(t)})$ ? Also explain why.

*Note: You may want to set the learning rates properly so that the convergence behaviour is clear.*

- (d) Redo part (c) with a different initial point  $(x^{(0)}, y^{(0)}) = (0.1, \pi - 0.1)$ .

**Prob 9.7 (Alternating gradient descent)** Consider a function:

$$f(x, y) = x^2 - y^2 \quad (3.178)$$

where  $x, y \in \mathbf{R}$ .

- (a) Solve the following optimization:

$$\min_x \max_y f(x, y). \quad (3.179)$$

- (b) Suppose that we perform 1 : 1 alternating gradient descent for  $f(x, y)$  with an initial point  $(x^{(0)}, y^{(0)}) = (1, 1)$ . Plot  $f(x^{(t)}, y^{(t)})$  as a function of  $t$  where  $(x^{(t)}, y^{(t)})$  denotes the estimate at the  $t$ th iteration. What are the limiting values of  $(x^{(t)}, y^{(t)})$ ? Also explain why.

*Note: You may want to set the learning rates properly so that the convergence behaviour is clear.*

- (c) Redo part (c) with a different initial point  $(x^{(0)}, y^{(0)}) = (-1, -1)$ .

**Prob 9.8 (True or False?)**

- (a) Consider the following optimization:

$$\min_{x \in \mathbf{R}} \max_{y \in \mathbf{R}} x^2 - y^2.$$

With 1:1 alternating gradient descent with a proper choice of the learning rates, one can achieve the optimal solution to the above.

- (b) Consider the following optimization:

$$\min_{x \in \mathbf{R}} \max_{y \in \mathbf{R}} (2 + \cos x)(2 + \cos y).$$

Suppose we perform 1:1 alternating gradient descent with a proper choice of the learning rates. Then, the converging points can be distinct depending on different initial points.

- (c) Autoencoder can be categorized as a *feature learning* method.

### 3.9 Wasserstein GAN I

**Recap** During a couple of past sections, we formulated the optimization problem for GANs. Given data  $\{y^{(i)}\}_{i=1}^m$ ,

$$\min_{G(\cdot) \in \mathcal{N}} \max_{D(\cdot) \in \mathcal{N}} \frac{1}{m} \sum_{i=1}^m \log D(y^{(i)}) + \log(1 - D(\hat{y}^{(i)})) \quad (3.180)$$

where  $G(\cdot)$  and  $D(\cdot)$  indicate the functions of the generator and the discriminator, respectively; and  $\mathcal{N}$  denotes a set of DNN-based functions. We then showed that the GAN optimization belongs to a generic divergence-based optimization problem (an age-old problem in statistics), assuming that DNNs can represent any arbitrary function.

At the end of the previous section, however, we claimed that there is a critical issue in the GAN optimization, and this is what Léon Bottou figured out. We also claimed that in the course of addressing the issue, Bottou came up with a variant of GANs, which he named: Wasserstein GAN (WGAN for short).

**Outline** Supporting these claims form the contents of this section. We will cover the following three stuffs. First, we will figure out what the critical issue is. We will then investigate how Bottou addressed the issue. Lastly we will discuss how Bottou's way leads to the optimization problem for WGAN.

**What is the critical issue that arises in the GAN optimization?** Recall in the GAN optimization that the optimal generator  $G^*(\cdot)$  reads:

$$G^* = \arg \min_{G(\cdot)} \text{JSD}(\mathbb{Q}_Y, \mathbb{Q}_{\hat{Y}}) \quad (3.181)$$

where  $\mathbb{Q}_Y$  and  $\mathbb{Q}_{\hat{Y}}$  indicate the empirical distributions of real samples  $Y \in \{y^{(1)}, \dots, y^{(m)}\} =: \mathcal{Y}$  and fake samples  $\hat{Y} \in \{\hat{y}^{(1)}, \dots, \hat{y}^{(m)}\} =: \hat{\mathcal{Y}}$  respectively, and

$$\begin{aligned} & \text{JSD}(\mathbb{Q}_Y, \mathbb{Q}_{\hat{Y}}) \\ &= \frac{1}{2} \sum_{z \in \mathcal{Y} \cup \hat{\mathcal{Y}}} \mathbb{Q}_Y(z) \log \frac{\mathbb{Q}_Y(z)}{\frac{\mathbb{Q}_Y(z) + \mathbb{Q}_{\hat{Y}}(z)}{2}} + \mathbb{Q}_{\hat{Y}}(z) \log \frac{\mathbb{Q}_{\hat{Y}}(z)}{\frac{\mathbb{Q}_Y(z) + \mathbb{Q}_{\hat{Y}}(z)}{2}}. \end{aligned} \quad (3.182)$$

Here  $z$  indicates a dummy variable which takes an element either from  $\mathcal{Y}$  or from  $\hat{\mathcal{Y}}$ . One key observation is that in almost all practically-relevant settings, fake and real samples are *different* with each other:

$$\mathcal{Y} \cap \hat{\mathcal{Y}} = \emptyset. \quad (3.183)$$

Why? Consider an image data setting in which the dimension of data is usually very large, e.g., the dimension of an image in ImageNet is:  $227 \times 227 \times 3 = 154,587$ . In this setting, the probability that any fake image is exactly the same as one of the real images is almost 0, so it is a *measure-zero event*. This (3.183) together with the definitions of  $\mathbb{Q}_Y$  and  $\mathbb{Q}_{\hat{Y}}$  (that we established in Section 3.7) then yields:

$$\begin{aligned} \frac{\mathbb{Q}_Y(z)}{\frac{\mathbb{Q}_Y(z)+\mathbb{Q}_{\hat{Y}}(z)}{2}} &= \begin{cases} \frac{\frac{1}{m}}{\frac{1}{m}+0} = 2, & \text{if } z \in \mathcal{Y}; \\ \frac{0}{\frac{1}{m}+0} = 0, & \text{if } z \in \hat{\mathcal{Y}}; \end{cases} \\ \frac{\mathbb{Q}_{\hat{Y}}(z)}{\frac{\mathbb{Q}_Y(z)+\mathbb{Q}_{\hat{Y}}(z)}{2}} &= \begin{cases} \frac{0}{\frac{1}{m}+0} = 0, & \text{if } z \in \mathcal{Y}; \\ \frac{\frac{1}{m}}{0+\frac{1}{m}} = 2, & \text{if } z \in \hat{\mathcal{Y}}. \end{cases} \end{aligned} \quad (3.184)$$

Plugging this into (3.182), we get:

$$\begin{aligned} &\text{JSD}(\mathbb{Q}_Y, \mathbb{Q}_{\hat{Y}}) \\ &= \frac{1}{2} \sum_{z \in \mathcal{Y}} \mathbb{Q}_Y(z) \log \frac{\mathbb{Q}_Y(z)}{\frac{\mathbb{Q}_Y(z)+\mathbb{Q}_{\hat{Y}}(z)}{2}} + \frac{1}{2} \sum_{z \in \hat{\mathcal{Y}}} \mathbb{Q}_{\hat{Y}}(z) \log \frac{\mathbb{Q}_{\hat{Y}}(z)}{\frac{\mathbb{Q}_Y(z)+\mathbb{Q}_{\hat{Y}}(z)}{2}} \\ &= \frac{1}{2} \sum_{z \in \mathcal{Y}} \mathbb{Q}_Y(z) \log 2 + \frac{1}{2} \sum_{z \in \hat{\mathcal{Y}}} \mathbb{Q}_{\hat{Y}}(z) \log 2 \\ &= \log 2 \end{aligned} \quad (3.185)$$

where the last equality comes from  $\sum_{z \in \mathcal{Y}} \mathbb{Q}_Y(z) = 1$  and  $\sum_{z \in \hat{\mathcal{Y}}} \mathbb{Q}_{\hat{Y}}(z) = 1$ .

From (3.185), we can now see the critical issue:

$\text{JSD}(\mathbb{Q}_Y, \mathbb{Q}_{\hat{Y}})$  is *irrelevant* of how we choose  $G(\cdot)$ ,

meaning that

$$G^* = \arg \min_{G(\cdot)} \text{JSD}(\mathbb{Q}_Y, \mathbb{Q}_{\hat{Y}}) = \arg \min_{G(\cdot)} \log 2 \quad \text{could be anything.} \quad (3.186)$$

This implies that we may arrive at a stupid solution from the JSD-based optimization (3.181), since any  $G(\cdot)$  can be optimal.

Here you may see that something weird is happening. Why? We already knew that GANs are working well in practice. This suggests that the phenomena observed by many researchers look inconsistent with the theory due to the above simple derivation (3.186). Any mistake in the above derivation? Or something wrong in simulations done by many practitioners? Or something else? It turns out the answer

is “something else”. Remember in the GAN optimization (3.180) that  $G(\cdot)$  and  $D(\cdot)$  should be *DNN-based functions*, not arbitrary functions. So precisely speaking, the optimal generator should read:

$$G_{\text{GAN}}^* := \arg \min_{G(\cdot) \in \mathcal{N}} \max_{D(\cdot) \in \mathcal{N}} \frac{1}{m} \sum_{i=1}^m \log D(y^{(i)}) + \log(1 - D(\hat{y}^{(i)})).$$

In practice, DNN is not perfectly expressive, and hence:

$$G_{\text{GAN}}^* \neq G^*.$$

This is the reason why there is inconsistency between the theory and the practice. There are some groups of people (including Prof. Sanjeev Arora at Princeton, a theoretical computer scientist) who have been investigating why the GANs with DNN-function constraints lead to good performances (Arora *et al.*, 2017). Nonetheless, we have no clear understanding on this.

**Motivated the use of the Wasserstein distance** The critical issue, reflected in (3.186), motivated Bottou to reconsider the generic divergence-based optimization problem:

$$\min_{G(\cdot)} D(\mathbb{Q}_Y, \mathbb{Q}_{\hat{Y}}) \quad (3.187)$$

where  $D(\cdot, \cdot)$  is of our design choice. He knew that there are some good divergence measures which do not yield the critical issue (3.186). One of the measures that he chose was the 1st order Wasserstein distance that we studied in Part I. This led him to obtain:

$$\min_{G(\cdot)} W(\mathbb{Q}_Y, \mathbb{Q}_{\hat{Y}}) \quad (3.188)$$

where

$$\begin{aligned} W(\mathbb{Q}_Y, \mathbb{Q}_{\hat{Y}}) &= \min_{\mathbb{Q}_{Y, \hat{Y}}} \mathbb{E}[\|Y - \hat{Y}\|] \\ &= \min_{\mathbb{Q}_{Y, \hat{Y}}} \sum_{i=1}^m \sum_{j=1}^m \mathbb{Q}_{Y, \hat{Y}}(y^{(i)}, \hat{y}^{(j)}) \|y^{(i)} - \hat{y}^{(j)}\|. \end{aligned} \quad (3.189)$$

Notice that  $\|y^{(i)} - \hat{y}^{(j)}\|$  placed inside the doubled summation (marked in blue) depends on the values of  $\{\hat{y}^{(i)}\}_{i=1}^m$  themselves. Hence, we can readily see that the objective is indeed a function of  $G(\cdot)$  which directly controls  $\{\hat{y}^{(i)}\}_{i=1}^m$ .

**How to solve the Wasserstein-distance-based optimization?** Replacing  $D(\cdot, \cdot)$  with the Wasserstein distance in (3.187), we can rewrite the optimization problem (3.188) as:

$$\begin{aligned} & \min_{G(\cdot)} \min_{\mathbb{Q}_{Y, \hat{Y}}} \sum_{i=1}^m \sum_{j=1}^m \mathbb{Q}_{Y, \hat{Y}}(y^{(i)}, \hat{y}^{(j)}) \|y^{(i)} - \hat{y}^{(j)}\| : \\ & \quad \sum_{j=1}^m \mathbb{Q}_{Y, \hat{Y}}(y^{(i)}, \hat{y}^{(j)}) = \mathbb{Q}_Y(y^{(i)}), \quad i \in \{1, \dots, m\}; \\ & \quad \sum_{i=1}^m \mathbb{Q}_{Y, \hat{Y}}(y^{(i)}, \hat{y}^{(j)}) = \mathbb{Q}_{\hat{Y}}(\hat{y}^{(j)}), \quad j \in \{1, \dots, m\}. \end{aligned} \quad (3.190)$$

Consider the inner optimization problem in (3.190). This is the problem that we are familiar with. That is an LP. We know how to solve an LP using the simplex algorithm. Then, no problem? Unfortunately, that is not the case. There are some issues in solving the above problem. We encounter two major issues.

First, there is no closed-form solution for LP, so this gives a challenge in finding  $G^*(\cdot)$  in the end. The second issue is a more critical one. Notice in practice that the number of optimization variables, which are  $m^2$  of  $\mathbb{Q}_{Y, \hat{Y}}(y^{(i)}, \hat{y}^{(j)})$ , in the inner problem is huge. In the big data era,  $m$  is typically an order of more than thousands or million, or even billion. So  $m^2$  is typically a huge number. Even if we use a fast algorithm, like the simplex algorithm, it would take long time. So it is computationally expensive.

Bottou recognized the issues. More importantly, he knew how to address them. The idea is to rely on the father of LP: Kantorovich. In fact, Kantorovich already established the strong duality theorem for the Wasserstein-distance-based LP, called Kantorovich duality or Kantorovich-Rubinstein duality ([Villani, 2009](#)). What Kantorovich showed is that the dual problem of the Wasserstein-distance-based LP yields exactly the same solution as that of the primal problem,<sup>7</sup> and more importantly, the dual problem is computationally much more efficient. So he simply applied Kantorovich duality to come up with an optimization problem, which is now known as the WGAN optimization. We will describe Kantorovich duality in detail below.

---

<sup>7</sup> This is what we already know. But at that time, the strong duality theorem for convex optimization was not established yet. In fact, Kantorovich duality formed the basis of the strong duality theorem for generic convex problems.

**Notational simplification** Let us consider the inner optimization problem in (3.190). For notational simplification, we employ the dummy variable  $z$  that we introduced earlier:  $z \in \mathcal{Y} \cup \hat{\mathcal{Y}}$ . Let us use  $z$  and  $\hat{z}$  to indicate  $y^{(i)}$  and  $\hat{y}^{(j)}$ , respectively. Using this notation, we can then rewrite the inner optimization problem as:

$$\begin{aligned} \min_{\mathbb{Q}_{Y,\hat{Y}}} & \sum_{z \in \mathcal{Y} \cup \hat{\mathcal{Y}}} \sum_{\hat{z} \in \mathcal{Y} \cup \hat{\mathcal{Y}}} \mathbb{Q}_{Y,\hat{Y}}(z, \hat{z}) \|z - \hat{z}\| : \\ & \sum_{\hat{z} \in \mathcal{Y} \cup \hat{\mathcal{Y}}} \mathbb{Q}_{Y,\hat{Y}}(z, \hat{z}) = \mathbb{Q}_Y(z) \quad \forall z \in \mathcal{Y} \cup \hat{\mathcal{Y}}; \\ & \sum_{z \in \mathcal{Y} \cup \hat{\mathcal{Y}}} \mathbb{Q}_{Y,\hat{Y}}(z, \hat{z}) = \mathbb{Q}_{\hat{Y}}(\hat{z}) \quad \forall \hat{z} \in \mathcal{Y} \cup \hat{\mathcal{Y}}. \end{aligned} \quad (3.191)$$

For further notational simplification, let

$$x(z, \hat{z}) := \mathbb{Q}_{Y,\hat{Y}}(z, \hat{z}) \geq 0. \quad (3.192)$$

Then, the problem can be rewritten as:

$$\begin{aligned} \min_{x(z,\hat{z})} & \sum_{z \in \mathcal{Y} \cup \hat{\mathcal{Y}}} \sum_{\hat{z} \in \mathcal{Y} \cup \hat{\mathcal{Y}}} \|z - \hat{z}\| x(z, \hat{z}) : \\ & -x(z, \hat{z}) \leq 0, \quad \forall z, \hat{z} \in \mathcal{Y} \cup \hat{\mathcal{Y}}; \\ & \sum_{\hat{z} \in \mathcal{Y} \cup \hat{\mathcal{Y}}} x(z, \hat{z}) = \mathbb{Q}_Y(z) \quad \forall z \in \mathcal{Y} \cup \hat{\mathcal{Y}}; \\ & \sum_{z \in \mathcal{Y} \cup \hat{\mathcal{Y}}} x(z, \hat{z}) = \mathbb{Q}_{\hat{Y}}(\hat{z}) \quad \forall \hat{z} \in \mathcal{Y} \cup \hat{\mathcal{Y}}. \end{aligned} \quad (3.193)$$

**Lagrange function, dual function & dual problem** In an effort to derive the dual problem, we first consider the Lagrange function:

$$\begin{aligned} \mathcal{L}(x, \lambda, \nu, \mu) = & \sum_{z \in \mathcal{Y} \cup \hat{\mathcal{Y}}} \sum_{\hat{z} \in \mathcal{Y} \cup \hat{\mathcal{Y}}} \|z - \hat{z}\| x(z, \hat{z}) \\ & - \sum_{z \in \mathcal{Y} \cup \hat{\mathcal{Y}}} \sum_{\hat{z} \in \mathcal{Y} \cup \hat{\mathcal{Y}}} \lambda(z, \hat{z}) x(z, \hat{z}) \\ & + \sum_{z \in \mathcal{Y} \cup \hat{\mathcal{Y}}} \nu(z) \left( \mathbb{Q}_Y(z) - \sum_{\hat{z} \in \mathcal{Y} \cup \hat{\mathcal{Y}}} x(z, \hat{z}) \right) \end{aligned}$$

$$\begin{aligned}
& + \sum_{\hat{z} \in \mathcal{Y} \cup \hat{\mathcal{Y}}} \mu(\hat{z}) \left( \mathbb{Q}_{\hat{\mathcal{Y}}}(\hat{z}) - \sum_{z \in \mathcal{Y} \cup \hat{\mathcal{Y}}} x(z, \hat{z}) \right) \\
& = \sum_{z \in \mathcal{Y} \cup \hat{\mathcal{Y}}} \sum_{\hat{z} \in \mathcal{Y} \cup \hat{\mathcal{Y}}} (\|z - \hat{z}\| - \lambda(z, \hat{z}) - v(z) - \mu(\hat{z})) x(z, \hat{z}) \\
& + \sum_{z \in \mathcal{Y} \cup \hat{\mathcal{Y}}} v(z) \mathbb{Q}_Y(z) + \sum_{\hat{z} \in \mathcal{Y} \cup \hat{\mathcal{Y}}} \mu(\hat{z}) \mathbb{Q}_{\hat{\mathcal{Y}}}(\hat{z})
\end{aligned} \tag{3.194}$$

where  $(\lambda, v, \mu)$  are Lagrange multipliers. Notice that the multiplication factors associated with  $x(z, \hat{z})$ 's in the double summation term in the above last equation (marked in red) should be zeros:

$$\|z - \hat{z}\| - \lambda(z, \hat{z}) - v(z) - \mu(\hat{z}) = 0 \quad \forall z, \hat{z} \in \mathcal{Y} \cup \hat{\mathcal{Y}}. \tag{3.195}$$

Otherwise, one can set  $x(z, \hat{z}) = \infty$  (or  $-\infty$ ) depending on the sign of a non-zero such term while setting  $x(z, \hat{z}) = 0$  for the other terms. This then yields  $\mathcal{L}(x, \lambda, v, \mu) = -\infty$ , and hence  $g(\lambda, v, \mu) = -\infty$ . Obviously this is not an interested case. Hence, applying (3.195), we derive the dual function as:

$$g(\lambda, v, \mu) = \sum_{z \in \mathcal{Y} \cup \hat{\mathcal{Y}}} v(z) \mathbb{Q}_Y(z) + \sum_{\hat{z} \in \mathcal{Y} \cup \hat{\mathcal{Y}}} \mu(\hat{z}) \mathbb{Q}_{\hat{\mathcal{Y}}}(\hat{z}). \tag{3.196}$$

Now notice that  $\lambda(z, \hat{z}) \geq 0$  is a constraint that appears in the dual problem. This together with (3.195) then yields:

$$\|z - \hat{z}\| - v(z) - \mu(\hat{z}) = \lambda(z, \hat{z}) \geq 0 \quad \forall z, \hat{z} \in \mathcal{Y} \cup \hat{\mathcal{Y}}. \tag{3.197}$$

Using this, we can formulate the dual problem as:

$$\begin{aligned}
d^* := \max_{v, \mu} & \sum_{z \in \mathcal{Y} \cup \hat{\mathcal{Y}}} v(z) \mathbb{Q}_Y(z) + \sum_{\hat{z} \in \mathcal{Y} \cup \hat{\mathcal{Y}}} \mu(\hat{z}) \mathbb{Q}_{\hat{\mathcal{Y}}}(\hat{z}) : \\
& v(z) + \mu(\hat{z}) \leq \|z - \hat{z}\| \quad \forall z, \hat{z} \in \mathcal{Y} \cup \hat{\mathcal{Y}}.
\end{aligned} \tag{3.198}$$

**How to deal with two functions in the optimization?** How to solve the dual problem (3.198)? Actually it is not that simple. One may think of the following native approach: Searching for all the possible functions of  $v(\cdot)$  and  $\mu(\cdot)$  in finding the maximum.

Kantorovich did not take this approach. Instead he came up with a very interesting and smart idea. The idea is to translate the problem (3.198) with *two* functions ( $v(\cdot)$  and  $\mu(\cdot)$ ) that one can control over, into an equivalent problem but with *only*

*one* function, say  $\psi(\cdot)$ . It turns out the idea led Kantorovich to come up with the following equivalent problem:

$$\begin{aligned} d^{**} := \max_{\psi} \sum_{z \in \mathcal{Y} \cup \hat{\mathcal{Y}}} \mathbb{Q}_Y(z) \psi(z) - \sum_{\hat{z} \in \mathcal{Y} \cup \hat{\mathcal{Y}}} \mathbb{Q}_{\hat{Y}}(\hat{z}) \psi(\hat{z}) : \\ |\psi(z) - \psi(\hat{z})| \leq \|z - \hat{z}\| \quad \forall z, \hat{z} \in \mathcal{Y} \cup \hat{\mathcal{Y}}. \end{aligned} \quad (3.199)$$

Notice in the translated problem that we have only one function to optimize over, so the complexity is significantly reduced.

Relying on the proof of  $d^* = d^{**}$  together with the outer optimization problem (w.r.t.  $G(\cdot)$ ), Bottou was able to derive a simpler form of an optimization problem, which is now known as the WGAN optimization.

**Look ahead** In the next section, we will prove that  $d^*$  is indeed  $d^{**}$ . We will then demonstrate that this proof leads to the WGAN optimization.

### 3.10 Wasserstein GAN II

**Recap** In the previous section, we figured out there is a critical issue in GANs:  $\text{JSD}(\mathbb{Q}_Y, \mathbb{Q}_{\hat{Y}})$  is *irrelevant* of  $G(\cdot)$ , which in turn suggests that the optimal  $G^*$  could be anything – this is definitely not what we want. In an effort to address this issue, we considered the 1st-order Wasserstein distance which does not have the undesirable property:

$$\min_{G(\cdot)} W(\mathbb{Q}_Y, \mathbb{Q}_{\hat{Y}}) \quad (3.200)$$

where  $\mathbb{Q}_Y$  and  $\mathbb{Q}_{\hat{Y}}$  indicate the empirical distributions of real samples  $Y \in \{y^{(1)}, \dots, y^{(m)}\} =: \mathcal{Y}$  and fake samples  $\hat{Y} \in \{\hat{y}^{(1)}, \dots, \hat{y}^{(m)}\} =: \hat{\mathcal{Y}}$ , respectively. We then checked that the objective function in (3.200) is a sensitive function of  $G(\cdot)$ .

Since the inner optimization in (3.200) involves so many optimization variables (whose number scales at  $m^2$ ) and hence is not computationally tractable (which is often the case in the current big data era), we started considering the dual problem which is known to be computationally tractable due to Kantorovich duality. Applying a bunch of dual problem tricks together with introducing  $(z, \hat{z})$  notations and the extended set  $\bar{\mathcal{Y}} := \mathcal{Y} \cup \hat{\mathcal{Y}}$  notation, we expressed the dual problem as:

$$\begin{aligned} d^* := \max_{v, \mu} \sum_{z \in \mathcal{Y} \cup \hat{\mathcal{Y}}} \mathbb{Q}_Y(z)v(z) + \sum_{\hat{z} \in \mathcal{Y} \cup \hat{\mathcal{Y}}} \mathbb{Q}_{\hat{Y}}(\hat{z})\mu(\hat{z}) : \\ v(z) + \mu(\hat{z}) \leq \|z - \hat{z}\| \quad \forall z, \hat{z} \in \mathcal{Y} \cup \hat{\mathcal{Y}} \end{aligned} \quad (3.201)$$

where  $v(z)$  and  $\mu(z)$  denote Lagrange multipliers w.r.t. the marginal-distribution-associated equality constraints, one for  $\mathbb{Q}_Y$  and the other for  $\mathbb{Q}_{\hat{Y}}$ .

At the end of the last section, we claimed that the above problem is equivalent to the following simpler optimization problem containing only one function  $\psi(\cdot)$ :

$$\begin{aligned} d^{**} := \max_{\psi} \sum_{z \in \mathcal{Y} \cup \hat{\mathcal{Y}}} \mathbb{Q}_Y(z)\psi(z) - \sum_{\hat{z} \in \mathcal{Y} \cup \hat{\mathcal{Y}}} \mathbb{Q}_{\hat{Y}}(\hat{z})\psi(\hat{z}) : \\ |\psi(z) - \psi(\hat{z})| \leq \|z - \hat{z}\| \quad \forall z, \hat{z} \in \mathcal{Y} \cup \hat{\mathcal{Y}}. \end{aligned} \quad (3.202)$$

Also we mentioned that this simpler optimization leads to the WGAN optimization.

**Outline** In this section, we will prove the above claim and then derive the WGAN optimization accordingly. What we are going to cover are three folded. We will first prove  $d^* = d^{**}$ . We will then use the claim to derive an optimization for WGAN.

Finally we will discuss on the optimality of the Wasserstein distance for divergence-measure based optimization problems.

**Proof of  $d^* \geq d^{**}$**  We will show the following two inequalities:  $d^{**} \leq d^*$  and  $d^* \leq d^{**}$ . First let us prove the former. Consider:

$$\begin{aligned} d^{**} &:= \max_{\psi} \sum_{z \in \mathcal{Y} \cup \hat{\mathcal{Y}}} \mathbb{Q}_Y(z)\psi(z) - \sum_{\hat{z} \in \mathcal{Y} \cup \hat{\mathcal{Y}}} \mathbb{Q}_{\hat{Y}}(\hat{z})\psi(\hat{z}) \\ &\leq \max_{\psi} \max_{\mu} \sum_{z \in \mathcal{Y} \cup \hat{\mathcal{Y}}} \mathbb{Q}_Y(z)\psi(z) + \sum_{\hat{z} \in \mathcal{Y} \cup \hat{\mathcal{Y}}} \mathbb{Q}_{\hat{Y}}(\hat{z})\mu(\hat{z}) \\ &= \max_{v} \max_{\mu} \sum_{z \in \mathcal{Y} \cup \hat{\mathcal{Y}}} \mathbb{Q}_Y(z)v(z) + \sum_{\hat{z} \in \mathcal{Y} \cup \hat{\mathcal{Y}}} \mathbb{Q}_{\hat{Y}}(\hat{z})\mu(\hat{z}) \end{aligned} \quad (3.203)$$

where the inequality follows from the fact that  $-\psi(\hat{z})$  (next to  $\mathbb{Q}_{\hat{Y}}(\hat{z})$  in the first equation) can be interpreted as a particular choice among general functions represented by  $\mu(\cdot)$ ; and the last equality comes from a notational change:  $\psi(\cdot) \rightarrow v(\cdot)$ .

On the other hand, with the notational changes w.r.t. functions  $(-\psi(\hat{z}) \rightarrow \mu(\hat{z})$  and  $\psi(z) \rightarrow v(z)$ ), one can represent the constraint in (3.202) as:

$$|v(z) + \mu(\hat{z})| \leq \|z - \hat{z}\| \quad \forall z, \hat{z} \in \mathcal{Y} \cup \hat{\mathcal{Y}}. \quad (3.204)$$

Since the RHS in the above is non-negative, the constraint implies that:

$$v(z) + \mu(\hat{z}) \leq \|z - \hat{z}\| \quad \forall z, \hat{z} \in \mathcal{Y} \cup \hat{\mathcal{Y}}. \quad (3.205)$$

This constraint (3.205) definitely yields a larger search space relative to (3.204), since (3.205) does not necessarily imply (3.204). Since the above optimization is about *maximization*, the larger search space gives:

$$\begin{aligned} d^{**} &\leq \max_v \max_{\mu} \sum_{z \in \bar{\mathcal{Y}}} \mathbb{Q}_Y(z)v(z) + \sum_{\hat{z} \in \bar{\mathcal{Y}}} \mathbb{Q}_{\hat{Y}}(\hat{z})\mu(\hat{z}) : \\ &\quad v(z) + \mu(\hat{z}) \leq \|z - \hat{z}\| \quad \forall z, \hat{z} \in \bar{\mathcal{Y}}. \end{aligned} \quad (3.206)$$

Notice that the RHS in the above is exactly the same as the original dual problem (3.201) with the optimal value  $d^*$ . Hence, this proves:

$$d^{**} \leq d^*. \quad (3.207)$$

**Proof of  $d^* \leq d^{**}$**  Let us start by recalling the original dual problem (3.201):

$$\begin{aligned} d^* := \max_{\nu, \mu} & \sum_{z \in \bar{\mathcal{Y}}} \mathbb{Q}_Y(z) \nu(z) + \sum_{\hat{z} \in \bar{\mathcal{Y}}} \mathbb{Q}_{\hat{Y}}(\hat{z}) \mu(\hat{z}) : \\ & \nu(z) + \mu(\hat{z}) \leq \|z - \hat{z}\| \quad \forall z, \hat{z} \in \bar{\mathcal{Y}}. \end{aligned} \quad (3.208)$$

By massaging the constraint in the above dual problem, we can relate it to the constraint in the translated problem (3.202). To see this, let us first move the  $\nu(z)$  term in the constraint to the RHS. This then yields:

$$\mu(\hat{z}) \leq -\nu(z) + \|z - \hat{z}\|. \quad (3.209)$$

Since this holds for all  $z, \hat{z} \in \bar{\mathcal{Y}}$ , we obtain:

$$\mu(\hat{z}) \leq \min_{z \in \bar{\mathcal{Y}}} \{-\nu(z) + \|z - \hat{z}\|\}. \quad (3.210)$$

Next we define the RHS in the above as  $-\psi(\hat{z})$ :

$$-\psi(\hat{z}) := \min_{z \in \bar{\mathcal{Y}}} \{-\nu(z) + \|z - \hat{z}\|\}. \quad (3.211)$$

This definition then yields:

$$\mu(\hat{z}) \leq -\psi(\hat{z}) \quad \forall \hat{z} \in \bar{\mathcal{Y}}. \quad (3.212)$$

On the other hand, in (3.211), consider a particular choice for  $z$  as  $\hat{z}$ . This then gives:

$$-\psi(\hat{z}) \leq -\nu(\hat{z}) \quad \forall \hat{z} \in \bar{\mathcal{Y}}. \quad (3.213)$$

With a notational change in the above from  $\hat{z}$  to  $z$ , we get:

$$\nu(z) \leq \psi(z) \quad \forall z \in \bar{\mathcal{Y}}. \quad (3.214)$$

Applying the derived inequalities (3.214) and (3.212) into the original dual problem (3.208), we get:

$$d^* \leq \max_{\psi} \sum_{z \in \bar{\mathcal{Y}}} \mathbb{Q}_Y(z) \psi(z) - \sum_{\hat{z} \in \bar{\mathcal{Y}}} \mathbb{Q}_{\hat{Y}}(\hat{z}) \psi(\hat{z}). \quad (3.215)$$

This coincides with the objective function in the other interested optimization (3.202).

The next is to figure out how the constraint  $|\psi(z) - \psi(\hat{z})| \leq \|z - \hat{z}\|$  in the translated optimization (3.202) comes up. It turns out that the definition (3.211)

incurs the constraint:  $|\psi(z) - \psi(\hat{z})| \leq \|z - \hat{z}\|$ ,  $\forall z, \hat{z} \in \bar{\mathcal{Y}}$ . To see this clearly, consider:

$$\begin{aligned}\psi(z) &= \max_{t \in \bar{\mathcal{Y}}} \{\nu(t) - \|t - z\|\} \quad \forall z \in \bar{\mathcal{Y}}; \\ -\psi(\hat{z}) &= \min_{t' \in \bar{\mathcal{Y}}} \{-\nu(t') + \|t' - \hat{z}\|\} \quad \forall \hat{z} \in \bar{\mathcal{Y}}.\end{aligned}\tag{3.216}$$

This comes simply from the definition (3.211). Adding the above two, we get:  $\forall z, \hat{z} \in \bar{\mathcal{Y}}$ ,

$$\begin{aligned}\psi(z) - \psi(\hat{z}) &= \max_{t \in \bar{\mathcal{Y}}} \{\nu(t) - \|t - z\|\} + \min_{t' \in \bar{\mathcal{Y}}} \{-\nu(t') + \|t' - \hat{z}\|\} \\ &\stackrel{(a)}{\leq} \max_{t \in \bar{\mathcal{Y}}} \{\nu(t) - \|t - z\| - \nu(\textcolor{blue}{t}) + \|\textcolor{blue}{t} - \hat{z}\|\} \\ &\stackrel{(b)}{\leq} \|z - \hat{z}\|\end{aligned}\tag{3.217}$$

where (a) follows from choosing  $t' = t$  in the minimization part; and (b) comes from the triangular inequality:

$$\|t - \hat{z}\| \leq \|t - z\| + \|z - \hat{z}\|.$$

Swapping the roles of  $z$  and  $\hat{z}$  in (3.217), one can also get:  $\forall z, \hat{z} \in \bar{\mathcal{Y}}$ ,

$$\psi(\hat{z}) - \psi(z) \leq \|z - \hat{z}\|. \tag{3.218}$$

This together with (3.217) then yields:

$$|\psi(\hat{z}) - \psi(z)| \leq \|z - \hat{z}\| \quad \forall z, \hat{z} \in \bar{\mathcal{Y}}. \tag{3.219}$$

The above implied constraint definitely yields a larger search space relative to the definition (3.211). Applying this to (3.215), we get:

$$d^* \leq \max_{\psi} \sum_{z \in \bar{\mathcal{Y}}} \mathbb{Q}_Y(z) \psi(z) - \sum_{\hat{z} \in \bar{\mathcal{Y}}} \mathbb{Q}_{\hat{Y}}(\hat{z}) \psi(\hat{z}): \tag{3.220}$$

$$|\psi(\hat{z}) - \psi(z)| \leq \|z - \hat{z}\| \quad \forall z, \hat{z} \in \bar{\mathcal{Y}}.$$

This completes the proof  $d^* \leq d^{**}$ .

**1-Lipschitz constraint** Notice the constraint in (3.202). Actually this is a very well-known constraint in math and stats, called the 1-Lipschitz constraint. It comes from the definition of an 1-Lipschitz function. We say that a function  $f(\cdot)$  is 1-Lip if

$$|f(x_1) - f(x_2)| \leq \|x_1 - x_2\| \quad \forall x_1, x_2. \tag{3.221}$$

An intuition of an 1-Lip function is that the function values evaluated at  $x_1$  and  $x_2$  are not different too much as long as the distance  $\|x_1 - x_2\|$  between the two points is small, meaning that the function changes sort of *smoothly* over  $x$ .

Using this definition, one can therefore say that the function  $\psi(\cdot)$  in (3.202) is 1-Lip. Applying this to (3.202), we can obtain a simpler expression for the optimization problem as:

$$\max_{\psi(\cdot): \text{1-Lip}} \sum_{z \in \bar{\mathcal{Y}}} \mathbb{Q}_Y(z) \psi(z) - \sum_{\hat{z} \in \bar{\mathcal{Y}}} \mathbb{Q}_{\hat{Y}}(\hat{z}) \psi(\hat{z}). \quad (3.222)$$

**Wasserstein GAN** Recall the definitions of  $\mathbb{Q}_Y$  and  $\mathbb{Q}_{\hat{Y}}$ :

$$\begin{aligned} \mathbb{Q}_Y(z) &= \begin{cases} \frac{1}{m}, & \text{if } z \in \mathcal{Y}; \\ 0, & \text{if } z \in \hat{\mathcal{Y}} \setminus \mathcal{Y}; \end{cases} \\ \mathbb{Q}_{\hat{Y}}(\hat{z}) &= \begin{cases} \frac{1}{m}, & \text{if } \hat{z} \in \hat{\mathcal{Y}}; \\ 0, & \text{if } \hat{z} \in \mathcal{Y} \setminus \hat{\mathcal{Y}}. \end{cases} \end{aligned}$$

Also in the original optimization (3.200), we have the outer minimization over  $G(\cdot)$ . Taking all of these into consideration, we can translate the original Wasserstein-distance-based optimization into:

$$\min_{G(\cdot)} \max_{\psi(\cdot): \text{1-Lip}} \frac{1}{m} \sum_{i=1}^m \psi(y^{(i)}) - \frac{1}{m} \sum_{i=1}^m \psi(\hat{y}^{(i)}). \quad (3.223)$$

Obviously this is a function optimization problem. Hence, as Goodfellow did, Bottou employed neural networks for  $G(\cdot)$  and  $\psi(\cdot)$  to approximate the optimization problem (3.223) as:

$$\min_{G(\cdot) \in \mathcal{N}} \max_{\psi(\cdot) \in \mathcal{N}: \text{1-Lip}} \frac{1}{m} \sum_{i=1}^m \psi(y^{(i)}) - \frac{1}{m} \sum_{i=1}^m \psi(\hat{y}^{(i)}) \quad (3.224)$$

where  $\mathcal{N}$  indicates a set of DNN-based functions. This is exactly the optimization problem for WGAN. It turns out the WGAN works pretty well. So as of now, it is the state of the art – many GAN variants that work best for some applications are based on WGAN.

**A fundamental question** Recall the generic divergence-based optimization problem:

$$\min_{G(\cdot)} \mathsf{D}(\mathbb{Q}_Y, \mathbb{Q}_{\hat{Y}}).$$

Obviously the WGAN optimization (3.200) belongs to the above generic problem. Then, one very natural question arises. Is the Wasserstein distance the best choice for  $D(\cdot, \cdot)$ ? In other words,

$$D^*(\cdot, \cdot) = W(\cdot, \cdot)? \quad (3.225)$$

**A special case** In an effort to address this question, a few groups including our group have investigated a special setting in which the optimal generator  $G^*$  is known. The special setting is so called the *Gaussian linear-generator setting* wherein data  $\{y^{(i)}\}_{i=1}^m$  follows a Gaussian distribution, say:

$$y^{(i)} \sim \mathcal{N}(0, K_Y) \quad \text{where } K_Y = U \Lambda U^T, \quad (3.226)$$

and the generator is subject to a linear operation:

$$\hat{y}^{(i)} := G(x^{(i)}) = Gx^{(i)} \quad \text{where } G \in \mathbf{R}^{n \times k}. \quad (3.227)$$

Here one natural assumption that one can make on the distribution of  $x^{(i)}$  is:

$$x^{(i)} \sim \mathcal{N}(0, I), \quad (3.228)$$

as this way suggests that fake samples are also Gaussian, which coincides with the same type of distribution as that of real samples:

$$\hat{y}^{(i)} \sim \mathcal{N}(0, \mathbb{E}[(Gx^{(i)})(Gx^{(i)})^T]) = \mathcal{N}(0, GG^T). \quad (3.229)$$

Fortunately, under the above Gaussian setting, the optimal  $G^*$  is well-known. Here what it means by being optimal is in a sense of maximizing the likelihood of the data, as we adopted while discussing on the optimality of cross entropy loss in Section 3.2. It turns out the optimal  $G^*$  is the one that performs Principle Component Analysis (PCA):

$$\mathbb{E}[\hat{y}^{(i)}\hat{y}^{(i)T}] = G^*G^{*T} = U \text{diag}(\lambda_1, \dots, \lambda_k, 0, \dots, 0) U^T \quad (3.230)$$

where  $(\lambda_1, \dots, \lambda_k)$  denote the  $k$  principal (largest) eigenvalues<sup>8</sup> of  $K_Y = U \Lambda U^T$ . In a usual setting in which  $k < n$ , the PCA solution looks making sense. The rank of  $GG^T$  is limited by  $k$ , so it may not fully represent  $K_Y$  as the rank of  $K_Y$  can be  $n$ . In this case, what one can do for the best is to make  $GG^T$  as close as possible to  $K_Y$ . One such natural way is to take the  $k$  largest eigenvalues of  $K_Y$  to form a covariance

8. One may ask how to compute the principal eigenvalues when  $K_Y$  is unknown, which is often the case in reality. In this case, the optimal way is to compute an *empirical* covariance matrix  $S := \frac{1}{m} \sum_{i=1}^m y^{(i)}y^{(i)T}$  and then to take the  $k$  largest eigenvalues of  $S$ .

matrix. It turns out it is the best way in a sense of maximizing the likelihood of the data. The proof of this will be explored in Prob 10.4.

Now under the special Gaussian linear-generator setting, one can ask the fundamental question (3.225):

$$G^* = G_{\text{WGAN}}^* = \arg \min_{G \in \mathbb{R}^{n \times k}} W(\mathbb{Q}_Y, \mathbb{Q}_{\hat{Y}}) \quad (3.231)$$

It turns out the answer is yes. Actually the proof is not that short. So due to the interest of other topics, we will not prove it; if you are interested, you can consult with a paper ([Cho and Suh, 2019](#)).

However, the answer holds under the special Gaussian setting. So you may wonder if that is the case also under general settings in which the data distribution is *arbitrary*. Unfortunately it has been unanswered. We believe this is one of the fundamental and intriguing questions in the context of the GAN-based framework. Someone may believe that the answer depends on what distribution of data we consider. This may be the case, but even this was not answered. So any progress on this will be interested.

**Look ahead** In the past two sections, we derived the WGAN optimization. In the next section, we will investigate how to implement the optimization problem via TensorFlow.

### 3.11 Wasserstein GAN: TensorFlow Implementation

**Recap** In the prior section, we employed Kantorovich duality to translate the Wasserstein-distance-based optimization into:

$$\min_{G(\cdot)} \max_{\psi(\cdot): \text{1-Lip}} \frac{1}{m} \sum_{i=1}^m \psi(y^{(i)}) - \frac{1}{m} \sum_{i=1}^m \psi(\hat{y}^{(i)}). \quad (3.232)$$

Then, by parameterizing  $G(\cdot)$  and  $\psi(\cdot)$ , we derived the WGAN optimization:

$$\min_{G(\cdot) \in \mathcal{N}} \max_{\psi(\cdot) \in \mathcal{N}: \text{1-Lip}} \frac{1}{m} \sum_{i=1}^m \psi(y^{(i)}) - \frac{1}{m} \sum_{i=1}^m \psi(\hat{y}^{(i)}) \quad (3.233)$$

where  $\mathcal{N}$  indicates a set of DNN-based functions. We also discussed on the optimality of WGAN under a simple Gaussian linear-generator setting.

**Outline** In this section, we will study how to solve the optimization (3.233) as well as how to implement it via TensorFlow. Specifically we are going to cover three stuffs. First we will investigate a practical method to respect the 1-Lipschitz constraint that appears in the design of  $\psi(\cdot)$  in (3.233). Next, we will explore implementation details that the WGAN paper ([Arjovsky et al., 2017](#)) introduced, regarding optimizers, neural network architecture and activation functions. Lastly, we will study TensorFlow implementation in the context of the same task considered in the GAN implementation: MNIST style handwritten digit image generation (see Fig. 3.40).

**A practical method for ensuring the 1-Lipschitz constraint** Under the neural network architecture, it is difficult to fully respect the 1-Lipschitz constraint. Hence, the WGAN paper came up with sort of a heuristic for satisfying the constraint. The heuristic is based on the following observation: a small range of model

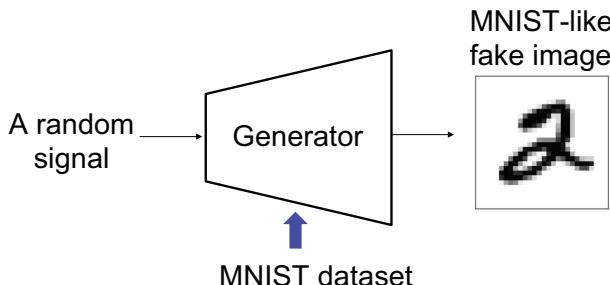


Figure 3.40. MNIST-style handwritten digit image generation.

parameters yields a small variation of the neural network function, thus encouraging the 1-Lipschitz continuity. So the method confines the values of parameter into a small range, say  $[-c, c]$  where  $c$  is a certain positive value. For instance,  $c$  was set to 0.01 in the WGAN paper. This method is called *weight clipping* and its operation reads:

$$w = \begin{cases} -c, & \text{if } w < -c; \\ w, & \text{if } w \in [-c, c]; \\ c, & \text{if } w > c. \end{cases} \quad (3.234)$$

We will employ this for our implementation.

**RMSprop optimizer (Hinton et al., 2012)** Recall the Adam optimizer that we learned in Section 3.5. The weights  $w^{(t)}$  therein are updated as per:

$$w^{(t+1)} = w^{(t)} + \alpha \frac{m^{(t)}}{\sqrt{s^{(t)} + \epsilon}} \quad (3.235)$$

where  $m^{(t)}$  indicates a weighted average of the current and past gradients and  $s^{(t)}$  is a normalization factor that makes the effect of the gradient  $\nabla J(w^{(t)})$  almost constant over  $t$ . The WGAN paper employs a simpler version of Adam that takes only care of the normalization factor  $s^{(t)}$  while ignoring the momentum  $m^{(t)}$ . The simpler optimizer is also a famous one, named RMSprop, and it had been widely used until Adam came around. What the WGAN paper found is that RMSprop is enough to achieve good performances. So in our implementation, we will use this simpler version. Details on the weight update in RMSprop are given below:

$$w^{(t+1)} = w^{(t)} - \alpha \frac{\nabla J(w^{(t)})}{\sqrt{s^{(t)} + \epsilon}}. \quad (3.236)$$

Here the normalization factor  $s^{(t)}$  is updated according to:

$$s^{(t)} = \beta s^{(t-1)} - (1 - \beta)(\nabla J(w^{(t)}))^2 \quad (3.237)$$

where  $\beta \in [0, 1]$  denotes a hyperparameter that captures the weight of past values, typically set to 0.9. In our implementation, we will employ the same hyperparameters as in the WGAN paper: the learning rate  $\alpha = 0.00005$ ; and  $\beta = 0.9$ .

**Leaky ReLU** We mentioned several times that the default choice for activation functions at hidden layers is ReLU. Actually there are many ReLU variants used in practice. One such variant is “leaky ReLU” and it is the one that the WGAN paper employed. The operation is very similar to that of ReLU. The only distinction is

that the output is a scaled version of the input for negative values:

$$\alpha = \begin{cases} z, & \text{if } z \geq 0; \\ \alpha_{\text{slope}}z, & \text{if } z < 0 \end{cases} \quad (3.238)$$

where  $\alpha_{\text{slope}}$  is a hyperparameter indicating a small slope applied for a negative input. Notice that  $\alpha_{\text{slope}} = 0$  gives ReLU while  $\alpha_{\text{slope}} = 1$  yields linear activation. Hence, one can view this as a generalized version of ReLU. In our implementation, we will employ  $\alpha_{\text{slope}} = 0.2$ .

**Convolutional neural networks (CNNs)** The WGAN paper uses a certain type of DNNs, specialized for image data. That is, convolutional neural networks, CNNs for short. So we will also utilize this in our implementation. Since there are lots of stuffs for studying CNNs, here we explain some key features of CNNs in a brief manner and then present the architectures of the CNN-based models to be employed in our implementation.

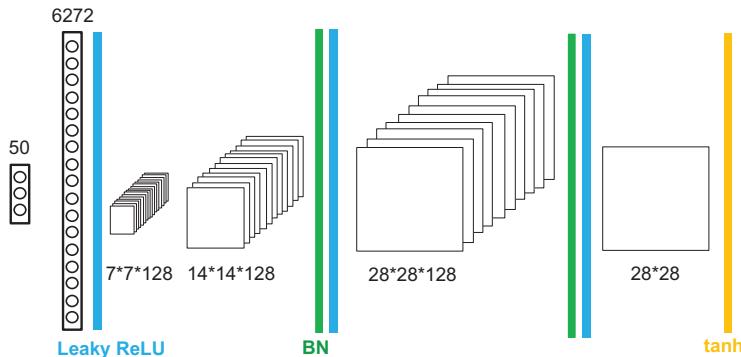
CNNs consist of two building blocks. The first is the conv layer. The key feature of the conv layer is that it is sparsely connected with input values. The sparse-connectivity feature comes from an interesting scientific finding w.r.t. visual neurons of intelligent beings: visual neurons *react only to a limited region of an image* (not the entire region). The second building block is the pooling layer. The role of pooling is not inspired by how visual neurons work. Rather it is mainly for implementation. The role is to downsample signals in an effort to reduce the computational load and the memory size.

**Model for generator** As a generator model, we employ a 5-layer CNN with four hidden layers, as depicted in Fig. 3.41. For activation at each hidden layer, we utilize leaky ReLU (marked in light blue). As in the GAN implementation (in Section 3.8), we also use Batch Normalization (BN), marked in green, prior to activation at the second and third hidden layers. Since an MNIST image consists of 28-by-28 pixels, the output layer has 28-by-28 neurons spread in the 2-dimensional space, and we use tanh activation (a shifted version of logistic activation) to ensure the range of  $[-1, 1]$ .

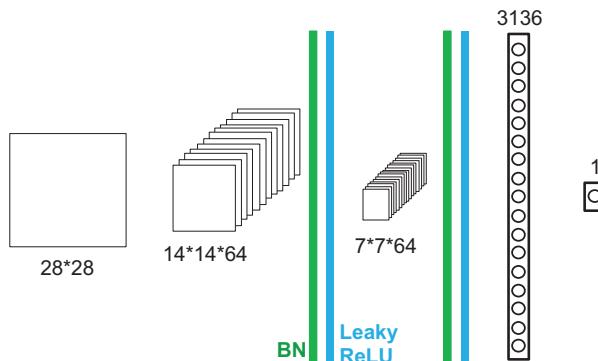
As the first hidden layer, we employ a fully-connected dense layer with 6272 ( $= 7 \times 7 \times 128$ ) neurons. It is then reshaped into a 3D tensor with a size of  $7 \times 7 \times 128$ . Next, we upsample it to have an expanded tensor of size  $14 \times 14 \times 128$ .<sup>9</sup> Again we upsample it to yield another expanded tensor of size  $28 \times 28 \times 128$ . Lastly we have a conv layer to output 28-by-28 sized 2D neurons.

---

9. Explanation for its detailed operations is omitted here. We only present its role and will leave TensorFlow implementation in the sequel.



**Figure 3.41.** Generator: A 5-layer CNN where the input size (the dimension of a latent signal) is 50; we use a 6272-sized dense layer for the first hidden layer; and we utilize conv layers for the remaining layers. The role of the second and third hidden layers is to *upsample* input to yield an expanded 3D tensor (e.g., from  $7 \times 7 \times 128$  to  $14 \times 14 \times 128$  in the second hidden layer). We employ leaky ReLU activation for every hidden layer, and tanh activation for the output layer to ensure  $-1\text{-to-}1$  output signals.



**Figure 3.42.** Critic: A 4-layer CNN where the input size (the dimension of a real or fake image) is  $28 \times 28$ ; we use conv layers for the first and second hidden layers; and we utilize a dense layer in the last layer. The role of the conv layers here is to *downsample* input unlike generator. We employ leaky ReLU activation for every hidden layer, and linear activation for the output layer.

**Model for critic** Instead of using a discriminator to classify generated images as being real or fake, WGAN replaces the discriminator with a *critic* that scores the realness or fakeness of a given image. So we call it critic here. As a model for critic, we use a 4-layer CNN with three hidden layers, illustrated in Fig. 3.42. Here the input size must be the same as that of a real (or fake) image, so it should read  $28 \times 28$ . In the first hidden layer, unlike the generator operation, we *downsample* the input to yield a shrunked map of size  $14 \times 14$ . We generate 64 different maps independently. We then stack all of them to construct a 3D tensor of size  $14 \times 14 \times 64$ . In the next layer, we perform a similar operation to generate another 3D tensor of size

$7 \times 7 \times 64$ . It is then flattened to form a vector of size 3136( $= 7 \times 7 \times 64$ ). Lastly, we have a fully-connected layer to output a single neuron with linear activation (no activation).

**TensorFlow: Model for generator** TensorFlow implementation for the generator model described in Fig. 3.41 is given below.

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import Reshape
from tensorflow.keras.layers import BatchNormalization
from tensorflow.keras.layers import Conv2D
from tensorflow.keras.layers import Conv2DTranspose
from tensorflow.keras.layers import ReLU
from tensorflow.keras.layers import LeakyReLU
from tensorflow.keras.initializers import RandomNormal

latent_dim = 50
# weight initialization
init = RandomNormal(stddev=0.02)

generator=Sequential()
generator.add(Dense(128*7*7,input_dim=latent_dim))
generator.add(LeakyReLU(0.2))
generator.add(Reshape((7,7,128)))
# upsample to 14*14
generator.add(Conv2DTranspose(128, (4,4), strides=(2,2),
                             padding='same', kernel_initializer = init))
generator.add(BatchNormalization())
generator.add(LeakyReLU(0.2))
# upsample to 28*28
generator.add(Conv2DTranspose(128, (4,4), strides=(2,2),
                             padding='same', kernel_initializer = init))
generator.add(BatchNormalization())
generator.add(LeakyReLU(0.2))
# output 28*28*1
generator.add(Conv2D(1, (7,7), activation='tanh',
                     padding='same', kernel_initializer = init))
```

Here we use a built-in class Conv2DTranspose for the purpose of upsampling. It has a couple of input arguments concerning strides, padding and kernel\_initializer. These are all hyperparameters subject to our design choice. We omit all the details in order not to distract you. Remember that these are just particular choices.

**TensorFlow: Model for critic** Unlike generator, the critic model intends to respect the 1-Lipschitz constraint. As mentioned earlier, to this end, we

employ *weight clipping*. Specifically we rely upon the following class for code implementation:

```
from tensorflow.keras import backend
from tensorflow.keras.constraints import Constraint

# clip model weights to a given hypercube
class ClipConstraint(Constraint):
    # set clip value when initialized
    def __init__(self, clip_value):
        self.clip_value = clip_value
    # clip model weights to hypercube
    def __call__(self, weights):
        return backend.clip(weights,
                            -self.clip_value, self.clip_value)
    def get_config(self):
        return {'clip_value': self.clip_value}
```

We use a built-in function `backend.clip` to implement weight clipping (3.234). This can then be employed to apply weight clipping in the design of a critic model. See below for TensorFlow implementation.

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Flatten
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import Reshape
from tensorflow.keras.layers import BatchNormalization
from tensorflow.keras.layers import Conv2D
from tensorflow.keras.layers import Conv2DTranspose
from tensorflow.keras.layers import ReLU
from tensorflow.keras.layers import LeakyReLU
from tensorflow.keras.initializers import RandomNormal

in_shape = (28,28,1)
# clip value
c = 0.01
# weight initialization
init = RandomNormal(stddev=0.02)
# weight constraint
const = ClipConstraint(c)

critic=Sequential()
# downsample to 14*14
critic.add(Conv2D(64,(4,4),strides=(2,2),padding='same',
                 kernel_initializer=init,
                 kernel_constraint=const,
                 input_shape = in_shape))
```

```

critic.add(BatchNormalization())
critic.add(LeakyReLU(0.2))
# downsample to 7*7
critic.add(Conv2D(64,(4,4),strides=(2,2),padding='same',
                 kernel_initializer=init,
                 kernel_constraint=const,
                 input_shape = in_shape))
critic.add(BatchNormalization())
critic.add(LeakyReLU(0.2))
# scoring, linear activation
critic.add(Flatten())
critic.add(Dense(1))

```

Here we use  $c = 0.01$  for a clipping value as in the WGAN paper, and ClipConstraint(c) is employed as an argument in Conv2D to apply weight clipping.

**TensorFlow: Generator input** As a generator input, we use a random signal with the *Gaussian* distribution. In particular, we use:

$$x \in \mathbf{R}^{\text{latent\_dim}} \sim \mathcal{N}(\mathbf{0}, \mathbf{I}_{\text{latent\_dim}}).$$

Here is how to generate the Gaussian random signal in TensorFlow:

```

from tensorflow.random import normal
x = normal([batch_size,latent_dim])

```

**TensorFlow: Optimizers for generator & critic** We use RMSprop optimizers with lr=0.00005 and the default choice of  $\beta = 0.9$ . We can readily implement it via a built-in-class RMSprop:

```

from tensorflow.keras.optimizers import RMSprop
opt = RMSprop(lr=0.00005)

```

Now how about loss functions for generator and critic? To answer this, consider the batch version of the WGAN optimization (3.233):

$$\min_w \max_{\theta} \frac{1}{m_B} \sum_{i \in \mathcal{B}} \psi_{\theta}(y^{(i)}) - \frac{1}{m_B} \sum_{i \in \mathcal{B}} \psi_{\theta}(G_w(x^{(i)})). \quad (3.239)$$

In light of critic, taking the minus sign in the objective, we obtain the equivalent minimization problem:

$$\underbrace{\min_{\theta} -\frac{1}{m_B} \sum_{i \in \mathcal{B}} \psi_{\theta}(y^{(i)}) + \frac{1}{m_B} \sum_{i \in \mathcal{B}} \psi_{\theta}(G_w(x^{(i)}))}_{\text{critic loss}}$$

where the critic loss is defined as the minus version. Here is how to implement the critic loss:

```
from tensorflow.keras import backend
def wasserstein_loss(y_true,y_pred):
    return backend.mean(y_true*y_pred)
```

For a real image, we can set  $y_{\text{true}} = -1$  and take its corresponding critic output for  $y_{\text{pred}}$ . On the other hand, for a fake image,  $y_{\text{true}} = +1$  and  $y_{\text{pred}}$  should read the critic output fed by the fake image.

Since  $\psi_\theta(y^{(i)})$  is irrelevant of the generator weights  $w$  in (3.239), the generator loss is:

$$\min_w - \underbrace{\frac{1}{m_B} \sum_{i \in \mathcal{B}} \psi_\theta(G_w(x^{(i)}))}_{\text{generator loss}}.$$

Hence, we can also implement this via the class `wasserstein_loss` that we defined above. Here  $y_{\text{true}}$  should always be set to  $-1$  and  $y_{\text{pred}}$  should be the critic output fed by a fake image.

Taking all the above into consideration, we can compile generator and critic as below.

```
from tensorflow.keras.layers import Input
from tensorflow.keras.models import Model
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.optimizers import RMSprop
from tensorflow.keras import backend

# RMSprop optimizer
opt = RMSprop(lr=0.00005)

# Define "Wasssterstein loss"
def wasserstein_loss(y_true,y_pred):
    return backend.mean(y_true*y_pred)

# critic compile
critic.compile(loss=wasserstein_loss, optimizer=opt)

# define the GAN model with fake input and critic output
# fix critic's weights while training generator
critic.trainable = False
gan_input = Input(shape=(latent_dim,))
```

```
x = generator(inputs=gan_input)
output = critic(x)
gan = Model(gan_input, output)
# generator compile
gan.compile(loss=wasserstein_loss, optimizer=opt)
```

**TensorFlow: Getting batches** Since we use batches with batch\_size, we provide a code that segments data in the form of batches.

```
import numpy as np
def get_batches(data, batch_size):
    batches = []
    for i in range(int(data.shape[0] // batch_size)):
        batch = data[i*batch_size:(i+1)*batch_size]
        batches.append(batch)
    return np.asarray(batches)
```

**TensorFlow: Training** Putting all of the above together, we can now train generator and critic via *alternating gradient descent*:

```
import numpy as np
from tensorflow.random import normal
EPOCHS = 50
k=2 # k:1 alternating gradient descent
c_losses = []
g_losses = []

for epoch in range(1,EPOCHS + 1):
    # train per each batch
    np.random.shuffle(X_train)
    for i, real_imgs in enumerate(get_batches(X_train, batch_size)):
        ######
        # train critic
        #####
        # fake input generation
        gen_input = normal([batch_size,latent_dim])
        # fake images
        gen_imgs = generator.predict(gen_input)
        real_imgs = real_imgs.reshape(-1,28,28,1)
        # input for critic
        c_input = np.concatenate([real_imgs,gen_imgs])
        # label for critic
```

```
# first half: real (-1); second half: fake (+1)
c_label = np.ones(2*batch_size)
c_label[:batch_size] = -1
# train critic
c_loss = critic.train_on_batch(c_input,c_label)

#####
# train generator
#####
if i % k: # train once every k steps
    # fake input generation
    g_input = normal([batch_size,latent_dim])
    # label for fake images
    # Create inverted labels for fake images
    g_label = - np.ones(batch_size)
    # train generator
    g_loss = gan.train_on_batch(g_input, g_label)

c_losses.append(c_loss)
g_losses.append(g_loss)
```

**Look ahead** So far we have investigated several applications that arise in machine learning, ranging from logistic regression, deep learning, unsupervised learning, GANs, all the way up to WGAN. From the next section, we will move onto the last application which concerns societal issues relevant to optimization techniques. That is, *fair machine learning*.

## Problem Set 10

---

Prob 10.1 (An issue in Goodfellow's GAN) Let

$$\mathbb{Q}_Y(z) = \begin{cases} \frac{1}{m}, & \text{if } z \in \mathcal{Y}; \\ 0, & \text{if } z \in \hat{\mathcal{Y}} \setminus \mathcal{Y}; \end{cases} \quad (3.240)$$

$$\mathbb{Q}_{\hat{Y}}(\hat{z}) = \begin{cases} \frac{1}{m}, & \text{if } \hat{z} \in \hat{\mathcal{Y}}; \\ 0, & \text{if } \hat{z} \in \mathcal{Y} \setminus \hat{\mathcal{Y}}, \end{cases}$$

where  $\mathcal{Y} := \{y^{(1)}, \dots, y^{(m)}\}$  and  $\hat{\mathcal{Y}} := \{\hat{y}^{(1)}, \dots, \hat{y}^{(m)}\}$  indicate the sets of real and fake samples, respectively.

- (a) In Section 3.9, we argued that in many practical settings,

$$\mathcal{Y} \cap \hat{\mathcal{Y}} = \emptyset. \quad (3.241)$$

Explain why.

- (b) Assuming (3.241), show that

$$\text{JSD}(\mathbb{Q}_Y, \mathbb{Q}_{\hat{Y}}) = \log 2. \quad (3.242)$$

- (c) In Section 3.7, we showed that the original GAN by Goodfellow can be translated into the JSD-based optimization problem under a certain assumption. Explain why the original GAN works well in practice although the JSD-based optimization does not guarantee a good performance due to (3.242).

Prob 10.2 (Wasserstein GAN) Consider the 1st-order Wasserstein distance:

$$W(\mathbb{Q}_Y, \mathbb{Q}_{\hat{Y}}) := \min_{\mathbb{Q}_{Y, \hat{Y}}} \mathbb{E} \left[ \|Y - \hat{Y}\| \right] \quad (3.243)$$

where  $\mathbb{Q}_Y$  and  $\mathbb{Q}_{\hat{Y}}$  indicate the empirical distributions of real and fake samples defined as:

$$\mathbb{Q}_Y(z) = \begin{cases} \frac{1}{m}, & \text{if } z \in \mathcal{Y}; \\ 0, & \text{if } z \in \hat{\mathcal{Y}} \setminus \mathcal{Y}; \end{cases} \quad (3.244)$$

$$\mathbb{Q}_{\hat{Y}}(\hat{z}) = \begin{cases} \frac{1}{m}, & \text{if } \hat{z} \in \hat{\mathcal{Y}}; \\ 0, & \text{if } \hat{z} \in \mathcal{Y} \setminus \hat{\mathcal{Y}}. \end{cases}$$

Here  $\mathbb{Q}_{Y, \hat{Y}}$  should respect the marginal distributions of  $\mathbb{Q}_Y$  and  $\mathbb{Q}_{\hat{Y}}$ . Let  $\bar{\mathcal{Y}} := \mathcal{Y} \cup \hat{\mathcal{Y}}$ .

(a) Show that the dual problem of (3.243) can be derived as:

$$\max_{\nu, \mu} \sum_{z \in \bar{\mathcal{Y}}} \mathbb{Q}_Y(z) \nu(z) + \sum_{\hat{z} \in \bar{\mathcal{Y}}} \mathbb{Q}_{\hat{Y}}(\hat{z}) \mu(\hat{z}) : \\ \nu(z) + \mu(\hat{z}) \leq \|z - \hat{z}\| \quad \forall z, \hat{z} \in \bar{\mathcal{Y}}. \quad (3.245)$$

(b) Show that the dual problem (3.245) is equivalent to the following optimization:

$$\max_{\psi} \sum_{z \in \bar{\mathcal{Y}}} \mathbb{Q}_Y(z) \psi(z) - \sum_{\hat{z} \in \bar{\mathcal{Y}}} \mathbb{Q}_{\hat{Y}}(\hat{z}) \psi(\hat{z}) : \\ |\psi(z) - \psi(\hat{z})| \leq \|z - \hat{z}\| \quad \forall z, \hat{z} \in \bar{\mathcal{Y}}. \quad (3.246)$$

**Prob 10.3 (2nd-order Wasserstein distance)** Consider the 2nd-order Wasserstein distance, defined as:

$$W_2^2(\mathbb{Q}_Y, \mathbb{Q}_{\hat{Y}}) := \min_{\mathbb{Q}_{Y, \hat{Y}}} \mathbb{E} \left[ \|Y - \hat{Y}\|^2 \right] \quad (3.247)$$

where  $\mathbb{Q}_Y$  and  $\mathbb{Q}_{\hat{Y}}$  indicate the empirical distributions of real and fake samples defined as in (3.244). Here  $\mathbb{Q}_{Y, \hat{Y}}$  should respect the marginal distributions of  $\mathbb{Q}_Y$  and  $\mathbb{Q}_{\hat{Y}}$ . Let  $\bar{\mathcal{Y}} := \mathcal{Y} \cup \hat{\mathcal{Y}}$ .

(a) Show that

$$W_2^2(\mathbb{Q}_Y, \mathbb{Q}_{\hat{Y}}) = \mathbb{E} [\|Y\|^2] + \mathbb{E} [\|\hat{Y}\|^2] + 2 \min_{\mathbb{Q}_{Y, \hat{Y}}} \mathbb{E} [-\hat{Y}^T Y]. \quad (3.248)$$

(b) Let

$$T(\mathbb{Q}_Y, \mathbb{Q}_{\hat{Y}}) := \min_{\mathbb{Q}_{Y, \hat{Y}}} \mathbb{E} [-\hat{Y}^T Y]. \quad (3.249)$$

Show that the dual problem of (3.249) can be derived as:

$$d^* := \max_{\nu, \mu} \sum_{z \in \bar{\mathcal{Y}}} \mathbb{Q}_Y(z) \nu(z) + \sum_{\hat{z} \in \bar{\mathcal{Y}}} \mathbb{Q}_{\hat{Y}}(\hat{z}) \mu(\hat{z}) : \\ \nu(z) + \mu(\hat{z}) \leq -\hat{z}^T z \quad \forall z, \hat{z} \in \bar{\mathcal{Y}}. \quad (3.250)$$

(c) Show that the dual problem (3.250) is equivalent to the following optimization:

$$d^{**} := \max_{\psi(\cdot): \text{convex}} - \sum_{z \in \bar{\mathcal{Y}}} \mathbb{Q}_Y(z) \psi(z) - \sum_{\hat{z} \in \bar{\mathcal{Y}}} \mathbb{Q}_{\hat{Y}}(\hat{z}) \hat{\psi}(\hat{z}) \quad (3.251)$$

where

$$\hat{\psi}(\hat{z}) := \max_{z \in \bar{\mathcal{Y}}} \left\{ -\psi(z) + \hat{z}^T z \right\} \quad \forall \hat{z} \in \bar{\mathcal{Y}}. \quad (3.252)$$

**Prob 10.4 (Gaussian linear generator)** Consider a Gaussian linear-generator setting in which the data  $\{y^{(i)}\}_{i=1}^m$  follows a Gaussian distribution, say:

$$y^{(i)} \sim \mathcal{N}(0, K_Y) \quad \text{where } K_Y = U \Lambda U^T, \quad (3.253)$$

and the generator is subject to a linear operation:

$$\hat{y}^{(i)} := G(x^{(i)}) = Gx^{(i)} \quad \text{where } G \in \mathbf{R}^{n \times k}. \quad (3.254)$$

Assume that  $\text{rank}(K_Y) = n$ ;  $k < n$ ; and

$$x^{(i)} \sim \mathcal{N}(0, I). \quad (3.255)$$

Also assume that  $x^{(i)}$ 's are independent and identically distributed (i.i.d.); so are  $y^{(i)}$ 's.

(a) Show that the distribution of  $\hat{y}^{(i)}$  is:

$$\hat{y}^{(i)} \sim \mathcal{N}(0, GG^T). \quad (3.256)$$

(b) Let  $K := GG^T$ . With the eigenvalue decomposition, let us express  $K$  as:

$$K = V \Sigma V^T \quad (3.257)$$

where  $\Sigma := \text{diag}(\sigma_1, \dots, \sigma_k, 0, \dots, 0)$  and  $V \in \mathbf{R}^{n \times n}$  indicates a unitary matrix. Here  $\sigma_i$ 's are eigenvalues of  $K$ . Let  $K^\dagger := V \Sigma^{-1} V^T$  be the pseudo-inverse of  $K$  where  $\Sigma^{-1}$  is defined as:

$$\Sigma^{-1} := \text{diag}(\sigma_1^{-1}, \dots, \sigma_k^{-1}, 0, \dots, 0). \quad (3.258)$$

Consider a density function defined on the projected space w.r.t.  $G$ :

$$f_{\hat{Y}}(t) = \frac{1}{\sqrt{(2\pi)^k |K|}} \exp \left( -\frac{1}{2} t^T K^\dagger t \right) \quad (3.259)$$

where  $t \in \mathbf{R}^n$  and  $|K| := \prod_{i=1}^k \sigma_i$ . Express

$$\log \left\{ \prod_{i=1}^m f_{\hat{Y}}(y^{(i)}) \right\} \quad (3.260)$$

in terms of a sample covariance matrix  $S := \frac{1}{m} \sum_{i=1}^m y^{(i)} y^{(i)T}$ .

- (c) Derive the optimal  $K^* = G^* G^T$  such that the log-likelihood function (3.260) is maximized.

**Prob 10.5 (Uniform linear generator)** Suppose  $Y = [Y_1, Y_2]^T$  is a two-dimensional random vector where  $Y_i$ 's are independent and identically distributed (i.i.d.)  $\sim \text{Unif}[0, 1]$ , i.e., the probability density function  $f_{Y_i}(y) = 1$  for  $y \in [0, 1]$  and  $i \in \{1, 2\}$ . Let  $X \sim \text{Unif}[0, 1]$ , being independent of  $Y$ . Let  $\hat{Y} = [g_1 X, g_2 X]^T$  where  $g_1, g_2 \in \mathbf{R}$ . Let  $f_{\hat{Y}}(\cdot)$  be the probability density function w.r.t.  $\hat{Y}$ . A student claims that  $\text{JSD}(f_Y, f_{\hat{Y}})$  is irrelevant of how we choose  $(g_1, g_2)$ . Prove or disprove this claim.

### Prob 10.6 (True or False?)

- (a) Let  $\{y^{(i)}\}_{i=1}^m$  and  $\{\hat{y}^{(i)}\}_{i=1}^m$  be real and fake samples in generative modeling. Let  $\mathcal{Y} := \{y^{(1)}, \dots, y^{(m)}\}$  and  $\hat{\mathcal{Y}} := \{\hat{y}^{(1)}, \dots, \hat{y}^{(m)}\}$ . Suppose  $|\mathcal{Y}| = |\hat{\mathcal{Y}}| = m$  and  $\mathcal{Y} \cap \hat{\mathcal{Y}} = \emptyset$ . Then, the Jensen-Shannon divergence between the empirical distributions of real and fake samples is irrelevant of the generator function  $G(\cdot)$ , i.e.,  $\text{JSD}(\mathbb{Q}_Y, \mathbb{Q}_{\hat{Y}})$  does not change over  $G(\cdot)$ .
- (b) Consider a Gaussian linear-generator setting in which the data  $\{y^{(i)}\}_{i=1}^m$  follows a Gaussian distribution, say  $y^{(i)} \sim \mathcal{N}(0, K_Y)$ , where  $y^{(i)} \in \mathbf{R}^n$  and  $K_Y$  has full rank; the generator is subject to a linear operation  $\hat{y}^{(i)} = Gx^{(i)}$  where  $G \in \mathbf{R}^{n \times k}$  and  $x^{(i)} \in \mathbf{R}^k$ . Assume that  $k < n$  and  $x^{(i)} \sim \mathcal{N}(0, I)$ . Also assume that  $x^{(i)}$ 's are independent and identically distributed (i.i.d.); so are  $y^{(i)}$ 's. Consider a regime in which  $m \rightarrow \infty$ . Then, under this setting, Wasserstein GAN can yield the optimal  $G$  that maximizes the likelihood (probability density function) of the data.
- (c) Suppose  $Y = [Y_1, Y_2]^T$  is a two-dimensional random vector where  $Y_i$ 's are i.i.d.  $\sim \text{Unif}[0, 1]$ , i.e.,  $f_{Y_i}(y) = 1$  for  $y \in [0, 1]$ . Let  $X \sim \text{Unif}[0, 1]$ , being independent of  $Y$ . Let  $\hat{Y} = [g_1 X, g_2 X]^T$  where  $g_i \in \mathbf{R}$ . Consider  $\text{KLD}(f_Y, f_{\hat{Y}})$  and  $\text{JSD}(f_Y, f_{\hat{Y}})$ . Both of them are invariant of the values of  $(g_1, g_2)$ .

## 3.12 Fair Machine Learning

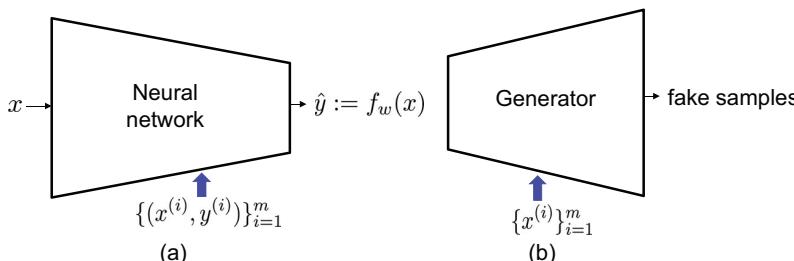
**Recap** During the past sections, we have explored two prominent methodologies for machine learning: (i) supervised learning; and (ii) unsupervised learning. The goal of supervised learning is to estimate the function  $f(\cdot)$  of an interested system from input-output example pairs, as illustrated in Fig. 3.43(a).

In an effort to translate a function optimization problem (a natural formulation of supervised learning) into a parameter-based optimization problem, we parameterized the function with a deep neural network. We also investigated some common practices adopted by many practitioners: Employing ReLU (or leaky ReLU) activation at hidden layers and softmax at the output layer; using *cross entropy* loss (the optimal loss function in a sense of maximum likelihood); and applying advanced versions of gradient descent, Adam and RMSprop optimizers.

We also learned about unsupervised learning. We put a special emphasis on one famous unsupervised learning method: generative modeling, wherein the goal is to generate fake examples so that their distribution is as close as possible to that of real examples; see Fig. 3.43(b). In particular, we focused on one powerful generative model based on Generative Adversarial Networks (GANs), which have played a revolutionary role in the modern AI field. We explored its interesting connection to a well-known divergence measure in statistics: Jensen-Shannon divergence. We also studied a variant of GAN, named Wasserstein GAN, which addresses a critical issue that arises in Goodfellow's GAN under very expressive DNN architectures. Moreover, we learned how to do TensorFlow implementation both for GAN and WGAN.

**Next application** As the final application, we will explore one recent trending topic that arises in the modern machine learning: *Fair machine learning*. There are three reasons that we emphasize this topic.

The first reason is motivated by the recent trend in the machine learning field. As machine learning becomes prevalent in our daily lives involving a widening array of



**Figure 3.43.** (a) Supervised learning: Learning the function  $f(\cdot)$  of an interested system from input-output example pairs  $\{(x^{(i)}, y^{(i)})\}_{i=1}^m$ ; (b) Generative modeling (an unsupervised learning methodology): Generating fake data that resemble real data, reflected in  $\{x^{(i)}\}_{i=1}^m$ .



**Figure 3.44.** Machine learning-based recidivism score predictor of the US Supreme Court: Black defendants were 77.3 percent more likely than white defendants to receive high recidivism scores.

applications such as medicine, finance, job hiring and criminal justice, one morally & legally motivated need in the design of machine learning algorithms is to ensure *fairness* for disadvantageous against advantageous groups. The fairness issue has received a particular attention from the learning algorithm by the US Supreme Court that yields unbalanced recidivism (criminal reoffending) scores across distinct races, e.g., predicts higher scores for blacks against whites ([Larson et al.](#)); see Fig. 3.44. Hence, we wish to touch upon the trending & important topic for this book. The second is regarding an interesting connection to two contents that we learned in Parts I and II, respectively: (i) the regularization technique; and (ii) the optimality condition of convex optimization, characterized by the KKT condition. It turns out that these two play a key role in formulating an optimization problem for fair machine learning algorithms. The last reason is that the optimization problem is closely related to the GAN optimization that we learned in the past sections. You may see the perfect coherent sequence of applications, from supervised learning, GANs to fair machine learning.

**During upcoming sections** For a couple of upcoming sections, we will investigate fair machine learning in depth. What we are going to cover are four folded. First off, we will figure out what fair machine learning is. We will then study two prominent fairness concepts that have been established in the recent literature. We will also formulate an optimization for fair machine learning algorithms which respect the fairness constraints based on the concepts. Next we will demonstrate that the regularization technique forms the basis of such an optimization and the optimization can be rewritten as the GAN optimization that we learned in the prior application. Lastly we will learn how to solve the optimization and implement via TensorFlow. In this section, we will cover the first two.

**Fair machine learning** Fair machine learning is a subfield of machine learning that focuses on the theme of *fairness*. In view of the definition of machine learning,

fair machine learning can concretely be defined as a field of algorithms that train a machine so that it can perform a specific task in a *fair* manner.

Like traditional machine learning, there are two methodologies for fair machine learning. One is fair supervised learning, wherein the goal is to develop a *fair* classifier using input-output sample pairs:  $\{(x^{(i)}, y^{(i)})\}_{i=1}^m$ . The second is the unsupervised learning counterpart. In particular, what is a proper counterpart for generative modeling? A natural one is: *fair generative modeling* in which the goal is to generate *fairness-ensured* fake data which are also realistic. For instance, we may want to generate class-balanced generated samples even when trained with size-biased real data across different demographics. In this book, we will focus only on fair supervised learning.

**Two major fairness concepts** In order to develop a *fair* classifier, we first need to understand what it means by *fairness*. Fairness is a terminology that arises in law that deals with justice. So it has a long and rich history, and there are numerous concepts prevalent in the field of law. We focus only on two major and prominent concepts on fairness, which have received particular attention in the modern machine learning field.

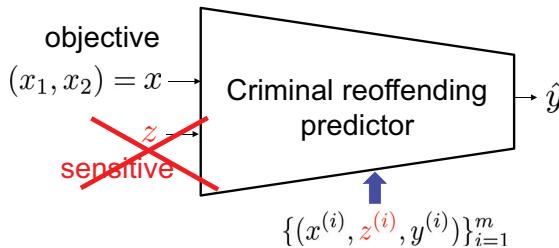
The first is *disparate treatment (DT)*. This means an unequal treatment that occurs *directly* because of some sensitive information (such as race, sex, and religion), often called *sensitive attributes* in the literature. It is also called *direct discrimination*, since such attributes directly serve to incur discrimination.

The second is *disparate impact (DI)*. This means an action that adversely affects one group against another *even with formally neutral rules* wherein sensitive attributes are never used in classification and therefore the DT does not occur. It is also called *indirect discrimination*, since a disparate action is made *indirectly* through biased historical data.

**Criminal reoffending predictor** How to design a fair classifier that respects the above two fairness concepts: DT and DI? For simplicity, let us explore this in the context of a simple yet concrete classification setting: *Criminal reoffending prediction*, wherein the task is to predict whether or not an interested individual with criminal records would reoffend in the near future, say within two years. This is indeed the classification being done by the US Supreme Court for the purpose of deciding parole.

**A simple setting** For illustrative purpose, we consider a simplified version of the predictor wherein only a few information are employed for prediction. See Fig. 3.45.

There are two types of data employed: (i) *objective* data; (ii) *sensitive* data (or called *sensitive attributes*). For objective data that we denote by  $x$ , we consider only



**Figure 3.45.** A criminal reoffending predictor.

two features, say  $x_1$  and  $x_2$ . Let  $x_1$  be the number of prior criminal records. Let  $x_2$  be a criminal type, e.g., misdemeanour or felony. For *sensitive* data, we employ a different notation, say  $z$ . We consider a simple scalar and binary case in which  $z$  indicates a race type only among white ( $z = 0$ ) and black ( $z = 1$ ). Let  $\hat{y}$  be the classifier output which aims to represent the ground-truth conditional distribution  $\mathbb{P}(y|x, z)$ . Here  $y$  denotes the ground-truth label:  $y = 1$  means reoffending within 2 years;  $y = 0$  otherwise. This is a supervised learning setup, so we are given  $m$  example triplets:  $\{(x^{(i)}, z^{(i)}, y^{(i)})\}_{i=1}^m$ .

**How to avoid disparate treatment?** First of all, how to deal with disparate treatment? Recall the DT concept: An unequal treatment *directly* because of sensitive attributes. Hence, in order to avoid the DT, we should ensure that the prediction should not be a function of the sensitive attribute. A mathematically precise expression for this is:

$$\mathbb{P}(y|x, z) = \mathbb{P}(y|x) \quad \forall z. \quad (3.261)$$

How to ensure the above? The solution is very simple: Not using the sensitive attribute  $z$  at all in the prediction, as illustrated with a red-colored “x” mark in Fig. 3.45. Here an important thing to notice is that the sensitive attribute is offered as part of training data although it is not used as part of input. So  $z^{(i)}$ ’s can be employed in the design of an algorithm.

**What about disparate impact?** How about for the other fairness concept: disparate impact? How to avoid DI? Again recall the DI concept: An action that adversely affects one group against another even with formally neutral rules. Actually it is not that clear as to how to implement this mathematically.

To gain some insights, let us investigate the precise mathematical definition of DI. To this end, let us introduce a few notations. Let  $Z$  be a random variable that indicates a sensitive attribute. For instance, consider a binary case, say  $Z \in \{0, 1\}$ . Let  $\tilde{Y}$  be a binary hard-decision value of the predictor output  $\hat{Y}$  at the middle threshold:  $\tilde{Y} := \mathbf{1}\{\hat{Y} \geq 0.5\}$ . Observe a ratio of likelihoods of positive example

events  $\tilde{Y} = 1$  for two cases:  $Z = 0$  and  $Z = 1$ .

$$\frac{\mathbb{P}(\tilde{Y} = 1|Z = 0)}{\mathbb{P}(\tilde{Y} = 1|Z = 1)}. \quad (3.262)$$

One natural interpretation is that a classifier is more fair when the ratio is closer to 1; becomes unfair if the ratio is far away from 1. The DI is quantified based on this, so it is defined as (Zafar *et al.*, 2017):

$$\text{DI} := \min \left( \frac{\mathbb{P}(\tilde{Y} = 1|Z = 0)}{\mathbb{P}(\tilde{Y} = 1|Z = 1)}, \frac{\mathbb{P}(\tilde{Y} = 1|Z = 1)}{\mathbb{P}(\tilde{Y} = 1|Z = 0)} \right). \quad (3.263)$$

Notice that  $0 \leq \text{DI} \leq 1$  and the larger DI, the more fair the situation is.

**Two cases** In view of the mathematical definition (3.263), reducing disparate impact means maximizing the mathematical quantity (3.263). Now how to design a classifier so as to maximize DI then? Depending on situations, the design methodology can be different. To see this, think about two extreme cases.

The first is the one in which training data is already fair:

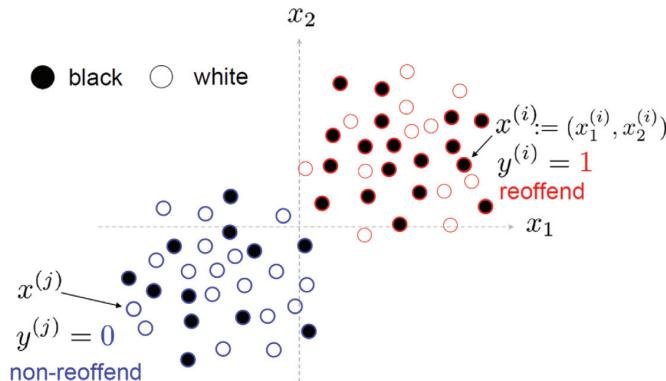
$$\{(x^{(i)}, z^{(i)}, y^{(i)})\}_{i=1}^m \rightarrow \text{large DI}.$$

In this case, a natural solution is to simply rely on a conventional classifier that aims to maximize prediction accuracy. Why? Because maximizing prediction accuracy would well respect training data, which in turn yields large DI. The second is a non-trivial case in which training data is far from being fair:

$$\{(x^{(i)}, z^{(i)}, y^{(i)})\}_{i=1}^m \rightarrow \text{small DI}.$$

In this case, the conventional classifier would yield a small value of DI. This is indeed a challenging scenario where we need to take some non-trivial action for ensuring fairness.

In fact, the second scenario can often occur in reality, since there could be biased *historical* records which form the basis of training data. For instance, the Supreme Court can make some biased decisions for blacks against whites, and these are likely to be employed as training data. See Fig. 3.46 for one such unfair scenario. In Fig. 3.46, a hollowed (or black-colored-solid) circle indicates a data point of an individual with white (or black) race; and the red (or blue) colored edge (ring) denotes the event that the interested individual **reoffends** (or **non-reoffends**) within two years. This is an *unfair* situation. Notice that for positive examples  $y = 1$ , there are more black-colored-solid circles than hollowed ones, meaning sort of biased historical records favouring whites against blacks. Similarly for negative examples  $y = 0$ , there are more hollowed circles relative to solid ones.



**Figure 3.46.** Visualization of a historically biased dataset: A hollowed (or black-colored-solid) circle indicates a data point of an individual with white (or black) race; the red (or blue) colored edge denotes  $y = 1$  reoffending (or  $y = 0$  non-reoffending) label.

**How to ensure large DI?** How to ensure large DI under all possible scenarios including the above unfair challenging scenario? To gain insights, first recall an optimization problem that we formulated earlier in the design of a conventional classifier:

$$\min_w \frac{1}{m} \sum_{i=1}^m \ell_{\text{CE}}(y^{(i)}, \hat{y}^{(i)}) \quad (3.264)$$

where  $\ell_{\text{CE}}(\cdot, \cdot)$  indicates binary cross entropy loss, and  $w$  denotes weights (parameters) for a classifier. One natural approach to ensure large DI is to incorporate an DI-related constraint in the optimization (3.264). Maximizing DI is equivalent to minimizing  $1 - \text{DI}$  (since  $0 \leq \text{DI} \leq 1$ ). So we can resort to the regularization technique that we learned in Part I. That is, adding the two objectives with different weights.

**Regularized optimization** Here is a regularized optimization:

$$\min_w \frac{1}{m} \sum_{i=1}^m \ell_{\text{CE}}(y^{(i)}, \hat{y}^{(i)}) + \lambda \cdot (1 - \text{DI}) \quad (3.265)$$

where  $\lambda$  denote a regularization factor that balances predication accuracy against the DI-associated objective (minimizing  $1 - \text{DI}$ ). However, here an issue arises in solving the regularized optimization (3.265). Recalling the definition of DI

$$\text{DI} := \min \left( \frac{\mathbb{P}(\tilde{Y} = 1|Z = 0)}{\mathbb{P}(\tilde{Y} = 1|Z = 1)}, \frac{\mathbb{P}(\tilde{Y} = 1|Z = 1)}{\mathbb{P}(\tilde{Y} = 1|Z = 0)} \right),$$

we see that DI is a complicated function of  $w$ . We have no idea as to how to express DI in terms of  $w$ .

**Another way** Since directly expressing DI as a function of  $w$  is not doable, one can rely on another way to go. Another way that we will take is inspired by one popular information-theoretic measure: *mutual information* (Cover, 1999). Notice that  $\text{DI} = 1$  means that the sensitive attribute  $Z$  is *independent* of the hard decision  $\tilde{Y}$  of the prediction. One key property of mutual information is that mutual information between two input random variables being zero is the “sufficient and necessary condition” for the independence between the two inputs. This motivates us to represent the constraint of  $\text{DI} = 1$  as:

$$I(Z; \tilde{Y}) = 0. \quad (3.266)$$

This captures the complete independence between  $Z$  and  $\tilde{Y}$ . Since the predictor output is  $\hat{Y}$  (instead of  $\tilde{Y}$ ), we consider another stronger condition that concerns  $\hat{Y}$  directly:

$$I(Z; \hat{Y}) = 0. \quad (3.267)$$

Notice that the condition (3.267) is indeed stronger than (3.266), i.e., (3.267) implies (3.266). This is because

$$\begin{aligned} I(Z; \tilde{Y}) &\stackrel{(a)}{\leq} I(Z; \tilde{Y}, \hat{Y}) \\ &\stackrel{(b)}{=} I(Z; \hat{Y}). \end{aligned} \quad (3.268)$$

Here the step (a) is due to two key properties that mutual information has: (i) the chain rule holds for mutual information, i.e.,  $I(Z; \tilde{Y}, \hat{Y}) = I(Z; \tilde{Y}) + I(Z; \hat{Y} | \tilde{Y})$ ; and (ii) mutual information is non-negative like Kullback-Leibler divergence; in this case,  $I(Z; \hat{Y} | \tilde{Y}) \geq 0$ . The step (b) is also because of the chain rule. To see this, we employ the chain rule to have:

$$I(Z; \tilde{Y}, \hat{Y}) = I(Z; \hat{Y}) + I(Z; \tilde{Y} | \hat{Y}).$$

Here  $I(Z; \tilde{Y} | \hat{Y}) = 0$  since  $\tilde{Y}$  is a function of  $\hat{Y}$ :  $\tilde{Y} := \mathbf{1}\{\hat{Y} \geq 0.5\}$ . Notice that (3.267) together with (3.268) gives (3.266).

**Strongly regularized optimization** In summary, the condition (3.267) indeed enforces the  $\text{DI} = 1$  constraint. This then motivates us to consider the following optimization:

$$\min_w \frac{1}{m} \sum_{i=1}^m \ell_{\text{CE}}(y^{(i)}, \hat{y}^{(i)}) + \lambda \cdot I(Z; \hat{Y}). \quad (3.269)$$

Now the question of interest is: How to express  $I(Z; \hat{Y})$  in terms of classifier parameters  $w$ ? It turns out interestingly there is a way to express it. Also it is intimately related to the GAN optimization that we learned.

**Look ahead** In the next section, we will employ the way to explicitly formulate an optimization for a fair classifier, and then make an interesting connection with the GAN optimization.

### 3.13 A Fair Classifier and Its Connection to GANs

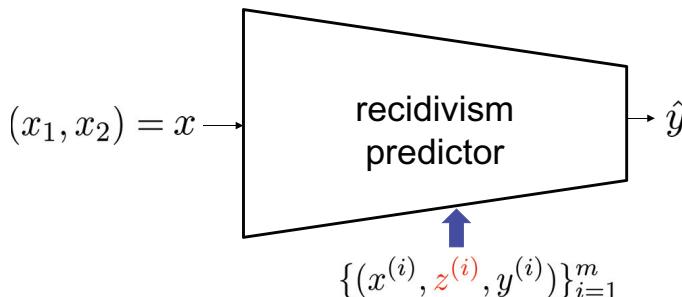
**Recap** In the previous section, we introduced the last machine learning application: A fair classifier. As an example of a fair classifier, we considered a recidivism predictor wherein the task is to predict if an interested individual with prior criminal records would reoffend within two years; see Fig. 3.47 for illustration.

In order to avoid *disparate treatment* (one famous fairness notion), we made the sensitive attribute not included as part of the input. To address another fairness notion (*disparate impact*, DI for short), we introduced a regularized term into the conventional optimization (taking only into account prediction accuracy), thus arriving at:

$$\min_w \frac{1}{m} \sum_{i=1}^m \ell_{\text{CE}}(y^{(i)}, \hat{y}^{(i)}) + \lambda \cdot I(Z; \hat{Y}) \quad (3.270)$$

where  $\lambda \geq 0$  is a regularization factor that balances prediction accuracy (reflected in cross entropy loss) against the quantified fairness constraint, reflected in  $I(Z; \hat{Y})$ . Remember that  $I(Z; \hat{Y}) = 0$  is a sufficient condition for ensuring  $\text{DI} = 1$ . At the end of the last section, we then claimed that one can express the mutual information  $I(Z; \hat{Y})$  in terms of an optimization parameter  $w$ , thereby enabling us to train the model parameterized by  $w$ . We also mentioned that the expressible optimization to be formulated has an intimate connection to GANs.

**Outline** In this section, we will support the claim. Specifically we are going to cover the following four stuffs. First we will explore an interesting connection between mutual information and a well-known divergence measure that we introduced in Prob 8.4: the Kullback-Leibler (KL) divergence. Building upon the connection and applying the optimality condition of convex optimization (fully



**Figure 3.47.** A simple recidivism predictor: Predicting a recidivism score  $\hat{y}$  from  $x = (x_1, x_2)$ . Here  $x_1$  indicates the number of prior criminal records;  $x_2$  denotes a criminal type (misdemeanor or felony); and  $z$  is a race type among white ( $z = 0$ ) and black ( $z = 1$ ).

characterized by the KKT condition), we will show that  $I(Z; \hat{Y})$  can be expressed in terms of a model parameter  $w$ . Next, we will translate the expressible optimization into an implementable form, thereby coming up with a concrete way to solve the optimization. Lastly we will make an analogy with GANs.

**Connection between mutual information vs. KL divergence** There are two versions of definition for mutual information. The first is based on the Shannon entropy that we mentioned in Section 3.2. The second is expressed in terms of the KL divergence. Here we adopt the second version to explore an connection between mutual information and the KL divergence (Cover, 1999):

$$I(Z; \hat{Y}) := \text{KLD} \left( \mathbb{P}_{\hat{Y}, Z}, \mathbb{P}_{\hat{Y}} \mathbb{P}_Z \right). \quad (3.271)$$

If you think about it, this definition makes an intuitive sense. Notice that the independence between  $Z$  and  $\hat{Y}$  implies  $\mathbb{P}_{\hat{Y}, Z} = \mathbb{P}_{\hat{Y}} \mathbb{P}_Z$ , which in turn leads to  $\text{KLD}(\mathbb{P}_{\hat{Y}, Z}, \mathbb{P}_{\hat{Y}} \mathbb{P}_Z) = 0$ , thereby  $I(Z; \hat{Y}) = 0$ .

**Manipulation of (3.271)** Starting with (3.271), we can express the mutual information as:

$$\begin{aligned} I(Z; \hat{Y}) &= \text{KLD} \left( \mathbb{P}_{\hat{Y}, Z}, \mathbb{P}_{\hat{Y}} \mathbb{P}_Z \right) \\ &\stackrel{(a)}{=} \sum_{\hat{y} \in \hat{\mathcal{Y}}, z \in \mathcal{Z}} \mathbb{P}_{\hat{Y}, Z}(\hat{y}, z) \log \frac{\mathbb{P}_{\hat{Y}, Z}(\hat{y}, z)}{\mathbb{P}_{\hat{Y}}(\hat{y}) \mathbb{P}_Z(z)} \\ &\stackrel{(b)}{=} \sum_{\hat{y} \in \hat{\mathcal{Y}}, z \in \mathcal{Z}} \mathbb{P}_{\hat{Y}, Z}(\hat{y}, z) \log \frac{\mathbb{P}_{\hat{Y}, Z}(\hat{y}, z)}{\mathbb{P}_{\hat{Y}}(\hat{y})} \\ &\quad + \sum_{z \in \mathcal{Z}} \mathbb{P}_Z(z) \log \frac{1}{\mathbb{P}_Z(z)} \\ &\stackrel{(c)}{=} \sum_{\hat{y} \in \hat{\mathcal{Y}}, z \in \mathcal{Z}} \mathbb{P}_{\hat{Y}, Z}(\hat{y}, z) \log \frac{\mathbb{P}_{\hat{Y}, Z}(\hat{y}, z)}{\mathbb{P}_{\hat{Y}}(\hat{y})} + H(Z) \end{aligned} \quad (3.272)$$

where (a) is due to the definition of the KL divergence; (b) comes from the total probability law; and (c) is due to the definition of the Shannon entropy.

Now define the term placed in the last line marked in blue as:

$$D^*(\hat{y}, z) := \frac{\mathbb{P}_{\hat{Y}, Z}(\hat{y}, z)}{\mathbb{P}_{\hat{Y}}(\hat{y})}. \quad (3.273)$$

Due to the total probability law,  $D^*(\hat{y}, z)$  should respect the sum-up-to-one constraint w.r.t.  $z$ :

$$\sum_z D^*(\hat{y}, z) = 1 \quad \forall \hat{y}. \quad (3.274)$$

**Mutual information via function optimization** Instead of  $D^*(\hat{y}, z)$ , one can think about another function, say  $D(\hat{y}, z)$ , which respects only the sum-up-to-one constraint (3.274). It turns out  $D^*(\hat{y}, z)$  is the optimal choice among such  $D(\hat{y}, z)$  in a sense of maximizing:

$$\sum_{\hat{y}, z} \mathbb{P}_{\hat{Y}, Z}(\hat{y}, z) \log D(\hat{y}, z), \quad (3.275)$$

and this gives insights into expressing  $I(Z; \hat{Y})$  in terms of  $w$ . To see this clearly, let us formally state that  $D^*(\hat{y}, z)$  is indeed the optimal choice via the following theorem.

**Theorem:** The mutual information  $I(Z; \hat{Y})$ , reflected in the last line of (3.272), can be represented as the following function optimization:

$$I(Z; \hat{Y}) = H(Z) + \max_{D(\hat{y}, z): \sum_z D(\hat{y}, z) = 1} \sum_{\hat{y}, z} \mathbb{P}_{\hat{Y}, Z}(\hat{y}, z) \log D(\hat{y}, z). \quad (3.276)$$

**Proof:** The proof relies upon what we learned in Part II: the optimality condition for convex optimization. Notice that the optimization (3.276) is *convex* in  $D(\cdot, \cdot)$ , since the log function is concave and the convexity preserves under additivity. Hence, by checking the KKT condition (the optimality condition for convex optimization), one can prove that the optimal  $D(\cdot, \cdot)$  indeed respects (3.273) and (3.274). See below for details. Consider the Lagrange function:

$$\mathcal{L}(D(\hat{y}, z), v(\hat{y})) = \sum_{\hat{y}, z} \mathbb{P}_{\hat{Y}, Z}(\hat{y}, z) \log D(\hat{y}, z) + \sum_{\hat{y}} v(\hat{y}) \left( 1 - \sum_z D(\hat{y}, z) \right) \quad (3.277)$$

where  $v(\hat{y})$ 's indicate Lagrange multipliers w.r.t. the equality constraints. Consider the KKT condition:

$$\frac{d\mathcal{L}(D(\hat{y}, z), v(\hat{y}))}{dD(\hat{y}, z)} \Big|_{D=D_{\text{opt}}, v=v_{\text{opt}}} = \frac{\mathbb{P}_{\hat{Y}, Z}(\hat{y}, z)}{D_{\text{opt}}(\hat{y}, z)} - v_{\text{opt}}(\hat{y}) = 0; \quad (3.278)$$

$$\sum_z D_{\text{opt}}(\hat{y}, z) = 1. \quad (3.279)$$

So we get  $D_{\text{opt}}(\hat{y}, z) = \frac{\mathbb{P}_{\hat{Y}, Z}(\hat{y}, z)}{\nu_{\text{opt}}(\hat{y})}$ . Plugging this into (3.279), we obtain:

$$\sum_z D_{\text{opt}}(\hat{y}, z) = \frac{\sum_z \mathbb{P}_{\hat{Y}, Z}(\hat{y}, z)}{\nu_{\text{opt}}(\hat{y})} = 1, \quad (3.280)$$

which yields:

$$\nu_{\text{opt}}(\hat{y}) = \sum_z \mathbb{P}_{\hat{Y}, Z}(\hat{y}, z) = \mathbb{P}_{\hat{Y}}(\hat{y}). \quad (3.281)$$

This together with (3.278) then gives:

$$D_{\text{opt}}(\hat{y}, z) = \frac{\mathbb{P}_{\hat{Y}, Z}(\hat{y}, z)}{\nu_{\text{opt}}(\hat{y})} = \frac{\mathbb{P}_{\hat{Y}, Z}(\hat{y}, z)}{\mathbb{P}_{\hat{Y}}(\hat{y})} = D^*(\hat{y}, z). \quad (3.282)$$

This completes the proof of the theorem. ■

**How to express  $I(Z; \hat{Y})$  in terms of  $w$ ?** Are we done with expressing  $I(Z; \hat{Y})$  in terms of  $w$ ? No. This is because  $P_{\hat{Y}, Z}(\hat{y}, z)$  that appears in (3.276) is not available. To resolve this problem, we rely upon the empirical distribution instead:

$$\mathbb{Q}_{\hat{Y}, Z}(\hat{y}^{(i)}, z^{(i)}) = \frac{1}{m} \quad \forall i \in \{1, \dots, m\}.$$

In practice, the empirical distribution is very likely to be uniform, since  $\hat{y}^{(i)}$  is real-valued and hence the pair  $(\hat{y}^{(i)}, z^{(i)})$  is unique with high probability. By parametrizing the function  $D(\cdot, \cdot)$  with another, say  $\theta$ , we can approximate  $I(Z; \hat{Y})$  as:

$$I(Z; \hat{Y}) \approx H(Z) + \max_{\theta: \sum_z D_\theta(\hat{y}, z) = 1} \sum_{i=1}^m \frac{1}{m} \log D_\theta(\hat{y}^{(i)}, z^{(i)}). \quad (3.283)$$

From the above parameterization building upon the function optimization (3.276), we can now approximately express  $I(Z; \hat{Y})$  in terms of  $w$  and  $\theta$ .

**Implementable optimization (Cho et al., 2020)** Notice in (3.283) that  $H(Z)$  is irrelevant to the introduced optimization variables  $(w, \theta)$ . Hence, the mutual information (MI)-based optimization (3.270) that we started with can be (approximately) translated into:

$$\min_w \max_{\theta: \sum_z D_\theta(\hat{y}, z) = 1} \frac{1}{m} \left\{ \sum_{i=1}^m \ell_{CE}(y^{(i)}, \hat{y}^{(i)}) + \lambda \sum_{i=1}^m \log D_\theta(\hat{y}^{(i)}, z^{(i)}) \right\}. \quad (3.284)$$

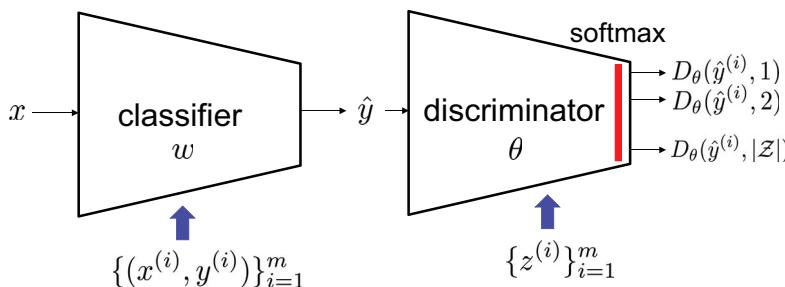
The objective function is a function of  $(w, \theta)$  and hence it is implementable, for instance, via famous neural networks. Since we have “min max”, we can apply the

variant of gradient descent that we learned in Section 2.2. That is, *alternating gradient descent*, in which given  $w$ ,  $\theta$  is updated via the inner optimization and then given the updated  $\theta$ ,  $w$  is newly updated via the outer optimization, and this process iterates until it converges.

**The architecture of the fair classifier** The architecture of the implementable optimization (3.284) is illustrated in Fig. 3.48. On top of a classifier, we introduce a new entity, called *discriminator*, which corresponds to the inner optimization. In discriminator, we wish to find  $\theta^*$  that maximizes  $\frac{1}{m} \sum_{i=1}^m \log D_\theta(\hat{y}^{(i)}, z^{(i)})$ . On the other hand, the classifier wants to *minimize* the term. Hence,  $D_\theta(\hat{y}, z)$  can be viewed as the ability to figure out  $z$  from prediction  $\hat{y}$ . Notice that the classifier wishes to minimize the ability for the purpose of fairness, while the discriminator has the opposite goal. So one natural interpretation that can be made on  $D_\theta(\hat{y}, z)$  is that it captures the probability that  $z$  is indeed the ground-truth sensitive attribute for  $\hat{y}$ . Here the softmax function is applied to ensure the sum-up-to-one constraint.

**Analogy with GANs** Since the classifier and the discriminator are competing, one can make an analogy with Goofellow's GAN, in which the generator and the discriminator also compete like a two-player game. While the fair classifier and the GAN bear strong similarity in their nature, these two are distinct in their roles. See Fig. 3.49 for the detailed distinctions.

**Look ahead** We are now done with the optimization formulation for a fair classifier. In the next section, we will study how to solve the optimization (3.284), as well as how to implement it via TensorFlow.



**Figure 3.48.** The architecture of the MI-based fair classifier. The prediction output  $\hat{y}$  is fed into the discriminator wherein the goal is to figure out sensitive attribute  $z$  from  $\hat{y}$ . The discriminator output  $D_\theta(\hat{y}, z)$  can be interpreted as the probability that  $\hat{y}$  belongs to the attribute  $z$ . Here the softmax function is applied to ensure the sum-up-to-one constraint.

MI-based fair classifier	GAN
discriminator	discriminator
Figure out sensitive attribute from prediction	<b>Goal:</b> Distinguish real samples from fake ones.
classifier	generator
Maximize prediction accuracy	Generate realistic fake samples

**Figure 3.49.** MI-based fair classifier vs. GAN: Both bear similarity in structure (as illustrated in Fig. 3.48), yet distinctions in role.

## 3.14 A Fair Classifier: TensorFlow Implementation

**Recap** Previously we formulated an optimization that respects two fairness constraints: disparate treatment (DT) and disparate impact (DI). Given  $m$  example triplets  $\{(x^{(i)}, z^{(i)}, y^{(i)})\}_{i=1}^m$ :

$$\min_w \frac{1}{m} \sum_{i=1}^m \ell_{\text{CE}}(y^{(i)}, \hat{y}^{(i)}) + \lambda \cdot I(Z; \hat{Y})$$

where  $\hat{y}^{(i)}$  indicates the classifier output, depending only on  $x^{(i)}$  (not on the sensitive attribute  $z^{(i)}$  due to the DT constraint); and  $\lambda$  is a regularization factor that balances prediction accuracy against the DI constraint, quantified as  $I(Z; \hat{Y})$ . Using the connection between mutual information and KL divergence, as well as the KKT condition (the optimality condition for convex optimization), we could approximate  $I(Z; \hat{Y})$  in the form of optimization:

$$I(Z; \hat{Y}) \approx H(Z) + \max_{\sum_z D(\hat{y}, z) = 1} \sum_{i=1}^m \frac{1}{m} \log D(\hat{y}^{(i)}, z^{(i)}). \quad (3.285)$$

We then parameterized  $D(\cdot)$  with  $\theta$  to obtain:

$$\min_w \max_{\theta: \sum_z D_\theta(\hat{y}, z) = 1} \frac{1}{m} \left\{ \sum_{i=1}^m \ell_{\text{CE}}(y^{(i)}, \hat{y}^{(i)}) + \lambda \sum_{i=1}^m \log D_\theta(\hat{y}^{(i)}, z^{(i)}) \right\}. \quad (3.286)$$

Two questions that arise are: (i) how to solve the optimization (3.286)?; and (ii) how to implement it via TensorFlow?

**Outline** In this section, we will address these two questions. What we are going to do are four folded. First we will investigate a practical algorithm that allows us to attack the optimization (3.286). We will then do a case study for the purpose of exercising the algorithm. The case study is the one that we introduced earlier: recidivism prediction. In the process, we will put a special emphasis on one implementation detail: synthesizing an *unfair* dataset that we will use in our experiments. Lastly we will learn how to implement programming via TensorFlow. For illustrative purpose, we will focus on a simple binary sensitive attribute setting.

**Observation** Let us start by translating the optimization (3.286) into the one that is more programming-friendly:

$$\begin{aligned}
& \min_w \max_{\theta: \sum_z D_\theta(\hat{y}, z) = 1} \frac{1}{m} \left\{ \sum_{i=1}^m \ell_{\text{CE}}(y^{(i)}, \hat{y}^{(i)}) + \lambda \sum_{i=1}^m \log D_\theta(\hat{y}^{(i)}, z^{(i)}) \right\} \\
& \stackrel{(a)}{=} \min_w \max_{\theta} \frac{1}{m} \left\{ \sum_{i=1}^m \ell_{\text{CE}}(y^{(i)}, \hat{y}^{(i)}) \right. \\
& \quad \left. + \lambda \left( \sum_{i: z^{(i)}=1} \log D_\theta(\hat{y}^{(i)}) + \sum_{i: z^{(i)}=0} \log (1 - D_\theta(\hat{y}^{(i)})) \right) \right\} \\
& \stackrel{(b)}{=} \min_w \max_{\theta} \frac{1}{m} \left\{ \sum_{i=1}^m \ell_{\text{CE}}(y^{(i)}, \hat{y}^{(i)}) \right. \\
& \quad \left. + \lambda \left( \sum_{i=1}^m z^{(i)} \log D_\theta(\hat{y}^{(i)}) + (1 - z^{(i)}) \log (1 - D_\theta(\hat{y}^{(i)})) \right) \right\} \\
& \stackrel{(c)}{=} \min_w \max_{\theta} \frac{1}{m} \left\{ \sum_{i=1}^m \ell_{\text{CE}}(y^{(i)}, \hat{y}^{(i)}) - \lambda \sum_{i=1}^m \ell_{\text{CE}}(z^{(i)}, D_\theta(\hat{y}^{(i)})) \right\} \\
& \stackrel{(d)}{=} \underbrace{\min_w \max_{\theta} \frac{1}{m} \left\{ \sum_{i=1}^m \ell_{\text{CE}}(y^{(i)}, G_w(x^{(i)})) - \lambda \ell_{\text{CE}}(z^{(i)}, D_\theta(G_w(x^{(i)}))) \right\}}_{=: J(w, \theta)}
\end{aligned}$$

where (a) is because we consider a binary sensitive attribute setting and we denote  $D_\theta(\hat{y}^{(i)}, 1)$  simply by  $D_\theta(\hat{y}^{(i)})$ ; (b) is due to  $z^{(i)} \in \{0, 1\}$ ; (c) follows from the definition of binary cross entropy loss  $\ell_{\text{CE}}(\cdot, \cdot)$ ; and (d) comes from  $G_w(x^{(i)}) := \hat{y}^{(i)}$ .

Notice that  $J(w, \theta)$  contains two cross entropy loss terms, each being a non-trivial function of  $G_w(\cdot)$  and/or  $D_\theta(\cdot)$ . Hence, in general,  $J(w, \theta)$  is highly non-convex in  $w$  and non-concave in  $\theta$ .

**Alternating gradient descent** Similar to the prior GAN setting in Section 3.8, what we can do in this context is to apply the only technique that we are aware of: *alternating gradient descent*. And then hope for the best. So we employ  $k : 1$  alternating gradient descent:

1. Update classifier (generator)'s weight:

$$w^{(t+1)} \leftarrow w^{(t)} - \alpha_1 \nabla_w J(w^{(t)}, \theta^{(t,k)}).$$

2. Update discriminator's weight  $k$  times while fixing  $w^{(t+1)}$ : for  $i=1:k$ ,
$$\theta^{(t+k+i)} \leftarrow \theta^{(t+k+i-1)} + \alpha_2 \nabla_{\theta} J(w^{(t+1)}, \theta^{(t+k+i-1)}).$$
3. Repeat the above.

Similar to the GAN setting, one can use the Adam optimizer possibly together with the batch version of the algorithm.

**Optimization used in our experiments** Here is the optimization that we will use in our experiments:

$$\min_w \max_{\theta} \frac{1}{m} \left\{ \sum_{i=1}^m (1 - \lambda) \ell_{\text{CE}} \left( y^{(i)}, G_w(x^{(i)}) \right) - \lambda \ell_{\text{CE}} \left( z^{(i)}, D_{\theta}(G_w(x^{(i)})) \right) \right\}. \quad (3.287)$$

In order to restrict the range of  $\lambda$  into  $0 \leq \lambda \leq 1$ , we apply the  $(1 - \lambda)$  factor to the loss term w.r.t. prediction accuracy.

Like the prior GAN setting, let us define two loss terms. One is “classifier (or generator) loss”:

$$\underbrace{\min_w \max_{\theta} \frac{1}{m} \left\{ \sum_{i=1}^m (1 - \lambda) \ell_{\text{CE}} \left( y^{(i)}, G_w(x^{(i)}) \right) - \lambda \ell_{\text{CE}} \left( z^{(i)}, D_{\theta}(G_w(x^{(i)})) \right) \right\}}_{\text{"classifier (generator) loss"}}$$

Given  $w$ , discriminator wishes to maximize:

$$\max_{\theta} - \frac{\lambda}{m} \sum_{i=1}^m \ell_{\text{CE}} \left( z^{(i)}, D_{\theta}(G_w(x^{(i)})) \right).$$

This is equivalent to minimizing the minus of the objective:

$$\underbrace{\min_{\theta} \frac{\lambda}{m} \sum_{i=1}^m \ell_{\text{CE}} \left( z^{(i)}, D_{\theta}(G_w(x^{(i)})) \right)}_{\text{"discriminator loss"}}. \quad (3.288)$$

This is how we define “discriminator loss”.

**Performance metrics** Unlike to the prior settings (supervised learning and GAN), here we need to introduce another performance metric that captures the degree of fairness. To this end, let us first define the hard-decision value of the prediction output w.r.t. a test example:

$$\tilde{Y}_{\text{test}} := \mathbf{1}\{\hat{Y}_{\text{test}} \geq 0.5\}.$$

The test accuracy is then defined as:

$$\frac{1}{m_{\text{test}}} \sum_{i=1}^{m_{\text{test}}} \mathbf{1}\{\hat{y}_{\text{test}}^{(i)} = \tilde{Y}_{\text{test}}^{(i)}\}$$

where  $m_{\text{test}}$  denotes the number of test examples. This is an empirical version of the ground truth  $\mathbb{P}(Y_{\text{test}} = \tilde{Y}_{\text{test}})$ .

How to define a fairness-related performance metric? Recall the mathematical definition of DI:

$$\text{DI} := \min \left( \frac{\mathbb{P}(\tilde{Y} = 1|Z = 0)}{\mathbb{P}(\tilde{Y} = 1|Z = 1)}, \frac{\mathbb{P}(\tilde{Y} = 1|Z = 1)}{\mathbb{P}(\tilde{Y} = 1|Z = 0)} \right). \quad (3.289)$$

Here you may wonder how to compute two probabilities of interest:  $\mathbb{P}(\tilde{Y} = 1|Z = 0)$  and  $\mathbb{P}(\tilde{Y} = 1|Z = 1)$ . Using their empirical versions together with the Law of Large Numbers (i.e., the empirical mean converges to the true mean as the number of samples tends to infinity), we can estimate them. For instance,

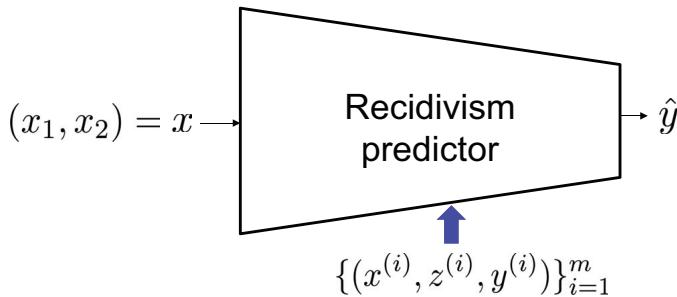
$$\mathbb{P}(\tilde{Y} = 1|Z = 0) = \frac{\mathbb{P}(\tilde{Y} = 1, Z = 0)}{\mathbb{P}(Z = 0)} \approx \frac{\sum_{i=1}^{m_{\text{test}}} \mathbf{1}\{\tilde{Y}_{\text{test}}^{(i)} = 1, z_{\text{test}}^{(i)} = 0\}}{\sum_{i=1}^{m_{\text{test}}} \mathbf{1}\{z_{\text{test}}^{(i)} = 0\}}$$

where the first equality is due to the definition of conditional probability and the second approximation comes from the Law of Large Numbers. The above approximation is getting more and more accurate as  $m_{\text{test}}$  gets larger. Similarly we can approximate the other interested probability  $\mathbb{P}(\tilde{Y} = 1|Z = 1)$ . This way, we can evaluate DI (3.289).

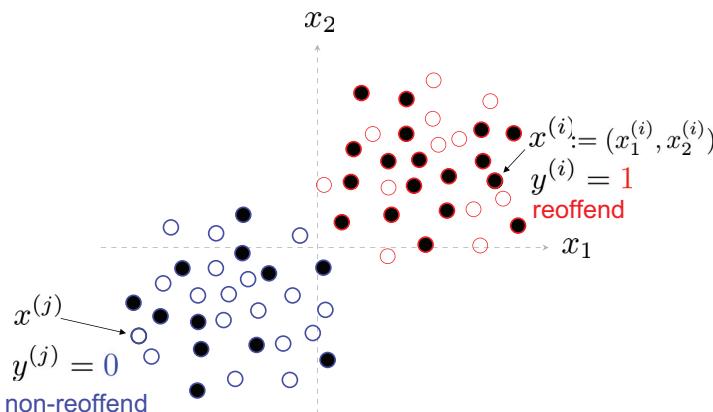
**A case study** Let us exercise what we have learned so far with a simple example. As a case study, we consider the same simple setting that we introduced earlier: recidivism prediction, wherein the task is to predict if an interested individual reoffends within two years, as illustrated in Fig. 3.50.

**Synthesizing an unfair dataset** One thing that we need to be careful about in the context of fair machine learning is w.r.t. *unfair* datasets. For simplicity, we will employ a *synthetic* dataset, not a real-world dataset. In fact, there is a real-world dataset that concerns the recidivism prediction, called COMPAS ([Angwin et al., 2020](#)). But this contains many attributes, so it is a bit complicated. Hence, we will take a particular yet simple method to synthesize a much simpler unfair dataset.

Recall the visualization of an unfair data scenario that we investigated in Section 3.12 and will form the basis of our synthetic dataset (to be explained in the sequel); see Fig. 3.51 for the visualization. In Fig. 3.51, a hollowed (or black-colored-solid) circle indicates a data point of an individual with white (or black)



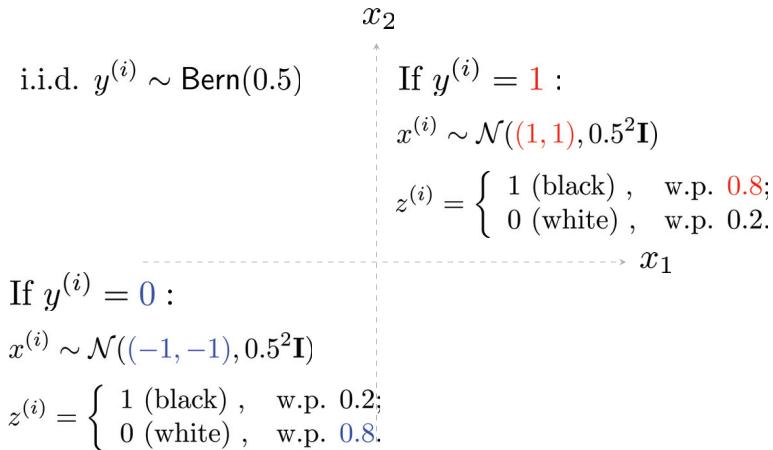
**Figure 3.50.** Predicting a recidivism score  $\hat{y}$  from  $x = (x_1, x_2)$ . Here  $x_1$  indicates the number of prior criminal records;  $x_2$  denotes a criminal type: misdemeanor or felony; and  $z$  is a race type among white ( $z = 0$ ) and black ( $z = 1$ ).



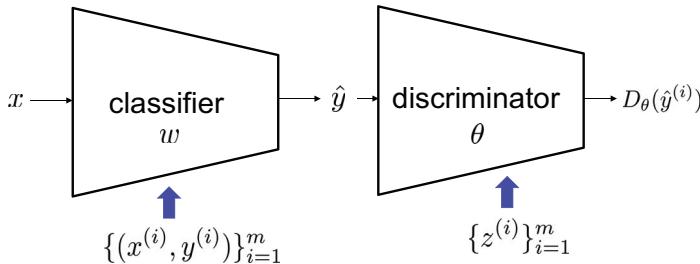
**Figure 3.51.** Visualization of a historically biased dataset: A hollowed (or black-colored-solid) circle indicates a data point of an individual with white (or black) race; the red (or blue) colored edge denotes  $y = 1$  reoffending (or  $y = 1$  non-reoffending) label.

race; and the red (or blue) colored edge (ring) denotes the event that the interested individual **reoffends** (or **non-reoffends**) within two years. This is indeed an *unfair* scenario: for  $y = 1$ , there are more black-colored-solid circles than hollowed ones; similarly for  $y = 0$ , there are more hollowed circles relative to solid ones.

To generate such an unfair dataset, we employ a simple method. See Fig. 3.52 for illustration of the method. We first generate  $m$  labels  $y^{(i)}$ 's so that they are independent and identically distributed (i.i.d.), each being according to a uniform distribution, i.e.,  $\mathbb{P}(Y = 1) = \mathbb{P}(Y = 0) = \frac{1}{2}$ . We denote the uniform distribution by  $\text{Bern}(\frac{1}{2})$ , since the associated random variable is known as a Bernoulli random variable. Here the number  $\frac{1}{2}$  inside the parenthesis indicates the probability of a Bernoulli random variable being 1. For indices of positive examples ( $y^{(i)} = 1$ ), we then generate i.i.d.  $x^{(i)}$ 's according to  $\mathcal{N}((1, 1), 0.5^2 \mathbf{I})$ ; and i.i.d.  $z^{(i)}$ 's as per  $\text{Bern}(0.8)$ , meaning that 80% are blacks ( $z = 1$ ) and 20% are whites ( $z = 0$ ).



**Figure 3.52.** A simple way to synthesize an unfair dataset.

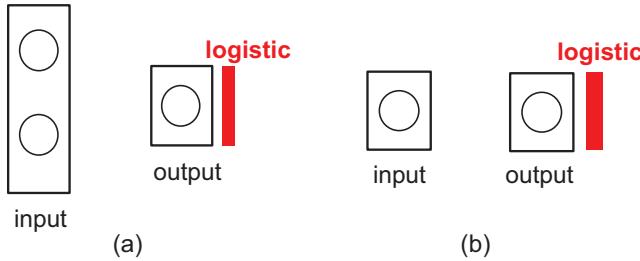


**Figure 3.53.** The architecture of the MI-based fair classifier.

among the positive individuals. Notice that the generation of  $x^{(i)}$ 's is not quite realistic. The first and second components in  $x^{(i)}$  do not precisely capture the number of priors and a criminal type. You can view this generation as sort of a crude *abstraction* of the realistic data. On the other hand, for negative examples ( $y^{(i)} = 0$ ), we generate i.i.d.  $(x^{(i)}, z^{(i)})$ 's with different distributions:  $x^{(i)} \sim \mathcal{N}((-1, -1), 0.5^2 \mathbf{I})$  and  $z^{(i)} \sim \text{Bern}(0.2)$ , meaning that 20% are blacks ( $z = 1$ ) and 80% are whites ( $z = 0$ ). This way,  $z^{(i)} \sim \text{Bern}(\frac{1}{2})$ . This is because

$$\begin{aligned} \mathbb{P}(Z = 1) &\stackrel{(a)}{=} \mathbb{P}(Y = 1)\mathbb{P}(Z = 1|Y = 1) + \mathbb{P}(Y = 0)\mathbb{P}(Z = 1|Y = 0) \\ &\stackrel{(b)}{=} \frac{1}{2} \cdot 0.8 + \frac{1}{2} \cdot 0.2 = \frac{1}{2} \end{aligned}$$

where (a) follows from the total probability law and the definition of conditional probability; and (b) is due to the rule of the data generation method employed. Here  $Z$  and  $Y$  denote generic random variables for  $z^{(i)}$  and  $y^{(i)}$ , respectively.



**Figure 3.54.** Models for (a) the classifier and (b) the discriminator.

**Model architecture** Fig. 3.53 illustrates the architecture of the MI-based fair classifier. Since we focus on the binary sensitive attribute, we have a single output  $D_\theta(\hat{y})$  in the discriminator. For models of the classifier and discriminator, we employ very simple single-layer neural networks with logistic activation in the output layer; see Fig. 3.54.

**TensorFlow: Synthesizing an unfair dataset** Let us discuss how to implement such details via TensorFlow. First consider the synthesis of an unfair dataset. To generate i.i.d. Bernoulli (binary) random variables for labels, we use:

```
import numpy as np  
y_train = np.random.binomial(1,0.5,size=(train_size,))
```

where the first two arguments of (1,0.5) specify  $\text{Bern}(0.5)$ ; and the null space followed by `train_size` indicates a single dimension. Remember we generate i.i.d. Gaussian random variables for  $x^{(i)}$ 's. To this end, one can use:

```
x = np.random.normal(loc=(1,1),scale=0.5, size=(train_size,2))
```

**TensorFlow: Optimizers for classifier & discriminator** For classifier, we use the Adam optimizer with the learning rate of 0.005 and  $(\beta_1, \beta_2) = (0.9, 0.999)$ . For discriminator, we use another much simpler optimizer, named Stochastic Gradient Descent, SGD for short. SGD is the naive gradient descent yet with a batch size of 1. We use SGD with the learning rate of 0.005.

```
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.optimizers import SGD
adam=Adam(learning_rate=0.005,beta_1=0.9, beta_2=0.999)
sgd=SGD(learning_rate=0.005)
```

**TensorFlow: Classifier (generator) loss** Recall the optimization problem of interest:

$$\min_w \max_{\theta} \frac{1}{m} \left\{ \sum_{i=1}^m (1 - \lambda) \ell_{\text{CE}} \left( y^{(i)}, G_w(x^{(i)}) \right) \right. \\ \left. - \lambda \sum_{i=1}^m \ell_{\text{CE}} \left( z^{(i)}, D_{\theta}(G_w(x^{(i)})) \right) \right\}.$$

To implement the classifier loss (the objective in the above), we use:

```
from tensorflow.keras.losses import BinaryCrossentropy
CE_loss = BinaryCrossentropy(from_logits=False)
p_loss = CE_loss(y_pred,y_train)
f_loss = CE_loss(discriminator(y_pred),z_train)
c_loss = (1-lamb)*p_loss - lamb*f_loss
```

where  $y_{\text{pred}}$  indicates the classifier output;  $y_{\text{train}}$  denotes a label; and  $z_{\text{train}}$  is a binary sensitive attribute.

**TensorFlow: Discriminator loss** Recall the Discriminator loss that we defined in (3.288):

$$\min_{\theta} \frac{\lambda}{m} \sum_{i=1}^m \ell_{\text{CE}} \left( z^{(i)}, D_{\theta}(G_w(x^{(i)})) \right).$$

To implement this, we use:

```
f_loss = CE_loss(discriminator(y_pred),z_train)
d_loss = lamb*f_loss
```

**TensorFlow: Evaluation** Recall the DI performance:

$$\text{DI} := \min \left( \frac{\mathbb{P}(\tilde{Y} = 1|Z = 0)}{\mathbb{P}(\tilde{Y} = 1|Z = 1)}, \frac{\mathbb{P}(\tilde{Y} = 1|Z = 1)}{\mathbb{P}(\tilde{Y} = 1|Z = 0)} \right).$$

To evaluate the DI performance, we rely on the following approximation:

$$\mathbb{P}(\tilde{Y} = 1|Z = 0) \approx \frac{\sum_{i=1}^{m_{\text{test}}} \mathbf{1}\{\tilde{y}_{\text{test}}^{(i)} = 1, z_{\text{test}}^{(i)} = 0\}}{\sum_{i=1}^{m_{\text{test}}} \mathbf{1}\{z_{\text{test}}^{(i)} = 0\}}.$$

Here is how to implement this in detail:

```
import numpy as np
y_tilde = (y_pred>0.5).int().squeeze()
z0_ind = (z_train == 0.0)
z1_ind = (z_train == 1.0)
z0_sum = int(np.sum(z0_ind))
z1_sum = int(np.sum(z1_ind))
P_y1_z0 = float(np.sum((y_tilde==1)[z0_ind]))/z0_sum
P_y1_z1 = float(np.sum((y_tilde==1)[z1_ind]))/z1_sum
```

**Closing** Let us conclude the book. In Part I, we investigated several instances of convex optimization problems, ranging from LP, Least Squares, QP, SOCP, and all the way up to SDP. We studied how such problems are categorized, as well as how to formulate some real-world problems into such specialized problems via some translation techniques possibly aided by matrix-vector notations. For some certain settings including LP, unconstrained optimization and equality-constrained QP, we also studied how to solve the problems explicitly.

In Part II, we studied two important theorems: (1) strong duality theorem; (2) weak duality theorem. With the strong duality theorem, we came up with a generic algorithm which provides detailed guidelines as to how to solve arbitrary convex optimization problems: the interior point method. With the weak duality theorem, we investigated a certain yet powerful method, called Lagrange relaxation, which can provide reasonably-good approximation solutions for a variety of *non-convex* problems.

In Part III, we explored one recent killer application where optimization tools that we learned play central roles: Machine learning. In particular, we explored two certain yet popular methodologies of machine learning: (1) supervised learning; and (2) unsupervised learning. For supervised learning, we put an emphasis on deep learning, which is based on deep neural network architectures which received significant attention recently. We found that the optimization tools and concepts that we learned are instrumental particularly in choosing objective functions as well as gaining algorithmic insights. As for unsupervised learning, we investigated the most fundamental learning method, called generative modeling, and then studied one specific yet powerful framework for generative modeling, named GANs. In this context, we observed that the duality theorems play a crucial role in enabling a practical implementation for the state-of-the-art GAN, which is WGAN. As the last application, we explored fair machine learning to demonstrate the power of the regularization technique and the KKT condition.

It is no doubt that tools for convex optimization are very powerful. The usefulness has already been proved by many researchers working on a wide variety of fields. While this book puts an emphasis on a particular application (machine learning), it is shown to have much broader applicability. So we hope you would find all of these useful in your own research field.

## Problem Set 11

---

**Prob 11.1 (Equalized Odds)** In Section 3.12, we studied two fairness concepts: (i) disparate treatment; and (ii) disparate impact. In this problem, we explore another prominent fairness notion that arises in the field: *Equalized Odds (EO for short)* (Hardt *et al.*, 2016). Let  $Z \in \mathcal{Z}$  be a sensitive attribute. Let  $Y$  and  $\hat{Y}$  be the ground-truth label and its prediction.

- (a) For illustrative purpose, let us investigate a simple setting where  $Z$  and  $Y$  are binary. Let  $\tilde{Y} = \mathbf{1}\{\hat{Y} \geq 0.5\}$ . For this setting, EO is defined as:

$$\text{EO} := \min_{y \in \{0,1\}} \min_{z \in \{0,1\}} \frac{\mathbb{P}(\tilde{Y} = 1 | Y = y, Z = 1 - z)}{\mathbb{P}(\tilde{Y} = 1 | Y = y, Z = z)}. \quad (3.290)$$

Show that  $I(Z; \hat{Y} | Y) = 0$  implies  $\text{EO} = 1$ .

- (b) Suppose now that  $Z$  and  $Y$  are not necessarily binary. The conditional mutual information is defined as:

$$I(Z; \hat{Y} | Y) := \text{KLD}(\mathbb{P}_{\hat{Y}, Z|Y}, \mathbb{P}_{\hat{Y}|Y} \mathbb{P}_{Z|Y})$$

where  $\mathbb{P}_{\hat{Y}, Z|Y}$ ,  $\mathbb{P}_{\hat{Y}|Y}$  and  $\mathbb{P}_{Z|Y}$  indicate the conditional probability of  $(\hat{Y}, Z)$ ,  $\hat{Y}$ , and  $Z$ , respectively, given  $Y$ . Using this definition, show that

$$I(Z; \hat{Y} | Y) = \sum_{y \in \mathcal{Y}, \hat{y} \in \hat{\mathcal{Y}}, z \in \mathcal{Z}} \mathbb{P}_{\hat{Y}, Z, Y}(\hat{y}, z, y) \log \frac{\mathbb{P}_{\hat{Y}, Z|y}(\hat{y}, z)}{\mathbb{P}_{\hat{Y}|y}(\hat{y})} + H(Z|Y) \quad (3.291)$$

where  $\mathbb{P}_{\hat{Y}, Z, Y}$  indicates the joint distribution of  $(\hat{Y}, Z, Y)$ ; and  $\mathbb{P}_{\hat{Y}, Z|y}$  and  $\mathbb{P}_{\hat{Y}|y}$  denote the conditional distributions of  $(\hat{Y}, Z)$  and  $\hat{Y}$ , respectively, given  $Y = y$ .

- (c) Show that

$$\begin{aligned} I(Z; \hat{Y} | Y) &= H(Z|Y) \\ &+ \max_{D(\hat{y}, z, y) : \sum_{z \in \mathcal{Z}} D(\hat{y}, z, y) = 1} \\ &\times \sum_{\hat{y} \in \hat{\mathcal{Y}}, y \in \mathcal{Y}, z \in \mathcal{Z}} \mathbb{P}_{\hat{Y}, Z, Y}(\hat{y}, z, y) \log D(\hat{y}, z, y). \end{aligned} \quad (3.292)$$

- (d) Explain the rationale behind the following approximation:

$$I(Z; \hat{Y}|Y) \approx H(Z|Y)$$

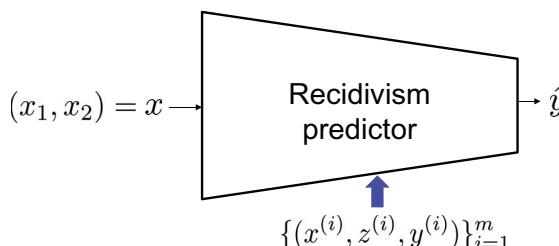
$$+ \max_{D(\hat{y}, z, y) : \sum_{z \in \mathcal{Z}} D(\hat{y}, z, y) = 1} \sum_{i=1}^m \frac{1}{m} \log D(\hat{y}^{(i)}, z^{(i)}, y^{(i)}). \quad (3.293)$$

- (e) Formulate an optimization for a fair classifier that attempts to minimize both predication accuracy and the approximated  $I(Z; \hat{Y}|Y)$ , reflected in (3.293). Use a notation  $\lambda$  for a regularization factor that balances prediction accuracy against the quantified fairness constraint. Also draw the classifier-&-discriminator architecture which respects the formulated optimization.

**Prob 11.2 (A variant of the MI-based fair classifier)** Let  $Z \in \{0, 1\}$  be a binary sensitive attribute. Let  $Y$  and  $\hat{Y}$  be the ground-truth label and its prediction of a classifier. Let  $\tilde{Y} = \mathbf{1}\{\hat{Y} \geq 0.5\}$ .

- (a) Show that  $I(Z; \tilde{Y}) = 0$  is a necessary and sufficient condition for  $\text{DI} = 1$ .
- (b) Approximate  $I(Z; \tilde{Y})$  as claimed in part (d) in Prob 11.1. Also explain the rationale behind the approximation.
- (c) Formulate an optimization for a fair classifier that attempts to minimize both prediction accuracy and the approximated  $I(Z; \tilde{Y})$ , done in the prior part. Use a notation  $\lambda$  for a regularization factor that balances prediction accuracy against the fairness constraint. Also draw the classifier-&-discriminator architecture which respects the formulated optimization.

**Prob 11.3 (TensorFlow implementation of the MI-based fair classifier)** Consider the MI-based fair classifier that we learned in Sections 3.13 and 3.14. In this problem, you are asked to build a simple fair classifier that predicts recidivism scores of individuals with prior criminal records. See Fig. 3.55. We employ very



**Figure 3.55.** Predicting a recidivism score  $\hat{y}$  from  $x = (x_1, x_2)$ . Here  $x_1$  indicates the number of prior criminal records;  $x_2$  denotes a criminal type: misdemeanor or felony; and  $z$  is a race type among white ( $z = 0$ ) and black ( $z = 1$ ).

simple single-layer neural networks for classifier (generator) and discriminator with logistic activation in the output layer.

- (a) (*Unfair dataset synthesis*) Explain how an unfair dataset is generated in the following code:

```
import numpy as np
n_samples = 2000
p = 0.8
# numbers of positive and negative examples
n_Y1 = int(n_samples*0.5)
n_Y0 = n_samples - n_Y1
# generate positive samples
Y1 = np.ones(n_Y1)
X1 = np.random.normal(loc=[1,1],scale=0.5,
                      size=(n_Y1,2))
Z1 = np.random.binomial(1,p,size=(n_Y1,))
# generate negative samples
Y0 = np.zeros(n_Y0)
X0 = np.random.normal(loc=[-1,-1],scale=0.5,
                      size=(n_Y0,2))
Z0 = np.random.binomial(1,1-p,size=(n_Y0,))
# merge
Y = np.concatenate((Y1,Y0))
X = np.concatenate((X1,X0))
Z = np.concatenate((Z1,Z0))
Y = Y.astype(np.float32)
X = X.astype(np.float32)
Z = Z.astype(np.float32)
# shuffle and split into train & test data
shuffle = np.random.permutation(n_samples)
X_train = X[shuffle][:int(n_samples*0.8)]
Y_train = Y[shuffle][:int(n_samples*0.8)]
Z_train = Z[shuffle][:int(n_samples*0.8)]
X_test = X[shuffle][int(n_samples*0.8):]
Y_test = Y[shuffle][int(n_samples*0.8):]
Z_test = Z[shuffle][int(n_samples*0.8):]
```

- (b) (*Data visualization*) Using the following code or otherwise, plot randomly sampled data points (say 200 random points) among the entire data points generated in part (a).

```
import matplotlib.pyplot as plt
# randomly select the number n_s of samples
n_s = 200
Xs = X_train[:n_s]
Ys = Y_train[:n_s]
```

```

Zs = Z_train[:n_s]
# choose part of X and Y assiated with a certain Z
X_Z0 = Xs[Zs==0.0]
X_Z1 = Xs[Zs==1.0]
Y_Z0 = Ys[Zs==0.0]
Y_Z1 = Ys[Zs==1.0]
# plot
plt.figure(figsize=(14,10))
plt.scatter(
    X_Z0[Y_Z0==1.0][:,0], X_Z0[Y_Z0==1.0][:,1],
    color='red',marker='o',facecolors='none',
    s=120, linewidth=1.5, label='White reoffend')
plt.scatter(
    X_Z0[Y_Z0==0.0][:,0], X_Z0[Y_Z0==0.0][:,1],
    color='blue',marker='o',facecolors='none',
    s=120, linewidth=1.5, label='White non-reoffend')
plt.scatter(
    X_Z1[Y_Z1==1.0][:,0], X_Z1[Y_Z1==1.0][:,1],
    color='red',marker='o',facecolors='black',
    s=120, linewidth=1.5, label='Black reoffend')
plt.scatter(
    X_Z1[Y_Z1==0.0][:,0], X_Z1[Y_Z1==0.0][:,1],
    color='blue',marker='o',facecolors='black',
    s=120, linewidth=1.5, label='Black non-reoffend')
plt.legend(fontsize=16)

```

- (c) (*Classifier & discriminator*) Draw block diagrams for the classifier and the discriminator implemented by the following:

```

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

classifier=Sequential()
classifier.add(Dense(1,input_dim=2,
                     activation='sigmoid'))
discriminator=Sequential()
discriminator.add(Dense(1,input_dim=1,
                      activation='sigmoid'))

```

- (d) (*Optimizers and loss functions*) Explain how the optimizers and loss functions for the discriminator and the classifier are implemented in the following code. Also draw a block diagram for the GAN model implemented as the name of gan.

```

from tensorflow.keras.layers import Input
from tensorflow.keras.models import Model
from tensorflow.keras.optimizers import Adam

```

```
from tensorflow.keras.optimizers import SGD
from tensorflow.keras.losses import BinaryCrossentropy
from tensorflow.keras.layers import Concatenate

# optimizers for classifier & discriminator
c_opt=Adam(learning_rate=0.005,beta_1=0.9,beta_2=0.999)
d_opt=SGD(learning_rate=0.005)

# define dicriminator loss
def d_loss(y_true,y_pred):
    CE_loss = BinaryCrossentropy(from_logits=False)
    lamb = 0.1
    return lamb*CE_loss(y_pred,y_true)

# discriminator compile
discriminator.compile(loss=d_loss, optimizer=d_opt)

# define classifier (generator) loss
def c_loss(y_true,y_pred):
    # y_true[:,0]: Y_train (label)
    # y_true[:,1]: Z_train (sensitive attribute)
    # y_pred[:,0]: classifier output G(x)
    # y_pred[:,1]: discriminator output fed by
    #           classifier output D(G(x))
    CE_loss = BinaryCrossentropy(from_logits=False)
    lamb = 0.1
    p_loss = CE_loss(y_pred[:,0],y_true[:,0])
    f_loss = CE_loss(y_pred[:,1],y_true[:,1])
    return (1-lamb)*p_loss - lamb*f_loss

# define the GAN model
# input: x
# output: [G(x), D(G(x))]
discriminator.trainable = False
gan_input = Input(shape=(2,))
Gx = classifier(inputs=gan_input)
DGx = discriminator(Gx)
output = Concatenate()([Gx,DGx])
gan = Model(gan_input, output)

# The GAN model compile
gan.compile(loss=c_loss, optimizer=c_opt)
```

- (e) (*Training*) Explain how classifier and discriminator are trained in the following code:

```
import numpy as np

EPOCHS = 400
k=2 # k:1 alternating gradient descent
c_losses = []
d_losses = []

for epoch in range(1,EPOCHS+1):
    #####
    # train discriminator
    #####
    # input for discriminator
    d_input = classifier.predict(X_train)
    # label for discriminator
    d_label = Z_train
    # train discriminator
    d_loss = discriminator.train_on_batch(d_input,d_label)

    #####
    # train classifier
    #####
    if epoch % k == 0: # train once every k steps
        # label for classifier
        # 1st component: Y_train
        # 2nd component: Z_train (sensitive attribute)
        c_label = np.zeros((len(Y_train),2))
        c_label[:,0] = Y_train
        c_label[:,1] = Z_train
        # train classifier
        c_loss = gan.train_on_batch(X_train,c_label)

        c_losses.append(c_loss)
        d_losses.append(d_loss)
```

- (f) (*Evaluation*) Suppose we train classifier and discriminator using the code in part (e) with EPOCHS=400. Plot the tradeoff performance between test accuracy and DI by sweeping  $\lambda$  from 0 to 1. Also include the Python script.

## Appendix A

# Python Basics

## A.1 Jupyter Notebook

---

**Outline** Python requires another software platform which serves to play around it. The platform is Jupyter notebook. In this section, we will learn about some basic stuffs regarding Jupyter notebook. What we are going to do are four folded. First we will figure out the role of Jupyter notebook in light of Python. We will then investigate how to install the software, as well as how to launch a new file for scripting a code. Next, we will study some useful interfaces which enable an easy scripting of a Python code. Lastly, we will introduce several shortcuts that are frequently used in writing and executing a code.

**What is Jupyter notebook?** Jupyter notebook is a powerful tool for writing and running a Python code. As you can see below, the key benefit of the tool is that we are able to execute *each line* of the code rather than the *entire* code. Hence, we are particularly benefiting from an easy debugging especially when the code is very long.

```
a=1  
b=2  
a+b
```

3

a=1

b=2

a+b

3

There are two typical ways to use Jupyter notebook. The first runs a code based on the server (or cloud) machine. The second is via a local machine. Here we will present the second way.

**Installation & launch** The use of local machine requires the installation of a popular software tool, named Anaconda. You can download and install the latest version of it via

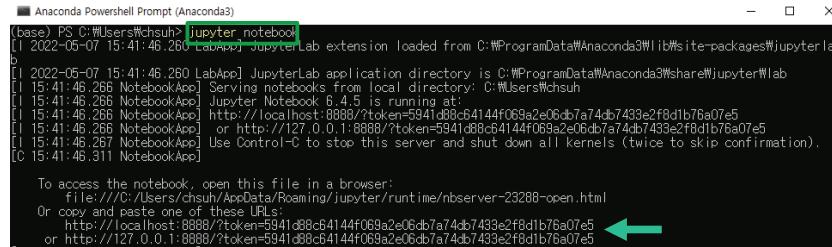
<https://www.anaconda.com/products/individual>

You can choose one of the Anaconda installers depending on the operating system of your machine. Please see the three versions presented in Fig. A.1. During installation, you may encounter some errors. One error that often occurs is regarding ‘non-ascil character’. To resolve the error, you should make sure that the name of your destination folder path for Anaconda does not include any non-ascil character like Korean. Another error message that you may see is about permission for access to the indicated path. To avoid this, run the Anaconda installer under the ‘run as administrator’ mode. The mode can be seen by right-clicking the downloaded execution file.

In order to launch Jupyter notebook, you can use anaconda prompt (for Windows) or terminal (for mac and linux). The way it works is very simple. You can just type Jupyter notebook in the prompt and then press Enter. Then, a Jupyter notebook window will pop up accordingly. If the window does not appear



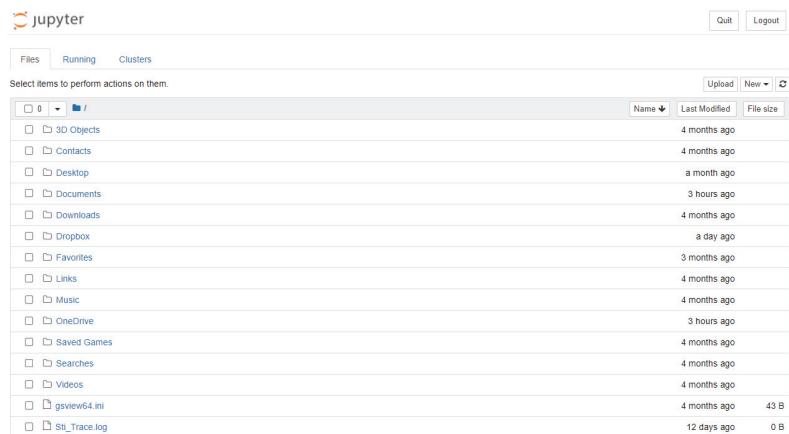
**Figure A.1.** Three versions of Anaconda installers.



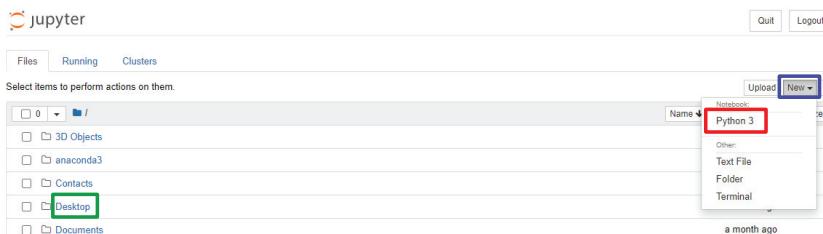
```
[base] PS C:\Users\chsuh> jupyter notebook
[1] 2022-05-07 15:41:46.260 LabApp JupyterLab extension loaded from C:\ProgramData\Anaconda3\lib\site-packages\jupyterlab
[1] 2022-05-07 15:41:46.260 LabApp JupyterLab application directory is C:\ProgramData\Anaconda3\share\jupyter\lab
[1] 2022-05-07 15:41:46.266 NotebookApp Serving notebooks from local directory: C:\Users\chsuh
[1] 2022-05-07 15:41:46.266 NotebookApp Jupyter Notebook 6.4.5 is running at:
[1] 2022-05-07 15:41:46.266 NotebookApp http://localhost:8888/?token=5941d88c64144f069a2e06db7a74db7433e2f8d1b76a07e5
[1] 2022-05-07 15:41:46.267 NotebookApp or http://127.0.0.1:8888/?token=5941d88c64144f069a2e06db7a74db7433e2f8d1b76a07e5
[1] 2022-05-07 15:41:46.311 NotebookApp Use Control-C to stop this server and shut down all kernels (twice to skip confirmation).
```

To access the notebook, open this file in a browser:  
file:///C:/Users/chsuh/AppData/Roaming/jupyter/runtime/nbserver-23288-open.html  
Or copy and paste one of these URLs:  
http://localhost:8888/?token=5941d88c64144f069a2e06db7a74db7433e2f8d1b76a07e5 ←  
or http://127.0.0.1:8888/?token=5941d88c64144f069a2e06db7a74db7433e2f8d1b76a07e5

**Figure A.2.** How to launch Jupyter notebook in the Anaconda prompt.



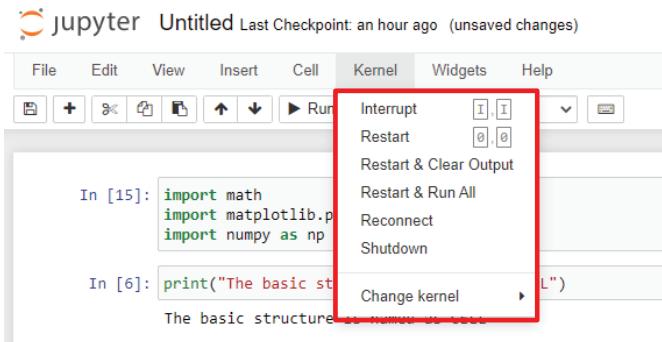
**Figure A.3.** Web browser of a successfully launched Jupyter notebook.



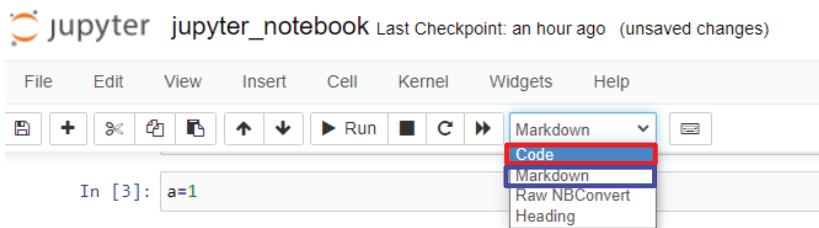
**Figure A.4.** How to create a Jupyter notebook file on the web browser.

automatically, you can instead copy and paste the URL (indicated by the arrow in Fig. A.2) on your web browser, so as to manually open it. If it works properly, you should be able to see the window like the one in Fig. A.3.

Creating a new notebook file is also simple. First navigate a folder in which you want to save a notebook file. Next you can click the New tap placed on the top right (marked in a blue box) and then click the Python 3 tap (as indicated in a red box). See Fig. A.4 for the location of the taps.



**Figure A.5.** Kernel is a computational engine which runs the code. There are several relevant functions under the Kernel tap.



**Figure A.6.** How to choose the Code or Markdown option in the edit mode.

**Interface** In Jupyter notebook, there are two key components required to run a code. The first is a computational engine which does execute the code. The engine is named Kernel and it can be controlled via several functions provided in the Kernel tab. See Fig. A.5 for details.

The second component is an entity, called cell, in which you can write a script. The cell consists of two modes. The first is so called the edit mode which allows you to type either: (i) a code script for running a program; or any text like a normal text editor. The code script can be written under the Code tap (marked in a red box) as illustrated in Fig. A.6. Text-editing can be done under the Markdown tap, marked in a blue box. The other mode is the one, named the command mode. Under this mode, we can edit a notebook as a whole. For instance, we can copy or delete some cells, and move around cells under the command mode.

**Shortcuts** There are many shortcuts that are quite instrumental in editing and navigating a notebook. Here we emphasize three types of shortcuts frequently used. The first is a set of the shortcuts for changing a state across the edit and command modes. We type Esc for changing from the edit to command modes. We use Enter for the other way around. The second is for inserting or deleting a cell. The shortcut,

a, is for inserting a new cell *above* the current cell. Another shortcut b plays a similar role, yet inserting it *below* the current cell. d+d is for deleting a cell. Notice that these should be typed under the command mode to serve proper roles. The last is a set of the shortcuts for executing a cell. Arrow keys are used to move around distinct cells. Shift + Enter is for running the current cell (and move to the next cell). In order to stay in the current cell (even after execution), we use Ctrl + Enter instead.

## A.2 Basic Python Syntaxes

**Outline** In this section, we will learn some basic Python syntaxes required to write a script for convex optimization problems of this book's focus. In particular, three basic concepts are emphasized: (i) class; (ii) package; and (iii) function. We will also introduce a collection of optimization-related Python packages.

### A.2.1 Data Structure

There are two prominent data-structure components in Python: (i) list; and (ii) set.

**(i) List:** List is a built-in data type which allows us to store multiple elements in a single variable. The elements are listed with an order and the list allows for duplication. Please see below for some popular use.

```
x = [1, 2, 3, 4] # construct a simple list  
print(x)
```

[1, 2, 3, 4]

```
x.append(5) # add an item  
print(x)
```

[1, 2, 3, 4, 5]

```
x.pop() # delete an item located in the last  
print(x)
```

[1, 2, 3, 4]

```
# checking if a particular element exists in the list  
if 3 in x:  
    print(True)  
if 5 in x:  
    print(True)  
else:  
    print(False)
```

True  
False

```
# A single-line construction of a list
y = [x for x in range(1,10)]
print(y)
```

[1, 2, 3, 4, 5, 6, 7, 8, 9]

```
# Retrieving all the elements through a "for" loop
for i in x:
    print(i)
```

1  
2  
3  
4

(ii) **Set:** Set is another built-in data type which plays a similar role as List. Two key distinctions are: (i) it is unordered; and (ii) it does not allow for duplication. See below for some examples of how to use.

```
x = set({1, 2, 3}) # construct a set
print(f"x: {x}, type of x: {type(x)}")
```

x: {1, 2, 3}, type of x: <class 'set'>

Here the f in front of strings in the print command tells Python to look at the values inside {}.

```
x.add(1) # add an existing item
print(x)
```

{1, 2, 3}

```
x.add(4) # add a new item
print(x)
```

{1, 2, 3, 4}

```
# checking if a particular element exists in the list
if 1 in x:
    print(True)
if 5 in x:
    print(True)
else:
    print(False)
```

True  
False

```
# Retrieving all the elements through a "for" loop
```

```
for i in x:  
    print(i)
```

1  
2  
3  
4

## A.2.2 Package

We will investigate five popular packages which are particularly instrumental in scripting a code for convex optimization problems: (i) math; (ii) random; (iii) itertools; (iv) numpy; and (v) scipy.stats.

**(i) math:** This module provides a collection of useful math expressions such as exponent, log, square root and power. See some relevant examples below.

```
import math
```

```
math.exp(1) # exp(x)
```

2.718281828459045

```
print(math.log(1, 10)) # log(x, base)  
print(math.log(math.exp(20))) # base-e logarithm  
print(math.log2(4)) # base-2 logarithm  
print(math.log10(1000)) # base-10 logarithm
```

0.0  
20.0  
2.0  
3.0

```
print(math.sqrt(16)) # square root  
print(math.pow(2,4)) # x raised to y (same as x**y)  
print(2**4)
```

4.0  
16.0  
16

```
print(math.cos(math.pi)) # cosine of x radians  
print(math.dist([1,2],[3,4])) # Euclidean distance
```

-1.0  
2.8284271247461903

```
# The erf() function can be used to compute traditional  
# statistical functions such as the CDF of  
# the standard Gaussian distribution  
def phi(x):  
    # CDF of the standard Gaussian distribution  
    return (1.0 + math.erf(x/math.sqrt(2.0)))/2.0  
  
phi(1)
```

0.8413447460685428

(ii) **random**: This module yields random number generation. See below for some examples.

```
import random  
  
random.randrange(start=1, stop=10, step=1)  
# a random number in range(start, stop, step)  
random.randrange(10) # integer from 0 to 9 inclusive
```

5

```
# returns random integer n such that a<=n<=b  
random.randint(1, 10)
```

7

(iii) **itertools**: This package provides a succinct way of searching for all the possible cases in many combinatorics-related scenarios. It is particularly useful for solving Boolean problems (that we learned in Section 1.5) in brute force.

```
from itertools import permutations, combinations  
  
# generating all permutations of [1, 2, 3]  
p = permutations([1, 2, 3])  
  
for i in p:  
    print (i)
```

(1, 2, 3)  
(1, 3, 2)  
(2, 1, 3)  
(2, 3, 1)  
(3, 1, 2)  
(3, 2, 1)

```
# generating all length-2 combinations of [1, 2, 3]
c = combinations([1, 2, 3], 2)

for i in c:
    print (i)
```

```
(1, 2)
(1, 3)
(2, 3)
```

```
# generating all length-3 combinations of [1, 2, 3, 4, 5]
c = combinations([1, 2, 3, 4, 5], 3)

for i in c:
    print (i)
```

```
(1, 2, 3)
(1, 2, 4)
(1, 2, 5)
(1, 3, 4)
(1, 3, 5)
(1, 4, 5)
(2, 3, 4)
(2, 3, 5)
(2, 4, 5)
(3, 4, 5)
```

**(iv) numpy:** Numpy is the most popular package for handling matrices and vectors. It offers many useful functions. We list a couple of the prominent functions frequently employed.

**(a) numpy.array():** numpy.array() is a specialized array data structure in numpy. This differs from Python data type array().

```
import numpy as np

np.array([1, 2, 3]) # construct an array
```

```
array([1, 2, 3])

np.array([[1, 2], [3, 4]]) # construct a 2D array

array([[1, 2],
       [3, 4]])
```

```
x = np.ones((2,2))  
# construct an all-one matrix with size of 2-by-2  
x = np.zeros((2,2))  
# construct an all-zero matrix with size of 2-by-2  
print(np.ones_like(x))  
# all-one matrix with the same shape and type of input  
print(np.zeros_like(x))  
# all-zero matrix with the same shape and type of input
```

```
[[1. 1.]  
 [1. 1.]]  
[[0. 0.]  
 [0. 0.]]
```

**(b) numpy.random()** This module is designed to perform random sampling from various probability distributions. Below we list a few popular examples. To learn more, you may want to consult with:

<https://numpy.org/doc/1.16/reference/routines.random.html>

```
# sampling a number from standard Gaussian distribution  
np.random.normal(loc = 0, scale = 1)  
# loc: mean, scale: standard deviation  
np.random.randn() # plays the same role
```

-2.5459976698222495

```
# sampling multiple numbers as per the standard Gaussian  
np.random.normal(0, 1, size = (2, 2))  
# Here the size determines the output shape  
np.random.randn(2,2) # plays the same role
```

array([[-1.8133258 , -1.01151295],  
 [-0.37375747, 0.36005748]])

```
np.random.rand(2,2) # Uniform over [0,1]
```

array([[0.06535694, 0.2507505 ],  
 [0.17559137, 0.60967901]])

**(c) numpy.linalg** This package offers many useful linear-algebra related functions. Here are a few of them.

```
from numpy import linalg
```

```
x = np.random.randn(2,2)  
print(linalg.det(x)) # Determinant of a matrix x
```

```
print(linalg.inv(x))    # Inverse of a matrix x
print(linalg.norm(x))   # Matrix or vector norm
print(linalg.svd(x))   # Singular value decomposition
print(linalg.eig(x))   # Eigenvalue decomposition
```

```
0.7125655927348966
[[ 0.77007826 -0.38835738]
 [ 2.33455331  0.64504946]]
1.832010151997132
(array([[-0.2060815,  0.97853483],
       [ 0.97853483,  0.2060815]]), array([1.78814528, 0.39849424]),
array([[-0.96330981,  0.2683919],
       [ 0.2683919,  0.96330981]]))
(array([0.50418566+0.67702467j, 0.50418566-0.67702467j]),
array([[0.02479485-0.37684352j, 0.02479485+0.37684352j],
       [0.92594502+0.j         , 0.92594502-0.j         ]]))
```

**(d) resizing:** The resizing is often used for transforming the dimension of one into another.

```
x = np.random.randn(4,4,1)
y = x.view(dtype=np.float_).reshape(-1,2)
# '-1' can be inferred from the context: Shape of (8,2)
print(y)
z = x.squeeze()
print(z.shape)
```

```
[-0.85719316  2.99692221]
 [ 1.16327996 -0.11955541]
 [-0.76229609  0.79871494]
 [ 0.99757568  0.69329723]
 [-1.52198295 -0.74430996]
 [ 0.17174063  0.25343301]
 [ 0.07151011 -2.90945412]
 [ 1.1874155  -0.64209109]]
(4, 4)
```

**(v) scipy.stats:** This module provides a large collection of probability distributions and related statistics. Below we present only a few of them. For more information, you can refer to:

<https://docs.scipy.org/doc/scipy/reference/stats.html>

```
from scipy import stats
```

```
# A random variable with the standard Gaussian
X = stats.norm(loc = 0, scale = 1)
# loc:mean, scale:standard deviation
print(X.cdf(np.array([-1, 0, 1])))
# computes the CDF at each numpy array
print(X.rvs(size = 3))
# generating a sequence of random variables
```

```
[0.15865525 0.5      0.84134475]
[ 0.39460402 -0.8042592 -0.71404882]
```

```
# Another random variable with the uniform distribution
Y = stats.uniform(loc = 0, scale = 1)
# uniform distribution in [loc, loc + scale]
print(Y.cdf(np.array([-1, 0, 0.5, 1])))
print(Y.rvs(size = 3))
```

```
[0.  0.  0.5 1.]
[0.72953474 0.67879248 0.47947748]
```

### A.2.3 Visualization

The most popular function for drawing a graph is `matplotlib.pyplot`. Here is how to use it.

```
import matplotlib.pyplot as plt

x_value = [x for x in range(10)]
y_value = [y for y in range(10, 20)]
plt.plot(x_value, y_value) # create a plot
plt.xlabel('x')           # labeling x-axis
plt.ylabel('y')           # labeling y-axis
plt.show
# No need to use show() in jupyter notebook.
```

```
<function matplotlib.pyplot.show(close=None, block=None)>
```

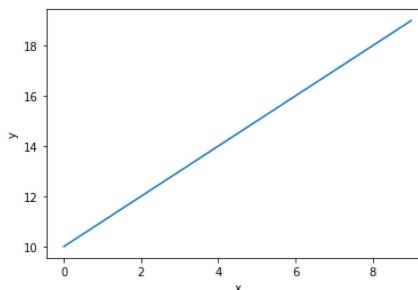


Figure A.7. Plotting a simple function via `matplotlib.pyplot`.

```
# Drawing multiple graphs in a single plot
```

```
x = [x for x in range(10)]
y_1 = [3*y for y in range(10)]
y_2 = [2*y for y in range(10)]

plt.plot(x, y_1) # plot_1
plt.plot(x, y_2) # plot_2
plt.xlabel('x') # labeling x-axis
plt.ylabel('y') # labeling y-axis
plt.legend(['y=3x', 'y=2x'])
```

```
<matplotlib.legend.Legend at 0x1d9e45c7a00>
```

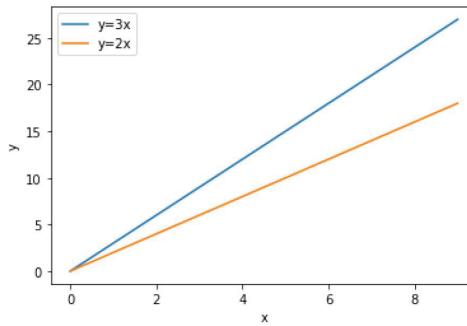


Figure A.8. Multiple functions and legend.

## Appendix B

# CVXPY Basics

**Outline** There are a couple of high-level software packages specialized for solving convex optimization problems. One very convenient and user-friendly package is CVX which has been developed by Michael Grant and Stephen P. Boyd. While it is very intuitive and therefore easy to learn, it is built upon a non-open-source platform MATLAB which requires a license for use. In contrast, there are two other packages that run in an open-source platform Python. These are CVXPY and `scipy.optimize`. Among these two, this book focuses on the use of CVXPY which we believe is more friendly and is evolving from the contributions of many researchers and engineers. In this section, we are going to cover four very basic stuffs for CVXPY. First, we will present how to install CVXPY library in Python. Writing a CVXPY script consists of three key procedures: (i) defining an optimization problem; (ii) calling a solver for the problem; and (iii) obtaining the solution. So in the second part, we will figure out how to define an optimization problem based on some known concepts (that we have learned thus far) like variables, constraints and objective. We will then study how to solve the problem accordingly. For ease of illustration, we will demonstrate the whole procedure via a simple example.

**Installation** The use of CVXPY requires the installation of Python 3. For Python 3, we can rely upon virtual environment tools like Anaconda which we described in Section A.1 for installation.

Installing CVXPY library is very simple. We can do it by using the library manager pip.

```
pip install cvxpy
```

```
Collecting cvxpy
  Downloading cvxpy-1.2.0-cp39-cp39-win_amd64.whl (832 kB)
Collecting ecos>=2Note:
  you may need to restart the kernel to use updated packages.
  Downloading ecos-2.0.10-cp39-cp39-win_amd64.whl (68 kB)
Collecting scs>=1.1.6
  Downloading scs-3.2.0-cp39-cp39-win_amd64.whl (8.1 MB)
Requirement already satisfied: numpy>=1.15 in
c:\programdata\anaconda3\lib\site-packages (from cvxpy) (1.20.3)
Collecting osqp>=0.4.1
  Downloading osqp-0.6.2.post5-cp39-cp39-win_amd64.whl (278 kB)
Requirement already satisfied: scipy>=1.1.0 in
c:\programdata\anaconda3\lib\site-packages (from cvxpy) (1.7.1)
Collecting qldl
  Downloading qldl-0.1.5.post2-cp39-cp39-win_amd64.whl (83 kB)
Installing collected packages: qldl, scs, osqp, ecos, cvxpy
Successfully installed cvxpy-1.2.0 ecos-2.0.10 osqp-0.6.2.post5
qldl-0.1.5.post2 scs-3.2.0
```

The command pip list allows us to check whether CVXPY is successfully installed. Please see below the created cvxpy in the middle.

```
pip list
```

Package	Version
alabaster	0.7.12
anaconda-client	1.9.0
anaconda-navigator	2.1.1
anaconda-project	0.10.1
-----	-----
-----	-----
cvxpy	1.2.0
-----	-----
-----	-----
jupyter	1.0.0
jupyter-client	6.1.12
jupyter-console	6.4.0
jupyter-core	4.8.1

jupyter-server	1.4.1
jupyterlab	3.2.1
jupyterlab-pygments	0.1.2
jupyterlab-server	2.8.2
jupyterlab-widgets	1.0.0
-----	
-----	
-----	
-----	
zict	2.0.0
zipp	3.6.0
zope.event	4.5.0
zope.interface	5.4.0

Alternatively, one can attempt to import CVXPY library.

```
import cvxpy
```

Without any error, you are ready to use. For any errors that you may encounter yet we do not mention here, you may want to consult with the installation guide on:

<https://www=cvxpy.org/install/index.html>

**How to define an optimization problem** In CVXPY, an optimization problem is comprised of four components: (i) variables; (ii) constraints; (iii) objective; and (iv) parameters. Let us dig into details on how to define them.

*1. Variables:* The variables refer to the optimization variables that we have played with throughout. We define a scalar variable via `cp.Variable()`. We construct a *vector* by putting the size inside the parenthesis. We can even create a *matrix* by indicating two numbers inside the parenthesis. But the variable cannot go beyond a 2D matrix, so it cannot be a 3D or 4D tensor. Here are some examples.

```
import cvxpy as cp
# Construct a scalar opt. var. with a blank parenthesis
x = cp.Variable()
# Can construct another in case we have more variables.
y = cp.Variable()
```

```
# Or we can construct a 3-by-1 vector.
z = cp.Variable(3)
# Or a matrix with a proper size, say (2,3)
w = cp.Variable((2,3))
```

*2. Constraints:* There are two types of constraints: (i) equality; and (ii) inequality constraints. We do not rely upon the standard form that we learned in Section 1.2, allowing for more flexible descriptions. We use `==`, `<=` and `<=` symbols to implement. Here is an example:

```
constraints = [x+y == 10, x-y <= 3]
```

*3. Objective:* The objective refers to the objective function that we wish to minimize or maximize. Depending on the optimization type between min and max, we use either of the following two:

```
obj_min = cp.Minimize((x-3*y)**2)
obj_max = cp.Maximize((x-3*y)**2)
```

*4. Parameters:* Parameters indicate the ones that are employed for perturbing an optimization problem. For instance, we introduce a parameter, say  $a$ , which plays a multiplication role in front of  $y$  in the above minimization, e.g.,  $cp.Minimize((x-a*y)**2)$ . The way to introduce the parameter is very simple:

```
a = cp.Parameter(nonneg=True)
a.value = 3 # assigning a value 3 for 'a'
```

Here `nonneg=True` indicates the non-negativity property of the parameter. If it is non-positive, we can type it like `nonpos=True`. A good thing about the use of parameters is that the change of an optimization problem does not require re-defining it from scratch.

Once the variables, constraints, objective and parameters are defined as above, the last step is to formulate a problem object via `cp.Problem()`.

```
prob = cp.Problem(obj_min, constraints)
```

**How to solve an optimization problem** We can solve a formulated problem by calling a solver. The implementation is very simple, requiring a single-line code:

```
prob.solve() # Solve 'prob' and print the optimal value
```

16.0

The command `prob.status` allows us to check whether or not the derived solution is optimal. The optimal value and the corresponding variables can also be retrieved with `prob.value`, `x.value` and `y.value`.

```
print('status: ', prob.status)
print('optimal value: ', prob.value)
print('optimal variables: ', x.value, y.value)
```

```
status:    optimal
optimal  value: 16.0
optimal  variables: 6.5 3.5
```

Here is a script for the entire code that also incorporates the use of the parameter  $a$  in the objective  $\text{cp.Minimize}((x-a*y)^{**2})$ .

```
x = cp.Variable()
y = cp.Variable()

constraints = [x+y == 10, x-y <= 3]
a = cp.Parameter(nonneg=True)
a.value = 3
obj_min = cp.Minimize((x-a*y)**2)
prob = cp.Problem(obj_min, constraints)
prob.solve()

print('status: ', prob.status)
print('optimal value: ', prob.value)
print('optimal variables: ', x.value, y.value)
```

```
status: optimal
optimal value: 16.0
optimal variables: 6.5 3.5
```

Notice that we have exactly the same solution as before. A different value of  $a.value=5$  leads to a different solution below. Please check.

```
status: optimal
optimal value: 121.0
optimal variables: 6.5 3.5
```

## Appendix C

# TensorFlow and Keras Basics

**Outline** One of the machine learning applications that we studied in Part III is deep learning. Deep learning is a learning methodology that takes a deep neural network (DNN) as a basic model for an interested prediction module. There are many prevalent software tools that ease deep learning implementation. Such tools are called the machine learning frameworks or application programming interfaces (APIs). Examples include TensorFlow, Keras, Pytorch, DL4J, Caffe and mxnet. The frameworks exhibit pros and cons, depending on what we pursue in the design of a deep learning model like good usability, fast training, functionality, high scalability in distributed training. This book aims at the great usability aspect, so we are going to study the most high-level API with a focus on enabling fast user experimentation: Keras. The Keras API allows us to go from idea to implementation with very few steps. In this appendix, we are going to present four very basic contents regarding Keras. In fact, Keras is fully integrated with TensorFlow, meaning that Keras comes completely packaged with the TensorFlow installation. So in the first part, we will learn how to install TensorFlow. Deep learning implementation requires three key procedures: (i) preparing and processing data; (ii) building a neural network model; and (iii) training a model and testing the trained model. So in the second part, we will touch upon some easy way to deal with data, offered in Keras. We will then study how to build a neural network model based on some popular packages including `keras.models` and `keras.layers`. Lastly, we will investigate how to train/test a model accordingly. For ease of illustration, we will demonstrate the entire procedures via a simple example.

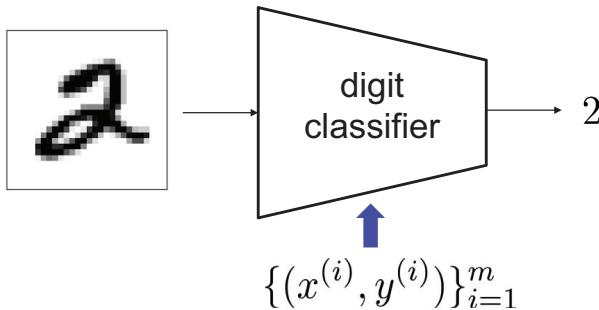


Figure C.1. Handwritten digit classification.

**Installation** The use of Keras requires the installation of the TensorFlow package. The installation procedure is very simple:

```
pip install tensorflow
```

TensorFlow 2 packages that fully support Keras and hence we will use require a pip version higher than 19.0 (or higher than 20.3 for macOS). So you may need to install the lastest pip via: `pip install --upgrade pip`. To figure out whether it is successfully installed, one can attempt to import keras as follows.

```
from tensorflow import keras
```

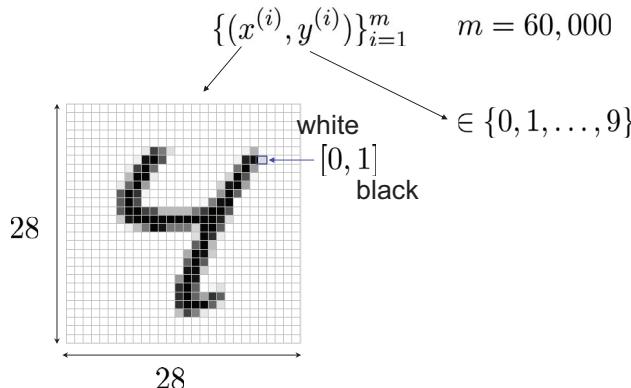
Without any error, you are ready to use. For any errors that you may encounter, you may want to consult with the installation guideline on:

<https://www.tensorflow.org/install>

**A simple task of focus** A simple task that we will focus on for illustration of how to use Keras is handwritten digit classification wherein the goal is to figure out a digit from a handwritten image. See Fig. C.1 for a sample image. The figure illustrates an instance in which an image of digit 2 is correctly recognized.

**Preparing and processing data** There is a popular dataset associated with the digit classification task. That is, MNIST (Modified National Institute of Standards and Technology) dataset. It contains  $m = 60,000$  training images and  $m_{\text{test}} = 10,000$  testing images. Each image, say  $x^{(i)}$ , consists of  $28 \times 28$  pixels, each indicating a gray-scale level raning from 0 (white) to 1 (black). It also comes with a corresponding label, say  $y^{(i)}$ , that takes one of the 10 classes  $y^{(i)} \in \{0, 1, \dots, 9\}$ . See Fig. C.2.

One great benefit about Keras is that such popular datasets including MNIST are stored in a sub-library: `keras.datasets`. Even more, train and test datasets are



**Figure C.2.** MNIST dataset: An input image is of 28-by-28 pixels, each indicating an intensity from 0 (white) to 1 (black); each label with size 1 takes one of the 10 classes from 0 to 9.

already therein with a proper split ratio. So we don't need to worry about how to split them. The only script that we should write for importing MNIST is:

```
from tensorflow.keras.datasets import mnist
(X_train, y_train), (X_test, y_test) = mnist.load_data()
X_train = X_train/255.
X_test = X_test/255.
```

Downloading data from <https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz>

11493376/11490434 [=====] - 1s Ous/step  
11501568/11490434 [=====] - 1s Ous/step

Here we divide the input ( $X_{\text{train}}$  or  $X_{\text{test}}$ ) by its maximum value 255 for the purpose of normalization. This procedure is often done as a part of data preprocessing. In case `keras.datasets` does not offer a dataset of our interest, we have to know some data preprocessing techniques that require the use of another prominent library, named `pandas`. `pandas` is particularly instrumental in handling `.csv` files. Here we will not explain how to use `pandas` in detail. If you want to learn more, you may want to consult with:

<https://pandas.pydata.org/>

For data visualization, we employ `matplotlib.pyplot`. Below is a simple code for plotting one sample image.

```
import matplotlib.pyplot as plt
plt.imshow(X_train[0], cmap = 'gray_r')
```

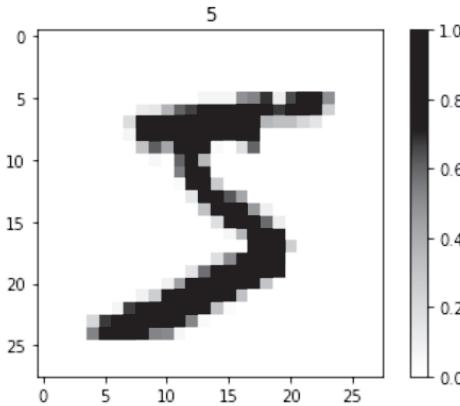


Figure C.3. A sample image in MNIST dataset.

```
plt.colorbar()  
plt.title('{}'.format(y_train[0], fontsize=30))  
  
Text(0.5, 1.0, '5')
```

See Fig. C.3 for the output. Here the option `cmap = 'gray_r'` is for enabling the white background and a black letter. We use `cmap = 'gray'` for the flipped one, i.e., a white letter with the black background. The `colorbar()` function displays the color bar on the right as in Fig. C.3.

We can also plot many images in one figure. Here is an example of displaying 60 images.

```
num_of_images = 60  
for index in range(1,num_of_images+1):  
    plt.subplot(6,10, index)  
    plt.axis('off')  
    plt.imshow(X_train[index], cmap = 'gray_r')
```

See Fig. C.4 for the output.

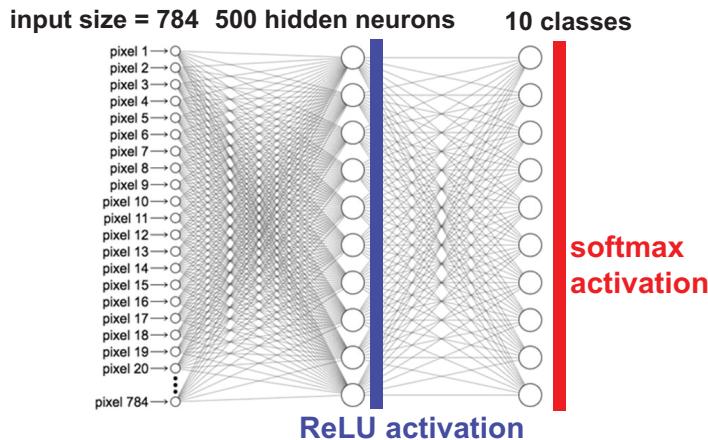
**Building a neural network model** Let us employ a two-layer neural network that we studied in Sections 3.3 and 3.4. Specifically we introduce a hidden layer with 500 neurons. See Fig. C.5 for illustration of the network architecture. As we learned in Section 3.4, we employ the ReLU activation in the hidden layer, and softmax activation in the output layer.

Keras includes two major packages:

- (i) `tensorflow.keras.models`;
- (ii) `tensorflow.keras.layers`.



**Figure C.4.** Plotting many image samples in a single figure.



**Figure C.5.** A two-layer fully-connected neural network where input size is  $28 \times 28 = 784$ , the number of hidden neurons is 500 and the number of classes is 10. We employ ReLU activation for the hidden layer, and softmax activation for the output layer.

The `models` package contains several functionalities regarding a neural network itself. One major module is `Sequential` which is a neural network entity and hence can be described as a linear stack of layers. The `layers` package includes many elements that constitute a neural network. Examples include fully-connected dense layers and activation functions. These two allow us to readily construct a model illustrated in Fig. C.5.

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten

model = Sequential()
model.add(Flatten(input_shape=(28,28)))
model.add(Dense(500, activation='relu'))
model.add(Dense(10, activation='softmax'))
model.summary()
```

Model: "sequential\_1"

Layer (type)	Output Shape	Param #
<hr/>		
flatten (Flatten)	(None, 784)	0
dense (Dense)	(None, 500)	392500
dense_1 (Dense)	(None, 10)	5010
<hr/>		
Total params: 397,510		
Trainable params: 397,510		
Non-trainable params: 0		

Here Flatten is an entity that indicates a vector expanded from a higher dimensional one, like a 2D matrix. In this example, a digit image of size 28-by-28 is flattened into a vector of size 784( $= 28 \times 28$ ). add is a method for attaching an interested layer to the last part in the sequential model. Dense refers to a fully-connected layer. The input size is automatically determined by the last part that it will be attached to in the sequential model. So the only thing to specify is the number of output neurons. In this example, 500 refers to the number of hidden neurons. We can also set an activation function with another argument, like activation='relu'. The output layer comes with 10 neurons (coinciding with the number of classes) and softmax activation (representing the probability of an output belonging to a certain class). The summary() method presents a list of the entire layers specifying the size and the number of associated parameters.

**Training a model** For training, we need to first set up an algorithm (optimizer) to be employed. One popular algorithm that we have often played with is gradient descent. But here we will utilize its advanced version that we also learned in Section 3.5. That is, the Adam optimizer. The Adam optimizer can be viewed as a smart tweak of gradient descent that enables more stable training. As mentioned earlier, Adam has three key hyperparameters: (i) the learning rate  $\alpha$ ; (ii)  $\beta_1$  (capturing the weight of past gradients); and (iii)  $\beta_2$  (indicating the weight of the square of past gradients). The default choice reads:  $(\alpha, \beta_1, \beta_2) = (0.001, 0.9, 0.999)$ . So these values would be set if nothing is specified.

We also need to specify a loss function. As we learned in Section 3.2 (also via Prob 8.2 for the more-than-two class case), the optimal choice in a sense of maximizing likelihood is cross entropy. A performance metric that we will look at during training and testing can also be specified. One metric frequently employed is accuracy. One can set all of these via another method compile.

```
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['acc'])
```

Here the option `optimizer='adam'` sets the default choice of the learning rate and betas. For a manual choice, we first define:

```
opt= tensorflow.keras.optimizers.Adam(
    learning_rate=0.01,
    beta_1 = 0.92,
    beta_2 = 0.992)
```

We then replace the above option with `optimizer=opt`. The option `loss='sparse_categorical_crossentropy'` means the use of the cross entropy loss.

Now we can bring this to train the model on MNIST data. During training, we often employ a part of the entire examples to compute a gradient of a loss function. The part is called *batch*. Two more terminologies. One is the *step* which refers to a loss computation procedure spanning the examples only in a single batch. The other is the epoch which refers to the entire procedure associated with all the examples. In our experiment, let us use the batch size of 64 and the number 20 of epochs.

```
history = model.fit(X_train, y_train, batch_size=64, epochs=20)
```

```
Epoch 1/20
938/938 [=====] - 2s 2ms/step - loss: 0.0025 - acc: 0.9992
Epoch 2/20
938/938 [=====] - 2s 2ms/step - loss: 0.0059 - acc: 0.9981
Epoch 3/20
938/938 [=====] - 2s 2ms/step - loss: 0.0031 - acc: 0.9990
Epoch 4/20
938/938 [=====] - 2s 2ms/step - loss: 0.0074 - acc: 0.9976
Epoch 5/20
938/938 [=====] - 2s 2ms/step - loss: 0.0025 - acc: 0.9993
Epoch 6/20
938/938 [=====] - 2s 2ms/step - loss: 0.0043 - acc: 0.9984
Epoch 7/20
938/938 [=====] - 2s 2ms/step - loss: 0.0044 - acc: 0.9984
Epoch 8/20
938/938 [=====] - 2s 2ms/step - loss: 0.0010 - acc: 0.9998
Epoch 9/20
938/938 [=====] - 2s 2ms/step - loss: 1.2813e-04 - acc: 1.0000
Epoch 10/20
938/938 [=====] - 2s 2ms/step - loss: 3.5169e-05 - acc: 1.0000
Epoch 11/20
938/938 [=====] - 2s 2ms/step - loss: 2.1899e-05 - acc: 1.0000
```

```
Epoch 12/20
938/938 [=====] - 2s 2ms/step - loss: 1.6756e-05 - acc: 1.0000
Epoch 13/20
938/938 [=====] - 2s 2ms/step - loss: 1.2778e-05 - acc: 1.0000
Epoch 14/20
938/938 [=====] - 2s 2ms/step - loss: 9.8947e-06 - acc: 1.0000
Epoch 15/20
938/938 [=====] - 2s 2ms/step - loss: 0.0082 - acc: 0.9981
Epoch 16/20
938/938 [=====] - 2s 2ms/step - loss: 0.0090 - acc: 0.9971
Epoch 17/20
938/938 [=====] - 2s 2ms/step - loss: 0.0016 - acc: 0.9995
Epoch 18/20
938/938 [=====] - 2s 2ms/step - loss: 3.9583e-04 - acc: 0.9999
Epoch 19/20
938/938 [=====] - 2s 2ms/step - loss: 7.6672e-05 - acc: 1.0000
Epoch 20/20
938/938 [=====] - 2s 2ms/step - loss: 2.4958e-05 - acc: 1.0000
```

One good thing about the `fit()` function is that it returns a dictionary of the metrics collected during training. We can check the collected metrics via:

```
# list all data in history object
print(history.history.keys())
```

```
dict_keys(['loss', 'acc'])
```

Using this data, we can also plot an accuracy curve as a function of epochs.

```
plt.plot(history.history['acc'])
plt.title('model accuracy')
plt.xlabel('epoch')
plt.ylabel('accuracy')
```

```
Text(0, 0.5, 'accuracy')
```

**Testing the trained model** For testing, we first need to make a prediction from the model output. To this end, we use the `predict()` function as follows:

```
model.predict(X_test).argmax(1)
```

```
array([7, 2, 1, ..., 4, 5, 6], dtype=int64)
```

Here `argmax(1)` returns the class w.r.t. the highest softmax output among the 10 classes. In order to evaluate the test accuracy, we use the `evaluate()` function:

```
model.evaluate(X_test, y_test)
```

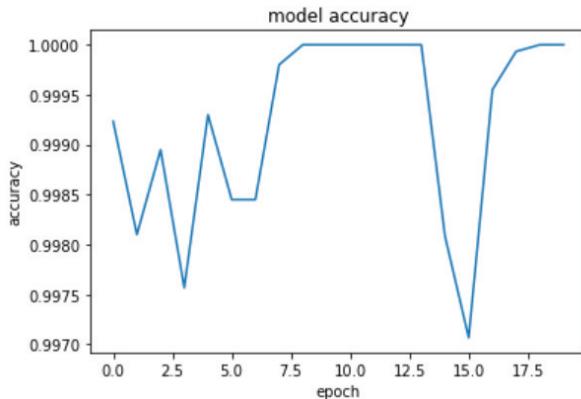


Figure C.6. Accuracy as a function of epochs.

```
313/313 [=====] - Os 751us/step - loss: 0.1001 - acc: 0.9847
```

```
[0.10007859766483307, 0.9847000241279602]
```

**Saving and loading** Saving the trained model and loading the saved model later is very simple. See below.

```
model.save('saved_classifier')
```

```
INFO:tensorflow:Assets written to: saved_classifier\assets
```

```
import tensorflow
loaded_model = tensorflow.keras.models.load_model(
    'saved_classifier')
```

## References

- Alizadeh, F. (1991). “Combinatorial optimization with interior point methods and semi-definite matrices”. *Ph. D. thesis, University of Minnesota*.
- Angwin, J., J. Larson, S. Mattu, and L. Kirchner. (2020). “There’s software used across the country to predict future criminals and it’s biased against blacks. 2016”.
- Arjovsky, M., S. Chintala, and L. Bottou. (2017). “Wasserstein generative adversarial networks”. In: *International conference on machine learning*. PMLR. 214–223.
- Arora, S., R. Ge, Y. Liang, T. Ma, and Y. Zhang. (2017). “Generalization and equilibrium in generative adversarial nets (gans)”. In: *International Conference on Machine Learning*. PMLR. 224–232.
- Beckmann, E. C. (2006). “CT scanning the early days”. *The British journal of radiology*. 79(937): 5–8.
- Ben-Tal, A. and A. Nemirovski. (1998). “Robust convex optimization”. *Mathematics of operations research*. 23(4): 769–805.
- Bhattacharyya, K. B. (2016). “Godfrey Newbold Hounsfield (1919–2004): The man who revolutionized neuroimaging”. *Annals of Indian Academy of Neurology*. 19(4): 448.
- Candès, E. J. and B. Recht. (2009). “Exact matrix completion via convex optimization”. *Foundations of Computational mathematics*. 9(6): 717–772.
- Cho, J., G. Hwang, and C. Suh. (2020). “A fair classifier using mutual information”. In: *2020 IEEE International Symposium on Information Theory (ISIT)*. IEEE. 2521–2526.
- Cho, J. and C. Suh. (2019). “Wasserstein GAN can perform PCA”. In: *2019 57th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*. IEEE. 895–901.

- Cormack, A. M. and G. N. Hounsfield. ‘The Nobel Prize in Physiology or Medicine 1979 Explore Perspectives’.
- Cover, T. M. (1999). *Elements of information theory*. John Wiley & Sons.
- Dantzig, G. B. (1951). “Maximization of a linear function of variables subject to linear inequalities”. *Activity analysis of production and allocation*. 13: 339–347.
- Dijkstra, E. W. et al. (1959). “A note on two problems in connexion with graphs”. *Numerische mathematik*. 1(1): 269–271.
- Dikin, I. (1967). “Iterative solution of problems of linear and quadratic programming”. In: *Doklady Akademii Nauk*. Vol. 174. No. 4. Russian Academy of Sciences. 747–748.
- Drezner, Z. and H. W. Hamacher. (2004). *Facility location: applications and theory*. Springer Science & Business Media.
- El Ghaoui, L. and H. Lebret. (1997). “Robust solutions to least-squares problems with uncertain data”. *SIAM Journal on matrix analysis and applications*. 18(4): 1035–1064.
- Frank, M. and P. Wolfe. (1956). “An algorithm for quadratic programming”. *Naval research logistics quarterly*. 3(1–2): 95–110.
- Fukushima, K. (1980). “A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position”. *Biol. Cybern.*. 36: 193–202.
- Gardner, R. (1990). “LV Kantorovich: the price implications of optimal planning”. *Journal of Economic Literature*. 28(2): 638–648.
- Garnier, J.-G. and A. Quetelet. (1838). *Correspondance mathématique et physique*. Vol. 10. Impr. d'H. Vandekerckhove.
- Gauss, C. F. (1887). *Abhandlungen zur Methode der kleinsten Quadrate*. P. Stankiewicz.
- Geletu, A., M. Klöppel, H. Zhang, and P. Li. (2013). “Advances and applications of chance-constrained approaches to systems optimisation under uncertainty”. *International Journal of Systems Science*. 44(7): 1209–1232.
- Glorot, X., A. Bordes, and Y. Bengio. (2011). “Deep sparse rectifier neural networks”. In: *Proceedings of the fourteenth international conference on artificial intelligence and statistics*. JMLR Workshop and Conference Proceedings. 315–323.
- Goodfellow, I., J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. (2014). “Generative adversarial nets”. *Advances in neural information processing systems*. 27.
- Gutmann, M. and A. Hyvärinen. (2010). “Noise-contrastive estimation: A new estimation principle for unnormalized statistical models”. In: *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*. Organization. PMLR. 297–304.

- Hardt, M., E. Price, and N. Srebro. (2016). “Equality of opportunity in supervised learning”. *Advances in neural information processing systems*. 29.
- Hinton, G., N. Srivastava, and K. Swersky. (2012). “Neural networks for machine learning lecture 6a overview of mini-batch gradient descent”. *Cited on*. 14(8): 2.
- Ioffe, S. and C. Szegedy. (2015). “Batch normalization: Accelerating deep network training by reducing internal covariate shift”. In: *International conference on machine learning*. PMLR. 448–456.
- Ivakhnenko, A. G. (1971). “Polynomial theory of complex systems”. *IEEE transactions on Systems, Man, and Cybernetics*. (4): 364–378.
- Kantorovich, L. V. (1960). “Mathematical methods of organizing and planning production”. *Management science*. 6(4): 366–422.
- Kantorovich, L. V. (1989). “Mathematics in economics: Achievements, difficulties, perspectives”. *The American Economic Review*. 79(6): 18–22.
- Karp, R. M. (1972). “Reducibility among combinatorial problems”. In: *Complexity of computer computations*. Springer. 85–103.
- Karush, W. (1939). “Minima of functions of several variables with inequalities as side constraints”. *M. Sc. Dissertation. Dept. of Mathematics, Univ. of Chicago*.
- Kingma, D. P. and J. Ba. (2014). “Adam: A method for stochastic optimization”. *arXiv preprint arXiv:1412.6980*.
- Krizhevsky, A., I. Sutskever, and G. E. Hinton. (2012). “Imagenet classification with deep convolutional neural networks”. *Advances in neural information processing systems*. 25.
- Kuhn, H. W. and A. W. Tucker. (2014). “Nonlinear programming”. In: *Traces and emergence of nonlinear programming*. Springer. 247–258.
- Larson, J., S. Mattu, L. Kirchner, and J. Angwin. (2016). ‘How we analyzed the COMPAS recidivism algorithm’.
- LeCun, Y., B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. (1989). “Backpropagation applied to handwritten zip code recognition”. *Neural computation*. 1(4): 541–551.
- Lin, J. (1991). “Divergence measures based on the Shannon entropy”. *IEEE Transactions on Information theory*. 37(1): 145–151.
- Linnainmaa, S. (1970). “Alogritmin kumulatiivinen pyöristysvirhe yksittäisten pyöristysvirheiden Taylor-kehitelmänä”. *Ph.D. thesis*, Master’s thesis, University of Helsinki.
- Minsky, M. and S. Papert. (1969). “Perceptrons.”
- Monge, G. (1781). “Mémoire sur la théorie des déblais et des remblais”. *Histoire de l’Académie Royale des Sciences de Paris*.
- Nesterov, Y. (1998). “Semidefinite relaxation and nonconvex quadratic optimization”. *Optimization methods and software*. 9(1–3), 141–160.

- Nesterov, Y. and A. Nemirovskii. (1994). *Interior-point polynomial algorithms in convex programming*. SIAM.
- News, B. (2016). “Artificial Intelligence: Google’s AlphaGo Beats Go Master Lee Se-Dol”. *Science*. 352(6285): 1289–1290.
- Pearson, K. (1901). “LIII. On lines and planes of closest fit to systems of points in space”. *The London, Edinburgh, and Dublin philosophical magazine and journal of science*. 2(11): 559–572.
- Rosenblatt, F. (1958). “The perceptron: a probabilistic model for information storage and organization in the brain.” *Psychological review*. 65(6): 386.
- Rosenbusch, G. and A. de Knecht-van Eekelen. (2019). “A New Kind of Rays”. In: *Wilhelm Conrad Röntgen*. Springer. 79–113.
- Rumelhart, D. E., G. E. Hinton, and R. J. Williams. (1986). “Learning representations by back-propagating errors”. *Nature*. 323(6088): 533–536.
- Russakovsky, O., J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, et al. (2015). “Imagenet large scale visual recognition challenge”. *International journal of computer vision*. 115(3): 211–252.
- Samuel, A. L. (1967). “Some studies in machine learning using the game of checkers. II—Recent progress”. *IBM Journal of research and development*. 11(6): 601–617.
- Serio, G. F., A. Manara, P. Sicoli, and W. F. Bottke. (2002). *Giuseppe Piazzi and the discovery of Ceres*. University of Arizona Press.
- Shannon, C. E. (2001). “A mathematical theory of communication”. *ACM SIGMOBILE mobile computing and communications review*. 5(1): 3–55.
- Silver, D., A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, et al. (2016). “Mastering the game of Go with deep neural networks and tree search”. *Nature*. 529(7587): 484–489.
- Slater, M. (2014). “Lagrange multipliers revisited”. In: *Traces and emergence of nonlinear programming*. Springer. 293–306.
- Vaserstein, L. N. (1969). “Markov processes over denumerable products of spaces, describing large systems of automata”. *Problemy Peredachi Informatsii*. 5(3): 64–72.
- Villani, C. (2009). “Optimal Transport. Grundlehren der Mathematischen Wissenschaften”.
- Wright, M. (2005). “The interior-point revolution in optimization: history, recent developments, and lasting consequences”. *Bulletin of the American mathematical society*. 42(1): 39–56.

- Zafar, M. B., I. Valera, M. G. Rogriguez, and K. P. Gummadi. (2017). “Fairness constraints: Mechanisms for fair classification”. In: *Artificial Intelligence and Statistics*. PMLR. 962–970.
- Zhang, F. (2006). *The Schur complement and its applications*, Vol. 4. Springer Science & Business Media.

# Index

- 1-Lipschitz constraint, 281, 285, 289
- 1<sup>st</sup>-order Wasserstein distance, 33, 54, 273, 278, 295
- 2<sup>nd</sup>-order Wasserstein distance, 296
- a priori probability, 68
- activation, 38, 190, 191, 193, 202–206, 208–213, 221–224, 226, 229, 231, 232, 234, 237, 239, 240, 255–257, 260, 263, 264, 285–289, 299, 319, 325, 351–353
- adam, 221, 225, 226, 254, 286, 353
- adam optimizer, 224–226, 236–238, 259, 286, 315, 319, 353
- affine functions, 16, 18, 126, 165
- affine set, 13, 14
- algorithms, 6, 9, 18, 21–23, 28, 41, 42, 115, 124, 127, 130–132, 135, 139, 155, 162, 180, 186, 187, 191, 193, 207, 212, 241, 300, 301
- AlexNet, 200, 201
- AlphaGo, 187
- alternating gradient descent, 134, 135, 137, 138, 166, 253, 269, 270, 293, 311, 314
- Anaconda, 50, 330, 343
- application programming interfaces (APIs), 348
- artificial intelligence (AI), 187
- autoencoder, 241, 270
- backprop, 221, 235, 240
- backpropagation, 212, 216–218, 220, 221, 234, 240
- barrier function, 135–138, 141, 165, 166
- batch, 227, 254, 256, 259, 260, 264, 291, 315, 319, 354
- batch normalization (BN), 256, 287
- betas, 227, 354
- bi-directed graph, 39
- bias-allowing LS classifier, 62
- bias correction, 236
- binary cross entropy (BCE), 259, 304, 314
- bipartite matching problem, 55
- Boolean optimization problem, 55
- Boolean problem, 39, 41, 42, 168, 169, 175, 182, 183, 337
- bounded set, 157
- cat-dog classification problem, 37
- chain rule, 20, 194, 217, 218, 305
- chance program (CP), 97, 122
- classification problem, 23, 35–37, 42, 53, 54, 58, 60, 72

- classifier loss, 320  
closed set, 157  
clustering, 241  
colorbar, 351  
COMPAS, 316  
COMPAS dataset, 316  
complementary slackness condition, 128  
computed tomography (CT), 74, 82, 85  
concave functions, 16, 27  
concavity, 140  
constrained least squares, 86, 129, 131, 132  
constrained optimization, 37  
constraints, 3, 8, 11, 17, 30, 32, 33, 36, 37, 43–47, 49, 51, 56, 91, 100, 105, 112, 113, 125, 126, 128, 131, 135, 143, 150, 155, 167, 170, 173, 176, 179, 273, 278, 300, 309, 343, 345, 346  
convex classifier, 192  
convex combination, 12–14, 16, 103, 104, 146, 152  
convex functions, 10–12, 15, 16, 18, 19, 25, 126, 135  
convexity, 16, 25, 81, 86, 93, 94, 103, 104, 107, 146, 150–152, 164, 165, 189, 194, 195, 198, 309  
convex optimization, 8–12, 17–19, 22, 28, 42, 49, 50, 53, 58, 64, 72, 79, 86, 92, 93, 102, 113–115, 124, 125, 127, 129, 132, 133, 135, 136, 139–142, 154–158, 160, 162–165, 172, 180, 189, 193, 208, 274, 300, 307, 309, 313, 321, 322, 336, 343  
convex sets, 10, 12, 13, 16–18, 24, 25, 145, 146, 157, 161  
conv layer, 287, 288  
convolutional neural networks (CNNs), 211, 287  
covariance matrix, 98, 122, 283, 284, 297  
critic, 288–293  
critic loss, 292  
cross entropy, 192, 193, 198, 220, 230, 238, 259, 353  
cross entropy loss, 192, 193, 197, 198, 201, 208, 217, 221, 224, 227, 231, 232, 234, 238, 240, 283, 307, 314, 354  
cumulative density function (CDF), 99, 123  
cumulative distribution function (CDF), 99  
CVXPY, 2, 3, 23, 49–55, 57, 58, 64, 70, 71, 81, 85, 86, 90, 93, 100, 109, 113, 122, 343–345  
data, 35–38, 55, 60, 61, 64, 65, 67, 69, 70, 84, 85, 97, 98, 100, 103, 180, 196, 207, 225, 226, 240–244, 246, 247, 257, 271, 272, 274, 278, 283, 284, 293, 297–299, 301–304, 316–318, 325, 334, 335, 338  
deep learning, 2, 200–202, 206, 207, 210–212, 221, 222, 227, 248, 252, 294, 321, 348  
deep neural networks, 38, 200–202, 240, 249, 252  
density estimation problem, 244  
differentiable, 18, 24, 25, 28, 81, 136, 140, 191–193, 213, 218  
discriminator, 247, 248, 252–254, 257–261, 263, 265–267, 271, 288, 311, 315, 319, 320, 324–326, 328  
discriminator loss, 260, 315, 320  
disparate impact (DI), 301, 313  
disparate treatment (DT), 301, 313  
divergence measure, 231, 245, 246, 251, 273, 299, 307

- domain, 14, 20, 72, 212  
 downsample, 287, 288  
 dual function, 126, 128–130, 139–141, 144, 148, 157–160, 163–165, 167, 170–172, 175, 177, 184, 275, 276  
 dual problem, 124–129, 131, 140–144, 150, 157–160, 163–167, 169, 171, 172, 175, 178–180, 182–184, 274–276, 278–280, 296  
 duality, 9, 22, 36, 42, 88, 90, 100, 113, 124, 127–131, 133, 139, 143, 144, 150, 155, 160, 162–165, 179–181, 184, 185, 228, 242, 274, 285, 321  
 duality gap, 167, 169  
 eigenvalue, 86, 95, 109, 112, 113, 119, 120, 123, 195, 297  
 eigenvalue decomposition (EVD), 95  
 eigenvector, 112  
 empirical distributions, 245, 246, 271, 278, 295, 296, 298  
 entropy, 192, 193, 197, 198, 201, 208, 217, 221, 224, 227, 231, 232, 234, 238, 240, 283, 307, 308, 314, 354  
 epoch, 235, 238, 267, 355, 356  
 equality constraints, 11, 17, 44, 45, 51, 126, 131, 170, 278, 309  
 equality-constrained LS, 87, 93  
 equalized odds (EO), 323  
 Euclidean norm, 5, 6, 24, 32, 93, 94, 96, 97  
 examples, 2, 11, 13, 15, 16, 23, 28, 35, 41, 50, 52, 65, 86, 97, 219, 222, 226, 227, 230, 234, 241, 242, 248, 249, 254, 256, 260, 299, 303, 317, 318, 335, 336, 348, 352, 354  
 fair classifiers, 9, 181  
 fair generative modeling, 301  
 fair machine learning, 294, 299–301, 316, 321  
 fake data, 243–247, 299, 301  
 fake samples, 252, 262, 271, 278, 283, 295, 296, 298  
 false alarm, 66  
 false negative rate (FNR), 66  
 false positive rate (FPR), 66  
 fat matrix, 59  
 feasible set, 103, 126  
 feature learning, 241, 270  
 features, 287  
 function class, 188  
 function optimization, 188, 245, 246, 282, 299, 309, 310  
 GAN optimization, 246, 249, 251, 259, 261, 271, 273, 300, 306  
 Gauss's trick, 96  
 Gaussian approximation, 98  
 Gaussian distribution, 98, 99, 283, 297, 298  
 Gaussian elimination, 4, 6, 47, 48  
 generalized Schur complement lemma, 119, 173, 175  
 generative adversarial networks (GANs), 9, 33, 181, 242, 243, 246, 299  
 generative model, 242–244, 246, 299  
 generative modeling, 241–244, 246, 252, 298, 299, 301, 321  
 generator, 248, 253–255, 258, 259, 262, 264–266, 271, 273, 283, 287–289, 291–293, 297, 298, 311  
 generator loss, 254, 260, 267, 292  
 genetic natural selection in evolution, 205  
 global minimum, 214  
 gradient, 21, 25, 58, 86, 128, 133, 134, 191, 199, 213, 217, 218, 227, 286, 354  
 gradient ascent, 134, 253  
 gradient descent, 18, 21, 22, 26, 28, 37, 131, 134, 138, 162, 165, 180, 191, 193, 198, 215, 219, 221, 224, 235,

- 236, 240, 253, 269, 270, 299, 311, 319, 353
- half-plane, 14
- handwritten digit classification, 221, 349
- Hessian, 194, 195, 213
- hidden layer, 38, 202–205, 208, 210, 212, 216, 222, 223, 231, 232, 234, 237, 255–257, 263, 264, 286–288, 299, 351, 352
- hyperparameter, 68, 69, 82, 225, 236, 286, 287
- hyperplane, 13, 24, 145–147, 149, 150, 152, 154, 155, 157, 158, 160
- ice-cream cone, 94
- ImageNet, 200, 207
- ImageNet competition, 200
- indicator function, 65, 191
- inequality constraints, 11, 17, 30, 36, 44, 105, 128, 131, 150, 170, 345
- inner optimization problem, 250, 274, 275
- input layer, 202, 203, 205
- interior point method, 42, 131, 135, 137, 140, 155, 162, 165, 166, 169, 180, 185, 191, 321
- invertible, 88, 89, 105, 108, 116, 140
- iterative algorithm, 26, 28, 45
- itertools, 336, 337
- Jensen-Shannon (JS) divergence, 251, 252, 262, 298, 299
- Jensen's inequality, 25, 198, 251
- Jupyter notebook, 2, 329–332
- Kantorovich duality, 274, 278, 285
- Kantorovich's plywood cutting problem, 29, 35, 42, 50, 52
- Kantorovich-Rubinstein distance, 33
- Kantorovich-Rubinstein duality, 274
- Keras, 221, 226, 237, 257–260, 264–266, 289–292, 319, 320, 326, 348, 349
- kernel, 344
- KKT conditions, 88, 93, 113, 124, 129–133, 135–143, 155, 162, 181
- Kullback-Leibler (KL) divergence, 307
- label, 55, 61, 65, 180, 192, 193, 221, 222, 224, 229, 323, 324, 350
- labeled data, 241, 244
- Lagrange dual function, 126
- Lagrange function, 88, 125, 128, 129, 132, 133, 136–142, 159, 163, 166, 170–172, 176, 179, 180, 184, 275, 309
- Lagrange multipliers, 125, 163, 276, 278, 309
- Lagrange relaxation, 167–170, 174–176, 178, 180, 183, 321
- Laplace transform, 117
- leaky ReLU, 286–288, 299
- learning rate, 21, 26, 134, 135, 199, 215, 224, 226, 227, 235, 253, 269, 270, 286, 319, 353, 354
- least squares (LS), 53, 58, 72
- least-squares classifier, 60–63, 65, 67, 80, 81, 193
- legitimate-vs-spam emails classification, 60
- lifting, 111, 123, 179, 183
- likelihood, 195, 196, 198, 199, 201, 221, 224, 230, 240, 283, 284, 297–299, 302, 353
- linear activation, 287–289
- linear algebra, 1, 2
- linear classification, 42, 54
- linear classifiers, 60, 98
- linear matrix inequality (LMI), 103, 118
- linear program (LP), 7, 10, 17, 18, 22, 28
- linear projector, 61
- linearly independent, 59, 87, 90
- linearly separable, 37
- list, 16, 50, 206, 234, 242, 334, 338, 339, 353

- local minimum, 214
- log-likelihood, 297
- log-likelihood function, 297
- logarithmic barrier, 136, 137, 141, 165, 166, 169
- logistic function, 26, 192, 193, 201, 208, 210–212, 217, 221, 223, 229, 231, 240
- logistic regression, 191–193, 195, 196, 198, 199, 201, 229, 240, 294
- Lorentz cone, 94
- LP relaxation, 23, 34, 35, 39, 41, 42, 50, 55, 57, 86, 167, 168, 182, 183
- machine learning, 2, 9, 23, 33, 35–38, 53, 58, 60, 68, 72, 180, 181, 185–187, 189, 193, 228, 231, 244, 294, 299–301, 307, 321, 322, 348
- margin-based linear classifier, 64, 67, 80, 98
- MATLAB, 50, 218, 343
- matplotlib, 55, 85, 237, 239, 265, 325, 341, 350
- matplotlib.pyplot, 55, 341, 350
- matrix completion, 109, 120, 121
- MAXCUT problem, 110
- maximum likelihood, 195, 198, 299
- medical imaging, 72, 73
- minimax theorem, 267, 268
- minimizer, 12, 30, 130, 139, 144, 164, 172
- misdetection, 66, 85
- misdetection rate, 66, 85
- MNIST, 222, 225, 227, 237–239, 252, 254, 255, 257, 264, 285, 287, 349–351, 354
- MNIST data, 222, 225, 227, 237, 238, 257, 264, 350, 351, 354
- Modified National Institute of Standards and Technology (MNIST) data, 222, 349
- momentum, 225, 286
- momentum optimizer, 236
- Monge’s problem, 31, 54, 57
- Monge’s transportation problem, 31, 35, 42
- multiclass classifier, 229
- mutual information, 305, 307–310, 313, 323
- natural genetic selection, 205
- network flow problem, 23
- neural networks, 38, 189, 193, 200, 201, 240, 248, 252, 253, 282, 287, 310, 319, 325
- neurons, 190, 191, 202–205, 210, 212, 222, 226, 234, 240, 255–257, 263, 287, 352, 353
- non-convex optimization, 108, 129, 130, 162, 164, 169, 180, 185, 210, 214
- non-convex optimization problem, 108, 129, 162, 180, 185
- non-linear classifier, 38
- normal Gaussian distribution, 99
- numpy, 70, 81, 85, 91, 101, 114, 235, 239, 264, 266, 293, 319, 321, 325, 328, 336, 338, 339
- numpy.array, 71, 338
- numpy.linalg, 339
- numpy.random, 339
- objective function, 6, 8, 11, 17–19, 26, 29, 30, 33, 37, 43, 44, 51, 55, 58, 61, 70, 71, 95, 111, 112, 118, 135, 136, 144, 172, 176, 189, 191, 198, 199, 208, 210, 213–215, 217, 231, 232, 234, 239, 245, 250, 251, 259, 278, 280, 310, 321, 346
- one-hot vector, 224
- open set, 157
- optimal solution, 8, 10, 12, 18–20, 26, 37, 41, 46, 47, 49, 51, 52, 55, 63, 87, 88, 122, 127, 134, 135, 137, 138,

- 150, 156, 162, 176, 180, 185, 198, 269  
optimal value, 12, 51–53, 55, 59, 122, 123, 127, 130, 132, 133, 140–142, 145, 160, 163, 167, 169, 178, 183, 184, 279, 346  
optimization, 1–11, 17, 18, 22, 23, 26, 29, 32, 33, 36–40, 43, 48, 51, 54, 55, 59–62, 80, 86, 88, 92, 96, 97, 100, 110–114, 121–127, 129, 130, 132, 138–141, 143, 160, 162–164, 166, 169, 171, 173, 175, 177–180, 183–185, 188, 189, 191–194, 198, 200–202, 208–210, 213–215, 220, 221, 228, 240–242, 244–246, 248–254, 260, 262, 269, 271, 273–275, 277–280, 282, 284, 285, 294–296, 299, 300, 304–308, 310, 311, 313–315, 320–322, 324, 334, 336, 343, 345, 346  
optimization variable, 2, 3, 11, 23, 29, 32, 33, 36, 37, 39, 43, 51, 59, 96, 100, 110, 111, 114, 125, 141, 163, 169, 171, 173, 175, 179, 189, 212, 274, 278, 310, 345  
optimization variable, 51, 70, 91, 101, 114  
output layer, 38, 201, 202, 204, 205, 211, 222, 223, 226, 231, 232, 234, 237, 239, 240, 255, 257, 287, 288, 299, 319, 325, 351–353  
padding, 289–291  
parameters, 5, 97, 103, 114, 137, 193, 238, 240, 243, 252, 256, 263, 286, 306, 345, 346  
pattern recognition, 206  
penalized LS, 97  
Perceptron, 189, 199–202, 205, 229  
pointwise minimum, 126  
polygon, 14  
polyhedron, 14  
polytope, 14, 25, 49  
pooling layer, 287  
positive semidefinite, 81, 82  
positive semidefinite matrix (PSD), 82  
primal problem, 125, 126, 129, 135, 141, 144, 145, 163, 164, 167, 176, 179, 274  
principal component analysis (PCA), 113  
probability, 1, 2, 4, 33, 54, 68, 98, 117, 196, 224, 229, 243, 244, 247, 249, 250, 272, 298, 308–311, 316–318, 323, 340, 353  
probability distribution, 241, 339  
pseudo-inverse, 119, 297  
Python, 2, 3, 23, 26, 50, 55, 80, 81, 85, 234, 235, 267, 328, 329, 331, 334, 335, 338, 343  
quadratic cone, 94  
quadratic program (QP), 8, 86  
radiology, 72, 73  
random processes, 2  
range, 60, 81, 195, 211–213, 255, 285–287, 315  
range space, 59, 60  
real data, 241–244, 246, 247, 299, 301  
real samples, 271, 278, 283  
recidivism score predictor, 300  
rectified linear unit (ReLU), 211  
regularization, 63, 64, 68–70, 81, 84, 85, 97, 181, 300, 304, 307, 313, 321, 324  
regularization factor, 68  
regularization technique, 67  
representation learning, 241  
resizing, 340  
resource allocation, 28  
resource allocation problems, 28  
RMSprop, 286, 291, 299  
RMSprop optimizer, 286, 291, 299  
robust LP, 97

- robust LS, 118
- samples, 35, 85, 187, 188, 193, 196, 198, 201, 207, 240, 252, 301, 316
- Schur complement, 103, 105, 107, 109, 173, 175
- Schur complement lemma, 103, 105, 107, 109, 119, 173, 175
- scipy, 340
- scipy.stats, 336, 340
- SDP, 103, 108, 109, 112, 113, 122, 123, 167, 168, 175, 178, 179, 183
- SDP relaxation, 108, 109, 112, 121–123, 168, 175, 178, 179, 183
- second-order cone (SOC) constraint, 96, 100, 102, 104, 105
- second-order cone program (SOCP), 8, 92, 93
- seen data, 69
- semi-definite program (SDP), 8, 51, 102, 103
- sensitive attributes, 301, 302
- separating hyperplane theorem, 146, 150, 152, 154, 157, 158, 160
- set, 2, 11–14, 16, 17, 24, 25, 27, 29, 37–40, 45–48, 56, 69, 77, 81, 94–96, 100, 103, 110, 111, 119, 120, 122, 123, 134–136, 138, 145–147, 151, 152, 154, 157, 172, 173, 175, 177, 186, 212, 226, 227, 235, 249, 258, 269, 271, 276, 278, 282, 285, 286, 292, 332, 333, 353
- SGD, 319
- Shannon entropy, 198, 230, 238, 308
- shortest path problem, 39, 41, 42, 57
- sigmoid function, 192
- simplex algorithm, 42, 43, 45–47, 49, 50, 53, 56, 57, 162, 180, 274
- singular values, 109
- singular value decomposition (SVD), 340
- slack form, 43, 44, 56, 57
- slack variable, 44, 45, 56
- Slater’s condition, 158
- SOC, 94, 96, 102, 104
- SOC constraint, 96, 102, 104
- SOCP, 9, 93, 96, 100, 103, 104, 107, 109, 115, 124, 162
- softmax, 221–224, 226, 227, 229, 237, 299, 311, 351–353, 355
- spurious local minima, 214, 221, 240
- squared error, 61, 87, 188, 239
- standard form, 11, 17, 28–30, 36, 42, 43, 54, 56, 84, 86, 90, 93, 95, 100, 103–106, 111, 113, 125, 135, 150, 163, 166, 170, 172, 175, 176, 345
- standard Gaussian distribution, 99
- standard Gaussian distribution, 337, 339
- stationary point, 19, 28, 58, 134, 137, 138, 142, 166, 171, 253
- stochastic gradient descent (SGD), 319
- stride, 289, 290
- strong duality, 9, 22, 28, 36, 37, 42, 88, 90, 100, 113, 115, 124, 127–131, 133, 138, 139, 143, 144, 150, 155, 157, 158, 160, 162, 163, 179–181, 185, 274, 321
- strong duality theorem, 127, 133, 139, 143, 150, 155, 158, 160, 162, 179, 185, 274, 321
- supervised learning, 9, 180, 185, 187, 188, 193, 201, 220, 221, 227, 238, 240, 241, 299–302, 315, 321
- support vector machines (SVMs), 207
- SVD, 340
- symmetric matrix, 112, 171
- synapses, 190
- tall matrix, 59
- tanh function, 211–213
- TensorFlow, 2, 3, 200, 220, 221, 225, 240, 251, 252, 257, 259, 261, 284, 285,

- 287, 290, 291, 299, 300, 311, 313, 319, 348, 349  
test data, 65, 69, 85, 225  
test error, 63–69, 72, 80, 81, 85  
TNR, 66  
total probability law, 33, 54, 308, 309, 318  
TPR, 66  
trace, 114, 115, 119–121, 159  
tractable optimization problems, 8, 10  
train data, 65, 69  
training instability, 268  
transportation problems, 23, 33  
transportation theory, 28, 31, 33  
true negative rate (TNR), 66  
true positive rate (TPR), 66  
two-player game, 248, 311  
two-sided Laplace transform, 117  
unbounded below, 171  
unconstrained optimization, 58, 143, 150, 166, 321  
unconstrained QP, 171, 178  
undirected graph, 183  
uniqueness theorem of Laplace transform, 117  
unseen data, 65  
unsupervised learning, 9, 180, 228, 241, 242, 246, 294, 299, 301, 321  
validation data, 69  
Wasserstein distance, 33, 273, 274, 278, 279, 283, 295, 296  
Wasserstein GAN, 34, 261, 271, 282, 285, 295, 298, 299  
weak duality, 9, 130, 156, 162–165, 167, 169, 180, 184, 185, 321  
weak duality theorem, 162, 164, 169, 180, 184, 185, 321  
weight clipping, 286, 290, 291  
weights, 110, 121, 188, 189, 195, 199, 212, 215, 231, 235, 239, 240, 253, 262, 286, 292, 304  
WGAN, 271, 274, 277, 278, 282–288, 291, 294, 299, 321  
WGAN optimization, 274, 277, 278, 283–285, 291  
wide matrix, 59  
X-ray, 72–76, 78, 82, 84, 85

## About the Author



**Dr. Changho Suh** is an Associate Professor of Electrical Engineering at KAIST. He received the B.S. and M.S. degrees in Electrical Engineering from KAIST in 2000 and 2002 respectively, and the Ph.D. degree in Electrical Engineering and Computer Sciences from UC Berkeley in 2011. From 2011 to 2012, he was a postdoctoral associate at the Research Laboratory of Electronics in MIT. From 2002 to 2006, he was with Samsung Electronics.

Prof. Suh is a recipient of numerous awards in research and teaching: the 2022 Google Research Award, the 2021 James L. Massey Research & Teaching Award for Young Scholars from the IEEE Information Theory Society, the 2020 LINKGENESIS Best Teacher Award (the campus-wide Grand Prize in Teaching), the 2019 AFOSR Grant, the 2019 Google Education Grant, the 2018 IEIE/IEEE Joint Award, the 2015 IEIE Haedong Young Engineer Award, the 2015 Bell Labs Prize finalist, the 2013 IEEE Communications Society Stephen O. Rice Prize, the 2011 David J. Sakrison Memorial Prize (the best dissertation award in UC Berkeley EECS), the 2009 IEEE ISIT Best Student Paper Award, and the five Department Teaching Awards (2013, 2019, 2020, 2021, 2022). Dr. Suh is a Distinguished Lecturer of the IEEE Information Theory Society from 2020 to 2022, the General Chair of the Inaugural IEEE East Asian School of Information Theory 2021, an Associate Head of the KAIST AI Institute from 2021 to 2022, and a Member of the Young Korean Academy of Science and Technology. He is also an Associate Editor of Machine Learning for IEEE TRANSACTIONS ON INFORMATION THEORY, a Guest Editor for

the IEEE JOURNAL ON SELECTED AREAS IN INFORMATION THEORY, the Editor for IEEE INFORMATION THEORY NEWSLETTER, an Area Editor for IEEE BITS the Information Theory Magazine, an Area Chair of NeurIPS 2021–2022 and a Senior Program Committee of IJCAI 2019–2021.