# Algorithmic Mathematics in Machine Learning

# Data Science Book Series



Daniela Calvetti and Erkki Somersalo, *Mathematics of Data Science: A Computational Approach to Clustering and Classification*
Nicolas Gillis, *Nonnegative Matrix Factorization*
Bastian Bohn, Jochen Garcke, and Michael Griebel, *Algorithmic Mathematics in Machine Learning*

# Algorithmic Mathematics in Machine Learning

Bastian Bohn
University of Bonn
Bonn, Germany

Jochen Garcke
University of Bonn
Bonn, Germany
*and*
Fraunhofer SCAI
St. Augustin, Germany

Michael Griebel
University of Bonn
Bonn, Germany
*and*
Fraunhofer SCAI
St. Augustin, Germany

siam
Society for Industrial and Applied Mathematics
Philadelphia

10 9 8 7 6 5 4 3 2 1



| | |
|---|---|
| *Publications Director* | Kivmars H. Bowling |
| *Executive Editor* | Elizabeth Greenspan |
| *Acquisitions Editor* | Paula Callaghan |
| *Developmental Editor* | Rose Kolassiba |
| *Managing Editor* | Kelly Thomas |
| *Production Editor* | Ann Manning Allen |
| *Copy Editor* | Ann Manning Allen |
| *Production Manager* | Rachel Ginder |
| *Production Coordinator* | Cally A. Shrader |
| *Compositor* | Cheryl Hufnagle |
| *Graphic Designer* | Doug Smock |

**Library of Congress Cataloging-in-Publication Data**

Names: Bohn, Bastian (Mathematician), author. | Garcke, Jochen, author. | Griebel, Michael, 1960- author.

Title: Algorithmic mathematics in machine learning / Bastian Bohn (University of Bonn, Bonn, Germany), Jochen Garcke (University of Bonn, Bonn, Germany), Michael Griebel (University of Bonn, Bonn, Germany).

Description: Philadelphia : Society for Industrial and Applied Mathematics, [2024] | Includes bibliographical references and index. | Summary: "Explores several well-known machine learning and data analysis approaches from a mathematical perspective and also implements and applies the underlying algorithms to achieve a programming and practical perspective"-- Provided by publisher.

Identifiers: LCCN 2023056447 (print) | LCCN 2023056448 (ebook) | ISBN 9781611977875 (paperback) | ISBN 9781611977882 (ebook)

Subjects: LCSH: Machine learning--Mathematics. | AMS: Computer science -- Artificial intelligence -- Learning and adaptive systems.

Classification: LCC Q325.5 .B594 2024 (print) | LCC Q325.5 (ebook) | DDC 006.3/1--dc23/eng/20240201

LC record available at *https://lccn.loc.gov/2023056447*

LC ebook record available at *https://lccn.loc.gov/2023056448*

Cover illustration reproduced with permission. Ziwei Liu, Ping Luo, Xiaogang Wang, and Xiaoou Tang. *Deep Learning Face Attributes in the Wild; International Conference on Computer Vision (ICCV), 2015.*

# Contents

# Preface

The story of machine learning is one of rigorous success. It is frequently employed by scientists and practitioners around the globe in various areas of application ranging from economics to chemistry, from medicine to engineering, from gaming to astronomy, and from speech processing to computer vision. While the remarkable success of machine learning methods speaks for itself, they are often applied in an ad hoc manner without much care for their mathematical foundation or for their algorithmic intricacies. Therefore, we decided to write this book. Our goal is to provide the necessary background on commonly used machine learning algorithms and to highlight important implementational and numerical details. The book is based on a well-received practical lab course, which we established within the mathematics studies course at the University of Bonn, Germany, in 2017. The course has been taught and successively enhanced each year since then.

In contrast to other books on machine learning, we aim to give mathematicians, computer scientists, and practitioners with a basic mathematical background in analysis and linear algebra an introduction to the algorithmic mathematics of many important machine learning methods. The book is composed of introductory parts, which present a specific machine learning method; of information parts, which recapitulate the underlying mathematics; and of practical exercises, which deepen the understanding of the considered method.

**Acknowledgments** We would like to thank all our co-workers, friends, and students who helped us to bring this book to life with their valuable input of various kinds. We especially thank Jannik Schürg, Arno Feiden, and Lisa Beer for their specific text contributions and remarks. Furthermore, we would like to thank Olmo Chiara Llanos, Paolo Climaco, Sara Hahner, Jan Hamaekers, Ivan Lecei, Sebastian Mayer, Daniel Oeltz, and Moritz Wolter for their valuable feedback during the practical lab courses and the writing process of this book.

We would also like to acknowledge the support by the German Federal Ministry of Education and Research, which partially funded the development phase of the practical lab within the project *P3ML* (project code 01IS17064).

Bastian Bohn, Jochen Garcke, and Michael Griebel

Bonn, October 2023

# List of Figures

# Chapter 1
# The Basics of Machine Learning

In many branches of science, economy, and industry, the amount of available data has become immense during recent years. Most of these data do not contain any valuable information. However, the differentiation between useful data and "data waste" is seldom straightforward. Moreover, in applications where expensive (practical or numerical) experiments are involved in the data acquisition process, the available data sets are rather small. To meet the different challenges on real-world data sets, like, for example, describing useful information in a more compact format (*dimensionality reduction*) or making predictions on unseen data (*regression*), many different ideas and approaches have emerged. These are usually grouped together under the name ***machine learning*** (ML) methods.

Machine learning itself is a subarea of *artificial intelligence* (AI) and a related subject to *data mining* (DM). As a core topic, AI is concerned with all aspects of machine intelligence, i.e., with cases where a (real or virtual) artificial device is able to take actions to achieve a given goal by using information, e.g., from sensors, on the surrounding environment in which it lives. The term *data mining*, on the other hand, is usually used to describe methodologies with the goal of detecting statistical trends or relations within or in between different data sets. Oftentimes, ML and DM are used somewhat ambiguously. To better distinguish these two areas, ML is typically used when the goal is to predict certain values or outcomes from given data, whereas DM is used when there is no prediction sought, but rather a statistical analysis of the data.

Generally, machine learning refers to the generation of knowledge by using computational (and statistical) methods on given data. This means that ML instances automatically *learn* from experience in order to achieve a certain *goal*, i.e., they adapt solutions to given data by optimizing some criterion. An important aspect to note is that the corresponding algorithms are not explicitly programmed to produce certain solutions in certain cases, but they determine—or *infer*—their solutions according to the given data.

## 1.1 ▪ Aim, organization, and notation

In this book, we explore several well-known machine learning and data analysis algorithms from both a ***mathematical*** perspective and a ***programming*** perspective. These two aspects will always serve as the main guidelines throughout all chapters and tasks. In this way, we want to specifically address readers with a background in mathematics, who are interested in both the mathematical foundation of the most commonly used modern-day machine learning algorithms and the practical knowledge on implementing and using them. Furthermore, we will apply the algorithms to real-world data sets to get an intuition on the specific needs in different relevant

applications. In particular, we will get to know use-cases and data sets from the fields of handwritten digit recognition, pedestrian detection in images, and biological single-cell analysis, for instance.

**Programming exercises** Most chapters of this book will be accompanied by a variety of programming exercises, which are intended to be solved in JUPYTER notebooks. A short introduction on how to set up the appropriate programming environment can be found at https://bookstore.siam.org/di03/bonus. There, we also provide templates for some of the programming exercises. We assume that the reader is familiar with basic PYTHON tools and libraries. However, we also provide a brief tutorial on the most important PYTHON and NUMPY concepts. Moreover, we will learn how to use the machine learning libraries SCIKIT-LEARN and KERAS. The latter serves as an interface to TENSORFLOW.

**Info-boxes** Three different categories of info-boxes throughout the book will help as an additional source of information.

These boxes serve as a brief overview of important, newly introduced ML terminologies. They are not meant to be comprehensive, but their content aims to illustrate the idea behind the new concept.

These boxes provide brief background information on mathematical concepts, which are not necessarily needed to understand the most important content in the corresponding section. However, they serve as a reminder on these concepts and hint at additional facts for a more detailed explanation.

These boxes contain the practical exercises. Here, the reader is encouraged to implement and test the introduced concepts and algorithms from the specific chapter. This serves to deepen understanding and to get some feeling for potential pitfalls when working with these algorithms.

**Content** The remainder of this book is organized as follows:

- Chapter 1 introduces the basic terminology used in machine learning without going into details on specific methods and without providing any mathematical background. The main intention of Chapter 1 is to create merely a common understanding of fundamental concepts of ML, which serves as a preparation for the following chapters.
- Chapter 2 deals with two commonly used simple *supervised learning* algorithms serving as an entry into the world of machine learning: *linear least squares* and *k-nearest neighbors*. Moreover, first tasks help to familiarize the reader with the PYTHON library NUMPY.
- Chapter 3 introduces *support vector machines* as a more complicated ML method. After a thorough mathematical derivation of the basic algorithm, we have a look at nonlinear variants and get to know the SCIKIT-LEARN library and the MNIST data set of handwritten digits.
- Chapter 4 provides the first *unsupervised learning* algorithm of the book: *principal component analysis*. After a look at different mathematical interpretations of this *dimensionality reduction* algorithm, we examine an image data set with photos of pedestrians.

- Chapter 5 deals with nonlinear approaches to dimensionality reduction in form of *Isomap* and *diffusion maps*. Here, we also introduce the use-case of biological single-cell analysis.
- Chapter 6 provides an introduction to the topic of *artificial neural networks* and *deep learning*. Here, the KERAS and TENSORFLOW libraries are used to construct neural networks and to analyze given data sets with them.
- Chapter 7 specializes on a specific type of neural network: *variational autoencoders*. After first introducing *autoencoders* and highlighting their connection to the principal component analysis of Chapter 4, we introduce the necessary statistical concepts to also establish their variational variant.
- Chapter 8 is concerned with the connection between deep learning algorithms and *ordinary differential equations* or *partial differential equations*. We present the concepts of *neural tangent kernels*, *neural ODEs*, and *generative diffusion models*.
- Chapter 9 introduces the reader to the world of *reinforcement learning*. Here, the necessary basics of *optimal control theory* are provided. Then, we have a look at typical problems that can be tackled by (deep) reinforcement learning algorithms.
- Chapter 10 concludes this book by giving short introductions to important ML concepts and algorithms that we did not cover in detail in the previous chapters. More specifically, we will have a look at *recurrent neural networks* and *transformer networks*, as well as *non-vectorial data*, *ensemble methods*, and methods for *interpretability*.

**Notation** Since the main intention of this chapter is to create common ground for comprehending the contents of the further chapters, we aim to keep the notation for the remainder of this book mostly consistent with that used in this chapter. To this end, let us provide the naming conventions, which are valid throughout most of this book:

- Vectors are commonly written with small letters in boldface notation, e.g., $\boldsymbol{x}$, or with an arrow on top, e.g., $\vec{x}$, to avoid confusion. The $j$th entry of an indexed vector $\boldsymbol{x}_i$ is denoted by $[\boldsymbol{x}_i]_j$.
- Matrices are commonly written with capital letters in boldface notation, e.g., $\boldsymbol{W}$.
- $n$ and $\bar{n}$ denote the number of given training and test data points, respectively.
- $\mathcal{D} := \{(\boldsymbol{x}_i, y_i) \in \Omega \times \Gamma \mid i = 1, \dots, n\}$ describes the training data set in supervised learning. The $y_i$ are omitted in unsupervised learning.
- $\bar{\mathcal{D}} := \{(\bar{\boldsymbol{x}}_i, \bar{y}_i) \in \Omega \times \Gamma \mid i = 1, \dots, \bar{n}\}$ describes the test data set in supervised learning. The $\bar{y}_i$ are omitted in unsupervised learning.
- $d$ denotes the dimension of a data point $\boldsymbol{x}$.
- $\tilde{d}$ and $q$ are commonly taken to denote dimensions different from $d$.
- "$k$-dimensional" is abbreviated by "$k$D" for any $k \in \mathbb{N}$.
- The norm $\|\cdot\|$ refers to the Euclidean norm $\|\cdot\|_2$ in the corresponding ambient space, e.g., $\mathbb{R}^d$, unless otherwise noted.
- $\mathcal{M}$ denotes the model space.
- $\mathcal{L}$ denotes the loss function.
- $\mu$ is the measure according to which training data have been drawn. In the case of supervised learning we also use $\mu_X$ as the marginal measure on $\Omega$ and $\mu_{Y|X}$ as the conditional measure of $Y$ given $X$ on $\Gamma$.

In the remainder of this chapter, we introduce the basic terminology and concepts of machine learning that will be relevant throughout this book. This serves as a motivation and as an overview on fundamental aspects for readers who are unfamiliar with the ML jargon. To this end, we deliberately omit technical subtleties here.

We begin by introducing the necessary mathematical concepts. In particular, Section 1.2 is dedicated to the topics of supervised and unsupervised learning. The mathematical formulation of the underlying optimization problem is dealt with in Section 1.3. Section 1.4 presents hyperparameters of machine learning models. Feature engineering and feature selection are discussed in Section 1.5. The concentration of measure phenomenon and the curse of dimensionality for high-dimensional problems are the topics of Section 1.6. Subsequently, Section 1.7 and Section 1.8 are dedicated to more applied aspects of machine learning. In particular, Section 1.7 deals with general process models for tackling real-world machine learning problems, e.g., the CRISP-DM model. Section 1.8 discusses important implementational aspects, such as preferable programming languages and libraries for machine learning. Finally, Section 1.9 provides references to other important textbooks on mathematical and implementational aspects of machine learning.

Note again that we deliberately only provide motivational and fundamental mathematical and implementational concepts in this chapter. The corresponding details on the topics discussed here as well as specific instances of ML methods and algorithms will follow in the later chapters.

## 1.2 ▪ Supervised and unsupervised learning

Many tasks and algorithms can be roughly divided into two different categories: *supervised learning* and *unsupervised learning*.

**Supervised learning**

We are given data samples of *inputs* and *outputs*. The goal is to predict the output to an input. This can be seen as a function (or density) reconstruction task. The prediction should also generalize well to yet unseen data.

**Example:** Assume we are given real values $\boldsymbol{x}_1, \dots, \boldsymbol{x}_5$ as inputs and real values $y_1, \dots, y_5$ as outputs. A possible *predictor* would be the piecewise linear interpolant of the $(\boldsymbol{x}_i, y_i)$ pairs (indicated by the blue graph below).

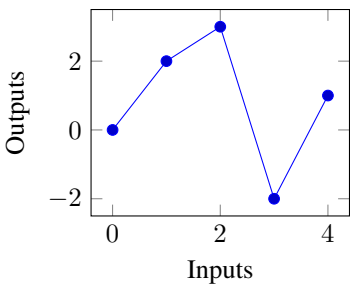



Figure 1.1

**Unsupervised learning**

We are only given *input* samples. Common tasks are clustering/segmentation of the data or compression/dimensionality reduction, where we look for a low-dimensional representation of the high-dimensional input data.

**Example:** Assume we are given 12 two-dimensional points $\boldsymbol{x}_1, \dots, \boldsymbol{x}_{12}$. A possible goal could be to assign each point to one of three groups/*clusters* (indicated by color and shape below) according to their distances to each other.

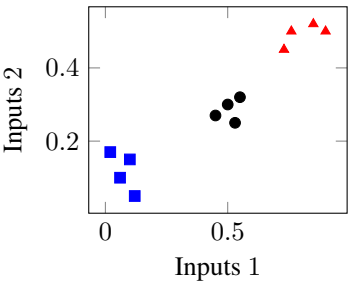



Figure 1.2

**Supervised learning** Let us focus on supervised learning first. Here a function[1] $f$ is learned from input-output samples $(\boldsymbol{x}_i, y_i)$, also referred to as inputs and *labels*. The goal is not only that the samples—usually called *training data*—be (approximately) fitted by $f$, but also that new data—usually called *test data* or *evaluation data*—which stem from the same distribution as the training data, be approximated well. Some specific examples of tasks in supervised learning are:

- Classify images according to their content (e.g., cats versus dogs) [KSH12].
- Estimate the risk of disease from patient data [KZK12].
- Mark email messages that are spam [DBC+19].
- Detect critical failures in industrial facilities [CGIT23].
- Predict future values of financial assets [HSK19].
- Categorize musical pieces according to their genre [COSJ17].

In supervised learning, we assume that we are given an input training data set

$$\mathcal{D} := \{(\boldsymbol{x}_i, y_i) \in \Omega \times \Gamma \mid i = 1, \dots, n\} \quad \text{with} \quad (\boldsymbol{x}_i, y_i) \overset{\text{i.i.d.}}{\sim} \mu \; \forall \, i \in \{1, \dots, n\}.$$

Here, the $n$ training data points are independent and identically distributed (i.i.d.) samples of an unknown probability measure $\mu$ on $\Omega \times \Gamma$. Usually (and most of the time in this book) the setting $\Omega \subseteq \mathbb{R}^d$ and $\Gamma \subseteq \mathbb{R}$ is considered, which already covers a lot of interesting machine learning problems. Note however that different settings, such as non-scalar $\Gamma$ or categorical $\Omega$, may be encountered when considering certain applications, see Section 10.1.

In supervised learning we are looking for $f : \Omega \to \Gamma$ such that

$$f(\boldsymbol{x}_i) \approx y_i \quad \forall \, i \in \{1, \dots, n\}. \tag{1.1}$$

But instead of just choosing an interpolant in $\boldsymbol{x}_i$, which does the job, we more importantly also aim for

$$f(\boldsymbol{x}) \approx y \quad \forall (\boldsymbol{x}, y) \sim \mu. \tag{1.2}$$

If $\Gamma$ is endowed with a distance measure, this is called a *regression* problem. In the special case of $|\Gamma| < \infty$ and where there is no defined order on $\Gamma$, i.e., the values of $y$ represent categories, we are dealing with a *classification* problem. To make the search for such a function $f$ more feasible, we usually restrict ourselves to some model space or class $\mathcal{M}$ and try to find the "best" $f \in \mathcal{M}$ that fulfills our requirements of fitting the data. We will go into more detail on model classes and what we mean by "best" $f$ in the next sections.

**Regression**

***Regression*** refers to the determination of the values of a continuous variable, given the values of other variables. We are mostly dealing with the reconstruction of a real-valued function $g : \Omega \subset \mathbb{R}^d \to \Gamma \subset \mathbb{R}$ from given values $(\boldsymbol{x}_i, y_i)$ for $i = 1, \dots, n$, where $y_i = g(\boldsymbol{x}_i) + \varepsilon_i$ is a function evaluation of $g$ perturbed by some noise $\varepsilon_i$.

**Classification**

***Classification*** deals with the determination of the class/group which a data point belongs to. Often, we aim to reconstruct the class assignment operator $g : \Omega \subset \mathbb{R}^d \to \Gamma = \{1, \dots, K\}$ from data points $(\boldsymbol{x}_i, y_i)$ for $i = 1, \dots, n$, where $y_i = g(\boldsymbol{x}_i)$ is the class to which $\boldsymbol{x}_i$ belongs. Here, $K \in \mathbb{N}$ is the number of possible classes.

[1] More generally, we could consider learning a measure that models the connection between the input-output pairs.

In contrast to general supervised learning methods, where we assume that the training data $\mathcal{D}$ is given beforehand, the class of *active learning* methods, which is sometimes considered as a subclass of supervised learning, deals with a scenario where the algorithm determines the training data points itself; see [Set12]. More specifically, in active learning we start with a large base set of $\boldsymbol{x}_i$ for $i = 1, \dots, N$. Then, in each step of the active learning algorithm, one of the points which has not been chosen before is included in the training data set. In particular $\boldsymbol{x}_{i_k}$ for some $i_k \in \{1, \dots, N\} \setminus \{i_1, \dots, i_{k-1}\}$ is chosen in the $k$th step and the corresponding label $y_{i_k}$ is determined. Then, a supervised learning algorithm is applied to the new training data set. Moreover, a case-dependent heuristic is employed to determine if the currently selected data points suffice or if more steps in the active learning algorithm are necessary, i.e., if more points need to be included in the training data set. This type of algorithm is useful when we have the possibility to infer the correct $y_{i_k}$—or at least a good approximation thereof—on the fly, while the process of doing so involves intensive computations or is very costly and the number of chosen data points should thus be kept small.

**Unsupervised learning** In unsupervised learning we are usually interested in an efficient compression/representation of the input data. This can be done by determining an adequate low-dimensional representation system for high-dimensional input vectors. Then, this procedure is referred to as *dimensionality reduction*.

**Dimensionality reduction**

In ***dimensionality reduction***, we usually have data $\boldsymbol{x}_i \in \Omega = \mathbb{R}^d$ for $i = 1, \dots, n$ and aim to find a "good" representation $\boldsymbol{\eta}_i \in \mathbb{R}^q$ with $q < d$. Here, "good" can mean that the data points retain their main characteristics, such as pairwise distances for instance. But "good" can also refer to the fact that it should be possible to approximately *reconstruct* the original data points $\boldsymbol{x}_i$ given just $\boldsymbol{\eta}_i$ and using an appropriate reconstruction algorithm. Oftentimes we assume that the original data stem from some $k$-dimensional surface or manifold, which we need to detect in order to determine the $\boldsymbol{\eta}_i$.

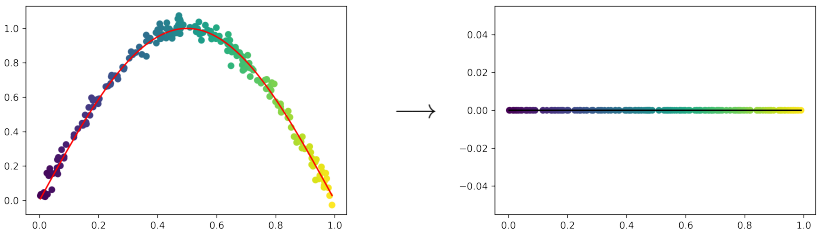



Figure 1.3

Above is an example for $d = 2$ and $q = 1$. The data points on the left approximately reside on the red curve, which is just one-dimensional. Thus, the data can be represented in one dimension as points on the black line, shown on the right.

Sometimes, instead of performing dimensionality reduction of the data, one is merely interested in compressing the data with respect to the number $n$ of data points. Here, resampling approaches like bootstrapping play an important role; see Section 10.2.4 for more details. Furthermore, it is a common goal in unsupervised learning to find yet undetected patterns in the data, which is often pursued for already dimension-reduced data. This is for instance achieved by segmentation or *clustering* of the input data into several classes.

**Clustering**

The goal of ***clustering*** is to group together sets of data points (so-called ***clusters***) according to an underlying similarity measure. In particular, we divide a data set into $k \in \mathbb{N}$ different parts $C_1, \ldots, C_k$. Here, similar points are assigned to the same cluster, whereas dissimilar points are assigned to different clusters. There exists a variety of clustering methods which differ in the choice of the underlying similarity measure. A thorough discussion of clustering and a description of suitable algorithms can be found in [XW08].

Common examples of applications for unsupervised clustering algorithms are:

- Grouping of gene or protein sequences according to their similarity [NZM+19].
- Separation of tissues in medical images [RBU+20].
- Anomaly/outlier detection in industrial time series data [HGG+20].
- Organization of communities in social networks [ARB19].
- Visual exploration of large-scale databases [YHW+07].

The similarity measure in a clustering algorithm can be defined in various ways using point distances or densities, for instance. While standard algorithms like $k$-means, which we will explore in more detail in Section 5.3, are commonly based on the Euclidean distance between two data points, it can also be meaningful to choose the underlying similarity measure according to the application at hand. For example, when clustering different shapes of the human body, $k$-means can be used with the so-called *Procrustes* distance measure instead; see [ADLS10] for more details. Here a rotation- and translation-invariant distance measure is employed to determine the coherence of two different shapes. A thorough collection of similarity measures and divergences, which also stem from practical applications, can be found in [DD09].

Note that clustering can also be used to reduce the number of data points. To this end, a representative data point for each cluster is chosen. Subsequently, all data points in a particular cluster are identified with its representative.

## 1.3 ▪ The underlying optimization problem

Now that we have seen what typical supervised and unsupervised learning problems look like, we want to highlight the fact that there usually is an underlying optimization problem behind every ML algorithm. Consider supervised learning; here we look for a function $f$, and the quantification of what is meant by $\approx$ in (1.1) and (1.2) as suggested for regression or classification is implicitly contained in the formulation of the optimization problem. Before we can formulate an ML optimization problem, we need to introduce a respective model class $\mathcal{M}$ and a respective loss function $\mathcal{L}$.

### 1.3.1 ▪ Model classes

The choice of an appropriate *model class* $\mathcal{M}$ is itself an art. From an algorithmic and function approximation point of view, it is a balancing act between a too-small class, which does not lead to a satisfactory solution because there is no $f \in \mathcal{M}$ that fits the data, and a too-large class, which makes solving the optimization problem tedious because $\mathcal{M}$ cannot be efficiently and exhaustively searched for a good solution. Generally, from an ML point of view, we seek a tradeoff between *bias* and *variance*. A high bias means that the model is very coarse and does not

represent the given data very well. A high variance, however, means that the model is sensitive to small fluctuations in the data and therefore does not generalize well to yet unseen data, i.e., the model has large errors on the test data. In particular, a model that fits the training data well, but does not perform equally well on the test data, has *low bias and high variance*, i.e., the model class is too large. The opposite problem of *high bias and low variance* is present when the model class is too small for the given data. In general, more data typically reduce the variance of the obtained model. Therefore, we can obtain a smaller variance for larger model classes as the amount of data increases. In contrast, aiming to minimize the variance by using a small model class is usually more relevant in a setting where only a few data points are available.

**Model class**

The ***model class*** $\mathcal{M}$ is the set of all possible functions considered as solutions to the machine learning problem at hand. Famous model classes include affine linear functions, splines, kernel spaces, and (deep) neural networks. Oftentimes functions in the model class are determined by certain parameters $\boldsymbol{p} \in \mathbb{R}^{d_p}$. In this case, the model class is called *parametrized*.

Let us consider the simple example of affine linear functions in the setting $\Omega = \mathbb{R}^d$ and $\Gamma = \mathbb{R}$. Here, the parametrized model class is given by

$$\mathcal{M}_{\text{Lin}} = \left\{ g : \mathbb{R}^d \to \mathbb{R} \;\middle|\; g(\boldsymbol{t}) = \gamma_0 + \sum_{i=1}^{d} \gamma_i t_i \text{ for some fixed } \gamma_0, \ldots, \gamma_d \in \mathbb{R} \right\}.$$

The $d_p = d + 1$ parameters are $\boldsymbol{p} = \boldsymbol{\gamma} := (\gamma_0, \ldots, \gamma_d)^T$. In the case of the affine linear model class, we will often use the more compact notation $g(\boldsymbol{t}) = \boldsymbol{\gamma}^T \cdot \hat{\boldsymbol{t}}$ with $\hat{\boldsymbol{t}} := (1, t_1, \ldots, t_d)^T$. In the case of parametrized model classes, we sometimes write $g_{\boldsymbol{p}}$ instead of $g$ to make the dependence on $\boldsymbol{p}$ more distinct.

### 1.3.2 ▪ Loss function

The *loss* $\mathcal{L}$—sometimes also referred to as *cost*—is the function whose minimization serves as the goal. In many cases in ML, loss functions are defined in such a way that they are convex. Depending on the model class $\mathcal{M}$, this convexity allows us to draw conclusions on the existence and uniqueness of minimizers $f \in \mathcal{M}$ of $\mathcal{L}$. Another important aspect of loss functions is their smoothness with respect to the inputs. A smooth loss function is beneficial when its derivatives are needed within an optimization procedure.

**Loss function**

The determination of $f \in \mathcal{M}$ is done with the help of a ***loss function*** $\mathcal{L}$. For supervised learning it is often defined as a function $\mathcal{L} : (\Gamma \times \Gamma)^n \to [0, \infty]$. Note that most loss functions can be written in terms of a *one-sample* loss function $\tilde{\mathcal{L}} : \Gamma \times \Gamma \to [0, \infty]$. For instance, a so-called *additive* loss function $\mathcal{L}$ can be written with one-sample loss $\tilde{\mathcal{L}}$ as

$$\mathcal{L}\left((z_1, \tilde{z}_1), \ldots, (z_n, \tilde{z}_n)\right) = \frac{1}{n} \sum_{i=1}^{n} \tilde{\mathcal{L}}\left((z_i, \tilde{z}_i)\right).$$

With the help of a model class and a loss function, we can now write down the actual minimization problem underlying most (supervised) machine learning algorithms.

**The ML minimization problem**

The continuous machine learning problem corresponding to the model class $\mathcal{M}$ and the additive loss $\mathcal{L}$ with one-sample loss $\tilde{\mathcal{L}}$ can be written as

$$\min_{g\in\mathcal{M}} \mathbb{E}_\mu \left[\tilde{\mathcal{L}}((g(X),Y))\right] = \min_{g\in\mathcal{M}} \int_{\Omega\times\Gamma} \tilde{\mathcal{L}}((g(\boldsymbol{x}),y))\, \mathrm{d}\mu(\boldsymbol{x},y), \tag{1.3}$$

i.e., we average the loss over all possible data points drawn i.i.d. according to $\mu$. The resulting quantity is also known as the ***estimation error*** or ***generalization error***. However, we usually do not have direct access to the underlying measure $\mu$ but only to the training samples $(\boldsymbol{x}_i, y_i)$ for $i = 1,\dots,n$. Therefore, we substitute the problem of minimizing the generalization error by the problem of minimizing the training error (or *empirical loss*), i.e., we aim to determine

$$f := \operatorname*{arg\,min}_{g\in\mathcal{M}} \frac{1}{n}\sum_{i=1}^{n} \tilde{\mathcal{L}}\left((g(\boldsymbol{x}_i), y_i)\right) = \operatorname*{arg\,min}_{g\in\mathcal{M}} \mathcal{L}\left((g(\boldsymbol{x}_1), y_1), \dots, (g(\boldsymbol{x}_n), y_n)\right). \tag{1.4}$$

In the case of a parametrized model class $\mathcal{M}$, we can rewrite this as

$$f := \operatorname*{arg\,min}_{\boldsymbol{p}\in\mathbb{R}^{d_p}} \mathcal{L}\left((g_{\boldsymbol{p}}(\boldsymbol{x}_1), y_1), \dots, (g_{\boldsymbol{p}}(\boldsymbol{x}_n), y_n)\right), \tag{1.5}$$

where the function $g_{\boldsymbol{p}} \in \mathcal{M}$ implicitly depends on $\boldsymbol{p}$. When it comes to unsupervised learning it is not straightforward how to write down a general optimization problem that is valid for most methods. However, an appropriate minimization problem can often still be formulated on a case by case basis, as we will see in Chapter 4 and Chapter 7.

One of the most common loss functions is the *least squares* loss

$$\mathcal{L}_{\mathrm{ls}}\left((z_1,\tilde{z}_1),\dots,(z_n,\tilde{z}_n)\right) := \frac{1}{n}\sum_{i=1}^{n}(z_i-\tilde{z}_i)^2,$$

which is an additive loss function with one-sample loss $\tilde{\mathcal{L}}(z,\tilde{z}) = (z-\tilde{z})^2$. The least squares loss computes the normalized squared Euclidean norm of the difference of the vectors $\boldsymbol{z} := (z_1,\dots,z_n)^T$ and $\tilde{\boldsymbol{z}} := (\tilde{z}_1,\dots,\tilde{z}_n)^T$. When evaluating the loss in an ML setting with model function $g$, the $z_i$ and $\tilde{z}_i$ are instantiated by the point evaluations $g(\boldsymbol{x}_i)$ and $y_i$ for $i = 1,\dots,n$. In particular, by using the least squares loss function as in the general optimization problem (1.4), we obtain the least squares ML minimization problem

$$f := \operatorname*{arg\,min}_{g\in\mathcal{M}} \frac{1}{n}\sum_{i=1}^{n}\left(g(\boldsymbol{x}_i)-y_i\right)^2.$$

Besides the least squares loss many other options exist for $\mathcal{L}$. Simple examples are different vector norms. For classification problems, *logistic loss* functions or *cross entropies* are often used; see, e.g., Section 6.4.5. In the case of more complicated data sets, e.g., when $y_i$ is a function or a density for all $i = 1,\dots,n$, the choice of an appropriate loss function becomes more tricky; see also Section 10.1.

### 1.3.3 ▪ Solving the optimization problem

We have seen that different combinations of the model class $\mathcal{M}$ and the loss function $\mathcal{L}$ lead to different optimization problems (1.3) and thus to different ML methods. Although we are essentially after minimizing the generalization error in machine learning, we usually cannot achieve this directly. Therefore, we solve (1.4) instead, which leads us to the question of how to choose $\mathcal{M}$ and $\mathcal{L}$ and to the question of how to actually solve (1.4).

**Properties of minima** Let us consider the parametrized variant (1.5) of the optimization problem. If $\mathcal{L}$ is continuously differentiable, we know from basic calculus that

$$\nabla_{\boldsymbol{p}} \mathcal{L}((g_{\boldsymbol{p}}(\boldsymbol{x}_1), y_1), \ldots, (g_{\boldsymbol{p}}(\boldsymbol{x}_n), y_n)) = 0$$

is a necessary condition for $g_{\boldsymbol{p}}$ to be a minimizer of the loss on the training data. Points that fulfill this equation are called *critical points*. Additionally, if the Hessian $\nabla^2_{\boldsymbol{p}} \mathcal{L}$ exists and is positive definite for $g_{\boldsymbol{p}}$, we encountered a minimizer. While these conditions help us to check for minima, we do not know if we encountered a local or a global minimum in general.

The reason for the popularity of convex loss functions, like the least squares loss, is that they ensure that any critical point of $\mathcal{L}$ is automatically a *global* minimizer if a convex model class is employed, i.e., a model class that is a convex set of functions. While this does not necessarily mean that the minimizer is unique, each minimizer is equally good in the sense that it achieves the same minimum value of $\mathcal{L}$ as the others do.

**Numerical optimizer** While the check for the optimality conditions above can be done by hand for specific choices of $\mathcal{L}$ and $\mathcal{M}$, a minimizer usually needs to be determined by numerical optimization algorithms. Here, a suitable choice of an algorithm has to be made in accordance with the properties of the optimization problem (1.4).

For instance, for the least squares loss and an affine linear model class, the minimizer is unique, and its naive determination boils down to solving a system of linear equations, as we will see in Chapter 2. An example for a supervised learning problem whose optimization problem deviates from the form (1.4) is given in Chapter 3. Here, a constrained convex optimization problem is tackled by an iterative solver. When we look at more complicated settings where the loss function or the model class is no longer convex, the numerical optimization becomes more intricate. Here, we usually encounter many local minima.

In general, a local minimum resulting from an iterative numerical optimizer depends on the initial value and the respective optimization method, provided that it is locally convergent in the first place. Which local minimum is attained and how much it differs from any global minimum are mostly unclear and difficult to determine in the non-convex situation. Even in the case of deep neural networks, for instance, the mathematical analysis of the optimization problem is hard, and its properties are not completely understood to this day; see Chapter 6. Here, iterative stochastic methods have heuristically proven to be the numerical optimizers of choice.

### 1.3.4 ▪ Overfitting and regularization

When the combination of $\mathcal{M}$, $\mathcal{L}$, and the data at hand does not lead to a unique minimum of (1.4), the problem is called *ill-posed*. In these cases, *regularization* is often applied, which, in the best case, leads to a *well-posed* problem, i.e., a problem with a unique solution. Regularizing a problem describes the process of adding additional information or constraints to the problem such that the numerical optimization becomes easier.

**Overfitting and regularization**

Usually regularization is done by adding constraints to the model class, either directly, e.g., by making it smaller, or indirectly, e.g., by adding an additional term to $\mathcal{L}$ that penalizes certain functions from $\mathcal{M}$ more than others. In ML, regularization prevents so-called ***overfitting***. This describes the event that a function is fitted to the training data too strictly and does not generalize well to unseen test data, which is an indirect effect of too-large model spaces, i.e., of the low bias and high variance regime.

Let us look at an example for adding a penalty term to a loss function in the case of linear least squares regression, i.e., when minimizing the least squares loss $\mathcal{L}_{\text{ls}}$ over the model class $\mathcal{M}_{\text{Lin}}$. After adding a penalty (or regularization) term to the parametrized variant (1.5) of the optimization problem, we obtain

$$\boldsymbol{\alpha} = \underset{\boldsymbol{\gamma}\in\mathbb{R}^{d+1}}{\arg\min} \frac{1}{n}\sum_{i=1}^{n} \left(\boldsymbol{\gamma}^T \cdot \hat{\boldsymbol{x}}_i - y_i\right)^2 + \lambda\|\boldsymbol{\gamma}\|_2^2, \tag{1.6}$$

where $f_{\boldsymbol{\alpha}}(\boldsymbol{t}) = \boldsymbol{\alpha}^T \cdot \hat{\boldsymbol{t}}$ is the resulting affine linear function with $\hat{\boldsymbol{t}} = (1, t_1, \ldots, t_d)^T$, as noted before. Here, we added the weighted (for a fixed $\lambda > 0$) squared $\ell_2$ norm of the coefficient vector as penalty. Now solutions with large coefficients are penalized more than those with small ones. This is known as *Tikhonov regularization* (see [EHN96, Tik63]) or *ridge regression* (see, e.g., [HTF09]).

## 1.4 ▪ Hyperparameters

Often there are so-called *hyperparameters* in ML algorithms, i.e., parameters that have to be set by the user. They have to be carefully chosen in order to achieve good results. Having a large number of hyperparameters is undesirable since their proper choice gets extremely complicated and often even practically impossible. We address the problem of hyperparameter search in more detail in Section 3.6.

**Hyperparameters**

***Hyperparameters*** are any parameters of the model $f \in \mathcal{M}$ or the loss $\mathcal{L}$ that are not determined by the optimization of the loss function but which have to be fixed before the loss minimization can be tackled. An example of a hyperparameter is the regularization parameter $\lambda$ in ridge regression introduced above.
An easy but computationally intensive way to search for the optimal hyperparameters is to compute a lot of solutions for different values of the hyperparameters on a part of the training data and then evaluate these solutions on another part of the training data to see which hyperparameter choice performed best.

## 1.5 ▪ Feature engineering and feature selection

A *feature* in ML is a measurable property or characteristic of the (input) data, which is then fed into an ML algorithm to determine a model and an output. Relevant features, which mainly guide the decision process of the ML algorithm, are gathered into a *feature vector*, which serves as the

input of an ML algorithm. In this way, the feature construction and selection can also be seen as a data preprocessing task, which is of major importance within the machine learning pipeline.

**Feature engineering and selection**

At a first stage, the process of constructing—or ***engineering***—and ***selecting*** features can be performed by hand if there is prior knowledge about important features. For example, when learning a radial property of a two-dimensional input $(x_1, x_2)$, a meaningful feature would be $r := \sqrt{x_1^2 + x_2^2}$ instead of the bare coordinates $x_1$ and $x_2$. Modern machine learning algorithms like deep neural networks perform feature engineering by themselves. However, selecting meaningful features beforehand is almost always beneficial as it saves computational resources and avoids time-consuming automated feature detection. Moreover, meaningful features allow a more direct interpretation of the obtained model.
Naturally, the process of feature construction and selection is also performed by a dimensionality reduction method. Here, a *more efficient* representation of the data is sought, which contains the most important information. Thus, the reduced low-dimensional coordinates can be seen as the features that have been automatically generated from the input data.

A *feature map* $\phi : \Omega \to \tilde{\Omega}$ assigning the input data $\boldsymbol{x}_i$ to some meaningful features $\phi(\boldsymbol{x}_i)$ in the so-called *feature space* $\tilde{\Omega}$ is a major component of several machine learning algorithms, e.g., of support vector machines; see Chapter 3.

## 1.6 ▪ Effects in high dimensions

Our intuition of distances and locality breaks down in high dimensions due to the *concentration of measure* principle [Led01] or the so-called *curse of dimensionality* [Bel61]. Let us briefly explore two examples of these effects.

**Concentration of measure**

The ***concentration of measure*** effect describes the fact that high-dimensional random, independent vectors concentrate in Euclidean space, i.e., the probability of them residing in certain regions becomes very large. To illustrate this, consider $n$ uniformly distributed points $\boldsymbol{x}_i$, $i = 1, \dots, n$, in the unit ball

$$B_1(\mathbf{0}) := \left\{ \boldsymbol{x} \in \mathbb{R}^d \;\middle|\; \|\boldsymbol{x}\|_2 \leq 1 \right\}$$

as training data set $\mathcal{D}$. If we look at their nearest point $\boldsymbol{c}$ to the origin, i.e., $\boldsymbol{c} := \arg\min_{\boldsymbol{x}_i, i=,1\dots,n} \|\boldsymbol{x}_i\|_2$, then the median of the norm of $\boldsymbol{c}$ over all different realizations of the $n$-element training data set $\mathcal{D}$ fulfills

$$M(d, n) := \text{median}\left(\|\boldsymbol{c}\|_2\right) = \left(1 - \left(\frac{1}{2}\right)^{\frac{1}{n}}\right)^{\frac{1}{d}};$$

see also [HTF09]. Thus, we have, for example, for $d = 10$ and $n = 500$ that $M(d, n) \approx 0.52$. This means that already in 10 dimensions and for a small number

of data points, the point $\boldsymbol{c}$, which is closest to $\mathbf{0}$, is indeed closer to the boundary of the unit ball than to its origin $\mathbf{0}$. Thus, uniformly distributed data in the unit ball in Euclidean space tend to *concentrate* along the boundary for large $d$. Therefore, in high dimensions and for the Euclidean distance setting, data tend to behave counter-intuitively.

To the right, we give a schematic illustration of the concentration of measure effect in high dimensions in a projected 2D plot (blue = center point, red = data points). The red data points $\boldsymbol{x}_i$, $i = , 1, \ldots, n$, tend to concentrate close to the boundary of $B_1(\mathbf{0})$.

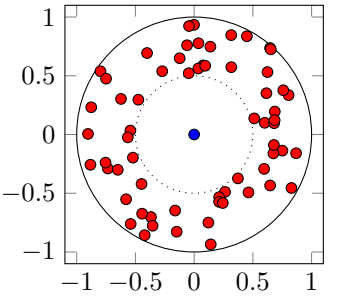



Figure 1.4

The concentration of measure effect shows us that the Euclidean distance does not seem to be intuitive or even meaningful anymore when relying on point to point distances in high-dimensional spaces. This is the reason why other choices of distance measures are often considered in such applications. Here, various $\ell_p$ norms, weighted Gaussian distances, or even divergences are reasonable alternatives in certain cases [AHK01, SFSW12]. However, recall that such deviations from the Euclidean distance often result in non-convex loss functions and/or non-unique minima, which make the ML minimization problem more complicated.

In the ML community, the terms concentration of measure and curse of dimensionality are often used ambiguously. From a mathematical perspective, the curse of dimensionality is more about properties on the necessary sample size.

**Curse of dimensionality**

The classical ***curse of dimensionality*** effect resembles the fact that we need exponentially (in the domain dimension $d$) many points to have a sampling that is as dense as in one dimension. Such a sampling is a necessary foundation for learning an arbitrary function equally well in $d$ dimensions. This is a famous problem in approximation theory [NW08].

To the right, we see an illustration of the curse of dimensionality effect (blue = 1D grid; red = 2D grid). To approximate a univariate (differentiable [NW08]) function up to a fixed accuracy, an interpolant on the $N = 10$ blue sample nodes can be built. For a bivariate function the $N^2 = 100$ red points are needed to (asymptotically) achieve the same accuracy. In $d$ dimensions we would need $N^d = 10^d$ many points. Thus, the computational costs scale exponentially in $d$.

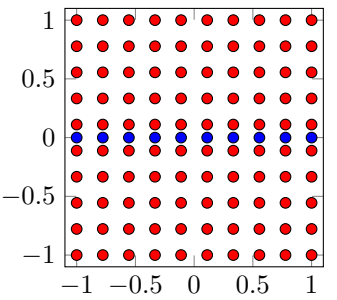



Figure 1.5

## 1.7 ▪ Process model for machine learning

Up to now, we have presented foundational mathematical concepts which are relevant in the machine learning context. But since we also discuss specific implementations of ML algorithms in this book, process models and computational aspects are important as well. To this end, let us now consider these topics in detail.

Usually the process of applying data mining and machine learning algorithms to a problem at hand involves the same steps independently of the application. Therefore, an open standard process model has been introduced, which describes how an ML problem is treated.

**Cross-industry standard process for data mining (CRISP-DM)**

The ***cross-industry standard process*** for data mining (CRISP-DM) [She00] has been developed by leading machine learning experts of different companies in the context of a European union project. As of today, the CRISP-DM model is the most commonly used process model when tackling machine learning tasks because of its interdisciplinarity and its generality. A schematic overview of the so-called *CRISP-DM cycle* is given in Figure 1.6.

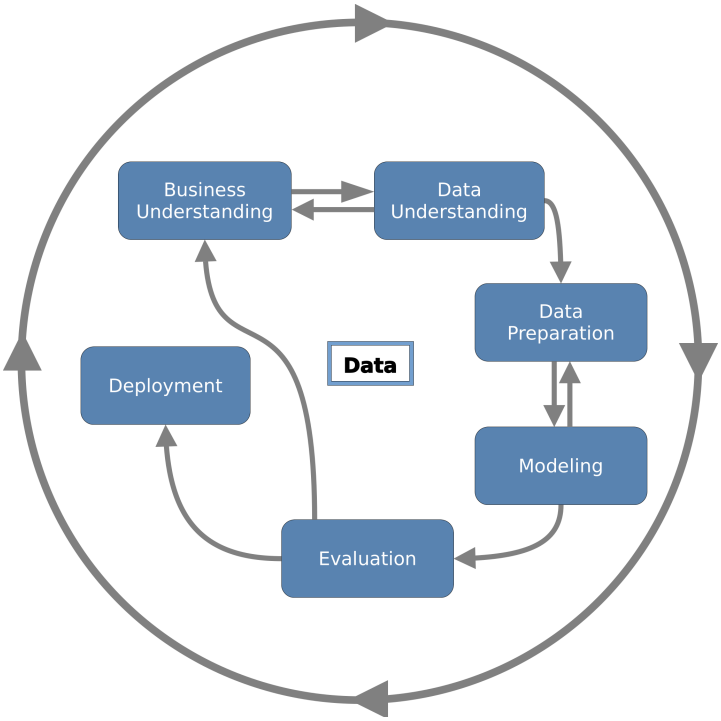



Figure 1.6: The schematic CRISP-DM cycle.

The CRISP-DM cycle consists of six blocks in the following order:

- Business understanding: The situation is assessed, and the goals for the application of machine learning tools are defined in terms of business objectives and success criteria.
- Data understanding: Data are collected from various sources and merged into a single data set. An initial data exploration is performed.
- Data preparation: The data set is cleaned (e.g., outliers are removed, and noise is reduced) and formatted. Irrelevant data points are eliminated.

- Modeling: Appropriate machine learning models are built, and hyperparameters are determined.
- Evaluation: The results of the ML model on the data set are gathered and evaluated according to the chosen success criteria.
- Deployment: Monitoring and maintenance methods for the whole data pipeline are employed to enable a continuous process. Reports are created.

While some of these topics are closely intertwined, and while there are dependencies between earlier and later phases, the overall process follows the aforementioned order. Furthermore, the cyclic nature of the CRISP-DM model illustrates the fact that the data mining process does not end after the deployment phase, but new insights gained from the later phases usually lead to a new understanding of the problem at hand and may trigger the whole process again.

While CRISP-DM is still the most often adopted process model for treating machine learning problems, there also exist other models such as *SEMMA* [RM10], which is tailored towards the usage of SAS software, or *Six Sigma*,[2] which is a more general model for data-based business processes. More recently, modern and agile frameworks such as the *Team Data Science Process* (TDSP)[3] and the *VDI/VDE guideline 3714*[4] are being employed in industrial data analysis.

## 1.8 ▪ Implementation of machine learning algorithms

Besides the purely theoretical side of discussing the inner workings and properties of machine learning algorithms, a successful ML practitioner also knows how to use codes and implementations of these algorithms to tackle given problems. Throughout this book, our task-boxes aim to encourage readers to both implement certain ML methods on their own and familiarize themselves with ready-to-use machine learning libraries and packages.

While there are many possible choices for adequate programming languages to use when tackling ML problems, e.g., C/C++, R, JULIA, SCALA, or JAVA, the most popular one to this day is PYTHON.[5] It nicely combines the possibility to quickly try out new ideas (*prototyping*) with fast numerical computations (*efficiency*) when using the NUMPY[6] library. Furthermore, it provides many well-established machine learning libraries for various kinds of purposes, and its basic syntax is quite easy to learn, even for beginners without any programming knowledge. Therefore, we also decided to use PYTHON as the programming language for the tasks and templates in this book.

When using specific machine learning algorithms we will mainly work with the SCIKIT-LEARN[7] and KERAS[8] libraries. Those libraries provide directly applicable implementations of many famous machine learning methods, and the initial hurdle that one has to overcome to use these libraries is quite small. However, we want to emphasize that the use of other tools and libraries might be more appropriate when implementing highly flexible machine learning models or when aiming for highly optimized production code. Here, libraries like PYTORCH[9] and TENSORFLOW[10] for production-grade deep learning models or JAX[11] for highly efficient numerical operations are common software platforms of choice.

[2] https://www.iso.org/standard/52901.html
[3] https://learn.microsoft.com/en-us/azure/architecture/data-science-process/overview
[4] https://www.vdi.de/richtlinien/details/vdivde-3714-blatt-1-implementierung-und-betrieb-von-big-data-anwendungen-in-der-produzierenden-industrie-durchfuehrung-von-big-data-projekten
[5] https://www.python.org/
[6] https://www.numpy.org/
[7] https://scikit-learn.org/
[8] https://keras.io/
[9] https://pytorch.org/
[10] https://www.tensorflow.org/
[11] https://jax.readthedocs.io

To dive directly into the basics of PYTHON and NUMPY, let us begin with the first tasks of this book. They serve to familiarize the reader with the coding environment as well as the programming language.

**Task 1.1.** Make yourself familiar with programming in PYTHON and its libraries NUMPY and MATPLOTLIB[12]. Furthermore, you will need the application JUPYTER NOTEBOOK[13] to run the template codes at https://bookstore.siam.org/di03/bonus. To this end, the "README.md" file provides a guideline on setting up the programming environment by hand. Furthermore, we encourage you to have a look at the tutorial notebooks "Introduction to Python.ipynb" and "Introduction to NumPy.ipynb" and to familiarize yourself with the concept of vectorization, i.e., using operations on whole arrays instead of using loops and operating on single array elements.

**Task 1.2.** Create a JUPYTER notebook in which you construct an array $z$ consisting of $10000$ random numbers drawn from $\{0, 1, 2\}$. Implement two versions of a function that counts the number of appearances of the subsequence $(2, 0, 1)$ in $z$. The first version should work with a loop that accesses the array $z$ elementwise and makes elementwise comparisons. The second version should be a vectorized one, which operates on (almost) the whole array $z$. (Hint: The NUMPY function `logical_and` might help you.) Compare the runtime of the two versions.

## 1.9 ▪ Further references on machine learning topics

In this section, we provide references to other textbooks that give an overview of various machine learning topics. In contrast to our intention of providing solid mathematical background together with first hands-on coding experience for a broad range of machine learning methods, these textbooks have a different perspective on the addressed topics.

In [Bis06, HTF09, JWHT21, Mur22, Mur23] the interested reader can find a good overview of the most important machine learning topics from a statistical point of view. These books provide an intuitive and conceptual approach to the presented methods. For an introduction to the main mathematical concepts commonly used in data analysis and machine learning, see [DFO20, Phi21, Str19]. A special overview on the specific topic of neural networks and their applications can be found in [GBC16]. This book provides a computer scientist's perspective on the state of the art of deep learning and the corresponding algorithms. An introduction to implementing these deep learning algorithms in PYTHON can be found in [Cho17, Gér19]. A more mathematical point of view on neural networks is taken in [Cal20]. Here, the mathematical background behind the inner workings of deep learning algorithms is highlighted. An intuitive overview on many important dimensionality reduction methods can be found in [LV07]. The authors use the same benchmark data sets for all algorithms to make their comparison more natural. [WR22] presents a thorough discussion of optimization problems, which appear for many kinds of tasks in data science. The authors introduce the most common optimization algorithms used in machine learning and provide a mathematical analysis thereof. Finally, [CZ07] and [Vap99] provide a mathematical foundation for specific types of machine learning algorithms, namely so-called *kernel* algorithms, which we will encounter in Chapter 3.

[12] https://matplotlib.org/
[13] https://jupyter.org/

## Chapter 2

# Basic Supervised Learning Algorithms

In this chapter we will introduce specific instances of machine learning algorithms, and we will see how to evaluate them on given test data. Our first approach, which is probably the most simple one, namely linear regression, is discussed in Section 2.1. Here, the model class, over which the learning problem is solved, consists of (affine) linear functions. This leads to an easy numerical treatment of the underlying optimization problem. Section 2.2 is dedicated to the introduction of different measures that serve to quantify how well a regression (or classification) algorithm performs. Subsequently, we encounter our first programming tasks in Section 2.3. In addition to the direct solvers, which we discuss in Section 2.1, iterative solvers can be employed to tackle the linear regression problem. They are studied in Section 2.4. Section 2.5 illustrates the importance of data normalization, especially for iterative solvers. Finally, we will consider the *$k$-nearest neighbors* algorithm in Section 2.6, which provides another intuitive way to solve a supervised learning task. More details on linear regression and the $k$-nearest neighbors method can be found in [HTF09, Mur22].

## 2.1 ▪ Linear least squares

We first address the problem of linear least squares regression, which we briefly introduced in the last chapter. To this end, recall that we aim to minimize the least squares loss

$$\mathcal{L}_{\text{ls}}\left(\left(z_1, \tilde{z}_1\right), \ldots, \left(z_n, \tilde{z}_n\right)\right) = \frac{1}{n}\sum_{i=1}^{n}(z_i - \tilde{z}_i)^2$$

over the parametrized model class of affine linear functions

$$\mathcal{M}_{\text{Lin}} = \left\{ g : \mathbb{R}^d \to \mathbb{R} \;\middle|\; g(\boldsymbol{t}) = \gamma_0 + \sum_{i=1}^{d} \gamma_i t_i \text{ for some fixed } \gamma_0, \ldots, \gamma_d \in \mathbb{R} \right\}.$$

Therefore, the *linear least squares* (LLS) problem reads

$$\begin{aligned} &\text{Find } f := \operatorname*{arg\,min}_{g \in \mathcal{M}_{\text{Lin}}} \mathcal{L}_{\text{ls}}\left(\left(g(\boldsymbol{x}_1), y_1, \right), \ldots, \left(g(\boldsymbol{x}_n), y_n\right)\right) \\ \Leftrightarrow \quad &\operatorname*{arg\,min}_{\gamma_0, \ldots, \gamma_d \in \mathbb{R}} \frac{1}{n}\sum_{i=1}^{n}\left(\boldsymbol{\gamma}^T \cdot \hat{\boldsymbol{x}}_i - y_i\right)^2 \end{aligned}$$

with parameter vector $\boldsymbol{\gamma} = (\gamma_0, \ldots, \gamma_d)^T \in \mathbb{R}^{d+1}$. Here, $\hat{\boldsymbol{x}}_i := (1, [\boldsymbol{x}_i]_1, \ldots, [\boldsymbol{x}_i]_d)^T \in \mathbb{R}^{d+1}$ is the augmented vector based on a data point $\boldsymbol{x}_i \in \mathbb{R}^d$ for all $i = 1, \ldots, n$, and $[\cdot]_j$ denotes the

$j$th component of a vector. The above expression for the LLS problem can be rewritten as

$$\boldsymbol{\alpha} := \underset{\boldsymbol{\gamma}\in\mathbb{R}^{d+1}}{\arg\min} \frac{1}{n} \left(\boldsymbol{X}\boldsymbol{\gamma} - \boldsymbol{y}\right)^T \left(\boldsymbol{X}\boldsymbol{\gamma} - \boldsymbol{y}\right) = \underset{\boldsymbol{\gamma}\in\mathbb{R}^{d+1}}{\arg\min} \boldsymbol{\gamma}^T \boldsymbol{X}^T \boldsymbol{X}\boldsymbol{\gamma} - 2\boldsymbol{\gamma}^T \boldsymbol{X}^T \boldsymbol{y} + \boldsymbol{y}^T \boldsymbol{y} \quad (2.1)$$

with $\boldsymbol{X} := \left(\hat{\boldsymbol{x}}_1, \hat{\boldsymbol{x}}_2, \ldots, \hat{\boldsymbol{x}}_n\right)^T \in \mathbb{R}^{n\times(d+1)}$ and $\boldsymbol{y} := \left(y_1, \ldots, y_n\right)^T \in \mathbb{R}^n$.

### 2.1.1 ▪ Normal equations

Note that (2.1) is a quadratic and, thus, a convex minimization problem in the parameters $\boldsymbol{\gamma}$. Therefore, a critical point is automatically a global minimizer since our model class is also convex. Next let us check for critical points, i.e., let us set the derivatives $\frac{\partial}{\partial\gamma_i}$ to 0 for all $i = 0, \ldots, d$ to obtain a global minimizer $\boldsymbol{\alpha}$. This results in

$$\boldsymbol{X}^T \boldsymbol{X}\boldsymbol{\alpha} = \boldsymbol{X}^T \boldsymbol{y}, \quad (2.2)$$

i.e., a minimizer $\boldsymbol{\alpha}$ of (2.1) fulfills this equation and vice versa. This system of linear equations is also known as the *normal equations* of the least squares problem.

Furthermore, we note that the minimizer of (2.1) is unique if $\boldsymbol{X}^T\boldsymbol{X}$ has full rank, i.e., if it is an invertible $(d+1)\times(d+1)$ matrix. In this case we have

$$\boldsymbol{X}^T \boldsymbol{X}\boldsymbol{\alpha} = \boldsymbol{X}^T \boldsymbol{y} \quad \Leftrightarrow \quad \boldsymbol{\alpha} = \left(\boldsymbol{X}^T \boldsymbol{X}\right)^{-1} \boldsymbol{X}^T \boldsymbol{y}.$$

Thus, to tackle the LLS regression problem, we need to solve a system of linear equations. Moreover, if $\boldsymbol{X}^T\boldsymbol{X}$ happens to have a non-trivial null space and is, therefore, not invertible, we can still look for solutions of (2.2) to find a minimizer, but a solution is no longer unique.

**Linear system solvers: LU and Cholesky decomposition**

One important topic in numerical mathematics is the development and analysis of algorithms to solve systems of linear equations $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ with a regular matrix $\boldsymbol{A} \in \mathbb{R}^{\tilde{d}\times\tilde{d}}$ and vectors $\boldsymbol{x}, \boldsymbol{b} \in \mathbb{R}^{\tilde{d}}$ for arbitrary $\tilde{d} \in \mathbb{N}$. The most straightforward algorithm to solve such a system would be ***LU decomposition***. In this case, a decomposition $\boldsymbol{A} = \boldsymbol{L}\boldsymbol{U}$ into a lower and an upper triangular matrix is computed via Gaussian elimination, which can be done in $\mathcal{O}\left(\tilde{d}^3\right)$ floating point operations. Then,

$$\boldsymbol{b} = \boldsymbol{A}\boldsymbol{x} = \boldsymbol{L}\boldsymbol{U}\boldsymbol{x} \;\Leftrightarrow\; \boldsymbol{U}^{-1}\boldsymbol{L}^{-1}\boldsymbol{b} = \boldsymbol{x}.$$

Note that the application of the inverses of $\boldsymbol{U}$ and $\boldsymbol{L}$ can be easily computed by forward and backward substitution, i.e., by successively determining each entry of the resulting vector. In this way, a complicated, direct matrix inversion can be avoided.
If $\boldsymbol{A}$ is symmetric and positive definite, there exists a unique so-called ***Cholesky*** decomposition $\boldsymbol{A} = \boldsymbol{L}\boldsymbol{L}^T$ for a lower-triangular matrix $\boldsymbol{L}$, which can be computed via a modified Gaussian elimination. The computation of a Cholesky decomposition is twice as fast as the one of the LU decomposition. Furthermore, it is stable with respect to small input changes, which cannot be guaranteed for the LU decomposition unless pivoting is used. For more details on computing the solution of a system of linear equations, we refer the reader to [GVL13].

### 2.1.2 ▪ Pseudo-inverse solutions

To use a Cholesky decomposition or an LU decomposition to compute a solution of the normal equations (2.2), we would first need to assemble the matrix $\boldsymbol{A} := \boldsymbol{X}^T\boldsymbol{X} \in \mathbb{R}^{(d+1)\times(d+1)}$, which needs $\mathcal{O}(d^2 n)$ floating point operations. However, avoiding the computation of the matrix $\boldsymbol{X}^T\boldsymbol{X}$ is preferable when solving (2.1). The reason for this lies in the *condition number* of this matrix, which we will explain soon. Let us first explore another way to numerically solve (2.1) using the *singular value decomposition* (SVD).

**Singular value decomposition (SVD)**

The decomposition of a general matrix $\boldsymbol{A} \in \mathbb{R}^{n\times\tilde{d}}$ into

$$\boldsymbol{A} = \boldsymbol{U}\boldsymbol{D}\boldsymbol{W}^T$$

with orthogonal matrices $\boldsymbol{U} \in \mathbb{R}^{n\times n}$ and $\boldsymbol{W} \in \mathbb{R}^{\tilde{d}\times\tilde{d}}$ and the diagonal matrix $\boldsymbol{D} = \operatorname{diag}(\sigma_1, \ldots, \sigma_{\min(n,\tilde{d})}) \in \mathbb{R}^{n\times\tilde{d}}$ of non-negative ***singular values*** is called singular value decomposition. Here, the singular values are non-negative and sorted according to their size, i.e., $\sigma_1 \geq \cdots \geq \sigma_{\min(n,\tilde{d})} \geq 0$. The singular values of $\boldsymbol{A}$ are uniquely determined. The SVD can be computed with $\mathcal{O}(\min(\tilde{d}, n)^2 \max(\tilde{d}, n))$ floating point operations. For details on the SVD, we refer the reader to [GVL13].

Besides the SVD, we will also need the *pseudo-inverse* of a matrix in the following.

**Moore–Penrose pseudo-inverse**

The (Moore–Penrose) ***pseudo-inverse*** of $\boldsymbol{A} \in \mathbb{R}^{n\times\tilde{d}}$ with SVD $\boldsymbol{A} = \boldsymbol{U}\boldsymbol{D}\boldsymbol{W}^T$ is a generalization of the inverse of the matrix $\boldsymbol{A}$. It is given by

$$\boldsymbol{A}^\dagger := \boldsymbol{W}\boldsymbol{D}^\dagger\boldsymbol{U}^T$$

with a matrix $\boldsymbol{D}^\dagger \in \mathbb{R}^{\tilde{d}\times n}$ that fulfills

$$\boldsymbol{D}^\dagger_{ij} := \begin{cases} \frac{1}{\sigma_i} & \text{if } i = j \text{ and } \sigma_i > 0, \\ 0 & \text{else.} \end{cases}$$

More generally, the pseudo-inverse is uniquely defined by the four properties

$$\begin{aligned} \boldsymbol{A}\boldsymbol{A}^\dagger\boldsymbol{A} &= \boldsymbol{A}, & \boldsymbol{A}^\dagger\boldsymbol{A}\boldsymbol{A}^\dagger &= \boldsymbol{A}^\dagger, \\ \left(\boldsymbol{A}\boldsymbol{A}^\dagger\right)^T &= \boldsymbol{A}\boldsymbol{A}^\dagger, & \left(\boldsymbol{A}^\dagger\boldsymbol{A}\right)^T &= \boldsymbol{A}^\dagger\boldsymbol{A}. \end{aligned}$$

If $\boldsymbol{A}$ is an invertible square matrix, the pseudo-inverse and the inverse of $\boldsymbol{A}$ coincide. More details on the pseudo-inverse can be found in, e.g., [GVL13] and [Hig02].

In the case where $\boldsymbol{X}^T\boldsymbol{X}$ is a full-rank matrix, the pseudo-inverse of $\boldsymbol{X}$ fulfills

$$\boldsymbol{X}^\dagger = \left(\boldsymbol{X}^T\boldsymbol{X}\right)^{-1}\boldsymbol{X}^T.$$

Therefore, the solution to the normal equations (2.2) is just

$$\boldsymbol{\alpha} = \boldsymbol{X}^\dagger\boldsymbol{y}, \tag{2.3}$$

which can be computed by using an SVD $\boldsymbol{X} = \boldsymbol{U}\boldsymbol{D}\boldsymbol{W}^T$ and employing the pseudo-inverse of $\boldsymbol{D}$ instead of inverting $\boldsymbol{X}^T\boldsymbol{X}$. However, we can use (2.3) not only in the case of full-rank $\boldsymbol{X}^T\boldsymbol{X}$ but also in the general case to solve (2.1).

Let us explore why (2.3) is a solution to (2.1). Note that (2.1) essentially reads

$$\boldsymbol{\alpha} = \underset{\boldsymbol{\gamma}\in\mathbb{R}^{d+1}}{\arg\min} \frac{1}{n}\|\boldsymbol{X}\boldsymbol{\gamma} - \boldsymbol{y}\|_2^2.$$

Therefore, we are looking for a $\boldsymbol{\gamma}$ such that its image under $\boldsymbol{X}$ is closest to $\boldsymbol{y}$ in the Euclidean norm. We know from basic linear algebra that this is fulfilled for a $\boldsymbol{\gamma}$ such that $\boldsymbol{X}\boldsymbol{\gamma}$ is an orthogonal projection of $\boldsymbol{y}$ onto the image space of $\boldsymbol{X}$. To this end, let $\boldsymbol{C} \in \mathbb{R}^{(d+1)\times n}$ be a matrix such that

$$(\boldsymbol{X}\boldsymbol{C})^2 = \boldsymbol{X}\boldsymbol{C} = (\boldsymbol{X}\boldsymbol{C})^T,$$

i.e., $\boldsymbol{X}\boldsymbol{C}$ is an orthogonal projector onto the image of $\boldsymbol{X}$. Then, $\boldsymbol{\gamma} := \boldsymbol{C}\boldsymbol{y}$ is already a solution to (2.1) since

$$\boldsymbol{X}\boldsymbol{\gamma} = \boldsymbol{X}\boldsymbol{C}\boldsymbol{y}$$

is an orthogonal projection of $\boldsymbol{y}$ onto the image of $\boldsymbol{X}$. Finally, to prove that (2.3) is a solution to (2.1), we have to show that $\boldsymbol{X}^\dagger$ is a valid candidate for such a matrix $\boldsymbol{C}$. This is straightforward since

$$\left(\boldsymbol{X}\boldsymbol{X}^\dagger\right)^2 = \left(\boldsymbol{U}\boldsymbol{D}\boldsymbol{W}^T\boldsymbol{W}\boldsymbol{D}^\dagger\boldsymbol{U}^T\right)^2 = \left(\boldsymbol{U}\boldsymbol{D}\boldsymbol{D}^\dagger\boldsymbol{U}^T\right)^2 = \boldsymbol{U}\boldsymbol{D}\boldsymbol{D}^\dagger\boldsymbol{D}\boldsymbol{D}^\dagger\boldsymbol{U}^T = \boldsymbol{X}\boldsymbol{X}^\dagger$$

due to the orthogonality of $\boldsymbol{U}$ and $\boldsymbol{W}$ and because $\boldsymbol{D}\boldsymbol{D}^\dagger$ is idempotent. The fact that $(\boldsymbol{X}\boldsymbol{X}^\dagger)^T = \boldsymbol{X}\boldsymbol{X}^\dagger$ is one of the four defining properties of the pseudo-inverse. Therefore, $\boldsymbol{C} = \boldsymbol{X}^\dagger$ is a valid choice, and (2.3) solves (2.1).

### 2.1.3 ▪ Stability of the problem

We have seen that we can solve (2.1) by using either a Cholesky or an LU decomposition to solve the normal equations (2.2) or by using the SVD to compute (2.3). Although employing the SVD is usually more expensive with respect to the number of floating point operations being used (see, e.g., [GVL13], [Hig02], [PTVF07]), it is the preferable way to solve (2.1) in general. The reason for this is that we avoid the explicit use of the assembled matrix $\boldsymbol{X}^T\boldsymbol{X}$, whose condition number $\kappa(\boldsymbol{X}^T\boldsymbol{X})$ is larger than the condition number $\kappa(\boldsymbol{X})$ of $\boldsymbol{X}$.

**Matrix condition numbers**

The ***condition number*** of a non-singular matrix $\boldsymbol{A}$ is given as the ratio of its largest and its smallest singular values

$$\kappa(\boldsymbol{A}) := \frac{\sigma_{\max}(\boldsymbol{A})}{\sigma_{\min}(\boldsymbol{A})}.$$

The condition number is a measure for how sensitive the solution $\boldsymbol{x}$ of $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ is to changes in the right-hand side $\boldsymbol{b}$. Therefore, larger condition numbers introduce potentially larger instabilities when solving a system of linear equations. In particular, let $\boldsymbol{b} + \Delta_{\boldsymbol{b}}$ be an altered right-hand side term. We write the corresponding solution as $\boldsymbol{x} + \Delta_{\boldsymbol{x}}$ for some perturbation $\Delta_{\boldsymbol{x}}$ to $\boldsymbol{x}$, i.e., we have the two systems of equations

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b} \quad \text{and} \quad \boldsymbol{A}(\boldsymbol{x} + \Delta_{\boldsymbol{x}}) = (\boldsymbol{b} + \Delta_{\boldsymbol{b}}).$$

Then it holds that

$$\frac{\|\Delta_{\boldsymbol{x}}\|_2}{\|\boldsymbol{x}\|_2} \leq \kappa(\boldsymbol{A}) \frac{\|\Delta_{\boldsymbol{b}}\|_2}{\|\boldsymbol{b}\|_2},$$

i.e., the relative change in the solution is bounded by the condition number multiplied by the relative change in the right-hand side. For more details on condition numbers see [Hig02].

Because of its definition, the best condition number we could hope for when solving a system $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ is $\kappa(\boldsymbol{A}) = 1$. In this case, all singular values would be equal to 1. In general, we are aiming for a condition number that is small, i.e., close to 1.

In the least squares setting the system matrix is usually determined by the drawn samples and is thus fixed. However, while the SVD can be computed by matrix-vector multiplications with $\boldsymbol{X}$ (see [GVL13] for details), the normal equations (2.2) involve the system matrix $\boldsymbol{X}^T\boldsymbol{X}$. Assuming that the SVD of $\boldsymbol{X}$ is given by $\boldsymbol{X} = \boldsymbol{U}\boldsymbol{D}\boldsymbol{W}^T$, the system matrix is given by

$$\boldsymbol{X}^T\boldsymbol{X} = \boldsymbol{W}\boldsymbol{D}^T\boldsymbol{U}^T\boldsymbol{U}\boldsymbol{D}\boldsymbol{W}^T = \boldsymbol{W}\boldsymbol{D}^T\boldsymbol{D}\boldsymbol{W}^T.$$

Since $\boldsymbol{W}$ is an orthogonal matrix, the decomposition on the right-hand side is already the SVD of $\boldsymbol{X}^T\boldsymbol{X}$. Therefore, its singular values are given as the diagonal values of $\boldsymbol{D}^T\boldsymbol{D}$, which are the squared singular values of $\boldsymbol{X}$. This leads to the fact that

$$\kappa(\boldsymbol{X}^T\boldsymbol{X}) = \kappa(\boldsymbol{X})^2,$$

which is the reason why it is beneficial in terms of numerical stability to use (2.3) instead of solving (2.2).

Besides employing the SVD to compute the pseudo-inverse in (2.3), using a *QR decomposition* of $\boldsymbol{X}$ is another way to solve (2.1) without having to deal with the normal equations (2.2); see [GVL13] for details.

## 2.2 ▪ Evaluation

After we trained the LLS model, i.e., after we determined the parameters $\boldsymbol{\alpha}$ from (2.1), we want to study how good the model actually is.

**Evaluation of ML algorithms**

To assess the quality of an already trained ML model, we need a way to quantify its performance. Usually, we do not have direct access to the measure $\mu$ from which our training data was generated, but we assume that we can sample a test data set

$$\bar{\mathcal{D}} := \{(\bar{\boldsymbol{x}}_i, \bar{y}_i) \in \Omega \times \Gamma \mid i = 1, \ldots, \bar{n}\}$$

i.i.d. from $\mu$. Then, we *approximately* quantify how small the ***test error*** of the model is, i.e., the error that we get when taking $\boldsymbol{\alpha}$—or equivalently $f(\boldsymbol{t}) = \alpha_0 + \sum_{i=1}^{d} \alpha_i t_i$—to predict the outputs/labels of the test data points. There are many possible criteria to measure the error on the test data but usually the employed loss function is taken (without all additional regularization terms).
In cases where not enough test data points are available to compute a representative test error, (re)sampling techniques like *bootstrapping* can be employed; see

Section 10.2.4 for details. Of course, it is of major importance that the test data are not part of the training data in order to avoid an evaluation bias due to overfitting. Finally, note that the assumption of the test data following the same distribution as the training data does not necessarily hold in practical applications. For instance, a *covariate shift* can occur in the data set, i.e., the measure $\mu$ has changed during or after training; see, e.g., [SK12].

**Evaluation measures for regression** Since we have chosen a least squares loss to train the model, it is only natural to evaluate the *mean squared error* (MSE) on the test data set $\bar{\mathcal{D}}$, i.e.,

$$\mathrm{MSE}\left(f, \bar{\mathcal{D}}\right) := \frac{1}{\bar{n}} \sum_{i=1}^{\bar{n}} \left(f\left(\bar{\boldsymbol{x}}_i\right) - \bar{y}_i\right)^2 .$$

Other commonly used error measures include the *mean absolute error*

$$\mathrm{MAE}\left(f, \bar{\mathcal{D}}\right) := \frac{1}{\bar{n}} \sum_{i=1}^{\bar{n}} \left|f\left(\bar{\boldsymbol{x}}_i\right) - \bar{y}_i\right| ,$$

the *relative absolute error*

$$\mathrm{RAE}\left(f, \bar{\mathcal{D}}\right) := \frac{\sum_{i=1}^{\bar{n}} \left|f\left(\bar{\boldsymbol{x}}_i\right) - \bar{y}_i\right|}{\sum_{i=1}^{\bar{n}} \left|\frac{1}{\bar{n}} \sum_{j=1}^{\bar{n}} \bar{y}_j - \bar{y}_i\right|},$$

and the *coefficient of determination*

$$R^2\left(f, \bar{\mathcal{D}}\right) := 1 - \frac{\sum_{i=1}^{\bar{n}} \left(f\left(\bar{\boldsymbol{x}}_i\right) - \bar{y}_i\right)^2}{\sum_{i=1}^{\bar{n}} \left(\frac{1}{\bar{n}} \sum_{j=1}^{\bar{n}} \bar{y}_j - \bar{y}_i\right)^2},$$

which is close to 1 for a good model and close to 0 (or even negative) for a poor one.

**Evaluation measures for classification** Since $|\Gamma| < \infty$ for classification, it only makes sense to consider functions $f : \Omega \to \Gamma$ with finite image set. An easy way to obtain an appropriate classifier $f : \Omega \to \Gamma$ from a regression function $\tilde{f} : \Omega \to \mathbb{R}$ is to use a so-called *level set* classifier. For example, let us consider a two-class problem with $\Gamma = \{a, b\} \subset \mathbb{R}$. Then we use

$$f(\boldsymbol{x}) := f_{\text{class}}(\boldsymbol{x}) := \begin{cases} a & \text{if } |\tilde{f}(\boldsymbol{x}) - a| < |\tilde{f}(\boldsymbol{x}) - b|, \\ b & \text{else.} \end{cases} \tag{2.4}$$

From now on, we implicitly assume that the function under consideration has image $\Gamma$ when we use the evaluation measures for classification below. However, by slightly abusing notation, we can also use them for a real-valued function $\tilde{f} : \Omega \to \mathbb{R}$ when we assume that the level set classifier (2.4) is employed instead of $\tilde{f}$ in that case.

The most common error measure for classification is the so-called *accuracy*

$$\mathrm{Acc}\left(f, \bar{\mathcal{D}}\right) := \frac{\left|\left\{i \in \{1, \ldots, \bar{n}\} \mid f\left(\bar{\boldsymbol{x}}_i\right) = \bar{y}_i\right\}\right|}{\bar{n}}, \tag{2.5}$$

which measures the proportion of correct classifications among all test data. Further error measures in binary classification tasks are the sensitivity and the precision, for instance. To this end,

let us briefly introduce the concept of true/false positives and negatives for a two-class problem, where $\Gamma = \{-1, 1\}$. After training a model $f$, the numbers of true positives ($TP$) and true negatives ($TN$) on the test data set $\bar{\mathcal{D}}$ are given by

$$TP := TP(f, \bar{\mathcal{D}}) := \left|\{i \in \{1, \dots, \bar{n}\} \mid f(\bar{\boldsymbol{x}}_i) = \bar{y}_i = 1\}\right|,$$
$$TN := TN(f, \bar{\mathcal{D}}) := \left|\{i \in \{1, \dots, \bar{n}\} \mid f(\bar{\boldsymbol{x}}_i) = \bar{y}_i = -1\}\right|.$$

Analogously, the numbers of false positives ($FP$) and false negatives ($FN$) are defined as

$$FP := FP(f, \bar{\mathcal{D}}) := \left|\{i \in \{1, \dots, \bar{n}\} \mid f(\bar{\boldsymbol{x}}_i) = 1, \bar{y}_i = -1\}\right|,$$
$$FN := FN(f, \bar{\mathcal{D}}) := \left|\{i \in \{1, \dots, \bar{n}\} \mid f(\bar{\boldsymbol{x}}_i) = -1, \bar{y}_i = 1\}\right|.$$

With these definitions, we can define the *sensitivity* or *recall* as the rate of true positives

$$\text{Sens}(f, \bar{\mathcal{D}}) = \frac{TP}{TP + FN}.$$

Analogously, the *specificity* is defined as the rate of true negatives

$$\text{Spec}(f, \bar{\mathcal{D}}) = \frac{TN}{TN + FP}.$$

Another commonly used measure is the *precision*

$$\text{Prec}(f, \bar{\mathcal{D}}) = \frac{TP}{TP + FP},$$

which describes the ratio between the true positives and all predicted positives. Using these measures makes more sense than using the accuracy when it is important to minimize the number of false negatives (or the number of false positives), as in safety-critical applications for instance. Moreover, these measures are of particular relevance for imbalanced data, i.e., when the number of instances per class varies strongly.

More sophisticated measures, e.g., the $\text{F}_1$ *score*

$$\text{F}_1(f, \bar{\mathcal{D}}) = 2\,\frac{\text{Prec}(f, \bar{\mathcal{D}}) \cdot \text{Sens}(f, \bar{\mathcal{D}})}{\text{Prec}(f, \bar{\mathcal{D}}) + \text{Sens}(f, \bar{\mathcal{D}})},$$

can be derived from combinations of the sensitivity, specificity, and precision values.

A meaningful measure for multi-class applications, i.e., when $|\Gamma| > 2$, is the so-called *confusion matrix* $C \in \mathbb{N}^{|\Gamma| \times |\Gamma|}$, which displays the true and false positives on a class-by-class basis. In particular, it has the entries

$$C_{ij}(f, \bar{\mathcal{D}}) = \#\left\{\text{Points classified as } i \text{ by } f \text{ where the true label is } j\right\}.$$

This metric is especially helpful when certain classes get misclassified as one particular other class, for instance. The entries of the confusion matrix can further be used to compute more sophisticated correlation measures between two classes, e.g., the *Matthews correlation coefficient*

$$\text{MCC}(i, j) = \frac{C_{ii}C_{jj} - C_{ij}C_{ji}}{(C_{ii} + C_{ij})(C_{ji} + C_{jj})(C_{ii} + C_{ji})(C_{ij} + C_{jj})}$$

between classes $i$ and $j$. A generalization of the MCC including all classes at once can also be considered; see [JRF12] for details.

## 2.3 ▪ Tasks on linear least squares

Let us now begin implementing the linear least squares algorithm and testing it on different data sets. We start with a simple one-dimensional regression problem.

**Task 2.1.** First, draw $n = 100$ random numbers $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_{100} \in [0, 1)$, which are uniformly distributed. To this end, you can use the `numpy.random.rand` routine. Subsequently, compute

$$y_i := 1 + 2\boldsymbol{x}_i + 0.2\varepsilon_i \text{ with } \varepsilon_i \sim \mathcal{N}(0, 1)$$

for $i = 1, \ldots, 100$. Use `matplotlib` to generate a scatter plot of the 100 two-dimensional points $(\boldsymbol{x}_i, y_i)$ for $i = 1, \ldots, 100$ (i.e., plot the points in a 2D coordinate system).

**Task 2.2.** Implement a linear least squares algorithm using the SVD to compute (2.3). *Hint:* You can use `numpy.linalg.pinv` to compute the pseudo-inverse of a matrix using the SVD. Apply the algorithm to the data from Task 2.1. Plot the scattered input data as in Task 2.1 together with the regressor $f : \mathbb{R} \to \mathbb{R}$ given by

$$f(\boldsymbol{x}) := \alpha_0 + \alpha_1 \boldsymbol{x}.$$

Compare the resulting coefficients $\alpha_0$ and $\alpha_1$ to the ones you would expect for the data.

Next, let us consider a two-dimensional classification problem, where we use the level set function $f = f_{\text{class}}$ (see (2.4)) of a linear least squares regressor $\tilde{f}$ as a solution. To this end, we first create a suitable two-dimensional data set $\mathcal{D}$ in Task 2.3. Subsequently, we will train a linear least squares regressor $\tilde{f}$ on $\mathcal{D}$ and visualize the hyperplane which is defined by the corresponding level set in Task 2.4. In particular, for a two-class problem with $\Gamma = \{a, b\} \subset \mathbb{R}$, the hyperplane

$$H := \left\{ \boldsymbol{x} \in \mathbb{R}^2 \;\middle|\; |\tilde{f}(\boldsymbol{x}) - a| = |\tilde{f}(\boldsymbol{x}) - b| \right\}$$

splits the ambient space $\mathbb{R}^2$ into two parts, which indicate what class $f$ evaluates to.

In the specific use case from Task 2.3 and Task 2.4 the data has labels $\Gamma = \{0, 1\}$. Therefore, if $\tilde{f}(\boldsymbol{x}) < 0.5$, the point $\boldsymbol{x}$ is assigned to class 0. Otherwise, it is assigned to class 1, i.e.,

$$f(\boldsymbol{x}) := \begin{cases} 0 & \text{if } \tilde{f}(\boldsymbol{x}) < \frac{1}{2}, \\ 1 & \text{else.} \end{cases}$$

Because of the affine linear structure of $\tilde{f}$, this results in

$$H := \left\{ \boldsymbol{x} \in \mathbb{R}^2 \;\middle|\; \alpha_0 + \alpha_1 x_1 + \alpha_2 x_2 = \frac{1}{2} \right\}$$

with $\tilde{f}(\boldsymbol{x}) = \boldsymbol{\alpha}^T \hat{\boldsymbol{x}}$ and $\hat{\boldsymbol{x}} = (1, [\boldsymbol{x}]_1, \ldots, [\boldsymbol{x}]_d)^T$.

**Task 2.3.** Create $n = 200$ data points in the following way:

(a) Draw ten random i.i.d. samples from the two-variate normal distribution $\mathcal{N}\left((\frac{3}{2}\ 0)^T, \boldsymbol{I}\right)$ and store them in a `numpy` array `a`. Draw another ten samples according to $\mathcal{N}\left((0\ \frac{3}{2})^T, \boldsymbol{I}\right)$ and store them in another `numpy` array `b`. Use `matplotlib` to generate a scatter plot of the elements in `a` and the elements in `b` using different colors for the two arrays.

(b) Pick 100 equidistributed indices $i_1, \ldots, i_{100}$ from $\{1, 2, \ldots, 10\}$ and set the $j$th data point $\boldsymbol{x}_j$ to

$$\boldsymbol{x}_j := \underbrace{\mathtt{a}[i_j]}_{i_j\text{th element of a}} + \varepsilon_j \text{ for all } j = 1, \ldots, 100 \text{ with } \varepsilon_j \sim \mathcal{N}\left((0\ 0)^T, \frac{1}{4}\boldsymbol{I}\right).$$

Proceed analogously for $j = 101, \ldots, 200$ by substituting `a` by `b`. Generate a scatter plot for the data points $\boldsymbol{x}_j$ with $j = 1, \ldots, 200$ with different colors for the first 100 points and the second 100 points.

(c) The first $j = 1, \ldots, 100$ data points get assigned the label $\boldsymbol{y}_j = 0$; the next $j = 101, \ldots, 200$ ones get assigned the label $\boldsymbol{y}_j = 1$.

**Task 2.4.** Apply the LLS algorithm from Task 2.2 to the data from Task 2.3. Plot the scattered input data as in step (b) of Task 2.3 together with the separating hyperplane $H$, i.e., the contour line given by

$$\alpha_0 + \alpha_1 x_1 + \alpha_2 x_2 = \frac{1}{2},$$

where $x_1$ and $x_2$ denote the coordinates in $\mathbb{R}^2$ (not to be confused with the data $\boldsymbol{x}_i$). The result for the task should look (approximately) like Figure 2.1.

As mentioned above, the separating hyperplane from Task 2.4 can be used to divide or *classify* the data into two parts by the level set classifier (2.4). Let us quantify how good our classifier really is.

**Task 2.5.** Build the confusion matrix $C$ for the data and the hyperplane from Task 2.4. In our case this is a $2 \times 2$ matrix with $i, j \in \{0, 1\}$, since $|\Gamma| = 2$. Calculate the accuracy $\frac{\operatorname{trace}(C)}{n}$.

**Task 2.6.** Create 10 000 test points for each of the two classes in the same way as you created the training data in step (b) of Task 2.3. Evaluate the LLS classifier, which was built on the training data, now on the test data and compute the confusion matrix and the accuracy of the test data. Compare your results to the ones from Task 2.5.

Next, we will try our LLS classifier on the *Iris* data set [DG17, Fis36]. This data set consists of 150 points in $\mathbb{R}^4$, which describe three different types of Iris plants; see Figure 2.2 for an example. We have three classes:

$$\{\text{Iris-setosa}, \text{ Iris-versicolor}, \text{ Iris-virginica}\}.$$

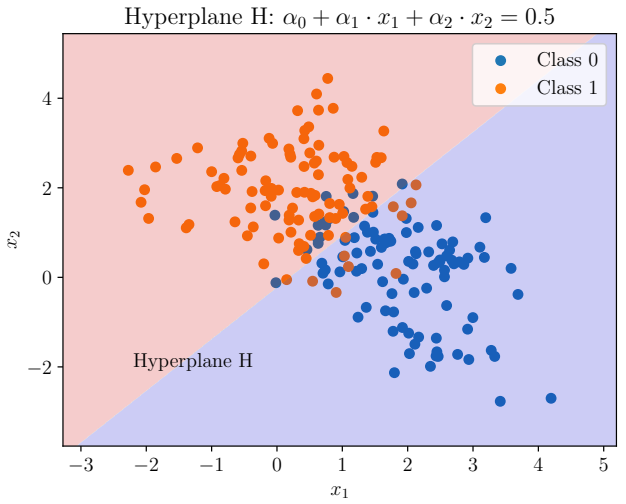



Figure 2.1: An example for a contour line plot for a separating hyperplane of two data classes. Note that the linear separation does not work very well due to the large overlap of the two data classes.

(a) Iris Setosa.[a] (b) Iris Versicolor.[b] (c) Iris Virginica.[c]

Figure 2.2: Pictures of three Iris plants.

[a] Photo by E. Forsberg licensed under the Creative Commons Attribution 2.0 generic license.
[b] Photo by Cliff1066 (Flickr) licensed under the Creative Commons Attribution 2.0 generic license.
[c] Photo by C. T. Johansson licensed under the Creative Commons Attribution 2.0 generic license.

The four *features*, i.e., the coordinates in $\Omega = \mathbb{R}^4$, refer to certain length and width measurements of the plants.

We will classify one of the three plant classes against both of the remaining classes by using our LLS algorithm. Thus, we encounter a two-class problem. To this end, we first have to read in the data set and cast the class names to $\Gamma = \{0, 1\}$. Then, we can proceed as for the previous tasks, i.e., we build an LLS regressor $\tilde{f}$ and construct the corresponding level set classifier $f = f_{\text{class}}$ (see (2.4)), for which we compute the confusion matrix and the accuracies. Instead of reading in the data by hand, we employ the very useful PANDAS library (https://pandas.pydata.org/) in PYTHON:

```
import pandas as pd
url = 'https://archive.ics.uci.edu/ml/machine-learning-databases/iris/
    iris.data'
irisDataFrame = pd.read_csv(url, header=None)
```

In PANDAS, the data are stored in an instance of `DataFrame`, on which many useful operations can be run.

**Task 2.7.** Make yourself familiar with the basics of PANDAS.

(a) Read in the Iris data set and use the data labels $\boldsymbol{y}_i = 0$ for the Iris-setosa instances and $\boldsymbol{y}_i = 1$ for the Iris-versicolor and Iris-virginica classes:

a.1. Run the LLS algorithm by using only the first two dimensions of $\Omega$ in the input data, i.e., only look at the first two features. Plot the scattered data and the separating hyperplane as in Task 2.4.

a.2. Now run the LLS algorithm by using all four features/dimensions of the input data. Compute the confusion matrix and the accuracy.

(b) Finally, run the same two steps as in (a), but now try to classify Iris-versicolor instances (label $\boldsymbol{y}_i = 0$) against both Iris-setosa and Iris-virginica (label $\boldsymbol{y}_i = 1$). What do you observe?

## 2.4 ▪ Iterative solvers

We have seen that solving the loss minimization (2.1) reduces either to solving the system of normal equations (2.2) with the symmetric, positive semi-definite matrix $\boldsymbol{X}^T\boldsymbol{X}$ or to computing the pseudo-inverse of $\boldsymbol{X}$ via SVD and using (2.3). While we have observed that the latter is preferable because of its numerical stability, we want to consider solving (2.2) here in order to introduce iterative solvers for linear systems of equations. One of the most famous iterative solvers, which we will also see again in Chapter 6, is *gradient descent*.

**The gradient descent method**

The motivation behind the ***gradient descent*** method is that solving the system of linear equations at hand can be recast into a minimization problem. To this end, consider the system $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ with a symmetric, positive definite matrix $\boldsymbol{A} \in \mathbb{R}^{\tilde{d}\times\tilde{d}}$ and vectors $\boldsymbol{x}, \boldsymbol{b} \in \mathbb{R}^{\tilde{d}}$. Solving this system is equivalent to minimizing the functional

$$J(\boldsymbol{x}) := \|\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b}\|_2^2 = (\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b})^T(\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b}) = \boldsymbol{x}^T\boldsymbol{A}^2\boldsymbol{x} - 2\boldsymbol{x}^T\boldsymbol{A}\boldsymbol{b} + \boldsymbol{b}^T\boldsymbol{b},$$

since $\nabla J(\boldsymbol{x}) = 2\boldsymbol{A}(\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b})$ and setting $\nabla J(\boldsymbol{x}) = \boldsymbol{0}$ yields $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ after multiplying with $\frac{\boldsymbol{A}^{-1}}{2}$. Gradient descent (Algorithm 1) now takes iterative steps of size $\nu > 0$ in the direction of the negative gradient of $J$ at the current iterate, i.e.,

$$\boldsymbol{x} \leftarrow \boldsymbol{x} - \nu\nabla J(\boldsymbol{x}).$$

Note that gradient descent can also be applied for nonlinear problems. In this case, it can be seen as iteratively solving a linear approximation to the nonlinear problem; see [BV04] for more details.

**Algorithm 1:** A gradient descent algorithm with step size (or *learning rate*) $\nu > 0$

**Input:** loss function $J$, step size $\nu > 0$, tolerance $\varepsilon > 0$.

```
Initialize α (e.g., randomly).
step ← 0.
while ‖∇J(α)‖ > ε and step ≤ maxSteps do
    α ← α − ν∇J(α).
    step ← step +1.
end while
```

In the case of linear least squares, we iteratively update our solution by taking steps in the negative direction of the gradient of the loss function. To this end, let

$$J(\boldsymbol{\alpha}) := \mathcal{L}\left((g(\boldsymbol{x}_1), y_1), \ldots, (g(\boldsymbol{x}_n), y_n)\right)$$

with $g(\boldsymbol{x}) = \boldsymbol{\alpha}^T \hat{\boldsymbol{x}}$. Thus, for $\mathcal{L} = \mathcal{L}_{\text{ls}}$, we obtain $J(\boldsymbol{\alpha}) = \frac{1}{n}\sum_{i=1}^{n}(g(\boldsymbol{x}_i) - y_i)^2$. Although it is not necessary to run a gradient descent optimizer in the LLS case, we will encounter more elaborate minimization tasks in later parts of this book, where this will be useful.

Besides gradient descent, there exists a variety of other iterative solvers for systems of linear equations. Here, standard iterative solvers like *Jacobi*, *Gauss–Seidel*, or *SOR* methods have to be mentioned. Furthermore, *multi-grid* or *multi-pole* methods can reduce the runtime for (approximately) solving a system of linear equations to $\mathcal{O}(\tilde{d})$ or $\mathcal{O}(\tilde{d}\log(\tilde{d}))$ for $\tilde{d} \times \tilde{d}$ matrices; see, e.g., [Hac13] or [Saa03]. However, all of the aforementioned methods have very special conditions on the structure of the underlying matrix, which are usually not directly fulfilled in the least squares setting or for other general machine learning tasks. A survey of iterative methods for the solution of linear systems of equations can be found in [Hac16].

**Task 2.8.** Implement the gradient descent method and run an LLS algorithm with a gradient descent optimizer for the data from Task 2.7 (a.1.). Choose $\nu \in \{1, 10^{-1}, 10^{-2}, \ldots\}$ as the largest value such that convergence is achieved. Create a plot of the value of $J$ versus the actual iteration number. What do you observe?

## 2.5 ▪ Data normalization

An underestimated preprocessing step in data analysis is *data normalization* or *data scaling*.

**Data normalization**

The need for normalizing the data is often overlooked when applying ML algorithms. However, it is a crucial data preprocessing step, which can significantly alter the performance and even the outcome of an ML algorithm. The most common method of data normalization is ***data standardization***. Here, the vector of feature means is subtracted from the data and the result is divided elementwise by the feature's standard deviation. Additionally, other data normalization techniques like clipping at certain boundaries are used in practical applications.

**Task 2.9.** Normalize the data from Task 2.7 (a.1.) by standardization. To this end, calculate the mean $m_j$ and the standard deviation $\sigma_j$ for each feature $j$ (i.e., each coordinate direction $j$ of the data set) and set the $j$th component of the $i$th data point to

$$[\boldsymbol{x}_i]_j := \frac{[\boldsymbol{x}_i]_j - m_j}{\sigma_j}.$$

Now run the gradient descent LLS algorithm on the normalized data. Similarly as in Task 2.8, choose $\nu$ as the largest value such that convergence is achieved. Compare the first 100 iteration steps by plotting the value of $J$ versus the iteration number for both the normalized and the unnormalized cases. What do you observe?

## 2.6 ▪ $k$-nearest neighbors

Another quite simple supervised learning method is the so-called *$k$-nearest neighbors* algorithm.

**$k$-nearest neighbors**

The idea of the ***k-nearest neighbors*** ($k$-nn) algorithm on data $(\boldsymbol{x}_i, y_i) \in \mathbb{R}^d \times \mathbb{R}$ is to build the solution to the least squares problem directly from the labels $y_i$. To this end, the labels of the $k$ data points closest to an evaluation point $\boldsymbol{x}$ are taken into account. In particular, we use the piecewise constant approximation

$$f(\boldsymbol{x}) := \text{NearNeigh}_k(\boldsymbol{x}) := \frac{1}{k} \sum_{\{i \mid \boldsymbol{x}_i \in N_k(\boldsymbol{x})\}} y_i \tag{2.6}$$

with the $k$-neighborhood[14] $N_k(\boldsymbol{x}) := \{\boldsymbol{x}_i \mid \exists\, I \subset \{1, \dots, n\} \setminus \{i\}, |I| \geq n - k$ such that $\|\boldsymbol{x} - \boldsymbol{x}_i\| \leq \|\boldsymbol{x} - \boldsymbol{x}_j\| \; \forall\, j \in I\}$.

To justify why the $k$-nn algorithm actually constructs a meaningful solution, we first have a look at the continuous least squares problem with model class $\mathcal{M}$. We are looking for

$$\begin{aligned}
&\min_{g \in \mathcal{M}} \mathbb{E}_\mu \left[ (Y - g(X))^2 \right] \quad \text{(expected squared loss)} \\
&= \min_{g \in \mathcal{M}} \int_{\Omega \times \Gamma} (y - g(\boldsymbol{x}))^2 \, \mathrm{d}\mu(\boldsymbol{x}, y) \\
&= \min_{g \in \mathcal{M}} \int_\Omega \int_\Gamma (y - g(\boldsymbol{x}))^2 \, \mathrm{d}\mu_{Y|X}(y|\boldsymbol{x}) \, \mathrm{d}\mu_X(\boldsymbol{x})
\end{aligned}$$

with marginal measure $\mu_X$ on $\Omega$ and conditional measure $\mu_{Y|X}$ of $Y$ given $X$ on $\Gamma$. We observe that the optimal $g(\boldsymbol{x})$ at $\boldsymbol{x} \in \Omega$ is

$$\arg\min_{c \in \Gamma} \int_\Gamma (y - c)^2 \, \mathrm{d}\mu_{Y|X}(y|\boldsymbol{x}) = \mathbb{E}_\mu \left[ Y | X = \boldsymbol{x} \right].$$

[14] Note that distance measures other than the Euclidean distance can also be used to define the $k$-neighborhood.

Having a closer look at the $k$-nn estimator, we see that it essentially takes two steps to approximate the quantity $\mathbb{E}_\mu[Y|X=\boldsymbol{x}]$:

1. Since only finitely many samples are given, the expected value is approximated by the sample mean
$$\mathbb{E}_\mu[Y|X=\boldsymbol{x}] \approx \frac{1}{p}\sum_{j=1}^{p} y_{i_j},$$
where the $p \leq n$ indices $i_j \in \{1,\dots,n\}$ are picked corresponding to data pairs $(\boldsymbol{x}_{i_j}, y_{i_j})$ with $\boldsymbol{x}_{i_j} = \boldsymbol{x}$.
2. Since there usually do not exist (m)any $\boldsymbol{x}_i$ in the data set with $\boldsymbol{x}_i = \boldsymbol{x}$, we also take values sufficiently close to $\boldsymbol{x}$ into account. To this end, we consider the $k$-neighborhood $N_k(\boldsymbol{x})$ of $\boldsymbol{x}$ and use the corresponding $p = k$ indices of the data points in $N_k(\boldsymbol{x})$ for the approximation in step 1.

Note that if $|N_k(\boldsymbol{x})| > k$, we randomly choose points in $N_k(\boldsymbol{x})$ with the largest distance to $\boldsymbol{x}$ and remove them until $|N_k(\boldsymbol{x})| = k$ to get the appropriate neighborhood size.

**Task 2.10.** Let us now test how the algorithm performs for every possible $k$. For the implementation of $k$-nearest neighbors you can use the `scipy.spatial.distance.cdist` and `numpy.argpartition` functions, for example. To this end, use the $k$-nearest neighbors algorithm with the level set classifier (2.4).

(a) Run the $k$-nearest neighbors algorithm for the data from Task 2.3 for all $k = 1,\dots,200$ and store the accuracy for each $k$.

(b) Do the same as in step (a) but now use the data created in Task 2.6 as test data.

(c) Plot the accuracies from steps (a) and (b) versus the value of $k$. What do you observe?

As we have seen in Task 2.10, we can use the $k$-nearest neighbors formula (2.6) together with a level set classifier for binary classifications tasks. However, to classify a data point $\boldsymbol{x}$ in a multiclass setting, we determine the majority class among the $k$-nearest neighbors of $\boldsymbol{x}$ in the training data set. Note that this gives the same result as the level set classifier in the binary classification setting.

## 2.7 ▪ Further topics

**Stochastics** Since we can usually assume that the input data are drawn according to an (unknown) probability distribution, we can formulate the problem of finding an optimal classifier/regressor also as a stochastic problem. In this context especially the so-called *Bayesian methods* are commonly used. Here, Bayes' theorem is applied to obtain a solution to the corresponding stochastic optimization problem; see, e.g., [HTF09, Mur22]. A special case of such a Bayesian learning problem will be tackled in Chapter 7.

**Logistic regression** Another famous linear model to obtain optimal classifiers is the *logistic regression* model, by which the distribution of the underlying random variables is modeled; see [HTF09, Mur22]. This approach also involves a loss function that is more suitable for classification than the least squares loss.

**Regularization** Instead of simply minimizing a loss function as in the case of linear least squares in (2.1), we could add a *regularization* term to the minimization problem, as we already sketched at the end of Section 1.3. This can be interpreted as a tradeoff between minimizing the loss on the training data and obtaining a simple or sparse model; see [HTF09, Mur22]. Examples for such regularization terms are $\ell_p$ norms of the coefficients (*Lasso, Tikhonov*) or more complex norms involving derivatives of the minimizer. Note that this approach can also recast an ill-posed linear least squares problem (2.2) into a well-posed one when the matrix $\boldsymbol{X}^T\boldsymbol{X}$ does not have full rank.

Chapter 3

# Optimal Separating Hyperplanes and Support Vector Machines

We have seen in Task 2.4 how a linear model can be used to obtain a separating hyperplane for classification. A drawback of the LLS approach is the fact that the least squares error does not capture what we expect from an optimal separation. For illustration, let us consider an example where the $\boldsymbol{x}_i$ belonging to one of the two classes $\Gamma = \{-1, 1\}$ are spread very widely over the domain, but the $\boldsymbol{x}_i$ belonging to the other class are very densely concentrated. In such a case we might encounter a situation where there exists an element in $\mathcal{M}_{\text{Lin}}$ such that its $0$-level set function perfectly separates the two classes $-1$ and $1$, but the solution to (2.2) does not; see Figure 3.1. The reason is that the linear least squares algorithm minimizes the squared distances between the true class label (either $-1$ or $1$) and the prediction (a real number). Thus, its objective is not to determine a separating hyperplane but rather to avoid predictions that deviate significantly from the true label.

In this chapter, we now consider methods to obtain a so-called *separating hyperplane* for a two-class classification problem. Section 3.1 deals with the concept of *optimal (margin) separating hyperplanes* and the corresponding optimization problem. Since this might not be solvable at all in certain situations, we introduce a slightly weakened form in Section 3.2, which leads to the well-known *support vector machines*. To solve the underlying optimization problem, we study a special optimization algorithm for quadratic optimization problems with linear constraints called *sequential minimal optimization* in Section 3.3. Subsequently, we encounter our first programming tasks on separating hyperplanes in Section 3.4. While the initial model space for support vector machines contains only affine linear functions, the so-called *kernel trick*, which we discuss in Section 3.5, allows us to successfully deal with nonlinear problems too. Since the support vector machine algorithm employs certain hyperparameters, we consider their optimization in Section 3.6. Finally, we introduce the PYTHON library SCIKIT-LEARN, which we employ to deal with an image classification task in Section 3.7.

## 3.1 ▪ Optimal separating hyperplanes

Instead of calculating the $0$-level set of the LLS function, we now have a look at the so-called *optimal separating hyperplane* (OSH) $H$ that separates the data such that the *margin*,

$$M := \min_{i=1,\dots,n} \operatorname{dist}(\boldsymbol{x}_i, H),$$

is maximized. To explicitly define the distance $\operatorname{dist}(\boldsymbol{x}, H)$ of a point $\boldsymbol{x}$ to $H$, let

$$H := \left\{ \boldsymbol{x} \in \mathbb{R}^d \mid f(\boldsymbol{x}) = 0 \right\}$$

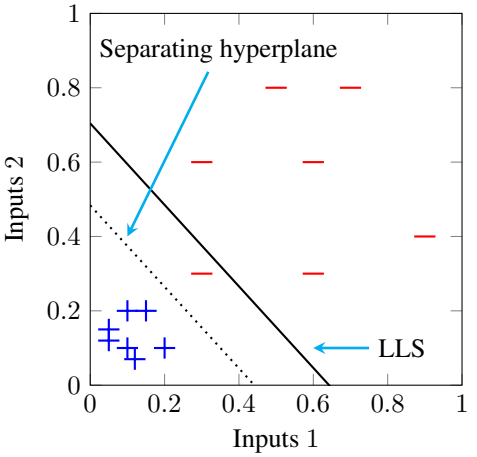



Figure 3.1: Example for a scenario where the 0-level set function of the linear least squares (LLS) solution does not separate the data classes (depicted by $-$ and $+$) although there exists a linear separating hyperplane.

be the 0-level set of a function

$$f(\boldsymbol{x}) := \boldsymbol{\gamma}^T \boldsymbol{x} + \gamma_0.$$

Note that $\boldsymbol{\gamma}^* := \frac{\boldsymbol{\gamma}}{\|\boldsymbol{\gamma}\|}$ is a unit normal vector to $H$ since

$$\boldsymbol{\gamma}^T (\boldsymbol{t}_1 - \boldsymbol{t}_2) = f(\boldsymbol{t}_1) - f(\boldsymbol{t}_2) = 0 \ \ \forall\, \boldsymbol{t}_1, \boldsymbol{t}_2 \in H.$$

Therefore, the signed distance of a point $\boldsymbol{x} \in \mathbb{R}^d$ to $H$ can be computed by

$$(\boldsymbol{\gamma}^*)^T (\boldsymbol{x} - \boldsymbol{t}_0) = \frac{1}{\|\boldsymbol{\gamma}\|} \left(\boldsymbol{\gamma}^T \boldsymbol{x} + \gamma_0\right)$$

for any $\boldsymbol{t}_0 \in H$ and

$$\operatorname{dist}(\boldsymbol{x}, H) = \frac{1}{\|\boldsymbol{\gamma}\|} \left|\boldsymbol{\gamma}^T \boldsymbol{x} + \gamma_0\right|.$$

Now, analogously to $\boldsymbol{\alpha}$ in the LLS algorithm, we need to determine a $\boldsymbol{\gamma} := (\gamma_1, \ldots, \gamma_d)^T \in \mathbb{R}^d$ and a $\gamma_0 \in \mathbb{R}$ that satisfy our need.

**Optimal separating hyperplane (OSH)**

In a two-class problem, the ***optimal separating hyperplane*** (OSH) is defined as the hyperplane that separates the training data classes and maximizes its distance from the training data. Writing this as an optimization problem in terms of the so-called *margin* $M \in (0, \infty)$, we obtain

$$\max_{\gamma_0 \in \mathbb{R}, \boldsymbol{\gamma} \in \mathbb{R}^d} M \quad \text{s.t.} \quad y_i \left(\boldsymbol{\gamma}^T \boldsymbol{x}_i + \gamma_0\right) \geq \|\boldsymbol{\gamma}\| \cdot M \ \ \forall\, i \in \{1, \ldots, n\}. \tag{3.1}$$

Note that an OSH does not necessarily exist since the data might not be linearly separable at all; see Section 3.5 for more details.

Having a closer look at Figure 3.1, we observe that the dotted separating hyperplane is already the optimal separating hyperplane in the above sense.

Since we can use arbitrary positive multiples of $\boldsymbol{\gamma}$ and $\gamma_0$ and still obtain the same margin $M$ in (3.1), we set $\|\boldsymbol{\gamma}\| = \frac{1}{M}$ and obtain the equivalent minimization problem

$$\begin{aligned} &\min_{\gamma_0 \in \mathbb{R}, \boldsymbol{\gamma} \in \mathbb{R}^d} \frac{1}{2}\|\boldsymbol{\gamma}\|^2 \\ &\text{s.t.} \qquad y_i \left(\boldsymbol{\gamma}^T \boldsymbol{x}_i + \gamma_0\right) \geq 1 \qquad \forall\, i \in \{1, \ldots, n\}. \end{aligned} \tag{3.2}$$

By equivalence, we mean that a minimizer of (3.2) is also a maximizer of (3.1) and vice versa. As (3.2) is a convex minimization problem with linear inequality constraints, there exists a unique (global) minimizer, which is also a *Karush–Kuhn–Tucker* point.

**Karush–Kuhn–Tucker (KKT) conditions**

The ***Karush–Kuhn–Tucker*** (KKT) conditions are necessary criteria for an optimizer of a nonlinear minimization problem

$$\min_{\boldsymbol{t} \in \tau} F(\boldsymbol{t}) \quad \text{s.t.} \quad G_i(\boldsymbol{t}) \leq 0, H_j(\boldsymbol{t}) = 0 \,\forall\, i = 1, \ldots, m \text{ and } j = 1, \ldots, l \tag{3.3}$$

of a continuously differentiable function $F : \tau \subset \mathbb{R}^d \to \mathbb{R}$ under the constraints $G_i(\boldsymbol{t}) \leq 0$ and $H_j(\boldsymbol{t}) = 0$ for continuously differentiable $G_i, H_j : \tau \to \mathbb{R}$ with $i = 1, \ldots, m$ and $j = 1, \ldots, l$. With the definition

$$L(\boldsymbol{t}, \boldsymbol{a}, \boldsymbol{b}) := F(\boldsymbol{t}) + \sum_{i=1}^{m} a_i G_i(\boldsymbol{t}) + \sum_{j=1}^{l} b_j H_j(\boldsymbol{t})$$

of the so-called ***Lagrangian***, the KKT conditions for some $\boldsymbol{t} \in \tau$ and ***Lagrange multipliers*** $\boldsymbol{a} \in \mathbb{R}^m$ and $\boldsymbol{b} \in \mathbb{R}^l$ read

$$\begin{aligned} \nabla_{\boldsymbol{t}} L(\boldsymbol{t}, \boldsymbol{a}, \boldsymbol{b}) &= 0, \\ G_i(\boldsymbol{t}) &\leq 0 \quad \forall\, i = 1, \ldots, m, \\ H_j(\boldsymbol{t}) &= 0 \quad \forall\, j = 1, \ldots, l, \\ a_i &\geq 0 \quad \forall\, i = 1, \ldots, m, \\ a_i G_i(\boldsymbol{t}) &= 0 \quad \forall\, i = 1, \ldots, m. \end{aligned}$$

If the problem is convex, i.e., $F, G_i$ for $i = 1, \ldots, m$ and $\tau$ are convex and $H_j$ is (affine) linear for each $j = 1, \ldots, l$, and the triplet $(\boldsymbol{t}, \boldsymbol{a}, \boldsymbol{b})$ fulfills the KKT conditions, $\boldsymbol{t}$ is already a minimizer fulfilling the constraints. We refer the reader to [BV04] for more details.

Instead of directly looking for a minimum of (3.2), we can equivalently look for points fulfilling the KKT conditions. To this end, note that we have $m = n$ inequality constraints $G_i(\boldsymbol{t})$ and $l = 0$ equality constraints $H_j(\boldsymbol{t})$. Let

$$L((\boldsymbol{\gamma}, \gamma_0), \boldsymbol{\beta}) := \frac{1}{2}\|\boldsymbol{\gamma}\|^2 + \sum_{i=1}^{n} \beta_i \left(1 - y_i \left(\boldsymbol{\gamma}^T \boldsymbol{x}_i + \gamma_0\right)\right)$$

be the Lagrangian function. The KKT conditions for a minimizer now read

$$\begin{aligned}
&\text{(a)} && \nabla_{\boldsymbol{\gamma}} L = \nabla_{\gamma_0} L = 0, && \\
&\text{(b)} && y_i \left(\boldsymbol{\gamma}^T \boldsymbol{x}_i + \gamma_0\right) \geq 1 && \forall\, i \in \{1, \ldots, n\}, \\
&\text{(c)} && \beta_i \geq 0 && \forall\, i \in \{1, \ldots, n\}, \\
&\text{(d)} && \beta_i \left(y_i \left(\boldsymbol{\gamma}^T \boldsymbol{x}_i + \gamma_0\right) - 1\right) = 0 && \forall\, i \in \{1, \ldots, n\}.
\end{aligned}$$

Evaluating condition (a) further leads to

$$\text{(a1)} \quad \boldsymbol{\gamma} = \sum_{i=1}^{n} \beta_i y_i \boldsymbol{x}_i,$$

$$\text{(a2)} \quad 0 = \sum_{i=1}^{n} \beta_i y_i.$$

Because of (a1) the $\boldsymbol{x}_i$ with $\beta_i > 0$ are actually called *support vectors* since they span the hyperplane vector $\boldsymbol{\gamma}$. If we now substitute (a1) into the Lagrangian and apply (a2), we obtain

$$L((\boldsymbol{\gamma}, \gamma_0), \boldsymbol{\beta}) = -\frac{1}{2} \sum_{i,j=1}^{n} \beta_i \beta_j y_i y_j \boldsymbol{x}_i^T \boldsymbol{x}_j - \underbrace{\sum_{i=1}^{n} \beta_i y_i \gamma_0}_{=0} + \sum_{i=1}^{n} \beta_i.$$

Therefore, the so-called *Wolfe dual problem* reads

$$\begin{aligned}
&\max_{\boldsymbol{\beta} \in \mathbb{R}^n} \sum_{i=1}^{n} \beta_i - \frac{1}{2} \sum_{i,j=1}^{n} \beta_i \beta_j y_i y_j \boldsymbol{x}_i^T \boldsymbol{x}_j \\
&\text{s.t.} \quad \beta_i \geq 0 \qquad \forall\, i \in \{1, \ldots, n\} \\
&\text{and} \quad \sum_{i=1}^{n} \beta_i y_i = 0.
\end{aligned} \tag{3.4}$$

**Lagrange and Wolfe duality**

The problem (3.3) is usually referred to as the ***primal problem***. The ***Lagrangian dual problem*** to (3.3) is given as

$$\max_{\substack{\boldsymbol{a} \in \mathbb{R}^m \\ \boldsymbol{b} \in \mathbb{R}^l}} \inf_{\boldsymbol{t} \in \tau} L(\boldsymbol{t}, \boldsymbol{a}, \boldsymbol{b}) \quad \text{s.t.} \quad a_i \geq 0 \, \forall\, i = 1, \ldots, m. \tag{3.5}$$

If the problem is convex and there are no equality constraints $H_j$ present, the above formulation becomes the so-called ***Wolfe dual problem***

$$\max_{\substack{\boldsymbol{t} \in \tau \\ \boldsymbol{a} \in \mathbb{R}^m}} L(\boldsymbol{t}, \boldsymbol{a}, \boldsymbol{b}) \quad \text{s.t.} \quad \nabla_{\boldsymbol{t}} L(\boldsymbol{t}, \boldsymbol{a}, \boldsymbol{b}) = 0 \text{ and } a_i \geq 0 \, \forall\, i = 1, \ldots, m. \tag{3.6}$$

If there exists some $\boldsymbol{t} \in \tau$ such that $G_i(\boldsymbol{t}) < 0$ for all $i = 1, \ldots, m$, it can be shown that the minimum of (3.3) is a maximum of (3.6) and vice versa, i.e., by solving one problem we already solved the other one. More details can again be found in [BV04].

In our case, we have *strong duality*, i.e., the maximal value of the dual problem equals the minimal value of the primal problem (3.2). Furthermore, because of the convex nature of the

primal problem, the maximizer of (3.4) defines a minimizer to (3.2) via condition (a1); see [BV04]. In particular, condition (a1) lets us write the solution to (3.4) as

$$f(\boldsymbol{t}) = \gamma_0 + \sum_{i=1}^{d} \gamma_i t_i = \gamma_0 + \sum_{i=1}^{n} \beta_i y_i \boldsymbol{x}_i^T \boldsymbol{t}. \tag{3.7}$$

This representation is very efficient if the dimension $d$ is very large but most of the $\beta_i$ are actually 0. Note that, after solving (3.4), we still have to determine $\gamma_0 \in \mathbb{R}$ to obtain the final representation. This will be dealt with in the next section.

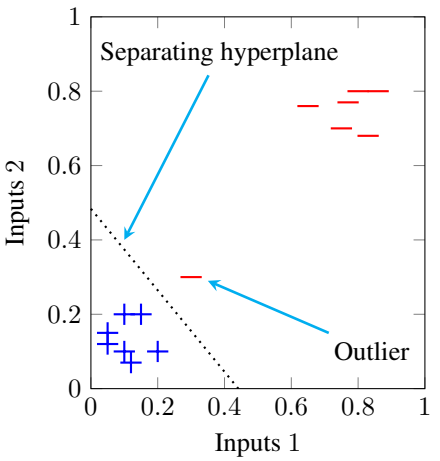
## 3.2 ▪ Soft margin hyperplanes and support vector machines

While solving (3.4) will give the optimal separating hyperplane as described in the previous section, there are situations in which this is not the most desirable solution or in which the algorithm will even fail to find a meaningful solution. For example, if the data have *outliers*, the OSH solution can be heavily influenced by them; see Figure 3.2(a). Furthermore, if the data are not linearly separable at all, the algorithm might fail; see Figure 3.2(b).

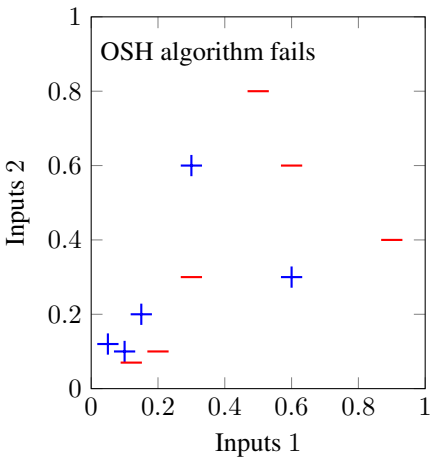
**Outliers**

A data point $(\boldsymbol{x}, y)$ that does not exhibit the general behavior of the rest of the data or that is drawn according to a different distribution from the rest of the data is called an ***outlier***. Outlier detection is a research area on its own, which we do not address here. A survey on outlier detection methods is given in [HA04].

To cope with these situations we have to come up with a more sophisticated (and regularized) version of the algorithm: The so-called *soft-margin hyperplanes* method, also known as *support vector machine*.




(a) An example where one outlier heavily influences the solution of the OSH algorithm.




(b) An example where an OSH solution does not exist since the data are not linearly separable.

Figure 3.2: Two situations in which the OSH method does not work well or at all.

**Support vector machines (SVMs)**

***Support vector machines*** (SVMs)—as introduced by Cortes and Vapnik in [CV95]—inherit their name from the support vectors already discussed in the OSH context. The idea behind SVMs is simple: Instead of using the hard constraints from OSH, some slack for misclassified values and vectors close to the hyperplane is allowed. This leads to a more flexible algorithm. For more details on support vector machines, we refer the reader to [CV95, SS02].

We now introduce *slack variables* $\xi_i \geq 0$ for all $i = 1, \ldots, n$. Instead of using the hard constraints from OSH, we allow some slack for misclassified values and vectors close to the hyperplane. To this end, we reformulate the OSH optimization problem (3.2) with the help of the slack variables to obtain

$$\begin{aligned} \min_{\gamma_0 \in \mathbb{R}, \boldsymbol{\gamma} \in \mathbb{R}^d, \boldsymbol{\xi} \in \mathbb{R}^n} \; & \frac{1}{2}\|\boldsymbol{\gamma}\|^2 + C \sum_{i=1}^{n} \xi_i \\ \text{s.t.} \quad & y_i \left(\boldsymbol{\gamma}^T \boldsymbol{x}_i + \gamma_0\right) \geq 1 - \xi_i \qquad \forall\, i \in \{1, \ldots, n\} \\ \text{and} \quad & \xi_i \geq 0 \qquad \forall\, i \in \{1, \ldots, n\}, \end{aligned}$$

where $C > 0$ is the regularization parameter. We can recover OSH by setting $C = \infty$, i.e., by enforcing $\xi_i = 0$ for all $i = 1, \ldots, n$. A large $C$ results in better fits of the training data, while a small $C$ allows for larger slack and leads to larger margins.

By introducing new Lagrange multipliers $r_i \geq 0$ for all $i = 1, \ldots, n$, we get the Lagrangian

$$L((\boldsymbol{\gamma}, \gamma_0), \boldsymbol{\xi}, \boldsymbol{\beta}, \boldsymbol{r}) = \frac{1}{2}\|\boldsymbol{\gamma}\|^2 + C \sum_{i=1}^{n} \xi_i - \sum_{i=1}^{n} \beta_i \left(y_i \left(\boldsymbol{\gamma}^T \boldsymbol{x}_i + \gamma_0\right) - 1 + \xi_i\right) - \sum_{i=1}^{n} r_i \xi_i.$$

Again, we are interested in the KKT conditions. Calculating $\nabla_{\boldsymbol{\gamma}} L$ and $\nabla_{\gamma_0} L$ gives the same conditions (a1) and (a2) from OSH. Furthermore, a KKT point fulfills

$$\frac{\partial}{\partial \xi_i} L = C - \beta_i - r_i = 0 \;\Leftrightarrow\; r_i = C - \beta_i \quad \forall\, i = 1, \ldots, n.$$

Substituting this together with (a1) and (a2) into the Lagrangian, we obtain

$$\begin{aligned} L((\boldsymbol{\gamma}, \gamma_0), \boldsymbol{\xi}, \boldsymbol{\beta}, \boldsymbol{r}) &= -\frac{1}{2} \sum_{i,j=1}^{n} \beta_i \beta_j y_i y_j \boldsymbol{x}_i^T \boldsymbol{x}_j + C \sum_{i=1}^{n} \xi_i - \underbrace{\sum_{i=1}^{n} \beta_i y_i \gamma_0}_{=0} \\ &\quad + \sum_{i=1}^{n} \beta_i - \sum_{i=1}^{n} \beta_i \xi_i - \sum_{i=1}^{n} (C - \beta_i) \xi_i \\ &= -\frac{1}{2} \sum_{i,j=1}^{n} \beta_i \beta_j y_i y_j \boldsymbol{x}_i^T \boldsymbol{x}_j + \sum_{i=1}^{n} \beta_i, \end{aligned}$$

which leads to the Wolfe dual problem

$$\begin{aligned} \max_{\boldsymbol{\beta} \in \mathbb{R}^n} \; & \sum_{i=1}^{n} \beta_i - \frac{1}{2} \sum_{i,j=1}^{n} \beta_i \beta_j y_i y_j \boldsymbol{x}_i^T \boldsymbol{x}_j \\ \text{s.t.} \quad & 0 \leq \beta_i \leq C \qquad \forall\, i \in \{1, \ldots, n\} \\ \text{and} \quad & \sum_{i=1}^{n} \beta_i y_i = 0. \end{aligned} \tag{3.8}$$

Note that this differs from (3.4) only in the upper bound $C$ on each Lagrange parameter $\beta_i$ for $i = 1, \ldots, n$, which stems from the additional KKT condition for the Lagrange multipliers $r_i$, i.e., $0 \leq r_i = C - \beta_i$. The remaining conditions are derived completely analogously to (3.4).

Finally, we need to fit the so-called *bias* $\gamma_0 \in \mathbb{R}$. To this end, note that computing the KKT conditions for the Lagrangian $L((\boldsymbol{\gamma}, \gamma_0), \boldsymbol{\xi}, \boldsymbol{\beta}, \boldsymbol{r})$ leads to $r_i \xi_i = 0$ and

$$\beta_i \left( y_i \left( \boldsymbol{\gamma}^T \boldsymbol{x}_i + \gamma_0 \right) - 1 + \xi_i \right) = 0$$

for all $i = 1, \ldots, n$. If $\beta_i > 0$, we obtain

$$\sum_{j=1}^{n} y_i y_j \beta_j \boldsymbol{x}_j^T \boldsymbol{x}_i - 1 + \xi_i = -\gamma_0 y_i$$

by using (a1). Since $y_i \in \{-1, 1\}$, we get

$$\sum_{j=1}^{n} y_j \beta_j \boldsymbol{x}_j^T \boldsymbol{x}_i - y_i + y_i \xi_i = -\gamma_0$$

by multiplying with $y_i$. Since $r_i \xi_i = 0$ and $r_i = C - \beta_i$, we have $\xi_i = 0$ if $\beta_i < C$ and therefore

$$y_i - \sum_{j=1}^{n} y_j \beta_j \boldsymbol{x}_j^T \boldsymbol{x}_i = \gamma_0$$

for all $i \in \{1, \ldots, n\}$ for which $0 < \beta_i < C$. Thus, to average out possible numerical errors, we choose $\gamma_0$ to be the mean

$$\gamma_0 := \text{mean} \left( y_k - \sum_{i=1}^{n} \beta_i y_i \boldsymbol{x}_i^T \boldsymbol{x}_k \right)$$

taken over all indices $k \in \{i \in \{1, \ldots, n\} \mid 0 < \beta_i < C\}$.

## 3.3 ▪ Sequential minimal optimization

Having a closer look at (3.8), we realize that we have to deal with a quadratic optimization problem with linear constraints and box constraints. We can easily write it in the more convenient matrix-vector format

$$\begin{aligned} &\min_{\boldsymbol{\beta} \in \mathbb{R}^n} \frac{1}{2} \boldsymbol{\beta}^T \boldsymbol{Q} \boldsymbol{\beta} - \mathbf{1}^T \boldsymbol{\beta} \\ &\text{s.t.} \quad 0 \leq \beta_i \leq C \qquad \forall\, i \in \{1, \ldots, n\} \\ &\text{and} \quad \boldsymbol{\beta}^T \boldsymbol{y} = 0 \end{aligned}$$

with $\boldsymbol{Q} \in \mathbb{R}^{n \times n}$ with entries $Q_{ij} = y_i y_j \boldsymbol{x}_i^T \boldsymbol{x}_j$ for all $i, j = 1, \ldots, n$ and label vector $\boldsymbol{y} := (y_1, \ldots, y_n)^T \in \mathbb{R}^n$.

**Chunking** If $n$ is quite large, it makes sense to use a so-called *chunking* approach, i.e., to split $\{1, \ldots, n\}$ into two disjoint sets $w$ (working set) and $\kappa$ (fixed set) such that $w \cup \kappa = \{1, \ldots, n\}$. We write $\boldsymbol{\beta}_w$ and $\boldsymbol{\beta}_\kappa$ for the subvectors of $\boldsymbol{\beta}$ corresponding to the indices in $w$ and $\kappa$. The idea is now to solve the problem only with respect to the variables in $w$ and to repeat this process

several times for different working sets. To this end, the subproblem of optimizing with respect to $w$ reads

$$\begin{aligned}\min_{\boldsymbol{\beta}_w \in \mathbb{R}^{|w|}} \; & \frac{1}{2}\boldsymbol{\beta}_w^T \boldsymbol{Q}_{ww}\boldsymbol{\beta}_w - \left(\mathbf{1}_w - \boldsymbol{Q}_{w\kappa}\boldsymbol{\beta}_\kappa\right)^T \boldsymbol{\beta}_w + \underbrace{\frac{1}{2}\boldsymbol{\beta}_\kappa^T \boldsymbol{Q}_{\kappa\kappa}\boldsymbol{\beta}_\kappa + \mathbf{1}_\kappa^T \boldsymbol{\beta}_\kappa}_{\text{constant}} \\ \text{s.t.} \quad & 0 \leq \beta_i \leq C \qquad \forall\, i \in w \\ \text{and} \quad & \boldsymbol{\beta}_w^T \boldsymbol{y}_w = -\boldsymbol{\beta}_\kappa^T \boldsymbol{y}_\kappa,\end{aligned}$$

where $\boldsymbol{Q}_{ww}$, $\boldsymbol{Q}_{w\kappa}$, and $\boldsymbol{Q}_{\kappa\kappa}$ canonically denote rows and columns of the matrix $\boldsymbol{Q}$ with respect to the corresponding index set. Let us have a look at one of the most successful optimization algorithms using chunking to solve (3.8). The so-called *sequential minimal optimization* (SMO) algorithm [Pla98] uses chunking with the smallest possible subsets $w$, i.e., $|w| = 2$ (since nothing can be optimized for $|w| = 1$ due to the equality constraints). To this end, the algorithm works in an iterative manner: At first the values for $\boldsymbol{\beta}$ and the bias $\gamma_0$ from the representation (3.7) are initialized (e.g., as 0). In every iteration step we select two indices $i, j \in \{1, \ldots, n\}$ and solve the corresponding quadratic optimization problem, i.e., we employ the chunking approach and choose $w = \{i, j\}$ as the working set.

**One step of SMO** Let us now have a closer look at one iteration of the SMO algorithm with $w = \{i, j\}$. The working set optimization problem reads

$$\begin{aligned}\min_{\beta_i, \beta_j \in \mathbb{R}} \; & \frac{1}{2}\left(\beta_i^2 \boldsymbol{x}_i^T \boldsymbol{x}_i + \beta_j^2 \boldsymbol{x}_j^T \boldsymbol{x}_j + 2\beta_i\beta_j y_i y_j \boldsymbol{x}_i^T \boldsymbol{x}_j\right) - c_i\beta_i - c_j\beta_j \\ \text{s.t.} \quad & 0 \leq \beta_i \leq C \\ \text{and} \quad & 0 \leq \beta_j \leq C \\ \text{and} \quad & y_i\beta_i + y_j\beta_j = -\sum_{l \notin \{i,j\}} y_l\beta_l = y_i\beta_i^{\text{old}} + y_j\beta_j^{\text{old}}\end{aligned} \tag{3.9}$$

with $\beta_k^{\text{old}}$ being $\beta_k$ from before the optimization and

$$c_k := 1 - y_k \cdot \sum_{l \notin \{i,j\}} \beta_l y_l \boldsymbol{x}_k^T \boldsymbol{x}_l$$

for $k = i, j$.

Note that due to $y_j y_j = 1$ the equality constraint is equivalent to $\beta_j = \vartheta - y_i y_j \beta_i$ for $\vartheta := y_i y_j \beta_i^{\text{old}} + \beta_j^{\text{old}}$. Therefore, we can substitute $\beta_j$ into the minimization problem above. This leads to the following formulation:

$$\begin{aligned}(3.9) \Leftrightarrow \min_{\beta_i \in \mathbb{R}} \; & \frac{1}{2}\beta_i^2 \left(\underbrace{\boldsymbol{x}_i^T \boldsymbol{x}_i + \boldsymbol{x}_j^T \boldsymbol{x}_j - 2\boldsymbol{x}_i^T \boldsymbol{x}_j}_{=:\,\chi}\right) \\ & -\beta_i \left(\underbrace{c_i - y_i y_j c_j - \vartheta y_i y_j \boldsymbol{x}_i^T \boldsymbol{x}_j + \vartheta y_i y_j \boldsymbol{x}_j^T \boldsymbol{x}_j}_{=:\,\psi}\right) \\ \text{s.t.} \; & \text{Lo} \leq \beta_i \leq \text{Up}\end{aligned}$$

with

$$\text{Lo} := \begin{cases} \max(0, \vartheta - C) & \text{if } y_i y_j = 1, \\ \max(0, -\vartheta) & \text{else} \end{cases}$$

and

$$\mathrm{Up} := \begin{cases} \min(\vartheta, C) & \text{if } y_i y_j = 1, \\ \min(C - \vartheta, C) & \text{else.} \end{cases}$$

As this is simply a quadratic minimization problem in one variable with box constraints, the solution is given by

$$\beta_i = \begin{cases} \min\left(\max\left(\frac{\psi}{\chi}, \mathrm{Lo}\right), \mathrm{Up}\right) & \text{if } \chi > 0, \\ \mathrm{Up} & \text{if } \chi = 0 \text{ and } \psi \geq 0, \\ \mathrm{Lo} & \text{if } \chi = 0 \text{ and } \psi < 0. \end{cases}$$

Using the representation for $f(\boldsymbol{x})$ from (3.7), we can write

$$\begin{aligned} c_i &= 1 - y_i \sum_{l \neq i,j} \beta_l y_l \boldsymbol{x}_l^T \boldsymbol{x}_i = y_i \left(y_i - f\left(\boldsymbol{x}_i\right) + \gamma_0\right) + \beta_i^{\text{old}} \boldsymbol{x}_i^T \boldsymbol{x}_i + \beta_j^{\text{old}} y_i y_j \boldsymbol{x}_i^T \boldsymbol{x}_j, \\ c_j &= 1 - y_j \sum_{l \neq i,j} \beta_l y_l \boldsymbol{x}_l^T \boldsymbol{x}_j = y_j \left(y_j - f\left(\boldsymbol{x}_j\right) + \gamma_0\right) + \beta_j^{\text{old}} \boldsymbol{x}_j^T \boldsymbol{x}_j + \beta_i^{\text{old}} y_i y_j \boldsymbol{x}_i^T \boldsymbol{x}_j, \end{aligned}$$

and due to $\vartheta = \beta_j^{\text{old}} + y_i y_j \beta_i^{\text{old}}$ we obtain

$$\begin{aligned} \psi &= y_i \left(y_i - f\left(\boldsymbol{x}_i\right) + \gamma_0\right) + \beta_i^{\text{old}} \boldsymbol{x}_i^T \boldsymbol{x}_i + \beta_j^{\text{old}} y_i y_j \boldsymbol{x}_i^T \boldsymbol{x}_j \\ &\quad - y_i \left(y_j - f\left(\boldsymbol{x}_j\right) + \gamma_0\right) - \beta_j^{\text{old}} y_i y_j \boldsymbol{x}_j^T \boldsymbol{x}_j - \beta_i^{\text{old}} \boldsymbol{x}_i^T \boldsymbol{x}_j \\ &\quad + \left(\beta_i^{\text{old}} + \beta_j^{\text{old}} y_i y_j\right) \left(\boldsymbol{x}_j^T \boldsymbol{x}_j - \boldsymbol{x}_i^T \boldsymbol{x}_j\right) \\ &= y_i \left(y_i - f(\boldsymbol{x}_i) - (y_j - f(\boldsymbol{x}_j))\right) + \beta_i^{\text{old}} \chi. \end{aligned}$$

Therefore, we now have an easy way to compute the true solution for the size-2 subproblem with indices $i$ and $j$; see Algorithm 2. There, $\langle \boldsymbol{x}_i, \boldsymbol{x}_j \rangle_\Omega = \boldsymbol{x}_i^T \boldsymbol{x}_j$ for $\Omega \subseteq \mathbb{R}^d$. Note that we can directly compute the true solution without any numerical approximation (besides floating point accuracy issues, etc.), which can be an important advantage over chunking algorithms working with subproblems for which only a numerically approximate solution can be computed. This is due to the potential accumulation of approximation errors in subsequent optimization steps.

**Choosing working sets** Finding good candidates for possible subsets $w$ in each iteration of the SMO algorithm is quite crucial. To this end, note that—due to the KKT conditions for the soft margin problem—we have

$$\mathrm{KKT}_i := r_i \xi_i + \beta_i \left(y_i f(\boldsymbol{x}_i) - 1 + \xi_i\right) = 0 \quad \forall\, i = 1, \ldots, n.$$

One can show that in our case

$$\mathrm{KKT}_i = (C - \beta_i) \max\left(0, 1 - y_i f(\boldsymbol{x}_i)\right) + \beta_i \max\left(0, y_i f(\boldsymbol{x}_i) - 1\right).$$

The following theorem [OFG97] helps us to identify possible candidates for $w$.

**Theorem 3.3.1** (Osuna, Freund, Girosi 1997)**.** *If we choose $w = \{i, j\} \subset \{1, \ldots, n\}$ such that $KKT_l \neq 0$ for any $l \in w$ for each SMO step, the SMO algorithm, i.e., the successive iteration of SMO steps with Algorithm 2, converges to the solution of* (3.8)*.*

Therefore, when we choose the working sets according to the non-vanishing KKT conditions from the theorem above, we are guaranteed that the SMO algorithm, i.e., taking iterative steps by running Algorithm 2, converges to the true solution of (3.8).

**Algorithm 2:** OneStep algorithm performing a single iteration of SMO to update the coefficients $\beta_i, \beta_j$ and the bias $\gamma_0$ of $f(\cdot) = \sum_{l=1}^{n} \beta_l y_l \langle \cdot, \boldsymbol{x}_l \rangle_\Omega + \gamma_0$. Here, $\langle \cdot, \cdot \rangle_\Omega$ is the inner product on $\Omega \subset \mathbb{R}^d$.

**Input:** indices $i, j \in \{1, \ldots, n\}$.
**Output:** updates of $\beta_i$, $\beta_j$, and $\gamma_0$.

$\delta \leftarrow y_i \left(y_i - f(\boldsymbol{x}_i) - (y_j - f(\boldsymbol{x}_j))\right)$.
$s \leftarrow y_i \cdot y_j$.
$\chi \leftarrow \langle \boldsymbol{x}_i, \boldsymbol{x}_i \rangle_\Omega + \langle \boldsymbol{x}_j, \boldsymbol{x}_j \rangle_\Omega - 2 \cdot \langle \boldsymbol{x}_i, \boldsymbol{x}_j \rangle_\Omega$.
$\vartheta \leftarrow s\beta_i + \beta_j$.
**if** $s = 1$ **then**
  $\mathrm{Lo} \leftarrow \max(0, \vartheta - C)$.
  $\mathrm{Up} \leftarrow \min(\vartheta, C)$.
**else**
  $\mathrm{Lo} \leftarrow \max(0, -\vartheta)$.
  $\mathrm{Up} \leftarrow \min(C, C - \vartheta)$.
**end if**
**if** $\chi > 0$ **then**
  $\beta_i \leftarrow \min\left(\max\left(\beta_i + \frac{\delta}{\chi}, \mathrm{Lo}\right), \mathrm{Up}\right)$.
**else if** $\delta \geq 0$ **then**
  $\beta_i \leftarrow \mathrm{Up}$.
**else**
  $\beta_i \leftarrow \mathrm{Lo}$.
**end if**
$\beta_j \leftarrow \vartheta - s\beta_i$.
Update function evaluations $f(\boldsymbol{x}_l), l = 1, \ldots, n$.
$\gamma_0 \leftarrow \gamma_0 - \frac{1}{2}(f(\boldsymbol{x}_i) - y_i + f(\boldsymbol{x}_j) - y_j)$.

## 3.4 ▪ Tasks on (linear) support vector machines

**Task 3.1.** Implement the function `OneStep` from Algorithm 2, which takes one iterative step of the SMO algorithm for two selected indices $i$ and $j$.

**Task 3.2.** To have a data set on which we can test our algorithm, draw 20 two-dimensional vectors according to an exponential distribution with parameter value 4 in each of the coordinate directions, i.e., the $j$th coordinate of the $i$th vector is drawn i.i.d. according to $[\boldsymbol{x}_i]_j \sim \exp(4)$ for all $i = 1, \ldots, 20$ and $j = 1, 2$. Assign the label $-1$ to these $\boldsymbol{x}_i$. Then, draw 20 two-dimensional vectors according to $\exp(0.5)$ in the same way and assign the label 1 to them.

**Task 3.3.** Implement a function `SMO` that initializes $\boldsymbol{\beta} = 0$ and $\gamma_0 = 0$, then—in each iteration step—randomly picks $i, j \in \{1, \ldots, n\}$ such that $i \neq j$, and calls `OneStep` with indices $i, j$ to perform an optimization. Use it to complete the following tasks:

(a) After the last iteration step, we need to compute a final estimate for $\gamma_0$. To this end, calculate the mean $\boldsymbol{m}$ of $f(\boldsymbol{x}_k) - y_k$ for all indices $k \in \{1, \dots, n\}$ for which $0 < \beta_k < C$. Then, set $\gamma_0 \leftarrow \gamma_0 - \boldsymbol{m}$.

(b) Run the SMO function with $10,000$ iteration steps to compute a support vector regressor $\tilde{f}$ according to (3.7), which maximizes (3.8) for the $n = 40$ data points from Task 3.2. Compute the results for $C = 0.01$, $C = 1$, and $C = 100$. For each $C$, plot the scattered data and compute the hyperplane corresponding to $\tilde{f} = 0$, i.e., compute the hyperplane corresponding to the level set classifier $f = f_{\text{class}}$; see (2.4). Compare your results to the separating hyperplane computed by the linear least squares algorithm implemented in Task 2.4.

(c) Count the number of support vectors. Mark the corresponding $\boldsymbol{x}_k$ in your scattered data plot.

(d) Furthermore, also count the number of *margin defining vectors*, i.e., the number of indices $k \in \{1, \dots, n\}$ for which $C > \beta_k > 0$, and mark the corresponding $\boldsymbol{x}_k$ in the scattered data plot. An example for such a plot can be found in Figure 3.3.

What influence does the parameter $C$ have on the number of the support vectors and on the position of the separating hyperplane?

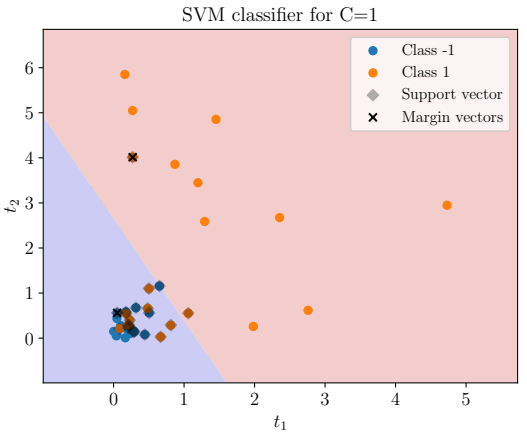



Figure 3.3: Support vector classifier for $C = 1$.

Now let us check how our classifiers perform if we evaluate them on some test data.

**Task 3.4.** Draw $2,000$ test data points according to the distributions from Task 3.2 ($1,000$ points for class $-1$ and $1,000$ points for class 1). Evaluate the accuracy (percentage of correctly classified data points; see (2.5)) for the LLS and SVM models calculated in Task 3.3.

The random picks of $i, j$ in the SMO algorithm can be very ineffective for large data sets. Therefore, we have to come up with a better approach to choose appropriate indices in each step of the SMO algorithm. There exist many heuristics to choose suitable indices in each step;

see [CL11, Pla98, SS02]. We will employ the *Karush–Kuhn–Tucker* terms $\mathrm{KKT}_i$ of the dual minimization problem.

**Task 3.5.** Repeat Task 3.3 and Task 3.4, but, instead of drawing the indices $i, j$ for each SMO-step randomly, write an outer loop that iterates over all $i \in \{1, \dots, n\}$ and check if $\mathrm{KKT}_i > 0$. If this is the case, randomly pick a $j \neq i$ for which $0 < \beta_j < C$. If no such $j$ exists, randomly pick a $j \in \{1, \dots, n\} \setminus \{i\}$. Subsequently, run the `OneStep` function for the pair $(i, j)$. If $\mathrm{KKT}_i = 0$ for each $i$ or if the maximum number of `OneStep` calls $(10,000)$ is reached, the algorithm terminates. Compare the results achieved with this heuristic with the results achieved by randomly picking $i$ and $j$. How do their runtimes compare?

## 3.5 ▪ Nonlinear support vector machines

A major drawback of the linear least squares approach and the support vector machines above is the fact that the resulting functions are linear. However, in cases where the distribution of the input data is such that a linear hyperplane is not suitable to classify the data, it is advantageous to consider nonlinear approaches. An example of such a situation is given in Figure 3.4.

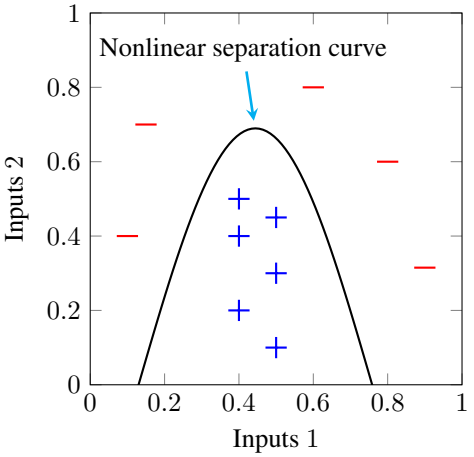



Figure 3.4: An example of a two-dimensional data set that is not linearly separable. However, there exists a nonlinear separation curve.

### 3.5.1 ▪ Feature maps

To create nonlinear support vector machines, we can simply apply a suitable feature map; compare also Section 1.5. To this end, we employ a function $\phi : \Omega \subset \mathbb{R}^d \to \mathbb{R}^{\tilde{d}}$ (with $\tilde{d} > d$) that transforms the nonlinearly separable input data in $\mathbb{R}^d$ in such a way that they become linearly separable in $\mathbb{R}^{\tilde{d}}$; see, e.g., Figure 3.5. In this way, we can preprocess the input data by applying $\phi$ and then run a linear SVM in $\mathbb{R}^{\tilde{d}}$. Note that such a feature map is not necessarily unique. Indeed, for the example from Figure 3.5, $\phi(s,t) = (s,t,g(s^2+t^2))$ is also a possible candidate for any monotonically increasing or monotonically decreasing $g$. The theoretical motivation behind employing a feature map to obtain linearly separable data is *Cover's theorem* [Cov65].

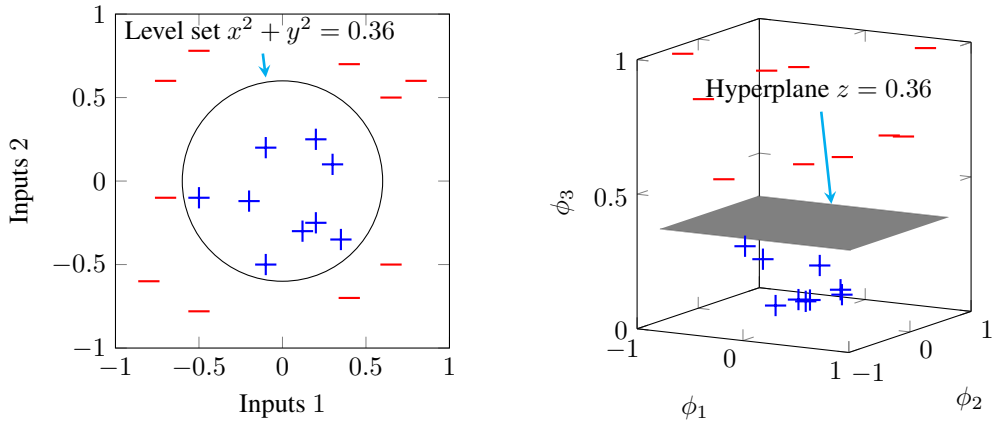



Figure 3.5: Example for applying a feature map to a two-dimensional nonlinearly separable data set (left) such that it becomes linearly separable in three dimensions (right). Here, $\Omega = [-1, 1]^2$, and the feature map $\phi : \mathbb{R}^2 \to \mathbb{R}^3$ is given by $\phi(s, t) := (s, t, s^2 + t^2)^T$.

**Theorem 3.5.1** (Cover 1965)**.** *There are $C(n, d)$ many homogeneously linearly separable dichotomies of $n$ points in general position in $\mathbb{R}^d$, where*

$$C(n, d) := 2 \sum_{k=1}^{d-1} \binom{n-1}{k}.$$

Let us have a closer look at the statement of Cover's theorem. To this end, let us first explain the terminology being used.

- A *dichotomy* of a set $X$ refers to a splitting of $X$ into two mutually exclusive subsets $Y$ and $Z$, i.e., $Y \cap Z = \emptyset$ and $Y \cup Z = X$.
- A dichotomy is called *homogeneously linearly separable* if there exists a homogeneous linear separator, i.e., if there exists a linear function[15] $f(\boldsymbol{x}) := \boldsymbol{\gamma}^T \boldsymbol{x}$ whose 0-level set separates the dichotomy.
- A set of points $X \subset \mathbb{R}^d$ is in *general position* if any subset of $d$ (or fewer) points is linearly independent.

Note that the condition of the data points being in general position in $\mathbb{R}^d$ might not be true for real-world data sets with a lot of correlations between data points. However, for randomly drawn data sets this is valid with probability 1.

For us, the most important conclusion to draw from Cover's theorem is the fact that, for fixed $n$, the number of dichotomies that are linearly separable grows when the dimension $d$ grows. Thus, it is more likely that the data are linearly separable after being mapped into a higher-dimensional ambient space. This is the motivation for using a feature map $\phi : \mathbb{R}^d \to \mathbb{R}^{\tilde{d}}$ with $\tilde{d} > d$. However, we cannot draw any conclusions from Cover's theorem for a specific data set at hand. Thus, the careful construction of a suitable feature map to obtain linearly separable data in $\mathbb{R}^d$ is still an important task.

[15] Note that this has to be understood as a truly linear separator in contrast to an affine linear separator $f(\boldsymbol{x}) := \boldsymbol{\gamma}^T \boldsymbol{x} + \gamma_0$ with $\gamma_0 \neq 0$.

**Task 3.6.** Generate 50 uniformly distributed i.i.d. points that lie in $\{\boldsymbol{t} \in \mathbb{R}^2 \mid \|\boldsymbol{t}\|_2 < 1\}$ (e.g., by drawing uniformly distributed points in $(-1,1)^2$ until 50 of them are within the unit sphere) and label them by $-1$. Now generate 50 data points, which are uniformly distributed in $\{\boldsymbol{t} \in \mathbb{R}^2 \mid 1 < \|\boldsymbol{t}\|_2 < 2\}$, and label them by 1.

(a) Fit a linear SVM for $C = 10$ to the data and plot the scattered data as well as the separating hyperplane.

(b) Transform the data by the feature map $\phi : \mathbb{R}^2 \to \mathbb{R}^3$ defined as

$$\phi(\boldsymbol{t}) := \left(t_1, t_2, t_1^2 + t_2^2\right).$$

Fit an SVM for $C = 10$ to the transformed data. Depict the scattered data and the nonlinear separation curve in a 2D plot (i.e., in the same way as in (a)). What does the feature map do, and why does it work so well?

### 3.5.2 ▪ Kernel trick

Note that—in the dual formulation (3.8) of the SVM—the input vectors $\boldsymbol{x}_i$ only appear in the scalar products $\boldsymbol{x}_i^T \boldsymbol{x}_j$. Thus, using a feature map $\phi : \mathbb{R}^d \to \mathbb{R}^{\tilde{d}}$, we can define the so-called *kernel* $K : \Omega \times \Omega \to \mathbb{R}$ corresponding to $\phi$ by

$$K(\boldsymbol{x}_i, \boldsymbol{x}_j) := \phi(\boldsymbol{x}_i)^T \phi(\boldsymbol{x}_j).$$

Now, instead of constructing a feature map $\phi$ explicitly, we can directly work with a kernel $K$ since it realizes the inner products needed to solve (3.8). The approach of substituting Euclidean products of the data by kernel evaluations is usually referred to in the machine learning community as the so-called *kernel trick* [SS02], which is based on the results of Mercer [Mer09]. The kernel trick is often applied to transform a nonlinearly separable data set to a linearly separable data set in higher dimensions. Let us now consider which kernels are suitable to describe inner products of feature maps.

**Positive (semi-)definiteness and Mercer kernels**

A kernel function $K : \Omega \times \Omega \to \mathbb{R}$ is called symmetric if $K(\boldsymbol{t}, \boldsymbol{s}) = K(\boldsymbol{s}, \boldsymbol{t})$ for all $s, t \in \Omega$. We say that it is positive semi-definite if

$$\sum_{i,j=1}^{N} \zeta_i \zeta_j K(\boldsymbol{t}_i, \boldsymbol{t}_j) \geq 0 \quad \forall\, \boldsymbol{t}_i \in \Omega, \zeta_i \in \mathbb{R}, i = 1, \ldots, N$$

holds for arbitrary $N \in \mathbb{N}$. If the inequality is strict, we say that the kernel is positive definite. A positive semi-definite, continuous, and symmetric kernel which is $L_2$ integrable with respect to the marginal data measure $\mu_X$ on $\Omega$, i.e., $K \in L_{2,\mu_X}(\Omega \times \Omega)$, is called a ***Mercer kernel***; see also [Mer09, RW06, SS02].

We will observe that there is a close relationship between Mercer kernels and feature maps. This can be exploited to implicitly build suitable feature maps for SVMs. The corresponding feature space is known to be a so-called *reproducing kernel Hilbert space*.

**Reproducing kernel Hilbert spaces**

There is a one-to-one relation between symmetric, positive-definite kernel functions and certain Hilbert spaces. Let $H$ be a Hilbert space of real-valued functions on $\Omega$. Let furthermore the point evaluation functionals $p_{\boldsymbol{t}} : H \to \mathbb{R}$ defined by $p_{\boldsymbol{t}}(f) = f(\boldsymbol{t})$ be bounded operators on $H$ for all $\boldsymbol{t} \in \Omega$. Then, the space $H$ is called a ***reproducing kernel Hilbert space*** (RKHS); see, e.g., [Aro50, RW06, SS16, SS02]. The reason for the name RKHS is the fact that there exists a symmetric, positive-definite kernel function $K : \Omega \times \Omega \to \mathbb{R}$, called ***reproducing kernel***, such that $K(\boldsymbol{t}, \cdot) \in H$ and

$$\langle f, K(\boldsymbol{t}, \cdot)\rangle_H = f(\boldsymbol{t}) \quad \text{(reproducing property)}$$

for all $\boldsymbol{t} \in \Omega$. Here, $\langle \cdot, \cdot \rangle_H$ denotes the inner product associated with $H$. The converse to the above statement is also true due to the Moore–Aronszajn theorem [Aro50]: For every symmetric positive definite kernel $K$, there exists a unique RKHS $H$ to which $K$ is the reproducing kernel.

### 3.5.3 ▪ Mercer's theorem

The following theorem gives an important characterization of Mercer kernels; see [Mer09, SS02, Vap99].

**Theorem 3.5.2** (Mercer 1909)**.** *Let $\Omega \subseteq \mathbb{R}^d$ be closed and let $\mu_X$ be a strictly positive Borel measure on $\Omega$, i.e., $\mu_X(U) > 0$ for all open $\emptyset \neq U \subseteq \Omega$. Let $K : \Omega \times \Omega \to \mathbb{R}$ be a Mercer kernel. Then, there exists a sequence of numbers $(\lambda_i)_{i=1}^{\infty} \subset [0, \infty]$ and $L_{2,\mu_X}$-orthonormal functions $(\Phi_i)_{i=1}^{\infty}$ such that*

$$K(\boldsymbol{s}, \boldsymbol{t}) = \sum_{i=1}^{\infty} \lambda_i \Phi_i(\boldsymbol{s}) \Phi_i(\boldsymbol{t}),$$

*where the convergence is uniform on $\Omega$ and absolute.*

To see how this theorem helps, let us define a feature map $\phi : \Omega \to \ell_2(\mathbb{R})$ by

$$\phi(\boldsymbol{x}) := \left(\sqrt{\lambda_1}\Phi_1(\boldsymbol{x}), \sqrt{\lambda_2}\Phi_2(\boldsymbol{x}), \ldots\right)$$

with the $\lambda_i$ and $\Phi_i$, $i \in \mathbb{N}$, from Mercer's theorem. Then, the theorem tells us that

$$K(\boldsymbol{s}, \boldsymbol{t}) = \langle \phi(\boldsymbol{s}), \phi(\boldsymbol{t})\rangle_{\ell_2(\mathbb{R})}.$$

A Mercer kernel thus implicitly defines a feature map into "$\mathbb{R}^{\infty}$" or, more accurately, into the sequence space $\ell_2(\mathbb{R})$. Therefore, we can simply take a Mercer kernel $K$ and substitute the inner products $\boldsymbol{x}_i^T \boldsymbol{x}_j$ in (3.8) by $K(\boldsymbol{x}_i, \boldsymbol{x}_j)$ for all $i, j = 1, \ldots, n$.

Furthermore, the solution of (3.8) can now be written as the finite sum

$$f(\boldsymbol{x}) = \gamma_0 + \sum_{i=1}^{n} \beta_i y_i K(\boldsymbol{x}_i, \boldsymbol{x}). \tag{3.10}$$

This fact is known as the so-called *representer theorem*; see, e.g., [KW70, SS02]. It gives a clear advantage over the *primal* representation

$$f(\boldsymbol{x}) = \gamma_0 + \boldsymbol{\gamma}^T \phi(\boldsymbol{x}), \tag{3.11}$$

where we have to deal with the (potentially) infinite-dimensional vectors $\boldsymbol{\gamma}$ and $\phi(\boldsymbol{x})$.

Famous examples of Mercer kernels (with respect to the Lebesgue measure $\mu_X$) are

- the polynomial kernel

$$K_q(\boldsymbol{s}, \boldsymbol{t}) := (\boldsymbol{s}^T \boldsymbol{t} + c)^q \quad \text{for some } q \geq 2 \text{ and}$$

- the Gaussian (or radial basis function (RBF)) kernel

$$K_\sigma(\boldsymbol{s}, \boldsymbol{t}) := \exp\left(-\frac{\|\boldsymbol{s} - \boldsymbol{t}\|^2}{2\sigma^2}\right) \quad \text{for some } \sigma > 0.$$

For further examples of Mercer kernels used in machine learning and the analysis thereof, we refer the interested reader to [SW06, Wah90, Wen95].

Note that, besides the constrained SVM optimization problem (3.8) discussed in Section 3.2, there exists another way to compute the SVM solution using the formulation (3.10). To this end, a so-called *hinge loss* function is employed to measure the error between $f(\boldsymbol{x}_i)$ and $y_i$. Furthermore, the RKHS norm of $f$ corresponding to the chosen kernel is used in a weighted regularization term; see [SS02] for details. This can be seen as a generalization of (1.6) with a different loss function and regularization term.

**Task 3.7.** Change your SMO code such that it allows you to use a kernel function instead of the scalar product of the input data, i.e., substitute all scalar products by the evaluation of the kernel function. Perform an SVM classification ($C = 10$) with Gaussian kernel ($\sigma = 1$) for the data from Task 3.6. Depict the scattered data and the nonlinear separation curve in a 2D plot. The result should look similar to Figure 3.6.

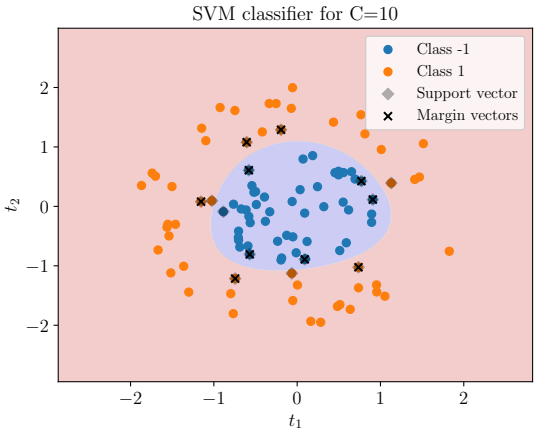



Figure 3.6: Support vector classifier with Gaussian kernel for $C = 10, \sigma = 1$.

## 3.6 ▪ Hyperparameter fitting

As introduced in Section 1.4, so-called *hyperparameters* are often encountered in learning algorithms, i.e., parameters that have to be fixed by the user before applying a method. Examples are

the regularization parameter $C$ for soft-margin classifiers (see (3.8)), and the kernel parameters $q$ for the polynomial kernel and $\sigma$ for the RBF kernel.

A common method to determine the optimal hyperparameters $\boldsymbol{p} \in P$ from a finite set $P \subset \mathbb{R}^{d_H}$ of potential parameter configurations is *grid search* with either a single *validation set* or *cross-validation* (CV). Here, $d_H \in \mathbb{N}$ is the number of hyperparameters that need to be chosen and $|P|$ is the number of possible/suitable hyperparameter combinations. Oftentimes, $P = \otimes_{i=1}^{d_p} P_i$ is chosen, where $P_i$ contains suitable values for the $i$th hyperparameter, $i = 1, \ldots, d_H$. However, if $d_H$ is large, we encounter the curse of dimensionality in the size of $P$; see also Section 1.6. Then, it makes sense to choose a significantly smaller set of possible configurations $P$, e.g., by so-called *Monte Carlo* or *quasi Monte Carlo* methods; see [BB12].

To look for the best $\boldsymbol{p} \in P$, we compute values of an evaluation measure, e.g., the accuracy, for the results of our ML algorithm for all $|P|$ possible choices of hyperparameters (*grid search*). To obtain representative values, we should not use parts of the training data for this purpose. Instead, a *validation set* is usually employed. To this end, the underlying idea is to partition the original data into two disjoint sets, one training data set and one *artificial test data set/validation set*. We then train the $|P|$ different models on the training data set and measure their performance on the validation set. Finally, the best hyperparameters $\boldsymbol{p} \in P$ are those for which the evaluation measure on the validation set was best. Commonly, the data is split by using roughly 80% for training the model and 20% for validating the model. Note that we could also employ other quality measures instead of the accuracies; see Section 2.2.

While using grid search with a validation set often works well, small data sets can lead to unreliable estimates. In these cases, *$k$-fold cross-validation* is employed. Here, several validation sets are used instead of only one. More precisely, the original data set is randomly split into $k$ parts, also called *folds*, of approximately equal size. One fold is chosen as the validation set while the remaining $k-1$ folds serve as training data for our algorithm. Subsequently, we take a different fold as validation data and the rest as training data and repeat the process $k$ times until each fold has been used as validation data once. The (arithmetic) average of the $k$ accuracies calculated on the validation data then serves as the quality measure. Typically, $k = 5$ or $k = 10$ is recommended; see [HTF09].

Now, to determine the best choice of hyperparameters, we choose small candidate sets. For example, for the two hyperparameters of SVM with a Gaussian kernel, namely the regularization parameter $C$ from (3.8) and the Gaussian kernel bandwidth $\sigma$, we choose $C \in \{0.01, 0.1, 1, 10, 100\}$ and $\sigma \in \{1, 10, 100\}$ and run a $k$-fold cross-validation for all possible combinations of parameter pairs. The pair $(C, \sigma)$ with the best average accuracies in the cross-validation process is the winner. The corresponding pseudocode can be found in Algorithm 3. Note that it is absolutely crucial to avoid using the test data during the model development phase, i.e., when tweaking hyperparameters or changing model parts. Otherwise, overfitting effects might be present and there is no chance to detect them on the test data. More details on this approach and variants thereof (e.g., leave-one-out cross-validation) can be found in [HTF09, JWHT21, Mur22, SS02], for example.

After determining the optimal parameters $\boldsymbol{p}_{\text{best}} \in P$ as described in Algorithm 3, we can build a model on the whole training data set $\mathcal{D}$ with parameters $\boldsymbol{p}_{\text{best}}$ and evaluate it on the real, unseen test data $\bar{\mathcal{D}}$.

## 3.7 ▪ Image classification with support vector machines

We will now apply a support vector machine to a famous benchmark classification problem. We consider the task of classifying images of handwritten digits from the so-called *MNIST* data set.

**Algorithm 3:** Abstract $k$-fold cross-validation scheme

**Input:** $k \in \mathbb{N}$, training data $\mathcal{D}$, possible combinations of hyperparameters $P$.

Randomly split $\mathcal{D}$ into $k$ parts $\mathcal{D}_1, \ldots, \mathcal{D}_k$ of (almost) equal size.
**forall** $\boldsymbol{p} \in P$ **do**
  **forall** $i = 1, \ldots, k$ **do**
    Run ML algorithm with input data $\cup_{j \neq i} \mathcal{D}_j$ and parameters $\boldsymbol{p}$.
    Evaluate resulting model on $\mathcal{D}_i$ and store accuracy $A_i$.
  **end forall**
  Average over the accuracies: $A^{\boldsymbol{p}} \leftarrow \frac{1}{k} \sum_{i=1}^{k} A_i$.
**end forall**
Determine $\boldsymbol{p}_{\text{best}} \leftarrow \arg\max_{\boldsymbol{p} \in P} A^{\boldsymbol{p}}$.

### 3.7.1 ▪ The MNIST data set

The MNIST data set (http://yann.lecun.com/exdb/mnist/) consists of $70,000$ gray-scale images ($28 \times 28$ pixels) of handwritten digits. Four exemplary images can be found in Figure 3.7. Our goal will be to construct an algorithm that is able to identify the correct digit from an image of the handwritten one.

Figure 3.7: Four example images ($28 \times 28$ pixels) created from the MNIST data set.

You can download and extract the data set by hand, or you can use the following lines of PYTHON code. As you can see, you might need to install the `urllib` library. To this end, just run `pip install urllib3` in your shell.

```
 #Load MNIST Data
import os
import gzip
from urllib.request import urlretrieve

def download(filename, source='http://yann.lecun.com/exdb/mnist/'):
    print("Downloading %s" % filename)
    urlretrieve(source + filename, filename)

def load_mnist_images(filename):
    if not os.path.exists(filename):
        download(filename)
    with gzip.open(filename, 'rb') as f:
        data = np.frombuffer(f.read(), np.uint8, offset=16)
    data = data.reshape(-1, 28, 28)
    return data / np.float32(256)

def load_mnist_labels(filename):
    if not os.path.exists(filename):
```

```
        download(filename)
    with gzip.open(filename, 'rb') as f:
        data = np.frombuffer(f.read(), np.uint8, offset=8)
    return data

X_train = load_mnist_images('train-images-idx3-ubyte.gz')
y_train = load_mnist_labels('train-labels-idx1-ubyte.gz')
X_test = load_mnist_images('t10k-images-idx3-ubyte.gz')
y_test = load_mnist_labels('t10k-labels-idx1-ubyte.gz')
```

### 3.7.2 ▪ Multi-class learning

Up to now, we always considered classification problems, where our label set $\Gamma$ was of size two, i.e., we just had two different classes. In real-world applications one often encounters so-called *multi-class* classification problems, where $|\Gamma| > 2$. The above handwritten digits data classification problem is an example with $\Gamma = 10$. In this case, a common idea is to use $\frac{|\Gamma|(|\Gamma|-1)}{2}$ pairwise classifiers, i.e., classifiers to distinguish between each possible pair $\gamma_1 \neq \gamma_2$ of classes in $\Gamma$. To decide in which class a data point $\boldsymbol{x}$ lies, each pairwise classifier is evaluated and the class $\gamma \in \Gamma$ to which $\boldsymbol{x}$ is assigned the most is the one that wins. In this way, we can apply standard two-class algorithms to solve multi-class problems. We refer the reader to [HTF09, JWHT21, Mur22, SS02] for more details on different approaches to multi-class problems.

### 3.7.3 ▪ SCIKIT-LEARN: A machine learning library in PYTHON

For the sake of understanding the basic programming and machine learning paradigms, we did (and will) implement the learning algorithms on our own. However, we will also study how to use important PYTHON machine learning libraries such as SCIKIT-LEARN (http://scikit-learn.org). This is an efficient and easy-to-use library in which we can find variants of all algorithms that we have discussed so far (LLS, $k$-NN, SVM) and many more. Note that the SVM method of SCIKIT-LEARN is using LIBSVM [CL11], which is an implementation of an SMO-type decomposition algorithm with a working set selection using second order information. For a linear kernel, a more efficient variant based on a coordinate descent approach for the dual problem [FCH+08] is also available.

**Task 3.8.** Make yourself familiar with the `SVC` function in SCIKIT-LEARN, which implements a support vector classifier.

(a) Choose a random subset of size 500 from the MNIST training data and use it as your new training data set for cross-validation. Perform a 5-fold cross-validation SVM to determine the optimal parameters among $C \in \{1, 10, 100\}$ and $\gamma = \frac{1}{2\sigma^2} \in \{0.1, 0.01, 0.001\}$. (Hint: You can use the SCIKIT-LEARN function `GridSearchCV`.)

(b) Use the determined optimal parameters to learn a support vector classifier on a random $2,000$ point subset of the MNIST training data and evaluate the confusion matrix and the accuracy on the whole MNIST test data set. (Hint: You can use the SCIKIT-LEARN module `metrics`.)

(c) Predict the labels for the first 200 data points of the test data with the classifier trained in (b). Print out the predicted label and the true label for all misclassified instances and visualize them, i.e., plot the corresponding digit images.

## 3.8 ▪ Further topics

**Non-numerical data** Note that the feature map approach allows us to also classify data that do not reside in Euclidean space by building appropriate feature maps that assign a value (in $\mathbb{R}^{\tilde{d}}$) to each element of the input data. This is often useful when it comes to practical applications where data are not directly given as numerical values or vectors; see also Section 10.1. Also in the case of structured data, specially tailored kernels (or feature maps) exist; see, e.g., [KJM20] for kernels on graph-based data structures.

**Kernel choice** We have not discussed how an appropriate kernel type, e.g., Gaussian, polynomial, etc., for the task at hand can be chosen. If we have some a priori problem knowledge (such as smoothness of the "true" separation function), for example, we can exploit this in order to choose an appropriate kernel type; see also [CZ07]. This is an *informed* machine learning approach; see also [vRMB$^+$23]. Furthermore, the kernel type (and its optimal hyperparameters) can also be chosen by cross-validation over a finite set of fixed kernel functions, for instance. Moreover, so-called *multiple kernel learning* approaches make use of convex combinations of different kernel functions; see [GA11].

**Regression** The linear least squares and the $k$-nearest neighbors algorithms also apply to the regression case, where we look for a function $f$ such that $f(\boldsymbol{x}_i) \approx y_i$ and the $y_i$ can take arbitrary values in $\mathbb{R}$ instead of only discrete ones as in classification. However, for support vector machines this is not so straightforward since our optimization problem (3.8) originated from the optimal separating hyperplane formulation. Nevertheless, there exists a support vector machines regression algorithm based on the minimization of the so-called $\varepsilon$-insensitive loss function together with a weighted regularization term, which involves the norm of the solution in the corresponding kernel space; see [Mur22, SS02].

**Gaussian processes** Besides the SVM there is another famous example for using kernels in classification/regression methods, namely *Gaussian process regression*, also known as *kriging*. Here, the employed kernel is determined by the covariance matrix of the training data. With the help of this kernel, a stochastic process is defined for which every point evaluation is normally distributed with certain mean and covariance structure. Algorithmically, the minimization of a least squares loss function together with a weighted regularization term is computed there. For more details on this stochastically motivated regression method, we refer the reader to [Mur22, See04, RW06].

## Chapter 4

# Linear Dimensionality Reduction

One encounters large and nominally high-dimensional data sets in many real-world applications. However, the information in such data sets is often redundant in the sense that the different coordinates of a data set, stemming, e.g., from different physical measurements, are typically not independent but correlated. Thus, the high-dimensional representation of the data is not in a compact form. To determine a more effective representation of the data, we need to find a mapping into a lower-dimensional representation space that preserves the relevant information. But *relevant* can be understood in different ways, e.g., as preserving pairwise distances between data points or as allowing for almost lossless reconstruction of the original data set. The process of determining such a low-dimensional representation of the data is known as *dimensionality reduction*; see also Section 1.2.

Dimensionality reduction methods serve many purposes, e.g., to reduce the computational costs of a task due to the resulting smaller dimension or to gain new insights into inner structures of the data set. To this end, a good visualization is a powerful tool to develop an idea or intuition on what is happening. But the more dimensions our data has, the more challenging it is to visualize it. While finding a suitable visualization is a topic on its own, reducing the dimension can be a helpful start. A computational reason for reducing the dimension is scaling behavior. The runtime of many algorithms scales polynomially or even exponentially in the dimension of the input data. Furthermore, some methods do not perform well in high dimensions due to the curse of dimensionality and the concentration of measure effect; see Section 1.6. Therefore, dimensionality reduction as a preprocessing step can provide more expressive features for a learning algorithm; see Section 1.5. While the main object of interest is the representation/projection of the high-dimensional data into the low-dimensional space, we are sometimes also interested in the representation system itself. This is, for example, the case if we also want to deal with new data, e.g., when projecting new test data.

A difference in methodology in this chapter compared to the previous chapters is that a straightforward evaluation criterion for a given solution is usually not directly available. While a good solution for a regression problem has to provide a good approximation to the underlying data measure $\mu$, there is often no single *best* answer in dimensionality reduction. Two representations could capture or highlight different aspects of the data and might be useful for different applications. One could say that the criterion of "preserving information" or "reducing redundancy" leaves more space for different objective functions to minimize and also depends on the task at hand.

This chapter is structured as follows. Section 4.1 introduces the *principal component analysis* (PCA), which is the most popular and most commonly used dimensionality reduction method,

even today. In Section 4.2 we discuss the connection between the PCA and the SVD, which we already studied in Section 2.1. Subsequently, we give interpretations of the PCA as an algorithm working on distances between data points in Section 4.3 and as a statistical method to preserve the data variance as well as possible in Section 4.4. Finally, Section 4.5 presents tasks on a pedestrian detection problem tackled with the PCA.

## 4.1 ▪ Principal component analysis

In the case of linear dimensionality reduction, we project the high-dimensional data points in $\mathbb{R}^d$ onto a linear subspace. Suppose we conducted an experiment with $n$ i.i.d. measurements in the two correlated variables $A_1$ and $A_2$, i.e., we are given $d = 2$-dimensional points $\boldsymbol{x}_i$, where the coordinates of each measurement are random draws of $A_1$ and $A_2$ but the 2D data points are highly correlated. An example for such data is plotted in Figure 4.1. The correlation of the variables can be seen clearly. In this case, we can (approximately) reduce the dimension to one and project data points onto the blue line determined by the blue arrow. Then, the one-dimensional values will correspond to points on this line.

The most famous and most frequently employed dimensionality reduction method is the so-called *principal component analysis*. Note that this method is also known under different names depending on the field of application in which data analysis is applied: It is known, e.g., as *Karhunen–Loéve decomposition* [KS90] in stochastics and image processing, as *proper orthogonal decomposition* [BHL93, Lum67] in the analysis of physical fields, and as *empirical orthogonal functions* (EOF) [Lor56] in meteorology, among others. From an abstract mathematical point of view, it is related to the *singular value decomposition* [GR70] and, thus, to the eigenvalue decomposition of a matrix in linear algebra. Note that we already encountered the SVD in the context of linear least squares regression in Section 2.1.

**Principal component analysis (PCA)**

The first formulation of the ***principal component analysis*** goes back to Pearson, who derived it already in 1901 [Pea01]. Here, the low-dimensional linear subspace, onto which the input data are projected, is determined by solving a linear least squares-type problem. To this end, a (vector-valued) affine linear function $f : \mathbb{R}^q \to \mathbb{R}^d$ is fitted to the input data $\boldsymbol{x}_i \in \mathbb{R}^d$, $i = 1, \ldots, n$, by minimizing the least squares loss between the $f(\boldsymbol{\eta}_i)$ values and the $\boldsymbol{x}_i$ values, i.e.,

$$\min_{f, \boldsymbol{\eta}_i \in \mathbb{R}^q} \frac{1}{n} \sum_{i=1}^{n} \|\boldsymbol{x}_i - f(\boldsymbol{\eta}_i)\|_2^2.$$

Here, the minimization is done not only with respect to the degrees of freedom of $f$, but also over the low-dimensional variables $\boldsymbol{\eta}_i$. One of the major benefits of PCA is that the solution can be quite easily computed by an eigendecomposition of the covariance matrix of the data. More detailed discussions of this algorithm can be found in [Hot33, Jol02, LV07].

Note that there are different derivations of the PCA approach, just as there are different names for it. As briefly mentioned, we consider an affine linear model

$$f \in \mathcal{M}_{\text{Lin}}^{q,d} := \left\{ g : \mathbb{R}^q \to \mathbb{R}^d \;\middle|\; g(\boldsymbol{\eta}) = \boldsymbol{\nu} + \boldsymbol{V}_q \boldsymbol{\eta} \text{ with } \boldsymbol{\nu} \in \mathbb{R}^d, \boldsymbol{V}_q \in S_q(\mathbb{R}^d) \right\}, \tag{4.1}$$

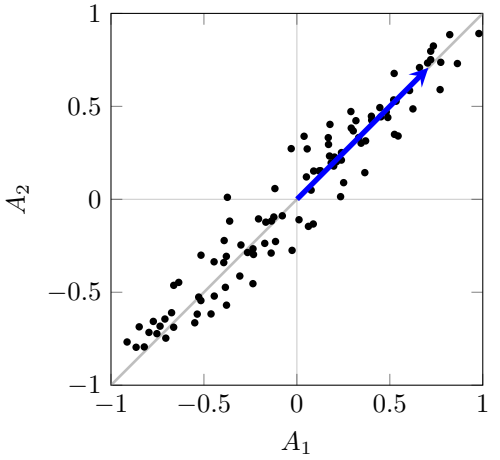



Figure 4.1: Highly correlated two-dimensional data. The blue arrow indicates the direction of the one-dimensional linear subspace along which the data vary the most.

where

$$S_q(\mathbb{R}^d) := \left\{ \boldsymbol{V} \in \mathbb{R}^{d\times q} \;\middle|\; \boldsymbol{V}^T\boldsymbol{V} = \boldsymbol{I} \right\}$$

is the so-called *Stiefel manifold* of $d \times q$ matrices with orthonormal columns. Note that the class $\mathcal{M}_{\text{Lin}}^{q,d}$ is the vector-valued analogue to the class $\mathcal{M}_{\text{Lin}}$, which we encountered in the LLS setting; see Section 2.1. We want to fit $f$ to the data $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n \in \mathbb{R}^q$, i.e., we need to determine $\boldsymbol{\nu} \in \mathbb{R}^d$, $\boldsymbol{V}_q \in \mathbb{R}^{d\times q}$, and $\boldsymbol{\eta}_i \in \mathbb{R}^q$ for $i = 1, \ldots, n$ such that

$$f(\boldsymbol{\eta}_i) = \boldsymbol{\nu} + \boldsymbol{V}_q\boldsymbol{\eta}_i \approx \boldsymbol{x}_i \quad \forall i = 1, \ldots, n. \tag{4.2}$$

As in the LLS setting, we use the least squares loss, but this time for vector-valued functions, i.e.,

$$f(\boldsymbol{\eta}_i)_{i=1}^n := \operatorname*{arg\,min}_{g\in\mathcal{M}_{\text{Lin}}^{q,d}, \boldsymbol{\eta}_i\in\mathbb{R}^q} \frac{1}{n}\sum_{i=1}^n \|\boldsymbol{x}_i - g(\boldsymbol{\eta}_i)\|_2^2. \tag{4.3}$$

Moreover, in contrast to the supervised learning setting, where the $\boldsymbol{\eta}_i$ would be known values, we have to determine them here. This is typical for an unsupervised problem; see Section 1.2. Note furthermore that the model class $\mathcal{M}_{\text{Lin}}^{q,d}$ is no longer convex. So we cannot expect to obtain a global minimizer by looking for critical points. We will now study how we can still obtain a solution to (4.3). To this end, we will take two steps: First, we minimize the right-hand side of (4.3) with respect to $\boldsymbol{\nu}$ and $\boldsymbol{\eta}_i$ for given $\boldsymbol{V}_q$. When $\boldsymbol{V}_q$ is fixed, the remaining optimization problem is convex again. Therefore, we only need to compute the corresponding gradients of the least squares function and set them to zero, i.e.,

$$\text{(i)} \sum_{i=1}^n (\boldsymbol{\nu} - \boldsymbol{x}_i + \boldsymbol{V}_q\boldsymbol{\eta}_i) = 0,$$
$$\text{(ii)} \; \boldsymbol{V}_q^T\boldsymbol{V}_q\boldsymbol{\eta}_i - \boldsymbol{V}_q^T(\boldsymbol{x}_i - \boldsymbol{\nu}) = \boldsymbol{\eta}_i - \boldsymbol{V}_q^T(\boldsymbol{x}_i - \boldsymbol{\nu}) = 0 \quad \text{for all} \quad i = 1, \ldots, n.$$

Thus, the $\boldsymbol{\eta}_i = \boldsymbol{V}_q^T(\boldsymbol{x}_i - \boldsymbol{\nu})$ are completely determined by (ii). A possibility to fulfill (i) is to set $\boldsymbol{\nu} := \boldsymbol{m} := \frac{1}{n}\sum_{i=1}^n \boldsymbol{x}_i$, i.e., $\boldsymbol{\nu}$ is set to the mean $\boldsymbol{m}$ of the data points. In a second step, we determine an appropriate matrix $\boldsymbol{V}_q$ that, together with the above values for $\boldsymbol{\eta}_i$ and $\boldsymbol{\nu}$,

solves (4.3). Plugging the values for $\boldsymbol{\nu}$ and $\boldsymbol{\eta}_i$ into the minimization problem (4.3), we are left with

$$\begin{aligned}\boldsymbol{V}_q &= \operatorname*{arg\,min}_{\boldsymbol{V}\in S_q(\mathbb{R}^d)} \frac{1}{n}\sum_{i=1}^{n}\left\|\boldsymbol{x}_i - \boldsymbol{m} - \boldsymbol{V}\boldsymbol{V}^T(\boldsymbol{x}_i - \boldsymbol{m})\right\|_2^2 \\ &= \operatorname*{arg\,min}_{\boldsymbol{V}\in S_q(\mathbb{R}^d)} \frac{1}{n}\sum_{i=1}^{n}\left\|\left(\boldsymbol{I} - \boldsymbol{V}\boldsymbol{V}^T\right)(\boldsymbol{x}_i - \boldsymbol{m})\right\|_2^2 \qquad \text{(PCAMin)} \\ &= \operatorname*{arg\,max}_{\boldsymbol{V}\in S_q(\mathbb{R}^d)} \frac{1}{n}\sum_{i=1}^{n}\left\|\boldsymbol{V}^T(\boldsymbol{x}_i - \boldsymbol{m})\right\|_2^2. \qquad \text{(PCAMax)}\end{aligned}$$

Note that $\boldsymbol{I} - \boldsymbol{V}_q\boldsymbol{V}_q^T$ is the orthogonal projector onto the orthogonal complement of $\operatorname{span}(\boldsymbol{V}_q)$. Thus, we are looking for the linear space spanned by the orthogonal columns of $\boldsymbol{V}_q$ such that the projection of the centered data $\boldsymbol{x}_i - \boldsymbol{m}$ onto this space has—on average—the largest Euclidean norm possible. Let $\boldsymbol{X} \in \mathbb{R}^{n\times d}$ be the *centered data matrix*[16] with rows $\boldsymbol{x}_i - \boldsymbol{m}$ for $i = 1, \ldots, n$, which can be computed by multiplying the so-called $n \times n$ *centering matrix* $\boldsymbol{H} = \boldsymbol{I} - \frac{1}{n}\mathbf{1}_n$ with the non-centered data matrix with row-wise entries $\boldsymbol{x}_i$, $i = 1, \ldots, n$, i.e.,

$$\boldsymbol{X} = \boldsymbol{H}\begin{pmatrix}\boldsymbol{x}_1 & \cdots & \boldsymbol{x}_n\end{pmatrix}^T. \tag{4.4}$$

Here, $\mathbf{1}_n$ is the $n \times n$ matrix in which all entries are 1. Furthermore, let the columns of $\boldsymbol{V}_q$ be $\boldsymbol{v}_i$ for $i = 1, \ldots, q$. Then, we have

$$\sum_{i=1}^{n}\|\boldsymbol{V}_q^T(\boldsymbol{x}_i - \boldsymbol{m})\|^2 = \sum_{i=1}^{n}\sum_{j=1}^{q}\left(\boldsymbol{v}_j^T(\boldsymbol{x}_i - \boldsymbol{m})\right)^2 = \sum_{j=1}^{q}\boldsymbol{v}_j^T\boldsymbol{X}^T\boldsymbol{X}\boldsymbol{v}_j = n\sum_{j=1}^{q}\boldsymbol{v}_j^T\boldsymbol{C}\boldsymbol{v}_j$$

with the empirical/data *covariance matrix* $\boldsymbol{C} := \frac{1}{n}\boldsymbol{X}^T\boldsymbol{X}$. Now let $\boldsymbol{C} = \boldsymbol{Z}\boldsymbol{\Lambda}\boldsymbol{Z}^T$ be an orthogonal diagonalization of $\boldsymbol{C}$, i.e., $\boldsymbol{Z}^{-1} = \boldsymbol{Z}^T$, with $\boldsymbol{\Lambda} = \operatorname{diag}(\lambda_1, \ldots, \lambda_d)$ with non-increasing values $\lambda_1 \geq \cdots \geq \lambda_d \geq 0$. Then

$$\begin{aligned}\boldsymbol{V}_q &= \operatorname*{arg\,max}_{\boldsymbol{V}\in S_q(\mathbb{R}^d)}\sum_{j=1}^{q}\boldsymbol{v}_j^T\boldsymbol{C}\boldsymbol{v}_j = \operatorname*{arg\,max}_{\boldsymbol{V}\in S_q(\mathbb{R}^d)}\sum_{j=1}^{q}\boldsymbol{v}_j^T\boldsymbol{Z}\boldsymbol{\Lambda}\boldsymbol{Z}^T\boldsymbol{v}_j \\ &= \operatorname*{arg\,max}_{\boldsymbol{V}\in S_q(\mathbb{R}^d)}\sum_{j=1}^{q}\tilde{\boldsymbol{v}}_j^T\boldsymbol{\Lambda}\tilde{\boldsymbol{v}}_j = \operatorname*{arg\,max}_{\boldsymbol{V}\in S_q(\mathbb{R}^d)}\sum_{i=1}^{d}\lambda_i\sum_{j=1}^{q}[\tilde{\boldsymbol{v}}_j]_i^2,\end{aligned}$$

where $[\tilde{\boldsymbol{v}}_j]_i$ denotes the $i$th entry of $\tilde{\boldsymbol{v}}_j$ and $\tilde{\boldsymbol{v}}_j := \boldsymbol{Z}^T\boldsymbol{v}_j$. Since $\boldsymbol{Z}$ and $\boldsymbol{V}_q$ are orthogonal matrices, we have $\sum_{j=1}^{q}[\tilde{\boldsymbol{v}}_j]_i^2 \leq 1$ for all $i = 1, \ldots, d$ and $\sum_{i=1}^{d}\sum_{j=1}^{q}[\tilde{\boldsymbol{v}}_j]_i^2 \leq q$. Thus, we obtain

$$\sum_{i=1}^{d}\lambda_i\sum_{j=1}^{q}[\tilde{\boldsymbol{v}}_j]_i^2 \leq \sum_{i=1}^{k}\lambda_i,$$

which shows that $\tilde{\boldsymbol{v}}_j = \boldsymbol{e}_j$ maximizes $\sum_{j=1}^{q}\tilde{\boldsymbol{v}}_j^T\boldsymbol{\Lambda}\tilde{\boldsymbol{v}}_j$. Therefore, the $j$th column of $\boldsymbol{V}_q$ is given by $\boldsymbol{v}_j = \boldsymbol{Z}\boldsymbol{e}_j$, which is just the $j$th column of $\boldsymbol{Z}^T$. Our results are summarized in the following theorem.

[16] Note that sometimes $\boldsymbol{X}^T$ is used instead of $\boldsymbol{X}$ in the literature.

**Theorem 4.1.1.** *Let $\boldsymbol{x}_1, \dots, \boldsymbol{x}_n \in \mathbb{R}^d$ and let $1 \leq q \leq d$ be a natural number. The minimization problem*

$$\min_{\substack{\boldsymbol{\nu} \in \mathbb{R}^d \\ (\boldsymbol{\eta}_i)_{i=1}^n \subset \mathbb{R}^q \\ \boldsymbol{V}_q \in S_q(\mathbb{R}^d)}} \sum_{i=1}^{n} \|\boldsymbol{x}_i - (\boldsymbol{\nu} + \boldsymbol{V}_q \boldsymbol{\eta}_i)\|^2$$

*is solved for $\boldsymbol{\nu} = \boldsymbol{m} = \frac{1}{n}\sum_{i=1}^n \boldsymbol{x}_i$ and $\boldsymbol{\eta}_i = \boldsymbol{V}_q^T(\boldsymbol{x}_i - \boldsymbol{m})$, where the columns of $\boldsymbol{V}_q$ are the orthogonal unit eigenvectors of $\boldsymbol{C} := \frac{1}{n}\boldsymbol{X}^T\boldsymbol{X}$ corresponding to the $q$ largest eigenvalues.*

The columns of $\boldsymbol{V}_q$ are called the $q$ first *principal axes* of the data $\boldsymbol{X}$, whereas the projections $\boldsymbol{\eta}_i$ are called *principal components*. While the above theorem shows us how to obtain the principal components, it can be problematic to compute the eigendecomposition of $\frac{1}{n}\boldsymbol{X}^T\boldsymbol{X}$ if the dimension $d$ of the data is quite large. Furthermore, the costs of computing $\boldsymbol{X}^T\boldsymbol{X}$ are already $\mathcal{O}(d^2 n)$.

**Numerical eigensolvers**

To find possible values for $\lambda \in \mathbb{R}$ and $\boldsymbol{x} \in \mathbb{R}^d$ in an eigenproblem of type

$$\boldsymbol{A}\boldsymbol{x} = \lambda \boldsymbol{x}$$

for a square, diagonalizable matrix $\boldsymbol{A} \in \mathbb{R}^{d \times d}$, there exist several numerical solvers, e.g., power methods, Rayleigh quotient iterations, pivoted Cholesky decompositions, or the Arnoldi/Lanczos algorithms. Note that, in contrast to computing the full eigendecomposition of $\boldsymbol{A}$, these solvers can be employed in such a way that only the largest $q$ eigenvalues and the corresponding eigenvectors are obtained with fewer computational costs. Due to the fact that computing these eigenpairs usually needs at least $\mathcal{O}(d^2 q)$ floating point iterations, the computation gets very costly for large dimension $d$. Also, many eigendecomposition algorithms can become instable. For more details on numerical eigensolvers, their implementation, and computational costs, we refer the reader to [GVL13, HPS12, PTVF07].

## 4.2 ▪ Connection to the singular value decomposition

Another way to compute the principal axes and components is to use the *singular value decomposition* (SVD) of $\boldsymbol{X}$, which we already introduced in Chapter 2. If we assume $n \geq d$, we get an SVD $\boldsymbol{X} = \boldsymbol{U}\boldsymbol{D}\boldsymbol{W}^T$ with singular values $\sigma_1 \geq \dots \geq \sigma_d \geq 0$. This leads to

$$n \cdot \boldsymbol{C} = \boldsymbol{X}^T\boldsymbol{X} = \boldsymbol{W}\boldsymbol{D}^T\boldsymbol{U}^T\boldsymbol{U}\boldsymbol{D}\boldsymbol{W}^T = \boldsymbol{W}\,\mathrm{diag}(\sigma_1^2, \dots, \sigma_d^2)\boldsymbol{W}^T$$

since $\boldsymbol{U}$ is orthogonal. This is an eigendecomposition of $n \cdot \boldsymbol{C}$. So, there is a one-to-one relationship between the eigenspaces spanned by the first $q$ columns of $\boldsymbol{W}$ and the ones spanned by $\boldsymbol{V}_q$. Furthermore, the eigenvalues of $\boldsymbol{C}$ fulfill $\lambda_i = \frac{\sigma_i^2}{n}$.

While computing the SVD of $\boldsymbol{X}$ usually costs $\mathcal{O}(\min(d,n)^2 \max(d,n))$ floating point operations, it can be done in a stable way in contrast to the direct eigendecomposition computation of $\boldsymbol{X}^T\boldsymbol{X}$; compare with Section 2.1 and see [GVL13, PTVF07]. Therefore, using an SVD is usually preferable over employing an eigensolver, e.g., a Lanczos iteration algorithm, to compute the eigenspaces of $\boldsymbol{C}$ or $\boldsymbol{G}$.

### Interpretation as optimal low-rank approximation

An alternative derivation of the PCA is based on an optimization problem involving low-rank matrix approximation, namely

$$\min_{\substack{\boldsymbol{A}\in\mathbb{R}^{n\times d}\\ \operatorname{rank}(\boldsymbol{A})\leq q}} \|\boldsymbol{A}-\boldsymbol{X}\|_F, \tag{4.5}$$

where $\|\cdot\|_F$ denotes the Frobenius norm. The *Schmidt–Eckart–Young–Mirsky* theorem states that the solution to this problem is given by a truncated version of the SVD [GVL13]. To this end, consider again

$$\boldsymbol{X}=\boldsymbol{U}\boldsymbol{D}\boldsymbol{W}^T,$$

i.e., the SVD of $\boldsymbol{X}$ with non-increasing singular values $\sigma_1 \geq \cdots \geq \sigma_{\min(n,d)} \geq 0$. We assume that $q \leq \min(n,d)$; otherwise we just take $\boldsymbol{X}$ as the solution to the optimization problem. Then the truncated version of the SVD is given by taking $\boldsymbol{U}_q \in \mathbb{R}^{n\times q}$ to be the first $q$ columns of $\boldsymbol{U}$ and by taking $\boldsymbol{W}_q \in \mathbb{R}^{d\times q}$ to be the first $q$ columns of $\boldsymbol{W}$. Finally we define $\boldsymbol{D}_q := \operatorname{diag}(\sigma_1,\ldots,\sigma_q) \in \mathbb{R}^{q\times q}$. Then, the Schmidt–Eckart–Young–Mirsky theorem states that

$$\boldsymbol{A} := \boldsymbol{U}_q\boldsymbol{D}_q\boldsymbol{W}_q^T$$

is the solution to the *best-rank-q-approximation* problem (4.5) and we have for the error

$$\|\boldsymbol{U}_q\boldsymbol{D}_q\boldsymbol{W}_q^T-\boldsymbol{X}\|_F^2 = \sum_{i=q+1}^{\min(n,d)} \sigma_i^2.$$

Furthermore, if $\sigma_{q+1} \neq \sigma_q$, the solution is unique.

## 4.3 ▪ Multi-dimensional scaling (MDS)

In Theorem 4.1.1 we derived the PCA via an eigendecomposition of the covariance matrix $\boldsymbol{C} = \frac{1}{n}\boldsymbol{X}^T\boldsymbol{X} \in \mathbb{R}^{d\times d}$ for the centered data matrix $\boldsymbol{X}$ from (4.4). However, we can observe that the $q$ largest eigenvalues $\lambda_1,\ldots,\lambda_q$ of $\boldsymbol{C}$ for $q \leq \min(d,n)$ are the same as the $q$ largest eigenvalues of the so-called *Gramian matrix* $\boldsymbol{G} := \frac{1}{n}\boldsymbol{X}\boldsymbol{X}^T \in \mathbb{R}^{n\times n}$ of pairwise scalar products of the centered data, i.e., $G_{ij} = \frac{1}{n}\langle \boldsymbol{x}_i-\boldsymbol{m}, \boldsymbol{x}_j-\boldsymbol{m}\rangle_2$ for $i,j=1,\ldots,n$. Therefore, we can also perform an eigendecomposition of $\boldsymbol{G}$, which costs $\mathcal{O}(n^2d)$ floating point operations, instead of an eigendecomposition of $\boldsymbol{C}$, which costs $\mathcal{O}(nd^2)$ floating point operations. To this end, note that

$$\boldsymbol{G}\boldsymbol{X}\boldsymbol{v}_j = \frac{1}{n}\boldsymbol{X}\boldsymbol{X}^T\boldsymbol{X}\boldsymbol{v}_j = \boldsymbol{X}\boldsymbol{C}\boldsymbol{v}_j = \lambda_j\boldsymbol{X}\boldsymbol{v}_j$$

holds for all eigenvectors $\boldsymbol{v}_j, j=1,\ldots,q$, of $\boldsymbol{C}$. This means that $\boldsymbol{X}\boldsymbol{v}_j$ is an eigenvector of $\boldsymbol{G}$ to the eigenvalue $\lambda_j$. Since $\|\boldsymbol{X}\boldsymbol{v}_j\|_2^2 = \boldsymbol{v}_j^T\boldsymbol{X}^T\boldsymbol{X}\boldsymbol{v}_j = n\lambda_j\|\boldsymbol{v}_j\|_2^2 = n\lambda_j$, the principal components are given by

$$\boldsymbol{\eta}_i = \boldsymbol{V}_q^T(\boldsymbol{x}_i-\boldsymbol{m}) = \begin{pmatrix} \boldsymbol{v}_1^T(\boldsymbol{x}_i-\boldsymbol{m}) \\ \vdots \\ \boldsymbol{v}_q^T(\boldsymbol{x}_i-\boldsymbol{m}) \end{pmatrix} = \begin{pmatrix} \sqrt{n\lambda_1}[\tilde{\boldsymbol{v}}_1]_i \\ \vdots \\ \sqrt{n\lambda_q}\,[\tilde{\boldsymbol{v}}_q]_i \end{pmatrix}, \tag{4.6}$$

where $[\tilde{\boldsymbol{v}}_j]_i$ denotes the $i$th entry of the (orthogonal) unit eigenvector $\tilde{\boldsymbol{v}}_j$, which corresponds to the $j$th largest eigenvalue of $\boldsymbol{G}$. Note that this only holds for one-dimensional eigenspaces. Otherwise we would also have to deal with possible rearrangements. Moreover, we observe that the left singular vectors $\boldsymbol{U}$ of $\boldsymbol{X}$ are the eigenvectors of the Gramian matrix $\boldsymbol{G}$.

**High-dimensional case** Sometimes we have to deal with situations where $d \gg n$. For instance, in gene expression analysis $d \approx 10^9$ and often $n \approx 10^3$. In these cases, we cannot directly compute the eigendecomposition of $\boldsymbol{C} = \frac{1}{n}\boldsymbol{X}^T\boldsymbol{X}$ when dealing with the eigenproblem

$$\boldsymbol{C}\boldsymbol{v} = \lambda\boldsymbol{v},$$

nor can we easily store the matrix $\boldsymbol{X}$ and compute the SVD, unless we exploit potential sparsity properties of $\boldsymbol{X}$. However, as already discussed, there are three ways to compute the principal components: an SVD of the $n \times d$ matrix $\boldsymbol{X}$, an eigenvalue decomposition of the $d \times d$ matrix $\boldsymbol{C}$, and an eigenvalue decomposition of the $n \times n$ matrix $\boldsymbol{G}$. Therefore, if $d \gg n$ and if direct access to $\boldsymbol{X}$ is unavailable, e.g., due to storage constraints or inaccessible data points, the latter variant is preferred since an SVD can no longer be computed and since the costs of $\mathcal{O}(n^2 d)$ floating point operations for the eigenvalue decomposition of $\boldsymbol{G}$ are cheaper than the costs of $\mathcal{O}(d^2 n)$ floating point operations for the eigenvalue decomposition of $\boldsymbol{C}$.

The computation of the PCA solution using the Gramian matrix is also known as *(classical) multi-dimensional scaling* (MDS) or *Torgerson MDS*. This approach was derived for applications where only the Gramian matrix $\boldsymbol{G}$ of pairwise similarities or the matrix $\boldsymbol{D}$ of squared pairwise Euclidean distances $\|\boldsymbol{x}_i - \boldsymbol{x}_j\|_2^2$ is given. The connection between $\boldsymbol{D}$ and the Gramian matrix $\boldsymbol{G}$ can be established by

$$\|\boldsymbol{x}_i - \boldsymbol{x}_j\|_2^2 = \boldsymbol{G}_{ii} + \boldsymbol{G}_{jj} - 2\boldsymbol{G}_{ij}. \tag{4.7}$$

Therefore, straightforward calculations show that the Gramian matrix $\boldsymbol{G}$ of the centered data $\boldsymbol{x}_i - \boldsymbol{m}$, $i = 1, \ldots, m$, can be computed by

$$\boldsymbol{G} = -\frac{1}{2}\boldsymbol{H}\boldsymbol{D}\boldsymbol{H}$$

when given $\boldsymbol{D}$; see also [CC00, LV07]. Here, $\boldsymbol{H} = \boldsymbol{I} - \frac{1}{n}\boldsymbol{1}_n$ is the centering matrix introduced earlier. In summary, we do not need to have access to the original data $\boldsymbol{X}$ if we have access to either $\boldsymbol{D}$ or $\boldsymbol{G}$ since we can work only with the Gramian matrix to compute the PCA of the data set.

**The underlying optimization problem of MDS** One can show that the MDS solution and thus the PCA solution solves the optimization problem

$$\min_{\boldsymbol{\eta}_i \in \mathbb{R}^q, i=1,\ldots,n} \sum_{i,j=1}^{n} \left(\langle \boldsymbol{x}_i, \boldsymbol{x}_j \rangle_2 - \langle \boldsymbol{\eta}_i, \boldsymbol{\eta}_j \rangle_2\right)^2, \tag{4.8}$$

and, equivalently,

$$\min_{\boldsymbol{\eta}_i \in \mathbb{R}^q, i=1,\ldots,n} \sum_{i,j=1}^{n} \left(\|\boldsymbol{x}_i - \boldsymbol{x}_j\|_2^2 - \|\boldsymbol{\eta}_i - \boldsymbol{\eta}_j\|_2^2\right)^2, \tag{4.9}$$

i.e., the low-dimensional representatives $\boldsymbol{\eta}_i$ of the data $\boldsymbol{x}_i, i = 1, \ldots, n$, are such that they (on average) preserve pairwise inner products, i.e., pairwise similarities, and pairwise squared distances best. For a proof of this property and more details on MDS, we refer the reader to [CC00, LV07, YH38].

## 4.4 ▪ Statistical interpretation

Viewing PCA as a statistical method, we can obtain yet another interpretation of it. To this end, let $\boldsymbol{x} \in \mathbb{R}^d$ be a random variable with zero mean, i.e., $\mathbb{E}[\boldsymbol{x}] = 0$. Its $d \times d$ covariance matrix

is defined as $\mathbb{E}[\boldsymbol{x}^T\boldsymbol{x}]$, and we assume that it has rank larger than or equal to $q$. Now we are asking for directions $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_q \in \mathbb{R}^d$ along which $\boldsymbol{x}$ varies the most, i.e., we first maximize the variance

$$\boldsymbol{v}_1 := \underset{\|\boldsymbol{v}\|_2=1}{\arg\max} \operatorname{Var}[\boldsymbol{v}^T\boldsymbol{x}].$$

Then, for $i = 1, \ldots, q$, we proceed successively by setting

$$\boldsymbol{v}_i := \underset{\substack{\|\boldsymbol{v}\|_2=1 \\ \boldsymbol{v}\perp\boldsymbol{v}_1,\ldots,\boldsymbol{v}_{i-1}}}{\arg\max} \operatorname{Var}[\boldsymbol{v}^T\boldsymbol{x}].$$

One can show that the resulting solution is given by the $q$ eigenvectors of $\mathbb{E}[\boldsymbol{x}^T\boldsymbol{x}]$ belonging to the largest $q$ eigenvalues; see [Jol02]. In fact, the variances are exactly the eigenvalues, i.e.,

$$\operatorname{Var}[\boldsymbol{v}_i^T\boldsymbol{x}] = \lambda_i \quad \forall\, i = 1, \ldots, q.$$

### Determining $q$ by kept variance

In applications of the PCA to real-world data, $q$ is often not fixed a priori but determined according to the decay of the eigenvalues $\lambda_i$. Since the eigenvalues directly correspond to the variance of the data along the eigenvector directions, this can be interpreted as keeping a certain percentage of the total variance $\operatorname{Var}[\boldsymbol{x}] := \operatorname{trace}(\boldsymbol{C}) = \sum_{i=1}^{\min(n,d)} \lambda_i$ of the data when reducing the dimensionality. To this end, we fix a threshold of $p$ percent with $0 < p < 100$ for kept variance percentage of the principal components. This means that $q \in \mathbb{N}$ is chosen as the smallest integer for which a dimension reduction by PCA, i.e., the computation of $\boldsymbol{v}_i = \boldsymbol{\eta}_i$ for $i = 1, \ldots, q$, according to Theorem 4.1.1, neglects at most $p$ percent of the variance. Thus, $q$ is the smallest integer such that

$$\frac{\sum_{i=1}^{q} \operatorname{Var}[\boldsymbol{v}_i^T\boldsymbol{x}]}{\operatorname{Var}[\boldsymbol{x}]} = \frac{\sum_{i=1}^{q} \lambda_i}{\sum_{i=1}^{\min(n,d)} \lambda_i} \geq \frac{p}{100}. \tag{4.10}$$

This establishes a practical criterion to determine a meaningful dimension $q$ a posteriori, i.e., after the computation of the eigendecomposition of the covariance matrix $\boldsymbol{C}$ is performed. On the other hand, recall that we do not necessarily have to compute the eigendecomposition of the whole matrix $\boldsymbol{C}$ but can only determine the largest few eigenvalues and the corresponding eigenvectors, as mentioned at the end of Section 4.1. Therefore, to obtain an efficient method, we can compute eigenvalue and eigenvector pairs successively until (4.10) is fulfilled.

## 4.5 ▪ Tasks on linear dimensionality reduction

Let us implement the PCA and test it on a toy example that we understand well. The accompanying material at https://bookstore.siam.org/di03/bonus includes a JUPYTER notebook `PCA_template.ipynb`, which should be used as a template for the solution. The material also includes the data sets and some PYTHON code one may use.

**Task 4.1.** Implement a PCA routine whose inputs are $\{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n\}$ and $q$. Either use a NUMPY/SCIPY routine to compute eigenvectors directly or use the SVD. In the case of eigenvectors, make sure your routine does actually return an orthonormal basis (consult the documentation of your solver).

### 4.5.1 ▪ Visualization of high-dimensional data

As a first test case, we consider a toy example with a 2D data set that is embedded into 4D by rotating and translating it. Additionally, some noise was added to the data. Our goal is to recover the 2D representation. Each point in the data set contains a label, which resembles a color assigned to that data point.

**Task 4.2.** Test your PCA implementation on the provided toy data set.

(a) Plot a 3D slice of the 4D toy data. Compute the PCA representation for $q = 2$ and plot it as described in the notebook. Check your result: The plot should reveal a perfectly round and familiar shape.

(b) Map the 2D representation onto a distorted ellipse in 3D; the code for the transformation is provided in the accompanying notebook. Perform a PCA of this 3D data for $q = 2$ and plot the result. Repeat this for a few ellipses and describe how the PCA picks the coordinate system $\boldsymbol{V}_q$.

Now let us revisit the Iris data set from Task 2.7 to check how PCA can be utilized there.

**Task 4.3.** Use PCA for the Iris data set.

(a) Compute all (four) singular values of $\boldsymbol{X}$ using a suitable function, e.g., `scipy.linalg.svdvals`. Compute the captured variance percentage when using only the first principal component and compute the captured variance when using the first two principal components.

(b) Compute the PCA transformation onto the first two principal components of the Iris data set. Plot the transformed data in a 2D scatter plot such that the three labels are distinguishable. Do the same for 1D and use the provided function to plot it. What do you observe?

(c) Use the insight from the visualization in (b) to build classifiers for the whole Iris data set. To this end, run three different experiments using a 4D, 2D, and 1D PCA as starting point. Then, apply two linear SVMs to classify a data point as one of the three Iris labels. You can copy your SVM code from Chapter 3, but we recommend using SCIKIT-LEARN.

With a better understanding of PCA we now turn our attention to the analysis of a larger data set.

### 4.5.2 ▪ Pedestrian classification

Detecting people in video footage is an interesting topic in many applications. One important area would be the self-driving car industry. The task can be split broadly into several connected steps:

- Gather the image material.
- Find region proposals in an image which could be interesting.
- Decide if a given region shows a pedestrian.

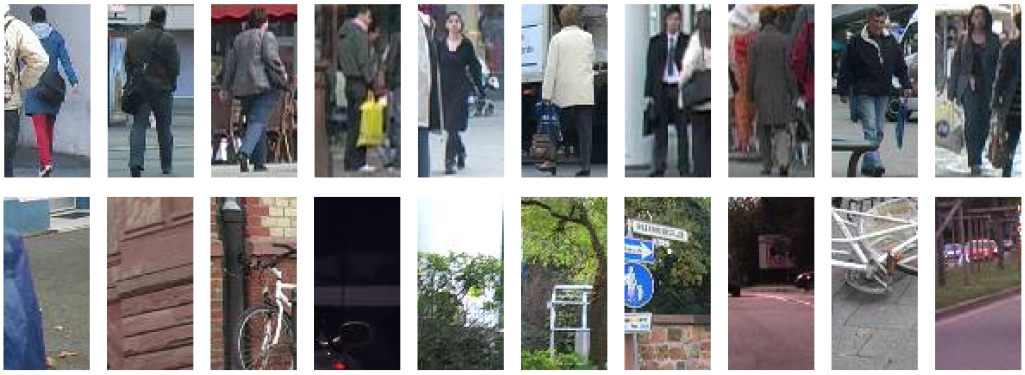


Figure 4.2: Ten pedestrian images and ten garbage (non-pedestrian) images from the TUD-Brussels data set [WWS09].

The second step is known as a *segmentation problem* in computer vision, which can also be interpreted as an unsupervised learning problem. It is often treated by thresholding methods, clustering approaches, or more sophisticated graph-based algorithms; see [ZA15] for further details. In the following, we will focus on the last task in more detail, i.e., on deciding whether or not a picture shows a pedestrian. Note that handling such a problem with a PCA is a rather "classical" approach in machine learning. Nowadays, these kinds of tasks are usually tackled with deep neural networks instead; see, e.g., [RDGF16, TMB+16]. For further details on pedestrian classification, we refer the reader to the reviews [BOHS15, DWSP12].

Our data set consists of labeled pictures of 100x50 pixels; see Figure 4.2. It is a downscaled subset of the TUD data set [WWS09]. Half of the pictures show a pedestrian; the other half does not. A separation into training and test data is provided. We begin by preparing the data.

**Task 4.4.** Prepare the pedestrian data set.

(a) Load the training and test images into NumPy arrays with the help of the routines in the template notebook. Normalize the pixel values to $[0, 1]$.

(b) Write a routine `plot_im` to plot an image using MatPlotLib's `imshow` (to get consistent contrast provide constant values for its arguments `vmin` and `vmax`). Create a plot with ten randomly chosen training images showing a pedestrian and ten randomly chosen training images not showing a pedestrian. You can use the `subplot` method[17] from MatPlotLib.

Our training data consists of $n = 2000$ points (images), with dimension $d = 15000$ (the pixels of an image for three colors). The dimension is quite large compared to our previous data sets. The goal is to classify the images (pedestrian or non-pedestrian) using the color values of the pixels.

After loading the data, we compute a PCA to reduce $d$ to a much smaller number in the next step. Note that the coordinate axes calculated by the PCA can be interpreted as images, and we represent the data in terms of coefficients for the corresponding eigenvectors, which are called *eigenpedestrians*; see also [SK87] for the general idea of using PCA for image analysis.

[17] You should implement an optional argument `ax` for `plot_im`, so you can use it for the subplot; see `plt.gca()`.

**Task 4.5.** Take a look at the eigenpedestrians. From now on use the PCA implementation of SCIKIT-LEARN.

(a) Compute the PCA with $q = 200$ for the full training set (i.e., with pedestrian and non-pedestrian samples combined).

(b) Plot the first 20 eigenpedestrians, as well as eigenpedestrians 50 to 60 and eigenpedestrians 100 to 110. What do you observe?

We now use our PCA representation to train a linear SVM.

**Task 4.6.** Train a linear SVM (use `LinearSVC` from SCIKIT-LEARN) employing the PCA representation of the full training data set for values of the dimension $q$ between 10 and 200 in steps of five. For each $q$ compute and store the prediction accuracy (use the `score` method of `LinearSVC`) on the training and test data set. Plot the scores for $q$. Which $q$ seems to be the best choice? Compare the situation to Task 4.3 (c) with respect to $q$.

The achieved prediction results are not bad. However, for safety-critical applications, e.g., in self-driving cars, they are not yet acceptable. Furthermore, the blind trust in error measures like the accuracy can be fatal in such tasks. Here, gaining insight on how the underlying ML algorithm came to its decision for certain input data is crucial. To this end, means of interpretability can be applied; see Section 10.5.

### 4.5.3 ▪ Histogram of oriented gradients

To improve the prediction quality, a handcrafted feature map can be employed in an additional computational step before applying the PCA. In particular, we will use a *histogram of oriented gradients* (HOG) of pixel values, which became popular after the well-received experiments in [DT05] were published. Here, *pixel gradients* are computed, i.e., discrete gradients in x- and y-directions of the color/gray-scale maps of images. Then, a histogram according to their magnitude is constructed and used as a feature for the subsequent machine learning algorithm. Applying HOG features for pedestrian detection has proven to be quite successful (see [BOHS15, DWSP12]) and allows for much better results than the use of the plain PCA for the linear SVM.

Note that tasks such as image classification or image segmentation are nowadays usually treated with automatic feature generation methods involving deep neural networks (see Chapter 6) instead of handcrafted HOG features. Nevertheless, it is instructive to understand the main aspects of the manual feature generation process, and we will therefore explain the details of the methodology in the following. Moreover, this section can be seen as an introduction to a historic method for feature selection which is interesting in its own right.

Before we explain HOG in detail, we make a quick digression into computer vision. To this end, let $\boldsymbol{I} \in \mathbb{R}^{h \times w}$ be a matrix representing one color channel of an $h \times w$-pixel image. Here, $h$ denotes the *height* (number of pixels in the $y$-direction) and $w$ denotes the *width* (number of pixels in the $x$-direction). The entries of the matrix represent pixel values for the respective color. Note that we thus have here the order $y, x$ (vertical, horizontal) instead of $x, y$ (horizontal, vertical) when talking about the entries of image matrices. However, this is consistent with the usual row, column order of matrices.

**Image gradients**

A very common tool in computer vision is ***image gradients***. Starting from the difference quotient

$$\lim_{\varepsilon \to 0} \frac{f(x+\varepsilon) - f(x)}{\varepsilon},$$

we can define a derivative for images by using a discrete approximation. For a single color channel, i.e., an image matrix $\boldsymbol{I} \in \mathbb{R}^{h \times w}$, we define the image gradient at position $y, x$ by the partial derivatives of $\boldsymbol{I}$ using a centered difference quotient

$$\nabla_{y,x} \boldsymbol{I} := \left( \frac{\boldsymbol{I}_{y+1,x} - \boldsymbol{I}_{y-1,x}}{s}, \frac{\boldsymbol{I}_{y,x+1} - \boldsymbol{I}_{y,x-1}}{s} \right)^T \tag{4.11}$$

with $s = 2$. Whenever there is overlap with the boundary, we set the corresponding pixel value to zero. Often the relative size of the derivatives is important instead of its scale. Therefore, the scaling factor $s$ in the denominator can be considered to be flexible. A popular choice is to take a denominator of $s = 1$ instead, which we will also use in the following.

Often, one can represent the discrete derivative calculation by a *convolution*—whose continuous counterpart might be known from integration theory.

**Convolutions**

The convolution $f * g$ of two real-valued functions $f, g : \mathbb{R}^d \to \mathbb{R}$ is defined by

$$f * g(\boldsymbol{x}) := \int_{\mathbb{R}^d} f(\boldsymbol{\tau}) g(\boldsymbol{x} - \boldsymbol{\tau}) \, \mathrm{d}\boldsymbol{\tau}.$$

If $f$ and $g$ are only given at discrete values $\mathbb{Z}^d$, the integral is substituted by a sum

$$f * g(\boldsymbol{z}) = \sum_{\boldsymbol{i} \in \mathbb{Z}^d} f(\boldsymbol{i}) g(\boldsymbol{z} - \boldsymbol{i}).$$

If $f, g$ are only defined on a finite domain, the above definitions can still be applied by extending the functions to $\mathbb{R}^d$ or $\mathbb{Z}^d$, respectively, and assigning $0$ as their function values outside the finite domain.

Given a *filter* matrix $\boldsymbol{K} \in \mathbb{R}^{p \times q}$, the convolution $\boldsymbol{I} * \boldsymbol{K}$ of $\boldsymbol{K}$ and $\boldsymbol{I}$ is defined pointwise by

$$(\boldsymbol{I} * \boldsymbol{K})_{y,x} := \sum_{\substack{0 < i \leq p \\ 0 < j \leq q}} \boldsymbol{K}_{i,j} \boldsymbol{I}_{y + k_y - i, x + k_x - j} \tag{4.12}$$

with *shifts*[18] $k_x$ and $k_y$. The result is a new color channel whose entries are computed using weighted sums of the surroundings of each pixel of $\boldsymbol{I}$. An example of a popular filter is Gaussian smoothing. Here the entries of $\boldsymbol{K}$ are computed using a Gaussian kernel.

Either the new color channel is smaller or one has to specify how the missing pixel values outside of the border of $\boldsymbol{I}$ are extrapolated. In our case we are extending $\boldsymbol{I}$ by the constant value 0. The $y$- and $x$-derivatives can be computed using `scipy.ndimage.convolve` with the filters $[-1, 0, 1]^T$ and $[1, 0, -1]$, which correspond to (4.11) with scaling factor $s = 1$.

[18] In SciPy these shifts are chosen to center the filter by default.

## Computing the HOG features

**Maximum norm gradients** To build the HOG features, we first compute the Euclidean norm and the direction of the image gradient $\nabla^{(j)}_{y,x}\boldsymbol{I}$ for each pixel $(y, x)$ and each color channel $j$ according to (4.11). For an RGB image, i.e., $j \in \{1, 2, 3\}$, this would give three sets of gradient norms and directions. To reduce this to one set, for each pixel $(y, x)$, we keep only the gradient for which $\left\|\nabla^{(j)}_{y,x}\boldsymbol{I}\right\|_2$ is the largest over all channels $j$, i.e., we keep

$$\nabla^{(\max)}_{y,x}\boldsymbol{I} := \nabla^{(j^*)}_{y,x}\boldsymbol{I} \quad \text{with} \quad j^* := \arg\max_j \left\|\nabla^{(j)}_{y,x}\boldsymbol{I}\right\|_2$$

for each $(y, x)$.

**Binning** Subsequently, each $(y, x)$ position is assigned (*binned*) to a square cell of size $|c| \in \mathbb{N}$. The cells span over $|c|$ pixels[19] in both directions and form a regular grid. Similarly, the orientations are binned into $\#b \in \mathbb{N}$ equally sized intervals which partition either the full circle $\alpha := 2\pi$ or only the half circle $\alpha := \pi$, in which case one calls the orientations *unsigned*. The corresponding intervals are

$$\left[\left(\frac{\alpha}{\#b}\right) i, \left(\frac{\alpha}{\#b}\right)(i+1)\right) \quad \text{for } i = 0, \ldots, \#b - 1.$$

**Computing the histogram** In a next step, the gradient norms are accumulated into a histogram[20] $H$. Here, the histogram entry $H(c_y, c_x, b_i)$ contains information about the gradient norms whose pixel $(y, x)$ resides in a cell indexed by $(c_y, c_x)$ and whose gradient orientation resides in the interval indexed by $b_i$.

Let us describe in more detail how the histogram entries are computed. For a gradient norm $\left\|\nabla^{(\max)}_{y,x}\right\|_2$ and a gradient direction $\phi \in [0, 2\pi)$ at position $(y, x)$, we first determine both the interval index $b_{\text{prec}}$ whose center is the largest among all centers which are smaller than or equal to

$$\hat{\phi} := \phi \mod \alpha$$

and the succeeding/next[21] interval index $b_{\text{succ}}$. Furthermore, we determine the preceding and succeeding cell indices $c_{y,\text{prec}}, c_{y,\text{succ}}, c_{x,\text{prec}}, c_{x,\text{succ}}$ in the $y$- and $x$-directions in the same fashion, i.e., with respect to their center. Here, out of bounds cells are ignored. Then, the histogram entries $H(c_y, c_x, b_i)$ for all possible combinations of the above mentioned indices are updated by adding a fraction of the gradient norm at position $(y, x)$. This fraction is individually defined for each $H(c_y, c_x, b_i)$ by the coefficients of a convex combination whose terms can be found in Algorithm 4.

**Block building** The last step is to combine the histogram entries of the cells into larger blocks, which are then normalized and clipped according to some clip value $T > 0$. If $|B| \in \mathbb{N}$ is the block size, then a block consists of $|B|$ consecutive cells in the $y$-direction and $|B|$ consecutive cells in the $x$-direction. The blocks do overlap, but only full blocks are considered. Consequently, $|B|$ has to be chosen less than or equal to the minimum number of cells in the $y$- and $x$-direction, respectively. The final HOG feature vector then consists of the entries of all blocks (in any order).

[19] Potential overlaps with the boundary of the image are ignored.

[20] While the method was known earlier, the term histogram (historical diagram) was introduced by Pearson, the inventor of PCA [Pea01].

[21] Note that we use a wrap-around when we are at the final index, i.e., the next index would be 0 again.

**Algorithm 4:** Computation of HOG features

**Input:** RGB image $\boldsymbol{I} \in \mathbb{R}^{h \times w \times 3}$, block size $|B|$ (default: 2), cell size $|c|$ (default: 8), number of bins $\#b$ (default: 9), clip value $T$ (default: 0.2), whether to use unsigned directions (default: yes).

**Output:** HOG feature vector.

$\alpha \leftarrow \begin{cases} \pi & \text{if use unsigned directions,} \\ 2\pi & \text{else.} \end{cases}$

$|b| \leftarrow \alpha / \#b$.

Initialize $H(c_y, c_x, b_i)$ to zero for all cell indices $c_y$, $c_x$ and all bin indices $b_i$.

**forall** pixel positions $(y, x)$ **do**

  **forall** $j = 1, 2, 3$ **do**

    Compute image gradients $\nabla_{y,x}^{(j)} \boldsymbol{I}$ at $y, x$ for channel $j$.

  **end forall**

  $j^* \leftarrow \arg\max_{j=1,2,3} \|\nabla_{y,x}^{(j)} \boldsymbol{I}\|_2$.

  $\nabla_{y,x}^{(\max)} \boldsymbol{I} \leftarrow \nabla_{y,x}^{(j^*)} \boldsymbol{I}$.

  $\hat{\phi} \leftarrow \operatorname{atan2}\left(\left[\nabla_{y,x}^{(\max)} \boldsymbol{I}\right]_1, \left[\nabla_{y,x}^{(\max)} \boldsymbol{I}\right]_2\right) \mod \alpha$.

  $b_{\text{prec}} \leftarrow$ index of orientation interval preceding the orientation $\hat{\phi}$ with respect to the interval's center (can be $-1$).

  $c_{x,\text{prec}} \leftarrow$ index of horizontally preceding cell with respect to its center (can be $-1$).

  $c_{y,\text{prec}} \leftarrow$ index of vertically preceding cell with respect to its center (can be $-1$).

  $f_b \leftarrow \frac{\hat{\phi} - [b_{\text{prec}}|b| + \frac{1}{2}|b|]}{|b|}$.

  $f_x \leftarrow \frac{x - [c_{x,\text{prec}}|c| + \frac{1}{2}|c| - \frac{1}{2}]}{|c|}$.

  $f_y \leftarrow \frac{y - [c_{y,\text{prec}}|c| + \frac{1}{2}|c| - \frac{1}{2}]}{|c|}$.

  $b_{\text{succ}} \leftarrow b_{\text{prec}} + 1 \mod \#b$, $b_{\text{prec}} \leftarrow b_{\text{prec}} \mod \#b$.

  $c_{x,\text{succ}} \leftarrow c_{x,\text{prec}} + 1$, $c_{y,\text{succ}} \leftarrow c_{y,\text{prec}} + 1$.

  (The cells with indices $-1$ and $\lfloor {}^{h}/_{|c|} \rfloor$ or $\lfloor {}^{w}/_{|c|} \rfloor$ can be stored in $H$ but must not be considered for the block computation at the end.)

  $H(c_{y,\text{prec}}, c_{x,\text{prec}}, b_{\text{prec}}) \leftarrow \text{add } \|\nabla_{y,x}\| \; (1 - f_x)(1 - f_y)(1 - f_b)$.

  $H(c_{y,\text{prec}}, c_{x,\text{prec}}, b_{\text{succ}}) \leftarrow \text{add } \|\nabla_{y,x}\| \; (1 - f_x)(1 - f_y) \quad f_b$.

  $H(c_{y,\text{succ}}, c_{x,\text{prec}}, b_{\text{prec}}) \leftarrow \text{add } \|\nabla_{y,x}\| \; (1 - f_x) \quad f_y \quad (1 - f_b)$.

  $H(c_{y,\text{succ}}, c_{x,\text{prec}}, b_{\text{succ}}) \leftarrow \text{add } \|\nabla_{y,x}\| \; (1 - f_x) \quad f_y \quad f_b$.

  $H(c_{y,\text{prec}}, c_{x,\text{succ}}, b_{\text{prec}}) \leftarrow \text{add } \|\nabla_{y,x}\| \quad f_x \quad (1 - f_y)(1 - f_b)$.

  $H(c_{y,\text{prec}}, c_{x,\text{succ}}, b_{\text{succ}}) \leftarrow \text{add } \|\nabla_{y,x}\| \quad f_x \quad (1 - f_y) \quad f_b$.

  $H(c_{y,\text{succ}}, c_{x,\text{succ}}, b_{\text{prec}}) \leftarrow \text{add } \|\nabla_{y,x}\| \quad f_x \quad f_y \quad (1 - f_b)$.

  $H(c_{y,\text{succ}}, c_{x,\text{succ}}, b_{\text{succ}}) \leftarrow \text{add } \|\nabla_{y,x}\| \quad f_x \quad f_y \quad f_b$.

**end forall**

**forall** blocks **do**

  Normalize all histogram values in the block with respect to the Euclidean (vector) norm of the block.

  Clip the block, i.e., apply $\min(\cdot, T)$ entrywise, then normalize again.

  Add the block entries to the final feature vector.

**end forall**

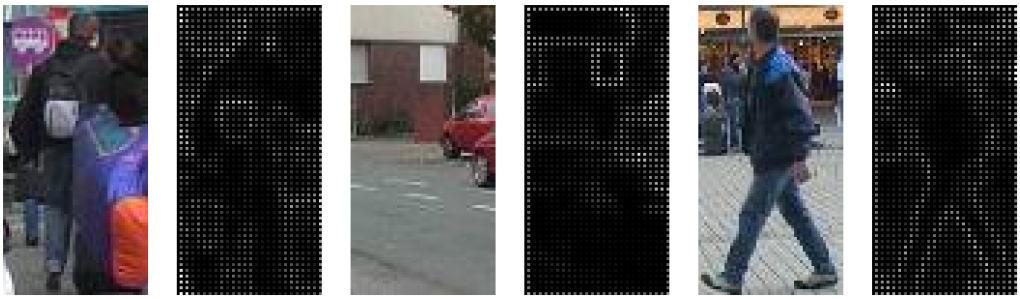


Figure 4.3: Three random images created from the TUD-Brussels data set [WWS09] together with their computed HOG features (right of the respective image). In each $(c_y, c_x)$-cell of the HOG histogram and for each orientation index $b_i$ we have drawn an arrow with direction corresponding to the orientation from bin $b_i$. The brightness of each arrow is proportional to the histogram entry $H(c_y, c_x, b_i)$. To obtain a better visualization we omitted the block building and normalizing steps at the end of Algorithm 4.

A visualization of computed HOG features can be found in Figure 4.3. We clearly see that the HOG features are able to capture coarse shapes and, thus, can be helpful in the detection of pedestrians. For more details on the actual computation of the HOG features, we refer the reader to Algorithm 4 and the PYTHON code template provided for the following exercise.

**Task 4.7.** Test the HOG features for the pedestrian data set. To this end, repeat the experiment from Task 4.6 for values of $q$ between 10 and 200 in steps of 5, but use the HOG features as input for the PCA instead. You can find a function to compute the HOG features in the template code.

## 4.6 ▪ Further topics

**Feature map construction** The HOG features allowed us to increase the prediction rate notably compared to plain PCA with linear SVM. They are one of many examples where sophisticated hand-crafted features help to improve on the solution to a problem. Often, a lot of work can go into such feature maps, and they might be problem specific. In recent years, the deep learning community has provided powerful tools to tackle computer image vision problems. One advantage there is that (convolutional) neural networks can learn feature maps on their own, i.e., somewhat automatically. However, the training is not well understood and can be quite tedious and involved; see Chapter 6. Nevertheless, manually created features can still be a valuable tool in the machine learning workflow in some situations.

**General multi-dimensional scaling** Besides classical multi-dimensional scaling, i.e., Torgerson MDS, which we introduced in Section 4.3, there exist many variations of the MDS algorithm. The class of *metric MDS* methods considers the preservation of distances, e.g., squared Euclidean distances as used in (4.9) for Torgerson MDS. More sophisticated variants of metric MDS employ general distance measures $\mathrm{dist}$ in the original data space and introduce additional (multiplicative) weights $w_{ij} \geq 0$ for $i, j = 1, \ldots, n$ in the $i, j$th summand of the loss function (4.9), i.e.,

$$\min_{\boldsymbol{\eta}_i \in \mathbb{R}^q, i=1,\ldots,n} \sum_{i,j=1}^{n} w_{ij} \left( \mathrm{dist}(\boldsymbol{x}_i, \boldsymbol{x}_j) - \|\boldsymbol{\eta}_i - \boldsymbol{\eta}_j\|_2 \right)^2 .$$

For these variants, well-known iterative solvers exist; see, e.g., [BG05]. Examples for metric MDS methods are *Sammon's mapping*, which uses $w_{ij} = 1/\operatorname{dist}(\boldsymbol{x}_i, \boldsymbol{x}_j)$ as weights, *Isomap*, which employs an approximation to the geodesic distance, and the so-called *kernel PCA*, where a kernel is used in (4.8) instead of a scalar product and distances are defined by a generalized version of (4.7); see also Chapter 5 and [BG05, LV07]. Note here that the name "kernel PCA" is quite misleading since the approach relates more closely to classical metric MDS than to PCA. Besides metric MDS, there also exist *non-metric* variants like the *Kruskal–Shepard* algorithm. Here, the corresponding optimization problem (4.8) is altered in such a way that ordinal information, i.e., proximity ranks, is used for determining the low-dimensional representation instead of pairwise scalar products. For more details on the above-mentioned variants, we refer the reader to [BG05, LV07].

## Chapter 5

# Nonlinear Dimensionality Reduction

In the last chapter we introduced PCA as the essential linear dimensionality reduction method. It is based on the restrictive assumption that the data resides in an affine linear subspace. In contrast to that, nonlinear dimensionality reduction methods consider general (curved) manifolds or aim to preserve aspects of the topology instead; see, e.g., [LV07].

Let us assume that $\boldsymbol{x}_i, i = 1, \ldots, n$, are given i.i.d. measurements from two random variables $A_1$ and $A_2$. Figure 5.1 shows an example, where the noisy data approximately stem from a one-dimensional curved submanifold of $\mathbb{R}^2$, i.e., it can be described by a nonlinear coordinate system. We will now extend the idea of linear dimensionality reduction to the nonlinear case. To this end, recall that we have briefly mentioned variants of multi-dimensional scaling algorithms at the end of Chapter 4, where the goal is to preserve pairwise distances between points as well as possible. In the following, we first introduce an MDS variant based on geodesic distances called *Isomap* in Section 5.1. Subsequently, we introduce *diffusion maps* in Section 5.2, where we consider preservation of so-called diffusion distances on the intrinsic (nonlinear) submanifold of $\mathbb{R}^d$ the data are living on. Section 5.3 then deals with clustering algorithms to segment a data set into different subsets. Combining clustering algorithms with dimensionality reduction, we obtain a method to visualize a high-dimensional data set in Section 5.4. Finally, tasks on nonlinear dimensionality reduction can be found in Section 5.5.

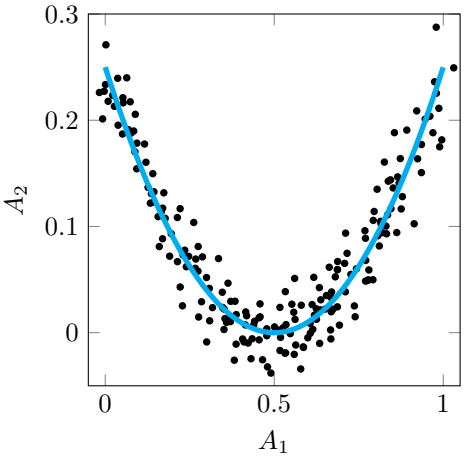



Figure 5.1: An example for a two-dimensional data set of noisy samples from an underlying curved, nonlinear structure.

## 5.1 ▪ Isomap

A key idea of many nonlinear dimensionality reduction approaches is to consider data that stem from (curved) manifolds $\mathfrak{M}$ instead of just linear subspaces, i.e., the model space contains nonlinear functions instead of only (affine) linear ones. In particular, Riemannian manifolds $\mathfrak{M}$ and the corresponding inner product $\langle \cdot, \cdot \rangle_{\mathfrak{M}}$ on the tangent space are of interest. On such a manifold $\mathfrak{M}$, the length of the shortest curve connecting two points $\boldsymbol{x}, \boldsymbol{y} \in \mathfrak{M}$ is the geodesic distance, which we denote by $\mathrm{dist}_{\mathfrak{M}}(\boldsymbol{x}, \boldsymbol{y})$; see [dC92] for definitions and more details on Riemannian manifolds and geodesic distances.

**Isomap**

The motivation behind the ***Isomap*** approach [TdL00] is to preserve the pairwise geodesic distances in the lower-dimensional representation. In Isomap the geodesic distances are approximated by graph distances that are computed on a graph that represents neighborhood relations of the data set. Using the graph distances, a corresponding centered Gramian matrix is obtained, whose eigendecomposition then gives the lower-dimensional representation of the data, analogously to MDS.

**Approximation of geodesic distances** Since we do not have access to the geodesic distances between data points directly, they have to be estimated in order to compute the Isomap solution. To this end, we construct a *neighborhood graph* $\mathcal{G}$ on the data set $\mathcal{X} := \{\boldsymbol{x}_i\}_{i=1,\ldots,n}$.

**Neighborhood graph**

A graph $\mathcal{G} := (V, E)$ with a so-called *vertex set* $V = \{v_1, \ldots, v_{|V|}\}$ and a so-called *edge set* $E \subseteq V \times V$ is often represented by its *adjacency matrix*, i.e., a matrix $\boldsymbol{A}$ of size $|V| \times |V|$, where $|V|$ is the number of nodes in the graph. The adjacency matrix has a non-zero entry at position $(i, j)$ if and only if the $i$th vertex is connected to the $j$th vertex by an edge in the graph, i.e.,

$$\boldsymbol{A}_{ij} = \begin{cases} w_{ij} & \text{if } (i,j) \in E, \\ 0 & \text{else,} \end{cases}$$

where $w_{ij}$ is a suitable non-zero edge weight, which reflects the importance of the connection between vertex $i$ and vertex $j$.
An (undirected) ***neighborhood graph*** is a graph on a given data set $V = \mathcal{X}$, where an edge between two data points $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$ exists if and only if these points are close to each other. Two famous examples of neighborhood graphs are the *$k$-nearest neighbors graph*, where each data point is connected to its $k$-nearest neighbors, and the *$r$-ball neighborhood graph*, where an edge between $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$ exists if and only if $\|\boldsymbol{x}_i - \boldsymbol{x}_j\|_2 \leq r$ for all $i, j = 1, \ldots, n$. In both cases, the edge weights are defined as the Euclidean distance[22] $w_{ij} := \|\boldsymbol{x}_i - \boldsymbol{x}_j\|_2$.

Given a neighborhood graph $\mathcal{G}$ on the data set $\mathcal{X}$, we define the *graph distance* $\mathrm{dist}_{\mathcal{G}} : \mathcal{X} \times \mathcal{X} \to [0, \infty]$ between two points $\boldsymbol{x}_i, \boldsymbol{x}_j \in \mathcal{X}$ according to the shortest path between these points, i.e.,

[22] Note that distance measures other than the Euclidean distance could also be employed here.

1. if $(\boldsymbol{x}_i, \boldsymbol{x}_j) \in E$, then let
$$\operatorname{dist}_{\mathcal{G}}(\boldsymbol{x}_i, \boldsymbol{x}_j) \coloneqq w_{ij};$$

2. else let
$$I_{ij} \coloneqq \left\{(i_k)_{k=1}^{s} \;\middle|\; s \in \mathbb{N}, i_1 = i, i_s = j \text{ and } \left(\boldsymbol{x}_{i_k}, \boldsymbol{x}_{i_{k+1}}\right) \in E \,\forall\, i = 1, \ldots, s-1\right\}$$
be the set of index sequences of all possible finite paths of any length, which we denote by $s$, between $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$, and let
$$\operatorname{dist}_{\mathcal{G}}(\boldsymbol{x}_i, \boldsymbol{x}_j) \coloneqq \inf_{(i_k)_{k=1}^{s} \in I_{ij}} \sum_{k=1}^{s-1} w_{i_k i_{k+1}}.$$

To compute all graph distances $\operatorname{dist}_{\mathcal{G}}(\boldsymbol{x}_i, \boldsymbol{x}_j)$ between points $\boldsymbol{x}_i, \boldsymbol{x}_j$, $i, j = 1, \ldots, n$, Dijkstra's algorithm [Dij59] or the Floyd–Warshall algorithm [Flo62] can be used, for example.

**Computing the Isomap solution** Given the graph distances, we can formulate the Isomap approach for reducing the data dimensionality[23] from $d$ to $q < d$. It is known to work well when the underlying manifold $\mathfrak{M}$ is isometric to a convex domain; see [ACJP20]. We assume $\mathcal{X} \subset \mathfrak{M} \subseteq \mathbb{R}^d$, i.e., the data stem from the manifold $\mathfrak{M}$, which is a submanifold of $\mathbb{R}^d$. Furthermore, if $\mathcal{X}$ is sampled densely enough from $\mathfrak{M}$, i.e., if the Euclidean distance between local data points is close to the geodesic distance, and if $\mathcal{G}$ only connects close neighbors to each other, we expect that
$$\operatorname{dist}_{\mathcal{G}}(\boldsymbol{x}_i, \boldsymbol{x}_j) \approx \operatorname{dist}_{\mathfrak{M}}(\boldsymbol{x}_i, \boldsymbol{x}_j).$$
This means that the graph distance matrix $\boldsymbol{D}_{\mathcal{G}}$ with entries $\operatorname{dist}^2_{\mathcal{G}}(\boldsymbol{x}_i, \boldsymbol{x}_j)$ for $i, j = 1, \ldots, n$ is a good approximation to the geodesic distance matrix $\boldsymbol{D}_{\mathfrak{M}}$ with entries $\operatorname{dist}^2_{\mathfrak{M}}(\boldsymbol{x}_i, \boldsymbol{x}_j)$ for $i, j = 1, \ldots, n$. For a formal proof under corresponding assumptions on the data and on the manifold, see [ACJP20, TdL00].

Our goal is now to find low-dimensional representations $\{\boldsymbol{\eta}_i\}_{i=1}^n \subset \mathbb{R}^p$ of the data points $\{\boldsymbol{x}_i\}_{i=1}^n \subset \mathfrak{M} \subseteq \mathbb{R}^d$ such that their Euclidean distance matrix $\boldsymbol{D}_{\boldsymbol{\eta}}$ with entries $\|\boldsymbol{\eta}_i - \boldsymbol{\eta}_j\|_2^2$ for $i, j = 1, \ldots, n$ fulfills
$$\boldsymbol{D}_{\mathcal{G}} \approx \boldsymbol{D}_{\boldsymbol{\eta}}.$$
This can be done by computing the solution to the Torgerson MDS problem (4.9) with $\operatorname{dist}^2_{\mathcal{G}}(\boldsymbol{x}_i, \boldsymbol{x}_j)$ instead of $\|\boldsymbol{x}_i - \boldsymbol{x}_j\|_2^2$. To this end, we first center $\boldsymbol{D}_{\mathcal{G}}$ to obtain $\boldsymbol{G} = -\frac{1}{2}\boldsymbol{H}\boldsymbol{D}_{\mathcal{G}}\boldsymbol{H}$ using the centering matrix from (4.4). Then, we compute the eigenvalue decomposition of $\boldsymbol{G}$ to obtain the low-dimensional vectors $\{\boldsymbol{\eta}_i\}_{i=1}^n$, which are the dimension-reduced representatives of the original data $\mathcal{X}$. We refer the reader to Algorithm 5 for details and to Section 4.3 for the analogy to MDS.

## 5.2 ▪ Diffusion maps

Next, we will discuss a more stochastically motivated nonlinear dimensionality reduction method, namely *diffusion maps*, which is based on so-called *diffusion distances*. Unlike the graph distance, which serves as an approximation to the geodesic distance, the diffusion distance is robust to noise and topological shortcuts because it involves sums over all paths of a certain length connecting two points of $\mathcal{X}$.

[23] Note that $q \in \mathbb{N}$ is usually chosen a priori. However, as described in Section 4.4, $q$ can also be chosen dynamically by taking the eigenvalue decay into account.

**Algorithm 5:** Isomap algorithm

**Input:** data $\mathcal{X}$, dimension $q < \min(n, d)$
**Output:** low-dimensional data representation $\mathcal{Y}$ of dimension $q$.

1 Build a neighborhood graph $\mathcal{G} = (\mathcal{X}, E)$ (e.g., $k$-nearest neighbors graph).
2 $(\boldsymbol{D}_{\mathcal{G}})_{ij} \leftarrow \mathrm{dist}_{\mathcal{G}}(\boldsymbol{x}_i, \boldsymbol{x}_j)\ \forall\, i, j = 1, \ldots, n$ (e.g., by Dijkstra's algorithm).
3 $\boldsymbol{G} \leftarrow -\frac{1}{2}\boldsymbol{H}\boldsymbol{D}_{\mathcal{G}}\boldsymbol{H}$.
4 Compute the first $q$ eigenvalues $\{\lambda_l\}_{l=1}^{q}$ and the corresponding eigenvectors $\{\psi_l\}_{l=1}^{q}$ of $\boldsymbol{G}$.
5 $(\boldsymbol{\eta}_1\ \boldsymbol{\eta}_2\ \cdots\ \boldsymbol{\eta}_n) \leftarrow \left(\sqrt{\lambda_1}\psi_1\ \sqrt{\lambda_2}\psi_2\ \cdots\ \sqrt{\lambda_q}\psi_q\right)^T \in \mathbb{R}^{q\times n}$.
6 $\mathcal{Y} \leftarrow \{\boldsymbol{\eta}_i\}_{i=1}^{n}$.
7 **return** $\mathcal{Y}$.

**Diffusion maps**

***Diffusion maps*** is a nonlinear dimensionality reduction method introduced by Coifman and Lafon; see [CL06]. Here, the value $K(\boldsymbol{x}_i, \boldsymbol{x}_j)$ of a nonlinear kernel $K$ in two data points $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$ is used to represent the similarity between them. The underlying idea of diffusion maps is to construct a random walk Markov chain on the data set by using the kernel evaluations to build a transition probability matrix $\boldsymbol{P} \in \mathbb{R}^{n\times n}$ on the data set. Then, the low-dimensional data representations can be constructed from an eigendecomposition of $\boldsymbol{P}$.

To consider the diffusion maps algorithm in more detail, let $\mathcal{X} = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n\} \subset \Omega \subseteq \mathbb{R}^d$ be our data set. We will now encode the geometry of the data in terms of similarities between two points by a kernel function. To this end, we choose a symmetric and positive kernel $K : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$, i.e., the kernel fulfills $K(\boldsymbol{x}, \boldsymbol{y}) > 0$ and $K(\boldsymbol{x}, \boldsymbol{y}) = K(\boldsymbol{y}, \boldsymbol{x})$ for all $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^d$. The value $K(\boldsymbol{x}, \boldsymbol{y})$ represents the similarity between the data points $\boldsymbol{x}$ and $\boldsymbol{y}$, i.e., large kernel values mean that the respective two points are similar to each other. A prominent example for a valid kernel, which is most commonly used in diffusion maps, is the Gaussian kernel $K_\sigma(\boldsymbol{x}, \boldsymbol{y}) = \exp\left(-\frac{\|\boldsymbol{x}-\boldsymbol{y}\|_2^2}{2\sigma^2}\right)$ with a variance hyperparameter $\sigma > 0$. With the help of this kernel, we now build a *Markov chain* transition process from a graph defined on the data points.

**Markov chains and random walks**

A family $Y = (X_i)_{i\in\mathbb{N}}$ of random variables on the countable set $\mathcal{X}$ is a ***Markov chain*** if the transition probabilities fulfill

$$\begin{aligned}&\mathbb{P}\left[X_m = \boldsymbol{x}_{i_m} \mid X_{m-1} = \boldsymbol{x}_{i_{m-1}}, \ldots, X_1 = \boldsymbol{x}_{i_1}\right]\\ &= \mathbb{P}\left[X_m = \boldsymbol{x}_{i_m} \mid X_{m-1} = \boldsymbol{x}_{i_{m-1}}\right]\end{aligned}$$

for arbitrary $m \in \mathbb{N}$ and $i_1, \ldots, i_m \in \{1, \ldots, n\}$. This can be interpreted such that the probability of the variable $X_i$ being in a certain state only depends on the state of $X_{i-1}$ and not on the complete history of the process. If the distribution is stationary, i.e., if $\mathbb{P}\left[X_m = \boldsymbol{x}_i \mid X_{m-1} = \boldsymbol{x}_{i_j}\right]$ is the same regardless of $m \in \mathbb{N}$, the Markov chain $Y$ is called the path of a random walk on the data. The analogy becomes clearer

when the data points are considered to be vertices of a fully connected, directed graph, where the edge weight from $\boldsymbol{x}_i$ to $\boldsymbol{x}_j$ is given by $\mathbb{P}[X_2 = \boldsymbol{x}_j \mid X_1 = \boldsymbol{x}_i]$. Then, following the path given by an instance of $Y$, we obtain a random walk on the graph. For more details, see [Geo12].

**Defining the transition probabilities** To construct the Markov chain, let each data point represent a vertex in a fully connected graph. The edge weights are now built according to the values of $K$, which serves as a similarity measure between two data points. To construct the transition probabilities of the Markov chain, we first normalize the kernel. To this end, we choose a parameter $\alpha \in [0, 1]$ and define

$$K^{(\alpha)}(\boldsymbol{x}, \boldsymbol{y}) := \frac{K(\boldsymbol{x}, \boldsymbol{y})}{\left(\sum_{\boldsymbol{z}\in\mathcal{X}} K(\boldsymbol{x}, \boldsymbol{z})\right)^{\alpha} \left(\sum_{\boldsymbol{z}\in\mathcal{X}} K(\boldsymbol{y}, \boldsymbol{z})\right)^{\alpha}}. \tag{5.1}$$

With the help of the rescaled kernel $K^{(\alpha)}$ we can define the transition probability from $\boldsymbol{x} \in \mathcal{X}$ to $\boldsymbol{y} \in \mathcal{X}$ by

$$\mathbb{P}[\boldsymbol{y} \mid \boldsymbol{x}] := P(\boldsymbol{x}, \boldsymbol{y}) := \frac{K^{(\alpha)}(\boldsymbol{x}, \boldsymbol{y})}{\sum_{\boldsymbol{z}\in\mathcal{X}} K^{(\alpha)}(\boldsymbol{x}, \boldsymbol{z})}. \tag{5.2}$$

**The choice of $\alpha$** Depending on the choice of $\alpha$, the transition probabilities will change significantly. One can show that the value of $\alpha$ corresponds to a specific type of *flow field* on the submanifold the data lie on; see [CL06]. In particular, $\alpha = 1$ leads to a finite-sample approximation of a so-called *Brownian motion* random process. This resembles the dynamics of a flow field following the *Laplace–Beltrami* operator, i.e., the Laplace operator on the submanifold. When choosing $\alpha = 0$, the flow field follows the so-called *normalized graph Laplacian* for a Gaussian kernel $K$. Finally, values of $\alpha$ between 0 and 1 introduce a *drift* term in addition to the Brownian motion. When choosing $\alpha = \frac{1}{2}$, for example, we obtain the *Fokker–Planck* dynamics of the random walk. For more details, we refer the reader to [CL06].

We will choose $\alpha = 1$ later on since this choice is optimally suited to get rid of effects of the data density on the constructed random walk and just recovers the geometry of the submanifold. For detailed information on the convergence properties of the discrete data manifold towards the underlying submanifold of $\mathbb{R}^d$, we refer the reader to [GTGHS20].

**Diffusion distances** If we think about the matrix $\boldsymbol{P} \in \mathbb{R}^{n\times n}$ with entries $\boldsymbol{P}_{i,j} = P(\boldsymbol{x}_i, \boldsymbol{x}_j)$ as the transition matrix of the Markov process, we obtain the $t$-step transition matrix by computing the power $\boldsymbol{P}^t$ for some $t \in \mathbb{N}$. This allows us to define the so-called *diffusion distance*.

**Diffusion distance**

The (squared) ***diffusion distance*** at time $t \in \mathbb{N}$ is defined by

$$\mathrm{DiffDist}_t^2(\boldsymbol{x}_i, \boldsymbol{x}_j) := \sum_{k=1}^{n} \left([\boldsymbol{P}^t]_{i,k} - [\boldsymbol{P}^t]_{j,k}\right)^2 \frac{1}{\pi(\boldsymbol{x}_k)}, \tag{5.3}$$

which resembles how far two points $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$ on the "data manifold" are away from each other in terms of *reachability* within $2t$ steps by the random walk defined by $\boldsymbol{P}$; see [CL06]. Here, $\pi$ is the stationary distribution of the random walk on the data set $\mathcal{X}$. In particular, $\boldsymbol{\pi} \in \mathbb{R}^n$ is defined by $\boldsymbol{\pi}_i = \pi(\boldsymbol{x}_i)$ for $i = 1, \ldots, n$ and fulfills $\boldsymbol{\pi}\boldsymbol{P} = \boldsymbol{\pi}$.

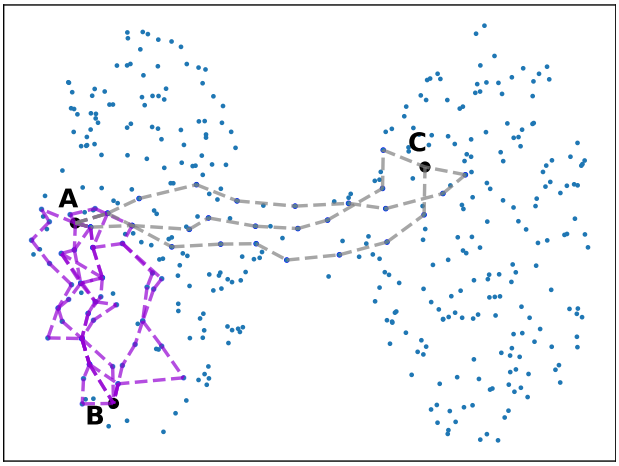



Figure 5.2: Random walks on a dumbbell-shaped data set (blue points). The diffusion distance between A and C is large since only a few random paths (gray) of length $2t = 10$ exist between them. However, the diffusion distance between A and B is small because there exist many paths (violet) of length $2t = 10$ between them. Note that only some exemplary paths are plotted here for reasons of clarity.

The diffusion distance provides a measure of distance on the data manifold. In Figure 5.2, for instance, the diffusion distance between the data points $A$ and $B$ is significantly smaller than the diffusion distance between $A$ and $C$ due to their different reachability in the data graph. Another interpretation is that $\mathrm{DiffDist}_t^2(\boldsymbol{x}_i, \boldsymbol{x}_j)$ is a weighted $L_2$ distance between the corresponding two probability distributions $\left([\boldsymbol{P}^t]_{i,k}\right)_{k=1,\dots,n}$ and $\left([\boldsymbol{P}^t]_{j,k}\right)_{k=1,\dots,n}$ over the data set. The difference between the distributions is small if their main probability masses after time $t$ are in similar regions of the data set, which means that the corresponding components in (5.3) cancel each other. This reflects that the two points $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$ have a large probability of being connected with a random path of length $2t$. In contrast to that, there is no cancellation effect in (5.3) if the main masses of the distributions lie in different regions of the data set.

By multiplying (5.3) out we can further see that

$$\begin{aligned}
\mathrm{DiffDist}_t^2(\boldsymbol{x}_i, \boldsymbol{x}_j) &= \sum_{k=1}^{n} \left( [\boldsymbol{P}^t]_{i,k}^2 - 2[\boldsymbol{P}^t]_{i,k}[\boldsymbol{P}^t]_{j,k} + [\boldsymbol{P}^t]_{j,k}^2 \right) \frac{1}{\pi(\boldsymbol{x}_k)} \\
&= \sum_{k=1}^{n} [\boldsymbol{P}^t]_{i,k}^2 \frac{1}{\pi(\boldsymbol{x}_k)} - 2 \sum_{k=1}^{n} \left( [\boldsymbol{P}^t]_{i,k}[\boldsymbol{P}^t]_{j,k} \right) \frac{1}{\pi(\boldsymbol{x}_k)} + \sum_{k=1}^{n} [\boldsymbol{P}^t]_{j,k}^2 \frac{1}{\pi(\boldsymbol{x}_k)} \\
&= \langle \boldsymbol{x}_i, \boldsymbol{x}_i \rangle_{p_t,\pi} - 2\langle \boldsymbol{x}_i, \boldsymbol{x}_j \rangle_{p_t,\pi} + \langle \boldsymbol{x}_j, \boldsymbol{x}_j \rangle_{p_t,\pi}.
\end{aligned}$$

Here, $\langle \cdot, \cdot \rangle_{p_t,\pi}$ is defined as

$$\langle \boldsymbol{x}_i, \boldsymbol{x}_j \rangle_{p_t,\pi} := \sum_{k=1}^{n} \left( [\boldsymbol{P}^t]_{i,k}[\boldsymbol{P}^t]_{j,k} \right) \frac{1}{\pi(\boldsymbol{x}_k)},$$

which is a weighted scalar product of a feature map that involves the probabilities of going from a point $\boldsymbol{x}_i$ (or $\boldsymbol{x}_j$, respectively) to any other node in $t$ steps. Therefore, the scalar product involving

the feature map defines a kernel on the data set. Note that the connection between distances and scalar products above is analogous to (4.7).

**Diffusion maps** To build now the so-called *diffusion map*, we exploit an important property of the eigendecomposition of $\boldsymbol{P}$: In fact, it can be shown that $\boldsymbol{P}$ admits a sequence of eigenvalues $1 = \lambda_0 > |\lambda_1| \geq \cdots \geq |\lambda_{n-1}|$ and corresponding orthonormal eigenvectors $\psi_i, i = 0, \ldots, n-1$, i.e., it holds that

$$\boldsymbol{P}\psi_i = \lambda_i \psi_i \;\; \forall\, i = 0, \ldots, n-1.$$

Moreover, $[\psi_0]_j = c$ for some $c \in \mathbb{R}$, for all $j = 1, \ldots, n$, i.e., $\psi_0$ is *constant* over $\mathcal{X}$. With the help of these eigenvectors, we can write the diffusion distance as

$$\mathrm{DiffDist}_t^2(\boldsymbol{x}_i, \boldsymbol{x}_j) = \sum_{k=1}^{n-1} \lambda_k^{2t} \left([\psi_k]_i - [\psi_k]_j\right)^2.$$

This is summarized in the following theorem from [CL06].

**Theorem 5.2.1** (Coifman, Lafon 2006)**.** *For an arbitrary, fixed $t \in \mathbb{N}$, let the diffusion map $\Psi_t : \mathcal{X} \to \mathbb{R}^{n-1}$ be defined by*

$$\Psi_t(\boldsymbol{x}_i) := \begin{pmatrix} \lambda_1^t [\psi_1]_i \\ \vdots \\ \lambda_{n-1}^t [\psi_{n-1}]_i \end{pmatrix}$$

*for all $i = 1, \ldots, n$. Then*

$$\|\Psi_t(\boldsymbol{x}_i) - \Psi_t(\boldsymbol{x}_j)\|_2 = \mathrm{DiffDist}_t(\boldsymbol{x}_i, \boldsymbol{x}_j).$$

This theorem tells us that the diffusion map $\Psi_t$ embeds the data into $\mathbb{R}^{n-1}$ in a distance preserving way if we take the diffusion distance as our underlying metric. This is not surprising since it is always possible to embed $n$ data points of arbitrary dimension into $\mathbb{R}^{n-1}$ in a distance preserving way. However, since the absolute value of the eigenvalues $\lambda_i$ is non-increasing for growing $i$, we can additionally truncate the vectors $\Psi_t(\cdot)$ after $q < n-1$ entries if the remaining $i = q+1, \ldots, n-1$ values $|\lambda_i|$ are small. In this way, we embed the data $\mathcal{X}$ into $\mathbb{R}^q$ while still approximately preserving the diffusion distance. The corresponding low-dimensional representation of $\boldsymbol{x}_i$ is then given by the first $q$ coordinates of $\Psi_t(\boldsymbol{x}_i)$, i.e.,

$$\boldsymbol{x}_i \longrightarrow \begin{pmatrix} [\Psi_t(\boldsymbol{x}_i)]_1 \\ \vdots \\ [\Psi_t(\boldsymbol{x}_i)]_q \end{pmatrix} = \begin{pmatrix} \lambda_1^t [\psi_1]_i \\ \vdots \\ \lambda_q^t [\psi_q]_i \end{pmatrix}.$$

The complete diffusion maps approach is given in Algorithm 6. Here, $\mathrm{diag}(\boldsymbol{v})$ for $\boldsymbol{v} \in \mathbb{R}^n$ denotes a diagonal matrix with entries

$$\mathrm{diag}(\boldsymbol{v})_{i,i} = [\boldsymbol{v}]_i. \tag{5.4}$$

Note that we added in lines 4 and 5 of Algorithm 6 an option to set the diagonal of $K^{(\alpha)}$ to zero. This is a common technique in biological cell development analysis, which we will deal with in the upcoming tasks. Here, one is interested in transition probabilities between different data points only and not in the on-point potentials imposed by local densities; see [HBT15] for

**Algorithm 6:** Diffusion maps algorithm

**Input:** data $\mathcal{X}$, $\alpha \in [0,1]$, dimension $q < \min(n,d)$, zeroDiag $\in \{\text{False}, \text{True}\}$.
**Output:** low-dimensional data representation $\mathcal{Y}$ of dimension $q$.

$\boldsymbol{K} \leftarrow [K(\boldsymbol{x}_i, \boldsymbol{x}_j)]_{i,j=1}^n$ with kernel function $K : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$.
$\boldsymbol{Q} \leftarrow \operatorname{diag}(\boldsymbol{K}\mathbf{1})$ with $\mathbf{1} = (1, \ldots, 1)^T \in \mathbb{R}^n$. ▷ see (5.4)
$\boldsymbol{K}^{(\alpha)} \leftarrow \boldsymbol{Q}^{-\alpha}\boldsymbol{K}\boldsymbol{Q}^{-\alpha}$. ▷ see (5.1)
**if** zeroDiag = True **then**
  $\boldsymbol{K}^{(\alpha)}_{i,i} \leftarrow 0$ for all $i = 1, \ldots, n$.
**end if**
$\boldsymbol{D}^{(\alpha)} \leftarrow \operatorname{diag}(\boldsymbol{K}^{(\alpha)}\mathbf{1})$. ▷ see (5.4)
$\boldsymbol{P} \leftarrow (\boldsymbol{D}^{(\alpha)})^{-1}\boldsymbol{K}^{(\alpha)}$. ▷ see (5.2)
Compute first $q+1$ eigenvalues $\{\lambda_l\}_{l=0}^q$ and corresponding eigenvectors $\{\psi_l\}_{l=0}^q$ of $\boldsymbol{P}$.
$\begin{pmatrix} \boldsymbol{\eta}_1 & \boldsymbol{\eta}_2 & \cdots & \boldsymbol{\eta}_n \end{pmatrix} \leftarrow \begin{pmatrix} \lambda_1\psi_1 & \lambda_2\psi_2 & \cdots & \lambda_q\psi_q \end{pmatrix}^T \in \mathbb{R}^{q \times n}$.
$\mathcal{Y} \leftarrow \{\boldsymbol{\eta}_i\}_{i=1}^n$.
**return** $\mathcal{Y}$.

more details. In fact, for recovering the structure of the data set based on the graph random walk and relations between different data points, it can be disadvantageous to have large nonzero entries on the diagonal of the transition matrix. In this case, setting its diagonal to zero is known to achieve better results.

## 5.3 ▪ Clustering

Besides dimensionality reduction methods like PCA and diffusion maps, we have already briefly mentioned clustering as another unsupervised learning approach in Section 1.2. The idea behind clustering is that we want to separate or segment the data set $\mathcal{X}$ into $k \in \mathbb{N}$ different subsets, called *clusters*, $C_1, \ldots, C_k \subset \mathcal{X}$. The data points within each cluster should be more closely related to each other than to data points in other clusters. Figure 5.3 illustrates the result of a

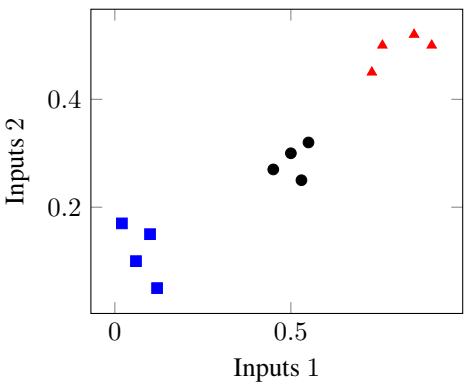



Figure 5.3: Application of clustering to a two-dimensional data set. The result is a partitioning of the data set into three clusters, indicated by the blue squares, the black circles, and the red triangles.

clustering algorithm on a two-dimensional data set. Note here that the quality of a clustering result of any given method is usually hard to evaluate [HTF09, Mur22, XW08]. In particular, every time clustering is performed on a data set we obtain a result, but it could be completely meaningless, e.g., due to noise effects; see [JWHT21].

While clustering does not reduce the dimensionality of the data set, it often goes hand in hand with dimensionality reduction algorithms when it comes to detecting and visualizing the most important properties of the data. We will discuss this combination in Section 5.4 in more detail.

In the remainder of this section, we will briefly introduce two of the most important clustering algorithms. First, we discuss *$k$-means clustering*, whose intrinsic similarity measure is based solely on the metric/distance measure of the surrounding space. Subsequently, we have a closer look at *spectral clustering*, which employs a data-dependent, intrinsic distance measure within $k$-means.

Due to the lack of an underlying model class in clustering algorithms, we cannot easily categorize a clustering method as being either linear or nonlinear. However, since $k$-means relies solely on Euclidean distance computations and on the averaging of data points, it could be considered a linear method. In contrast to this, spectral clustering is inherently based on the result of the spectral decomposition of the transition probability matrix $P$ of the neighborhood graph from the previous section. Thus, it can be considered to be a nonlinear method.

**$k$-means clustering**

***k-means*** is the most famous and straightforward way to cluster a data set. It relies on searching $k$ ***centroids*** $\boldsymbol{c}_1, \ldots, \boldsymbol{c}_k \in \mathbb{R}^d$, which serve as centers for the corresponding clusters $C_1, \ldots, C_k$. A data point $\boldsymbol{x}_i$ is then assigned to the cluster $C_j$ for which the Euclidean distance $\|\boldsymbol{x}_i - \boldsymbol{c}_j\|_2$ is smaller than the distance to any other centroid. In summary, the $k$-means algorithm aims to solve the minimization problem

$$\min_{\substack{\boldsymbol{c}_1,\ldots,\boldsymbol{c}_k \in \mathbb{R}^d \\ C_1,\ldots,C_k \subset \mathcal{X} \text{ disjoint} \\ \bigcup_{i=1}^k C_k = \mathcal{X}}} \sum_{i=1}^{k} \sum_{\boldsymbol{x} \in C_i} \|\boldsymbol{x} - \boldsymbol{c}_i\|_2^2.$$

The minimization objective is the so-called *within-cluster sum of squared errors*.

A famous approach to solve the $k$-means problem of minimizing the within-cluster sum of squared errors approximately is *Lloyd's algorithm* [Llo82]. It follows the heuristic of alternatingly minimizing the within-cluster sum according to the centroids and according to the cluster membership. To this end, cluster assignments are computed by determining the nearest centroid for each data point in the first part of an iteration step of the algorithm. In the second part of an iteration step the centroids are reassigned by computing the average of all data points within one cluster. A detailed description can be found in Algorithm 7. Note that the employed simple random initialization of the centroids can be improved by using certain heuristics; see, e.g., [VA06] for the so-called *$k$-means++* initialization. Furthermore, $k$-means can also be applied with non-Euclidean distance measures such as the *Procrustes* distance; see [ADLS10], for example.

While $k$-means is the most prominent clustering algorithm, its application can become problematic for high-dimensional data because the Euclidean distance might not be a good distance quantity due to the concentration of measure effect; see Section 1.6. Here, *spectral clustering* becomes a good alternative.

**Spectral clustering**

A ***spectral clustering*** algorithm is based on a graph representation of the data, e.g., a neighborhood graph. Subsequently, we define a similarity matrix $\boldsymbol{K}$, e.g., via the Gaussian kernel as in diffusion maps. From this we build either the normalized (random walk) *graph Laplacian* matrix $\boldsymbol{L}_{rw}$ on the data graph (see Algorithm 8 for its computation) or the transition matrix $\boldsymbol{P}$ from diffusion maps. After ordering the eigenvalues $\lambda_1, \ldots, \lambda_{n-1}$ and eigenvectors $\psi_1, \ldots, \psi_{n-1}$ of either $\boldsymbol{I} - \boldsymbol{L}_{rw}$ or $\boldsymbol{P}$ we determine the differences between consecutive eigenvalues and take a $k < n$ with relatively large[24] $\lambda_{k-1} - \lambda_k$ to be the number of clusters. Then, the standard clustering algorithm $k$-means is run on the entries of the eigenvectors $\psi_1, \ldots, \psi_{k-1}$; see Algorithm 9 for details.

**Algorithm 7:** The $k$-means algorithm by Lloyd

**Input:** data $\mathcal{X}$, number of clusters $k$.
**Output:** clusters $C_1, \ldots, C_K$.

Initialize $\boldsymbol{c}_1, \ldots, \boldsymbol{c}_k$ by drawing $k$ different random samples from $\mathcal{X}$.
**while** not converged **do**
  $C_1, \ldots, C_k \leftarrow \emptyset$.
  **forall** $i = 1, \ldots, k$ **do**
    $C_i \leftarrow \left\{ \boldsymbol{x} \in \mathcal{X} \;\middle|\; \|\boldsymbol{x} - \boldsymbol{c}_i\|_2 \leq \|\boldsymbol{x} - \boldsymbol{c}_j\|_2 \; \forall \, j \neq i \right\}$. ▷ Ties broken randomly.
  **end forall**
  **forall** $i = 1, \ldots, k$ **do**
    $\boldsymbol{c}_i \leftarrow \frac{1}{|C_i|} \sum_{\boldsymbol{x} \in C_i} \boldsymbol{x}$.
  **end forall**
**end while**

**Algorithm 8:** Computation of the normalized random walk graph Laplacian $\boldsymbol{L}_{rw}$

**Input:** data $\mathcal{X}$.
**Output:** normalized random walk graph Laplacian $\boldsymbol{L}_{rw}$.

Construct a graph $\mathcal{G}$ on the data set $\mathcal{X}$, e.g., a $k$-nearest neighbor graph or a fully connected graph.
Compute the weight matrix $\boldsymbol{K}$, e.g., using the Gaussian kernel $K$, i.e.,

$$[\boldsymbol{K}]_{ij} := \begin{cases} K(\boldsymbol{x}_i, \boldsymbol{x}_j) & \text{if there is an edge between } \boldsymbol{x}_i \text{ and } \boldsymbol{x}_j \text{ in } \mathcal{G}, \\ 0 & \text{else.} \end{cases}$$

$\boldsymbol{Q} \leftarrow \operatorname{diag}(\boldsymbol{K}\boldsymbol{1})$ with $\boldsymbol{1} = (1, \ldots, 1)^T \in \mathbb{R}^n$; see (5.4).
Compute the random walk graph Laplacian $\boldsymbol{L}_{rw} \leftarrow \boldsymbol{I} - \boldsymbol{Q}^{-1}\boldsymbol{K}$.

Note that, for a fully connected graph $\mathcal{G}$ in Algorithm 8 and for $\alpha = 0$ and zeroDiag = False in Algorithm 6, $\boldsymbol{I} - \boldsymbol{L}_{rw}$ equals $\boldsymbol{P}$. Thus, employing $\boldsymbol{P}$ for spectral clustering instead of $\boldsymbol{L}_{rw}$, which is the classical variant, gives us more flexibility by means of using different values for

[24] This is also known as the *spectral gap*.

**Algorithm 9:** Spectral clustering on either $\boldsymbol{M} = \boldsymbol{I} - \boldsymbol{L}_{rw}$ or $\boldsymbol{M} = \boldsymbol{P}$

**Input:** data $\mathcal{X}$.
**Output:** number of clusters $k$, clusters $C_1, \dots, C_K$.

Compute the eigenvalues $1 = \lambda_0 > \lambda_1 \geq \cdots \geq \lambda_{n-1}$ of $\boldsymbol{M}$.
Determine $k < n$ such that $\lambda_{k-1} - \lambda_k$ (*spectral gap*) is large.
Compute the corresponding eigenvectors $\psi_1, \dots, \psi_{k-1}$ of $\boldsymbol{M}$.
**forall** $i = 1, \dots, n$ **do**
  $\tilde{\boldsymbol{x}}_i \leftarrow \left([\psi_1]_i, \dots, [\psi_{k-1}]_i\right)^T$.
**end forall**
Run $k$-means on $\{\tilde{\boldsymbol{x}}_i \mid i = 1, \dots, n\}$ to determine clusters $\tilde{C}_1, \dots, \tilde{C}_k$.
**forall** $j = 1, \dots, k$ **do**
  $I_j \leftarrow \left\{i \in \{1, \dots, n\} \mid \tilde{\boldsymbol{x}}_i \in \tilde{C}_j\right\}$.
  $C_j \leftarrow \{\boldsymbol{x}_i \mid i \in I_j\}$.
**end forall**

$\alpha$, which resemble different random walk dynamics as discussed before. Moreover, note that a mathematical connection between graph theory and spectral clustering with $\boldsymbol{L}_{rw}$ can be drawn: One can show that, for $k = 2$, the resulting segmentation is equivalent to a *min-cut* on the underlying graph with weights $K(\boldsymbol{x}_i, \boldsymbol{x}_j)$ for all $i, j = 1, \dots, n$. The reason for this is that the data are clustered according to the eigenvectors of a Markov chain transition matrix. More details on spectral clustering and its mathematical and graph-theoretical background can be found in [vL07].

## 5.4 ▪ Visualization of high-dimensional data

As we have already seen throughout this book, a data set can be explored by visualizing it in various ways. For instance, we discussed the level set and scatter plots in Chapter 3 to visualize the optimal separating hyperplane and the application of a nonlinear feature map on the data set. However, if the data are high-dimensional, a visualization like that is not straightforward. Here, dimensionality reduction and clustering methods can be nicely combined to visualize a data set and to obtain an overview on similarities and dissimilarities between data points.

As we have explored in the tasks of Chapter 4, we can employ PCA to reduce the data dimensionality to two or three in order to visualize the result in a scatter plot. Of course this procedure can also be applied for other dimensionality reduction methods, e.g., for diffusion maps. Furthermore, in order to visualize similarities between data points, we can employ clustering algorithms, e.g., $k$-means or spectral clustering, on the original (high-dimensional) data set and visualize the cluster assignments via color coding in the scatter plot of the dimension-reduced data. We will explore this possibility in Section 5.5.5 in more detail. A schematic overview of this specific visualization technique is depicted in Figure 5.4. For an overview on visualization methods, especially for high-dimensional data, we refer the reader to the surveys [DOL03, LMW+17].

## 5.5 ▪ Tasks on nonlinear dimensionality reduction

Use the JUPYTER notebook template `DiffusionMaps_template.ipynb` which is provided at https://bookstore.siam.org/di03/bonus for the tasks in this section. First, let us consider Isomap; see Algorithm 5 for a simple 3D data set.

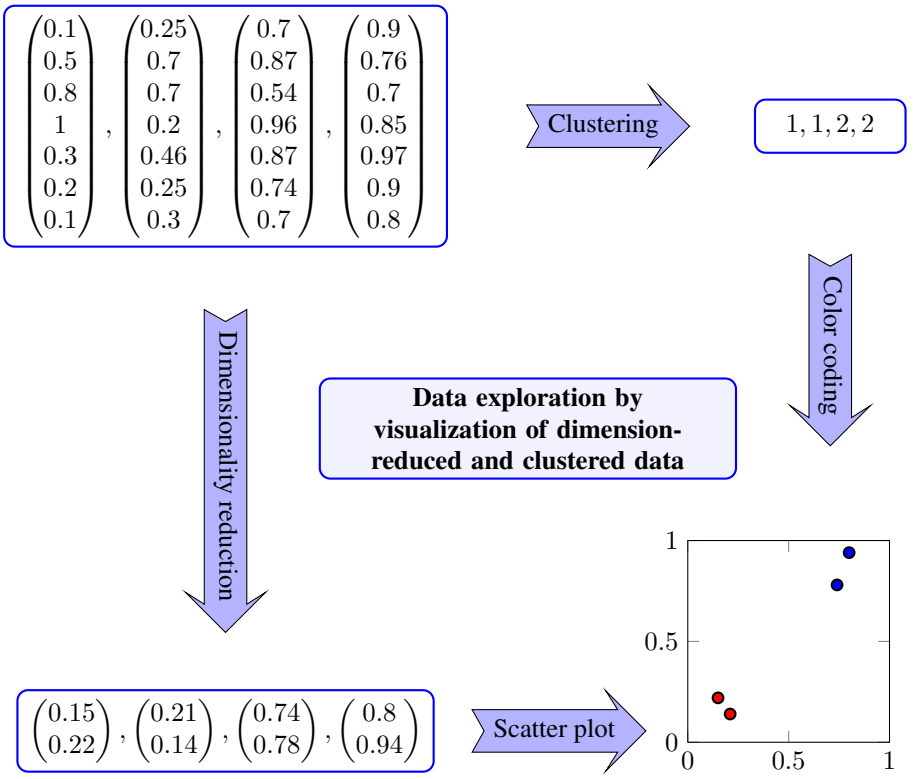



Figure 5.4: Schematic overview of data visualization via dimensionality reduction and clustering. First, the four high-dimensional input vectors (upper left) are processed by dimensionality reduction and with a clustering algorithm, respectively. Then, the resulting low-dimensional coordinates (lower left) are plotted in a scatter plot (lower right), where the cluster labels (upper right) are used for the color coding. Here, the first label is depicted by red, and the second label is depicted by blue.

**Task 5.1.** Use the Isomap algorithm (`sklearn.manifold.Isomap`) from SCIKIT-LEARN to analyze the *Swiss roll* data set. To this end, load and visualize 2000 sample points from the 3D Swiss roll data set with the help of the `sklearn.datasets.make_swiss_roll` routine. Embed the loaded data into 2D using Isomap with a 10-nearest neighbors graph. Compare the resulting embedding to a 2D PCA embedding. Use the color coding provided by the `make_swiss_roll` routine to interpret the results.

As we can see in Figure 5.5, the Isomap algorithm nicely captures the intrinsic two-dimensional structure of the Swiss roll. However, for practical applications, in particular with noisy data, diffusion maps often performs better than Isomap. Therefore, let us now consider the diffusion maps algorithm.

**Task 5.2.** Implement the diffusion maps method from Algorithm 6. To this end, you can use the `scipy.spatial.distance.pdist` function to efficiently compute the pairwise distances needed to evaluate the Gaussian kernel function.

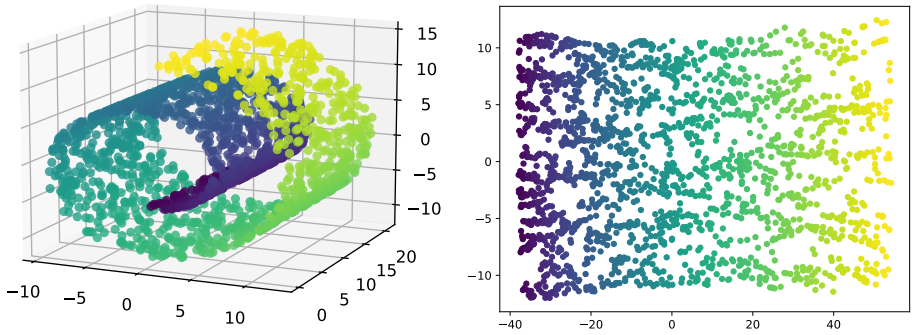



Figure 5.5: The Swiss roll data set (left) and its 2D embedding after the application of Isomap (right). The color coding serves to illustrate that Isomap nicely detects the intrinsic structure of the data set.

To study the performance of diffusion maps, we will explore its behavior on biological single-cell data. To this end, we start with the single-cell sequencing data set from [BNC+15] and subsequently analyze the *Guo* data set from [GHT+10]. These data sets contain certain measurements from genes of mouse embryonic stem cells at different developmental stages, which we will discuss in more detail later. We will explore how data preprocessing and hyperparameter choices influence the results of diffusion maps on this data set. Our overall goal is to detect the transition between different developmental stages by visualizing clustering results of the data set.

For the following exercises, we will always use the zeroDiag = True option of Algorithm 6 since we are interested in the inter-cell transitions. Let us begin with the data set from [BNC+15]. It contains 182 data points, which are subdivided into three different groups. Each data point has a dimension of 8989. While the data set has an underlying biological background, this is not relevant for the following two tasks. We will use this data set just as a first example to show that real-world data sets might contain nonlinear structures, which the PCA cannot capture well.

**Task 5.3.** A routine to load the data set from [BNC+15] has already been implemented in the template notebook we provide for this chapter. Now, reduce the dimension of the data to three by using diffusion maps. Use the Gaussian kernel with parameter $\sigma = 20$ and use $\alpha = 1$ in Algorithm 6. Visualize the result in a three-dimensional scatter plot ($q = 3$), i.e., plot the (scaled) second, third, and fourth eigenvectors of the transition matrix $\boldsymbol{P}$ from Algorithm 6 against each other. Do not forget to color (or label) your resulting points in the plot according to the given labels.

**Task 5.4.** Repeat Task 5.3 and employ the PCA and Isomap instead of diffusion maps to reduce the data set dimension. You can use your own PCA implementation from the last chapter or the one from SCIKIT-LEARN. Compare the PCA and Isomap results with the ones achieved with diffusion maps in Task 5.3.

### 5.5.1 ▪ Single-cell data analysis

In recent years, dimensionality reduction methods have become popular to extract valuable information from high-dimensional biological data. Biologists aim to discover how single cells (e.g., stem cells) differentiate over time and which developmental stages they pass. For this, cell

data are collected from different developmental time points and are then combined into a single data set. For each cell, gene expression analysis is done by measuring the expression intensity of several genes. However, the high number of genes measured for each cell often makes it difficult for biologists to detect cell differentiation progressions. Dimensionality reduction methods can help to extract information by projecting the data into a lower-dimensional space. If this so-called *embedding space* is two- or three-dimensional, the data can be visualized as described in Section 5.4. Different cell groups in the data should then be recognizable as different clusters in the embedding space.

**The Guo data set** In the following, we will apply diffusion maps to the *Guo* data [GHT+10]. First, let us study the structure of this data set in more detail. The measurements contained in it are single-cell qPCR *Ct*-values for $48$ genes of $442$ mouse embryonic stem cells at seven different developmental stages from zygote to blastocyst. The details of qPCR Ct-values will be explained below, after the description of this data set. Starting from the one-cell stage, cells transition smoothly either to the trophectoderm (TE) lineage or to the inner cell mass (ICM). Subsequently, cells transition from the inner cell mass either to the primitive endoderm (PE) or to the epiblast (EPI) lineage.

In Table 5.12, we can see an exemplary excerpt of the Guo data set. In the first row, the names of the measured genes (ranging from Actb to Tspan8) are given. The naming annotation in the first column refers to the embryonic stage, embryo number, and individual cell number; thus 64C 7.14 refers to the 14th cell harvested from the seventh embryo collected from the 64-cell stage. In the following, we are only interested in the embryonic stage of the cells, which is given by the first number (e.g., 64C).

| Cell | Actb | Ahcy | Aqp3 | ⋯ | Gapdh | ⋯ | Tspan8 |
|---|---|---|---|---|---|---|---|
| 1C 1.1 | 14.01 | 19.28 | 23.89 | ⋯ | 16.21 | ⋯ | 18.53 |
| 1C 1.2 | 13.68 | 18.56 | 28.00 | ⋯ | 15.69 | ⋯ | 18.29 |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋱ | ⋮ | ⋱ | ⋮ |
| 64C 7.14 | 13.78 | 25.46 | 20.79 | ⋯ | 17.43 | ⋯ | 18.47 |

Table 5.12: Table of the raw Guo data set. Each of the $n = 442$ rows contains the Ct-values for a specific cell at a certain developmental stage. Each of the $G = 48$ columns contains the Ct-values for a specific gene.

**Ct-measurements** To gain the actual data values presented in Table 5.12, a qPCR (real-time quantitative polymerase chain reaction) was conducted, which consists of several cycles. At each cycle, the amount of fluorescence is measured. A Ct-value (abbreviation for cycles-to-threshold) is then defined as the number of cycles for which the fluorescence significantly exceeds the background fluorescence, i.e., at which a clear fluorescence signal is first detected. Thus, a higher Ct-value means a lower DNA or gene concentration. For the Guo data set, a total of $28$ cycles of qPCR were performed. All genes that would need more cycles to exceed the background fluorescence were assigned the threshold value $28$. For further details on the analysis of single-cell development data, Ct-values, and qPCR data, we refer the reader to [BNC+15, GGF09, GHT+10, Hea23].

## 5.5.2 ▪ Preprocessing

To ensure an accurate and meaningful analysis, data sets often require preprocessing techniques, such as data cleaning, handling missing or uncertain values, and data normalization; see also

Section 2.5. In the following, we will learn how to preprocess the Guo data set. To this end, let us denote the raw data set by

$$\mathcal{X}_{\text{raw}} = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n\} \subset \mathbb{R}^G,$$

where $n = 442$ is now the number of cells, i.e., the number of rows of Table 5.12, and $G = 48$ is the number of genes, i.e., the number of columns of Table 5.12. This means that $[\boldsymbol{x}_i]_j$ is the expression value of the $j$th gene of the $i$th cell. For the raw Guo data set, we have the following information:

- Cells from the 1-cell stage embryos were treated differently in the experimental procedure.
- Genes that would need more than 28 cycles to exceed the background fluorescence were assigned the threshold value 28, as mentioned above. However, there still exist a few entries larger than 28 in the raw data set, which indicate undetectable data. Next, these data need to be deleted from the data set.

**Cleaning the data** Since cells from the 1-cell stage and cells with at least one entry larger than the threshold value have to be excluded from the analysis, a proper cleaning of the data will be our first step. The resulting cleaned data are given by

$$\mathcal{X} = \mathcal{X}_{\text{raw}} \setminus \Big( \mathcal{X}_{1C} \cup \big\{ \boldsymbol{x}_i \in \mathcal{X}_{\text{raw}} \mid \max_{j=1,\ldots,G} [\boldsymbol{x}_i]_j > 28 \big\} \Big),$$

where $\mathcal{X}_{1C}$ denotes the set of cells from the 1-cell stage.

**Normalizing the data** Next, we need to normalize the data in order to obtain more accurate results. A common strategy in biology is the normalization via reference genes. In our case, we subtract for each cell the mean expression of the endogenous control genes *Actb* and *Gapdh*. Moreover, we now also exclude entries which are identical to the threshold value 28, i.e.,

$$[\boldsymbol{x}_i]_j \leftarrow [\boldsymbol{x}_i]_j - \frac{1}{2}([\boldsymbol{x}_i]_{g_{\text{Actb}}} + [\boldsymbol{x}_i]_{g_{\text{Gapdh}}})$$

for all $i = 1, \ldots, n$ and $j = 1, \ldots, G$ for which $[\boldsymbol{x}_i]_j \neq 28$. Here, $g_{\text{Actb}}$ and $g_{\text{Gapdh}}$ denote the indices of the genes Actb and Gapdh, respectively.

**Rescaling the data** Subsequently, we need to set the entries with threshold value 28 to a new baseline. We define this baseline as the smallest integer greater than or equal to the maximum of the normalized data set, i.e., $\lceil \max_{i,j} \{ [\boldsymbol{x}_i]_j \mid [\boldsymbol{x}_i]_j \neq 28 \} \rceil$.

**Task 5.5.** Preprocess the Guo data set as described above by cleaning, normalizing, and rescaling it. Finally, round all entries to three digits.

Now we are ready to apply the diffusion maps algorithm to the resulting data set.

**Task 5.6.** Perform a diffusion maps analysis of the preprocessed Guo data set for the Gaussian kernel with $\sigma = 10$ and $\alpha = 1$ in Algorithm 6 and visualize the embedding in a two-dimensional scatter plot ($s = 2$). Interpret your result. Can you assign the branches revealed in the plot to the lineages of the Guo data set described in Section 5.5.1?

**Task 5.7.** Perform a diffusion maps analysis of the Guo data set with the same parameters as in Task 5.6, but without full preprocessing (still remove the cells with undetectable data and round all entries to three digits) and compare your result with the plot from Task 5.6.

### 5.5.3 ▪ Further dimensionality reduction methods

We have seen that preprocessing is an important step in data analysis. Thus, from now on, we use the fully preprocessed Guo data set. In the following, we want to compare the diffusion maps performance on the Guo data set to other dimensionality reduction methods, namely the PCA and *t-SNE*.

**t-SNE**

The general idea behind ***t-distributed stochastic neighbor embedding*** (t-SNE) is that the similarity between two high-dimensional points $\boldsymbol{x}_j$ and $\boldsymbol{x}_i$ is measured by the sum of Gaussian distances $\exp(-\|\boldsymbol{x}_i - \boldsymbol{x}_j\|^2/\sigma_i^2) + \exp(-\|\boldsymbol{x}_i - \boldsymbol{x}_j\|^2/\sigma_j^2)$ for certain bandwidths $\sigma_i, \sigma_j$ for $i, j = 1, \ldots, n$. Analogously, for the low-dimensional representatives $\boldsymbol{\eta}_i$ and $\boldsymbol{\eta}_j$, similarities are measured by a *student t-distribution*; see also [Geo12]. To determine the points $\boldsymbol{\eta}_i$, the Kullback–Leibler divergence (see Chapter 7 for a definition) between the Gaussian distribution and the student t-distribution is minimized. More details can be found in [VdMH08].

We will use the existing t-SNE implementation from SCIKIT-LEARN in the following.

**Task 5.8.** Embed the preprocessed Guo data set by using principal component analysis and t-SNE. You can use the corresponding implementations from SCIKIT-LEARN. Compare the results with the diffusion maps embedding from Task 5.6. Compare the computation times of the dimensionality reduction methods as well.

### 5.5.4 ▪ Hyperparameter selection

Up to now, we have used the bandwidth $\sigma = 10$ for the diffusion maps analysis, which has given a reasonable result. Nevertheless, the outcome might improve for a different (i.e., better) choice of $\sigma$. But hyperparameter selection is a difficult task in machine learning algorithms, as we have already seen in Chapter 3, and is even more difficult in unsupervised learning, where there are no clear performance measures.

**Task 5.9.** Compare the diffusion maps embedding of the Guo data set for several bandwidths $\sigma$.

Lafon [CL06] proposed a rule for the choice of a good value for $\sigma$ as

$$\sigma = \sqrt{\frac{1}{2n} \sum_{i=1}^{n} \min_{j \neq i} \{\|\boldsymbol{x}_i - \boldsymbol{x}_j\|^2\}}. \tag{5.5}$$

The radicand indicates half of the average of all nearest neighbor distances in the data set.

**Task 5.10.** Implement the rule (5.5) for the bandwidth $\sigma$. Plot the embedding for the Guo data set with the bandwidth chosen by this rule.

### 5.5.5 ▪ Cell group detection

So far, we had to determine the cell groups and lineages by looking at the plots from the previous tasks. We now aim to identify the cell lineages by using clustering. To this end, we perform spectral clustering on the transition matrix $\boldsymbol{P}$ from diffusion maps.

**Task 5.11.** Implement the spectral clustering method from Algorithm 9 with the transition matrix $\boldsymbol{P}$ from diffusion maps and using $k$-means for clustering (from SCIKIT-LEARN) for a given number of clusters $k$.

**Task 5.12.** Plot the largest 20 (ordered) eigenvalues of the transition matrix $\boldsymbol{P}$ for the preprocessed Guo data set and identify $k$ by determining the biggest spectral gap (use the parameters from Task 5.6).

**Task 5.13.** Employ the spectral clustering algorithm for the Guo data set with $k$ from Task 5.12 and plot the resulting points/clusters in 2D. Interpret your results. Does the clustering detect the different cell stages?

## 5.6 ▪ Further topics

**Other nonlinear dimensionality reductions methods** Besides Isomap and diffusion maps there is a plethora of other nonlinear dimensionality reduction methods based on various ideas on how to obtain a suitable low-dimensional embedding of the unlabeled data $\mathcal{X}$. We refer the reader to [LV07] for an overview and some mathematical background.

Besides Isomap, several approaches have been successfully employed which use neighborhood graphs based on specific local Euclidean distances in the original space. Examples are the *Laplacian eigenmaps* and *Maximum Variance Unfolding* (MVU) algorithms; see [LV07]. While Laplacian eigenmaps works similarly to diffusion maps, MVU solves a regularized shortest path problem. The goal of MVU is to obtain a Euclidean distance matrix for the embedded points which is closest to the graph distance matrix [PG12]. Parallel Transport Unfolding [BYF$^+$19], which is similar to Isomap, also works with shortest path distances. Here, geodesic distances are approximated using parallel transport along shortest paths on the neighborhood graph. This leads to a more robust and more accurate approximation of geodesic distances than that for Isomap.

Moreover, two commonly used but more recent methods are the already discussed t-SNE [VdMH08] (see Section 5.5.3) and Uniform Manifold Approximation and Projection (UMAP) [MHSG18]. They do not aim to preserve high-dimensional distances, but aim to preserve the local neighborhood of each data point in the embedding, while allowing more distortions for long range distances. As outlined in Section 5.5.3, t-SNE seeks to minimize the difference between specific probability distributions for the high- and low-dimensional data representations. On the other hand, UMAP models the data manifold with a fuzzy topological structure. Here, the embedding is found by searching for the low-dimensional projection of the data that has the closest

possible equivalent fuzzy topological structure. Furthermore, density preserving extensions for both t-SNE and UMAP were introduced in [NBC21].

**Other clustering methods** Apart from $k$-means and spectral clustering there are a variety of other clustering methods such as hierarchical (agglomerative and divisive) clustering, which successively builds a hierarchy of clusters, and density-based clustering algorithms like DBScan, which detect areas of large data density and combine them into clusters. For a discussion of different clustering approaches, we refer the reader to [XW08].

# Chapter 6
# Deep Neural Networks

We now focus on the model class of *artificial neural networks* and especially on so-called *deep neural networks*. This class constitutes the present state of the art when it comes to large-scale machine learning problems (many data points) and has proven to be very successful for various applications, e.g., in signal processing or image recognition; see [Agg18, GBC16], for example.

This chapter is structured as follows. In Section 6.1 we first introduce the model class of (deep) neural networks and their graphical representations. Section 6.2 is dedicated to the study of the approximation properties of the model class. There, we investigate how well the model class is suited to approximate functions from familiar function spaces, e.g., continuous functions or functions with smooth derivatives. Subsequently, we have a look at the loss function and the resulting optimization problem when employing neural network models in Section 6.3. Furthermore, we introduce an efficient algorithm to (approximately) minimize the loss there. Section 6.4 deals with a special subclass of neural networks, namely *weight-reduced* and *convolutional* neural networks. In Section 6.5 we study the algorithm from Section 6.3 in more detail and introduce variants thereof that have proven to be successful in practical applications. Finally, Section 6.6 presents tasks on deep neural network implementations, where we dive into the PYTHON library KERAS.

## 6.1 ▪ Feed-forward neural networks

In the following, we introduce the model class of artificial neural networks. Note that this section is solely dedicated to the model class itself. We will discuss the loss function and the corresponding optimization problem in Section 6.3 in detail.

**(Artificial) neural networks**

The basic idea of ***artificial neural networks*** (ANNs) is motivated by how information is propagated/processed between neurons in the human brain [VC19]: Based on the state of a neuron, a signal is passed to adjacent/neighboring neurons. Depending on the importance of a neuron-neuron connection, this is done in a weighted fashion. In this book, we will focus on so-called ***feed-forward*** networks, where information is propagated in a given direction. Moreover, there are also other important classes of neural networks (NNs) such as ***recurrent*** neural networks (RNNs), which are of special interest for, e.g., time series analysis; see Section 10.3 and [Agg18, GBC16] for details.

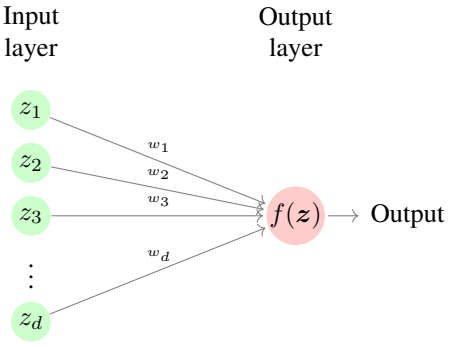



Figure 6.1: Graph representation for a single-layer neural network with $d$ input neurons and a single output neuron.

### 6.1.1 ▪ A single-layer neural network

Let us first discuss the most simple example of a single-layer feed-forward network. Commonly, NNs are represented by a directed graph. For the single-layer NN, the corresponding graph is given in Figure 6.1.

This has to be interpreted as follows: The input vector $\boldsymbol{z} = (z_1, \dots, z_d) \in \mathbb{R}^d$ is represented elementwise by the nodes/neurons of the input layer. This information is propagated to the output layer neuron, where the weighted sum with respect to the weights $w_i, i = 1, \dots, d$, is taken and a so-called bias $b$ is added, i.e.,

$$f(\boldsymbol{z}) := \sum_{i=1}^{d} w_i \cdot z_i + b.$$

Together, the weights and the bias are the parameters $\boldsymbol{p} = (w_1, \dots, w_d, b)^T$ in the parametrized model class $\mathcal{M}_{\text{1-layerNN}}$ of single-layer neural networks, i.e., these parameters are to be *learned*. This means that they need to be determined by the minimization of a loss function with respect to given training data; see Section 6.3. Note that the model class represented by this simple NN is already familiar to us: It is just the affine linear model class $\mathcal{M}_{\text{1-layerNN}} = \mathcal{M}_{\text{Lin}}$ from Section 2.1.

Moreover, a nonlinear *activation function* $\phi$ is applied to the result. For the most simple choice of the Heaviside function

$$\phi(t) := \begin{cases} 1 & \text{if } t > 0, \\ 0 & \text{else,} \end{cases}$$

we obtain the so-called *perceptron* network. It computes $f(\boldsymbol{z}) := \phi\left(\sum_{i=1}^{d} w_i z_i + b\right)$; see [Ros58]. This approach has been introduced by Rosenblatt in 1957 and was one of the first ANNs for machine learning.

**Activation functions**

An ***activation function*** $\phi : \mathbb{R} \to \mathbb{R}$ at a network node processes the information propagated to that node and thus defines the output of a node. It is applied after the summation of the weighted input information and after adding a bias, i.e.,

$$\phi\left(\sum_{i=1}^{d} w_i z_i + b\right)$$

is the output of a node with bias $b$, inputs $z_1, \dots, z_d$, and input weights $w_1, \dots, w_d$. Classical activation functions are the *hyperbolic tangent* $\phi(x) := \tanh(x)$ and the *sigmoid* function $\phi(x) := \mathrm{sigmoid}(x) = \frac{1}{1-e^{-x}}$. Nowadays, the so-called ***rectified linear unit*** (ReLU)

$$\phi(x) := \max(0, x)$$

is a popular choice because of its simple but nonlinear structure. Note that the task of finding suitable activation functions for an application at hand is still an area of active research; see, e.g., [SX22].

While there is a nonlinearity present in the Heaviside function, it only casts the real-valued output $\sum_{i=1}^{d} w_i z_i + b$ to 0 and 1 in the same fashion as a level set function; see also Section 2.2. Thus, the perceptron can only represent affine linear functions.

### 6.1.2 ▪ A two-layer neural network

Our next example for a neural network model is a *fully connected*[25] two-layer NN with $d_1 = d$ neurons in the input layer, $d_2$ neurons in the so-called *hidden layer*, and one neuron in the output layer. The corresponding graph representation can be found in Figure 6.2.

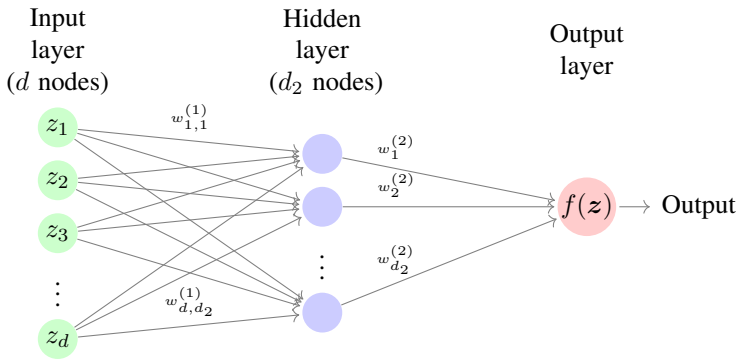



Figure 6.2: Graph representation for a two-layer neural network with $d$ input neurons, $d_2$ hidden neurons, and a single output neuron.

This network works as follows:

- Neuron $i$ of the input layer is assigned the value $z_i$.
- The weighted information $w_{i,j}^{(1)} \cdot z_i$ is propagated from neuron $i$ of the input layer to neuron $j$ of the hidden layer for all $i = 1, \dots, d$ and all $j = 1, \dots, d_2$.
- All incoming information at neuron $j$ of the hidden layer is summed up together with a bias $b_j^{(2)}$ to obtain the so-called *net sum*

$$\mathrm{net}_j^{(2)} := \sum_{i=1}^{d} w_{i,j}^{(1)} z_i + b_j^{(2)}.$$

- A (possibly nonlinear) activation function $\phi^{(2)}$ is applied to obtain

$$o_j^{(2)} := \phi^{(2)}\left(\mathrm{net}_j^{(2)}\right).$$

[25] This means that each neuron of a layer is connected to each neuron of the subsequent layer.

- The weighted information is passed to the output neuron to obtain

$$\mathrm{net}^{(3)} := \sum_{i=1}^{d_2} w_i^{(2)} o_i^{(2)} + b^{(3)}$$

and

$$f(\boldsymbol{z}) := o^{(3)} := \phi^{(3)}\left(\mathrm{net}^{(3)}\right)$$

for some activation function $\phi^{(3)}$. For regression, one usually takes $\phi^{(3)} := \mathrm{id}$ in the output layer. For classification, it is common to choose a *softmax* function, which we will introduce in more detail in Section 6.4.4.

The full model thus reads

$$f(\boldsymbol{z}) = o^{(3)} = \phi^{(3)}\left(\mathrm{net}^{(3)}\right) = \phi^{(3)}\left(\sum_{j=1}^{d_2} w_j^{(2)} \cdot \phi^{(2)}\left(\sum_{i=1}^{d} w_{i,j}^{(1)} z_i + b_j^{(2)}\right) + b^{(3)}\right)$$

for some weights $w_{i,j}^{(1)}, w_j^{(2)} \in \mathbb{R}$ and biases $b_j^{(2)}, b^{(3)} \in \mathbb{R}$ for $i = 1, \ldots, d$ and $j = 1, \ldots, d_2$. In contrast to the single-layer network model class $\mathcal{M}_{\text{1-layerNN}}$, this model class $\mathcal{M}_{\text{2-layerNN}}$ of two-layer neural networks with parameters $w_{i,j}^{(1)}$, $w_j^{(2)}$, $b_j^{(2)}$, and $b^{(3)}$ for $i = 1, \ldots, d_1$ and $j = 1, \ldots, d_2$ can now represent nonlinear functions of the input $\boldsymbol{z}$ if $\phi^{(2)}$ is a nonlinear function. Again, the model, i.e., its parameters, must now be fitted to the available training data, which we will discuss in Section 6.3.

### 6.1.3 ▪ Deep neural networks

Let us now introduce a general fully connected feed-forward neural network with $L \in \mathbb{N}$ layers.

**Deep neural networks**

The neural network model classes with one or two layers can easily be generalized to the $L$-layer case with $L-1$ hidden layers for an arbitrary $L \in \mathbb{N}$. For large $L$ (sometimes already for any $L > 2$), these networks are called ***deep neural networks*** (DNNs). Using an ML algorithm with such a DNN as a model is what the term ***deep learning*** refers to.

In an $L$-layer neural network, the weight matrices $\boldsymbol{W}^{(l)} \in \mathbb{R}^{d_l \times d_{l+1}}$ and bias vectors $\vec{b}^{(l+1)} \in \mathbb{R}^{d_{l+1}}$ for $l = 1, \ldots, L$ are the parameters of the model class $\mathcal{M}_{\text{L-layerNN}}$. Let us study how a function $f \in \mathcal{M}_{\text{L-layerNN}}$ is defined, i.e., how the network outputs $f(\boldsymbol{z})$ are computed for given weights and biases. To this end, let the values $\vec{o}^{(l)} := \left(o_1^{(l)}, \ldots, o_{d_l}^{(l)}\right)^T$ of the $l$th layer neurons be given. Note that $\vec{o}^{(1)} = \boldsymbol{z}$ is just the input of the network. We set

$$\vec{\mathrm{net}}^{(l+1)} := \left(\boldsymbol{W}^{(l)}\right)^T \cdot \vec{o}^{(l)} + \vec{b}^{(l+1)} \tag{6.1}$$

for the $l$th layer weight matrix $\boldsymbol{W}^{(l)}$ with entries $\boldsymbol{W}_{i,j}^{(l)} = w_{i,j}^{(l)}$ and the bias vector $\vec{b}^{(l+1)} = \left(b_1^{(l+1)}, \ldots, b_{d_{l+1}}^{(l+1)}\right)^T$. Slightly abusing notation, we write

$$\vec{o}^{(l+1)} := \phi^{(l+1)}\left(\mathrm{net}^{(l+1)}\right),$$

where the application of the activation function $\phi^{(l+1)}$ has to be understood elementwise. This is done for $l = 1, \dots, L$ to obtain

$$\vec{o}^{(L+1)} = o_1^{(L+1)} =: f(\boldsymbol{z})$$

for a single output layer neuron, i.e., $d_{L+1} = 1$.

This procedure of computing $f(\boldsymbol{z})$ for given weight matrices $\boldsymbol{W} := \left(\boldsymbol{W}^{(1)}, \dots, \boldsymbol{W}^{(L)}\right)$ and bias terms $\vec{b} := \left(\vec{b}^{(2)}, \dots, \vec{b}^{(L+1)}\right)$ by layerwise iterations is called *forward propagation*. Note however that we still need to optimize the weights and biases with respect to a given loss function; see Section 6.3.

Although perceptron-based neural networks, like the ones we have discussed so far, were introduced in the 1950s, they did not gain much interest back then since no efficient training algorithms existed for multi-layer architectures. This changed in the 1980s when such algorithms [Lin76, Wer82] were presented. Soon, they were employed successfully for several real-world applications; see, e.g., [LeC85, RHW86].

## 6.2 ▪ Universal approximation theorem

Now that we have seen how the class of (*fully connected*) $L$-layer neural networks is defined, we study how well functions from this model class approximate some known function classes in order to assess how well neural networks are suited for learning arbitrary functions. In the literature, the study of these approximation properties is also commonly referred to as the study of the *expressivity* of a neural network model class.

A first result on two-layer neural networks is the so-called *universal approximation theorem*. It goes back to [Cyb89] and [Hor91].

**Theorem 6.2.1** (Cybenko 1989, Hornik 1991)**.** *Let $\phi^{(2)} : \mathbb{R} \to \mathbb{R}$ be a nonconstant, bounded, and monotonically increasing function. For all $\varepsilon > 0$ and all $g \in C\left([0,1]^d\right)$, there exist $d_2 \in \mathbb{N}$ and $\vec{w}^{(2)} \in \mathbb{R}^{d_2}, \boldsymbol{W}^{(1)} \in \mathbb{R}^{d \times d_2}, \vec{b}^{(2)} \in \mathbb{R}^{d_2}$ such that*

$$\|f - g\|_\infty < \varepsilon$$

*for*

$$f(\boldsymbol{z}) := \left(\vec{w}^{(2)}\right)^T \cdot \phi^{(2)}\left(\left(\boldsymbol{W}^{(1)}\right)^T \cdot \boldsymbol{z} + \vec{b}^{(2)}\right).$$

This theorem essentially tells us that a two-layer network with an appropriate activation function is able to represent continuous functions on a compact domain with arbitrary precision. Note that, for polynomial activation functions, this statement is a trivial consequence of the classical *Stone–Weierstrass* theorem [Sto48], which states that linear combinations of polynomials can approximate continuous functions on an interval arbitrarily well.

Popular activation functions that fulfill the requirements of Theorem 6.2.1 are the hyperbolic tangent function $\phi(x) := \tanh(x)$ and the sigmoid function $\phi(x) := \frac{1}{1-e^{-x}}$. Interestingly, the commonly used ReLU function $\phi(x) := \max(0, x)$ does not meet the prerequisites of the above theorem. However, there exist more recent variants of approximation theorems for NNs, which cover the ReLU case and various other activation functions and also take the number of weights and layers into account, e.g., [Yar17, PV18, BGKP19, GK22, GP90, Mha96, Dah22, MZ22]. For a thorough survey on approximation properties of different types of neural networks, see [DHP21]. Let us have a look at the version from [PV18], which is valid for ReLU activations.

**Theorem 6.2.2** (Petersen, Voigtländer 2018)**.** *Let $\beta > 0$. There exists a $c > 0$ such that, for all $\varepsilon > 0$ and any piecewise $C^\beta$ function $g$ on $[0,1]^d$, there exists a DNN with ReLU activations and at most $\mathcal{O}\big(\varepsilon^{-\frac{2(d-1)}{\beta}}\big)$ non-zero weights and $c \cdot \log_2(\beta+2) \cdot \big(1+\frac{\beta}{d}\big)$ layers with output $f$ such that*

$$\|f - g\|_{L_2} < \varepsilon.$$

Similar results are known for functions of higher smoothness, e.g., holomorphic functions. There, even exponentially decaying bounds in $\varepsilon$ on the approximation error $\|f-g\|$ in certain Lipschitz norms can be proven to hold with similar constraints on the numbers of neurons and the numbers of layers; see [OSZ22]. To this end, the so-called *rectified power unit* (RePU) activation function $\mathrm{RePU}_p(x) := \max(0,x)^p$ with $p \geq 2$ is used in contrast to the ReLU function.

Besides approximation theorems stating how well neural networks approximate certain function spaces, the function classes that specific neural networks describe have also been studied. For two-layer feed-forward neural networks, for example, this class is called the *Barron space*. In the case of deep *residual* neural networks, which we will introduce in Section 8.2, this class is a so-called *flow-induced* function space. For more details, we refer the reader to [EMW22]. Furthermore, [Uns19] shows that determining the optimal activation functions of a neural network together with its weights and biases results in non-uniform linear spline activations,[26] i.e., the resulting network function is a piecewise linear spline with a priori unknown knot locations.

## 6.3 ▪ Training the neural network: Computing the weights and biases

Up to now, we have not answered the question of how to find optimal/good weights and biases. We have previously seen that a least squares error minimization for a supervised learning problem with a one-layer NN without nonlinear activation function leads to a system of linear equations; compare Section 2.1. However, the situation with nonlinear activations and $L$ layers is much more involved for $L \geq 2$. Here, given a loss function $\mathcal{L}$, we need to deal with a nonconvex and (usually) nonlinear minimization problem, e.g.,

$$\min_{g \in \mathcal{M}_{\text{L-layerNN}}} \mathcal{L}\left((g(\boldsymbol{x}_1), y_1), \ldots, (g(\boldsymbol{x}_n), y_n)\right)$$

for supervised learning. Note that this optimization problem might feature many local minima, even for the simple least squares loss.

To tackle such an optimization problem, iterative methods (such as gradient descent; compare Section 2.4), have proven to work well in practice. There, the fitting of the model parameters, i.e., the weights $\boldsymbol{W}^{(l)}$ and biases $\vec{b}^{(l+1)}$ for $l = 1, \ldots, L$, to given training data is iteratively done. Then, for a locally convergent method, a resulting local minimum depends on the initial choice of the model parameters, the respective minimization method (e.g., involving successive linearization steps), and the available training data set.

While there is not yet a mathematically sound theory/analysis, empirical results in the last decade have shown that gradient descent methods (see Section 2.4) are a good approach to deal with the optimization task of DNNs. In particular, we use *stochastic gradient descent*-type optimizers to numerically tackle the above optimization problem.

### 6.3.1 ▪ Stochastic minibatch gradient descent

Let us introduce the stochastic (*minibatch*) gradient descent algorithm in the following.

[26] This holds when a total variation regularization is employed.

**Stochastic gradient descent**

The ***stochastic gradient descent*** (SGD) method is a stochastic version of gradient descent from Algorithm 1. It runs the common gradient descent steps by only computing the gradient with respect to a random subset (*minibatch*) of the training data, i.e., each iterative step in the direction of the negative gradient of the loss function is taken based on the gradient on just a subset of the training data. This aims to reduce the overall costs of the gradient computations while maintaining a fast convergence of the method; see also Algorithm 10 and Section 6.5 for more details.

Throughout this section, we will stick to the least squares loss function to introduce an SGD variant for solving the corresponding optimization problem. Nevertheless, the use of other (differentiable) loss functions works in an analogous fashion. To reflect our setting, let

$$C_B(f) := \frac{1}{|B|} \sum_{i=1}^{|B|} C_i(f)$$

for a set $B \subseteq \{1, \dots, n\}$ be the so-called *minibatch loss* on $B$. Here, we define

$$C_i(f) := \tilde{\mathcal{L}}(f(\boldsymbol{x}_i), y_i) = \left(f\left(\boldsymbol{x}_i\right) - y_i\right)^2 \tag{6.2}$$

with the one-sample least squares loss $\tilde{\mathcal{L}}(z, \tilde{z}) = (z - \tilde{z})^2$. Thus $C_{\{1,\dots,n\}}(f)$ is our well-known least squares loss on the whole training data set

$$\mathcal{D} = \{(\boldsymbol{x}_i, y_i) \mid i = 1, \dots, n\}$$

of the supervised learning problem. Note that we will need to take derivatives of $C_B(f)$ with respect to all the weights $\boldsymbol{W} = \left(\boldsymbol{W}^{(1)}, \dots, \boldsymbol{W}^{(L)}\right)$ and biases $\vec{b} = \left(\vec{b}^{(2)}, \dots, \vec{b}^{(L+1)}\right)$ in order to determine the next step in a gradient descent method. It should be clear that the function $f \in \mathcal{M}_{\text{L-layerNN}}$, which the network realizes, implicitly depends on those variables. The SGD method for fixed step size $\nu > 0$, also called *learning rate*, a fixed subset size $0 < \kappa \leq n$, also called *minibatch size*, and a fixed step number $S \in \mathbb{N}$ is summarized in Algorithm 10.

Note that we can either run the algorithm for a fixed number $S$ of steps (so-called *epochs*), as described here, or use a tolerance criterion for the change of the gradients as done in Section 2.4.

The initialization of the weights and biases is usually done randomly, e.g., by $b_k^{(l)}, w_{i,j}^{(l)} \sim U\left(-\frac{1}{\sqrt{d_l}}, \frac{1}{\sqrt{d_l}}\right)$. Note however that the choice of the initial weights and biases can significantly influence the outcome and the performance of deep learning algorithms in particular cases; see, e.g., [GB10, MM15]. Note furthermore that the gradient $\nabla_{\boldsymbol{W},\vec{b}}\, C_B(f)$ is an unbiased estimator for $\nabla_{\boldsymbol{W},\vec{b}}\, C_{\{1,\dots,n\}}(f)$ in each SGD step. However, if $\kappa \ll n$, $\nabla_{\boldsymbol{W},\vec{b}}\, C_B(f)$ is much cheaper to evaluate than $\nabla_{\boldsymbol{W},\vec{b}}\, C_{\{1,\dots,n\}}(f)$. Nevertheless, to reduce the variance of this estimator, it makes sense to choose a reasonably large minibatch size $\kappa$. Usually, $\kappa$ is chosen to meet given hardware and time constraints.

The convergence properties of stochastic gradient descent are discussed in Section 6.5.

### 6.3.2 ▪ Backpropagation: Computing the gradient of $C_B(f)$

In order to run Algorithm 10, the only thing left to do is to compute

$$\nabla_{\boldsymbol{W},\vec{b}}\, C_B(f) = \frac{1}{|B|} \sum_{i \in B} \nabla_{\boldsymbol{W},\vec{b}}\, C_i(f).$$

**Algorithm 10:** Stochastic minibatch gradient descent with *learning rate* $\nu$

**Input:** learning rate $\nu > 0$, minibatch size $\kappa \leq n$, number of *epochs* $S \in \mathbb{N}$.

Initialize all weights $\boldsymbol{W}$ and biases $\vec{b}$.
**forall** $s = 1, \ldots, S$ **do**
  Subdivide $\{1, \ldots, n\}$ into $\lceil \frac{n}{\kappa} \rceil$ random disjoint subsets $B_1, \ldots, B_{\lceil \frac{n}{\kappa} \rceil}$ of size (approximately) $\kappa$.
  **forall** $B = B_1, \ldots, B_{\lceil \frac{n}{\kappa} \rceil}$ **do**
    Calculate $f(\boldsymbol{x}_i)\ \forall i \in B$ via forward propagation.
    Calculate the derivative $\nabla_{\boldsymbol{W},\vec{b}}\, C_B(f)$ with respect to all weights and biases.
    **forall** $l = 1, \ldots, L$ **do**
      **forall** $i = 1, \ldots, d_l$ and $j = 1, \ldots, d_{l+1}$ **do**
        Update each weight and bias, i.e.,

$$w_{i,j}^{(l)} \leftarrow w_{i,j}^{(l)} - \nu \frac{\partial}{\partial w_{i,j}^{(l)}} C_B(f) \text{ and } b_j^{(l+1)} \leftarrow b_j^{(l+1)} - \nu \frac{\partial}{\partial b_j^{(l+1)}} C_B(f).$$

      **end forall**
    **end forall**
  **end forall**
**end forall**

As mentioned before, we focus here on calculating $\nabla_{\boldsymbol{W},\vec{b}}\, C(f)$ for $C(f) := (f(x) - y)^2$, which resembles each $C_i$ for $i = 1, \ldots, n$, with an abstract data point $\boldsymbol{x}$. Nevertheless, the computations for other one-sample loss functions work analogously.

Now, if the input to our $L$-layer network is $\boldsymbol{z}$, then $o^{(L+1)} := o_1^{(L+1)} = f(\boldsymbol{z})$. We will show here only how to compute $\frac{\partial}{\partial w_{i,j}^{(l)}} C(f)$. Note that the calculation of $\frac{\partial}{\partial b_j^{(l+1)}} C(f)$ works in the same fashion. As a first step, we apply the chain rule to obtain

$$\frac{\partial C(f)}{\partial w_{i,j}^{(l)}} = \frac{\partial C(f)}{\partial o_j^{(l+1)}} \cdot \frac{\partial o_j^{(l+1)}}{\partial \operatorname{net}_j^{(l+1)}} \cdot \frac{\partial \operatorname{net}_j^{(l+1)}}{\partial w_{i,j}^{(l)}} = \frac{\partial C(f)}{\partial o_j^{(l+1)}} \cdot \left(\phi^{(l+1)}\right)' \left(\operatorname{net}_j^{(l+1)}\right) \cdot o_i^{(l)}.$$

Furthermore, we have

$$\frac{\partial C(f)}{\partial o_j^{(l+1)}} = \begin{cases} 2(f(\boldsymbol{z}) - y) & \text{if } l = L, \\ \sum_{i=1}^{d_{l+2}} \frac{\partial C(f)}{\partial \operatorname{net}_i^{(l+2)}} \cdot \underbrace{\frac{\partial \operatorname{net}_i^{(l+2)}}{\partial o_j^{(l+1)}}}_{=w_{j,i}^{(l+1)}} & \text{else.} \end{cases}$$

Since

$$\frac{\partial C(f)}{\partial \operatorname{net}_i^{(l+2)}} = \frac{\partial C(f)}{\partial o_i^{(l+2)}} \cdot \frac{\partial o_i^{(l+2)}}{\partial \operatorname{net}_i^{(l+2)}} = \frac{\partial C(f)}{\partial o_i^{(l+2)}} \cdot \left(\phi^{(l+2)}\right)' \left(\operatorname{net}_i^{(l+2)}\right),$$

we see that we can calculate $\frac{\partial C(f)}{\partial w_{i,j}^{(l)}}$ by iteratively working our way from layer $L$ to $L-1$ to $L-2$ and so on until we reach layer $l$. This process is called *backpropagation* or just *backprop*. To this end, we introduce

$$\vec{\delta}^{(l)} := \begin{cases} 2(f(\boldsymbol{z}) - y) & \text{if } l = L, \\ \boldsymbol{W}^{(l+1)} \cdot \left(\vec{\delta}^{(l+1)} \odot \left(\phi^{(l+2)}\right)' \left(\vec{\operatorname{net}}^{(l+2)}\right)\right) & \text{else,} \end{cases}$$

which yields

$$\nabla_{\boldsymbol{W}^{(l)}} C(f) = \vec{o}^{(l)} \cdot \left( \vec{\delta}^{(l)} \odot \left( \phi^{(l+1)} \right)' \left( \vec{\text{net}}^{(l+1)} \right) \right)^T,$$
$$\nabla_{\vec{b}^{(l+1)}} C(f) = \vec{\delta}^{(l)} \odot \left( \phi^{(l+1)} \right)' \left( \vec{\text{net}}^{(l+1)} \right)$$

for all $l = 1, \ldots, L$. Here, $\odot$ denotes the Hadamard product, i.e., the entrywise product between two vectors.

Note that it is crucial for the performance of the SGD algorithm that the derivatives of $C_B(f)$ can be computed by using basic linear algebra operations without using for-loops over $i \in B$. Furthermore, note that the ReLU activation function is not differentiable in the classical sense. However, the differentiation can be understood in a piecewise fashion here.

**Automatic differentiation**

The backpropagation procedure allows us to automatically infer the derivatives $\nabla_{\boldsymbol{W},\vec{b}}\, C_B(f)$ for different choices of activation functions and layer numbers. In contrast to numerical differentiation, where the derivative is approximated, e.g., by computing the difference quotient, the symbolic knowledge about the network is used here. In the mathematical community this is just a special case of ***automatic differentiation*** or ***autodiff***. Here, the derivative of a mathematical expression is numerically evaluated by iterative applications of the chain rule on mathematical expressions, which are built from combining basic mathematical functions, e.g., trigonometric functions or polynomials, together with basic mathematical operators, e.g., addition, division, etc. In this way, derivatives can be computed almost exactly (up to machine precision); see also [GW08, Nau11] for more details and [BPRS18] for a survey on automatic differentiation in ML.

Finally, we want to remark that the notation of the backpropagation process can differ throughout the literature. We presented above just one special way to denote it. Another possibility is a more compact gradient-type notation which is based on [GBC16]. Here, instead of storing $\vec{\delta}^{(l)}$ for $l = 1, \ldots, L$, we store the gradients $\vec{g}^{(l)}$ of the loss function with respect to the net sum. To this end, let

$$\vec{g}^{(l)} := \nabla_{\vec{\text{net}}^{(l)}} C(f) = \vec{h}^{(l)} \odot \left( \phi^{(l)} \right)' \left( \vec{\text{net}}^{(l)} \right)$$

for $l = 1, \ldots, L+1$ with

$$\vec{h}^{(l)} := \begin{cases} 2(f(\boldsymbol{z}) - y) & \text{if } l = L+1, \\ \nabla_{\vec{o}^{(l)}} C(f) = \boldsymbol{W}^{(l)} \cdot \vec{g}^{(l+1)} & \text{if } l = 1, \ldots, L. \end{cases}$$

Then, the derivatives of the loss with respect to the weights and biases can be written as

$$\nabla_{\boldsymbol{W}^{(l)}} C(f) = \vec{o}^{(l)} \cdot \left( \vec{g}^{(l+1)} \right)^T,$$
$$\nabla_{\vec{b}^{(l)}} C(f) = \vec{g}^{(l)}.$$

## 6.4 ▪ Weight reduction: Regularization

While fully connected networks are a very rich (approximation) model class, they involve a large number of parameters, and it can quickly become hard or infeasible to train them due to

overfitting problems and runtime complexity issues. Apart from smoothness regularizations and bounds on coefficient norms, as for SVM, a very successful method for regularizing NNs is the so-called *dropout* approach. Here, in each step of the iterative solver (e.g., SGD), a (hidden) node in the network is neglected with probability $0 < p < 1$. After the training, the whole network is considered again for testing. There however, the output of each node is usually multiplied by $1 - p$ such that the expected output $\mathbb{E}\big[o_i^{(l)}\big]$ is the same as in the training steps.

Apart from dropout, we could directly use a *weight-reduced* network, which has fewer parameters and, thus, less expressive power than a fully connected one, but which is often sufficient to get good results. To this end, certain neuron-neuron connections are omitted, as illustrated in Figure 6.3.

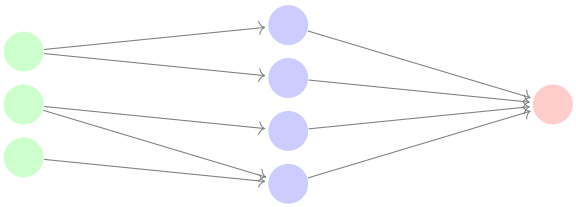


Figure 6.3: Graph representation for a two-layer neural network where weight reduction is used. Note that each input neuron is only connected to a subset of hidden neurons.

A special form of weight reduction is *weight sharing*. Here, while many (or all) neuron connections are still active in the network graph, some of them share the same weights as visualized by Figure 6.4, where shared weights have the same name and color.

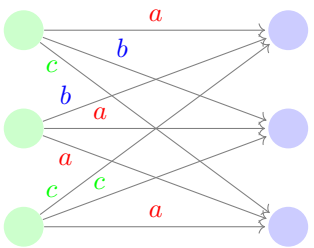



Figure 6.4: Graph representation for a layer employing weight sharing. Here, three different weights $a, b, c$ are shared among the connections from one layer to the next one.

### 6.4.1 ▪ Convolutional layers

A special layer with shared weights is the so-called *convolutional* layer; see, e.g., [GBC16, KSH12, RW17].

**Convolutional neural networks**

***Convolutional neural networks*** (CNNs) are NNs that contain a convolutional layer. The arithmetic operation performed when forward-propagating information to a convolutional layer resembles a mathematical convolution; see Section 4.5.3 for a definition. To be more precise, the incoming weights are shared in such a way that

the net sum at each node has a convolutional structure, i.e.,

$$\text{net}_j^{(l+1)} = \sum_{k=-m}^{m^*} o_{j+k}^{(l)} w_{m+1+k}^{(l)} + b_j^{(l+1)}, \tag{6.3}$$

where $m^* = m$ if we want an odd number of $2m+1$ weights $w_1^{(l)}, \dots, w_{2m+1}^{(l)}$ and $m^* = m-1$ if we want an even number of $2m$ weights $w_1^{(l)}, \dots, w_{2m}^{(l)}$. Here, the size $d_{l+1}$ of the $(l+1)$th layer depends on how we treat indices $i$ for which $o_i^{(l)}$ does not exist, i.e., for which $i \leq 0$ or $i > d_l$.

Note that the bias is often omitted in convolutional layers since it does not influence the result much for large networks. Note also that the indexing in (6.3) differs slightly from the indexing in the definition of convolutions in Section 4.5.3. We choose the specific definition above to be consistent with most of the literature on CNNs.

**Padding and stride** Besides the definition in (6.3), two additional parameters determine the results and layer size of a convolutional layer in practice: the *padding* parameter $\tilde{p} \in \mathbb{N}$ and the *stride* parameter $\tilde{s} \in \mathbb{N}$.

The padding determines how non-existing values $o_i^{(l)}$ for $i \leq 0$ and $i > d_l$ in (6.3) are treated. This means that all values $o_i^{(l)}$ for $-\tilde{p} < i \leq 0$ and $d_l < i \leq d_l + p$ are defined to be zero. Computations involving other non-existing values are neglected.

The stride determines how many $o_i^{(l)}$ values are skipped, i.e., how many of the $l$th layer neurons are neglected when computing the net sum for the next neuron in the $(l+1)$th layer. In particular, the formula (6.3) is only valid for $\tilde{s} = 1$, which is the default value. For other strides $\tilde{s}$ the corresponding formula becomes more complicated and reads

$$\text{net}_j^{(l+1)} = \sum_{k=-m}^{m^*} o_{\tilde{s}(j-1)+k+1}^{(l)} w_{m+1+k}^{(l)} + b_j^{(l+1)}, \tag{6.4}$$

where $m^* = m$ for an odd number of weights and $m^* = m-1$ for an even number of weights. A simple example of a graph for a convolutional layer with three weights and with $\tilde{p} = \tilde{s} = m = 1$ is depicted in Figure 6.5.

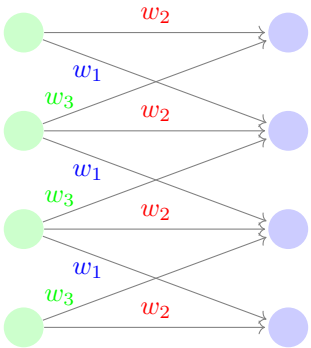



Figure 6.5: Graph representation for a convolutional layer. The shared weights $w_1, w_2, w_3$ are used to compute (6.3) for $m^* = m = 1$.

**Layer size** The layer size $d_{l+1}$ directly depends on the size of $\tilde{p}$ and $\tilde{s}$, i.e.,

$$d_{l+1} = \left\lfloor \frac{d_l + 2\,(\tilde{p} - m) - 1 + \tilde{s}}{\tilde{s}} \right\rfloor \tag{6.5}$$

for the case of $2m + 1$ weights and

$$d_{l+1} = \left\lfloor \frac{d_l + 2\,(\tilde{p} - m) + \tilde{s}}{\tilde{s}} \right\rfloor \tag{6.6}$$

for the case of $2m$ weights. In the following we will assume $\tilde{s} = 1$ unless specified otherwise.

**Parallel layers** Convolutional layers are often used in a *parallel* fashion, i.e., $P \in \mathbb{N}$ independent copies (so-called *channels*) of a convolutional layer are learned with different weights for each copy. The concatenated output of all these parallel layers serves as the input to the subsequent layer in the network. Another way to look at this is to think of a layer of size $d_{l+1}$ where every neuron stores a $P$-dimensional vector of convolutions, e.g.,

$$\mathrm{net}_j^{(l+1)} = \left( \sum_{k=-m}^{m} o_{j+k}^{(l)} w_{m+1+k}^{(l,p)} + b_j^{(l,p)} \right)_{p=1}^{P}$$

for $\tilde{s} = 1$ and an odd number of weights. We can think of this procedure as creating $P$ different feature maps for the data. Then, by learning the weights and biases of the subsequent layers, the most significant combination of those feature maps is chosen by the network.

Finally, let us consider the case of a parallel convolutional layer with $P_o$ (output) channels whose input already consists of $P_i$ (input) channels. In this case, for each output channel, a convolutional operator is applied to all $P_i$ input channels, but with different (learnable) weights per input channel. The results are then summed up over all input channels. In this way, each of the $P_o$ output channels contains information about each of the $P_i$ input channels.

### 6.4.2 ▪ Pooling

To reduce the size of a network further and since some information gained in an *overlapping*-type of convolutional layer, i.e., $\tilde{s} < 2m$, is redundant, a so-called *pooling* layer is often employed after a convolutional layer. Here, a simple mathematical operation is used to summarize/condense the information of several neurons into one neuron. The most common type of pooling is *max-pooling*, where the maximum of the values of the incoming neurons is stored. Let us consider the example graph for a $\max$-pooling layer with stride $\tilde{s} = 3$ in Figure 6.6. We can interpret this as a maximum operator with three arguments (*size* $= 3$), which is applied to the first layer successively and jumps three neurons ahead after each application (stride $\tilde{s} = 3$).

### 6.4.3 ▪ Multi-dimensional convolutional and pooling layer

Two-dimensional convolutional layers have been especially successful for image processing. Instead of just rearranging the pixels of an image in a 1D-type input layer and then applying a convolution as in (6.3), we keep the matrix-like structure of the image and apply a 2D convolutional layer with a local *2D stencil*. Note that, if the image has more than one color channel, it is interpreted as a *tensor*, i.e., for each color channel, the information is stored in a matrix. The same principle holds if we apply parallel 2D convolutional layers. They operate in the same fashion as the 1D parallel layers described in Section 6.4.1. For ease of notation, let us consider 2D convolutional layers with one input channel and one output channel in the following formulas. In

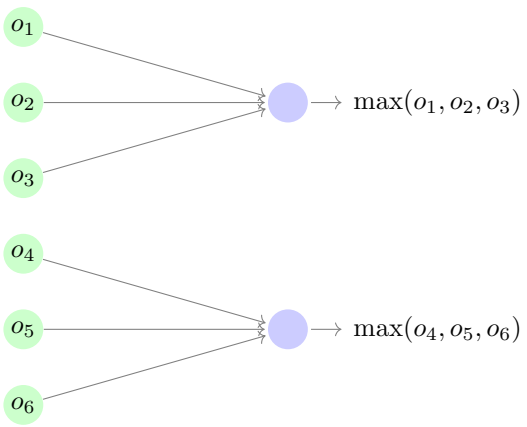



Figure 6.6: Graph representation for a $\max$-pooling layer of size 3 with a stride of 3.

particular, let $o_{i,j}^{(l)}$ denote the $(i,j)$th entry of the image/matrix at layer $l$. Then, a 2D convolution with a $2m_1 \times 2m_2$ weight stencil

$$\tilde{\boldsymbol{W}}^{(l)} := \begin{pmatrix} w_{1,1}^{(l)} & w_{1,2}^{(l)} & \dots & w_{1,2m_2}^{(l)} \\ w_{2,1}^{(l)} & w_{2,2}^{(l)} & \dots & w_{2,2m_2}^{(l)} \\ \vdots & \vdots & \vdots & \vdots \\ w_{2m_1,1}^{(l)} & w_{2m_1,2}^{(l)} & \dots & w_{2m_1,2m_2}^{(l)} \end{pmatrix},$$

i.e., with an even number of weights in both directions, is given by

$$\text{net}_{i,j}^{(l+1)} = \sum_{k_1=-m_1}^{m_1-1} \sum_{k_2=-m_2}^{m_2-1} o_{i+k_1,j+k_2}^{(l)} w_{m_1+1+k_1,m_2+1+k_2}^{(l)} + b_{i,j}^{(l+1)}, \tag{6.7}$$

where we assumed that $m_1$ and $m_2$ are both odd. In the case of odd stencil sizes we have to modify the formula accordingly; see (6.3) for the 1D case. A schematic illustration of the application of a 2D convolutional stencil can be found in Figure 6.7.

Analogously, a ($\max$-)pooling layer can also be defined in the same 2D fashion, as is shown in Figure 6.8.

A typical 2D-convolutional NN for image classification then consists of a sequence of alternating 2D convolutional layers (possibly $P$ parallel ones) and 2D pooling layers. At the end, a fully connected layer is usually added. An illustration can be found in Figure 6.9.

### 6.4.4 ▪ Softmax activation

When dealing with a multi-class problem ($M$ classes), we have to alter the structure of the output layer in order to account for the multiple possible outputs. Usually, a so-called *softmax*-activation function is used to model the output as a probability distribution, i.e.,

$$o_i^{(L+1)} := \frac{\exp\left(\text{net}_i^{(L+1)}\right)}{\sum_{j=1}^{M} \exp\left(\text{net}_j^{(L+1)}\right)} \qquad \forall\, i = 1, \dots, M.$$

Then, we have $M$ output neurons, whose values represent the probability that the input vector belongs to the corresponding class. As the softmax function depends on net-inputs that (in a strict

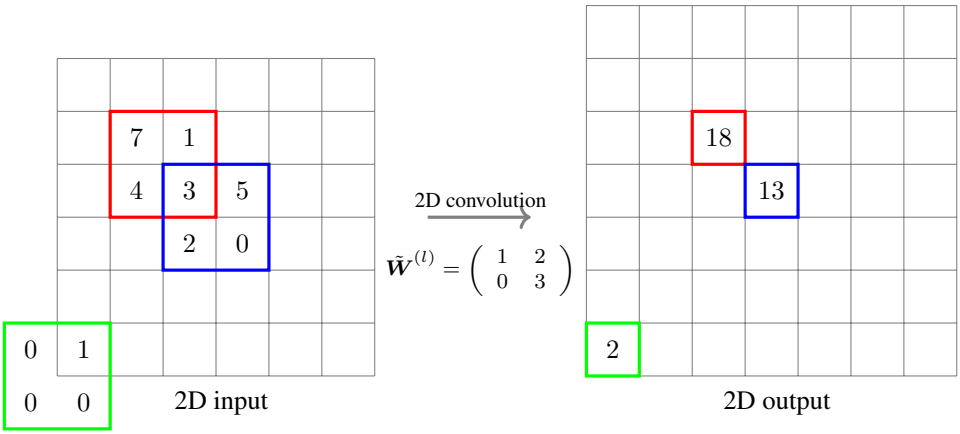



Figure 6.7: Schematic illustration for a 2D convolutional layer with $2 \times 2$ stencil $\tilde{\boldsymbol{W}}^{(l)}$ and $\tilde{s} = \tilde{p} = 1$ in each direction. The input is of size $6 \times 6$, and we exemplarily depict the entries within the red, blue, and green squares. Note that the padding at the boundary leads to the zeros within the green square. We illustrate the resulting values after the convolution operator is applied within the square of the same color in the next layer, e.g., $1 \cdot 7 + 2 \cdot 1 + 0 \cdot 4 + 3 \cdot 3 = 18$ in the red square in the output. Note that the subsequent layer size increased by one in each direction according to (6.6).

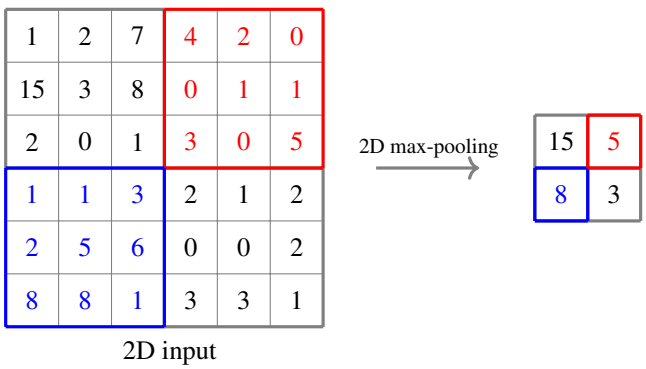



Figure 6.8: Illustration of a 2D max-pooling layer with size $3 \times 3$ and stride $\tilde{s} = 3$ in each direction.

sense) do not belong to the neuron at hand, the application of the softmax function is sometimes also modeled as an extra layer but without any weights/degrees of freedom.

### 6.4.5 ▪ Cross entropy loss

For multi-class classification with classes $\Gamma$ and a softmax layer at the end of the NN, the least squares loss does not make much sense anymore. Instead, a commonly used loss function to compare two probability distributions $p_1$ and $p_2$ is the *cross entropy*

$$H(p_1, p_2) := \mathbb{E}_{p_1}\left[-\log(p_2)\right] = -\sum_{c \in \Gamma} p_1(c) \log(p_2(c)).$$

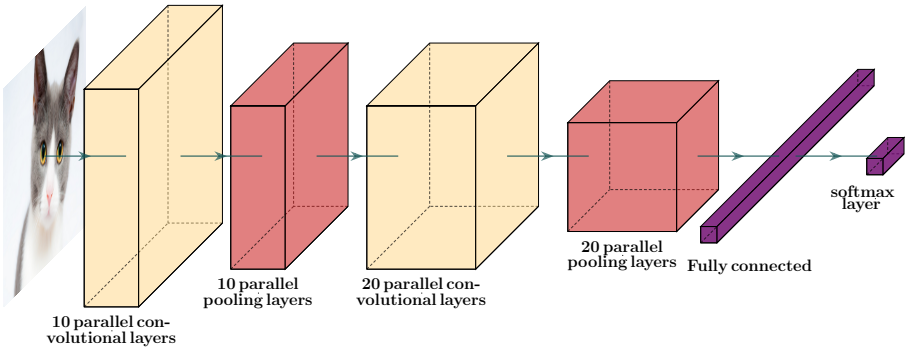



Figure 6.9: Typical 2D CNN for image classification with several parallel convolutional and pooling layers and a final fully connected layer, where the 2D data are stacked into a large 1D vector before applying a softmax activation function. The number of channels (10 and 20) of the convolutional layers has been chosen arbitrarily here. (Cat image licensed under Pixabay Content License.)

In our case, $p_1$ will be the distribution the data are drawn from, i.e., $p_1 = \mu$, and $p_2$ is the output of our neural network after the application of the softmax layer, i.e., $p_2 = f$. The motivation for employing a cross entropy loss is that its minimization is equivalent to the maximization of the so-called *likelihood* of the labels given the data and all weights $\boldsymbol{W}$ and biases $\vec{b}$ under the assumption that the data $(\boldsymbol{x}_i, y_i)$ are independent for different $i \in \{1, \dots, n\}$, i.e.,

$$\begin{aligned}\arg\max_{\boldsymbol{W},\vec{b}} \prod_{i=1}^{n} p_2\left(y_i \mid \boldsymbol{x}_i, \boldsymbol{W}, \vec{b}\right) &= \arg\max_{\boldsymbol{W},\vec{b}} \log\left(\prod_{i=1}^{n} p_2\left(y_i \mid \boldsymbol{x}_i, \boldsymbol{W}, \vec{b}\right)\right) \\ &= \arg\max_{\boldsymbol{W},\vec{b}} \sum_{i=1}^{n} \log\left(p_2\left(y_i \mid \boldsymbol{x}_i, \boldsymbol{W}, \vec{b}\right)\right) \\ &= \arg\max_{\boldsymbol{W},\vec{b}} \mathbb{E}_{p_1}\left[\log(p_2)\right].\end{aligned}$$

This holds since $p_2$ is non-negative and the logarithm is a monotonically increasing function, which does not change the $\arg\max$. Note that

$$\arg\max_{\boldsymbol{W},\vec{b}} \mathbb{E}_{p_1}\left[\log(p_2)\right] = \arg\min_{\boldsymbol{W},\vec{b}} \mathbb{E}_{p_1}\left[-\log(p_2)\right],$$

which shows the equivalence of minimizing the cross entropy between $p_1$ and $p_2$ and maximizing the likelihood of $p_2$ when the data are drawn according to $p_1$.

## 6.5 ▪ A closer look at different optimizers

While SGD is the most commonly used optimizer in deep learning, it might encounter problems due to its fixed learning rate $\nu$ and due to occasional saddle points, i.e., points where the zero-gradient is not sufficient to indicate an optimum. To remedy these issues, many variants and modifications of SGD have been proposed; see [BCN18, GHR20] for a survey.

### 6.5.1 ▪ Stochastic gradient descent

Let us first consider stochastic gradient descent in its most simple form, i.e., with minibatch size 1, see Algorithm 10 with $\kappa = 1$. Here it is known that certain properties of the minimization

problem at hand allow us to derive an error estimate. To this end, let $\tilde{C}$ be a random variable depending on a data point $(\boldsymbol{x}, y)$ drawn according to the data measure $\mu$. Then, let $C_i$ be instances of $\tilde{C}$ instantiated at the training data points $(\boldsymbol{x}_i, y_i)$ for $i = 1, \dots, n$. For example, $C_i(f) = (f(\boldsymbol{x}_i) - y_i)^2$ as in (6.2). This leads to the least squares loss $C(f) = \frac{1}{n}\sum_{i=1}^{n} C_i(f)$. Now we slightly reformulate the minimization problem such that the loss function does not depend on $f$ but rather on the parameters of $f$. To obtain a general formulation we denote the parameter vector by $\boldsymbol{p} \in \mathbb{R}^{d_p}$. Then, we arrive at the model problem

$$\min_{\boldsymbol{p}\in\mathbb{R}^{d_p}} \Theta(\boldsymbol{p}) := \min_{\boldsymbol{p}\in\mathbb{R}^{d_p}} \frac{1}{n}\sum_{i=1}^{n} \Theta_i(\boldsymbol{p}), \tag{6.8}$$

where $\Theta_i := C_i(f) = C_i \circ f$ and the function $f$ is fully defined by the parameters $\boldsymbol{p}$. In our case, $\boldsymbol{p}$ contains the weights $\boldsymbol{W}$ and biases $\vec{b}$ of a specific neural network class. The following theorem can be found in [GHR20]. To avoid any confusion about which derivative is considered, we will write $\nabla_{\boldsymbol{p}}$ for the derivative with respect to the parameter variable $\boldsymbol{p}$ in the following.

**Theorem 6.5.1** (Gorbunov, Hanzely, Richtárik 2020)**.** *Let the $\Theta_i$ from* (6.8) *be convex and let*

$$\|\nabla_{\boldsymbol{p}}\Theta_i(\bar{\boldsymbol{p}}) - \nabla_{\boldsymbol{p}}\Theta_i(\hat{\boldsymbol{p}})\| \leq L\|\bar{\boldsymbol{p}} - \hat{\boldsymbol{p}}\|$$

*hold for some $L \geq 0$ and for all $\bar{\boldsymbol{p}}, \hat{\boldsymbol{p}} \in \mathbb{R}^{d_p}$ and all $i = 1, \dots, n$ (regardless of the data point defining $C_i$). Furthermore, let $\Theta$ be $\xi$-strongly quasi-convex, i.e.,*

$$\Theta(\boldsymbol{p}^*) \geq \Theta(\boldsymbol{p}) + \langle \nabla_{\boldsymbol{p}}\Theta(\boldsymbol{p}), \boldsymbol{p}^* - \boldsymbol{p}\rangle + \frac{\xi}{2}\|\boldsymbol{p}^* - \boldsymbol{p}\|^2 \quad \forall\, \boldsymbol{p} \in \mathbb{R}^{d_p}$$

*for some $\xi > 0$, where $\boldsymbol{p}^* = \arg\min_{\boldsymbol{p}\in\mathbb{R}^{d_p}} \Theta(\boldsymbol{p})$. Then, choosing $\kappa = 1$ and a learning rate $0 < \nu \leq \frac{1}{2L}$ in Algorithm 10 leads to the upper bound*

$$\mathbb{E}\left[\|\boldsymbol{p}^* - \boldsymbol{p}_k\|^2\right] \leq (1 - \nu\xi)^k \|\boldsymbol{p}^* - \boldsymbol{p}_0\|^2 + \frac{2\nu\sigma^2}{\xi} \tag{6.9}$$

*for the difference between the true solution $\boldsymbol{p}^*$ and the parameter value $\boldsymbol{p}_k$ after the $k$th iteration, i.e., after the $k$th minibatch has been processed.*[27] *Here, $\boldsymbol{p}_0$ is the initial value and $\sigma^2 := \mathbb{E}\left[\|\nabla_{\boldsymbol{p}}\Theta_1(\boldsymbol{p}^*)\|^2\right]$.*

This theorem gives us an upper bound on the error after $k$ iterations of the SGD algorithm. Note that the upper bound only converges to 0 if $\sigma^2 = \mathbb{E}\left[\|\nabla_{\boldsymbol{p}}\Theta_i(\boldsymbol{p}^*)\|^2\right] = 0$ would hold for some/any $i$ (since they are identically distributed). Nevertheless, when choosing a very small learning rate $\nu$, the second term on the right-hand side of (6.9) almost vanishes. However, the first term decays then very slowly with increasing $k$. Therefore, a certain tradeoff is needed to obtain a fast convergence and a small overall upper error bound.

Many versions of the above theorem for different SGD variants can also be found in [GHR20], e.g., variants for a minibatch algorithm or a regularized version of it. A similar error bound for SGD with slightly different prerequisites and a result for a decaying learning rate can be found in [BCN18].

When applying the above theorem to neural networks we immediately encounter a problem: The prerequisites are usually not fulfilled. In particular, the convexity of the $\Theta_i$ and the $\xi$-strong quasi-convexity of $\Theta$ are only met for very special types of multi-layer networks. Therefore, a

[27] Note that the number of iterations $k$ refers to the number of times that line 6 in Algorithm 10 has been called. This means that we already performed $n$ iterations after one full epoch if the minibatch size is $\kappa = 1$ since we execute line 6 of Algorithm 10 once per epoch for each data point.

sound convergence theory in more general cases still needs to be established. Nonetheless, SGD is still frequently applied there. In the next sections we will encounter variants of SGD, which heuristically perform in some cases even better than the theory predicts. To this end, let us briefly introduce a compact way to write the $k$th iteration of the SGD algorithm as

$$\boldsymbol{p}_k = \boldsymbol{p}_{k-1} - \nu \nabla_{\boldsymbol{p}} \Theta_B(\boldsymbol{p}_{k-1}),$$

where $\Theta_B(\boldsymbol{p}_{k-1}) = C_B(f(\cdot; \boldsymbol{p}_{k-1}))$ is the minibatch variant of the notation from (6.8); see also Section 6.3. Here, we explicitly expressed the network function $f$ in dependence of both the input and the parameter vector $\boldsymbol{p}$, which we omitted previously for readability reasons.

### 6.5.2 ▪ Momentum based optimizers

The following *momentum-based* variants of SGD are quite popular in the ML community. Similar to plain SGD, there exist error bounds on them in the case of convex minimization; see [Nes83].

- **Momentum update [RHW86]**:

$$\begin{aligned} \boldsymbol{m}_k &:= \chi \cdot \boldsymbol{m}_{k-1} - \nu \cdot \nabla_{\boldsymbol{p}} \Theta_B(\boldsymbol{p}_{k-1}), \\ \boldsymbol{p}_k &= \boldsymbol{p}_{k-1} + \boldsymbol{m}_k. \end{aligned}$$

  Here, a momentum term $\boldsymbol{m}_k$ is incorporated, which contains the history of already encountered gradient steps. It is weighted by $0 < \chi < 1$ in each iteration. Therefore, the contribution of values of older gradients decays exponentially with the number of steps taken. Note that if the most recent gradients have a common direction in which they point, the contribution of this direction to the next step is largely increased. This can be interpreted as pushing a ball down a hill. Typically, $\chi \approx 0.9$ is chosen. An illustration of gradient descent with and without momentum updates can be found in Figure 6.10.

- **Nesterov update [Nes83]**:

$$\begin{aligned} \boldsymbol{m}_k &= \chi \cdot \boldsymbol{m}_{k-1} - \nu \cdot \nabla_{\boldsymbol{p}} \Theta_B(\boldsymbol{p}_{k-1} + \chi \cdot \boldsymbol{m}_{k-1}), \\ \boldsymbol{p}_k &= \boldsymbol{p}_{k-1} + \boldsymbol{m}_k. \end{aligned}$$

  A problem with the momentum update can be that we still follow the direction of older gradients while the gradient at our current iterate might point in a completely different direction. To remedy this issue, Nesterov came up with a clever idea: We first take $\boldsymbol{p}_{k-1} + \chi \cdot \boldsymbol{m}_{k-1}$ as an estimate of where we are going in the next step and then compute the gradient there. This approach has the chance to reach a convergence rate that is twice as good as that of plain SGD, at least for strongly quasi-convex problems; see [AR20].

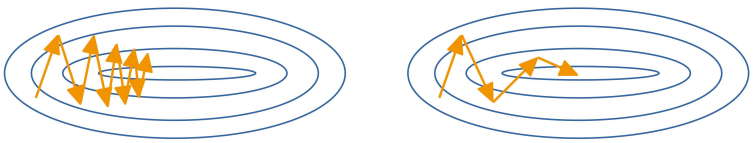


Figure 6.10: Gradient descent updates without (left) and with (right) momentum updates are depicted with orange arrows. The sought minimum is found in the middle of the blue ellipses, which reflect the isolines of the function under consideration.

### 6.5.3 ▪ Step size adaptation

Another improvement to SGD with fixed learning rate $\nu > 0$ is to adaptively adjust $\nu$ according to the currently encountered gradient instead and to even choose a separate learning rate for each entry of the gradient vector. To this end, let $\varepsilon > 0$ be a small threshold.

- **AdaGrad [DHS11]**:

$$\begin{aligned} \boldsymbol{h}_k &= \boldsymbol{h}_{k-1} + \nabla_{\boldsymbol{p}}\Theta_B(\boldsymbol{p}_{k-1}) \odot \nabla_{\boldsymbol{p}}\Theta_B(\boldsymbol{p}_{k-1}), \\ \boldsymbol{p}_k &= \boldsymbol{p}_{k-1} - \frac{\nu \nabla_{\boldsymbol{p}}\Theta_B(\boldsymbol{p}_{k-1})}{\underbrace{\sqrt{\boldsymbol{h}_k}}_{\text{elementwise}} + \varepsilon}. \end{aligned}$$

  The idea of AdaGrad is to now use the squared sum $\boldsymbol{h}_k$ of all previously encountered gradients to change the learning rate individually for each weight and bias. Here, the learning rate is damped for weights/biases that already changed largely in the past and it is increased for weights/biases that only underwent minor changes. Note that the division and the square-root have to be understood elementwise. Usually, a value of $\varepsilon \approx 10^{-4}$ is chosen as threshold to ensure that the denominator in the above equation does not vanish.

- **RMSProp [Hinton - unpublished]**:

$$\begin{aligned} \boldsymbol{h}_k &= \chi \cdot \boldsymbol{h}_{k-1} + (1-\chi)\nabla_{\boldsymbol{p}}\Theta_B(\boldsymbol{p}_{k-1}) \odot \nabla_{\boldsymbol{p}}\Theta_B(\boldsymbol{p}_{k-1}), \\ \boldsymbol{p}_k &= \boldsymbol{p}_{k-1} - \frac{\chi \nabla_{\boldsymbol{p}}\Theta_B(\boldsymbol{p}_{k-1})}{\sqrt{\boldsymbol{h}_k} + \varepsilon}. \end{aligned}$$

  A major problem with AdaGrad is that the whole history of squared gradient information is accumulated in $\boldsymbol{h}_k$ and leads to smaller and smaller updates with increasing iteration numbers. Therefore, the iteration process might be stuck without reaching an adequate minimum. For RMSProp, a moving average of squared gradients is chosen to remedy this problem. Usually, the convex combination coefficient is set to $\chi \approx 0.9$ in practice.

### 6.5.4 ▪ Combining momentum and adaptive step sizes

Considering the advantages of the SGD variants using either momentum based optimization or adaptive step sizes, it is only logical to combine the two strategies. The most famous optimizer that follows both strategies is *Adam*.

- **Adam (Adaptive moment estimation) [KB14]**: At step $k \in \mathbb{N}$, it reads as

$$\begin{aligned} \boldsymbol{m}_k &= \beta_1 \cdot \boldsymbol{m}_{k-1} + (1-\beta_1)\nabla_{\boldsymbol{p}}\Theta_B(\boldsymbol{p}_{k-1}), \\ \boldsymbol{h}_k &= \beta_2 \cdot \boldsymbol{h}_{k-1} + (1-\beta_2)\nabla_{\boldsymbol{p}}\Theta_B(\boldsymbol{p}_{k-1}) \odot \nabla_{\boldsymbol{p}}\Theta_B(\boldsymbol{p}_{k-1}), \\ \hat{\boldsymbol{m}}_k &= \frac{\boldsymbol{m}_k}{1-\beta_1^k}, \\ \hat{\boldsymbol{h}}_k &= \frac{\boldsymbol{h}_k}{1-\beta_2^k}, \\ \boldsymbol{p}_k &= \boldsymbol{p}_{k-1} - \frac{\nu \cdot \hat{\boldsymbol{m}}_k}{\sqrt{\hat{\boldsymbol{h}}_k} + \varepsilon}. \end{aligned}$$

  Here, the two ideas of momentum update and adaptive learning rate are combined. To this end, moving averages $\boldsymbol{m}_k$ of the gradients are used as a momentum term and moving averages $\boldsymbol{h}_k$ of the squared gradients are used to adaptively guide the step size. Furthermore,

a correction of $\frac{1}{1-\beta_1^k}$ is made to $\boldsymbol{m}_k$ and a correction of $\frac{1}{1-\beta_2^k}$ is made to $\boldsymbol{h}_k$. These corrections account for a potential initialization bias for large values of $\beta_1$ and $\beta_2$. Note that there also exists a Nesterov-version of Adam called NAdam [Doz16].

In practice, plain SGD and Adam are most frequently used, and we will encounter them in Section 6.6 in detail.

## 6.6 ▪ Tasks on deep neural networks

### 6.6.1 ▪ Two-layer neural network

We will start by implementing a two-layer fully connected neural network on our own to grasp the essentials of the architecture and the forward and backward propagation steps.

**Task 6.1.** Implement a class `TwoLayerNN`, which represents a (fully connected, feed-forward) two-layer neural network, i.e., $L = 2$. The activation functions should be $\phi^{(2)} = \mathrm{ReLU}$ and $\phi^{(3)} = \mathrm{id}$. The weights and biases can be initialized by drawing i.i.d. uniformly distributed random numbers in $(-1, 1)$. The class should contain a method `feedForward` to calculate the point evaluations of $f$ for a whole minibatch at once and a method `backprop` to calculate $\nabla_{\boldsymbol{W},\vec{b}} C_B(f)$. To this end, avoid using for-loops over the minibatch and use linear algebra operations (on vectors, matrices, or tensors) from NumPy instead.

**Task 6.2.** Augment the `TwoLayerNN` class by implementing a method to randomly draw a minibatch data set and a routine to perform the stochastic minibatch gradient descent algorithm.

**Task 6.3.** Test your implementation by drawing 250 uniformly distributed points $\boldsymbol{x}_i$ in $\mathbb{R}^2$ with norm $\|\boldsymbol{x}_i\| \leq 1$ and label them by $y_i = -1$. Now draw 250 uniformly distributed points $\boldsymbol{x}_i$ in $\mathbb{R}^2$ with $1 < \|\boldsymbol{x}_i\| \leq 2$ and label them by $y_i = 1$. Use your two-layer neural network with $d_2 = 20$ hidden layer neurons, $\kappa = 20$ and 50000 iterations, i.e., use $S = \frac{50000}{\kappa} = 2500$ epochs, to classify the data. Try different learning rates $\nu$. Output the least squares error every 5000 iterations. After the result has been computed, make a scatter plot of the data and draw the contour line of your learned classifier. What do you observe? What happens if you increase $S$?

### 6.6.2 ▪ Keras and TensorFlow

For the final tasks we use the Keras library [Cea15]. It provides a high-level and easy to use abstraction for the popular deep learning backend TensorFlow [Aea15].

Defining the network from Task 6.3 could be done as follows:

```
from tensorflow import keras
import tensorflow.keras.layers as layers

model = keras.models.Sequential()
model.add(layers.Dense(20, input_shape=(2,), activation='relu'))
model.add(layers.Dense(1))
```

You can print a summary with `print(model.summary())`.

Continue with compiling the model

```
model.compile(loss='mse', optimizer='sgd', metrics=['accuracy'])
```

Here, `mse` stands for mean squared error (cf. (6.2)), and `sgd` is an abbreviation for stochastic gradient descent; see Algorithm 10. You can start training the model by

```
k = 20
S = 50000
history = model.fit(X_train, Y_train, batch_size=k,
                    epochs=S / (500 / k), verbose=True)
```

The gradient implementation[28] is derived using automatic differentiation. KERAS has support for several gradient descent variants, which can also be configured (e.g., it allows us to change the step size). Of course, the final layer can have an activation function too, e.g., `softmax` when used for classification.

A convolutional layer with a ReLU activation can be added with

```
model.add(layers.Conv2D(16, kernel_size=(3, 3), activation='relu'))
```

which adds 16 parallel layers (i.e., 16 layers of the same shape at the same position[29]), each with a $3 \times 3$ convolutional matrix for 2D input. To flatten the result for classification use

```
model.add(layers.Flatten())
```

### 6.6.3 ▪ Regularization

To prevent overfitting many techniques are known, but most of them are not well understood theoretically. A very popular technique is the dropout approach (compare Section 6.4), where, during each training step, one neglects nodes in a layer with a given probability $p$. In this way, random sub-nets are trained. To add dropout regularization to a layer employ

```
model.add(layers.Dropout(p))
```

after it.

KERAS also has support for using regularization terms in the loss function, similar to what we have seen in the discussion of SVMs in Chapter 3.

**Task 6.4.** Use KERAS to build a classifier for the MNIST data set (see the template JUPYTER notebook).

(a) Build a model with the following layers:

- a fully connected layer (`Dense`) with 128 output nodes + ReLU,
- a fully connected layer with 128 output nodes + ReLU,
- a fully connected layer with 10 output nodes + softmax.

Use the SGD optimizer with a batch size of 128 and the categorical cross entropy loss (`loss="categorical_crossentropy"`) and train for 20 epochs. Use the accuracy metric (set in `model.compile`) and provide the test data as

[28] The network is at least piecewise differentiable.

[29] This can also be understood in the following way: Each neuron of the convolutional layer contains a 16-dimensional vector.

validation data to `model.fit` (set the parameter `validation_data` to appropriately reshaped versions of `(X_test, Y_test)`). Plot the fit history (return value of `model.fit`).

(b) Build a new network by adding dropout ($p = 0.3$) to the first and second layers of the model from (a). Train it for 250 epochs.

(c) Build a third network by using the model from (b) with the optimizer `"adam"` instead of SGD. What does this optimizer do in contrast to SGD? Train for 20 epochs.

**Task 6.5.** For the MNIST data set, build a CNN with KERAS with the following layers:

- 16 parallel convolutional layers with kernel size $3 \times 3$ + ReLU,
- 32 parallel convolutional layers with kernel size $3 \times 3$ + ReLU,
- a 2D max pooling layer of size $2 \times 2$, non-overlapping + dropout ($p = 0.25$),
- a flattening layer, which converts its input to a vector,
- a fully connected layer with 128 outputs + ReLU + dropout ($p = 0.5$),
- a fully connected layer with output size 10 + softmax.

Train for 15 epochs; the other parameters should be the same as in the previous model from Task 6.4.

Feel free to change the network/training in order to improve the error.

**Task 6.6.** Use a CNN to learn features for pedestrian classification. Proceed as follows:

- Design and train a CNN for pedestrian classification (use the data from Section 4.5.2). Here, you can start with the network from Task 6.5.
- Use the output after the flattening (see KERAS' FAQ on how to do this) as a feature vector for a linear SVM, maybe even together with the HOG features.

Try to use PCA in order to improve the accuracy, also tweak the HOG parameters. Make sure not to overfit (the pedestrian data set is small for deep learning standards). Hint: You can also install the AUGMENTOR library for PYTHON in order to enlarge the data set, which is a common technique in deep learning.

## 6.7 ▪ Further topics

**GPUs/TPUs** KERAS (more precisely its backends) can take advantage of a GPU (graphics processing unit) in order to speed up the training. When executing the code for the tasks above, your training was probably using the CPU, even if your machine has a GPU. Setting up KERAS to run on the GPU instead can be challenging. But if successful, the performance gain is usually significant. We ran Task 6.5 on six Xeon 3.6 GHz (Sandy Bridge) cores and on a Tesla P100 GPU. For the CPU, one epoch of training took 30 seconds (a 2012 Quad-Core laptop took over

a minute), while on the GPU an epoch was finished in three seconds. Even higher speedups are common.

Training a neural network involves a lot of trial-and-error and requires experience and patience. Modern networks can only be trained (in reasonable time) on a GPU or on dedicated hardware. For example, the famous *AlexNet* [KSH12] had 60 million parameters and was trained over six days on two GPUs in 2012. More recent approaches usually achieve smaller training times but exploit a much more expensive hardware setup. The authors of [GDG+17], for instance, trained their network within an hour on 256 GPUs in 2017. However, comparing many recent experiments on the same data set to older results is usually not straightforward since a lot of *pre-training* and model/architecture mixing is being done; see, e.g., [YWV+22].

Besides GPUs, so-called *tensor processing units* (TPUs) have been developed. These are specific hardware/chip architectures, which are even more efficient for deep learning problems than GPUs; see, e.g., [Jea17].

**References on general neural networks and CNNs** The free course at fast.ai[30] is a highly recommended reference when looking for further tutorials on neural networks from a programmer's perspective. A more thorough and complete consideration of many neural network types is given in the book [GBC16] or in the review [RW17] on CNNs. There you will also find references on the history of neural networks.

**Recurrent neural networks** An important type of network, which we only briefly discuss in Section 10.3, is the recurrent neural network; see also [GBC16]. Here, variable input sizes/dimensions and sequence data (such as time series) can be processed. To this end, an internal state variable is stored and recomputed for each new time step, for instance. The most famous recurrent neural networks are long short-term memory networks (LSTM) [HS97] or gated recurrent units [CvMG+14]. They allow us to process long and possibly instationary sequences by adaptively learning which time horizon to choose, while using past data in order to predict future values.

**Information-theoretic consideration of deep learning** Besides the approximation-theoretic results we briefly touched upon in Section 6.2, deep neural networks can also be considered from an information-theoretic viewpoint. The so-called *information bottleneck* method quantifies how much relevant information about the underlying data distribution is retained throughout the network. Subsequently, the network can be optimized such that an optimal tradeoff between kept relevant information and information compression is achieved. For more details we recommend [TZ15] and Chapter 12 from [Cal20].

[30] http://www.fast.ai/

## Chapter 7

# Variational Autoencoders

In this chapter, we will introduce neural networks specifically designed for the unsupervised learning task of dimensionality reduction. These networks are called *autoencoders*.

**Autoencoders**

***Autoencoders*** are special feed-forward neural networks. They consist of so-called ***encoder*** layers, which perform the reduction step from the high-dimensional data space to a certain low-dimensional space, and of ***decoder*** layers, where vectors from the low-dimensional space are mapped back to the high-dimensional space. In this fashion, two parts are trained simultaneously: A dimensionality reduction architecture and a high-dimensional vector reconstruction algorithm. For details, we refer the reader to [Bal12, GBC16, Sch15].

The autoencoder approach allows the user to traverse the low-dimensional space of hidden (or *latent*) variables, in which—in the best case—the directions can be associated with some meaningful key figures such as physical, biological, or sociological quantities.

**Latent variables and latent space**

In unsupervised learning, the term ***latent variables*** or ***hidden variables*** relates to the coordinates of the low-dimensional embedding space. The terminology stems from statistics, where these variables are considered as *unobservable* in contrast to the high-dimensional variables, which are observable via the given samples $\boldsymbol{x}_i$ for $i = 1, \ldots, n$. Therefore, the low-dimensional space itself is often referred to as ***latent space***.

It is oftentimes not a priori clear how to obtain meaningful and interpretable latent variables for the dimensionality reduction step. Furthermore, it is not clear if, for instance, the high-dimensional reconstruction of an interpolated vector between two data points in the low-dimensional space is meaningful at all. To this end, *variational autoencoders* [Doe16, KW13, KW19] have been created. They involve a probabilistic approach that ensures an appropriate data distribution within the latent space and a meaningful process of generating high-dimensional vectors from low-dimensional representatives. This serves as a first step in the direction of interpretable latent variables.

This chapter is structured as follows. Section 7.1 gives a general introduction to autoencoders and their connection to the PCA. Next, we present tasks on autoencoders in Section 7.2. Subsequently, we discuss the concept of latent variables and the necessary statistical background in Section 7.3, before we come to study variational autoencoders in Section 7.4. Finally, we give tasks on the analysis of the MNIST data set from Section 3.7 with variational autoencoders in Section 7.5.

## 7.1 ▪ Autoencoders

First, we introduce the concept of autoencoders represented by neural networks. To this end, we briefly review the idea behind these special architectures and their construction before considering variations thereof.

### 7.1.1 ▪ Motivation and idea

Autoencoders (AEs) were designed for the task of dimensionality reduction or merely compression of high-dimensional input vectors. Note that we already came across prominent and successful methods from this area, namely the linear PCA in Chapter 4 and the nonlinear diffusion maps algorithm in Chapter 5. As we learned there, the goal of dimensionality reduction is to obtain a *more effective* low-dimensional representation of the input vectors. Moreover, as we now have encountered neural networks in Chapter 6, it appears only natural to ask whether there exists a neural network construction for dimensionality reduction. Here, autoencoders represent a type of neural network for exactly this goal.

Essentially, an AE is built from two maps:

1. An encoder $E : \mathbb{R}^d \to \mathbb{R}^q$ mapping from the original data space $\mathbb{R}^d$ to the low-dimensional embedding space $\mathbb{R}^q$ with $q < d$.
2. A decoder $D : \mathbb{R}^q \to \mathbb{R}^d$ mapping in the opposite direction.

Then, the AE $h$ is just the concatenation of both maps, i.e.,

$$h := D \circ E : \mathbb{R}^d \to \mathbb{R}^d.$$

The main question is how to build the encoder and decoder. To this end, it is crucial to look at the motivation behind an AE, i.e., the mathematical task it tries to solve.

Recall that the idea behind Isomap or diffusion maps was to preserve a special type of distance: The Euclidean distance of the embedded vectors approximately matches the geodesic distance or the diffusion distance, respectively, in the original data space after the dimensionality reduction. For the linear PCA we also encountered the MDS interpretation (see Section 4.3), where we saw that pairwise Euclidean inner products between the data points in the low-dimensional embedding space should approximately match the ones of the high-dimensional vectors.

The main strategy behind autoencoders is instead related to the original idea behind the PCA of minimizing the (Euclidean) distance between the original vectors and their reconstructed counterparts; cf. (4.3). For AEs this translates to

$$\operatorname*{arg\,min}_{h=D\circ E} \frac{1}{n} \sum_{i=1}^{n} \|\boldsymbol{x}_i - h(\boldsymbol{x}_i)\|^2 = \operatorname*{arg\,min}_{D,E} \frac{1}{n} \sum_{i=1}^{n} \|\boldsymbol{x}_i - D \circ E(\boldsymbol{x}_i)\|^2. \tag{7.1}$$

Before we hint at possible difficulties and remedies for tackling this minimization problem, let us have a closer look at the remaining question of defining $E$ and $D$ or, equivalently, of choosing appropriate model spaces for the encoder and decoder. For our considerations we will stick to the case of $E$ and $D$ being special kinds of neural networks.

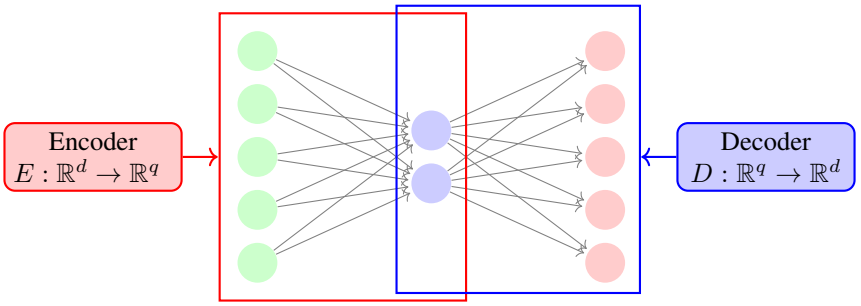



Figure 7.1: An autoencoder with $d = 5$ and $q = 2$.

### 7.1.2 ▪ Construction

In its most simple form, an autoencoder is just a fully connected two-layer (feed-forward) neural network, where the hidden layer contains fewer neurons than the input layer. Furthermore, the output layer has the same size as the input layer. In terms of the dimensions $q$ and $d$, which we assigned to the encoder $E$ and decoder $D$, we thus have $d$ input and output neurons and $q < d$ hidden neurons; see Figure 7.1.

The encoder can now be simply written as

$$E(\boldsymbol{x}) = \phi_E(\boldsymbol{W}_E\boldsymbol{x} + \boldsymbol{b}_E)$$

for a weight matrix $\boldsymbol{W}_E \in \mathbb{R}^{q\times d}$, a bias vector $\boldsymbol{b}_E \in \mathbb{R}^q$, and an activation function $\phi_E : \mathbb{R} \to \mathbb{R}$ that acts elementwise. Analogously, the decoder can be written as

$$D(\boldsymbol{z}) = \phi_D(\boldsymbol{W}_D\boldsymbol{z} + \boldsymbol{b}_D)$$

for a weight matrix $\boldsymbol{W}_D \in \mathbb{R}^{d\times q}$, a bias vector $\boldsymbol{b}_D \in \mathbb{R}^d$, and an activation function $\phi_D : \mathbb{R} \to \mathbb{R}$. In order to learn the weights and biases of $E$ and $D$, we minimize the distance between data points and their reconstructions as given in (7.1) by using one of the optimizers from Section 6.5 together with backpropagation; see Section 6.3. In this way, we learn the encoder and decoder simultaneously.

### 7.1.3 ▪ Comparison to PCA

We already noted above that the minimization criterion behind the learning process for autoencoders is closely related to the one for PCA. However, this is not the only relationship between these two dimensionality reduction methods: If $\phi_E = \phi_D = \mathrm{id}$, there is an inherent similarity to the original PCA minimization problem (4.3), which searches for

$$\operatorname*{arg\,min}_{f\in\mathcal{M}_{\mathrm{Lin}}^{q,d},\,\boldsymbol{\eta}_i\in\mathbb{R}^q} \frac{1}{n}\sum_{i=1}^{n} \|\boldsymbol{x}_i - f(\boldsymbol{\eta}_i)\|^2 = \operatorname*{arg\,min}_{\boldsymbol{W}\in\mathbb{R}^{d\times q},\,\boldsymbol{b}\in\mathbb{R}^d,\,\boldsymbol{\eta}_i\in\mathbb{R}^q} \frac{1}{n}\sum_{i=1}^{n} \|\boldsymbol{x}_i - \boldsymbol{W}\boldsymbol{\eta}_i + \boldsymbol{b}\|^2.$$

Since $\phi_D = \mathrm{id}$, the decoder also fulfills $D \in \mathcal{M}_{\mathrm{Lin}}^{q,d}$ and resembles the map $f$; see also (4.1). Now, in the PCA formulation, we aim to minimize with respect to the low-dimensional data points $\boldsymbol{\eta}_i$. In our autoencoder setting, they just resemble the encoded inputs $E(\boldsymbol{x}_i)$.

To see this resemblance more clearly, let us have a look at the autoencoder minimization problem in this setting and omit the activation functions $\phi_E, \phi_D$:

$$\begin{aligned}(D^*, E^*) &= \underset{(D,E)}{\arg\min} \frac{1}{n} \sum_{i=1}^{n} \|\boldsymbol{x}_i - D \circ E(\boldsymbol{x}_i)\|^2 \\ &= \underset{\boldsymbol{W}_E, \boldsymbol{W}_D, \boldsymbol{b}_E, \boldsymbol{b}_D}{\arg\min} \frac{1}{n} \sum_{i=1}^{n} \|\boldsymbol{x}_i - \boldsymbol{W}_D \left(\boldsymbol{W}_E \boldsymbol{x}_i + \boldsymbol{b}_E\right) - \boldsymbol{b}_D\|^2 .\end{aligned}$$

Note that the solution tuple $(D^*, E^*)$ is not unique since arbitrary basis changes in the latent space do not influence $D \circ E$. Now recall that we found in Section 4.1 that the PCA solution for the $\boldsymbol{\eta}_i$ is

$$\boldsymbol{\eta}_i = \boldsymbol{W}^T (\boldsymbol{x}_i - \boldsymbol{b}).$$

This led to

$$\boldsymbol{W} = \underset{\boldsymbol{A} \in S_q(\mathbb{R}^d)}{\arg\min} \frac{1}{n} \sum_{i=1}^{n} \left\| \left(\boldsymbol{I} - \boldsymbol{A}\boldsymbol{A}^T\right) (\boldsymbol{x}_i - \boldsymbol{b}) \right\|^2 .$$

Thus, if we set the encoder bias to $\boldsymbol{b}_E = -\boldsymbol{W}_E \boldsymbol{b}_D$, we obtain a very similar altered autoencoder problem

$$(D^*, E^*) = \underset{\boldsymbol{W}_E, \boldsymbol{W}_D, \boldsymbol{b}_D}{\arg\min} \frac{1}{n} \sum_{i=1}^{n} \|(\boldsymbol{I} - \boldsymbol{W}_D \boldsymbol{W}_E) (\boldsymbol{x}_i + \boldsymbol{b}_D)\|^2 .$$

Essentially, the only difference from PCA is that the encoder matrix $\boldsymbol{W}_E$ is not necessarily the transpose of the decoder matrix $\boldsymbol{W}_D$ for the autoencoder. Note furthermore that $\boldsymbol{I} - \boldsymbol{A}\boldsymbol{A}^T$ is an orthogonal projection onto the span of the columns of $\boldsymbol{A}$. However, for the autoencoder, $\boldsymbol{I} - \boldsymbol{W}_D \boldsymbol{W}_E$ is not necessarily an orthogonal projection onto the subspace defined by the columns of $\boldsymbol{W}_E^T$.

Omitting the biases for a moment and looking at the autoencoder from a matrix factorization point of view, we consider the decomposition of the matrix $\boldsymbol{X} := (\boldsymbol{x}_1 \dots \boldsymbol{x}_n)^T \in \mathbb{R}^{n \times d}$ into $\boldsymbol{X} \approx \boldsymbol{L}\boldsymbol{W}_D^T$ with a matrix of $n$ latent vectors $\boldsymbol{\eta}_i \in \mathbb{R}^q$, namely $\boldsymbol{L} \in \mathbb{R}^{n \times q}$. To this end, we aim to solve the problem

$$\underset{\boldsymbol{L}, \boldsymbol{W}_D}{\arg\min} \left\| \boldsymbol{X} - \boldsymbol{L}\boldsymbol{W}_D^T \right\|_F^2 := \underset{\boldsymbol{\eta}_i, \boldsymbol{W}_D}{\arg\min} \sum_{i=1}^{n} \|\boldsymbol{x}_i - \boldsymbol{W}_D \boldsymbol{\eta}_i\|^2.$$

If we assume that $\boldsymbol{W}_D$ has orthonormal columns, this problem is exactly solved by the PCA. For this case, let us write $\boldsymbol{W}_D$ as

$$\boldsymbol{W}_D = \boldsymbol{V}_q,$$

where $\boldsymbol{X} = \boldsymbol{U}\boldsymbol{S}\boldsymbol{V}^T$ is the SVD of $\boldsymbol{X}$ (see Section 4.2), and $\boldsymbol{V}_q$ contains the first $q$ columns of $\boldsymbol{V}$ (see also Section 4.2). Now we can write the transformed data $\boldsymbol{L}$ as

$$\boldsymbol{L} = \boldsymbol{X}\boldsymbol{V}_q^T = \boldsymbol{U}\boldsymbol{S}\boldsymbol{V}^T\boldsymbol{V}_q = \boldsymbol{U}\boldsymbol{S}\boldsymbol{I}_q = \boldsymbol{U}_q\boldsymbol{S}_q,$$

where $\boldsymbol{S}_q$ denotes the upper left $q \times q$ submatrix of $\boldsymbol{S}$.

Next, returning to the autoencoder, let us have a look at the result of feed-forwarding the input vectors $\boldsymbol{x}_i$. We then compute

$$\tilde{\boldsymbol{X}} := \boldsymbol{X}\boldsymbol{W}_E^T\boldsymbol{W}_D^T \in \mathbb{R}^{n \times d}.$$

If we look at our autoencoder loss, we are actually minimizing

$$\underset{\boldsymbol{W}_E,\boldsymbol{W}_D}{\arg\min} \left\| \tilde{\boldsymbol{X}} - \boldsymbol{X} \right\|_F^2 = \underset{\boldsymbol{W}_E,\boldsymbol{W}_D}{\arg\min} \left\| \boldsymbol{X}\boldsymbol{W}_E^T\boldsymbol{W}_D^T - \boldsymbol{X} \right\|_F^2 .$$

From linear algebra it is known that the minimizer $\boldsymbol{W}_E$ is the pseudo-inverse $\boldsymbol{W}_D^\dagger$ of $\boldsymbol{W}_D$ given by

$$\boldsymbol{W}_E = \boldsymbol{W}_D^\dagger := \left(\boldsymbol{W}_D^T\boldsymbol{W}_D\right)^{-1}\boldsymbol{W}_D^T$$

in the case that $\boldsymbol{W}_D$ has linear independent columns; see also Section 2.1. In the special case of the PCA this results in

$$\boldsymbol{W}_E = \left(\boldsymbol{V}_q^T\boldsymbol{V}_q\right)^{-1}\boldsymbol{V}_q^T = \boldsymbol{V}_q^T .$$

Therefore, the encoder is the transposed version of the decoder in this case. This can be seen as a special kind of weight sharing between the encoder and decoder in the neural network.

Note that we assumed at the very beginning that $\boldsymbol{W}_D$ has orthonormal columns to obtain the special case $\boldsymbol{W}_D = \boldsymbol{V}_q = \boldsymbol{W}_E^T$. However, any decoder matrix whose columns span the same subspace as the columns of $\boldsymbol{V}_q$ also results in an optimal recovery in the Frobenius norm sense. Therefore, there exist infinitely many autoencoder solutions with non-orthogonal columns of $\boldsymbol{W}_D$ which will result in the same low-dimensional space as the PCA does, but the low-dimensional vectors $\boldsymbol{\eta}_i$ will now be spanned by a non-orthogonal basis.

In general, the real strength of autoencoders—compared to PCA—is of course given by the additional nonlinearity, which enters through the activation functions $\phi_D$ and $\phi_E$.

### 7.1.4 ▪ Multi-layer autoencoders

After our introduction of the most simple form of an autoencoder, it is straightforward to construct more complicated *deep autoencoders*[31] by adding layers to the encoder and decoder networks.

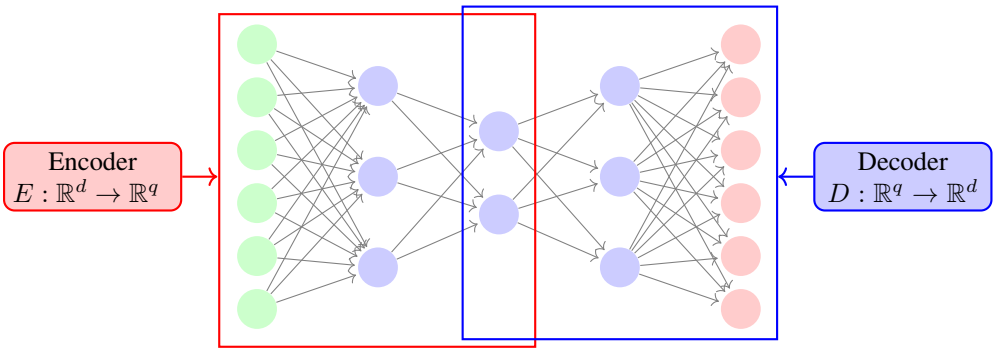



Figure 7.2: A multi-layer/deep autoencoder with $d = 6$ and $q = 2$.

Usually, the encoding layers shrink monotonically in size until the stage of highest compression is reached, whereas the decoding layers grow in size. An example of a four-layer autoencoder with a two-layer encoder and a two-layer decoder is depicted in Figure 7.2.

[31] The term *deep autoencoder* is also used to describe an architecture of several autoencoders chained together.

Besides fully connected networks, it is also possible to employ other types of networks within the AE framework. For instance, when working with images, it is quite natural to use 2D-convolutional layers in the encoder followed by pooling layers. The decoder is then built from convolutional layers and upsampling layers, which enlarge the image by using bilinear interpolation, for instance.

### 7.1.5 ▪ Regularization

As we have already seen in Section 7.1.3, the training of an AE is not a well-posed problem since it admits more than one possible solution. Indeed, already in the most simple case, there are infinitely many solutions for the weights even for the highly simplified setting from Section 7.1.3.

By adding nonlinearities via the activation functions this problem becomes much more severe. For instance, there exist nonlinear space-filling curves whose parametrization can be used as a decoder in order to perfectly reconstruct arbitrary high-dimensional vectors from real numbers; see [Sag12]. While this might be a meaningful way to compress the data, it is often desirable to have much more structured and smooth encoding and decoding functions. Numerically this smoothness serves to avoid instabilities when minimizing the loss functional, but it can also be beneficial to enable meaningful interpolations in the latent space.

We have already discussed that, with a rising number of degrees of freedom, overfitting quickly becomes a problem for deep neural networks. As a remedy, dropout or weight sharing were introduced in Chapter 6. Besides these methods, we want to hint at another way of regularizing the occurring minimization problem in the following: In order to regularize the problem, we can add a penalty term to the loss function. This enables us to steer the optimization algorithm to a more suitable solution, e.g., a smoother or more structured solution. For instance, we could penalize the (piecewise[32]) derivatives of the encoder and decoder to obtain smoother functions. This leads to the minimization problem

$$\arg\min_{D,E} \frac{1}{n} \sum_{i=1}^{n} \|\boldsymbol{x}_i - D \circ E(\boldsymbol{x}_i)\|^2 + \nu_1 \|\nabla_{\boldsymbol{x}} E(\boldsymbol{x})\|^2_{L_2(\mathbb{R}^d)} + \nu_2 \|\nabla_{\boldsymbol{\eta}} D(\boldsymbol{\eta})\|^2_{L_2(\mathbb{R}^q)}$$

when penalizing the (Bochner[33]) $L_2$ norms of the derivatives, for instance. Here, $\nu_1 > 0$ and $\nu_2 > 0$ are (hyperparameter) weights, which determine how strong the penalization should be.

A famous method of regularization is by penalizing large numbers of non-zero weights. For instance, we can sum the number of non-zero neurons in the output layer of the encoder and use this sum as a regularization term. This is called a *sparsity enforcing* penalty, which leads to so-called *sparse autoencoders*. Other variants compute the $\ell_1$- or $\ell_2$-norms of the encoder's last layer's neurons. More sophisticated ways to induce sparsity, e.g., by using *Kullback–Leibler divergences* of hidden unit activations, can be found in [Ng11].

**Kullback–Leibler divergence**

The ***Kullback–Leibler divergence*** is a specific measure for differences between two probability distributions. Given two distributions of continuous random variables with densities $p_1$ and $p_2$, the Kullback–Leibler divergence between $p_1$ and $p_2$ is defined by

$$\mathrm{KL}\left(p_1 \parallel p_2\right) := \mathbb{E}_{p_1}\left[\log\left(\frac{p_1}{p_2}\right)\right] = \int_{-\infty}^{\infty} p_1(x) \log\left(\frac{p_1(x)}{p_2(x)}\right) \mathrm{d}x. \tag{7.2}$$

[32] In the case of ReLU activations, the AE is non-differentiable.

[33] We can think of this as adding all squared $L_2$ norms of each component function of the derivatives.

For discrete distributions there exists an analogous expression with a sum over all possible values of the probability space instead of the integral. For more details on Kullback–Leibler divergences or the more general concept of ***Bregman divergences***, we refer the reader to [BMDG05].

A special variant of autoencoders called *denoising autoencoders* is trained on noisy/perturbed input vectors. To this end, the decoder aims to reconstruct the original unperturbed data pieces; see [GBC16]. This can also be interpreted as a special type of regularization by making the algorithm robust against noise. The methodology is comparable to the dropout regularization considered in Chapter 6.

## 7.2 ▪ Tasks on autoencoders

Before we introduce the variational variant of the autoencoder, we start with two tasks on standard autoencoders to get a feeling for the latent space. Furthermore, we compare the results of a PCA and an autoencoder for dimensionality reduction on the MNIST data set, which we introduced in Chapter 3.

**Task 7.1.** Import the MNIST data set. You can use the code introduced in Section 3.7. Trim the training and test data sets such that only the digits $0, 1, 2$, and $3$ are used. Run a PCA with latent dimension $q = 2$ on the training data and plot the images of the first ten digits of the original data set as well as the corresponding PCA reconstructions (i.e., project the images into two dimensions via the PCA and then calculate the corresponding high-dimensional representations thereof).
Create a plot of the latent space distribution of $1000$ random samples from the training data set, i.e., plot the two-dimensional coordinates of the embedded data points and use a color to indicate the corresponding label/digit. Furthermore, create a plot with a $25 \times 25$ grid of images corresponding to the high-dimensional representatives of the two-dimensional points at the latent space coordinates $x, y \in \{-4, -4+s, -4+2s, \ldots, 4-s, 4\}$ with $s = \frac{1}{3}$. An example of such a plot—but computed with a deep variational autoencoder instead of PCA—can be found in Figure 7.4 at the end of this chapter.

**Task 7.2.** Use KERAS to build a simple autoencoder with one hidden layer and $q = 2$ (see Figure 7.1) to encode and decode the trimmed MNIST training data set from Task 7.1. Train it by using Adam for a least squares loss with batch size $64$ for $15$ epochs. As for PCA in Task 7.1, plot the first ten digit reconstructions, the two-dimensional coordinates of the first $1000$ encoded samples, and the $25 \times 25$ grid with the images for the corresponding latent space coordinates. Compare the results to the ones from Task 7.1.

## 7.3 ▪ Latent variables and a probabilistic point of view

We will now adopt a more probabilistic point of view than before in order to introduce the concepts of *generative modeling* and *Bayesian inference*, which will be necessary to understand the motivation and mechanisms behind *variational autoencoders*.

**Generative modeling** Up to now we distinguished between unsupervised and supervised learning, where the former is characterized by the fact that the given data $\boldsymbol{x}_i$ from a data space $\Omega \subset \mathbb{R}^d$ is unlabeled. In contrast, the latter is characterized by the fact that the given data has some labels or target values, i.e., we have $(\boldsymbol{x}_i, \boldsymbol{y}_i)$ in a data space $\Omega \times \Gamma \subset \mathbb{R}^{d_1} \times \mathbb{R}^{d_2}$ for $i = 1, \dots, n$. Note that, in order to establish the following relationship between supervised and unsupervised learning, the target values may also be vectors here, i.e., we allow $d_2 > 1$. In supervised learning, we are interested in determining a generic $f : \Omega \to \Gamma$ such that $f(\boldsymbol{x}_i) \approx \boldsymbol{y}_i$. But besides the plain approximation of the given targets $\boldsymbol{y}_i$, we usually also aim for an $f$ that generalizes well to unseen data. At this point, recall the very beginning of Chapter 1, where the data was assumed to follow some unknown distribution $\mu$ on $\Omega \times \Gamma$, i.e.,

$$(\boldsymbol{x}_i, \boldsymbol{y}_i) \sim \mu \ \ \forall\, i = 1, \dots, n.$$

In statistics, this task can be looked at from a more general perspective. To this end, we consider the task of learning the conditional probabilities $\mathbb{P}_\mu(Y|X = \boldsymbol{x})$ of a random variable $Y$ on $\Gamma$ that is distributed according to the conditional measure $\mu_{Y|X}(\boldsymbol{x}, \cdot)$ of $Y$. Here, the measure is conditioned on the assumption that the outcome of a random variable $X$ on $\Omega$ distributed according to the marginal $\mu_X$ of $\mu$ on $\mathbb{R}^d$ is $X = \boldsymbol{x}$. This task is referred to as *discriminative* modeling; see also [NJ02]. Recall that for neural networks, for instance, we already learned such a probability distribution by applying the softmax function at the output layer for our classification tasks.

In the following, we return to the unsupervised learning task of dimensionality reduction, where only data points $\boldsymbol{x}_1, \dots, \boldsymbol{x}_n \in \mathbb{R}^d$ are present and latent space representatives $\boldsymbol{\eta}_1, \dots, \boldsymbol{\eta}_n \in \mathbb{R}^q$ are sought. We use this notation to reflect the resemblance to the dimensionality reduction tasks from before. Nonetheless, we retain the viewpoint of learning conditional probabilities between an ambient space variable $X \in \mathbb{R}^d$ and a latent space variable $Y \in \mathbb{R}^q$ in the same fashion as in the supervised example above.

**Generative modeling**

In contrast to discriminative modeling, where the conditional measure of $Y$ given $X$ is modelled, there is ***generative*** modeling [NJ02]. This term is sometimes used ambiguously to describe either the modeling of the joint distribution $\mathbb{P}_\mu(X, Y)$ or the modeling of the conditional distribution $\mathbb{P}_\mu(X|Y = \boldsymbol{\eta})$. Note that the latter distribution can be inferred from the joint one. In this book, we will stick to the case of approximating $\mathbb{P}_\mu(X|Y = \boldsymbol{\eta})$ when referring to generative modeling.

The idea of generative modeling might seem odd at first since we are aiming to infer information on the distribution of the observables given the latent variables. However, having a closer look at the decoder $D$ of our autoencoder in Section 7.1, we see that it is indeed trained to reconstruct the observables from their latent space representation. It therefore can be seen as a generative model. As a matter of fact, it *generates* samples given some latent coordinates.

**The latent space distribution** While the generative property of $D$ in our AE is nice to have (e.g., for data augmentation or for the computation of reconstruction errors), we did not yet model the decoder probabilistically and, thus, have no information on the latent space distribution. Thus, we are able to compute an $\boldsymbol{x} \in \mathbb{R}^d$ for a given $\boldsymbol{\eta} \in \mathbb{R}^q$, but we do not know how the reconstructed $\boldsymbol{x}$—or, more generally, $\mathbb{P}_\mu(X|Y)$—behaves if we vary $\boldsymbol{\eta}$ gradually. To this end, we need to have information (or a model) on the latent space distribution of $Y$ as well. This will be done in the following section by the variational autoencoders.

## 7.4 ▪ Variational autoencoders

To derive variational autoencoders, we now employ a generative modeling approach to approximate $\mathbb{P}_\mu(X|Y=\boldsymbol{\eta})$.

**Variational Autoencoders**

***Variational autoencoders*** (VAEs) create a generative decoder by modeling the latent space distribution and by approximating the conditionals $\mathbb{P}_\mu(X|Y)$ and $\mathbb{P}_\mu(Y|X)$. For a more detailed consideration of VAEs, we refer the reader to [Doe16, KW13, KW19, Mur23].

### 7.4.1 ▪ Motivation and idea

The latent space *prior*, i.e., the density $p(Y)$ that is assumed for $Y \in \mathbb{R}^q$, is chosen as a simple $q$-dimensional Gaussian density

$$p(Y) \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I}).$$

In order for this to be meaningful, the decoder density $p(X|Y)$ has to be powerful enough to adequately represent the true *conditional* $\mathbb{P}_\mu(X|Y)$. To this end, it is modeled as

$$p(X|Y=\boldsymbol{\eta}) \sim \mathcal{N}\left(f(\boldsymbol{\eta}), \sigma^2 \boldsymbol{I}\right),$$

where $\sigma^2 > 0$ is a fixed hyperparameter of the model and $f : \mathbb{R}^q \to \mathbb{R}^d$ is a (deep) neural network, which will be trained in such a way that it maps a latent vector $\boldsymbol{\eta}_i$ approximately to its high-dimensional representer $\boldsymbol{x}_i$. In this way, the reconstruction process can be quite complex, and the variance $\sigma^2$ takes account of possible noise or uncertainty in the reconstruction.

The criterion according to which we want to optimize the network's parameters is the maximization of the probability of obtaining the actual training data $\boldsymbol{X} = (\boldsymbol{x}_1 \dots \boldsymbol{x}_n)^T$ when we assume the above prior and conditional, i.e.,

$$\begin{aligned}
\arg\max_f \mathbb{P}_p(\boldsymbol{X}) &= \arg\max_f \log\left(\mathbb{P}_p(\boldsymbol{X})\right) = \arg\max_f \log(p(X=\boldsymbol{X})) \\
&= \arg\max_f \log\left(\prod_{i=1}^n p(X=\boldsymbol{x}_i)\right) = \arg\max_f \sum_{i=1}^n \log(p(X=\boldsymbol{x}_i)) \\
&= \arg\max_f \sum_{i=1}^n \log\left(\int_{\mathbb{R}^q} p(\boldsymbol{x}_i|Y=\boldsymbol{\eta}) p(\boldsymbol{\eta}) \, \mathrm{d}\boldsymbol{\eta}\right). \qquad (7.3)
\end{aligned}$$

This expression maximizes the probability of drawing $\boldsymbol{X}$ under the given model. The product over all data points appears due to the fact that the data $\boldsymbol{x}_i$ are assumed to be independent realizations of $X$. Computing or even approximating these integrals in every step of the optimization procedure is a hard and computationally intensive task. For instance, we could draw $m$ random samples $\boldsymbol{\eta}_j \sim p(Y)$ and compute $\sum_{i=1}^n \log\left(p(\boldsymbol{x}_i|Y=\boldsymbol{\eta}_j)\right)$ for each $j = 1, \dots, m$ to then approximate the above integrals by a $\frac{1}{m}$-normalized sum over the $m$ samples. But this so-called *Monte Carlo estimator* is known to converge only slowly to the true integral, and we thus need a very large number $m$ of samples to achieve a small approximation error.

**Monte Carlo estimator**

Let $N \in \mathbb{N}$ and let $f : \Omega \to \mathbb{R}$ be an integrable function. Furthermore, let $\boldsymbol{x}_i \in \Omega$ be independent samples drawn according to a measure $P$. The ***Monte Carlo*** (MC) estimator

$$MC_N(f) := \frac{1}{N} \sum_{i=1}^{N} f(\boldsymbol{x}_i)$$

uses $N$ samples to approximate the value of the integral $\int_\Omega f(\boldsymbol{x}) \, \mathrm{d}P(\boldsymbol{x})$. The motivation behind MC estimators is derived from the law of large numbers, which states that, for $N \to \infty$, the estimator will converge (e.g., in probability) with rate $\mathcal{O}\left(N^{-\frac{1}{2}}\right)$ to the true value of the integral if the variance of $MC_N(f)$ is bounded from above. More information on Monte Carlo estimators and variants thereof can be found in [Geo12].

An established method from statistics to reduce the number $m$ of necessary samples is *importance sampling*. Here, the $\boldsymbol{\eta}_j$ are drawn only in areas where they contribute much to the actual integrand, i.e., where $p(\boldsymbol{x}_i|Y = \boldsymbol{\eta}_j)$ is large. To this end, we construct a density $r(Y|X = \boldsymbol{x})$ on $\mathbb{R}^q$, which aims to sample those $\boldsymbol{\eta}$ that are likely to produce $\boldsymbol{x}$, i.e., for which $p(\boldsymbol{x}|Y = \boldsymbol{\eta})$ is large. Then, we sample according to $r(Y|X = \boldsymbol{x})$ instead of to the prior $p(Y)$. But we thus compute

$$\mathbb{E}_r \left[ p(\boldsymbol{x}|Y = \boldsymbol{\eta}) \right] := \int_{\mathbb{R}^q} p(\boldsymbol{x}|Y = \boldsymbol{\eta}) r(\boldsymbol{\eta}|X = \boldsymbol{x}) \, \mathrm{d}\boldsymbol{\eta} \tag{7.4}$$

instead of $\mathbb{E}_p \left[ p(\boldsymbol{x}|Y = \boldsymbol{\eta}) \right]$, which we wanted to maximize in the first place in (7.3). Therefore, we have to find a relation between the two quantities $\mathbb{E}_r \left[ p(\boldsymbol{x}|Y = \boldsymbol{\eta}) \right]$ and $\mathbb{E}_p \left[ p(\boldsymbol{x}|Y = \boldsymbol{\eta}) \right]$.

### 7.4.2 ▪ The ELBo as loss function

To derive a meaningful loss, which allows us to use (7.4) instead of computing $\mathbb{P}_p(\boldsymbol{x})$ directly, let us inspect the Kullback–Leibler divergence (7.2) between $r$ and the true *posterior density* $p(\boldsymbol{\eta}|X = \boldsymbol{x})$, i.e.,

$$\mathcal{K}_{r,p} := \mathrm{KL}\left( r(\boldsymbol{\eta}|X = \boldsymbol{x}) \parallel p(\boldsymbol{\eta}|X = \boldsymbol{x}) \right) := \mathbb{E}_r \left[ \log(r(\boldsymbol{\eta}|X = \boldsymbol{x})) - \log(p(\boldsymbol{\eta}|X = \boldsymbol{x})) \right].$$

To deal with this expression, we need to reformulate it first.

**Bayes' theorem**

***Bayes' theorem*** states that the posterior probability $\mathbb{P}[A \mid B]$ of event $A$ given event $B$ with $\mathbb{P}[B] > 0$ can be written as

$$\mathbb{P}[A \mid B] = \frac{\mathbb{P}[B \mid A] \, \mathbb{P}[A]}{\mathbb{P}[B]}.$$

Here, $\mathbb{P}[A]$ is the prior probability, $\mathbb{P}[B \mid A]$ is the ***likelihood*** of $B$ under $A$, and $\mathbb{P}[B]$ is called ***evidence***; see [Geo12] for more details. The application of Bayes' theorem to compute the posterior is called ***Bayesian inference***.

With Bayes' theorem we can relate the posterior to the prior via

$$p(\boldsymbol{\eta}|X = \boldsymbol{x}) = \frac{p(\boldsymbol{x}|Y = \boldsymbol{\eta}) p(Y = \boldsymbol{\eta})}{p(X = \boldsymbol{x})}.$$

Substituting this into the Kullback–Leibler divergence gives

$$\mathcal{K}_{r,p} = \mathbb{E}_r\left[\log(r(\boldsymbol{\eta}|X=\boldsymbol{x})) - \log(p(\boldsymbol{x}|Y=\boldsymbol{\eta})) - \log(p(Y=\boldsymbol{\eta}))\right] + \log(p(X=\boldsymbol{x})),$$

which is equivalent to

$$\underbrace{\log(p(X=\boldsymbol{x}))}_{\text{log-evidence}} - \mathcal{K}_{r,p} = \underbrace{\mathbb{E}_r\left[\log\left(p(\boldsymbol{x}|Y=\boldsymbol{\eta})\right)\right] - \mathrm{KL}\left(r(\boldsymbol{\eta}|X=\boldsymbol{x}) \parallel p(Y=\boldsymbol{\eta})\right)}_{\text{evidence lower bound (ELBo)}}. \tag{7.5}$$

This is the essential equation for the optimization of VAEs. The right-hand side resembles the so-called *evidence lower bound* (ELBo). The name stems from the fact that $\mathcal{K}_{r,p} \geq 0$ and $p(X = \boldsymbol{X})$ from (7.3) is the (model) evidence. We see that the ELBo equals the log-evidence up to the Kullback–Leibler divergence $\mathcal{K}_{r,p}$ between $r$ and the true posterior. Therefore, if $r(\boldsymbol{\eta}|X = \boldsymbol{x})$ is a good approximation to $p(\boldsymbol{\eta}|X = \boldsymbol{x})$, $\mathcal{K}_{r,p}$ becomes small and we are essentially maximizing (7.3) when maximizing the sum of the ELBos for each $\boldsymbol{x} = \boldsymbol{x}_i$. In this way, maximizing the ELBo achieves two things:

1. We maximize the likelihood of the given data $\boldsymbol{X}$, and, thus, our generative model $p(X|Y)$ becomes better.
2. We minimize the KL divergence between $r$ and the true posterior, and, thus, the so-called *inference* model $r$ becomes better.

### 7.4.3 ▪ Minimization of the negative ELBo

By our above considerations, we now see that drawing a sample according to $r(\cdot|X = \boldsymbol{x})$ acts like an encoder for $\boldsymbol{x} \in \mathbb{R}^d$, and drawing a sample according to $p(\cdot|Y = \boldsymbol{\eta})$ acts like a decoder for $\boldsymbol{\eta} \in \mathbb{R}^q$. In this sense, we have an encoder-decoder pair for variational autoencoders.

Before we can however maximize the ELBo, i.e., minimize the negative ELBo according to the model parameters, we still have to choose an appropriate model for $r$. To this end, one usually takes

$$r(\cdot|X=\boldsymbol{x}) \sim \mathcal{N}\left(g_1(\boldsymbol{x}), \exp(g_2(\boldsymbol{x})) \cdot \boldsymbol{I}\right)$$

for two (deep) neural networks $g_1, g_2 : \mathbb{R}^d \to \mathbb{R}^q$ encoding the mean and the variance structure of $r$. Here, $\exp$ has to be understood componentwise. Usually, $g_1$ and $g_2$ employ the same network up to the very last layer. Now the full task to learn a VAE becomes

$$(\tilde{f}, \tilde{g_1}, \tilde{g_2}) = \underset{f,g_1,g_2}{\arg\min} - \sum_{i=1}^{n} \mathbb{E}_r\left[\log\left(p(\boldsymbol{x}_i|Y=\boldsymbol{\eta})\right)\right] + \mathrm{KL}\left(r(\boldsymbol{\eta}|X=\boldsymbol{x}_i) \parallel p(Y=\boldsymbol{\eta})\right), \tag{7.6}$$

where the minimization has to be understood with respect to the weights and biases of the networks corresponding to $f$, $g_1$, and $g_2$. This can easily be done by the known optimizers from Section 6.5 if we are able to evaluate the ELBo in an efficient way. Note that we usually would not sum over all $i = 1, \ldots, n$ data points in (7.6) in each optimization step, but rather use a minibatch approach as in Algorithm 10 in Section 6.3.

Let us now consider the terms of the ELBo in more detail. By modeling the prior $p$ and the density $r$ according to the corresponding normal distributions above, the Kullback–Leibler divergences in (7.6) are taken between two Gaussians. In this case we have two densities $\hat{p}_k \sim \mathcal{N}(\boldsymbol{m}_k, \boldsymbol{\Sigma}_k)$ for $k = 1, 2$, given by

$$\hat{p}_k(\boldsymbol{\eta}) = \frac{1}{(2\pi)^{\frac{q}{2}} \det(\boldsymbol{\Sigma}_k)^{\frac{1}{2}}} \exp\left(-\frac{1}{2}(\boldsymbol{\eta} - \boldsymbol{m}_k)^T \boldsymbol{\Sigma}_k^{-1} (\boldsymbol{\eta} - \boldsymbol{m}_k)\right).$$

With $\mathcal{K}_{\hat{p}_1,\hat{p}_2} := \mathrm{KL}\left(\hat{p}_1 \parallel \hat{p}_2\right) = \mathbb{E}_{\hat{p}_1}\left[\log \hat{p}_1 - \log \hat{p}_2\right]$ this leads to

$$\begin{aligned}\mathcal{K}_{\hat{p}_1,\hat{p}_2} &= \frac{1}{2}\log\left(\frac{\det \boldsymbol{\Sigma}_2}{\det \boldsymbol{\Sigma}_1}\right) + \frac{1}{2}\mathbb{E}_{\hat{p}_1}\left[(\boldsymbol{\eta}-\boldsymbol{m}_2)^T\boldsymbol{\Sigma}_2^{-1}(\boldsymbol{\eta}-\boldsymbol{m}_2) - (\boldsymbol{\eta}-\boldsymbol{m}_1)^T\boldsymbol{\Sigma}_1^{-1}(\boldsymbol{\eta}-\boldsymbol{m}_1)\right] \\ &= \frac{1}{2}\log\left(\frac{\det \boldsymbol{\Sigma}_2}{\det \boldsymbol{\Sigma}_1}\right) + \frac{1}{2}\mathbb{E}_{\hat{p}_1}\left[\mathrm{tr}\left(\boldsymbol{\Sigma}_2^{-1}(\boldsymbol{\eta}-\boldsymbol{m}_2)(\boldsymbol{\eta}-\boldsymbol{m}_2)^T\right)\right] \\ &\quad -\frac{1}{2}\mathbb{E}_{\hat{p}_1}\left[\mathrm{tr}\left(\boldsymbol{\Sigma}_1^{-1}(\boldsymbol{\eta}-\boldsymbol{m}_1)(\boldsymbol{\eta}-\boldsymbol{m}_1)^T\right)\right]\end{aligned}$$

since $\boldsymbol{y}^T\boldsymbol{x} = \mathrm{tr}(\boldsymbol{x}\boldsymbol{y}^T)$ for any two vectors $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^q$, where $q$ is the dimension of the latent space. Because of $\mathbb{E}_{\hat{p}_1}\left[(\boldsymbol{\eta}-\boldsymbol{m}_1)(\boldsymbol{\eta}-\boldsymbol{m}_1)^T\right] = \boldsymbol{\Sigma}_1$, we obtain

$$\mathcal{K}_{\hat{p}_1,\hat{p}_2} = \frac{1}{2}\log\left(\frac{\det \boldsymbol{\Sigma}_2}{\det \boldsymbol{\Sigma}_1}\right) + \frac{1}{2}\mathbb{E}_{\hat{p}_1}\left[\mathrm{tr}\left(\boldsymbol{\Sigma}_2^{-1}\left(\boldsymbol{\eta}\boldsymbol{\eta}^T - 2\boldsymbol{m}_2\boldsymbol{\eta}^T + \boldsymbol{m}_2\boldsymbol{m}_2^T\right)\right)\right] - \frac{1}{2}\mathrm{tr}\left(\boldsymbol{\Sigma}_1^{-1}\boldsymbol{\Sigma}_1\right).$$

Note that $\mathrm{tr}\left(\boldsymbol{\Sigma}_1^{-1}\boldsymbol{\Sigma}_1\right) = \mathrm{tr}\left(\boldsymbol{I}_q\right) = q$, where $\boldsymbol{I}_q \in \mathbb{R}^{q\times q}$ is the identity matrix. Using this together with $\mathbb{E}_{\hat{p}_1}\left[\boldsymbol{\eta}\boldsymbol{\eta}^T\right] = \boldsymbol{\Sigma}_1 + \mathbb{E}_{\hat{p}_1}\left[2\boldsymbol{\eta}\boldsymbol{m}_1^T - \boldsymbol{m}_1\boldsymbol{m}_1^T\right] = \boldsymbol{\Sigma}_1 + \boldsymbol{m}_1\boldsymbol{m}_1^T$, we get

$$\begin{aligned}\mathcal{K}_{\hat{p}_1,\hat{p}_2} &= \frac{1}{2}\log\left(\frac{\det \boldsymbol{\Sigma}_2}{\det \boldsymbol{\Sigma}_1}\right) - \frac{1}{2}q + \frac{1}{2}\mathrm{tr}\left(\boldsymbol{\Sigma}_2^{-1}\left(\boldsymbol{\Sigma}_1 + \boldsymbol{m}_1\boldsymbol{m}_1^T - 2\boldsymbol{m}_2\boldsymbol{m}_1^T + \boldsymbol{m}_2\boldsymbol{m}_2^T\right)\right) \\ &= \frac{1}{2}\log\left(\frac{\det \boldsymbol{\Sigma}_2}{\det \boldsymbol{\Sigma}_1}\right) - \frac{1}{2}q + \frac{1}{2}\mathrm{tr}\left(\boldsymbol{\Sigma}_2^{-1}\boldsymbol{\Sigma}_1\right) + \frac{1}{2}\left(\boldsymbol{m}_2 - \boldsymbol{m}_1\right)^T\boldsymbol{\Sigma}_2^{-1}\left(\boldsymbol{m}_2 - \boldsymbol{m}_1\right).\end{aligned}$$

With $\hat{p}_1 = r(\cdot|X=\boldsymbol{x}) \sim \mathcal{N}(g_1(\boldsymbol{x}), \exp(g_2(\boldsymbol{x}))\boldsymbol{I})$ and $\hat{p}_2 = p(Y=\cdot) \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{I})$, we then obtain

$$\begin{aligned}\mathcal{K}_{\hat{p}_1,\hat{p}_2} &= \mathrm{KL}\left(r(\boldsymbol{\eta}|X=\boldsymbol{x}) \parallel p(Y=\boldsymbol{\eta})\right) \\ &= \frac{1}{2}\left(-\sum_{i=1}^{q}\left(g_2(\boldsymbol{x})\right)_i - q + \sum_{i=1}^{q}\exp\left(\left(g_2(\boldsymbol{x})\right)_i\right) + \sum_{i=1}^{q}\left(g_1(\boldsymbol{x})\right)_i^2\right),\end{aligned}$$

which lets us directly evaluate the KL divergence in (7.6).

To compute the term (7.4) appearing in (7.6), we could build a Monte Carlo estimator as described earlier. However, this would be quite expensive. Note again that our optimizer will be computing a minibatch variant of SGD. As mentioned in Section 6.3, this is an unbiased estimator for the expectation of our loss function. The same reasoning applies when we sample $r$. Therefore, instead of building a full Monte Carlo estimator for (7.4), we only draw a single sample $\boldsymbol{\eta}_i \sim r(\cdot|X=\boldsymbol{x}_i)$ for each $\boldsymbol{x}_i$ in the minibatch and use the *one-shot estimator* $\log p(\boldsymbol{x}_i|Y=\boldsymbol{\eta}_i)$ as an unbiased estimator[34] for (7.4). Then, we just average over all samples in the minibatch to evaluate the loss function.

### 7.4.4 ▪ Reparametrization trick

Now we are able to compute the ELBo in order to evaluate our loss in (7.6) quite efficiently for the optimization. But there is one additional issue to deal with: We cannot directly use backpropagation to take gradients with respect to the model parameters of $g_1$ and $g_2$ because of the expectation operator $\mathbb{E}_r$ in (7.6), which nonlinearly depends on $g_1$ and $g_2$. The problem boils down to the fact that $\nabla_{g_1,g_2}\mathbb{E}_r[\ldots] \neq \mathbb{E}_r[\nabla_{g_1,g_2}\ldots]$, where $\nabla_{g_1,g_2}$ refers to the gradient

[34] Ultimately, this is an unbiased estimator for $\mathbb{E}_\mu\left[\mathbb{E}_r\left[\log\left(p(\boldsymbol{x}_i|Y=\boldsymbol{\eta}_i)\right)\right]\right]$, where $\mu$ is the measure according to which we have drawn our training data.

with respect to the weights and biases of $g_1$ and $g_2$. Thus, we indeed can use the simple one-shot estimator described in Section 7.4.3 for forward propagation of the corresponding neural networks, but we cannot use it directly for backward propagation. To remedy this issue, we first need to employ the so-called *reparametrization trick*: We parametrize a random variable $\boldsymbol{\eta}(\boldsymbol{x})$ drawn according to $r(\boldsymbol{\eta}|X=\boldsymbol{x}) \sim \mathcal{N}(g_1(\boldsymbol{x}), \exp(g_2(\boldsymbol{x}))\boldsymbol{I})$ by

$$\boldsymbol{\eta}(\boldsymbol{x}) := g_1(\boldsymbol{x}) + \sqrt{\exp(g_2(\boldsymbol{x}))}\boldsymbol{I} \cdot \boldsymbol{z}, \tag{7.7}$$

where $\boldsymbol{z} \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{I})$ is a standard normal distributed random variable. Here, the square root of $\exp(g_2)$ has to be understood componentwise. Then,

$$\begin{aligned}\nabla_{g_1,g_2}\mathbb{E}_r\left[\log\left(p(\boldsymbol{x}|Y=\boldsymbol{\eta})\right)\right] &= \nabla_{g_1,g_2}\mathbb{E}_{\boldsymbol{z}}\left[\log\left(p\left(\boldsymbol{x}|Y=g_1(\boldsymbol{x})+\sqrt{\exp(g_2(\boldsymbol{x}))}\boldsymbol{I}\cdot\boldsymbol{z}\right)\right)\right] \\ &= \mathbb{E}_{\boldsymbol{z}}\left[\nabla_{g_1,g_2}\log\left(p\left(\boldsymbol{x}|Y=g_1(\boldsymbol{x})+\sqrt{\exp(g_2(\boldsymbol{x}))}\boldsymbol{I}\cdot\boldsymbol{z}\right)\right)\right],\end{aligned}$$

which is computable in a straightforward way again.

### 7.4.5 ▪ Training and generation

Now we are finally set to learn a variational autoencoder by using the reparametrization trick to apply a minibatch SGD algorithm (or variants thereof) for the minimization of the negative ELBo loss (7.6). This determines all weights involved in the representation of $f$, $g_1$, and $g_2$. The full VAE architecture is depicted in Figure 7.3.

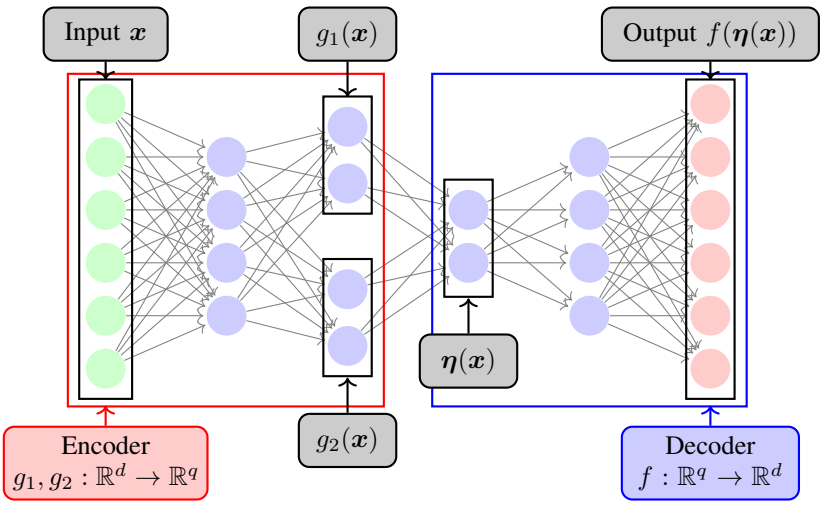



Figure 7.3: A variational autoencoder with $d = 6$ and $q = 2$. The encoder computes the vectors $g_1(\boldsymbol{x})$ and $g_2(\boldsymbol{x})$ for the mean and the log-variance of the distribution $r$. Then $\boldsymbol{\eta}(\boldsymbol{x})$ is formed according to (7.7). Consequently, the decoder computes $f(\boldsymbol{\eta}(\boldsymbol{x}))$.

The encoder, when called with a high-dimensional sample $\boldsymbol{x}$, now computes $g_1(\boldsymbol{x})$ and $g_2(\boldsymbol{x})$. A corresponding latent space sample (according to $r$) can then be drawn by the reparametrization trick (7.7).

In order to generate new high-dimensional samples from a random latent space element, we simply draw $\boldsymbol{\eta} \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{I})$ and then run our trained decoder to finally obtain $f(\boldsymbol{\eta})$.

## 7.5 ▪ Tasks on variational autoencoders

Now we implement a variational autoencoder to compare its results with the AE from Task 7.2.

**Task 7.3.** Implement a variational autoencoder (see Figure 7.3) with $q = 2$. The encoder only consists of the input layer and the layer for $g_1$ and $g_2$. The decoder only consists of the layer for $\boldsymbol{\eta}$ and the output layer. The variance of the decoder is set to $\sigma^2 = 1$. Plot the analogous reconstructions, the scattered data in the latent space, and the $25 \times 25$ latent space grid as in Task 7.2. What do you observe?
Hints:

- To draw samples in the latent space, you can use the `keras.backend` functions, e.g., `keras.backend.random_normal`.
- By using the `layers` and `Model` class from KERAS, you can build the encoder by defining the output as a list of outputs, e.g.,

  ```
  keras.Model(name="V_Encoder", inputs=inputs,
              outputs=[mean, log_var, sample]),
  ```

  where `mean`, `log_var`, and `sample` are the outputs of the corresponding layers evaluated on the inputs.
- Define a custom loss function

  ```
  def VAE_loss(data, results):
  ```

  in which you compute the negative ELBo as described in Section 7.4.3. The `keras.backend` functions, e.g., `mean`, `exp`, `sum`, should be helpful again.

Next, we investigate how a slightly deeper VAE performs.

**Task 7.4.** Add one intermediate layer to both the encoder and decoder from Task 7.3 with size half of the input layer size. Redo the experiments of Task 7.3 and observe the difference in the resulting plots. An example for the $25 \times 25$ latent space grid plot can be found in Figure 7.4.

## 7.6 ▪ Further topics

**Adversarial neural networks** Another interesting research topic on classification in combination with generative methods is the study of attacks against neural networks. For example, [BMR+17] designed a sticker that you can patch onto (or near) any object to make a CNN classify it as a toaster and not as the true object anymore. In [EEF+18] the authors investigate how robustly a CNN can detect traffic stop signs when the sign has graffiti on it. Finally, [PMG+17] provides examples for manipulated but visually indistinguishable images of traffic signs that get classified wrongly by a CNN.

To remedy the problems caused by such attacks *generative adversarial networks* (GANs) have been invented. The concept of GANs [GPAM+14] stems from *zero-sum games* in game theory. The idea is to train two adversarial networks: One (*discriminator*) that tries to classify the data correctly and one (*generator*) that generates data on-the-fly with the aim of maximizing the probability of the first network misclassifying this data. In this way, the generator mimics attacks

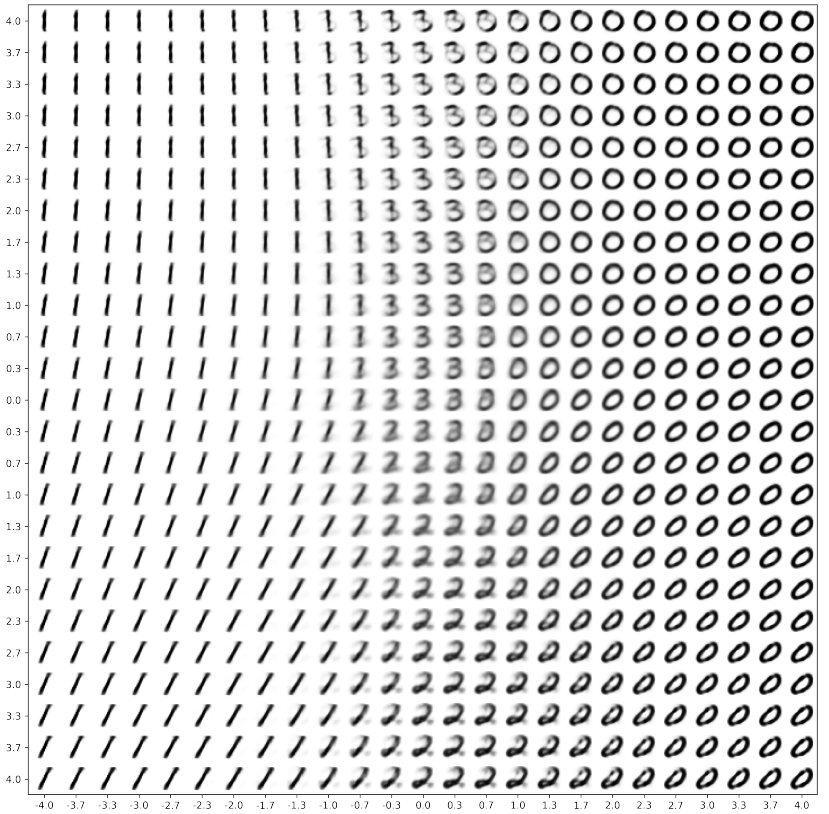



Figure 7.4: Illustration of the two-dimensional latent space for a variational autoencoder with three hidden layers trained on the first four digits (0, 1, 2, and 3) of the MNIST data set. We divided the area $[-4, 4]^2$ of the latent space into a $25 \times 25$ grid. Subsequently, we drew a sample at each grid point and used the decoder to obtain the corresponding image in the high-dimensional space. The resulting $25 \times 25$ images are shown at their respective coordinate positions. Created from the MNIST data set.

against the discriminator. By training the discriminator with the help of the data produced from the generator, it becomes more robust against such attacks.

**Interpretable latent spaces** Besides having a meaningful prior distribution in the latent space, which can be used to generate high-dimensional samples via the decoder, it is also desirable to be able to interpret the coordinates of the latent space. This can be of special interest if one tries to infer causal relationships between latent space variables and high-dimensional sample behavior. To this end, special regularization terms are employed to guarantee an interpretable latent space distribution; see, e.g., [TK21].

**Dynamical variational autoencoders** So far, the introduced autoencoders treated each input data point independently when building the latent space. However, when there are temporal correlations in the data (e.g., for time series data), it makes sense to use latent spaces and autoencoder models that take this into account as well. Here, so-called dynamical variational autoencoders aim to model joint distributions of whole sequences of input and latent space data. We refer the reader to [GLB+21] for a detailed survey.

Chapter 8

# Deep Neural Networks and Differential Equations

In this chapter, we will explore relationships between neural network architectures and differential equations, especially ordinary differential equations (ODEs). In particular, we will have a look at neural tangent kernels [JGH18] in Section 8.1, Hamiltonian networks [HR18] in Section 8.2, neural ODEs [CRBD18] in Section 8.3, and generative diffusion models [HJA20] in Section 8.4. Here, we aim to hint at the relation between those methods and differential equations without going into technical details. For a more thorough reference on these methods and their connection to differential equations, we refer the reader to the corresponding cited articles and to [EMW20].

In contrast to other chapters, we do not provide any tasks in this chapter since the covered topics are quite advanced and more theoretical in nature. Furthermore, they serve to highlight the connection between modern machine learning methods and advanced mathematical concepts instead of dealing with the algorithmic details of these methods. However, we feel that this chapter is still quite important when it comes to understanding the mathematical background of neural networks.

## 8.1 ▪ Neural tangent kernels

A connection between kernels and the learning process of neural networks with gradient descent-type optimizers like SGD or Adam (see Section 6.5) can be drawn by introducing the *neural tangent kernel* (NTK); see also [JGH18]. To this end, we write the output of a given neural network with parameters $\boldsymbol{p} \in \mathbb{R}^{d_p}$, i.e., weights and biases, and input vector $\boldsymbol{z} \in \mathbb{R}^d$ as $f(\boldsymbol{z}; \boldsymbol{p})$ in this chapter, whereas we mostly omitted the parameter dependence in Chapter 6 for reasons of readability. Furthermore, we here assume that our network has only one output neuron, i.e., $f(\boldsymbol{z}; \boldsymbol{p}) \in \mathbb{R}$.

### 8.1.1 ▪ Relationship between network features and kernel methods

While the weights and the biases are degrees of freedom of our neural network model class, the (possibly nonlinear) activation functions $\phi^{(l)}$ are usually fixed a priori. Let the activation function of the final layer be $\phi^{(L+1)} = \mathrm{id}$ and let $\phi^{(l)}$ be arbitrary for $l = 1, \dots, L$. We can then rewrite our model as

$$f(\boldsymbol{z}; \boldsymbol{p}) = \left(\vec{w}^{(L)}\right)^T \cdot \psi(\boldsymbol{z}; \boldsymbol{p}) + b^{(L+1)}$$

for a vector-valued function $\psi : \mathbb{R}^d \times \mathbb{R}^{d_p} \to \mathbb{R}^{d_L}$, which depends on the input, but also on the weights, biases, and activation functions of the previous $l = 1, \dots, L-1$ layers. In this way, we have a direct analogy to SVM or kernel methods in general, where $\psi$ reflects the feature map that is chosen to transform the data; see (3.10) and (3.11).

However, the difference between the SVM and the hidden-layer neural network model is that $\psi$—or equivalently the corresponding kernel $K$—has been chosen a priori for an SVM, whereas here, i.e., in the deep neural network model, $\psi$ depends on the degrees of freedom (namely the weights and biases of the hidden layers). Therefore, the kernel changes during the optimization steps of, e.g., stochastic gradient descent. The idea behind the neural tangent kernel is to define a kernel that—in special settings—achieves both: It resembles the network's structure and it stays constant during the learning process. In this way, we can use the kernel to run methods like SVM (see Section 3.5) or kernel ridge regression (see [Mur22, SS02]), which have a well-understood convergence behavior and guaranteed global optima, instead of running SGD-type optimizers for highly nonconvex and nonlinear optimization problems that stem from deep neural networks.

Besides the idea of neural tangent kernels, there also exist methods that are hybrids between kernel methods and deep neural networks called *deep kernel networks*; see, e.g., [BGR19].

### 8.1.2 ▪ Neural tangent kernels and gradient flows

We will now have a look at the definition of the neural tangent kernel. To this end, let all $\phi^{(l)}$ for $l = 1, \dots, L+1$ be arbitrary activation functions again. We will alter the network's forward propagation procedure (6.1) slightly by rescaling it with $d_l$, the number of weights in a layer, and adding a scaling parameter $\beta_{l+1} > 0$ for the bias. We define

$$\vec{\text{net}}^{(l+1)} := \frac{1}{\sqrt{d_l}} \left( \boldsymbol{W}^{(l)} \right)^T \cdot \vec{o}^{(l)} + \beta_{l+1} \vec{b}^{(l+1)}$$

to be the forward step in layer $l$ before the application of $\phi^{(l+1)}$. This redefinition is necessary to achieve the upcoming results for large to infinite layer sizes, which we will discuss later in this section.

**Neural tangent kernel**

The ***neural tangent kernel*** (NTK) of an artificial neural network function $f$ is a parameter-dependent kernel—see also Section 3.5 for the definition of kernels—$K_{\text{NTK}}(\cdot, \cdot; \boldsymbol{p}) : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$ defined by

$$K_{\text{NTK}}(\boldsymbol{x}, \boldsymbol{y}; \boldsymbol{p}) := \psi_{\text{NTK}}(\boldsymbol{x}; \boldsymbol{p})^T \psi_{\text{NTK}}(\boldsymbol{y}; \boldsymbol{p}) = \sum_{i=1}^{d_p} \frac{\partial}{\partial p_i} f(\boldsymbol{x}; \boldsymbol{p}) \frac{\partial}{\partial p_i} f(\boldsymbol{y}; \boldsymbol{p}) \quad (8.1)$$

with the corresponding feature map[35] $\psi_{\text{NTK}}(\boldsymbol{x}; \boldsymbol{p}) := \nabla_{\boldsymbol{p}} f(\boldsymbol{x}; \boldsymbol{p})$. The NTK serves to describe the dynamics of the training process of a neural network with gradient descent-type methods such as SGD or Adam. More details can be found in [JGH18, RYH22].

In order to establish the connection between the NTK and the network's training process, let us first look at the $k$th update step of SGD. Recalling Section 6.5, it can be written as

$$\boldsymbol{p}_{k+1} = \boldsymbol{p}_k - \nu \nabla_{\boldsymbol{p}} \Theta_B(\boldsymbol{p}_k),$$

[35] The name *neural tangent kernel* stems from the fact that the feature map $\psi_{\text{NTK}}$ is the tangent vector of $f$ in the space of weights and biases of the network.

with step size $\nu$ and $\Theta_B(\boldsymbol{p}_k)$ being the evaluation of the (batch) loss function $C_B$ of the network's output $f$ for the current weight and bias parameters $\boldsymbol{p}_k$. If we rewrite this as

$$\frac{\boldsymbol{p}_{k+1} - \boldsymbol{p}_k}{\nu} = -\nabla_{\boldsymbol{p}} \Theta_B(\boldsymbol{p}_k), \tag{8.2}$$

we can think of the left-hand side as a discrete derivative of the trajectory of the iterates $\boldsymbol{p}_k$. In the limit $\nu \to 0$ we would just encounter the derivative of a time-dependent $\boldsymbol{p}$, i.e., $\frac{\partial}{\partial t}\boldsymbol{p}(t)$. In this regard, we can reinterpret (8.2) as a *time-explicit Euler discretization* with *step width* $\nu$ for the ordinary differential equation

$$\frac{\partial}{\partial t}\boldsymbol{p}(t) = -\nabla_{\boldsymbol{p}} \Theta_B(\boldsymbol{p}(t)). \tag{8.3}$$

The trajectory described by this ordinary differential equation is often called *gradient flow*.

**Ordinary differential equations**

Differential equations of type (8.3), where only derivatives with respect to one single variable are present, are called ***ordinary differential equations*** (ODEs). The discretization, the stability analysis, and the numerical treatment of such equations have a long history and are well understood; see [WH96, WNH93] for details. A straightforward way to approximately solve an ODE of type (8.3) for a given start value $\boldsymbol{p}_0$ is to use the so-called time-explicit Euler scheme (8.2) and to iterate the solution $k$ steps forward in time with a step size $\nu > 0$ until a desired final point in time $T = \tilde{k}\nu$ is reached. To guarantee that this results in a convergent and stable algorithm, $\nu$ has to fulfill certain criteria (in view of Dahlquist's test equation and linear stability) based on the specific ODE terms; see, e.g., [WH96, WNH93]. More sophisticated algorithms like the Runge–Kutta methods use more than just one evaluation of the right-hand side of (8.3) per time step in order to achieve better accuracies.

Having a closer look at (8.3) and recalling Section 6.3, we see that the right-hand side is given by

$$\begin{aligned} -\nabla_{\boldsymbol{p}} \Theta_B(\boldsymbol{p}) &= -\frac{1}{|B|} \sum_{i \in B} \nabla_{\boldsymbol{p}}\, C_i(f(\cdot;\boldsymbol{p})) = -\frac{1}{|B|} \sum_{i \in B} C_i'(f(\cdot;\boldsymbol{p})) \nabla_{\boldsymbol{p}} f(\boldsymbol{x}_i, \boldsymbol{p}) \\ &= -\frac{1}{|B|} \sum_{i \in B} C_i'(f(\cdot;\boldsymbol{p})) \psi_{\mathrm{NTK}}(\boldsymbol{x}_i;\boldsymbol{p}), \end{aligned} \tag{8.4}$$

and the evaluation essentially boils down to computing the feature map evaluations $\psi_{\mathrm{NTK}}(\boldsymbol{x}_i, \boldsymbol{p})$ because the derivative $C_i'(f(\cdot, \boldsymbol{p}))$ of the one-sample least squares loss is directly accessible. For a least squares loss function, for example, we have $C_i'(f(\cdot, \boldsymbol{p})) = 2(f(\boldsymbol{x}_i;\boldsymbol{p}) - y_i)$; see also the backpropagation equations in Section 6.3.

The connection between the kernel and the dynamics of the gradient flow becomes even more obvious when we use the chain rule to compute

$$\begin{aligned} \frac{\partial f(\boldsymbol{x};\boldsymbol{p}(t))}{\partial t} &= (\nabla_{\boldsymbol{p}} f(\boldsymbol{x};\boldsymbol{p}(t)))^T \frac{\partial \boldsymbol{p}(t)}{\partial t} \\ &\overset{(8.3),(8.4)}{=} -\frac{1}{B} \sum_{i \in B} C_i'(f(\cdot;\boldsymbol{p}(t)))\ (\nabla_{\boldsymbol{p}} f(\boldsymbol{x};\boldsymbol{p}(t)))^T \nabla_{\boldsymbol{p}} f(\boldsymbol{x}_i;\boldsymbol{p}(t)) \\ &= -\frac{1}{B} \sum_{i \in B} C_i'(f(\cdot;\boldsymbol{p}(t)))\ K_{\mathrm{NTK}}(\boldsymbol{x}, \boldsymbol{x}_i;\boldsymbol{p}(t)). \end{aligned}$$

Here, we directly see that the NTK determines how $f$ evolves over the course of time, i.e., during the iteration process of SGD.

### 8.1.3 ▪ Convergence properties of stochastic gradient descent

One major benefit of analyzing the NTK is that it can give insights on the convergence properties of the SGD process. To this end, note that the time derivative of the loss function $\Theta_B(\boldsymbol{p})$ from (6.8) is given by

$$\begin{aligned}\frac{\partial \Theta_B(\boldsymbol{p}(t))}{\partial t} &= \left(\nabla_{\boldsymbol{p}} \Theta_B(\boldsymbol{p})\right)^T \frac{\partial \boldsymbol{p}(t)}{\partial t} = -\|\nabla_{\boldsymbol{p}} \Theta_B(\boldsymbol{p})\|^2 \\ &= -\frac{1}{|B|^2} \left\| \sum_{i \in B} \sum_{j \in B} C_i'(f(\cdot;\boldsymbol{p}(t))) C_j'(f(\cdot;\boldsymbol{p}(t))) K_{\text{NTK}}(\boldsymbol{x}_i, \boldsymbol{x}_j; \boldsymbol{p}(t)) \right\|^2 .\end{aligned}$$

From this equation, we can derive that $\Theta_B(\boldsymbol{p}(t))$ is strictly decreasing over time if the NTK is positive definite for all $\boldsymbol{p}(t)$; see also Section 3.5 for more details on kernel properties like positive definiteness. Therefore, if we have a convex loss function that is bounded from below, like, e.g., the least squares loss function, the SGD algorithm will converge to the *global* optimum if the NTK is positive definite.

In this way, the positive definiteness of the NTK serves as a sufficient criterion for convergence of the network's optimization process. However, analyzing $K_{\text{NTK}}$ for all possible trajectories of $\boldsymbol{p}(t)$ is usually not possible. Therefore, we aim for a setting where the kernel does not change (much) during the SGD process.

### 8.1.4 ▪ Neural tangent kernels in the infinite-width limit

Let us now consider a first-order Taylor approximation in $\boldsymbol{p}$ of the network's function $f(\boldsymbol{z};\boldsymbol{p})$.

**Taylor expansion formula**

The ***Taylor formula*** serves to approximate the value of a real-valued, $k$-times differentiable function $f : \mathbb{R}^d \to \mathbb{R}$ within a local neighborhood of a reference point $\boldsymbol{x}_0 \in \mathbb{R}^d$ by a polynomial of degree $k$. It reads

$$f(\boldsymbol{x}) = \sum_{|\boldsymbol{k}|_1 \leq k} \frac{\partial^{|\boldsymbol{k}|_1} f(\boldsymbol{x}_0)}{\partial t_1^{k_1} \cdots t_d^{k_d}} \cdot \frac{(\boldsymbol{x} - \boldsymbol{x}_0)^{\boldsymbol{k}}}{\boldsymbol{k}!} + \mathcal{O}\left(\|\boldsymbol{x} - \boldsymbol{x}_0\|^{k+1}\right),$$

where $|\boldsymbol{k}|_1$ is the $\ell_1$ norm of a multi-index $\boldsymbol{k} = (k_1, \ldots, k_d) \in \mathbb{N}^d$ and $\boldsymbol{k}! := k_1! \ldots k_d!$. Furthermore, $\boldsymbol{x}^{\boldsymbol{k}} := x_1^{k_1} \ldots x_d^{k_d}$. More details can be found, for example, in [Lan12].

We can write $f(\boldsymbol{z};\boldsymbol{p})$ as

$$\begin{aligned} f(\boldsymbol{z};\boldsymbol{p}) &= f(\boldsymbol{z};\boldsymbol{p}_0) + \psi_{\text{NTK}}(\boldsymbol{z};\boldsymbol{p}_0)^T(\boldsymbol{p} - \boldsymbol{p}_0) + \mathcal{O}\left(\|\boldsymbol{p} - \boldsymbol{p}_0\|^2\right) \\ &\overset{\boldsymbol{p} \approx \boldsymbol{p}_0}{\approx} f(\boldsymbol{z};\boldsymbol{p}_0) + \psi_{\text{NTK}}(\boldsymbol{z};\boldsymbol{p}_0)^T(\boldsymbol{p} - \boldsymbol{p}_0) =: f_{\text{lin}}(\boldsymbol{z};\boldsymbol{p}). \end{aligned} \tag{8.5}$$

If $\boldsymbol{p}$ does not vary too much from the initialization $\boldsymbol{p}_0$ of the SGD process, the second order term in $\|\boldsymbol{p} - \boldsymbol{p}_0\|$ can be neglected and $f_{\text{lin}}$ becomes a decent approximation of $f$. Moreover, $f_{\text{lin}}$ is

completely linear in $\boldsymbol{p}$, which means that taking gradient descent steps for the optimization of $f_{\text{lin}}$ with respect to $\boldsymbol{p}$ is equivalent to following the negative gradient direction $-\psi_{\text{NTK}}(\boldsymbol{z}; \boldsymbol{p}_0)$, which only depends on the initial parameters $\boldsymbol{p}_0$. Furthermore, to achieve global convergence of the SGD procedure, we would only have to ensure that $K_{\text{NTK}}(\cdot, \cdot; \boldsymbol{p}_0)$ is a positive definite kernel function because the NTK of $f_{\text{lin}}$ does not change over time.

The important observation, which enables us to work with (8.5), is that $\boldsymbol{p}$ changes less and less between iterations when we grow the size $M := d_2 = d_3 = \ldots, d_L$ of each hidden layer, i.e., when we increase the number of neurons in the hidden layers. More formally, for the least squares loss function and ReLU activations, it holds that the NTK converges (in probability) to a kernel function, which only depends on the initial weights and biases $\boldsymbol{p}_0$ and does not change during training via SGD when $M \to \infty$; see [JGH18]. Furthermore, if the limit kernel function is positive definite, we have

$$\sup_t |f(\boldsymbol{z}; \boldsymbol{p}(t)) - f_{\text{lin}}(\boldsymbol{z}; \boldsymbol{p}(t))| = \mathcal{O}\left(M^{-\frac{1}{2}}\right)$$

for all $\boldsymbol{z}$ with probability arbitrarily close to one; see [LXS$^+$19] for more details.

In this way, we obtain an NTK, which is constant during the training process. Additionally, if it is positive definite it is guaranteed that we find a global optimum with SGD.[36] If we consider only input vectors with norm one, for instance, it can indeed be proven that the corresponding infinite-width NTK is positive definite; see [JGH18]. However, we also see that this result only holds in the limit case of infinitely large layer sizes, i.e., infinitely many neurons in each hidden layer, which of course does not resemble a real neural network. Nevertheless there exist attempts to achieve similar results for finite networks; see [AZLS19]. First approximation-theoretic results on neural tangent kernel spaces can be found in [EW21].

To conclude this section, we present a plot of several NTKs for a toy example in Figure 8.1. To this end, we trained a three-layer fully connected network with varying sizes $M$ of the intermediate layers on 100 samples of the function $\tilde{f} : \mathbb{S}^2 \to \mathbb{R}$ defined on the unit circle as

$$\tilde{f}(z_1, z_2) := |\theta| \sin(\theta) \cos(2\theta) \quad \text{with} \quad \theta := \operatorname{atan2}(z_2, z_1). \tag{8.6}$$

Note that $\operatorname{atan2}(z_2, z_1)$ defines the angle between $(z_1, z_2)$ and the $x_1$-axis. As we see in Figure 8.1, the kernels for a fixed $M$ vary slightly depending on the weight initialization. However, when $M$ becomes larger, the variation of the NTK values becomes smaller. Furthermore, the NTK after 200 epochs of training is deviating a lot from the corresponding NTK after initialization for small $M$, but both are very close to each other for larger $M$. This resembles the fact that the NTK is constant during the training process when the number of intermediate layer neurons goes to infinity.

## 8.2 ▪ Residual neural networks and differential equations

While we interpreted the iteration process of SGD as a discretized form of a specific ODE in Section 8.1, we now change our perspective and draw a connection between ODEs and the layer-wise forward propagation of deep neural networks. To this end, we consider a special class of neural networks, namely so-called *residual networks* (ResNets). ResNets were designed to make the training process of deep neural networks more stable by introducing a residual-type forward propagation step. They have been successfully employed for machine learning tasks on image

[36] One could think that training becomes meaningless since the individual entries of $\boldsymbol{p}$ change less and less during SGD iterations when $M \to \infty$. However, the collective changes influence the network function $f$ just enough such that training remains meaningful; see [LXS$^+$19].

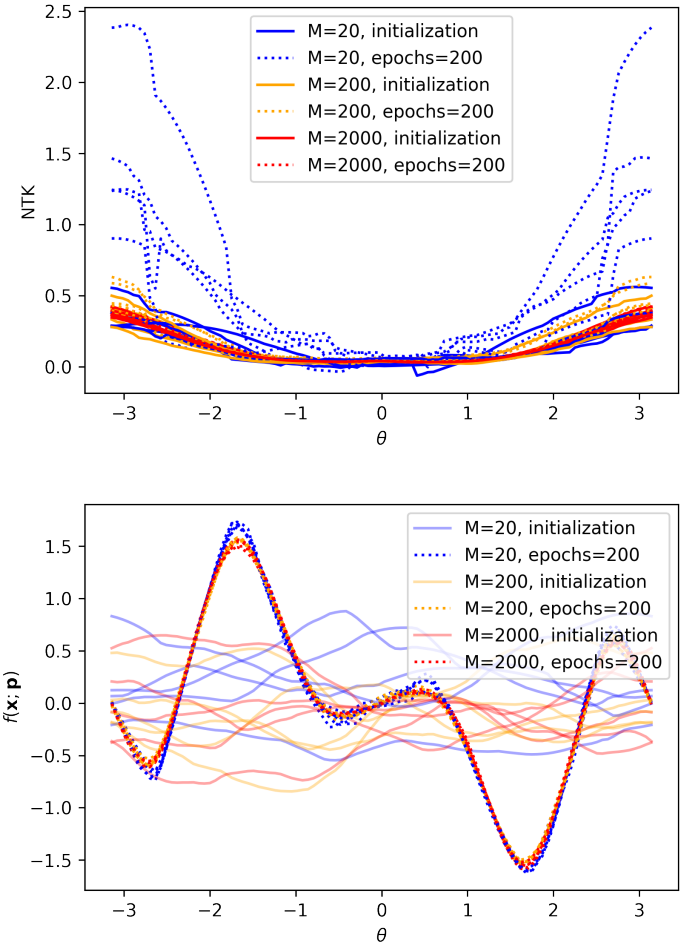



Figure 8.1: Top: Neural Tangent Kernel (NTK) for a three-layer network with 2 input neurons, $M$ neurons at both intermediate layers and 1 output neuron. We omit biases for simplicity. The weights are each initialized randomly according to $\mathcal{N}(0,1)$. We depict the resulting NTK for different intermediate layer sizes $M$ directly after the random initialization and after 200 epochs of training with 100 samples of the function (8.6). For each $M$ we ran 5 experiments with different random initializations. We depict the value of the corresponding kernel $K_{\text{NTK}}(\boldsymbol{z}, \boldsymbol{y}; \boldsymbol{p})$ for a fixed value of $\boldsymbol{y} \in \mathbb{R}^2$ and in dependence of $\theta = \operatorname{atan2}(z_2, z_1)$. Bottom: The corresponding network functions after initialization and after 200 epochs of training.

data, such as classification [HZRS16], semantic segmentation [CPK+18], and object detection [RHGS15], for instance. One step of a ResNet forward propagation can be written in the form

$$\vec{o}^{(l+1)} = \vec{o}^{(l)} + \delta t \cdot \phi^{(l+1)} \left( \left( \mathbf{W}^{(l)} \right)^T \cdot \vec{o}^{(l)} + \vec{b}^{(l+1)} \right), \tag{8.7}$$

where $\delta t > 0$ is some positive scaling parameter. Note that this implies that $\vec{o}^{(l)} \in \mathbb{R}^d$ is of the same size $d$ for each layer $0 \leq l \leq L$. To end up with only one output neuron, e.g., for real-valued regression, we would still need to add a final layer reducing the output size to one. If we assume that the activation function $\phi^{(l+1)} = \phi$ is the same in each iteration step $l = 0, \ldots, L$,

the above equation can be written as

$$\frac{\vec{o}^{(l+1)} - \vec{o}^{(l)}}{\delta t} = \phi\left(\left(\mathbf{W}^{(l)}\right)^T \cdot \vec{o}^{(l)} + \vec{b}^{(l+1)}\right). \tag{8.8}$$

Therefore, we can reinterpret this as a time-explicit Euler discretization with step width $\delta t$ for the ordinary differential equation

$$\dot{\vec{o}}(t) = \phi\left(\mathbf{W}^T(t)\vec{o}(t) + \vec{b}(t)\right). \tag{8.9}$$

This ODE now describes the forward propagation process of the neural network model. This is in contrast to (8.3), which resembles the iterative optimizer used to minimize the loss function. Note that we employed $\nu$ as a step size in (8.2) since we also used this letter to denote the learning rate in SGD; see Algorithm 10. Now, as (8.8) is no longer related to SGD, we use $\delta t$ in (8.8) instead.

### 8.2.1 ▪ Stability of feed-forward ResNets

A stable forward propagation (8.7) of a ResNet can only be guaranteed if the ODE itself admits stable solutions. From ODE theory (see [WNH93]) it is well known that this holds if the real parts of the eigenvalues of the Jacobian $\boldsymbol{J}$ of the right-hand side of (8.9) are non-positive. Furthermore, it is necessary that the explicit Euler scheme is stable, which holds if

$$|1 + \delta t \cdot \lambda_i(\boldsymbol{J}^{(l)})| \leq 1 \quad \forall\, l = 1, \ldots, L-1$$

is fulfilled for all eigenvalues $\lambda_i(\boldsymbol{J}^{(l)})$ of the $l$th layer Jacobian $\boldsymbol{J}^{(l)}$ of the right-hand side of (8.8).

This reinterpretation motivates the creation of new forms of neural networks that allow for stable evaluations, e.g., networks with anti-symmetric weight matrices or *Hamiltonian-based networks*. For the latter, the variable $\vec{o}(t)$ from (8.9) is split into two vectors $\vec{y}(t)$ and $\vec{z}(t)$ (e.g., of equal length), which are the solutions to the coupled Hamiltonian ODE system

$$\dot{\vec{y}}(t) = \phi\left(\mathbf{W}(t)\vec{z}(t) + \vec{b}(t)\right) \quad \text{and} \quad \dot{\vec{z}}(t) = -\phi\left(\mathbf{W}(t)^T\vec{y}(t) + \vec{b}(t)\right).$$

Because of its structure, this system is automatically stable and, thus, also allows for stable forward propagation if appropriate discretization schemes are chosen. An example presented in [HR18] is the symplectic so-called *Verlet* integration scheme (see also [WNH93]), which alternates between updating the $y$ and $z$ values, i.e.,

$$\begin{aligned} \vec{z}^{(l+\frac{1}{2})} &= \vec{z}^{(l-\frac{1}{2})} - \delta t \cdot \phi\left(\left(\mathbf{W}^{(l)}\right)^T \vec{y}^{(l)} + \vec{b}^{(l+1)}\right) \\ \text{and} \quad \vec{y}^{(l+1)} &= \vec{y}^{(l)} + \delta t \cdot \phi\left(\mathbf{W}^{(l)}\vec{z}^{(l+\frac{1}{2})} + \vec{b}^{(l+1)}\right) \end{aligned}$$

with appropriate time step size $\delta t > 0$.

### 8.2.2 ▪ Convolutional ResNets and (integro-)differential equations

For ResNets, where the right-hand side of (8.8) is of convolutional type instead of just feed-forward type, one can show that the convolutional operator can be seen as a spatial discretization of a partial differential operator. To this end, let us change our forward propagation process from (8.8) to

$$\frac{\vec{o}^{(l+1)} - \vec{o}^{(l)}}{\delta t} = -\boldsymbol{K}^T\phi\left(\boldsymbol{K}\vec{o}^{(l)}\right), \tag{8.10}$$

where $\boldsymbol{K}$ denotes a convolutional operator matrix applied to $\vec{o}^{(l)}$. The specific structure is chosen because of its benign stability properties; see [RH20] for more details. This leads to a direct connection between convolutional ResNets and partial differential equations (PDEs). For instance, for image data, $\boldsymbol{K}$ can be chosen as a discretized version of the two-dimensional $\nabla$ operator; see [PTVF07] for possible discretizations. If $\phi = \mathrm{id}$, this results in a discretized version of the heat equation

$$\dot{\vec{o}}(t) = -\nabla^T \nabla o(t) = -\Delta o(t),$$

where the Laplace operator on the right side operates on local patches in the pixel-space. Similar to the ODE setting, [RH20] shows that, for any non-decreasing activation function $\phi$, the forward propagation of (8.10) is stable.

To allow for more general architectures than just convolutional ResNets, [BGK22] investigates networks motivated by integro-partial differential equations, where not only local but also global interactions in the pixel space are taken into account.

## 8.3 ▪ Neural ODEs

In this section we consider a more general ResNet than the one from (8.7). To this end, let us redefine the right-hand side of (8.8) to obtain

$$\frac{\vec{o}^{(l+1)} - \vec{o}^{(l)}}{\delta t} = F\left(\vec{o}^{(l)}, \boldsymbol{W}^{(l)}, \vec{b}^{(l+1)}\right),$$

with the more general right-hand side $F$, which is a function of the layer outputs and parameters. This corresponds to the ODE

$$\dot{\vec{o}}(t) = F\left(\vec{o}(t), \boldsymbol{W}(t), \vec{b}(t)\right) \tag{8.11}$$

when the time step size $\delta t$ goes to $0$.

In Section 8.2 we used the connection between the forward propagation step and the corresponding ODE to derive results on the model's stability. Now we also consider solving the ODE (8.11) numerically with some initial condition $\vec{o}(0) = \boldsymbol{x}$. This way, we directly obtain $\vec{o}(L \cdot \delta t)$ for some fixed $\delta t$, which mimics the result of the corresponding $L$-layer neural network with input $\boldsymbol{x}$. This is known as solving the *Neural ODE*; see [CRBD18]. A major benefit of this approach is that the ODE solver can, e.g., choose to adapt the step size if necessary, which would not automatically be possible when training the network via SGD.

Now, let $T$ be the point in time at which we want to evaluate the ODE solution $\vec{o}(T)$ and let $\mathcal{S}$ be the outcome of some arbitrary numerical initial value problem solver depending on $T$, $\vec{o}(0)$, $F$, $\boldsymbol{W}(t)$, and $\vec{b}(t)$. Note that we assume that $\boldsymbol{W}(t)$ and $\vec{b}(t)$ are constant over time to be in accordance with the Neural ODE setting in [CRBD18]. Then, the problem of optimizing the weights and biases of the network becomes

$$\min_{\boldsymbol{W}, \vec{b}} \hat{C}(\vec{o}(T)) = \min_{\boldsymbol{W}, \vec{b}} \hat{C}\left(\mathcal{S}(T, \vec{o}(0), F, \boldsymbol{W}, \vec{b})\right),$$

where $\hat{C} : \mathbb{R}^d \to [0, \infty)$ is a loss function mapping the $d$-dimensional output to a positive, real number. To solve this optimization problem one needs to be able to compute gradients of the ODE solver with respect to the weights and biases. This can be done by using the so-called *adjoint sensitivity method* (see [CRBD18] for details), which just consists of solving a so-called *adjoint ODE* backwards in time. To this end, we define the adjoint

$$\vec{a}(t) := \frac{\partial \hat{C}(\vec{o}(T))}{\partial \vec{o}(t)}$$

as the derivative of the loss function $\hat{C}$ at the final result $\vec{o}(T)$ with respect to the intermediate outcome $\vec{o}(t)$ for some $0 \leq t \leq T$. It can then be shown that the derivatives of the loss function with respect to the weights can be obtained by computing the integral

$$\frac{\partial \hat{C}(\vec{o}(T))}{\partial \boldsymbol{W}} = \int_0^T \vec{a}(t)^T \frac{\partial F\left(\vec{o}(t), \boldsymbol{W}, \vec{b}\right)}{\partial \boldsymbol{W}} \mathrm{d}t; \tag{8.12}$$

see [CRBD18]. The formula for the bias derivative works analogously. To calculate the adjoint itself, we can solve the ODE

$$\dot{\vec{a}}(t) = -\vec{a}(t)^T \frac{\partial F\left(\vec{o}(t), \boldsymbol{W}, \vec{b}\right)}{\partial \vec{o}(t)} \tag{8.13}$$

with a given final value $\vec{a}(T)$. In this way, we obtain the adjoint at any time $0 \leq t \leq T$. For example, for the training data set $(\boldsymbol{x}_i, y_i) \in \mathbb{R}^d \times \mathbb{R}$ and for a vector-valued least squares loss

$$\hat{C}(\vec{o}(T)) = \frac{1}{dn} \sum_{i=1}^{n} \|\vec{o}(T)|i - y_i\|^2,$$

where $\vec{o}(T)|i$ denotes $\vec{o}(T)$ for the initial condition $\vec{o}(0) = \boldsymbol{x}_i$, the final value would be

$$\vec{a}(T) = \frac{2}{dn} \sum_{i=1}^{n} \sum_{j=1}^{d} \left( [\vec{o}(T)|i - y_i]_j \right).$$

Here, $[\cdot]_j$ denotes the $j$th component of a vector.

Combining (8.12) and (8.13), we have a direct way to take steps in the negative gradient direction of the loss function with respect to the weights and biases. Solving these equations can be done by a single call to an ODE solver; see [CRBD18] for more details. In this way, we substitute the backpropagation step for ResNet optimization with SGD by a call to an ODE solver where (8.11) is driving the dynamics.

One benefit of following the Neural ODE approach instead of doing regular backpropagation is that the output at intermediate layers does not need to be stored for Neural ODEs as it can be automatically inferred by the ODE solver when needed. Hence, the storage requirements for applying Neural ODEs are significantly reduced in contrast to using backpropagation for networks with many layers; see also Section 6.3. Furthermore, since standard ODE solvers are able to give a solution with a pre-defined accuracy in the fastest way possible, we benefit from an efficient balance of computational time and numerical error when running a Neural ODE algorithm. Because of their underlying continuous-time model, Neural ODEs are also able to deal with time series data that has been sampled in irregular intervals. This is usually a problem for standard recurrent neural networks; see Section 10.3. For more details on these topics, we refer the interested reader to [CRBD18, Kid22, RBZ23].

## 8.4 ▪ Generative diffusion models

In this section, we consider a class of generative models based on *diffusion processes*, i.e., specific *stochastic processes*, whose underlying dynamics resemble a diffusion equation.

### 8.4.1 ▪ Forward diffusion processes

Generative diffusion models aim to draw samples according to the distribution $\mu$ from which an unlabeled training set $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n \sim \mu$ has been drawn. To this end, a link between a known

distribution, i.e., the standard normal distribution in our case, and $\mu$ is established. This can be seen in correspondence to VAEs, where the latent space samples were assumed to follow a Gaussian distribution and were mapped to the original data space by the decoder; see Chapter 7.

**Generative diffusion processes**

A time-discrete ***forward diffusion process*** starting at a data point $\boldsymbol{x}^{(0)} \in \mathbb{R}^d$, which has been drawn according to some unknown data distribution $\mu$, is defined by

$$\boldsymbol{x}^{(i)} = \sqrt{1-\beta_i}\boldsymbol{x}^{(i-1)} + \sqrt{\beta_i}\varepsilon \tag{8.14}$$

for each $i \in \mathbb{N}$. Here, $\boldsymbol{x}^{(i)}$ denotes the $i$th iterate, $0 < \beta_i < 1$ is called the $i$th *noise level*, and $\varepsilon \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I})$ is a standard normally distributed random variable in $\mathbb{R}^d$. The original data point $\boldsymbol{x}^{(0)}$ is perturbed by Gaussian noise in each iteration step. In the limit $i \to \infty$, the iterates $\boldsymbol{x}^{(i)}$ become completely random, normally distributed vectors.
The main idea of ***generative diffusion processes*** is to invert the above forward diffusion process step by step to obtain data points distributed according to $\mu$ when starting at a random Gaussian vector following the distribution $\mathcal{N}(\mathbf{0}, \boldsymbol{I})$. Details can be found in [HJA20, Mur23, SDWMG15].

Before we discuss how to build a generative diffusion process, let us first consider the connection between diffusion processes and *stochastic differential equations*.

**Stochastic differential equations**

In contrast to ODEs, ***stochastic differential equations*** (SDEs) contain an additional term in the form of a stochastic process, i.e., a family of random variables.[37] In particular, a typical SDE can be written as

$$\mathrm{d}X(t) = \mu(X(t), t)\mathrm{d}t + \sigma(X(t), t)\mathrm{d}W(t)$$

in the so-called *Itô* notation, where the *drift* coefficient $\mu : \mathbb{R}^d \to [0, \infty) \to \mathbb{R}^d$ is a vector-valued function and the *diffusion* coefficient $\sigma : \mathbb{R}^d \times [0, \infty) \to \mathbb{R}$ is a real-valued function. Furthermore, $X, W : [0, \infty) \to \mathbb{R}^d$ are stochastic processes. The above equation has to be interpreted as

$$\frac{\mathrm{d}X(t)}{\mathrm{d}t} = \mu(X(t), t) + \sigma(X(t), t)\frac{\mathrm{d}W(t)}{\mathrm{d}t}. \tag{8.15}$$

For more details on SDEs and on how to comprehend the time-derivative of $W$ in (8.15), we refer the reader to [KP92, Pro05].
Oftentimes, e.g., in mathematical finance [Hul14, Sam65] or in particle physics [Ein56, KSZ96], $W$ is modeled as multi-dimensional *standard Brownian motion*, i.e., as a stochastic process whose increments $W(t_1) - W(t_0)$ are independent and distributed according to $\mathcal{N}(\mathbf{0}, (t_1 - t_0)\boldsymbol{I})$ for $t_1 > t_0 > 0$.

[37] Another way to interpret a stochastic process is as a random function. In particular, a stochastic process evaluates to a random element in a pre-defined function space.

In the context of generative diffusion models, we consider the so-called *forward diffusion* SDE

$$\frac{\mathrm{d}X(t)}{\mathrm{d}t} = -\frac{1}{2}\beta(t)X(t) + \sqrt{\beta(t)}\frac{\mathrm{d}W(t)}{\mathrm{d}t} \tag{8.16}$$

with some *noise variance* $\beta : [0, \infty) \to \mathbb{R}$. For our purposes, we are not interested in the time-continuous equation (8.16), but in the *Euler–Maruyama* discretization [KP92] thereof, which is a stochastic variant of the time-explicit Euler scheme; see, e.g., (8.2) and (8.8). By using a time step size $\delta t > 0$, we obtain

$$\frac{X(t+\delta t) - X(t)}{\delta t} = -\frac{1}{2}\beta(t)X(t) + \frac{\sqrt{\beta(t)}}{\sqrt{\delta t}}\varepsilon, \tag{8.17}$$

where $\varepsilon \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I})$ is a standard normally distributed random variable. Note that this can be rewritten as

$$X(t+\delta t) = \left(1 - \frac{\beta(t)\delta t}{2}\right) X(t) + \sqrt{\beta(t)\delta t}\,\varepsilon.$$

Now let us consider the forward diffusion equation (8.14) with $\boldsymbol{x}^{(i)} = X(i)$ and $\beta_i = \beta(i)\delta t$, i.e.,

$$X(i+\delta t) = \sqrt{1-\beta(i)\delta t}X(i) + \sqrt{\beta(i)\delta t}\,\varepsilon. \tag{8.18}$$

Apart from the drift coefficient, i.e., the term in front of $X(t)$ or $X(i)$, respectively, both equations are the same. Finally, to obtain the connection between these equations, note that a first order Taylor expansion of $\sqrt{1-\beta(i)\delta t}$ around $\delta t = 0$ yields

$$\sqrt{1-\beta(i)\delta t} = 1 - \frac{\beta(i)\delta t}{2} + \mathcal{O}\left((\delta t)^2\right).$$

Thus, the Euler–Maruyama discretization (8.17) of the forward diffusion SDE (8.16) resembles the forward diffusion process (8.14) up to second order in $\delta t$ for $\delta t \to 0$.

### 8.4.2 ▪ Reverse diffusion processes

To draw samples according to the data distribution $\mu$, we need to establish the *reverse diffusion process* $\bar{X}$, which (approximately) transforms $\boldsymbol{x}^{(T)}$ computed according to (8.14) for very large $T > 0$ back to $\boldsymbol{x}^{(0)}$. To this end, note that it can be proven that (8.16) can be reversed in time and the solution is given by

$$\frac{\mathrm{d}\bar{X}(t)}{\mathrm{d}t} = -\frac{1}{2}\beta(t)(\bar{X}(t) + g(\bar{X}(t), t)) + \sqrt{\beta(t)}\frac{\mathrm{d}\bar{W}(t)}{\mathrm{d}t} \tag{8.19}$$

for some function $g : \mathbb{R}^d \times [0, \infty) \to \mathbb{R}^d$; see [And82] for details. Here, $\mathrm{d}t$ has to be understood as an infinitesimal step backwards in time and $\bar{W}$ is a time-reversed Brownian motion. Applying an Euler–Maruyama discretization with time step size $\delta t$ to (8.19), we obtain

$$\bar{X}(t-\delta t) = \bar{X}(t) - \frac{1}{2}\beta(t)\delta t(\bar{X}(t) + g(\bar{X}(t), t)) + \sqrt{\beta(t)\delta t}\,\varepsilon \tag{8.20}$$

for an $\varepsilon \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I})$. This shows that for both the discretized forward diffusion SDE and the discretized reversed diffusion SDE, we only need to compute terms depending on the current iterate and a small Gaussian increment to obtain the next iterate. In fact, this also carries over to the reverse diffusion process. In particular, since the distribution of $\boldsymbol{x}^{(i)}$ given $\boldsymbol{x}^{(i-1)}$ can be written as

$$p^*\left(\boldsymbol{x}^{(i)} \mid \boldsymbol{x}^{(i-1)}\right) \sim \mathcal{N}(\sqrt{1-\beta_i}\boldsymbol{x}^{(i-1)}, \beta_i \boldsymbol{I}), \tag{8.21}$$

one can show that the (true) distribution of $\boldsymbol{x}^{(i-1)}$ given $\boldsymbol{x}^{(i)}$ and $\boldsymbol{x}^{(0)}$ can be written as

$$p^*\left(\boldsymbol{x}^{(i-1)} \mid \boldsymbol{x}^{(i)}, \boldsymbol{x}^{(0)}\right) \sim \mathcal{N}(\bar{\boldsymbol{m}}_i, \bar{\sigma}_i \boldsymbol{I}),$$

where both mean $\bar{\boldsymbol{m}}_i$ and noise level $\bar{\sigma}_i$ depend only on $\boldsymbol{x}^{(i)}$, $\boldsymbol{x}^{(0)}$, and $\beta_j$ for $1 \leq j \leq i$. For more details and explicit formulas for $\bar{\boldsymbol{m}}_i$ and $\bar{\sigma}_i$, we refer the reader to [HJA20, Mur23].[38] Since $\boldsymbol{x}^{(0)}$ is not known when reversing the forward diffusion process to infer $\boldsymbol{x}^{(i-1)}$ from $\boldsymbol{x}^{(i)}$, we approximate the distribution $p^*\left(\boldsymbol{x}^{(i-1)} \mid \boldsymbol{x}^{(i)}, \boldsymbol{x}^{(0)}\right)$ by

$$p\left(\boldsymbol{x}^{(i-1)} \mid \boldsymbol{x}^{(i)}\right) \sim \mathcal{N}(\boldsymbol{m}_i, \sigma_i \boldsymbol{I}) \tag{8.22}$$

with degrees of freedom $\boldsymbol{m}_i$ and $\sigma_i$ not depending on $\boldsymbol{x}^{(0)}$. Oftentimes, the variance coefficients are fixed a priori, e.g., $\sigma_i = \beta_i$; see also [HJA20]. This means we are left with the task of determining $m_i$ from given data $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$. To this end, we create a neural network taking $\boldsymbol{x}^{(i)}$ as input and computing $\boldsymbol{m}_i$ as output. To train the network, we could consider minimizing the cross entropy

$$-\mathbb{E}_{p^*(\boldsymbol{x}^{(0)})}\left[p(\boldsymbol{x}^{(0)})\right] \tag{8.23}$$

over the given training data, i.e., the expected value $\mathbb{E}_{p^*(\boldsymbol{x}^{(0)})}$ would become an average over the $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$. However, since $p(\boldsymbol{x}^{(0)})$ is not directly accessible, we follow the same idea as in Section 7.4.2 and (7.5), where we discussed the ELBo loss for variational autoencoders. Thus, instead of minimizing (8.23) to obtain $\boldsymbol{m}_1, \ldots, \boldsymbol{m}_T$, we minimize the ELBo

$$\mathbb{E}_{p^*}\left[\log\left(p^*\left(\boldsymbol{x}^{(T)}\right)\right) + \sum_{i=1}^{T} \log\left(\frac{p\left(\boldsymbol{x}^{(i-1)} \mid \boldsymbol{x}^{(i)}\right)}{p^*\left(\boldsymbol{x}^{(i)} \mid \boldsymbol{x}^{(i-1)}\right)}\right)\right], \tag{8.24}$$

where $T > 0$ is the number of steps of the forward diffusion process. For details on the training procedure, see Section 8.4.3 and [HJA20, Mur23, SDWMG15].

Finally, we observe that, after learning $\boldsymbol{m}_i$ from (8.22), both the forward diffusion process

$$\boldsymbol{x}^{(i)} = \sqrt{1-\beta_i}\boldsymbol{x}^{(i-1)} + \sqrt{\beta_i}\varepsilon_i$$

and the reverse process

$$\boldsymbol{x}^{(i-1)} = \boldsymbol{m}_i + \sqrt{\sigma_i}\bar{\varepsilon}_i$$

with $\varepsilon_i, \bar{\varepsilon}_i \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{I})$ are computed by sampling standard normally distributed random variables. This has to be seen in analogy to the Euler–Maruyama discretizations (8.18) and (8.20) of the corresponding forward and reverse diffusion SDEs (8.16) and (8.19).

### 8.4.3 ▪ Training and generation

As mentioned in the previous section, we fix the variances $\sigma_i$ for $i = 1, \ldots, T$ in (8.22), e.g., by $\sigma_i = \beta_i$, and employ a neural network to determine $\boldsymbol{m}_i$ for given $\boldsymbol{x}^{(i)}$. To train this network, we minimize the ELBo loss function (8.24) over the training data set $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$ by an SGD-type algorithm. To this end, we need to be able to evaluate the ELBo.[39] Note that (8.24) can get approximated by

$$\frac{1}{n}\sum_{j=1}^{n}\left(\log\left(p^*\left(\boldsymbol{x}_j^{(T)}\right)\right) + \sum_{i=1}^{T}\log\left(\frac{p\left(\boldsymbol{x}_j^{(i-1)} \mid \boldsymbol{x}_j^{(i)}\right)}{p^*\left(\boldsymbol{x}_j^{(i)} \mid \boldsymbol{x}_j^{(i-1)}\right)}\right)\right), \tag{8.25}$$

[38] An even more detailed derivation can be found in the excellent blog article https://lilianweng.github.io/posts/2021-07-11-diffusion-models.

[39] Its derivatives can then be computed by automatic differentiation.

i.e., we compute an average over the given training data. To evaluate (8.25), we assume that $T$ is large enough such that $\boldsymbol{x}_j^{(T)}$ is (approximately) standard normally distributed, i.e., we assume $p^*(\boldsymbol{x}_j^{(T)}) \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{I})$ for all $j = 1, \dots, n$. Furthermore, note that $p^*(\boldsymbol{x}_j^{(i)} \mid \boldsymbol{x}_j^{(i-1)})$ from (8.21) is known, i.e., we can evaluate it. Thus, we can compute $\boldsymbol{x}_j^{(i)}$ for each $i = 1, \dots, T$ and for each training data point $\boldsymbol{x}_j, j = 1, \dots, n$, by drawing samples according to (8.21), and we thus can evaluate (8.25).

Finally, assuming that $p^*(\boldsymbol{x}^{(T)}) \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{I})$, we minimize (8.25) with respect to the weights and biases of the network with an SGD-type algorithm.

To use the already trained generative diffusion process for data generation, we simply draw a random vector $\boldsymbol{x}^{(T)} \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{I})$, compute the corresponding $\boldsymbol{m}_T$ with the trained neural network, and apply (8.22) for $i = T$ to obtain $\boldsymbol{x}^{(T-1)}$. This process of computing $\boldsymbol{m}_i$ and applying (8.22) is then repeated for $i = T - 1, \dots, 1$ to finally obtain the new data point $\boldsymbol{x}^{(0)}$; see [HJA20, Mur23].

### 8.4.4 ▪ Application to image data

Now we will have a look at generative diffusion models for image generation, i.e., starting with a given image, we generate a slightly noisier image at each time step of the forward diffusion process. Similarly, we *denoise* an image step by step following the backward diffusion process. To this end, let $\boldsymbol{A}_0 \in \mathbb{R}^{h \times w \times c}$ be an image in pixel space, where $c$ denotes the number of color channels. We assume that $\boldsymbol{A}_0$ is drawn according to some image distribution $\mu$ on $\mathbb{R}^{h \times w \times c}$. Now, in order to generate a new image $\boldsymbol{A}_i$ from an old image $\boldsymbol{A}_{i-1}$, we draw a random sample from the distribution

$$p^*(\boldsymbol{A}_i \mid \boldsymbol{A}_{i-1}) \sim \mathcal{N}\left(\sqrt{1-\beta_i}\boldsymbol{A}_{i-1}, \beta_i \boldsymbol{I}\right)$$

with some hyperparameters $0 < \beta_i < 1$ successively for $i = 1, \dots, T$. Oftentimes the $\beta_i$ are chosen according to a certain schedule; see [HJA20, Mur23] for details. After the minimization of (8.25), and thus after the computation of $\boldsymbol{m}_i$ for $i = 1, \dots, T$, we can denoise an image step by step by drawing a new iterate according to the distribution

$$p(\boldsymbol{A}_{i-1} \mid \boldsymbol{A}_i) \sim \mathcal{N}\left(\boldsymbol{m}_i, \sigma_i \boldsymbol{I}\right). \tag{8.26}$$

Examples for images created by a generative diffusion model can be found in Figure 8.2. Here, the resulting denoised images depend on both the initial random image drawn according to $\mathcal{N}(\boldsymbol{0}, \boldsymbol{I})$ and the random path generated by consecutively sampling according to (8.26).

## 8.5 ▪ Further topics

**Neural networks for solving PDEs** Besides using the interpretation of ResNets as discretized numerical solvers for ODEs and PDEs in order to analyze the network's properties as we have done in Section 8.2, we can also look at this relation from the opposite perspective. To this end, we can use deep neural networks in order to approximately solve given ordinary and partial differential equations. An overview on methodologies which are used for this purpose can be found in [DWW22, EHJ21, HJKN20].

**Physics-informed neural networks** Since ODEs and PDEs often appear in applications from physics, special types of networks, namely *physics-informed* neural networks (PINNs), have been developed to specifically deal with such equations. Here, network structures, loss functions, and regularization terms are adapted to meet specific physical requirements, e.g., the loss can be chosen such that it penalizes unphysical behavior in the network output; see [BE21, RPK19] for an overview on PINNs and [vRMB+23] for a survey on *informed machine learning* in general.

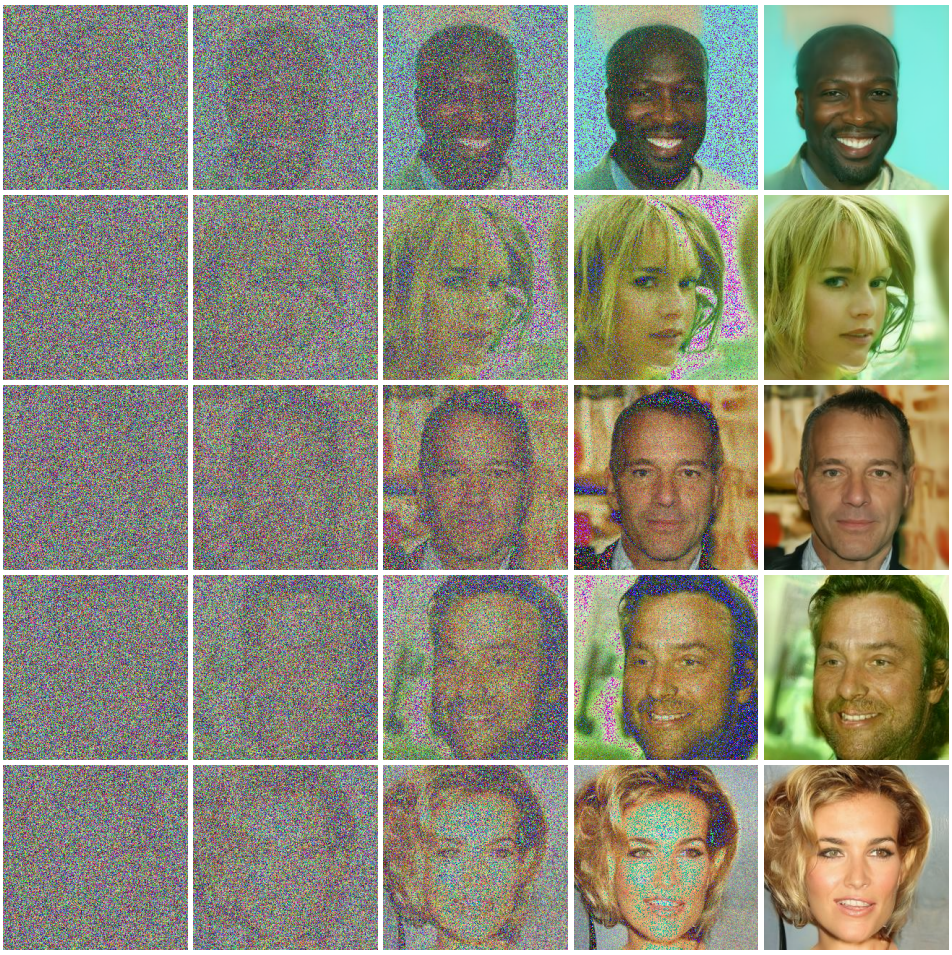


Figure 8.2: Example pictures (*deep fakes*) of fictitious people created by a generative diffusion model [HJA20] trained on the CelebA data set [LLWT15] of real celebrities. Each row contains (from left to right) the iterates $\boldsymbol{A}_i$ for $i = 16, 12, 8, 4, 0$ of the reverse diffusion process (8.26) for different random instances of $\boldsymbol{A}_T \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{I})$ with $T = 50$. For the corresponding code to generate the pictures, see https://huggingface.co/google/ddpm-celebahq-256. Created from the CelebA data set.

**Deep operator networks** When we are dealing with parametrized ODEs or PDEs for which many solutions have to be computed, e.g., in *uncertainty quantification*, it is computationally beneficial to directly approximate the solution operator corresponding to the parametrized equation instead of its results for a large discrete set of parameter values. Here, neural networks that learn nonlinear, continuous operators instead of functions have been developed, like so-called *DeepONets* or *Fourier neural operators* (FNOs). We refer the reader to Section 10.1 and [KLL$^+$23, KLM21, LMK22, LKA$^+$20, LJP$^+$21] for more details.

**SGD variants and ODEs** Besides analyzing the performance of SGD via the neural tangent kernel, which we did in Section 8.1, there also exist approaches studying properties of Nesterov's SGD variant (compare Section 6.5) and other momentum-based optimizers with the help of ODE systems and gradient flows; see [MJ21, SBC16, WWJ16].

# Chapter 9
# Reinforcement Learning

Besides supervised and unsupervised learning, *reinforcement learning* is the third main category in modern artificial intelligence. It is influenced by ideas of behaviorism that attribute learning to a feedback of positive and negative reinforcement.

**Reinforcement learning**

The goal in ***reinforcement learning*** (RL) is to make sequential decisions in a given environment, where a decision stems from a fixed set of *actions*. To this end, an *agent* is considered that acts within the environment, i.e., based on a *policy*, the agent decides which action to take in which *state* of the environment. A path of an agent in the *state space*, i.e., a sequence of states resulting from the taken actions, is called a *trajectory*. An agent is trained by using a quantitative feedback of how good (or bad) a decision or a sequence of decisions has been. To this end, the agent is guided in its decisions through *rewards*.
An important aspect of reinforcement learning is that the agent can only learn by applying actions and by observing rewards. Thus, no or only incomplete details about the environment are exposed to the agent. This a key difference from the domain of *optimal control* of dynamical systems [BCD97, Ber19, GP17, Pow22, Rec19, SB18, Vin00].
In general, an agent's decision that may lead to a high reward later, i.e., after following a trajectory of states, might yield no or even a negative immediate reward at the time of making it. Therefore, actions are not based on the immediate reward, but on their estimated long-term *value* over the whole trajectory, i.e., on their aggregated reward. To summarize, reinforcement learning is concerned with the interaction of an agent and its environment under uncertainty, in particular due to incomplete and stochastic information.

Reinforcement learning is successfully applied in many application areas. For instance, [Mea15, Sea18, Tes95] investigate the performance of RL algorithms in games. Furthermore, a deep RL approach for magnetic control of tokamak plasmas can be found in [Dea22]. For RL-based recommender systems, we refer the interested reader to [ACF22, PT13]. A reinforcement learning method for constructing fast matrix multiplication algorithms can be found in [Fea22]. [DBK+16] presents a reinforcement learning approach to financial signal representation and

trading. Surveys on reinforcement learning algorithms applied to tasks in robotics can be found in [KBP13, PN17].

This chapter is structured as follows: As a first step towards RL, we consider deterministic optimal control problems in Section 9.1. Here, we are given complete information about the environment, which is assumed to be deterministic. We introduce state and action spaces, deterministic state dynamics, and a reward function. The goal is to find an optimal policy, which determines the best—in terms of the value, i.e., the aggregated rewards—action to take when the system is in a particular state. To this end, we first introduce the *policy evaluation* algorithm, which computes the value for a given policy. Then, to actually determine an optimal policy, we consider the *value iteration* algorithm. Furthermore, we provide the *policy iteration* algorithm, which computes an optimal policy by alternating the optimization of a policy and its evaluation. Subsequently, we generalize the setting by allowing probabilistic state transitions, i.e., the state dynamics will be modeled by a probability distribution instead of a fixed function. Section 9.2 is then dedicated to the situation where only incomplete information about the environment is available for reinforcement learning. Here, information cannot be directly accessed but is only observed by following trajectories of state-action pairs. In such settings, an algorithm such as *Monte Carlo policy evaluation*, *SARSA*, or *Q-Learning* needs to be employed to compute values for trajectories and approximate optimal policies. We conclude this chapter with a discussion of *deep reinforcement learning* in Section 9.3, where deep neural networks are used to approximately determine optimal solutions. These function approximations are employed in case of large state spaces when the previously discussed methods can no longer be directly applied due to their huge costs.

## 9.1 ▪ Optimal control

Instead of starting in an environment with uncertainties or incomplete information and to simplify the introduction of essential concepts and algorithms, we first consider the easier deterministic case with complete information. This is also known as sequential deterministic optimal control.

### 9.1.1 ▪ Deterministic optimal control

We begin by describing the specific setting that we encounter in deterministic optimal control. To this end, let us briefly recapitulate the concept of deterministic Markov decision processes, which define our environment, i.e., the world in which an agent lives and acts. In particular, our environment is defined by a set of states, i.e., the states in which the environment can be, a set of actions, i.e., the actions that can be taken in a given state within the environment, and a function that describes state transitions, i.e., a function that outputs the state into which the environment changes when taking an action in a specific state.

**Deterministic Markov decision process**

A tuple $(\mathcal{S}, A, T, R, \gamma)$ is a ***deterministic, discounted Markov decision process*** (MDP) if

- $\mathcal{S}$ is a finite nonempty set, called the state space,
- $A = \cup_{s\in\mathcal{S}} A_s$ is a union of finite, nonempty sets $A_s$, where $A_s$ is a given set of actions that could be taken in state $s$,
- $T : \{(s, a) \mid s \in \mathcal{S},\ a \in A_s\} \to \mathcal{S}$ is a state-valued function,

- $R : \{(s, a) \mid s \in \mathcal{S},\ a \in A_s\} \to \mathbb{R}$ is a real-valued function, and
- $\gamma \in (0, 1]$ is a *discount factor*.

The set $A$ is the *action space*, and $A_s$ is the *set of admissible actions in state* $s$. The evaluation $T(s, a)$ of the *state dynamics* $T$ defines the next state when taking action $a \in A_s$ in state $s \in \mathcal{S}$, and $R(s, a)$ is the reward when taking the action $a \in A_s$ in state $s \in \mathcal{S}$. Note that a deterministic, discounted MDP can also be represented as a graph; see Figure 9.1. A function

$$\pi : \mathcal{S} \to A,$$

such that $\pi(s) \in A_s$, is a *policy* of an *agent*. Oftentimes, $\pi$ is also referred to as the *agent* itself.

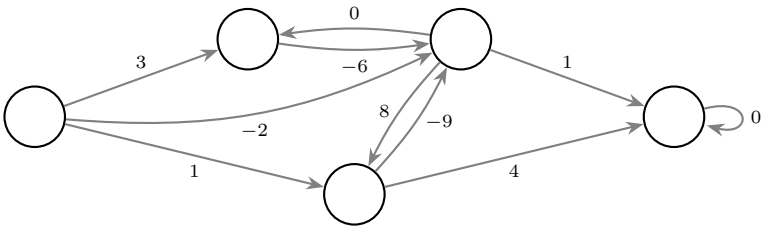



Figure 9.1: An example for the graph representation of a deterministic MDP environment. Here, every state $s \in \mathcal{S}$ is represented by a node and every action $a \in A_s$ is represented by an edge pointing from $s$ to another node. The label of an edge $a$ originating in $s$ is the reward $R(s, a) \in \mathbb{R}$ for taking action $a$ in state $s$.

Given a deterministic, discounted MDP as the environment, an agent can in any state $s \in \mathcal{S}$ interact with it by choosing an action $a \in A_s$. To be more precise, given $s_t \in \mathcal{S}$ and $a_t \in A_s$ for some time index $t \in \mathbb{N}$, the environment experiences a transition from the state $s_t$ to a state $s_{t+1} \in \mathcal{S}$, which is determined by the state dynamics map $T$, i.e., $s_{t+1} = T(s_t, a_t)$. Consequently, the agent obtains a reward $r_{t+1} \in \mathbb{R}$, which is defined by $R$, i.e., $r_{t+1} = R(s_t, a_t)$. The trivial situation of having one fixed action space $A = A_s$ for all $s \in \mathcal{S}$ represents the case of state-independent actions. The continuing interaction of the agent with the environment is illustrated in Figure 9.2. In summary, the agent encounters a sequence of events, represented by a trajectory of states $s_t \in \mathcal{S}$, the taken actions $a_t \in A_{s_t}$, the resulting next states $s_{t+1} = T(s_t, a_t) \in \mathcal{S}$, and the rewards $r_{t+1} = R(s_t, a_t) \in \mathbb{R}$ for $t \in \mathbb{N}$. The goal is now to find a policy $\pi^* : \mathcal{S} \to A$ that gives the best action to take in a state in terms of the aggregated rewards. The question is then how such a policy can be determined. This will be explained in more detail in the following.

First, to be able to compare the quality of different policies at all, we define the *discounted cumulative reward* for a given policy $\pi : \mathcal{S} \to A$ and a starting state $s_0 \in \mathcal{S}$ as

$$V^\pi(s_0) \coloneqq \sum_{t=0}^{\infty} \gamma^t R(s_t, \pi(s_t)). \tag{9.1}$$

$V^\pi$ is also called the *value function* for the policy $\pi$. In (9.1), $s_0, s_1, s_2, \ldots$ denotes the trajectory which we obtain when we follow $\pi$ for the initial state $s_0$, i.e., $s_{t+1} = T(s_t, \pi(s_t))$ for all $t \in \mathbb{N}$.

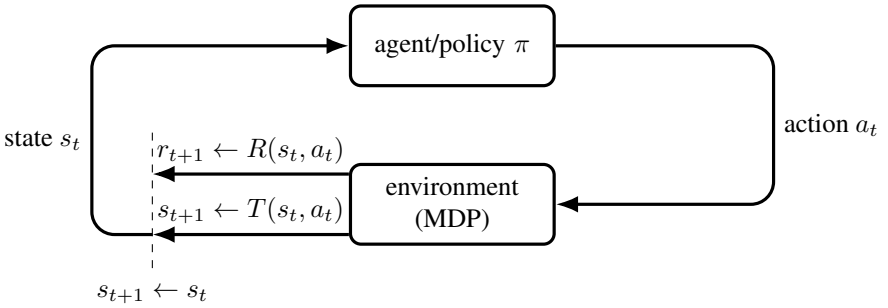



Figure 9.2: The repeated interaction between agent and environment. After each action the new state and the corresponding reward are observed.

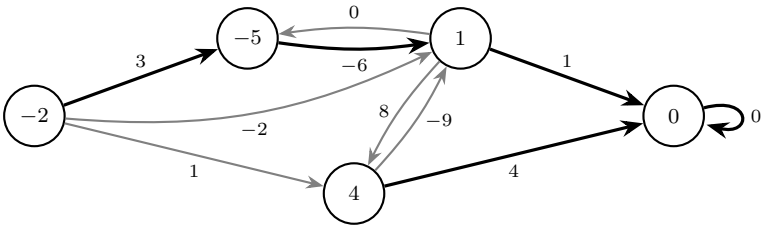



Figure 9.3: A policy for the example from Figure 9.1. Here, we visualize a specific policy $\pi : S \to A$ by bold edges. For this policy, the value of $V^\pi$ (for $\gamma = 1$) is denoted by the numbers inside the nodes. This means that the value in node $s$ is $V^\pi(s)$. Note that the rightmost node in the graph represents a terminal state, i.e., actions do not have any effect.

Here, the *discount factor* $\gamma \in (0, 1]$ models the effect of delayed rewards. We refer the reader to Figure 9.3 for an abstract example of such a sequential, deterministic Markov decision process with a value function $V^\pi$ for a chosen policy $\pi$.

**The optimization problem** A policy $\pi$ is called *optimal* if it maximizes the discounted cumulative reward $V^\pi$. Thus, the task of optimal control can now be stated as the following optimization problem: Find an optimal policy $\pi^* : \mathcal{S} \to A$ from the set $\{\pi : \mathcal{S} \to A\}$ of all policies such that

$$\pi^*(s) \coloneqq \arg\max_\pi V^\pi(s) \qquad \forall\, s \in \mathcal{S}. \tag{9.2}$$

The cumulative reward for the optimal policy $\pi^*$ is called the *optimal value function* and is denoted[40] by $V^*(s) \coloneqq V^{\pi^*}(s)$.

For $\gamma = 1$ we speak of an *undiscounted* optimal control problem; otherwise, for $0 < \gamma < 1$, the problem is called *discounted*. Note that, for an undiscounted problem, neither the existence nor the uniqueness of a solution of (9.2) can be guaranteed. However, as we will see later on, a solution exists for any discounted problem since the value function $V^\pi$ is bounded for $\gamma < 1$ and any bounded reward function $R$. Note furthermore that the sum in (9.1) is sometimes truncated

[40]Note that while the optimal value function can be shown to be unique, optimal policies (or actions) are not necessarily unique. In this chapter, $\arg\max$ is therefore understood as being one of the arguments that are maximizing the expression.

after a fixed number of steps $T \in \mathbb{N}$, i.e.,

$$V^\pi(s_0) := \sum_{t=0}^{T} R(s_t, \pi(s_t))$$

is used instead of (9.1). In such a case, the problem (9.2) is called a *finite horizon* problem. Otherwise, it is called an *infinite horizon* problem.

Certain states are often considered to be *terminal*. This means that actions in a terminal state no longer have any effect (e.g., when a game ends, when the agent exhausted his capital in trading, or when a robot has reached a destination in a navigation task), and one may think of those states as absorbing states. To include these states in our MDP model, we denote a state $s$ as terminal if all possible actions $a \in A_s$ taken in this state result in zero rewards and lead back to the same state $s$, i.e.,

$$T(s,a) = s, \quad R(s,a) = 0 \qquad \forall\, a \in A_s. \tag{9.3}$$

If an agent reaches a terminal state at time $T \in \mathbb{N}$, the discounted cumulative reward (9.1) is effectively finite because all summands for $t > T$ are zero. In practice, the environment provides the information on whether a state is terminal or not. For example, the rightmost state in Figure 9.3 is a terminal state since (9.3) holds.

**The Bellman equation** From the general definition of the discounted cumulative reward (9.1) we see that $V^\pi(s_0)$ can also be formulated in a recursive way. To this end, we split the sum into the first term and the remainder, which is itself again a representation of the value function, i.e.,

$$\begin{aligned} V^\pi(s_0) &= \sum_{t=0}^{\infty} \gamma^t R(s_t, \pi(s_t)) \\ &= R(s_0, \pi(s_0)) + \sum_{t=1}^{\infty} \gamma^t R(s_t, \pi(s_t)) \\ &= R(s_0, \pi(s_0)) + \gamma V^\pi(s_1) \end{aligned} \tag{9.4}$$

for some policy $\pi : \mathcal{S} \to A$. This yields the recursive formulation

$$V^\pi(s) = R(s,a) + \gamma V^\pi(T(s,a)) \tag{9.5}$$

with $a = \pi(s)$. Furthermore, the optimal value function $V^*$, i.e., the cumulative reward for the optimal policy (9.2), is given by the *Bellman equation*

$$V^*(s) = \max_{a \in A_s} \big( R(s,a) + \gamma V^*(T(s,a)) \big). \tag{9.6}$$

This way, $V^*$ can be characterized without referring to a specific policy $\pi^*$. In particular, choosing a maximizing action in each step yields an equation for the optimal policy

$$\pi^*(s) \coloneqq \arg\max_{a \in A_s} \big( R(s,a) + \gamma V^*(T(s,a)) \big) \tag{9.7}$$

in terms of the optimal value function $V^*$. The policy $\pi^*$ now maximizes $V^*(s)$ for every $s$. The idea behind the Bellman equation is *Bellman's optimality principle*.

**Bellman's optimality principle**

Suppose that $\pi^*$ is an optimal policy, i.e., it maximizes (9.2), and $s_0, s_1, s_2, \dots$ is the sequence of states when following $\pi^*$. Then, ***Bellman's optimality principle*** expresses that $\pi^*$ will also be an optimal policy if one starts in any other $s_i$ for $i > 0$. Specifically, the solution to the optimization problem starting at some state $s$ can be obtained by a combination of the optimal solutions to the following two subproblems: The problem of going from $s$ to the next state in an optimal way and the problem of obtaining the value function at the states $\mathcal{S} \setminus \{s\}$. This is also known as the *dynamic programming principle*, which states that for an optimal policy "the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decisions." For more details on Bellman's optimality principle and dynamic programming, we refer the reader to [Bel57, Ber12, Ber17].

**The Bellman operator** Let us now introduce the *Bellman operator* $\mathcal{B}_\pi$, which applies (9.5) for a policy $\pi$, i.e.,

$$\begin{aligned} \mathcal{B}_\pi \colon \big\{v \colon \mathcal{S} \to \mathbb{R}\big\} &\to \{v \colon \mathcal{S} \to \mathbb{R}\}, \\ v &\mapsto \big(s \mapsto \big(R(s,a) + \gamma v(T(s,a))\big)\big), \end{aligned} \tag{9.8}$$

with $a = \pi(s)$. Moreover, let us define the *Bellman optimality operator* $\mathcal{B}$, which applies (9.6), i.e.,

$$\begin{aligned} \mathcal{B} \colon \big\{v \colon \mathcal{S} \to \mathbb{R}\big\} &\to \{v \colon \mathcal{S} \to \mathbb{R}\}, \\ v &\mapsto \big(s \mapsto \max_{a \in A_s} \big(R(s,a) + \gamma v(T(s,a))\big)\big). \end{aligned} \tag{9.9}$$

For finite state spaces $\mathcal{S}$ and finite action spaces $A$, it is not difficult to verify that $\mathcal{B}_\pi$ and $\mathcal{B}$ are contractions for $\gamma < 1$. It thus holds that

$$\|\mathcal{B}_\pi v - \mathcal{B}_\pi w\|_\infty \leq \gamma \|v - w\|_\infty \quad \text{and} \quad \|\mathcal{B} v - \mathcal{B} w\|_\infty \leq \gamma \|v - w\|_\infty$$

for all $v, w : \mathcal{S} \to \mathbb{R}$ under natural conditions on the environment. Here, $\|\cdot\|_\infty$ denotes the $L_\infty$ norm of functions from $\mathcal{S}$ to $\mathbb{R}$. Using Banach's fixed point theorem one can now establish the following result; see, for instance, [Ber17].

**Theorem 9.1.1** (Bellman 1957)**.** *Let $v : \mathcal{S} \to \mathbb{R}$ and let $\gamma < 1$. The following hold:*

(i) *$V^\pi$ from* (9.5) *is the unique solution of the fixed point equation $\mathcal{B}_\pi v = v$.*

(ii) *$V^*$ from* (9.6) *is the unique solution of the fixed point equation $\mathcal{B} v = v$.*

(iii) *For any policy $\pi$ it holds that $\lim_{n\to\infty} \mathcal{B}_\pi^n v = V^\pi$.*

(iv) *It holds that $\lim_{n\to\infty} \mathcal{B}^n v = V^*$.*

The above theorem states that a fixed point of (9.5) or (9.6) is a value function for every $s \in \mathcal{S}$ when the equations are understood as recursive update equations.

### 9.1.2 ▪ Policy evaluation and value iteration

The idea of using fixed point iterations with the operators $\mathcal{B}_\pi$ and $\mathcal{B}$ in order to compute a value function for a given policy and the optimal value function leads to two algorithms, which are called *policy evaluation* and *value iteration*, respectively.

**Algorithm 11:** Policy evaluation

**Input:** deterministic, discounted MDP $(\mathcal{S}, A, T, R, \gamma)$, policy $\pi : \mathcal{S} \to A$, threshold $\delta > 0$.
**Output:** (approximation to) the value function $V^\pi$.

1 Initialize $V' : S \to \mathbb{R}$ arbitrarily such that $V'(s) \leftarrow 0$ for all terminal states $s \in S$.
2 **repeat**
3 $\quad V \leftarrow V'$.
4 $\quad$ **forall** $s \in S$ **do**
5 $\quad\quad V'(s) \leftarrow R(s, \pi(s)) + \gamma V(T(s, \pi(s)))$.
6 $\quad$ **end forall**
7 **until** $\max_{s \in S} |V - V'| \leq \delta$.
8 **return** $V'$.

The policy evaluation method computes (an approximation of) the value function $V^\pi$ for the policy $\pi$; see Algorithm 11. This is done by repeatedly applying the Bellman operator $\mathcal{B}_\pi$ to the current estimate of $V^\pi$.

Moreover, using a fixed point iteration with $\mathcal{B}$ instead of $\mathcal{B}_\pi$, we can compute an optimal value function. This is the so-called value iteration method; see Algorithm 12. Here, one computes (an approximation of) the optimal value function $V^*$, from which the corresponding (approximately) optimal policy $\pi^*$ is then derived.

**Algorithm 12:** Value iteration

**Input:** deterministic, discounted MDP $(\mathcal{S}, A, T, R, \gamma)$, threshold $\delta > 0$.
**Output:** (approximations to) the optimal value function $V^*$ and an optimal policy $\pi^*$.

1 Initialize $V' : \mathcal{S} \to \mathbb{R}$ arbitrarily such that $V'(s) \leftarrow 0$ for all terminal states $s \in S$.
2 **repeat**
3 $\quad V \leftarrow V'$.
4 $\quad$ **forall** $s \in S$ **do**
5 $\quad\quad V'(s) \leftarrow \max_{a \in A_s} (R(s, a) + \gamma V(T(s, a)))$.
6 $\quad$ **end forall**
7 **until** $\max_{s \in S} |V - V'| \leq \delta$.
8 $\pi \leftarrow \{s \to \arg\max_{a \in A_s} (R(s, a) + \gamma V'(T(s, a)))\}$.
9 **return** $V', \pi$.

The key difference between Algorithm 11 and Algorithm 12 is in line 5 of both algorithms: In the case of policy evaluation the action is given by the policy $\pi$, while in the case of value iteration the action is being optimized over all possible actions $A_s$ in state $s \in \mathcal{S}$. The value iteration algorithm and the policy evaluation algorithm first emerged in optimal control applications, also known as dynamic programming [FF13, Ber19, Pow22]. Variants of both will also be basic building blocks for more sophisticated reinforcement learning algorithms later on.

**Task 9.1.** First, we consider an environment which resembles a frozen lake. Here, the state space $S$ contains 16 different locations (tiles) on the lake, which are arranged in a $4 \times 4$ square grid. The agent has to travel from a given starting location to a fixed destination. To this end, the agent is allowed to move along one of four possible

directions (north, south, west, and east) to a spot adjacent to its current one in each step. However, the agent has to avoid falling into one of the lake's holes, which are present at certain locations that are unknown to the agent at the beginning. Moreover, the lake can be either non-slippery, i.e., the agent always reaches the lake tile which corresponds to his chosen action, or slippery, i.e., at random times the agent's action is ignored and a random move to an adjacent lake tile is taken instead. A visualization of the environment can be found in Figure 9.4.
First, get familiar with `gymnasium` [Tea23] and the ideas of value iteration. To this end, refer to the JUPYTER notebook `ReinforcementLearning_template.ipynb` at https://bookstore.siam.org/di03/bonus.

(a) We pre-defined a class `RandomAgent`, which takes a random step in any of the four directions. Implement the `action` function for this agent and experimentally estimate the expected value of the agent's policy on the FrozenLake-v1 environment with `is_slippery` first `False` then `True`. What do you observe and why?

(b) Implement the missing piece of iterative policy evaluation to calculate the value function for the random policy from task (a) with $\gamma = 0.9$ with `is_slippery = False`. What is the value function if `is_slippery` is `True`?

(c) Implement the missing piece of value iteration to calculate an optimal policy for the FrozenLake-v1 environment where `is_slippery` is `False`. Experimentally estimate the expected return for these optimal policies.

Figure 9.4: A visualization of the FrozenLake-v1 environment in `gymnasium` created with the `gymnasium.env.render` function; see [Tea23]. The agent (top left corner) has to travel to the destination (bottom right corner) and has to avoid the four holes in the lake. Modified image used with the kind permission of the Farama foundation and Francisco Coda.

### 9.1.3 ▪ Policy iteration

In Algorithm 12 we have seen how a fixed point iteration based on the contraction properties of $\mathcal{B}$ can be used to find an optimal policy. Note that, during the iteration process, we only keep track of a function $V$, which implicitly defines a policy $\pi$. Alternatively, we can compute and store an optimal policy directly. To this end, *policy iteration*, which is based on (9.7), is an appropriate method. Each iteration of this approach consists of two steps: First, *policy evaluation* is applied; see Algorithm 11. Second, the so-called *policy improvement* step is employed.

In particular, we start with an initial policy $\pi$ for which we first evaluate $V^\pi$ by policy evaluation (Algorithm 11). Then, we obtain a better policy

$$\pi'(s) \leftarrow \arg\max_{a \in A_s} \big(R(s,a) + \gamma V^\pi(T(s,a))\big) \tag{9.10}$$

by greedily picking the best action $a$ for each state $s$ according to $V^\pi$. This resembles the policy improvement step; see Figure 9.5 for an example of two steps of the overall iterative procedure. The policy iteration algorithm now alternates these two steps; see Algorithm 13.

**Algorithm 13:** Policy iteration

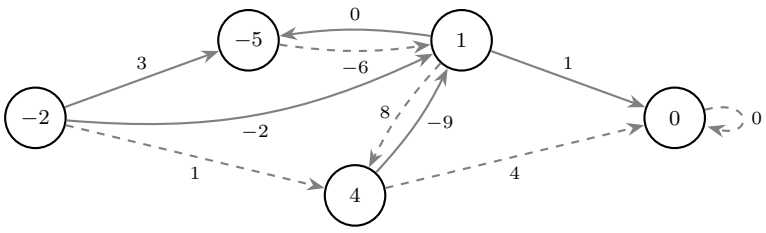
**Input:** deterministic, discounted MDP ($\mathcal{S}$, $A$, $R$, $T$, $\gamma$), threshold $\delta > 0$.
**Output:** (approximations to) an optimal policy $\pi^*$ and the corresponding value function $V^*$.

1 Initialize $\pi', \boldsymbol{v}'$ arbitrarily.
2 **repeat**
3 $\quad \boldsymbol{v} \leftarrow \boldsymbol{v}'$.
4 $\quad \pi \leftarrow \pi'$.
5 $\quad$ Solve $(\boldsymbol{I} - \gamma \boldsymbol{T})\boldsymbol{v}' = \boldsymbol{r}$ and set $V^\pi(s) \leftarrow \boldsymbol{v}'_s$ for all $s \in \mathcal{S}$ (or approximate $\boldsymbol{v}'_s = V^\pi(s)$ by policy evaluation with Algorithm 11).
6 $\quad$ **forall** $s \in \mathcal{S}$ **do**
7 $\quad\quad \pi'(s) \leftarrow \arg\max_{a \in A_s} \big(R(s,a) + \gamma V^\pi(T(s,a))\big)$.
8 $\quad$ **end forall**
9 **until** $\max_{s \in S} |\boldsymbol{v} - \boldsymbol{v}'| \leq \delta$.
10 **return** $V^{\pi'}$, $\pi'$.

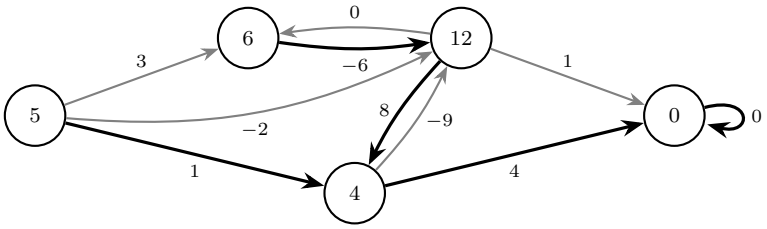
In comparison, Algorithm 12 (value iteration) updates the estimate of the values in each loop over the state space. Subsequently, after each such loop, it (implicitly) obtains a new policy based on the updated value function. In contrast to that, Algorithm 13 updates the estimate of the values for the current policy to a prescribed accuracy in each policy evaluation step. Then, given this (stable) value, a new policy is determined in the policy improvement step. In terms of the number of policy updates, policy iteration usually converges faster than value iteration, in part because more effort is spent between updates.

Note that in line 5 of Algorithm 13, instead of using Algorithm 11, one can solve a system of linear equations to obtain the current policy iterate. This is potentially more efficient. To see why both approaches are equivalent, we define the reward vector $\boldsymbol{r} \in \mathbb{R}^{|\mathcal{S}|}$ with entries $\boldsymbol{r}_s := R(s, \pi(s))$ for all states $s \in \mathcal{S}$. Moreover, we define the state transition matrix $\boldsymbol{T} \in \mathbb{R}^{|\mathcal{S}| \times |\mathcal{S}|}$ with entries $\boldsymbol{T}_{s,s'} := \delta_{T(s,\pi(s)),s'}$ for all states $s, s' \in \mathcal{S}$ with

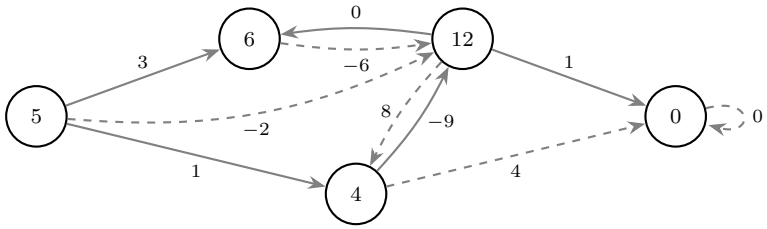
$$\delta_{T(s,\pi(s)),s'} := \begin{cases} 1 & \text{if } T(s,\pi(s)) = s', \\ 0 & \text{else.} \end{cases}$$



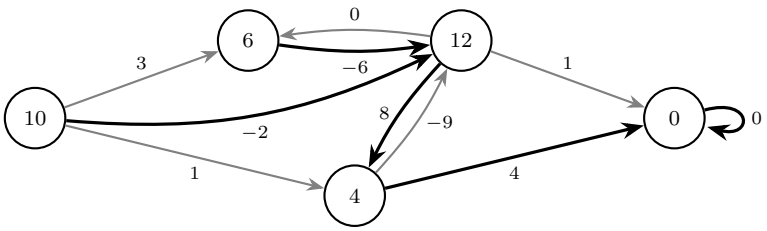

(a) Policy improvement.




(b) Policy evaluation.




(c) Policy improvement.




(d) Policy evaluation.

Figure 9.5: Subsequent steps (a)–(d) of the policy iteration algorithm for the example from Figure 9.3. The new action $\pi'(s)$ of a policy improvement step is indicated by a dashed edge for each $s \in \mathcal{S}$.

With this, the fixed point equation

$$\mathcal{B}_\pi v = v \tag{9.11}$$

for policy evaluation for a deterministic MDP can be written as

$$\boldsymbol{v} = \boldsymbol{r} + \gamma \boldsymbol{T} \boldsymbol{v},$$

where $\boldsymbol{v} \in \mathbb{R}^{\mathcal{S}}$ has the entries $\boldsymbol{v}_s = v(s)$. This indeed provides an alternative to Algorithm 11 since (9.11) is equivalent to solving the system of linear equations

$$(\boldsymbol{I} - \gamma \boldsymbol{T})\boldsymbol{v} = \boldsymbol{r}.$$

Overall, policy iteration, as shown in Algorithm 13, is an iterative method that searches in the space of policies until it has converged to an optimal policy. The approach was developed by Bellman [Bel57] and Howard [How60] and is also known as Howard's algorithm. It can be shown that policy iteration converges since the number of states and actions in the MDP is finite; see [Ber12, Ber19, SB18, Mur23]. Under a monotonicity assumption on the matrices $\boldsymbol{T}$ for each policy, one can indeed prove that policy iteration converges superlinearly. Moreover, under certain additional regularity assumptions, even quadratic convergence can be shown; see [BMZ09, SR04] for details. This superior result is related to an interpretation of policy iteration as a semismooth Newton method for finding a root of $\mathcal{B}v - v = 0$. In particular, line 5 of Algorithm 13, which reflects the linear problem from (9.8), can be seen as solving a linearization of the nonlinear problem (9.9), similar to a Newton method.

### 9.1.4 ▪ Stochastic environments

A more general setting than that of deterministic optimal control is that of stochastic optimal control. Here, the environment is considered in a stochastic fashion. To this end, a *stochastic Markov decision process* is employed as a model for the environment.

**Stochastic Markov decision process**

A tuple $(\mathcal{S}, A, \tau, r, \gamma)$ is called a ***stochastic, discounted Markov decision process*** (MDP) if

- $\mathcal{S}$ is a finite nonempty set, called the state space,
- $A = \cup_{s\in\mathcal{S}} A_s$ is a union of finite, nonempty sets $A_s$,
- $\tau$ is a family of probability measures $P_a(s, \cdot)$ on $\mathcal{S}$ depending on $s \in \mathcal{S}$ and indexed by $a \in A_s$,
- $r\colon \big\{(s, a, s') \mid s, s' \in \mathcal{S},\ a \in A_s\big\} \to \mathbb{R}$ is a real-valued function, and
- $\gamma \in (0, 1]$ is a *discount factor*.

As in the deterministic case, the set $A$ is the *action space* and $A_s$ is the *set of admissible actions in state* $s$. $P_a(s, s')$ is now the probability that the system will change to state $s'$ if action $a$ has been taken in state $s$. The corresponding *reward* is given by $r(s, a, s')$. In the stochastic setting, a family $\pi$ of probability measures $\pi(s)$ on $A_s$ indexed by $s \in \mathcal{S}$ is a *policy*, where an *agent* selects an action in $s$ based on $\pi(s)$.

Note that a stochastic Markov decision process can also be used to model a deterministic one. To this end, Dirac measures are used as the probability measures $P_a(s, \cdot)$. To be consistent with the deterministic case, where $\pi : \mathcal{S} \to A$ is a function, notation is oftentimes abused in the stochastic setting and one writes $\pi(s)$ instead of $a := \arg\max_{a'\in A_s} \pi(s)(a')$ when referring to the action $a$ that is most likely in state $s$ for the policy $\pi$. Stochastic policies are needed for tasks involving partially observable environments, e.g., for bluffing in card games with incomplete information, such as poker (see [SB18]), for disguising the state to obfuscate private information,

or for exploring the state space, as we will see in Section 9.2.5. One can formally define analogous expressions of (9.1) in the stochastic setting; see, e.g., [BS96, FS06, SB18]. Then, (9.2) gives the optimal deterministic policy as before. For details on stochastic policies, see [SB18].

All algorithms from the deterministic case can be employed to obtain a deterministic policy in a stochastic environment if we slightly alter the definitions: The Bellman optimality equation (9.6) can be redefined by taking the expected value over the probabilities to obtain

$$V^*(s) \coloneqq \max_{a \in A_s} \sum_{s' \in \mathcal{S}} P_a(s, s') \big( r(s, a, s') + \gamma V^*(s') \big) \tag{9.12}$$

in the case of stochastic transactions. Then, Algorithm 11 and Algorithm 12 can be reformulated for a stochastic environment. To this end, we replace line 5 in Algorithm 11 with

$$V'(s) \leftarrow \sum_{a \in A_s} \left( \pi(s)(a) \sum_{s' \in \mathcal{S}} P_a(s, s') (r(s, a, s') + \gamma V(s')) \right)$$

and line 5 in Algorithm 12 with

$$V'(s) \leftarrow \max_{a \in A_s} \left( \sum_{s' \in \mathcal{S}} P_a(s, s') (r(s, a, s') + \gamma V(s')) \right). \tag{9.13}$$

Here, $\pi(s)(a)$ denotes the probability that $\pi$ picks action $a$ given state $s$.

Besides policy evaluation and value iteration, we can also reformulate policy iteration (Algorithm 13) for a stochastic environment. To this end, the entries of the state transition matrix $\boldsymbol{T}$ become $\boldsymbol{T}_{s,s'} := P_{\pi(s)}(s, s')$. Furthermore, line 7 in Algorithm 13 becomes

$$\pi'(s) \leftarrow \arg\max_{a \in A_s} \sum_{s' \in \mathcal{S}} P_a(s, s') \big( r(s, a, s') + \gamma V^{\pi}(s') \big). \tag{9.14}$$

Note that one can further generalize the setting and also define the reward $r$ as probabilistic. In this case, Algorithm 11 and Algorithm 12 can be extended to cover such probabilistic rewards as well. For reasons of simplicity, we will only consider deterministic rewards in the following.

## 9.2 ▪ Classic reinforcement learning

So far, we have dealt with problems that were simple enough to understand them completely, i.e., we had access to both the transition map $T$ and the reward function $R$—or $\tau$ and $r$ in the stochastic setting—and we were able to evaluate them at arbitrary states $s \in \mathcal{S}$ and actions $a \in A_s$. In contrast, a reinforcement learning algorithm may not use (full) knowledge about the environment explicitly. In particular, for many real-world applications, the dynamics of the system are unknown.

For example, in [Mea15] several Atari games have been considered. Here, no rules of the game, such as the (potentially stochastic) system dynamics, are available to the agent. A state just consists of the most recent four picture frames of the game's screen. Clearly, the space of all conceivable states $\mathcal{S}$ is enormously large, and it is no longer feasible to compute or even to store all possible state transitions and rewards. In this setting, the authors achieve superhuman performance employing a reinforcement learning approach.

Generally, the specific information that is available about an environment will depend on the application at hand. In particular, $T$—or $\tau$ in the stochastic setting—is typically neither known nor (generally) accessible, but transitions can at least be observed. To tackle a given problem, RL algorithms use certain sampling techniques in the state space and in the action space. Based

on the sampled trajectories, approximations of the value function or the optimal policy are then computed, for instance.

In *model-based* reinforcement learning, algorithms are employed that train a model of the transition map from sampled trajectories. Here, while the true transition map is not available, both value iteration (Algorithm 12) and policy iteration (Algorithm 13) can in principle be applied as before, but now using the learned model. In the following, we will focus on *model-free* reinforcement learning approaches, where the sampled trajectories are directly used, e.g., to approximate the value function, without employing a model of the transition map.

The remainder of this section is structured as follows: After introducing the general reinforcement learning setting in Section 9.2.1, we discuss a Monte Carlo approach for policy evaluation in Section 9.2.2. Here, the agent learns from experience, i.e., from sampled trajectories, in order to estimate the value function. This is in line with the conceptual idea of mimicking learning through positive and negative reinforcement. A more advanced technique than simple Monte Carlo estimation is the *temporal difference* (TD) method, which we explore in Section 9.2.2. The combination of the concepts of temporal differences and *Q-functions* leads to the *SARSA* algorithm presented in Section 9.2.4. In Section 9.2.5 we discuss the tradeoff between *exploration and exploitation* and the resulting *Q-learning* algorithm. Subsequently, we introduce the *binning* approach to deal with infinite state spaces in Section 9.2.6. Finally, Section 9.2.7 provides tasks on the introduced RL algorithms.

## 9.2.1 ▪ Setting

**Reinforcement learning versus optimal control** In the optimal control setting we had perfect knowledge about the state transitions. But this is no longer the case in reinforcement learning and an algorithm cannot explicitly make use of $T$ or $\tau$ and the probability measures $P_a$ for any $a \in A$ anymore.

Thus, in the general reinforcement learning setting, information can only be obtained in an interactive fashion, i.e., one observes the next state $s_{t+1}$ and the reward $r_{t+1}$ without explicitly knowing the state dynamics $T$ or the reward function $R$ (or $\tau$ and $r$ in the stochastic setting, respectively). To avoid any confusion about the nature of the environment, we will restrict ourselves to the stochastic setting[41] for the remainder of this chapter and pose the following assumptions:

1. At any point $t \in \mathbb{N}$ in time the environment an agent is interacting with is in state $s_t \in \mathcal{S}$. We can choose $a_t \in A_s$ and then the environment transitions to the next state $s_{t+1}$, which is sampled from $P_{a_t}(s_t, s_{t+1})$. Note however that we do not assume that $P_{a_t}$ is explicitly known.
2. We cannot put the system into an arbitrary state $s \in \mathcal{S}$.
3. The system might reset (possibly also at our will) to a state from a set of initial states.[42]

The above assumptions model a human learner, who does not know a priori how the environment reacts when a certain action is taken. Of course, various modifications of these assumptions exist in the RL literature. Note that the line between classical optimal control and modern reinforcement learning is further blurred by the fact that, in case the state space $\mathcal{S}$ is very large, the application of RL algorithms using sampled trajectories is often computationally advantageous in contrast to classical optimal control methods, even in the case of perfect knowledge of the environment. Here, supervised learning techniques (e.g., regression) can be employed to learn $V^\pi$ from samples [Ber12].

[41] Note however that the deterministic setting is just a special case of the stochastic setting.

[42] This is necessary to be able to sample multiple trajectories.

**Bellman optimality operator in RL** To understand the implications of our above assumptions, let us consider the Bellman operator $\mathcal{B}_\pi$ (see (9.8)) and the Bellman optimality operator $\mathcal{B}$ (see (9.9)) in a stochastic environment, i.e.,

$$(\mathcal{B}_\pi v)(s) := \sum_{a \in A_s} \pi(s)(a) \sum_{s' \in \mathcal{S}} P_{\pi(s)}(s, s') \big( r(s, a, s') + \gamma v(s') \big), \tag{9.15}$$

and

$$(\mathcal{B} v)(s) := \max_{a \in A_s} \sum_{s' \in \mathcal{S}} P_a(s, s') \big( r(s, a, s') + \gamma v(s') \big). \tag{9.16}$$

Recall that we actually want to solve $\mathcal{B}_\pi v = v$ or $\mathcal{B} v = v$, respectively, over $v : \mathcal{S} \to \mathbb{R}$, which is equivalent to finding a root of $\mathcal{B}_\pi v - v$ or $\mathcal{B} v - v$, respectively; see Theorem 9.1.1. However, in contrast to the last section, we cannot compute (9.15) or (9.16) directly here since we do not have access to $P_a$. Therefore, we will now consider techniques to estimate $V^\pi$ and $V^*$ with sampling techniques instead.

## 9.2.2 ▪ Monte Carlo sampling

The overall idea of Monte Carlo (MC) sampling is to generate trajectories based on observed states, actions, and rewards from actual (or simulated) interactions with the environment. First, let us consider a scenario where we want to approximate the value function $V^\pi$ for a given policy $\pi$. The approximation of an optimal policy $V^*$ is discussed in later sections.

To this end, an episodic Monte Carlo-based policy evaluation variant of Algorithm 11 is given in Algorithm 14. Here, based on the collected observations, one approximates the value function by using averages of the values of the sampled trajectories. To ensure that the value function is well defined, we only consider episodes of length at most $l > 0$. This means that trajectories always end after at most $l$ steps, independent of the chosen actions.

In particular, we generate a sufficiently large number of trajectories and compute observed discounted cumulative rewards[43]

$$C(s_i) := \sum_{k=i}^{\ell-1} \gamma^{k-i} r_{k+1} \tag{9.17}$$

for the states $s_i$ occurring over a trajectory. We call the above quantity $C(s_i)$ the *target* for $V^\pi(s_i)$. Then we use the average of the observed targets over all trajectories as an estimate for $V^\pi(s_i)$.

In Algorithm 14 only the first visit of a state during an episode is considered when estimating the value function and later visits are neglected. It is easy to see that each $target$ in the list $Returns(s_i)$ is an independent, identically distributed estimate of $V^\pi$ with finite variance. Using the law of large numbers, one can obtain convergence of the averages of $Returns(s)$ to their expected value $V^\pi(s)$ in case of this first-visit Monte Carlo approach for all $s \in \mathcal{S}$ [SS96, SB18]. This basic Monte Carlo policy evaluation method can then be used instead of line 5 in policy iteration (Algorithm 13), for example.

## 9.2.3 ▪ Temporal difference learning

Another popular technique used when computing an estimate of a value function is *temporal difference* learning. It constitutes an important historical milestone in RL.

[43] Note at this point that fewer summands in $C(s_i)$ are employed for an $i$ close to $l - 1$ than for a smaller $i$.

**Algorithm 14:** Episodic first-visit Monte Carlo policy evaluation

**Input:** policy $\pi$, episode length $\ell > 0$.
**Output:** (approximation to) the value function $V^\pi$.

1 Initialize $Returns(s_i)$ as an empty list for all $s_i$.
2 **repeat**
3  Choose $s_0 \in \mathcal{S}$ uniformly at random among possible start states.
4  Sample $\pi$ to generate a trajectory $s_0, a_0, r_1, s_1, a_1, r_2, \ldots, s_{\ell-1}, a_{\ell-1}, r_\ell$.
5  $target \leftarrow 0$.
6  **forall** $i \in \ell - 1, \ell - 2, \ldots, 0$ **do**
7  $target \leftarrow r_{i+1} + \gamma \cdot target$.
8  **if** $s_i \notin \{s_0, s_1, \ldots, s_{i-1}\}$ **then**
9  Append $target$ to $Returns(s_i)$.
10  $V(s_i) \leftarrow average(Returns(s_i))$.
11  **end if**
12  **end forall**
13 **until** some stopping criterion is met.
14 **return** $V$.

**Temporal differences**

The idea behind ***temporal difference*** (TD) learning is to use incremental updates for the computation of $V^\pi$ after every state transition of the system. When transitioning from $s_i$ to $s_{i+1}$ by taking action $a_i$, the basic TD update step for the estimator $V$ of $V^\pi$ is given by

$$V(s_i) \leftarrow V(s_i) + \alpha d_i \tag{9.18}$$

for some appropriately chosen step size $\alpha > 0$ while using the temporal difference

$$d_i := r(s_i, a_i, s_{i+1}) + \gamma V(s_{i+1}) - V(s_i); \tag{9.19}$$

see also [Sut88, SB18]. Here, an update of $V$ can be computed immediately after a state transition. This is in contrast to Monte Carlo approaches, where the whole trajectory of length $l$ has to be known before an update of $V(s_i)$ can be computed for any $s_i$ contained in the trajectory.

**Relation between Monte Carlo and temporal differences** To see how TD is related to an episodic Monte Carlo policy evaluation, let us rewrite the latter. To this end, recall definition (9.17) for $C(s_i)$, which resembles the $target$ value from Algorithm 14. Then, defining $m(s_i)$ as the number of first visits of $s_i$ over all trajectories up to the current one, we can update the estimate $V$, initialized by zero, at each visit of $s_i$ by

$$V(s_i) \leftarrow V(s_i) + \frac{1}{m(s_i)}(C(s_i) - V(s_i)) = \frac{m-1}{m}V(s_i) + \frac{1}{m}C(s_i).$$

This update is equivalent to the computation in line 10 in Algorithm 14. Now we rewrite the above update step using the temporal differences $d_i$ from (9.19). To this end, we express the term

$C(s_i) - V(s_i)$ as

$$\begin{aligned} C(s_i) - V(s_i) &= r(s_i, a_i, s_{i+1}) + \gamma C(s_{i+1}) - V(s_i) \\ &= d_i + \gamma \left( C(s_{i+1}) - V(s_{i+1}) \right) \\ &= d_i + \gamma d_{i+1} + \gamma^2 \left( C(s_{i+2}) - V(s_{i+2}) \right) \\ &= \dots \\ &= \sum_{k=i}^{\ell-1} \gamma^{k-i} d_k. \end{aligned} \tag{9.20}$$

Thus we can rewrite line 10 of Algorithm 14 as

$$V(s_i) \leftarrow V(s_i) + \frac{1}{m(s_i)} \sum_{k=i}^{\ell-1} \gamma^{k-i} d_k, \tag{9.21}$$

which is computed at the end of the episode.

In a temporal difference method an update is performed earlier than in MC methods. For instance, (9.18) describes an update of the current estimate $V(s_i)$ directly after each transition. Here, $d_i$ from (9.19) is used to approximate $C(s_i) - V(s_i)$, i.e., only the first summand in (9.20) is employed, but with a prefactor $\alpha$, which is (possibly) different from the factor $\frac{1}{m(s_i)}$ in (9.21). The update is directly performed when the subsequent state $s_{i+1}$ is reached, i.e.,

$$V(s_i) \leftarrow V(s_i) + \alpha d_i = V(s_i) + \alpha \left[ r(s_i, a_i, s_{i+1}) + \gamma V(s_{i+1}) - V(s_i) \right], \tag{9.22}$$

with step size $\alpha > 0$. As we see, $d_i$ contains $r(s_i, a_i, s_{i+1}) + \gamma V(s_{i+1})$ as the target value of the update. Here, the current estimate for $V(s_{i+1})$ is used, which replaces $C(s_i)$ from the Monte Carlo update. In this way, the temporal difference can be understood as a residual (or "error") between the current estimate $V(s_i)$ and the observed value $r(s_i, a_i, s_{i+1}) + \gamma V(s_{i+1})$ after the state transition. This is the reason why $d_i$ is sometimes also referred to as *TD error*. One can show (under mild assumptions) that a policy evaluation algorithm using the TD update[44] will also converge to $V^\pi$ [Sut88, SB18].

A significant advantage of updating the value function by (9.22) compared to the update (9.20) is the possibility to use a TD update in an *online* fashion.[45] In contrast to that, one has to wait until a trajectory has been fully explored before an update can be computed with Monte Carlo.

### 9.2.4 ▪ Q-functions and SARSA

Instead of using a value function $V^\pi$ evaluated in states $s \in \mathcal{S}$, we can alternatively employ an action-value function $Q^\pi$, a so-called *Q-function*, evaluated in state-action pairs $(s, a)$ with $s \in \mathcal{S}$, $a \in A_s$. The state-action values $Q^\pi(s, a)$ are also known as *Q-values* or *Q-factors*. Here, $Q^\pi(s, a)$ is the expected total (discounted) reward that we receive by starting in state $s \in \mathcal{S}$, choosing action $a \in A_s$, and subsequently following the policy $\pi$. In particular, $Q^\pi$ is

[44] Note at this point that an approach of using a current estimate (in our case $V(s_{i+1})$) when computing an update for another state (in our case $V(s_i)$) is often referred to as *bootstrapping* in the RL literature. As we will see in Section 10.2.4, the term bootstrapping has a different meaning when we consider sampling methods.

[45] This means that each new iterate can be computed by using only the last iterate and the current reward.

recursively defined by

$$Q^\pi : \{(s,a) \mid s \in S, a \in A_s\} \to \mathbb{R}$$
$$(s,a) \to \sum_{s' \in \mathcal{S}} P_a(s,s') \left( r(s,a,s') + \gamma \sum_{a' \in A_{s'}} \pi(s')(a')\, Q^\pi(s',a') \right). \tag{9.23}$$

Expanding the recursion into infinitely many nested sums, we obtain

$$\begin{aligned} Q^\pi(s_0,a_0) = \sum_{s_1 \in \mathcal{S}} P_{a_0}(s_0,s_1) \Bigg( & r(s_0,a_0,s_1) + \gamma \sum_{a_1 \in A_{s_1}} \pi(s_1)(a_1) \\ & \cdot \sum_{s_2 \in \mathcal{S}} P_{a_1}(s_1,s_2) \Bigg( r(s_1,a_1,s_2) + \gamma^2 \sum_{a_2 \in A_{s_2}} \pi(s_2)(a_2) \\ & \cdot \sum_{s_3 \in \mathcal{S}} P_{a_2}(s_2,s_3) \left( r(s_2,a_2,s_3) + \gamma^3 \sum_{a_3 \in A_{s_3}} \pi(s_3)(a_3) \dots \right) \Bigg) \Bigg) \Bigg). \end{aligned}$$

Note that the definition of $Q^\pi$ is analogous to (9.4) but it is employed now in the stochastic setting and with an $(s,a)$-dependent action-value function $Q^\pi$ instead of an $s$-dependent value function $V^\pi$. Besides algorithmic aspects, such Q-functions are often employed to simplify the analysis of RL algorithms; see [SB18].

Note that employing a Q-function $Q^\pi$ instead of a value function $V^\pi$ implies more than just a notational difference. Indeed, a Q-function $Q^\pi(s,a)$ provides direct access to the value for *any* action $a \in A_s$ taken in state $s \in \mathcal{S}$. This has to be seen in contrast to $V^\pi(s)$, which is the value when taking the fixed action $\pi(s)$ in state $s$. Furthermore, $Q^\pi(s,a)$ provides the value of the action $a \in A_s$ for a given state $s \in \mathcal{S}$, even if we do not know which state $s'$ will occur after employing $a$. This is of particular importance in the case of model-free reinforcement learning, i.e., in the case where we no longer have a model for the transition probabilities.

**Optimal Q-functions** Let $Q^*$ be the *optimal Q-function*, i.e.,

$$Q^*(s,a) := \max_\pi Q^\pi(s,a).$$

Then, the optimal policy $\pi^*$ at state $s$ is given as the maximum over the Q-values of the available actions, i.e.,

$$\pi^*(s) := \arg\max_{a \in A_s} Q^*(s,a). \tag{9.24}$$

Furthermore, we can deduce from $Q^*$ the $s$-dependent value function by

$$V^*(s) = \max_{a \in A_s} Q^*(s,a).$$

The crucial difference between (9.24) and (9.7) is that we no longer explicitly need the reward $r$ or the transition probabilities $\tau$ in the definition of the optimal policy. However, the drawback

of the above approach is that more values need to be stored and computed for $Q^\pi$ than for the corresponding value function $V^\pi : \mathcal{S} \to \mathbb{R}$.

**Policy evaluation, value iteration, and policy iteration for Q-functions** Because of the recursive relationship (9.23), it is straightforward to formulate iterative optimization schemes in the fashion of Algorithm 11, Algorithm 12, and Algorithm 13, but with Q-functions instead of value functions. For example, we can use Q-functions in policy iteration with a Monte Carlo policy evaluation as outlined in Section 9.2.2. Furthermore, we are now able to define a Monte Carlo-based value iteration scheme with Q-functions, similar to Algorithm 12. For that, we alternate between the following steps:

1. In state $s_i$ we take an action $a_i \sim \pi(s_i)(\cdot)$ relying on a policy $\pi$ defined by the greedy rule (9.24) based on our current Q-function estimate, and we observe a reward $r_i$ and the new state $s_{i+1}$.
2. Similar to line 5 of Algorithm 12, we modify our estimate of the optimal Q-function by $Q(s_i, a_i) \leftarrow r_i + \gamma \max_{a \in A_{s_{i+1}}} Q(s_{i+1}, a)$.

In this way, our estimate of $Q$ will converge towards the optimal Q-function $Q^*$ under suitable assumptions. Note that, for constructing such a Monte Carlo-based value iteration scheme with a value function $V$ instead of a Q-function $Q$, either we would have to employ a model-based approach or we would need to sample each possible action in step 2 from above to obtain the next state and its value, which is typically not feasible.

**SARSA** Instead of considering these simple Monte Carlo-based approaches in detail, we will have a closer look at SARSA, a more advanced RL algorithm that is based on temporal differences and Q-functions. To this end, let us rewrite the TD update formula (9.22) with Q-functions to obtain

$$Q(s_i, a_i) \leftarrow Q(s_i, a_i) + \alpha \left( r(s_i, a_i, s_{i+1}) + \gamma Q(s_{i+1}, a_{i+1}) - Q(s_i, a_i) \right). \tag{9.25}$$

Note that, when replacing $V(s_{i+1})$ with the Q-function value $Q(s_{i+1}, a_{i+1})$, we already use $a_{i+1}$ when updating $Q(s_i, a_i)$.

Now, when evaluating a fixed policy $\pi$, we take an action $a_{i+1} \sim \pi(s_{i+1})(\cdot)$. In particular, in each step of (9.25), our knowledge of $Q(s_i, a_i)$ improves through the reward $r_i = r(s_i, a_i, s_{i+1})$ and through our current estimate of the value $Q(s_{i+1}, a_{i+1})$ of the next state $s_{i+1}$ and the next action $a_{i+1} \sim \pi(s_{i+1})(\cdot)$.

**SARSA**

***SARSA*** (or Sarsa) stands for **s**tate-**a**ction-**r**eward-**s**tate-**a**ction. The name refers to the five variables $s_i$, $a_i$, $r_i$, $s_{i+1}$, and $a_{i+1}$ at iteration $i$. The goal of SARSA is to approximate the optimal Q-function by sampling trajectories. For that, SARSA follows the idea of (9.25), i.e., the Q-function is learned from experience; see Algorithm 15. After $Q$ has been determined, an estimate of an optimal policy $\pi^*$ is derived by using $Q$ as a surrogate for the optimal Q-function $Q^*$ and by employing (9.24).

Note that SARSA is open-ended by nature, i.e., we investigate trajectory after trajectory. Therefore, we need to provide a stopping condition for the algorithm to terminate. For our purpose, we simply use a fixed number of $N > 0$ trajectories that we traverse.

**Algorithm 15:** SARSA

**Input:** stochastic MDP $(\mathcal{S}, A, \tau, r, \gamma)$, number of trajectories $N > 0$, step size $\alpha > 0$, action-selection strategy $\tilde{\pi}$ (e.g., $\varepsilon$-greedy).

**Output:** (approximations to) the optimal Q-function $Q^*$ and an optimal policy $\pi^*$.

Initialize $Q$ arbitrarily such that $Q(s,a) \leftarrow 0$ for all terminal states $s \in \mathcal{S}$ and all actions $a \in A_s$.
**for** $i \in \{0, \dots, N\}$ **do**
  Initialize a starting state $s \in \mathcal{S}$ randomly.
  Select $a$ according to $\tilde{\pi}(s)$.
  **repeat**
    Take action $a$, sample next state $s'$ according to $P_a(s,\cdot)$, and observe reward $r = r(s,a,s')$.
    Select $a'$ according to $\tilde{\pi}(s')$.
    $Q(s,a) \leftarrow Q(s,a) + \alpha\,(r + \gamma Q(s',a') - Q(s,a))$.
    $s \leftarrow s'$.
    $a \leftarrow a'$.
  **until** $s$ is terminal.
**end for**
Derive policy $\pi$ from $Q$ by the greedy rule (9.24).
**return** $Q, \pi$.

**Action-selection strategies** When sampling trajectories, we have so far employed a fixed[46] policy $\pi$ to calculate an action $\pi(s_i)$ in state $s_i$. However, for SARSA we now allow for a more general *action-selection strategy* $\tilde{\pi}$. For example, consider the *greedy* action-selection strategy

$$\tilde{\pi}(s) := \operatorname*{arg\,max}_{a \in A_s} Q(s,a), \tag{9.26}$$

which at each step follows a policy that is greedily determined from the current Q-function; see (9.24). This highlights that the action can be selected according to different policies in each iteration step. It is important to note that we deliberately do not denote $\tilde{\pi}$ as a policy since $\tilde{\pi}$ is not a function (or a family of probability measures) on $S$ but it inherently depends on $Q$.

A common alternative to the greedy strategy (9.26) is the so-called $\varepsilon$-*greedy* strategy, which is also typically used in SARSA (Algorithm 15) instead of a fixed policy $\pi$. The $\varepsilon$-greedy strategy picks a random action with probability $0 < \varepsilon \leq 1$ and a value-maximizing action—like the greedy strategy—with probability $1-\varepsilon$. This means that we will obtain the rewards of suboptimal actions throughout the learning process in the $\varepsilon$-greedy case, but we will observe more diverse actions than in the greedy case. This leads to better coverage of the state space and of the action space.

### 9.2.5 ▪ Exploration-exploitation tradeoff and Q-learning

As we have seen in SARSA, a greedy action-selection strategy returns the optimal action according to the current Q-function approximation. However, one typically needs to allow also non-optimal actions to explore the state and action spaces sufficiently, which can be achieved by an $\varepsilon$-greedy strategy, for example.

[46] Note that we determined an action also according to the current Q-function in the sketched Monte Carlo value iteration scheme earlier.

**Exploration and exploitation**

Two key aspects when determining an optimal policy $\pi^*$ are ***exploration*** and ***exploitation***. Initially, we traverse trajectories and use the collected state-action-reward information to determine an estimate of $\pi^*$. Given a current policy estimate, the question arises how to gather additional state-action-reward information. Here, either we could traverse the state and action spaces in an (approximately) optimal manner by following a greedy action-selection strategy—and thereby *exploit* the information that has already been observed—or we could further *explore* the state and action spaces, i.e., we (at least partially) ignore the optimal action. Note that, in order to learn the optimal policy $\pi^*$, the trajectories do not need to reflect the optimal paths according to the current Q-function estimate. In fact, if an action $a$ is always picked in a greedy manner, it is possible that a path for a more lucrative action in the long run is missed. The underlying problem is called the tradeoff (or dilemma) of exploration and exploitation; see [Ber19, SB18].

A common choice for exploring the state and action spaces is the $\varepsilon$-greedy action-selection strategy. Note that, at the start of the learning process, this strategy is usually employed with a large value of $\varepsilon$ to emphasize exploration. Then, the value of $\varepsilon$ is successively reduced over the course of training to converge to the greedy strategy to emphasize exploitation. This is one of the most popular approaches since it is easy to implement and yet very powerful. Another example of a commonly used explorative strategy is the *softmax exploration*, also called *Boltzmann exploration*. Here, an action $a$ is drawn randomly according to the softmax distribution; see [Ber19, SB18] for details.

**Q-learning** In SARSA, one action-selection strategy has been used to both determine the next action in the current trajectory and calculate the update step for $Q$ using (9.25). This is known as *on-policy learning*. In contrast to that, we now consider an *off-policy learning* approach, where the traversal of the state space and the update step are based on different strategies.[47]

**Q-learning**

The so-called ***Q-learning*** algorithm introduced in [Wat89] is similar to SARSA. However, while the action-selection strategy $\tilde{\pi}$ is still used to traverse the state space, there is a significant difference for the update of the Q-function in Q-learning. Here, the update step considers the maximum Q-function value over all possible actions $a' \in A_{s'}$, i.e., the update uses a greedy strategy; see Algorithm 16.

The estimation of $Q$ in Q-learning is analogous to the value iteration scheme from Algorithm 12. If we compare the update step from both algorithms, we observe that both depend on the optimal action given the current estimate of $V$ or $Q$, respectively. However, different update rules are employed: In Algorithm 12 the update formula stems from the dynamic programming principle, while in Algorithm 16 it stems from the temporal differences.

**Relation between SARSA, Q-learning, and policy iteration** Note that Q-learning and SARSA coincide if the action-selection strategy $\tilde{\pi}$ is chosen to be the greedy strategy, i.e., if

[47] Note that the action-selection strategy for the next trajectory sample can even be based on a conventional controller or on a human expert in off-policy learning.

**Algorithm 16:** Q-learning

**Input:** stochastic MDP $(\mathcal{S}, A, \tau, r, \gamma)$, number of trajectories $N > 0$, step size $\alpha > 0$, action-selection strategy $\tilde{\pi}$ (e.g., $\varepsilon$-greedy).

**Output:** (approximations to) the optimal Q-function $Q^*$ and an optimal policy $\pi^*$.

Initialize $Q$ arbitrarily such that $Q(s,a) \leftarrow 0$ for all terminal states $s \in \mathcal{S}$ and all actions $a \in A_s$.
**for** $i \in \{0, \dots, N\}$ **do**
  Initialize a starting state $s \in \mathcal{S}$ randomly.
  **repeat**
    Select $a$ according to $\tilde{\pi}(s)$.
    Take action $a$, sample next state $s'$ according to $P_a(s, \cdot)$, and observe reward $r = r(s, a, s')$.
    $Q(s,a) \leftarrow Q(s,a) + \alpha \left(r + \gamma \max_{a' \in A_{s'}} Q(s', a') - Q(s,a)\right)$.
    $s \leftarrow s'$.
  **until** $s$ is terminal.
**end for**
Derive policy $\pi$ from $Q$ by the greedy rule (9.24).
**return** $Q, \pi$.

$\tilde{\pi}(s) = \arg\max_{a \in A_s} Q(s,a)$. However, as outlined, one usually uses a less greedy approach when choosing the next action for exploration, e.g., the $\varepsilon$-greedy strategy or the softmax exploration.

Furthermore, comparing Algorithm 15 and Algorithm 16 with policy iteration from Algorithm 13, we note that SARSA and Q-learning can be considered as *optimistic* policy iteration methods, where only a single sample is obtained between policy updates instead of a whole trajectory; see also [Ber19]. Note that numerical issues such as cyclic or oscillating policies can occur in SARSA or Q-learning. For further details on such numerical problems and their stabilization by using joint updates after several state-action observations we refer the reader to [Ber12, Ber19].

### 9.2.6 ▪ Continuous state spaces and binning

So far we have been considering only discrete state spaces $\mathcal{S}$ and discrete action spaces $A$. To deal with continuous spaces, which are often encountered in real-world applications (e.g., autonomous driving and robotics), we now introduce *binning* for their discretization. Binning refers to a division of a space (or domain) into cells which represent all the continuous values that reside in that cell. For example, we can discretize the domain $[0, 1]$ into

$$\Upsilon \coloneqq \{[0, 0.1), [0.1, 0.2), \dots, [0.8, 0.9), [0.9, 1]\}$$

to obtain ten different bins. Then, after the binning process, we obtain indeed a problem with discrete state and action spaces by defining a Markov decision process (and the corresponding policies and Q-functions) on the set of bins instead of the continuous spaces. This allows us to treat the resulting problem with the reinforcement learning algorithms that we have already introduced in the previous sections. For a specific continuous state, we then simply look up the value of the corresponding state bin, e.g., for the state $0.23$ in the above example, we take the value assigned to the third bin $[0.2, 0.3)$. Discrete Q-functions are often called *Q-tables* to clearly differentiate between the discrete and the continuous approaches.

In the same way as above, we can use binning for a two-dimensional domain. For example, we can discretize $[0,1]^2$ into 10 times 10 bins $\Upsilon \times \Upsilon$. When proceeding in this way for more and more dimensions, the number of bins increases exponentially with the dimension, and we encounter the *curse of dimensionality* for high-dimensional problems; see also Section 1.6. However, in many practical applications, we encounter low-dimensional state and action spaces, which can be treated very well by a combination of binning and RL algorithms like SARSA or Q-learning, as we will see in the following task.

### 9.2.7 ▪ Tasks on SARSA and Q-learning

In the following task, we use the already familiar FrozenLake-v1 environment to test the SARSA and Q-learning algorithms. Furthermore, we explore a new environment, namely CartPole-v1.

**Task 9.2.** Experiment with the reinforcement learning algorithms introduced in this section. To this end, refer to the provided JUPYTER notebook template from Task 9.1.

(a) Implement SARSA or Q-Learning to solve the FrozenLake-v1 environment with `is_slippery=True` and $\gamma = 0.9$. Use an $\varepsilon$-greedy action-selection strategy with $\varepsilon = 0.1$. Estimate the expected value of this policy experimentally.

(b) We now use the CartPole-v1 environment from `gymnasium`. Here, the environment resembles a cart to which a pole is attached. The cart can move to the left and to the right. The goal is to balance the pole by accelerating the cart in a chosen direction at any given point in time. Get familiar with the environment by implementing a policy which picks admissible random actions. Traverse a trajectory of this policy and render/visualize it with the help of the `gym.make` method with `render_mode="human"`.

(c) Adapt the algorithm from (a) to solve the CartPole-v1 problem. Use binning to discretize the state space. Experimentally confirm the expected return for the learned policy and render/visualize one trajectory.

## 9.3 ▪ Deep reinforcement learning

With the help of deep neural networks, as introduced in Chapter 6, we can construct even more powerful reinforcement learning algorithms than those in the previous sections. To this end, we will have a closer look at possibilities to combine neural network model classes with reinforcement learning algorithms in the following. Albeit the term *deep reinforcement learning* is not clearly defined, we use it here to denote methods whose minimal building blocks are deep neural networks which work on a Markov decision process. First impressive results in this direction have been the effective use of so-called *deep Q-learning*, which we will describe in more detail later, for playing Atari games [Mea15], or of Monte Carlo-type approaches for determining board game moves, e.g., in chess and Go [Sea18], on a (super)-human level.

While bots that play (computer) games are not a new invention, these results are still remarkable since the only feedback that the reinforcement learning algorithm gets is the score or outcome of the game, respectively. Its input and output options resemble those a human player would have when looking at the screen/board. Thus, the learning process resembles how a human would tackle a new, complex, big, and uncertain problem. Earlier bots have usually been programmed to perform specific actions in specific situations by using internal data to reflect the state.

### 9.3.1 ▪ Continuous Q-functions and their approximation

**Q-function approximation** Besides binning, as introduced in Section 9.2.6, another approach to deal with large or continuous state spaces $\mathcal{S}$ is to use a cost-efficient approximation $Q^\pi_{\boldsymbol{\theta}}$ to the real Q-function $Q^\pi$ for a trajectory $\pi$. Here, a model function $Q^\pi_{\boldsymbol{\theta}}$, which is parametrized by $\boldsymbol{\theta} = (\theta_1, \dots, \theta_d)^T \in \mathbb{R}^d$, is chosen such that it can be easily stored and evaluated. Note that this approach is a good alternative to binning in the case of high-dimensional state spaces if $Q^\pi_{\boldsymbol{\theta}}$ is chosen in such a way that it does not suffer from the curse of dimensionality. The question is of course what a good model will be. To this end, we aim for a model which is computationally efficient and achieves only a small approximation error. A classic choice is

$$Q_{\boldsymbol{\theta}}(\cdot,\cdot) \coloneqq \sum_{k=1}^{d} \theta_k f_k(\cdot,\cdot), \tag{9.27}$$

which is linear in $\boldsymbol{\theta}$ and employs a set of $d$ maps $f_k$, which are usually chosen to be simple projections, (Fourier) polynomials, or piecewise constant functions.[48] If we assume that $f_k$, $k = 1, \dots, d$, are fixed functions, we now only need to store the vector $\boldsymbol{\theta} \in \mathbb{R}^d$ to represent $Q_{\boldsymbol{\theta}}$. This has to be compared to $|S| \cdot |A|$ values for a standard Q-function $Q$. Such an approximation $Q_{\boldsymbol{\theta}}$ then just has to be plugged in for $Q$ in Algorithm 15, for instance. Furthermore, infinite state spaces $S$ could also be dealt with straightforwardly when using the approximation (9.27) with fixed $f_k$, $k = 1, \dots, d$.

**Update of Q-function approximations** To work with Q-function approximations like (9.27) in SARSA or Q-learning, the update steps (see line 8 of Algorithm 15 and line 7 of Algorithm 16) need to be properly adapted. While we could try to find a $\boldsymbol{\theta} \in \mathbb{R}^d$ such that $Q_{\boldsymbol{\theta}}(s,a)$ is equal to the new value at $(s,a)$, this approach might significantly worsen the approximation at other state-action pairs $(\bar{s}, \bar{a})$. Instead, we will determine

$$\arg\min_{\boldsymbol{\theta}} \|Q - Q_{\boldsymbol{\theta}}\|_*$$

in some appropriate norm $\|\cdot\|_*$. To this end, let us consider the example of a least squares minimization, i.e., we aim to minimize

$$\|Q - Q_{\boldsymbol{\theta}}\|_* \coloneqq \frac{1}{2} \sum_{s\in\mathcal{S}, a\in A_s} \left(Q(s,a) - Q_{\boldsymbol{\theta}}(s,a)\right)^2.$$

Since we do not have access to the true Q-function $Q$, we instead use the updated target value $r(s,a,s') + \gamma \max_{a'} Q_{\boldsymbol{\theta}}(s',a')$ with $s' \sim P_a(s,\cdot)$ for a state-action pair $(s,a)$ to obtain

$$\|Q - Q_{\boldsymbol{\theta}}\|_* \approx \|r(\cdot,\cdot,S') + \gamma \max_{a'} Q_{\boldsymbol{\theta}}(S',a') - Q_{\boldsymbol{\theta}}\|_*,$$

where $S'(s,a)$ is a random variable distributed according to $P_a(s,\cdot)$. Now, we employ one step of gradient descent to move towards a minimizer of this expression, i.e.,

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \sum_{s\in\mathcal{S}, a\in A_s} \Big(r(s,a,S'(s,a)) + \gamma \max_{a'} Q_{\boldsymbol{\theta}}(S'(s,a),a') - Q_{\boldsymbol{\theta}}(s,a)\Big) \nabla_{\boldsymbol{\theta}} Q_{\boldsymbol{\theta}}(s,a) \tag{9.28}$$

with a learning rate $\alpha > 0$.

[48] In practice, one often handcrafts these $f_k$, $k = 1, \dots, d$. Then, $f_k$ assigns a pre-defined heuristic value to a state-action pair. In chess, for example, a weighted sum of the number of pieces that one possesses could be employed for $f_1$. Next, $f_2$ could be a function which evaluates to a positive value if a certain board configuration is present and to zero otherwise. Choosing $d-2$ further heuristics like that would lead to $f_3, \dots, f_d$.

**Algorithm 17:** Episodic semi-gradient SARSA

**Input:** deterministic MDP $(\mathcal{S}, A, T, R, \gamma)$, number of trajectories $N > 0$, step size $\alpha > 0$, action-selection strategy $\tilde{\pi}$ (e.g., $\varepsilon$-greedy).
**Output:** (approximations to) the optimal Q-function $Q^*$ and an optimal policy $\pi^*$.

Initialize $\boldsymbol{\theta} \in \mathbb{R}^d$ arbitrarily.
**for** $i \in \{0, \dots, N\}$ **do**
  Initialize a starting state $s \in S$ randomly.
  Select $a$ according to $\tilde{\pi}(s)$.
  **repeat**
    Take action $a$, sample next state $s'$ according to $P_a(s, \cdot)$, and observe reward $r = r(s, a, s')$.
    **if** $s'$ is terminal **then**
      $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \left(r - Q_{\boldsymbol{\theta}}(s, a)\right) \nabla Q_{\boldsymbol{\theta}}(s, a)$.
    **end if**
    **else**
      Select $a'$ according to $\tilde{\pi}(s')$.
      $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \left(r + \gamma Q_{\boldsymbol{\theta}}(s', a') - Q_{\boldsymbol{\theta}}(s, a)\right) \nabla Q_{\boldsymbol{\theta}}(s, a)$.
      $s \leftarrow s'$; $a \leftarrow a'$.
    **end if**
  **until** $s'$ is terminal.
**end for**
Derive policy $\pi$ from $Q_{\boldsymbol{\theta}}$ by the greedy rule (9.24).
**return** $Q_{\boldsymbol{\theta}}, \pi$.

Since it would be way too costly to evaluate (9.28) several times after each Q-function update step, we employ the minibatch idea of stochastic gradient descent; see Algorithm 10. Here, when we are currently in state $s \in \mathcal{S}$ and take action $a \in A_s$, we only use this specific state-action pair $(s, a)$ in (9.28), i.e., we compute

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \Big( r(s, a, S'(s, a)) + \gamma \max_{a'} Q_{\boldsymbol{\theta}}(S'(s, a), a') - Q_{\boldsymbol{\theta}}(s, a) \Big) \nabla_{\boldsymbol{\theta}} Q_{\boldsymbol{\theta}}(s, a). \tag{9.29}$$

A SARSA variant with a Q-function approximation based on such an update step is given in Algorithm 17. Note that, to resemble an SGD algorithm with minibatch size one, we would need that $(s, a)$ is an i.i.d. sample of all state-action pairs. But depending on the underlying MDP and the chosen actions, this might not be the case. However, in practice this strategy still works well. In fact, for the Q-function approximation (9.27), convergence and error bounds can be shown under certain assumptions; see [SB18].

### 9.3.2 ▪ Deep Q-networks

In Chapter 6 we have seen that deep neural networks are a well-suited model class for function approximation. Thus, besides an explicit approximation with fixed maps as in (9.27), we may resort to a deep neural network instead. This leads to Deep Q-Networks (DQN) for RL. The basic approach of Deep Q-Networks is the same as in Q-learning (see Algorithm 16), but the optimal Q-function $Q^*$ is now approximated by a properly chosen deep neural network. Note that, in contrast to Algorithm 17, where (9.27) was used to approximate $Q^*$, we now use a different approximation from the optimal Q-function $Q^*$ in DQN. In particular, a network architecture

**Algorithm 18:** Deep Q-learning

**Input:** deterministic MDP $(\mathcal{S}, A, T, R, \gamma)$, number of trajectories $N > 0$, batch size $n_{\text{batch}}$, step size $\alpha > 0$, action-selection strategy $\tilde{\pi}$ (e.g., $\varepsilon$-greedy).
**Output:** (approximations to) the optimal Q-function $Q^*$ and an optimal policy $\pi^*$.

Initialize $\boldsymbol{\theta} \in \mathbb{R}^d$ arbitrarily.
**for** $i \in \{0, \dots, N\}$ **do**
  Initialize a starting state $s \in S$ randomly.
  **repeat**
    Select $a$ according to $\tilde{\pi}(s)$.
    Take action $a$, sample next state $s'$ according to $P_a(s, \cdot)$, and observe reward $r = r(s, a, s')$.
    Store the transition $(s, a, r, s')$ in the replay memory.
    Sample a batch $(s_j, a_j, r_j, s'_j)_{j=1}^{n_{\text{batch}}}$ from replay memory.
    **for** $j \in \{1, \dots, n_{\text{batch}}\}$ **do**
      $$y_j = \begin{cases} r_j & \text{if } s'_j \text{ is terminal,} \\ r_j + \gamma \max_{a'} Q_{\boldsymbol{\theta}}(s'_j, a') & \text{else.} \end{cases}$$
    **end for**
    Perform a gradient descent step on (9.30).
    $s \leftarrow s'$.
  **until** $s'$ is terminal.
**end for**
Derive policy $\pi$ from $Q_{\boldsymbol{\theta}}$ by the greedy rule (9.24).
**return** $Q_{\boldsymbol{\theta}}, \pi$.

is fixed, i.e., the layers, nodes, and activation functions are chosen a priori. Then, the function $f$, which the corresponding network represents, is chosen as approximation $Q_{\boldsymbol{\theta}}$ to the optimal Q-function $Q^*$. Here, the parameter $\boldsymbol{\theta} = (\boldsymbol{W}, \vec{b})$ contains the weights and biases of the network. The resulting method is summarized in Algorithm 18. To train the neural network, a database of past steps called *replay memory* is used, which needs to have been collected and stored beforehand. Let us explain this in more detail next.

**Experience replay** To obtain less correlated samples than those obtained by just taking the most recent trajectory information, a technique called *experience replay* is used in Algorithm 18. Here, the observed transitions $(s,\ a,\ r,\ s')$ are collected in a replay memory. Each time a trajectory progresses, we draw $n_{\text{batch}}$ uniformly distributed samples $(s_j, a_j, r_j, s'_j)_{j=1}^{n_{\text{batch}}}$ from the replay memory, where the batch size $n_{\text{batch}} \in N$ is fixed a priori. Then, the targets $y_j := r_j + \max_{a' \in A_s} Q_{\boldsymbol{\theta}}(s', a')$ are computed for each transition $(s_j, a_j, r_j, s'_j)$ for $j = 1, \dots, n_{\text{batch}}$. Finally, we perform a gradient descent step using the loss

$$\frac{1}{n_{\text{batch}}} \sum_{j=1}^{n_{\text{batch}}} (y_j - Q_{\boldsymbol{\theta}}(s_j, a_j))^2. \tag{9.30}$$

The overall approach is given in Algorithm 18 in more detail.

**Decoupled targets** A variant of Algorithm 18 uses so-called *decoupled targets* to improve stability. Here, the targets $y_i$ are not computed using $Q_{\boldsymbol{\theta}}$, but instead a second, independent

instance of *target weights* $\hat{\boldsymbol{\theta}}$ is used to calculate

$$y_i = r_i + \gamma \max_{a_i'} Q_{\hat{\boldsymbol{\theta}}}(s_i', a_i'). \tag{9.31}$$

This can be seen in analogy to picking an action-selection strategy and, separately, a greedy update step for the Q-function. Here, we employ an action-selection strategy $\tilde{\pi}$, and the Q-function approximation is updated using a gradient descent algorithm based on the targets (9.31). A similar approach is followed in double Q-learning [vH10]. Here, two different Q-functions are used to reduce the expected error when determining the value of the next state.

**Task 9.3.** Let us now apply the deep reinforcement learning algorithms introduced in this section to the CartPole-v1 environment introduced in Task 9.2. To this end, refer again to the template JUPYTER notebook from Task 9.1.

(a) Use episodic SARSA (Algorithm 17) in the CartPole-v1 environment to balance the pole. To this end, use linear Q-functions to represent the value of each action $a$, i.e.,

$$Q_{\boldsymbol{\theta}}(s, a) := \sum_{k=1}^{d} [\boldsymbol{\theta}_a]_k \, [s]_k = \boldsymbol{\theta}_a^T s,$$

where $s \in \mathbb{R}^d$ is the vector representing the current state and $\boldsymbol{\theta}_a \in \mathbb{R}^d$ is the parameter vector for action $a \in A$. Employ an $\varepsilon$-greedy action-selection strategy with $\varepsilon = 0.9$. Experimentally estimate the expected return for the learned policy.

(b) Use deep Q-learning (Algorithm 18) in the CartPole-v1 environment to balance the pole. To this end, employ a deep neural network to approximate the Q-function. You can use the fully connected network, which we provided in the template notebook. The network has $|\mathcal{S}|$ many input neurons and $|A|$ many output neurons. Thus, the value at the $j$th output neuron for input $s \in \mathcal{S}$ represents the Q-function value $Q(s, a_j)$ for $j = 1, \ldots, |A|$. Employ the same action-selection strategy as in task (a) and experimentally estimate the expected return for the policy determined by deep Q-learning.

(c) Pick another one of the `gymnasium` environments and try to solve the corresponding tasks with one of the algorithms developed in task (a) or (b). You may need to change the network's hyperparameters. Moreover, you can implement additional tweaks like a decaying $\varepsilon$-strategy or decoupled targets and perform experiments with that as well.

## 9.4 ▪ Further topics

**Convergence results** In general, convergence results for the algorithms in optimal control and reinforcement learning are based on contraction properties of the iterative update steps. When these properties are present, the algorithm exhibits a unique solution that can be computed iteratively by fixed point iteration according to Banach's fixed point theorem [Ber22, SB18]. However, it is quite complicated or even impossible to show such contraction properties in the reinforcement learning setting. This is due to the involved sampled trajectories and the function approximation needed for intricate state spaces. Regarding the sampling, a common assumption is that all states, or state-action pairs, are regularly visited and the value function, or Q-function,

is updated accordingly [Ber12, Ber22, SB18]. This can however be infeasible in practice even for moderately large state spaces. In general, the analysis of reinforcement learning algorithms is said to be difficult, since "it relies on multiple interacting approximations whose effects are hard to predict and quantify in practise" [Ber19].

Overall, there are two contrasting aspects to consider when we train an RL system. First, from a theoretical point of view, we typically need to guarantee that every state-action pair is regularly visited to show convergence. Second, from an efficiency point of view, we aim to focus the limited learning resources on those situations that happen more often and lead to a high reward, while we mostly ignore theoretically possible but essentially irrelevant states. In general, we aim for an estimation of expected values under one distribution, reflecting the policy which we want to learn, while we obtain samples from another distribution, reflecting the action-selection strategy or other decisions affecting the sampled trajectory. Here, returns of the observed trajectories can be weighted according to the relative probability of the two distributions; see [SB18]. Additionally, sampling large reward terms more often can increase the sample efficiency; see [Ber12].

**$n$-step learning** The basic TD method (9.18) can be generalized to use *$n$-step learning*. To this end, the trajectory is observed for $n$ steps after the current state $s_i$, and the $n$-step discounted cumulative reward is employed together with an estimate of $V(s_{i+n})$ to obtain

$$R_{i:i+n} := \sum_{k=0}^{n-1} \gamma^k r(s_{i+k}, a_{i+k}, s_{i+k+1}) + \gamma^n V(s_{i+n}).$$

After observing $s_{i+n}$, the value $V(s_i)$ is updated by the $\alpha$-weighted difference between the $n$-step reward $R_{i:i+n}$ and the current estimate for $V(s_i)$, i.e.,

$$V(s_i) \leftarrow V(s_i) + \alpha \left( R_{i:i+n} - V(s_i) \right).$$

One can consider $n$-step methods as a generalization of TD and MC methods. Here, the one-step TD approach (9.18) with $n = 1$ is at one end of the scale, whereas the MC update (9.20) is at the other end with $n = l$ being the length of a whole trajectory.

The $n$-step idea is also used to improve deep Q-learning. In [HMvH$^+$18], for instance, several variants of Deep Q-networks were combined and tested for an arcade game environment. There, one major improvement over the basic DQN was attributed to $n$-step learning, where, instead of considering the reward of one step to compute a target $y = r + \gamma \max_{a'} Q(s', a)$, $n$ steps are combined to compute $y = \sum_{j=1}^{n} \gamma^{j-1} r_j + \gamma^n \max_{a_{n+1}} Q(s_{n+1}, a_{n+1})$; see also [Ber19, GP17, SB18] for more details.

**TD($\lambda$)** A further generalization of TD methods involves so-called *$\lambda$-returns*. Here, both extremes, the basic TD update and the MC update, are combined. By using exponentially decaying weights $(1 - \lambda)\lambda^{k-1}$ for $k \in \mathbb{N}$ with a parameter $\lambda \in (0, 1)$, we define the $\lambda$-return

$$R_i^\lambda := (1 - \lambda) \sum_{k=1}^{\infty} \lambda^{k-1} R_{i:i+k}.$$

The method using this value in the update is called TD($\lambda$). Note that one recovers the one-step TD-algorithm for $\lambda = 0$. Furthermore, one can show that $\lambda \to 1$ represents a Monte Carlo-type algorithm which is just slightly more general than Algorithm 14; see [SB18]. Moreover, one can show that one can adjust the tradeoff between bias and variance of the update by varying $\lambda$. This has a large influence on the speed of convergence of the corresponding algorithm. Finally, to obtain a memory-efficient policy evaluation method with $n$-step TD or TD($\lambda$) updates, one has to employ a special technique called *eligibility traces*. For more information on this technique and the corresponding algorithms, we refer the interested reader to [Ber12, SB18, vSMP$^+$16].

**Scalability of reinforcement learning algorithms** Another area of research aims at reducing computation time for RL algorithms. To this end, parallel and distributed methods are being developed. Here, a policy is computed using multiple machines (many CPUs, GPUs, TPUs, etc.) [MWG+21, Sea18]. Furthermore, to increase the sample efficiency, simulation data are used to pre-train a policy such that fewer samples are needed in the actual training algorithm [Dea22, ZQW20]. This makes reinforcement learning more applicable in non-simulated environments like robotics, where training without simulation data is too costly.

**Policy gradient methods** Note finally that there exist reinforcement learning algorithms that directly work with continuous action spaces, e.g., *policy gradient* methods [SMSM99]. The basic idea of policy gradient methods is to approximate a policy $\pi$. One example of a policy gradient method would be to use the softmax action-selection strategy from Section 9.2.5 and to update the functional representation of the softmax distribution by an (approximate) gradient ascent method according to a given scalar performance measure; see [Ber19, SB18]. Note that this approach is only valid if the functional representation of the softmax distribution is differentiable. Modern approaches, such as *proximal policy optimization* (PPO) [SWD+17], employ a deep neural network $\pi_{\boldsymbol{\theta}}$ as an approximation to $\pi$. Then, the network is optimized with respect to a chosen measure for the reward. Approaches that approximate the value function in addition to the policy are often called *actor-critic methods*. Here, *actor* refers to the policy that provides the action, and *critic* refers to the value function that evaluates the given action.

**Further literature** To obtain detailed insights into the theoretical aspects of reinforcement learning, we refer the interested reader to [Ber12, Ber17, Ber19, Ber23, Sze10]. Furthermore, an overview is given in [SB18]. In [Pow22], a unified framework for common sequential decision problems is proposed.

# Chapter 10
# Further Developments

This chapter serves as a rough overview on some important machine learning topics that we have not covered so far. Since they are quite relevant in data science but a thorough treatment of them would be beyond the scope of this book, we decided to present these techniques in short introductional overviews in this chapter without going into too much detail. We will not present any tasks regarding the following topics, but we encourage the interested reader to check out the given references to learn more in these directions.

In the following, in Section 10.1 we first encounter machine learning problems where the data is non-vectorial or the labels are non-scalar. Subsequently, we introduce ensemble methods in Section 10.2. These techniques combine several ML models to create a more sophisticated one. Recurrent neural networks, which deal with sequence data, are discussed in Section 10.3. Section 10.4 deals with the transformer network, which presently resembles the state of the art when it comes to sequence models in deep learning. Finally, we conclude this chapter in Section 10.5 with a digression into the topic of interpretability.

## 10.1 ▪ Intricate data structures

When considering supervised and unsupervised learning problems, we mostly dealt with vector-valued *data* points, i.e., $\Omega \subseteq \mathbb{R}^d$, in this book. Moreover, in supervised learning problems, we have usually only considered scalar-valued data *labels*, i.e., $\Gamma \subseteq \mathbb{R}$, so far. However, we want to emphasize in this section that non-vector-valued data and non-scalar-valued labels are of particular interest in several areas of application.

An important aspect that arises when taking into account non-scalar labels is the choice of an appropriate loss function since this can be quite complicated in this case. For instance, when dealing with probability densities as labels, *Kullback–Leibler divergences* or *Wasserstein distances* are possible choices; see Section 7.4 and [FZM+15, HWVG19]. Essentially, each distance measure can theoretically be used within the loss function and there exists a huge number of possible choices; see [DD09]. However, building a machine learning algorithm to successfully minimize the loss function for an arbitrary distance measure and especially for non-scalar-valued labels is usually not straightforward and can result in various issues for the respective optimization routines. Note finally that, when employing appropriate loss functions and the methodologies discussed in the following, the algorithms from this book can often be applied when dealing with intricate data structures as well.

### 10.1.1 ▪ Vector-valued data labels

While we have encountered vector-valued data points throughout this whole book, the question remains how to deal with vector-valued labels, i.e., with the setting $\Gamma \subseteq \mathbb{R}^{\tilde{d}}$ for some $\tilde{d} > 1$. Note that most of the mechanisms of Section 1.3 carry over to this case. Without any additional structural constraints on the data, scalar-valued model classes can be used separately for each coordinate $j = 1, \dots, \tilde{d}$. The resulting model class for the vector-valued problem is then just a combination of $\tilde{d}$ scalar-valued classes. However, employing more complicated model spaces can be meaningful, advantageous, and more effective in certain scenarios. For instance, in the realm of kernel-based methods (see Section 3.5), matrix-valued kernels can be employed, which lead to vector-valued model classes for which the covariances between the label dimensions are directly reflected by the kernel choice; see, e.g., [MP05].

Furthermore, many loss functions can be generalized to the vector-valued label case by employing corresponding vector norms. For instance, the common least squares loss becomes

$$\mathcal{L}_{\text{ls}}\left((\boldsymbol{z}_1, \tilde{\boldsymbol{z}}_1), \dots, (\boldsymbol{z}_n, \tilde{\boldsymbol{z}}_n)\right) := \frac{1}{n} \sum_{i=1}^{n} \|\boldsymbol{z}_i - \tilde{\boldsymbol{z}}_i\|_2^2,$$

i.e., we included the Euclidean norm to obtain a scalar quantity from the differences of the vector-valued data labels. We dealt with this particular setting already in Section 4.1 and Section 7.1 when discussing the PCA and autoencoders. Technically, however, we did not have vector-valued data labels in these cases since we only considered unsupervised learning problems there. Nevertheless, the reasoning behind the chosen model classes and loss functions is essentially the same as for vector-valued labels in supervised learning problems. Alternatively, and depending on the situation at hand, more involved vector norms than the $\|\cdot\|_2$ norm can be employed there.

### 10.1.2 ▪ Matrix- and manifold-valued data and labels

While matrix- (or even tensor-)valued data can just be interpreted as vector-valued data with large dimension, it might be sensible to directly work with matrix-valued model classes and the corresponding loss functions, especially when the matrices should have a specific structure. For instance, for symmetric, positive semi-definite (spd) matrix-valued data, e.g., when the model's output is a covariance matrix, specific loss functions have to be used. The reason for this is that the class of these matrices forms a manifold instead of a plain vector space (or even a Hilbert space); see, e.g., [JHS+13, MM18]. Distances should then be measured on the manifold and not just in the surrounding Euclidean space.

Covariance matrices play an important role in the representation of image and video data, for instance. In particular, an image can be described by a matrix of a fixed number of features (e.g., colors, intensities, filters, and gradients thereof) at each pixel. To this end, assume that $\boldsymbol{x}_i \in \mathbb{R}^d$ contains the $d$ feature values for pixel $i \in \{1, \dots, n\}$. Then, after computing the centered pixel feature matrix

$$\boldsymbol{X} := (\boldsymbol{x}_1 - \boldsymbol{m} \cdots \boldsymbol{x}_n - \boldsymbol{m}) \in \mathbb{R}^{d \times n}$$

with the empirical mean $\boldsymbol{m} := \frac{1}{n} \sum_{i=1}^{n} \boldsymbol{x}_i$, we can represent an image by the feature covariance matrix

$$\boldsymbol{C} := \frac{1}{n} \boldsymbol{X} \boldsymbol{X}^T.$$

Now, to compare images to each other, we can use norms and distances on the manifold of covariance matrices instead of having to deal with the high-dimensional pixel space. For details, we refer the reader to [MM18].

Furthermore, brain computer interfaces are another field of application where data are commonly represented as a covariance matrix. In particular, in the context of electroencephalography

(EEG), the corresponding time series data is represented as a matrix, which is then cast into a covariance matrix format; see, e.g., [CKBM21, CBB17].

Besides covariance matrices, symmetric positive-definite matrix-valued data also appear in the medical area of *diffusion tensor imaging* (DTI), for example; see [PFA06, JHS+13, PSF19] for an ML approach to DTI. The ultimate goal of DTI is to generate contrast in magnetic resonance images by determining the so-called *diffusion tensor* $\boldsymbol{T} \in \mathbb{R}^{3\times 3}$ for an anisotropic diffusion process, i.e., we learn the diffusion coefficient in Fick's first law

$$\boldsymbol{J}(\boldsymbol{x}) = \boldsymbol{T}(\boldsymbol{x})\nabla \boldsymbol{c}(\boldsymbol{x})$$

from given flow data. Here, $\boldsymbol{J}$ is the diffusion flux vector and $\nabla \boldsymbol{c}$ denotes the (spatial) gradient of the particle concentration. Employing a *log-Euclidean* distance

$$\operatorname{dist}(\boldsymbol{T}_1, \boldsymbol{T}_2) := \| \log(\boldsymbol{T}_1) - \log(\boldsymbol{T}_2)\|_F$$

in the loss function has proven to be both efficient and successful for DTI; see also [PSF19]. Here, $\log(\cdot)$ denotes the matrix logarithm and $\| \cdot \|_F$ is the Frobenius norm. Besides the log-Euclidean distance, there also exist other, more involved, choices, e.g., affine-invariant metrics, Cholesky distances, and Stein divergences; see [JHS+13] for details.

Another area of application where spd matrices are to be learned is robotics. In particular, when we deal with a robotic arm, the configuration[49] $\boldsymbol{x}(t) \in \mathbb{R}^6$ at time $t$ of the *end effector*, i.e., the end of the robotic arm, can be expressed in terms of the configurations $\boldsymbol{\theta}(t) \in \mathbb{R}^m$ of the joints of the robotic arm via

$$\boldsymbol{x}(t) = f(\boldsymbol{\theta}(t))$$

for some function $f : \mathbb{R}^m \to \mathbb{R}^6$. Here, $m$ depends on the number and the type of robot joints. To obtain the corresponding velocities, we compute the time derivative

$$\dot{\boldsymbol{x}}(t) = \frac{\partial f}{\partial \boldsymbol{\theta}}(\boldsymbol{\theta}(t))\dot{\boldsymbol{\theta}}(t).$$

Now the sphere of joint velocities $\dot{\boldsymbol{\theta}}(t)^T\dot{\boldsymbol{\theta}}(t)$ can be expressed as

$$\dot{\boldsymbol{\theta}}(t)^T\dot{\boldsymbol{\theta}}(t) = \dot{\boldsymbol{x}}(t)^T \left(\boldsymbol{J}(\boldsymbol{\theta}(t))\boldsymbol{J}(\boldsymbol{\theta}(t))^T\right)^{-1} \dot{\boldsymbol{x}}(t),$$

where $\boldsymbol{J} = \frac{\partial f}{\partial \boldsymbol{\theta}}$ denotes the Jacobian of $f$ with respect to the joint configurations. Then, the spd matrix $(\boldsymbol{J}\boldsymbol{J}^T)^{-1}$ defines the so-called *manipulability ellipsoid*, i.e., the end effector velocity ellipsoid corresponding to the joint velocity sphere $\dot{\boldsymbol{\theta}}^T\dot{\boldsymbol{\theta}}$. This is a cardinal structure since it describes the possible movement directions of the end effector at any given time $t$. Therefore, learning it from a set of given robot arm trajectories is an important task, e.g., when teaching a robot to track a reference motion; see [ADHSK21, JRCC21]. A similar machine learning problem with manifold-valued labels is encountered when attempting to recover robot orientation data represented by rotation matrices from the Lie manifold $\mathcal{SO}(3)$, i.e., the special orthogonal group in three dimensions; see [ZHS+17].

Besides the above applications, manifold-valued data also appear in the context of image synthesis when the corresponding image is characterized not by RGB color values but by different, more intricate representations; see, e.g., [HWVG19]. Furthermore, when processing manifold-valued data, e.g., matrices on *Stiefel* manifolds or *Grassman* manifolds, by neural networks which keep their structure intact, similar challenges are encountered as with manifold-valued labels; see [CBMV20].

[49] This is usually a combination of three spatial coordinates and three orientations.

### 10.1.3 ▪ Graph-valued data and labels

The research area of *graph learning* considers graphs as data points, whether as inputs or data labels; see Section 5.1 for the definition of a graph. For an overview on graph learning techniques, we refer the reader to [CAEHP+22]. Machine learning approaches with graph-valued data are of particular interest whenever a graph representation makes sense for the given data. For instance, in molecular biology and chemistry, the task of molecule structure prediction can be tackled with specialized machine learning algorithms considering a molecule as a graph; see [BBS21, BMFB+22]. Furthermore, graph-labeled data are used in brain signal analysis [DTRF19] when exploring relationships between certain areas of the brain. An approach for regression with graph-valued labels for the visualization of correlations in finance applications can be found in [CFV22]. Finally, the more general problem of graph generation with given input properties is an important research area of its own; see [BSZG18, SK18, YYR+18].

### 10.1.4 ▪ Function-valued data and labels

Besides manifold-valued and graph-valued data, infinite-dimensional data that represent functions are of particular interest in the area of *functional data analysis* (see [RS05, RS07]) and in *operator learning* (see [KDP+16, KLL+23]). In these areas, a functional $\mathcal{F} : V_1 \to \mathbb{R}$ or even an operator $\mathcal{O} : V_1 \to V_2$ is approximated, where $V_1$ and $V_2$ are appropriate (potentially infinite-dimensional) function spaces. This has to be seen in contrast to standard regression, where the input and output spaces are usually of finite dimension.

An important area where operator learning is useful is *uncertainty quantification*. Here, many solutions of parametrized partial differential equations, for instance, have to be computed in order to approximate a quantity of interest. Instead of building an algorithm to learn the solution for a fixed set of parameters, it makes sense to directly learn the solution operator corresponding to the parametrized equation instead. Here, deep learning approaches are especially important; see [KLM21, LMK22, LKA+20, LJP+21].

### 10.1.5 ▪ Non-numerical data and labels

To deal with non-numerical data, a feature map $\phi : \Omega \to \mathbb{R}^d$ is usually employed to map each data point to a numerical vector; see also Section 1.5 and Section 3.5.1. These maps are often handcrafted according to the specific machine learning problem. For instance, for categorical data, so-called *one-hot-encoding* maps or specific hash maps are applied. However, more sophisticated approaches based on a feature's distribution have also been successfully employed; see [HK20] for details on one-hot-encoding, hash maps, and a survey on other feature maps for categorical data.

If the data stem from a finite set of possible values, i.e., a dictionary, embedding techniques can be combined with one-hot-encoding to determine an appropriate latent space distribution of the data set; see, e.g., [LLHZ16] and Section 10.4 for word embedding feature maps. When employing a kernel-based method, e.g., an SVM analogous to that in Chapter 3, an appropriate kernel function can be employed instead of a corresponding feature map; see, e.g., [LSST+02].

Finally, note that the application of a feature map is not always necessary since there exist machine learning approaches which can be applied directly to non-numerical data. Examples for such methods are classification and regression trees (CART) and random forests, which can be altered in such a way that they also work for categorical data. We refer the interested reader to Section 10.2.5 for an introduction to decision trees and random forests and to [Au18, HTF09] for more details on their usage for categorical data.

### 10.1.6 ▪ Sequential data and labels

In the areas of time series analysis and natural language processing, one encounters sequential data, i.e., a single data point $\boldsymbol{x}$ is given as a sequence

$$\boldsymbol{x} = (\boldsymbol{a}_1, \ldots, \boldsymbol{a}_m)$$

of $m \in \mathbb{N}$ data points $\boldsymbol{a}_1, \ldots, \boldsymbol{a}_m$, which can be vectorial or even non-numerical themselves, for instance. Here, the two common approaches are either to use a feature map to transform (sub)sequences to vectors or to employ a model class which can deal with sequential data directly.

The feature map approach often consists of applying functions which highlight both spatial and temporal information, e.g., Fourier/Wavelet transformations [RBAF+19] or signature transformations [KBPA+19]. Besides, *delay embedding* techniques based on Takens' theorem are quite common [SDB16, XTL09]. Here, parts of the sequence are stacked into high-dimensional vectors; see [Tak81] for details. In recent years, the second approach of employing model classes which operate directly on sequential data became more popular. The reason for this is specialized deep neural network architectures like *long short-term memory* networks and *transformers*, which often outperform classical feature map approaches. We will consider these architectures in more detail in Section 10.3.2 and Section 10.4.

## 10.2 ▪ Ensemble learning

This section is dedicated to the more general topic of *ensemble methods*. Here, we will briefly touch on the most important mechanisms and techniques in ensemble learning.

**Ensemble learning**

The simple but powerful idea behind ***ensemble learning*** is to combine the results of several basic machine learning models to create a more complex and accurate model. Results achieved in this way can be superior to those achieved by each of the single models alone. Most ensemble learning methods rely on a combination of *bootstrapping*, *boosting*, and *stacking* mechanisms. We refer the reader to [DYC+20, HTF09, JWHT21, Mur22, SR18], for details.

### 10.2.1 ▪ Voting and averaging

Assume that we have trained $m$ different machine learning models $M_1, \ldots, M_m$ called *base models* on a certain problem for a given training data set. We now want to use the outputs of all these base models to compute an output for the whole ensemble of models when a new data point is presented. To this end, e.g., for a classification problem, one can first consider very simple approaches like *voting*. There, the result of the ensemble model is the *majority* of the class outcomes of the base models. An example illustration is given in Figure 10.1. Similarly, for a regression problem, a simple *average* of the results can be chosen as the overall result.

### 10.2.2 ▪ Stacking

The *stacking* approach is slightly more advanced than the simple voting and averaging approaches. Here, a *meta-model* is constructed to obtain a better result than those obtained by just considering the base models themselves.

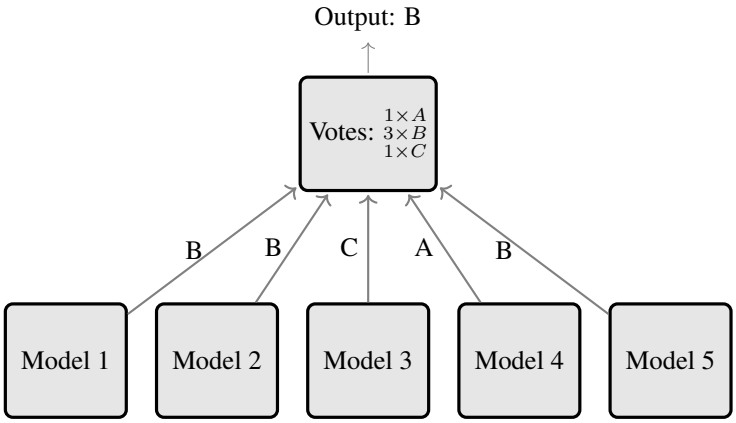



Figure 10.1: An example for voting with five different models for a classification problem with classes A, B, and C. After all five models have been trained, the majority vote of the five base models decides which class is predicted for a given input data point. For the example input above the model outcomes are B, B, C, A, B. This results in an ensemble vote of B.

**Stacking**

In ***stacking*** we build a meta-model $M$ on top of $m$ independently trained base models $M_1, \dots, M_m$. The meta-model's input for a specific data point $\boldsymbol{x}$ is the outputs $M_j(\boldsymbol{x})$ of the base models for $i = j, \dots, m$. Then, the training set[50] is taken to train the meta-model to deliver the best decision according to a given loss function. Often, the same loss function is chosen for the meta-model as for the base models. However, it is possible to employ a different meta-loss function as well.

As an example, let us consider the *stacking* algorithm from [Bre96b]. It uses a positive combination of the base models and the least squares loss to obtain the coefficients

$$(\alpha_1, \dots, \alpha_m) = \underset{\substack{\beta_j \geq 0 \\ j=1,\dots,m}}{\arg\min} \sum_{i=1}^{n} \left( y_i - \beta_j M_j^{(i)}(\boldsymbol{x}_i) \right)^2 \tag{10.1}$$

of the meta-model $M := \sum_{j=1}^{m} \alpha_j M_j$. Moreover, to avoid overfitting in the minimization procedure for (10.1), the models $M_j^{(i)}, i = 1, \dots, n$, have been used there instead of the model $M_j$ for each $j = 1, \dots, m$. Here, $M_j^{(i)}$ is the same as $M_j$, but it has been trained without using the data point $(\boldsymbol{x}_i, y_i)$ in its training data set; see [Bre96b] for details.

This way, the most common approach to stacking is to create a meta-model by linearly combining the base models. In contrast to voting or averaging, stacking allows us to learn how to weight the different outcomes of the different base model classes. This often leads to improved results over that of any of the base models alone; see [Wol92, Bre96b]. In Figure 10.2 we provide an example illustration for stacking with five base models. Moreover, there also exist nonlinear meta-model approaches; see, e.g., [MN15].

[50] Sometimes the training data set for the meta-model is chosen to be different from the one on which the base models have been trained.

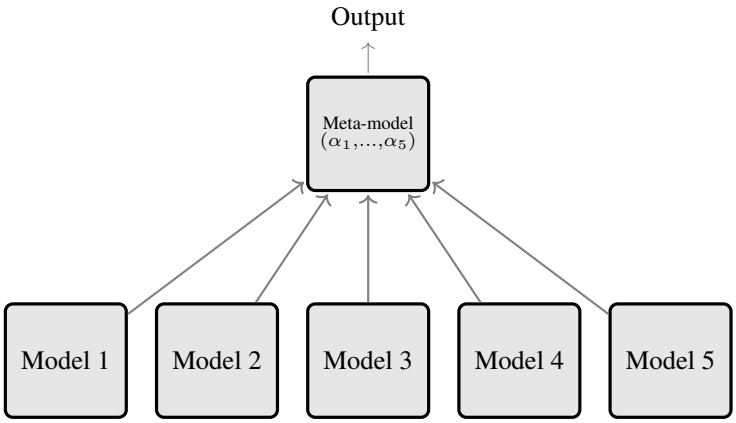



Figure 10.2: An example for stacking with five different models. After all five models have been trained, a meta-model is trained to decide what to output for a certain input depending on the outputs of the five base models.

### 10.2.3 ▪ Boosting

While for stacking all the $m$ different models have been trained independently from each other and while the meta-model is then built by minimizing a meta-loss function, we now intertwine these two steps in *boosting* by successively taking into account one base model after the other.

**Boosting**

In ***boosting*** we build the ensemble of models adaptively by starting with just one simple base model $M_1$. Subsequently, model $M_2$ is trained, where training data points that produced a large error (or were misclassified) for $M_1$ are given more importance during the training process of $M_2$. This process is iterated until $m$ base models have been built. Then, the different base models are combined to come to an ensemble decision. To this end, there are distinct specific approaches which lead to different types of boosting methods.

Similarly as in stacking, in boosting the models $M_1, \ldots, M_m$ are often combined in a weighted way to obtain an ensemble model

$$M := \sum_{i=1}^{m} \alpha_i M_i$$

with real-valued coefficients $\alpha_i$. But now, instead of directly optimizing a loss function for the complicated model $M$, as we did in stacking, the models $M_i$ and weights $\alpha_i$ are optimized successively by an iterative procedure. This is also known as *greedy* optimization. We define the iterates of this process

$$\tilde{M}_l := \tilde{M}_{l-1} + \alpha_l M_l$$

for $l = 1, \ldots, m$ as our new base models. Here, $\tilde{M}_0$ is set to zero. After $m$ iteration steps we obtain $M = \tilde{M}_m$. Here, in the $l$th iteration step, we compute $\alpha_l$ and $M_l$ by (approximately) determining

$$(\alpha_l, M_l) := \underset{\beta\in\mathbb{R}, \hat{M}\in\mathcal{M}}{\arg\min}\ C(\tilde{M}_{l-1} + \beta\hat{M}) \tag{10.2}$$

for a given function $C(f) := \mathcal{L}((f(\boldsymbol{x}_1), y_1), \dots, (f(\boldsymbol{x}_n), y_n))$, which defines the loss on the training data, e.g., for least squares loss $\mathcal{L}$. Here, the base model $\hat{M}$ is chosen from a fixed model class $\mathcal{M}$, such as the class of affine linear functions, for instance. However, for most model classes and loss functions, (10.2) is not straightforward to calculate and the minimum is only approximated. For example, in so-called *gradient boosting* [Fri01] for regression problems, $M_l$ is just taken to be the least squares fit to the residuals

$$\tilde{y}_i := y_i - \tilde{M}_{l-1}(\boldsymbol{x}_i)$$

of the true labels $y_i$ and the evaluations $\tilde{M}_{l-1}(\boldsymbol{x}_i)$ on the training data.

The most famous boosting method for classification problems is *AdaBoost*, which stands for *adaptive boosting*. Here, in the training of $\tilde{M}_l$, the training data points are weighted in such a way that points that were wrongly classified by $\tilde{M}_{l-1}$ are assigned larger weights. In particular, for a two-class problem with $y_i \in \{-1, 1\}$ for all $i = 1, \dots, n$, the exponential weights

$$w_i^{(l)} := \exp\left(-y_i \tilde{M}_{l-1}(\boldsymbol{x}_i)\right)$$

are used. Thus, each weight is either $\exp(1)$, namely if $y_i \neq \tilde{M}_{l-1}(\boldsymbol{x}_i)$, or $\exp(-1)$, otherwise. Then, $\tilde{M}_l = \tilde{M}_{l-1} + \alpha_l M_l$ is determined by solving

$$(\alpha_l, M_l) := \operatorname*{arg\,min}_{\beta \in \mathbb{R}, \hat{M} \in \mathcal{M}} \sum_{i=1}^{n} w_i^{(l)} \exp\left(-y_i \beta \hat{M}(\boldsymbol{x}_i)\right), \tag{10.3}$$

where $\hat{M}$ is optimized over some fixed model class $\mathcal{M}$. For more details on the weighting process and the greedy optimization, we refer the reader to [FS97, HTF09, HRZZ09]. A schematic illustration of the AdaBoost mechanism is given in Figure 10.3.

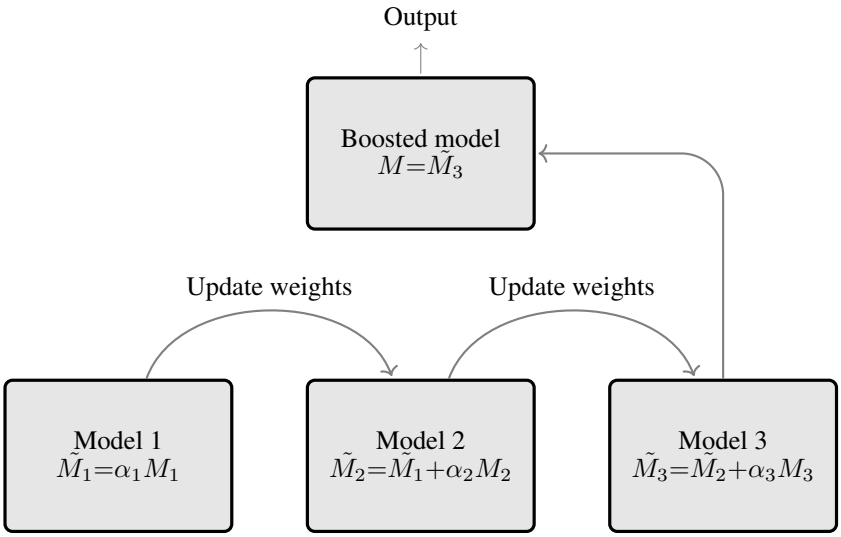



Figure 10.3: An example for AdaBoost with three base models. After a base model is trained, misclassified samples (or samples with high error) are given a larger weight before the next model is trained by (10.3).

### 10.2.4 ▪ Bagging

Stacking and boosting are ensemble methods which reduce (mainly) the bias of the model, i.e., they increase the model complexity either by a meta-model or by adaptively growing the ensemble model. This way, the ensemble model is able to generalize to a broader class of functions than the individual base models alone. However, another important quantity is the variance of the ensemble model, which is related to how robust the model is when having new input data; see also Section 1.3. A method that mainly aims at reducing the variance of the ensemble model is *bagging*.

**Bagging**

***Bagging*** is an abbreviation for *bootstrap aggregation*. It means that the base models $M_1, \dots, M_m$ are trained independently from each other on reduced data sets which were sampled by so-called *bootstrapping*. The ensemble model $M$ is then built by combining the base models appropriately, e.g., by averaging or voting on the results of the base models. In this way, the bagging ensemble model exhibits reduced variance compared to the original base models. For more details see [Bre96a].

In contrast to stacking, where the meta-model is the key part, or boosting, where the adaptive construction of the base models is the core idea, bagging has its focus on the choice of the input data sets for the base models. These data sets are bootstrapped variants of the original training data set.

**Bootstrapping**

The term ***bootstrapping*** describes a specific way to (re-)sample a data set. Assume we have an original data set $\mathcal{D}$ of $n$ samples. Then, a bootstrapped variant $\mathcal{B}$ thereof is a data set of $k \leq n$ samples that have been drawn from $\mathcal{D}$ randomly with replacement. Thus, there might be copies of the same data point in $\mathcal{B}$. If $\mathcal{D}$ contains a large number of independent and identically distributed samples from the underlying, unknown data distribution, $\mathcal{B}$ also can be regarded as representative and independent samples thereof; see [MD93] for more details.

One of the major benefits of bootstrapping is that we can draw samples from the original data distribution without knowing it and without having to acquire new data points according to it, which might be very costly or time-consuming in practical applications. However, the bootstrapped samples can of course only be representative of the underlying data distribution if the original data set is representative as well. A schematic illustration of bagging, i.e., of drawing samples for the base models via bootstrapping and combining these models via voting or averaging, can be found in Figure 10.4.

### 10.2.5 ▪ Random forests

One of the most famous bagging methods is *random forests*, which we will briefly introduce in this section.

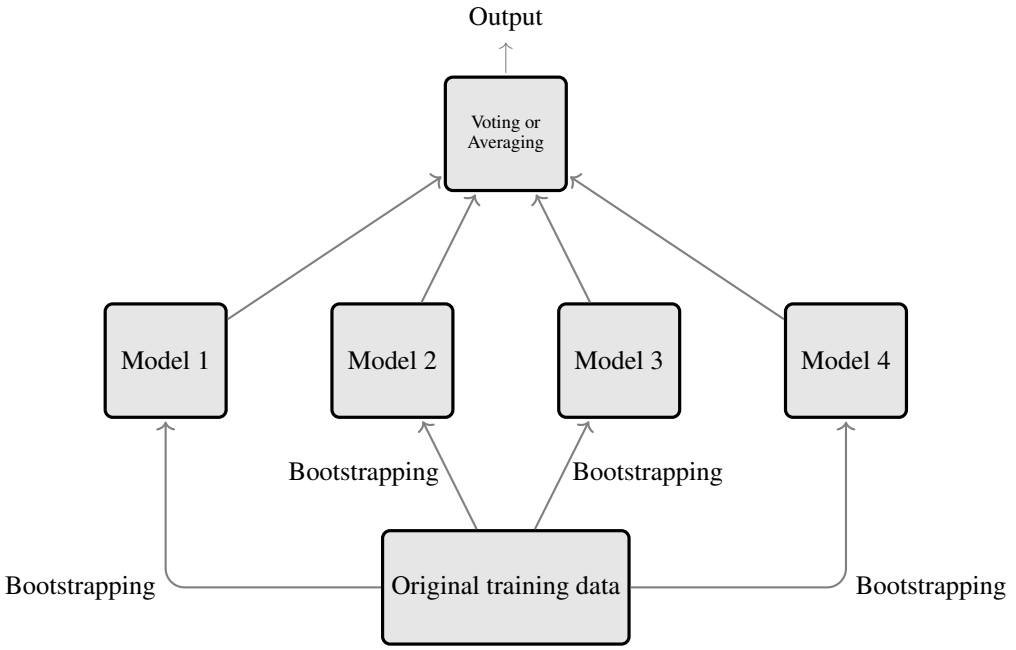



Figure 10.4: An example for bagging with four different base models. First, four bootstrapped sets of some fixed size $k \leq n$ of the original data set are created. After the four base models have been trained on the respective bootstrapped sample set, voting or averaging is performed to acquire the ensemble result.

**Decision trees and random forests**

A ***decision tree*** is a binary[51] tree model for supervised learning, i.e., two branches emerge from each node of the tree except the leaf nodes. It is trained to recursively partition the coordinate space $\mathbb{R}^d$ into small segments. In particular, for each partitioning step, a coordinate direction $c \in \{1, \dots, d\}$ and a value $t \in \mathbb{R}$ are chosen and the ambient space is split into two segments divided by the hyperplane

$$\left\{ \boldsymbol{x} \in \mathbb{R}^d \;\middle|\; [\boldsymbol{x}]_c = t \right\}.$$

The goal is that parts of the training data set $\boldsymbol{x}_1, \dots, \boldsymbol{x}_n$ which exhibit the same (or similar) labels reside in the same partition. To this end, the training is usually done by successively choosing partitions that minimize the *entropy* (or the *Gini impurity*) of the two resulting parts of the data set; see [Bre84] for more details.
A ***random forest*** is an ensemble model using bagging with decision trees as base models; see [Ho95]. Furthermore, each decision tree is only allowed to use a certain subset of the $d$ coordinates of the ambient space during training to reduce the complexity; see [Bre01].

[51] Note that a decision tree technically does not have to be a binary tree. However, for almost all practical applications, decision trees are taken to be binary trees.

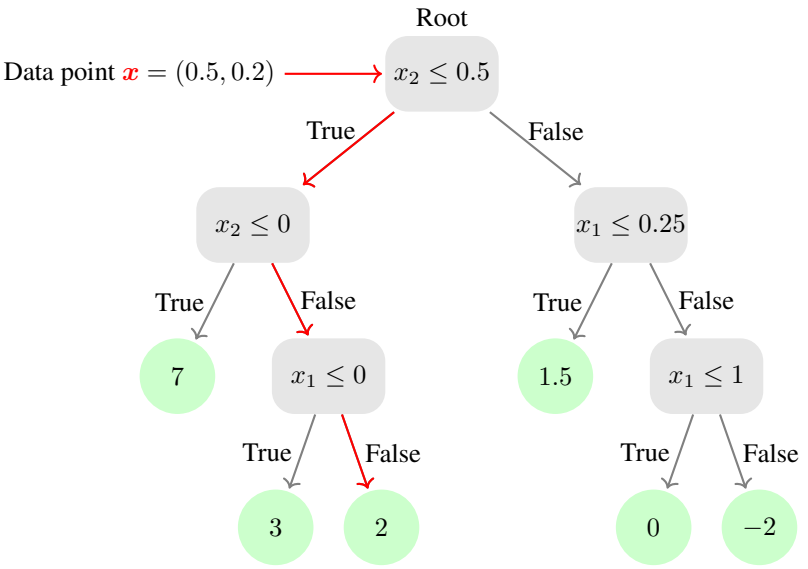



Figure 10.5: An example of an already trained decision tree $T$ of maximum depth 3 for two-dimensional data. The green leaf nodes illustrate the output values of the tree, i.e., for an input whose corresponding decision path leads to a specific leaf, the output value of that leaf is taken as the predicted label for the input. The red path shows how the tree is evaluated for the example input data point $\boldsymbol{x} = (0.5, 0.2)$. The result is $T(\boldsymbol{x}) = 2$.

After a decision tree model has been trained, we can evaluate it for a specific data point $\boldsymbol{x}$ by traversing the corresponding path within the tree beginning at the root and ending in a leaf. As we will explain in more detail in the following, each non-leaf node checks a condition on one of the coordinate dimensions $x_1, \ldots, x_d$ of the data point. If the condition is met, we follow the left branch; else we follow the right one. Once we arrive at a leaf, the value stored in the leaf node is the prediction for the data point $\boldsymbol{x}$ we started with. An example of a decision tree can be found in Figure 10.5.

Besides the maximum tree depth, the most important choice when training a decision tree $T$ is the choice of the splitting criterion for each node. At each node we have to decide which coordinate direction $c \in \{1, \ldots, d\}$ and which value $t \in \mathbb{R}$ to take for the split. Usually, some loss function $C$ or a measure for the homogeneity of the target variable within the subsets is used to evaluate the quality of a specific split, and it is then minimized (heuristically) with respect to the pair $(c, t)$.

Let us take the least squares loss for regression problems as an example. When splitting the root at $(c, t)$ for our training data set $(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_n, y_n)$, we create two new index sets

$$L_{c,t} = \left\{ i \;\middle|\; [\boldsymbol{x}_i]_c \leq t, i = 1, \ldots, n \right\} \quad \text{and} \quad R_{c,t} = \left\{ i \;\middle|\; [\boldsymbol{x}_i]_c > t, i = 1, \ldots, n \right\},$$

where $[\boldsymbol{x}_i]_c$ denotes the $c$th coordinate of the vector $\boldsymbol{x}_i$. Now we can compute the least squares error after the split by taking the average of all labels in the corresponding sets as the predicted value, i.e.,

$$\frac{1}{n} \sum_{i \in L_{c,t}} \left( y_i - \frac{1}{|L_{c,t}|} \sum_{j \in L_{c,t}} y_j \right)^2 + \frac{1}{n} \sum_{i \in R_{c,t}} \left( y_i - \frac{1}{|R_{c,t}|} \sum_{j \in R_{c,t}} y_j \right)^2 .$$

This quantity is then minimized to find the optimal $(c, t)$ for the actual splitting step. Subsequently, the same procedure is performed for the child nodes with the corresponding (smaller) data sets; see also [Bre84] for details. This is again a greedy type of optimization, as in boosting, since we optimize the criterion for all the nodes sequentially. The splitting procedure is stopped when the least squares error in a node is smaller than some threshold or when the maximum depth is reached.

Finally, to create an ensemble model, we build $m$ independent decision trees as base models. To this end, we use a random coordinate subset $\tilde{C}_j \subset \{1, \ldots, d\}$ for each of the $j = 1, \ldots, m$ models as possible splitting candidates, i.e., for the $j$th tree, a splitting in a node is only possible in a direction $c \in \tilde{C}_j$. Furthermore, bootstrapping is used to reduce the training data set size for each decision tree. In this way, we arrive at the random forest model, which mitigates the problem of high variance that single decision trees tend to have; see [Bre01].

## 10.3 ▪ Recurrent neural networks

In Chapter 6, Chapter 7, and Chapter 8 we have discussed feed-forward neural networks, which propagate information just in one direction, i.e., the data are passed through the network layerwise from the input layer to the final output. In contrast to that, so-called *recurrent neural networks* also allow information to be propagated in the opposite direction.

**Recurrent neural networks**

***Recurrent neural networks*** (RNNs) are neural networks for which data can be propagated in any direction. This way, loops can be created in the network graph and the network is no longer acyclic as for the feed-forward NNs from Chapter 6, Chapter 7, and Chapter 8. Recurrent neural networks are used to tackle problems with *sequential* information like time series data, for example. These architectures successively process single elements of a given sequence in contrast to feed-forward neural networks, which use the whole sequence as input. A major benefit of this approach is that RNNs can treat sequences with different lengths and that they need fewer weights to achieve good results in practice; see [GBC16] for details.

The idea behind RNNs is that past information can be used to alter the way in which new information is processed. Often the output of such a network depends on the most recent input and an additional *(hidden) state* $\boldsymbol{h} \in \mathbb{R}^{d_h}$ with fixed hidden state dimension $d_h \in \mathbb{N}$. The hidden state itself depends on earlier inputs and earlier hidden states of the network. This mechanism is beneficial when processing sequential data.

In the following, we will denote by $\boldsymbol{z} \in \mathbb{R}^d$ the network's input[52] sequence. Note that the sequence length $d$ and the hidden state dimension $d_h$ are not related in any way. For the sake of simplicity, we assume here that each element $z_j$ for $j = 1, \ldots, d$ of the input sequence is a real number. Note however that the $z_1, \ldots, z_d$ could be vectors themselves, i.e., $z_t \in \mathbb{R}^{d_i}$ for each $t = 1, \ldots, d$ with some application-dependent input dimension $d_i \in \mathbb{N}$. Their specific form and structure depends on the type of sequence which is to be investigated. Next, let $\boldsymbol{h}_t \in \mathbb{R}^{d_h}$ be the sequence of hidden state vectors for $t = 1, \ldots, d$. Finally, the network's output is denoted by $f(\boldsymbol{z})$. Depending on the task under consideration, this could be a whole sequence

[52] For input sequences with variable lengths, the length is also passed on as an input parameter and $d$ is usually chosen large enough to capture all sequences in the data set. The remaining coordinates of an input are then just set to 0.

$f(\boldsymbol{z}) = (f_1, \dots, f_d) \in \mathbb{R}^d$ or only the value $f(\boldsymbol{z}) = f_d \in \mathbb{R}$, which is referring to the final computation. The first situation is typically encountered for translator networks (see [CvMG+14]), while the second situation appears typically for sequence regression networks (see [QSC+17]). Note here that also the $f_1, \dots, f_d$ could be vectors themselves instead of real numbers, i.e., $f_t \in \mathbb{R}^{d_o}$ with some application-dependent dimension $d_o \in \mathbb{N}$ of the output data of the respective network. For the sake of simplicity, we consider $d_i = d_o = 1$ in the following. The graph representation of a simple RNN can be found in Figure 10.6.

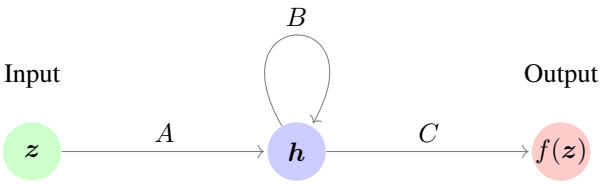



Figure 10.6: Graph representation of a simple recurrent neural network with input $\boldsymbol{z}$, state vector $\boldsymbol{h}$, and output $f(\boldsymbol{z})$.

An RNN graph as in Figure 10.6 has to be read differently from a graph for a feed-forward network. Here, $A : \mathbb{R} \times \mathbb{R}^{d_h} \to \mathbb{R}^{d_h}$ , $B : \mathbb{R}^{d_h} \to \mathbb{R}^{d_h}$, and $C : \mathbb{R}^{d_h} \to \mathbb{R}$ are not just simple weights but certain transformations that depend on the specific architecture of the network, as we will see later on. This representation is used as a compact form of the so-called *unfolded* graph, where the loop is completely expanded and, thus, no longer appears; see Figure 10.7.

Here, the unfolded graph has to be understood as follows:

- The calculation begins with the first element $z_1$ of the input sequence to which $A$ is applied to compute the first hidden state $\boldsymbol{h}_1 := A(z_1, \boldsymbol{0}) \in \mathbb{R}^{d_h}$ with some fixed hidden state dimension $d_h$. The hidden state is then transformed by $C$ to obtain the first output $f_1$.
- In a next step, $z_2$ is fed into the network. Then, by taking into account the values of $z_2$ and $B(\boldsymbol{h}_1)$, the network computes the next hidden state $\boldsymbol{h}_2 = A(z_2, B(\boldsymbol{h}_1))$ and the next output $f_2 = C(\boldsymbol{h}_2)$.
- This procedure is continued until we reach the end of the input sequence $z_d$, i.e., we have $\boldsymbol{h}_t = A(z_t, B(\boldsymbol{h}_{t-1}))$ and $f_t = C(\boldsymbol{h}_t)$ for all $t = 1, \dots, d$, where $\boldsymbol{h}_0 := \boldsymbol{0}$.

### 10.3.1 ▪ Simple recurrent networks

One simple RNN that can be written in the above fashion is the so-called *Elman* network; see also [Elm90]. This RNN is sometimes also referred to as *simple recurrent network*. In the Elman network we have

$$
\begin{aligned}
A(z, \boldsymbol{h}) &:= \phi(\boldsymbol{a}z + \boldsymbol{h} + \boldsymbol{a}_0) && \text{with } \boldsymbol{a}, \boldsymbol{a}_0, \boldsymbol{h} \in \mathbb{R}^{d_h}, z \in \mathbb{R}, \text{ and activation function } \phi, \\
B(\boldsymbol{h}) &:= \boldsymbol{B}\boldsymbol{h} && \text{with } \boldsymbol{B} \in \mathbb{R}^{d_h \times d_h} \text{ and } \boldsymbol{h} \in \mathbb{R}^{d_h}, \text{ and} \\
C(\boldsymbol{h}) &:= \psi(\boldsymbol{c}^T \boldsymbol{h} + c_0) && \text{with } \boldsymbol{c}, \boldsymbol{h} \in \mathbb{R}^{d_h}, c_0 \in \mathbb{R}, \text{ and activation function } \psi.
\end{aligned}
$$

Note that the application of the activation function $\phi$ has to be understood componentwise. As

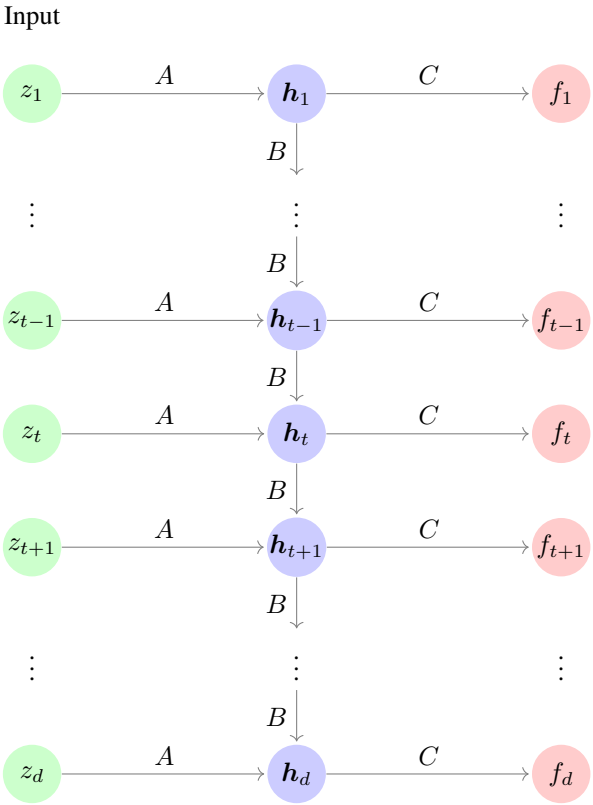



Figure 10.7: Unfolded graph representation of a simple recurrent neural network with input $\boldsymbol{z} = (z_1, \dots, z_d)$, state vectors $\boldsymbol{h}_t$ for $t = 1, \dots, d$, and output $f(\boldsymbol{z}) = (f_1, \dots, f_d)$, where $\boldsymbol{h}_t = A(z_t, B(\boldsymbol{h}_{t-1}))$ and $f_t = C(\boldsymbol{h}_t)$.

we see, the sequence part $z_t$ is used to compute the hidden state $\boldsymbol{h}_t$ and the output $f_t$ via

$$\boldsymbol{h}_t = \phi(\boldsymbol{a} z_t + \boldsymbol{B}\boldsymbol{h}_{t-1} + \boldsymbol{a}_0) \quad \text{and} \quad f_t = \psi(\boldsymbol{c}^T \boldsymbol{h}_t + c_0)$$

for each $t = 1, \dots, d$, where the variables $\boldsymbol{a}, \boldsymbol{a}_0, \boldsymbol{B}, \boldsymbol{c}, c_0$ have to be determined by an optimizer. To this end, backpropagation is applied to the unfolded graph after a loss function has been chosen. This approach is also known as *backpropagation through time*. While this works analogously to feed-forward neural networks, simple RNNs such as the one above, but with long input sequences, tend to encounter problems with many local minima and vanishing or exploding gradients during the optimization procedure with SGD-type methods; see [GBC16] for more details. Therefore, *skip connections* between far away sequence parts have been introduced to circumvent this problem to some extent, i.e., the output of neurons at part $t$ of the sequence can directly be accessed by neurons at part $t + \delta t$ of the sequence for a $\delta t \gg 1$. In other words, there are connections between neurons corresponding to far away sequence parts. Besides, more sophisticated RNNs have been proposed, which lead to a more stable optimization process.

### 10.3.2 ▪ Long short-term memory networks

One of the most famous RNNs is the long short-term memory network. It is able to deal with the problem of vanishing gradients.

**Long short-term memory network (LSTM)**

The idea of ***long short-term memory networks*** (LSTMs) is to automatically detect the time horizon $\delta t$ for which the relation between $\boldsymbol{h}_t$ and $\boldsymbol{h}_{t+\delta t}$ is still relevant for any $1 \leq t \leq d$, i.e., the time horizon is small enough such that $\boldsymbol{h}_t$ and $\boldsymbol{h}_{t+\delta t}$ are not independent of each other. While an LSTM updates the hidden state during the processing of the input sequence as simple recurrent networks do, it is also allowed to *forget* past information that is simply no longer relevant at a certain time. We refer the reader to [HS97] for details.

In contrast to previous models, an LSTM introduces a second hidden state variable, the so-called *cell state* $\boldsymbol{c}_t \in \mathbb{R}^{d_h}$. The visualization of an LSTM network is given in Figure 10.8. Here, we assume that each element of the input and output sequences is indeed a vector. For the sake of simplicity, we let $z_t \in \mathbb{R}^{d_i}$ be of input dimension $d_i \in \mathbb{N}$ at each time step $t = 1, \ldots, d$, and we let the corresponding output $f_t \in \mathbb{R}^{d_h}$ be of the same dimension $d_h$ as the hidden state. Although these are vectors, we do not use the boldsymbol vector notation for them here to be consistent with our previous notation from Section 10.3.1 and to avoid confusion.

The illustration in Figure 10.8 shows how the computation of $f_t \in R^{d_h}$ and the new hidden states $\boldsymbol{h}_t \in \mathbb{R}^{d_h}$ and cell states $\boldsymbol{c}_t \in \mathbb{R}^{d_h}$ works when given $z_t \in \mathbb{R}^{d_i}$ and $\boldsymbol{h}_{t-1} \in \mathbb{R}^{d_h}$, $\boldsymbol{c}_{t-1} \in \mathbb{R}^{d_h}$. Essentially, the computational graph is built by certain arithmetic operations and activation functions ($\sigma$ and $\tanh$). The involved weights and biases are omitted here from the

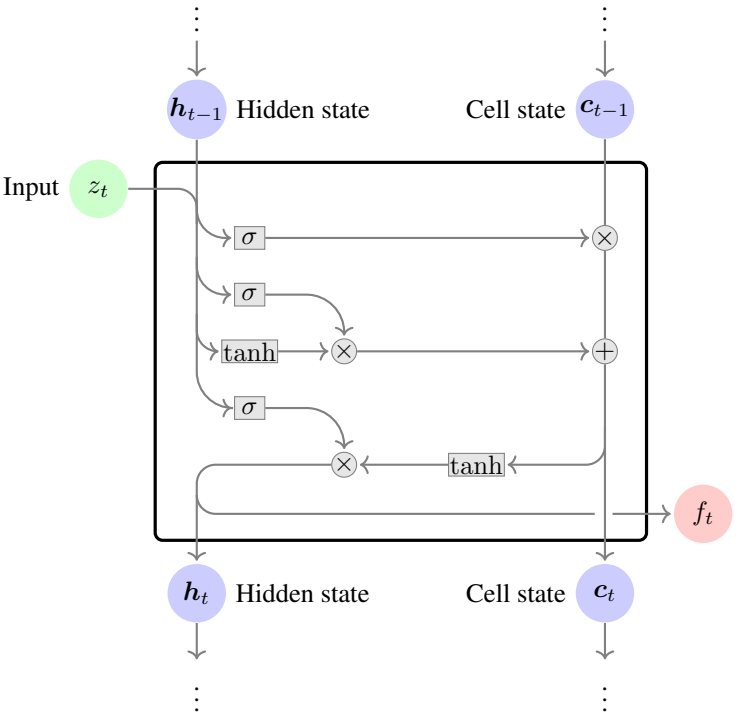



Figure 10.8: Graph representation of an LSTM network with input $\boldsymbol{z} = (z_1, \ldots, z_d)$, hidden state vectors $\boldsymbol{h}_t$, and cell state vectors $\boldsymbol{c}_t$ for $t = 1, \ldots, d$ and output $f(\boldsymbol{z}) = (f_1, \ldots, f_d)$.

graph for the sake of clarity. The forward pass of the illustrated LSTM cell then reads

$$\begin{aligned}
\boldsymbol{forget}_t &:= \sigma(\boldsymbol{A}_f z_t + \boldsymbol{B}_f \boldsymbol{h}_{t-1} + \boldsymbol{b}_f) && \text{(forget gate)},\\
\boldsymbol{update}_t &:= \sigma(\boldsymbol{A}_u z_t + \boldsymbol{B}_u \boldsymbol{h}_{t-1} + \boldsymbol{b}_u) && \text{(update gate)},\\
\boldsymbol{cellAct}_t &:= \tanh(\boldsymbol{A}_c z_t + \boldsymbol{B}_c \boldsymbol{h}_{t-1} + \boldsymbol{b}_c) && \text{(cell activation)},\\
f_t &:= \sigma(\boldsymbol{A}_o z_t + \boldsymbol{B}_o \boldsymbol{h}_{t-1} + \boldsymbol{b}_o) && \text{(output gate)},\\
\boldsymbol{c}_t &:= \boldsymbol{c}_{t-1} \odot \boldsymbol{forget}_t + \boldsymbol{cellAct}_t \odot \boldsymbol{update}_t && \text{(new cell state)},\\
\boldsymbol{h}_t &:= \tanh(\boldsymbol{c}_t) \odot f_t && \text{(new hidden state)}
\end{aligned}$$

with weights $\boldsymbol{A}_f, \boldsymbol{A}_u, \boldsymbol{A}_c, \boldsymbol{A}_o \in \mathbb{R}^{d_h \times d_i}$ and $\boldsymbol{B}_f, \boldsymbol{B}_u, \boldsymbol{B}_c, \boldsymbol{B}_o \in \mathbb{R}^{d_h \times d_h}$ and biases $\boldsymbol{b}_f, \boldsymbol{b}_u, \boldsymbol{b}_c$, $\boldsymbol{b}_o \in \mathbb{R}^{d_h}$. The first four lines compute the so-called *forget gate* activation, *update gate* activation, cell activation, and output gate activation by adding weighted input values, hidden values, and biases and applying certain activation functions. The latter are either the hyperbolic tangent $\tanh$ or the sigmoidal function $\sigma(x) := (1 + \exp(-x))^{-1}$. As before, the activation functions are applied coordinatewise. The last two lines compute the new cell and hidden states by using Hadamard products $\odot$ of the previously computed values. In contrast to the cell state $\boldsymbol{c}_t$, the hidden state $\boldsymbol{h}_t$ also contains information about the output $f_t$. Note that the information on the previous cell state $\boldsymbol{c}_{t-1}$ becomes less important if the forget gate value $0 < \boldsymbol{forget}_t \leq 1$ is small. In this way, the LSTM learns via the forget gate's weights and bias how important older cell states are. More details on LSTMs can be found in [HS97, She20].

Besides the depicted classical variant of an LSTM network, there are variations of such networks using gates. For instance, the *gated recurrent unit* network [CvMG+14] has more compact cells than the LSTM network and has been successful in practical applications as well; see also [GBC16] for details.

### 10.3.3 ▪ Further improvements

A direct way to circumvent the problem of vanishing and exploding gradients during the RNN optimization consists of using orthogonal/unitary hidden state transition matrices. Let for instance $z_t \in \mathbb{R}$ be scalar again and let us consider the evolution

$$\boldsymbol{h}_t = \phi\left(\boldsymbol{a} z_t + \boldsymbol{B}\boldsymbol{h}_{t-1} + \boldsymbol{a}_0\right)$$

of the hidden state vector for some $\boldsymbol{a}, \boldsymbol{a}_0 \in \mathbb{R}^{d_h}$, $\boldsymbol{B} \in \mathbb{R}^{d_h \times d_h}$. Then, one can show that, for an orthogonal (or unitary) matrix $\boldsymbol{B}$ and for $\phi = \mathrm{ReLU}$, the optimization with SGD-like methods works well, i.e., the norms of the involved gradients stay constant; see [ASB16, HSL16] for details. However, these networks suffer from a poorer expressivity than standard RNNs, and the class of functions that can be represented by such a network is thus more limited. Recent research addresses the question of a good tradeoff between the expressivity and the (almost-)orthogonality of the RNN; see [KGPT+19].

Another class of RNNs which are able to circumvent the problem of vanishing and exploding gradients is *continuous-time* RNNs. Here, similarly to Chapter 8, the RNN's evolution equations are interpreted as a time-discrete variant of an ODE. For example, an ODE of type

$$\dot{\boldsymbol{h}}(t) = \phi(\boldsymbol{a} z(t) + \boldsymbol{B}\boldsymbol{h}(t) + \boldsymbol{a}_0)$$

could be discretized as

$$\boldsymbol{h}_t = \boldsymbol{h}_{t-1} + \delta t \phi(\boldsymbol{a} z_{t-1} + \boldsymbol{B}\boldsymbol{h}_{t-1} + \boldsymbol{a}_0)$$

with time step $\delta t$, which just resembles a time-explicit Euler discretization with respect to $\boldsymbol{h}$ and $z$. This then defines the hidden state evolution equation for the corresponding RNN. For more details on the connection between certain ODEs and RNNs, see [She20].

Note that the connection to ODEs allows for a more elaborated stability analysis of the network's evolution and optimization processes. Thus, more complex ODEs and more intricate time-discretization schemes can lead to RNNs that have beneficial behavior when it comes to optimizing the network's weights and biases with SGD. This way, the problem of vanishing and exploding gradients can be circumvented; see, e.g., [CCHC19] for antisymmetric hidden state transition matrices, [EAQ+21] for a learnable scale of symmetrized or antisymmetric hidden state transition matrices, [RM21] for an RNN stemming from a second-order ODE system motivated by coupled nonlinear oscillators, and [RMEM21] for an RNN stemming from a multi-scale ODE system. These approaches manage to either reduce the problem of vanishing and exploding gradients or circumvent it completely.

## 10.4 ▪ Transformer neural networks

Up to the year 2017, the majority of neural network models for handling sequence data were based on recurrent neural networks. As we have seen in the last section, special care has to be taken when training these networks for long sequences or time series because of the vanishing and exploding gradients problem when doing backpropagation through time. In 2017, however, a new neural network architecture called *transformer* was presented, which naturally avoids the above-mentioned problems; see [VSP+17]. Since the topic of transformers is quite involved, we will not discuss every technical subtlety in detail but rather aim to give a short overview of the underlying mechanism. For a more detailed and illustrative introduction to transformers we recommend [VSP+17] and the tutorial at http://jalammar.github.io/illustrated-transformer.

**Transformer neural networks and attention modules**

A ***transformer*** is a neural network with a special structure that handles sequential input data. Similar to autoencoders, it is built by combining an encoder and a decoder, both of which consist of a combination of feed-forward neural networks, ***attention modules***, and normalization layers. The attention modules model the importance of the dependencies between parts of two sequences. In this way, the network can automatically learn which parts of the input sequence depend on each other, for instance. This is known as *self-attention*. More details on attention modules and transformers can be found in [LPM15, VSP+17].

Transformer-based neural networks have been especially successful in *natural language processing* tasks such as text generation or machine translation [Bea20, DCLT19, VSP+17]. To deal with sequence-to-sequence problems, e.g., in machine translation, the transformer employs two separate modules: an encoder and a decoder. First, the encoder processes a sequence, e.g., a sentence (sequence of words) from the source language, as an input. Its goal is to compute a compact representation of all the relevant information in the input sequence, similar to the hidden state in RNNs. Next, the representation that the encoder outputs is taken as an input for the decoder. Moreover, the decoder is usually also fed a piece of the desired output sequence, e.g., the already translated beginning of a sentence from the target language. Now, the decoder generates the next element of the output sequence based on the given representation of the input sequence and the given part of the output sequence. After adding the new element that the decoder produced to the output sequence, this process is repeated until the whole output sequence has been produced, e.g., until a full sentence has been output by the decoder.

One major benefit of transformer networks in contrast to recurrent neural networks is that transformers treat the whole input sequence at once because of the employed self-attention modules. These are able to efficiently model the semantic dependencies between different words of an input sentence. In this way, the transformer network can grasp the structure of a sentence much better than an LSTM network. In this way, long range dependencies, which were problematic for RNNs because of the computational costs and the information loss over long sequences, can be modeled much more easily. Moreover, even a parallelization of the computations is possible, which was not the case for RNNs because of their successive computation of the hidden state vectors.

Before we introduce the specific network architecture of the transformer, we need to have a look at attention modules to understand their basic mechanisms.

### 10.4.1 ▪ Attention modules

An attention module learns which parts of a sequence are most important in the current context, i.e., it learns on which parts most of the attention needs to be focused in the actual situation. Let us assume that each part of a sequence is represented by a vector $\boldsymbol{r}_i$ of a fixed dimension $\tilde{d}$ for $i = 1, \dots, d$, i.e., the sequence equals $(\boldsymbol{r}_1, \dots, \boldsymbol{r}_d) \in \mathbb{R}^{\tilde{d} \times d}$. For language models, for instance, a sentence (sequence) of length $d$ consists of $d$ separate words (parts of the sequence) that are then each assigned to a vector in $\mathbb{R}^{\tilde{d}}$ which should—in the best case—reflect their meaning, i.e., words with a similar meaning should be assigned to vectors that lie close to each other and vice versa. This process is called *word embedding*; see also [BCB15] for more details.

**Attention modules and database retrieval** For $V \in \mathbb{N}$, an attention module computes a weighted sum of *value* vectors $\boldsymbol{v}_1, \dots, \boldsymbol{v}_d \in \mathbb{R}^V$, i.e., we compute the expression

$$\sum_{i=1}^{d} w_i \boldsymbol{v}_i \quad \text{s.t.} \quad \sum_{i=1}^{d} w_i = 1 \text{ and } w_i \geq 0 \quad \forall\, i = 1, \dots, d \tag{10.4}$$

with certain weights $w_i$. Here, the value vectors $\boldsymbol{v}_1, \dots, \boldsymbol{v}_d$ are related to the representation vectors $\boldsymbol{r}_1, \dots, \boldsymbol{r}_d$. The exact relationship depends on the type of attention module and will become clearer in the next section.

More specifically, an attention module mimics the process of retrieving a *value* $\boldsymbol{v} \in \mathbb{R}^V$ based on a *query* $\boldsymbol{q} \in \mathbb{R}^Q$ and a *key* $\boldsymbol{k} \in \mathbb{R}^K$ from a database[53] for some fixed dimensions $V, Q, K \in \mathbb{N}$. To this end, let us assume that the $d$ value vectors $\boldsymbol{v}_1, \dots, \boldsymbol{v}_d$ correspond to $d$ key vectors $\boldsymbol{k}_1, \dots, \boldsymbol{k}_d$. Now, we choose the weights $w_i$ in (10.4) in such a way that they reflect the similarity between $\boldsymbol{q}$ and $\boldsymbol{k}_i$. The idea behind this mechanism is that values with keys which match the query the most should get most of the attention, i.e., they will get assigned larger weights.[54] Let us now discuss how the weights $w_i$ are determined for the dot product similarity measure, which is used in the transformer architecture.

**Dot product similarity** The most commonly used similarity measure in attention modules is a scaled dot product between the query vector and the key vectors when $Q = K$, i.e.,

$$\text{similarity}(\boldsymbol{q}, \boldsymbol{k}_i) := \frac{\boldsymbol{q}^T \boldsymbol{k}_i}{\sqrt{K}}.$$

[53] When we introduce the specific transformer architecture in Section 10.4.2, it will become evident that the database contains keys and values which represent feature vectors of words in either the input or the output language. These are then matched to query feature vectors.

[54] In the case of database retrieval, this means that a weighted sum of all values from the database is returned, where the weights reflect how well the query matched the corresponding keys.

Now, in order to fulfill the side conditions of non-negative weights that sum up to one, we apply the softmax function to the similarities, i.e., we set

$$w_i := \mathrm{softmax}(\mathrm{similarity}(\boldsymbol{q}, \boldsymbol{k}_i)) = \frac{\exp\left(\mathrm{similarity}(\boldsymbol{q}, \boldsymbol{k}_i)\right)}{\sum_{j=1}^{d} \exp\left(\mathrm{similarity}(\boldsymbol{q}, \boldsymbol{k}_j)\right)}. \tag{10.5}$$

Note that the weights do not depend on the values $\boldsymbol{v}_1, \dots, \boldsymbol{v}_d$ in general. However, in the specific case of the attention modules in the transformer, there is a linear relationship between the keys and the values, as we will see in Section 10.4.2. Thus, the weights do depend on $\boldsymbol{v}_1, \dots, \boldsymbol{v}_d$ there.

**Multiple queries** Finally, let us have a look at the case where we have $d_q$ different queries $\boldsymbol{q}_1, \dots, \boldsymbol{q}_{d_q}$ of dimension $Q = K$. If we stack the value, query, and key vectors row-wise into matrices $\boldsymbol{V} \in \mathbb{R}^{d \times V}, \boldsymbol{Q} \in \mathbb{R}^{d_q \times Q}$, and $\boldsymbol{K} \in \mathbb{R}^{d \times K}$ and if indeed $Q = K$, we can write the output of the attention module as

$$\mathrm{attention}(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) := \mathrm{softmax}\left(\frac{\boldsymbol{Q}\boldsymbol{K}^T}{\sqrt{K}}\right)\boldsymbol{V} \in \mathbb{R}^{d_q \times V}. \tag{10.6}$$

Here, $\frac{\boldsymbol{Q}\boldsymbol{K}^T}{\sqrt{K}} \in \mathbb{R}^{d_q \times d}$ has entries

$$\left[\frac{\boldsymbol{Q}\boldsymbol{K}^T}{\sqrt{K}}\right]_{i,j} = \mathrm{similarity}(\boldsymbol{q}_i, \boldsymbol{k}_j)$$

for $i = 1, \dots, d_q$ and $j = 1, \dots, d$. The softmax function from (10.5) is applied elementwise.

Although we now know how the attention module works for given queries, keys, and values, the question remains how the values, queries, and keys are computed in the first place. In the next section we will see how they are chosen in the transformer network.

## 10.4.2 ▪ The transformer architecture

The overall structure of the transformer network used for machine translation tasks can be found in Figure 10.9. Note again that we will not discuss every detail of it, but rather give a short overview on the underlying mechanisms. For a more detailed introduction to transformers we recommend [VSP+17] and the references therein. Before we explain in Section 10.4.3 how the transformer is trained and how the trained model is used to generate new sequence data, let us first briefly discuss the individual parts of the transformer network to understand its main building blocks. Our guideline will always be the illustration in Figure 10.9. Note that there are actually $N \in \mathbb{N}$ sequential instances of both the encoder and the decoder blocks that are omitted in the picture for better readability. The original transformer architecture from [VSP+17] uses $N = 6$.

**Input/Output embedding** Each element of a given input or output sequence is embedded into a suitable feature space. For instance, for machine translation, each word (or *token*/*subword unit* [SHB16]) of a sentence is embedded into $\mathbb{R}^{\tilde{d}}$ via word embedding. In particular, for the input embedding, we first fix the size $\tilde{s} \in \mathbb{N}$ of the possible vocabulary for the input language and store all possible words in an *input dictionary*.[55] Then, each word of the dictionary is assigned a different index $i \in \{1, \dots, \tilde{s}\}$. Thus, each word can be encoded by the so-called *one-hot*

[55] Note that a dictionary here only contains words from one specific language and not the corresponding translations into another language. This is consistent with the use of the term *dictionary* in the machine learning literature.

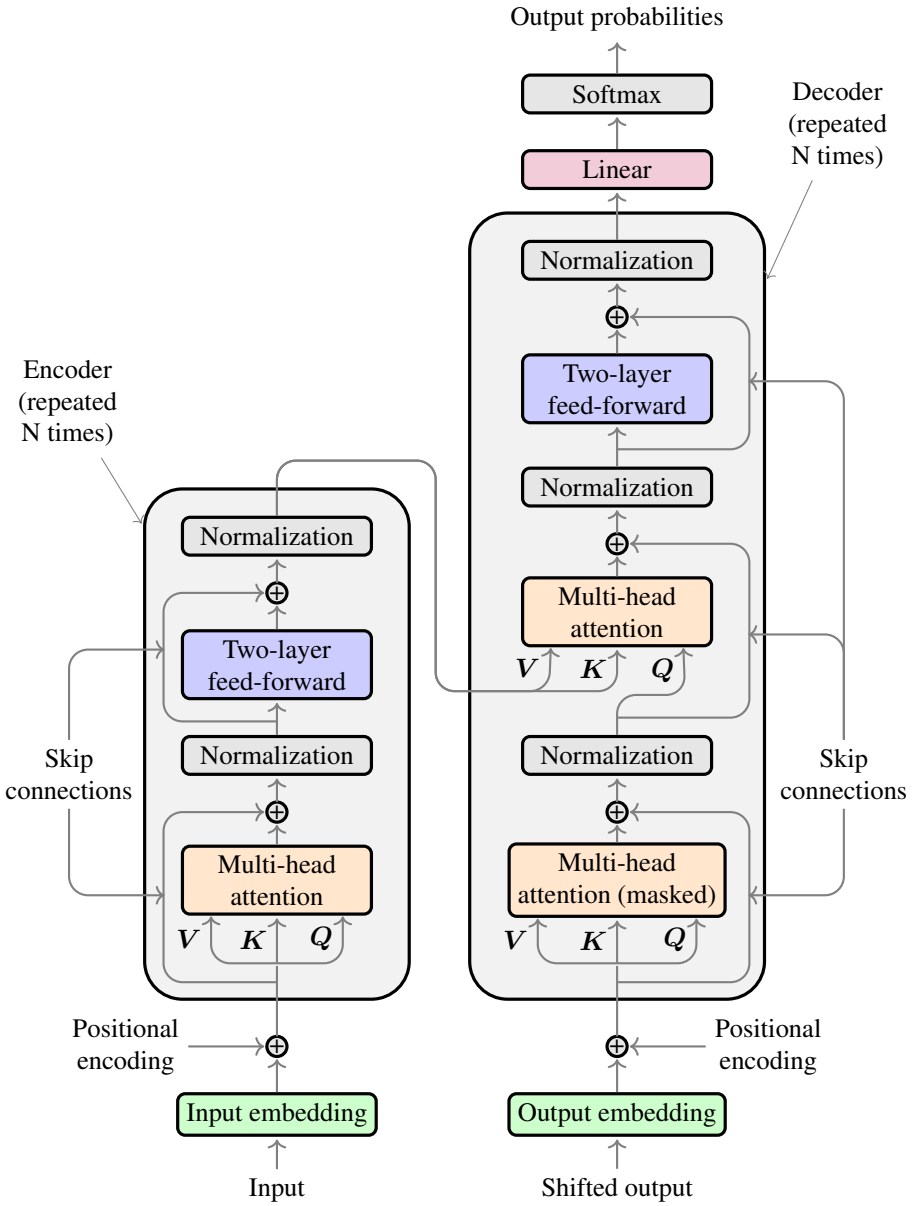



Figure 10.9: A schematic overview of the transformer network for translation. The main parts are N consecutive encoder (left) and decoder (right) blocks containing attention and feed-forward sub-blocks. Our illustration is based on Figure 1 of [VSP+17].

*encoding*, i.e., the word belonging to index $i$ is assigned to the $i$th unit vector $\boldsymbol{e}_i \in \mathbb{R}^{\tilde{s}}$ with entries

$$[\boldsymbol{e}_i]_j := \begin{cases} 1 & \text{if } i = j, \\ 0 & \text{else} \end{cases}$$

for $j = 1, \ldots, \tilde{s}$. The resulting embedding vector is obtained by multiplication of a scaled embedding matrix $\sqrt{\tilde{d}} \cdot \boldsymbol{E}^{(\text{in})} \in \mathbb{R}^{\tilde{d} \times \tilde{s}}$ with the vector $\boldsymbol{e}_i$, i.e., the $i$th column of $\sqrt{\tilde{d}} \cdot \boldsymbol{E}^{(\text{in})}$ is the

embedding vector for the $i$th word from the dictionary. The entries of $\boldsymbol{E}^{(\text{in})}$ are determined during the training phase of the transformer; see Section 10.4.3 for details. The same process is applied for the output embedding with a different learnable weight matrix $\boldsymbol{E}^{(\text{out})} \in \mathbb{R}^{\tilde{d} \times \tilde{t}}$, where $\tilde{t} \in \mathbb{N}$ is the dictionary size for the output language. In this way, the word embeddings are learned from scratch by the transformer.

**Positional encoding** Subsequently, special positional encoding vectors are added to the results. This is done to ensure that the network can identify the position in which a word occurred in the original sentence. As the remaining elements of the network, e.g., the attention modules, are permutation invariant, i.e., they do not work with the initial sequences but only with a set of vectors without particular order, the algorithm could not infer a word's position without the positional encoding. For details on the specific way the positional encoding is calculated in the case of the transformer, see [VSP+17].

**Multi-head attention in the encoder** After adding the positional encoding, the resulting vectors are passed to a so-called *multi-head* attention layer in the encoder. Let us call the incoming $d$ vectors $\boldsymbol{r}_1, \ldots, \boldsymbol{r}_d \in \mathbb{R}^{\tilde{d}}$ and let $\boldsymbol{R} = (\boldsymbol{r}_1, \ldots, \boldsymbol{r}_d)^T \in \mathbb{R}^{d \times \tilde{d}}$, where $\tilde{d}$ is the embedding dimension introduced at the beginning of this section. Note however that different sequences can have different sequence lengths $d$, i.e., the word count might vary between the sentences in the data set.

Now, let us answer the open question from Section 10.4.1 of how the vectors $\boldsymbol{r}_1, \ldots, \boldsymbol{r}_d$ are related to the attention module's queries, keys, and values. To this end, note that the attention module here is a self-attention module, meaning that queries, keys, and values are built from the same data. More specifically, they are constructed by multiplying the incoming vectors $\boldsymbol{r}_1, \ldots, \boldsymbol{r}_d$ of $\boldsymbol{R}$ by a matrix, i.e., by performing a linear transform

$$\boldsymbol{V} = \boldsymbol{R}\boldsymbol{W}_{\text{value}}, \quad \boldsymbol{K} = \boldsymbol{R}\boldsymbol{W}_{\text{key}}, \quad \boldsymbol{Q} = \boldsymbol{R}\boldsymbol{W}_{\text{query}}$$

for learnable weight matrices $\boldsymbol{W}_{\text{value}} \in \mathbb{R}^{\tilde{d} \times V}$, $\boldsymbol{W}_{\text{key}} \in \mathbb{R}^{\tilde{d} \times K}$, and $\boldsymbol{W}_{\text{query}} \in \mathbb{R}^{\tilde{d} \times Q}$, which are determined in the training phase of the transformer; see Section 10.4.3. Here, we choose the dimensions $V$ and $K$ a priori and set $Q = K$. The attention module's output is then computed by (10.6). Note that $d_q = d$ in the case of self-attention since we have as many queries as we have keys. Thus, we obtain $d_q$ many vectors of dimension $V$ as the result.

While the above procedure describes a single attention module, the transformer uses *multi-head* attention. This means that we have $H$ parallel single attention modules (so-called *heads*), i.e., $H$ heads are applied with different learnable weight matrices $\boldsymbol{W}_{h,*}$ for $h = 1, \ldots, H$ and associated $* =$ value, key, query to obtain $\boldsymbol{V}_h$, $\boldsymbol{K}_h$, and $\boldsymbol{Q}_h$. This introduces additional learnable weights and thus increases the expressivity of the transformer network. It can be seen in analogy to parallel convolutional layers, for instance; see Section 6.4.1. The results of the parallel heads are collocated to

$$(\text{attention}(\boldsymbol{V}_1, \boldsymbol{K}_1, \boldsymbol{Q}_1), \ldots, \text{attention}(\boldsymbol{V}_H, \boldsymbol{K}_H, \boldsymbol{Q}_H)) \in \mathbb{R}^{d_q \times HV},$$

which is finally multiplied by a learnable matrix $\boldsymbol{W}_{\text{conc}} \in \mathbb{R}^{HV \times \tilde{d}}$ to obtain $d_q$ vectors of dimension $\tilde{d}$.

**Masked multi-head attention in the decoder** Similarly to the multi-head attention module in the encoder, there is a *masked* multi-head attention module as a first building block in the decoder. Both work in the same fashion except for the fact that there is a positional restriction in the masked variant. More specifically, query vectors can only attend to key vectors that are

positioned in front of them in the sequence. Technically, this is done by setting the similarity to $-\infty$ in the case of invalid query-key pairs. This makes sure that the transformer's decoder is able to construct sentences word-by-word without using knowledge about parts of the sentence that have not been constructed yet.

**Cross-attention module** Finally, there is a second multi-head attention module in the decoder that mixes the information of the encoded input sequence and the output sequence. Here, $d_q$ query vectors, which have been computed from the output sequence, attend to $d$ key vectors from the encoded input. Note that $d_q$ and $d$ can be different this time. Similar to self-attention, the queries, keys, and values are constructed via learnable weight matrices

$$\boldsymbol{V} = \boldsymbol{R}\boldsymbol{W}_{\text{value}}, \quad \boldsymbol{K} = \boldsymbol{R}\boldsymbol{W}_{\text{key}}, \quad \boldsymbol{Q} = \boldsymbol{S}\boldsymbol{W}_{\text{query}},$$

where $\boldsymbol{R} \in \mathbb{R}^{d \times \tilde{d}}$ are the encoded input vectors and $\boldsymbol{S} \in \mathbb{R}^{d_q \times \tilde{d}}$ are the output vectors after the masked multi-head attention module and the following normalization. The learnable weight matrices have the same structure as in the multi-head attention modules described above. Furthermore, we again employ $H$ parallel heads in order to have a multi-head attention module with $H$ different instances of the learnable weight matrices, which are determined in the training phase of the transformer; see Section 10.4.3.

**Skip connections** Next to the multi-head attention modules and the two-layer feed-forward modules we always have a skip connection as in Figure 10.9, which adds the input of the corresponding module to its output. This gives the network the possibility to balance the original information with the result of the module.

**Layer normalization** The normalization blocks perform a *layer normalization*. Here, instead of using standardization (see Section 2.5), each entry of an incoming vector $\boldsymbol{x} = (x_1, \ldots, x_{\tilde{d}})$ is normalized by

$$x_j^{\text{norm}} := g_j \frac{x_j - m}{\sigma^2} \quad \text{with} \quad m := \frac{1}{\tilde{d}} \sum_{i=1}^{\tilde{d}} x_i \;\text{ and }\; \sigma^2 := \frac{1}{\tilde{d}} \sum_{i=1}^{\tilde{d}} (x_i - m)^2,$$

where $g_j$ is a learnable weight parameter for $j = 1, \ldots, \tilde{d}$, which is determined during the training phase; see Section 10.4.3. For more details on layer normalization see [BKH16].

**Two-layer feed-forward network** Besides multi-head attention modules and layer normalization, the transformer also employs fully connected two-layer feed-forward neural networks with ReLU activation functions as modules in both the encoder and decoder. The weights and biases of these networks are being determined in the training phase of the transformer; see also Section 10.4.3.

**Output probabilities** After applying $N$ consecutive decoder blocks, the resulting vectors are transformed from $\tilde{d}$ dimensions to $\tilde{t}$ dimensions, where $\tilde{t}$ is the size of the vocabulary of the output language. This transformation is usually fixed to be the transpose of $\boldsymbol{E}^{(\text{out})}$, which we used for the output word embedding. In particular, for a vector $\boldsymbol{x} \in \mathbb{R}^{\tilde{d}}$ resulting from the final decoder block, we compute $\left(\boldsymbol{E}^{(\text{out})}\right)^T \boldsymbol{x}$. We can think of this as a vector of Euclidean scalar products between $\boldsymbol{x}$ and the columns of $\boldsymbol{E}^{(\text{out})}$. After applying a final softmax activation function, the transformer returns the output probabilities of each word in the dictionary. This means that the

most probable word, i.e., the word for which the scalar product between its embedding vector and $\boldsymbol{x}$ is the largest among all words from the dictionary, is the best candidate to continue the given output sequence.

### 10.4.3 ▪ Training and generation

Note that the subtle difference between the self-attention module in the encoder and the masked self-attention module in the decoder is key to understanding what the main purposes of the encoder and the decoder are: The former aims to generate a representative of the *whole* input sequence, while the latter aims to construct an output sequence *word by word* (or token by token) and thus cannot attend to yet unknown words in the sequence. This shows us that the training and generation phases will always work on a word by word basis when computing the transformer's output. This means that the transformer predicts the next word for a given input sequence and a given (partial) output sequence. Let us explain this in more detail now.

**Training phase** As we have seen, the transformer employs an encoder and a decoder, each consisting of several modules, e.g., input/output embedding, multi-head attention, normalization, and two-layer feed-forward networks. The learnable parameters, which need to be determined during the training phase, are

- the input/output embedding matrices $\boldsymbol{E}^{(\text{in})}, \boldsymbol{E}^{(\text{out})}$,
- the weight matrices $\boldsymbol{W}_{h,*}$ for $h = 1, \ldots, H$ and $* =$ value, key, query in each multi-head attention module,
- the matrix $\boldsymbol{W}_{\text{conc}}$ in each multi-head attention module,
- the weights $g_j$ for $j = 1, \ldots, \tilde{d}$ in each layer normalization module,
- the weights and biases in each two-layer feed-forward module.

To determine these parameters, we use a training data set of pairs of input and corresponding output sequences, e.g., input sentences in a source language and their corresponding translation in a target language. A cross entropy loss function (see Section 6.4.5) is employed to compare the output probabilities after the final softmax layer of the transformer to the true output sequence from the training data. To this end, partial sequences, which only contain the start of each output sequence, are employed. In this way, the task becomes to predict the next word of the output sequence; see also the translation example below. Note that the output sequences from the training data set are also shifted by one word to the right and start with the placeholder "[start]". This is to make sure that the network can translate sentences from scratch without knowing the first word of the translation in the later generation phase, where only the input sequence is known. Finally, an SGD-type optimizer is used to minimize the cross entropy over all the training data in order to determine the free parameters of the transformer.

As an example, let us consider an English-to-German translator. During the training phase, we feed the network with pairs of English input sentences, e.g., "*I will meet you the day after tomorrow.*", and the corresponding German translations, e.g., "[start] *Ich werde dich übermorgen treffen.*" as output sentence. Regarding both sentences as word-by-word sequences, note that we included the placeholder [start] in the output sequence, which introduces the aforementioned shift. During training, we use all the partial sentences "[start]", "[start] *Ich*", "[start] *Ich werde*", "[start] *Ich werde dich*", "[start] *Ich werde dich übermorgen*", and "[start] *Ich werde dich übermorgen treffen*" of the output sequence, together with the whole input sequence, to train the transformer to predict the next word for each partial output sentence. Note that for the last output sentence "[start] *Ich werde dich übermorgen treffen*" the transformer needs to predict that the

translation is complete, i.e., it should predict a punctuation mark.[56] Since the transformer generates its output on a word by word basis, the optimal prediction from the network for the first output subsequence "[start]" would be "*Ich*", i.e., the word *Ich* should have the highest output probability among all words in the output language's dictionary. For the partial output sentence "[start] *Ich*" the transformer should predict "*werde*" and so on. By following this procedure for the whole training data set, we obtain prediction probabilities for all pairs of input sequences and corresponding partial output sequences. These probabilities are processed by the cross entropy loss function, which is then minimized with respect to the free parameters of the transformer.

**Generation phase** In the generation phase, we are only given an input sequence, e.g., a sentence in the source language, but not an output sequence, e.g., the corresponding translation in the target language. Thus, we have to generate the output sequence word by word from its beginning.

For our English-to-German translator from above this would mean that we are given the input sequence "*I will meet you the day after tomorrow.*" and no output sequence. Therefore, we evaluate the trained transformer for the given input sequence and the output sequence "[start]". Similar as in the training phase, the transformer now predicts the next word $a$ from the dictionary, i.e., $a$ is the word corresponding to the highest value in the transformer's output probabilities. In the optimal case, $a = \textit{Ich}$. Then, to continue the generation/translation of the output sentence, we evaluate the trained transformer again for the given input sequence but now with the output sequence "[start] $a$" to predict the next word. This process is continued until the transformer predicts that the sentence has ended, i.e., a punctuation mark or another end-of-sentence token has the highest probability. In this way, the transformer is employed to generate a whole output sequence for a given input sequence.

### 10.4.4 ▪ Further improvements

The revolutionary idea of the transformer was to trade the computational costs of $\mathcal{O}(d \cdot \tilde{d}^2)$ for applying the weight matrices[57] to a $\tilde{d}$-dimensional input (or hidden) vector in a recurrent neural network —an LSTM for instance—with an input sequence of length $d$ by the computational costs of $\mathcal{O}(d^2 \cdot \tilde{d})$ when computing the (self-)attention matrices in (10.6). Therefore, transformers work very well for short to moderate sequence lengths. For very large sequences, however, additional modifications have to be used to get rid of the $d^2$ factor in the computational costs; see, e.g., [DFE+22, KKL20, ZTS+21].

Moreover, pre-training transformer models on large databases of (unlabeled) text and then fine-tuning the resulting architecture for specific tasks like machine translation has shown to be a major improvement when solving many natural language processing tasks; see, e.g., [Bea20] for the GPT-3 (Generative Pre-trained Transformer) and [Ope23] for the GPT-4 models. The pre-training phase of the GPT algorithm employs a so-called *decoder-only* model, i.e., only the decoder part of the transformer is used and its cross-attention module is omitted. This means that there is no input sequence and the model is trained to infer the next word from the given part of the output sequence without any additional context. Such models are also commonly referred to as *foundation models*[58] because they have been pre-trained on a large training data set and can be easily adapted to tackle a variety of more specialized, so-called *downstream* tasks.

[56] Usually, a special end-of-sentence token is used instead of a punctuation mark.

[57] For an LSTM, for instance, these weight matrices are $A_*, B_*$ with $* \in \{f, u, c, o\}$ with input vectors $z_t$ with $\tilde{d} = d_i$ and hidden vectors $\boldsymbol{h}_t$ with $\tilde{d} = d_h$; see Section 10.3.2.

[58] Note that also the generative diffusion model for image generation from Section 8.4.4 is considered to be a foundation model in this sense.

Another improvement in many natural language processing tasks can be made when introducing *bidirectional* learning. Here, instead of inferring the next word in a sentence from the previous ones, a word is inferred from the whole context, i.e., from all other words in the sentence including the ones that appear later in the sentence. Together with pre-training on large databases, this is the basic idea of BERT (Bidirectional Encoder Representations from Transformers) [DCLT19] and RoBERTa (Robustly optimized BERT pre-training approach) [LOG+19]. These approaches employ *encoder-only* models, i.e., the decoder part of the transformer is omitted. To this end, to infer words from context, i.e., to predict a word at a given position, a final fully connected layer and a softmax activation are added to the encoder.

Most recently, OpenAI's ChatGPT[59] received a lot of attention. It is an AI chat-bot based on the GPT-3 model, i.e., an algorithm which reacts to both an input prompt from a user and the conversation history with that user by delivering an appropriate answer. Because of its decoder-only architecture, no source language is used for the GPT-3 model, and the conversation history is concatenated to the user prompt to obtain one long sequence that is fed to ChatGPT as the output sequence. Besides supervised learning on the GPT-3 model, ChatGPT's training process involved reinforcement learning from human feedback. Here, a so-called *proximal policy optimization* algorithm was employed. Besides ChatGPT, improved transformer models like GPT-3 and BERT have been the cornerstones for several other successful chat-bots and text-generation algorithms like Google's Bard,[60] Meta's Llama Chat,[61] AI21 Labs' Jurassic,[62] DeepMind's Sparrow,[63] and Jasper's Chat.[64] Overall, transformer-based architectures are the present state of the art when it comes to natural language processing tasks.

## 10.5 ▪ Interpretability

Usually machine learning algorithms are trained in such a way that they achieve a small loss. However, since metrics like the least squares error or the accuracy are not always sufficient to assess a trained model's quality in real-world applications, the demand for understanding how a specific trained model operates has risen drastically. Moreover, finding the underlying reasons for the produced decisions of a machine learning algorithm has become more and more important.

**Interpretability and explainability**

The topic of ***interpretability*** deals with methods and tools that are able to describe what the specific information is on which the algorithm bases its decision, e.g., which coordinates (and/or specific linear or nonlinear combinations thereof) of a data point $\boldsymbol{x} = (x_1, \ldots, x_d)$ are most important for the underlying machine learning model to come to the output $f(\boldsymbol{x})$.
Interpretability is closely connected to the topic of ***explainability***, where it is investigated if the algorithm's outcome is in coherence with contextual information and domain knowledge. For example, a model predicting a physical quantity should only produce results that are in accordance with the corresponding laws of physics. For more information on the topics of interpretability and explainability and for a survey of different approaches to these themes, we refer the reader to [Mol20, RBDG20].

[59] https://openai.com/blog/chatgpt/
[60] https://blog.google/technology/ai/bard-google-ai-search-updates
[61] https://ai.meta.com/llama/
[62] https://www.ai21.com/blog/announcing-ai21-studio-and-jurassic-1
[63] https://www.deepmind.com/blog/building-safer-dialogue-agents
[64] https://www.jasper.ai/chat

While some measures to achieve interpretability are already built into certain machine learning algorithms (see, e.g., [RPK19]), most interpretability methods are post-hoc in the sense that they are applied to the already trained model of a given machine learning algorithm.

The ultimate goals of applying interpretability methods can be quite diverse, e.g.,

- achieving more acceptance of machine learning algorithms in real-world application scenarios,
- disclosing security vulnerabilities, e.g., in autonomous driving [EEF+18],
- gaining scientific insight in the process underlying the measured data [RBDG20],
- detecting biases in the underlying data [Nea20],
- detecting unphysical parts of the machine learning model [RPK19].

While most interpretability methods usually cannot directly achieve these goals, their application provides a crucial first step in the direction of moving away from black-box machine learning models that are neither interpretable nor explainable to the user. Let us however emphasize that interpretability algorithms do not present a full transparent insight into the model's inner mechanisms. On the contrary, they usually just compute certain statistics (e.g., correlations) between input coordinates, features, and the output. There are more sophisticated approaches like causal machine learning models (see [Pea09, Sch22]), but such techniques are still quite novel areas of active research.

While the use of interpretability methods does not remedy negative properties that the underlying algorithm or data may have, it can still expose those properties and guide the user in the decision whether to trust a machine learning model or not. Furthermore, when we are able to distinguish if a model is explainable or not, we can use this information to validate our choices for the model design, e.g., the number of layers or neurons in deep learning.

We present in the rest of this section an introduction to easy-to-use interpretability algorithms that can be applied to a trained machine learning model. A more versatile discussion of these algorithms and more sophisticated variants of interpretability methods can be found in [Mol20].

### 10.5.1 ▪ Sensitivity analysis

Probably one of the most well-known interpretability methods is *sensitivity analysis*.

**Sensitivity analysis**

In ***sensitivity analysis*** the influence of an input coordinate (or a feature) is measured by quantifying how much the loss changes when changing the respective coordinate value. More generally, sensitivity analysis always answers the question of how robust a model is with respect to changes in the input.

We will here focus on the example of instance-based sensitivity analysis, which computes the sensitivities with respect to each input coordinate direction for one specific input data point $\boldsymbol{x}$. An alternative could be average sensitivity analysis, where the average importance of an input coordinate over all training data points is computed. Assume that the outcome of our trained machine learning model is the real-valued function $f$. Using notation similar to that in Section 6.3.1, let $C_{\boldsymbol{x}}(f)$ denote the loss on the data point $(\boldsymbol{x}, y)$. For instance, consider

$$C_{\boldsymbol{x}}(f) := (f(\boldsymbol{x}) - y)^2$$

for a least squares loss function. Then, the sensitivity $S$ in direction $j$ is computed as

$$S(f)_j := \left| \frac{\partial}{\partial x_j} C_{\boldsymbol{x}}(f) \right|.$$

Note that $x_j$ is here referring to the $j$th coordinate of the $d$-dimensional vector $\boldsymbol{x}$ and must not be confused with the $i$th training data point $\boldsymbol{x}_i$. After computing $S(f)_j$ for each $j = 1, \ldots, d$, the sensitivity with respect to each input coordinate $j$ is known. Often, the shorthand notation

$$S(f) := |\nabla_{\boldsymbol{x}} C_{\boldsymbol{x}}(f)|$$

is used to define $S$. Note that the absolute value $|\cdot|$ has to be understood componentwise here. Sometimes, the loss $C_{\boldsymbol{x}}$ is even discarded and the sensitivities are directly computed on the model's output $f(\boldsymbol{x})$ instead.

It now remains to compute the derivatives of $C_{\boldsymbol{x}}$ with respect to $\boldsymbol{x}$. In the general situation, this could only be done approximatively, e.g., by replacing the derivative $\frac{\partial}{\partial x_j} C_{\boldsymbol{x}}$ by some suitable finite difference for each $j = 1, \ldots, d$. It is noteworthy that, for the more sophisticated neural network models, we can directly use the automatic differentiation modules of deep learning libraries like KERAS and TENSORFLOW to compute the derivatives in the formula above. Thus, there is no need for additional implementational effort to calculate $S(f)$ in this case.

The quantity $S(f)$ tells us which coordinate directions are more important than others when it comes to minimizing the loss function for the specific data point $\boldsymbol{x}$. When we are dealing with image data, i.e., when $\boldsymbol{x}$ is an image, we can visualize the sensitivities as a *heat map*, sometimes also referred to as *saliency map*. For a gray-scale picture, each input coordinate represents a single pixel in the image. Thus, we can visualize the sensitivities by plotting them as color values (according to their magnitude) for each pixel. The same can be done for color images after averaging the sensitivities over the color channels, for instance. For an example see Figure 10.10.


Figure 10.10: An example of a heat map of sensitivity values of an image classification algorithm. As a model we used the pre-trained *GoogLeNet* [SLJ+15], which correctly classified the original image (left) as a cat. The sensitivities of the cat class output (right) have been averaged over the three color channels of the input picture. The color represents the magnitude of the sensitivity for each pixel. We see that the pixels which influence the loss the most are indeed the ones showing the cat's shape. (Cat image licensed under Pixabay Content License.)

### 10.5.2 ▪ Activation maximization

A special interpretability method for classification tasks, where a data point belongs to one of several output classes, is *activation maximization*, also known as *prototypes*.

**Activation maximization**

The task of finding a *prototypical* input for a certain output class is solved by ***activation maximization***. Here, an output class $c \in \Gamma$ is specified in a first step. Subsequently, a data point $\boldsymbol{x}$ is determined that the machine learning model $f$ most likely assigns to the specific class $c$, i.e., the probability $\mathbb{P}[f(\boldsymbol{x}) = c]$ that $\boldsymbol{x}$ belongs to the class $c$ is very large (or even the largest) compared to other possible inputs. The point $\boldsymbol{x}$ is also often called ***prototype*** for the class.

The naive way to compute a prototype of a class $c$ is to just maximize the probability of that class with respect to the input, i.e.,

$$\text{prototype}(c) := \underset{\boldsymbol{x} \in \mathbb{R}^d}{\arg\max}\, \mathbb{P}\left[f(\boldsymbol{x}) = c\right], \tag{10.7}$$

where $f : \mathbb{R}^d \to \Gamma$ is the classification model. However, without any further side conditions, the maximum in (10.7) might not be defined. This is due to the fact that the input coordinate values $(x_1, \ldots, x_d)$ of a data point $\boldsymbol{x} \in \mathbb{R}^d$ are unbounded, i.e., we could increase the size of the coordinates arbitrarily, which, depending on the model $f$, might lead to larger and larger probabilities $\mathbb{P}\left[f(\boldsymbol{x}) = c\right]$. Therefore, a regularization term like the (weighted) total variation of the input $\boldsymbol{x}$ is usually subtracted from the class probability in (10.7) to obtain a well-posed maximization problem and a more intuitive prototype; see [NYC16] for several examples of activation maximization methods and for more sophisticated variants that additionally take the training data distribution into account.

To compute a prototype we essentially need to maximize the class probability of a given machine learning model with respect to its input. This can be done by gradient ascent methods, for instance. To this end, we can again use the automatic differentiation modules of KERAS or TENSORFLOW for calculating prototypes of neural network models. In Figure 10.11 we depicted exemplary prototypes for certain classes of the *ImageNet* (https://www.image-net.org) data set.

### 10.5.3 ▪ More interpretability methods

Besides sensitivity analysis and prototypes there exists a plethora of other, also more sophisticated interpretability methods. A famous example is *LIME* (local interpretable model-agnostic explanations) [RSG16], where different simple models, e.g., linear functions, are used to approximate the original model and make it more accessible.

Another commonly used method based on the so-called *Shapley* values from game theory is called *SHAP* (SHapley Additive exPlanation) [LL17]. Here, the average marginal contribution of each input coordinate $(x_1, \ldots, x_d)$ of a data point $\boldsymbol{x} \in \mathbb{R}^d$ to the result of the model is approximated. This can be seen as a value for the benefit of including a single coordinate $x_j$ for a $j \in \{1, \ldots, d\}$ in the model instead of omitting it. In particular, the SHAP values quantify how much better a model performs when a certain coordinate, e.g., $x_j$ for a $j \in \{1, \ldots, d\}$, is considered in the training process compared to a model where the $j$th coordinate is removed from all data points before the training.

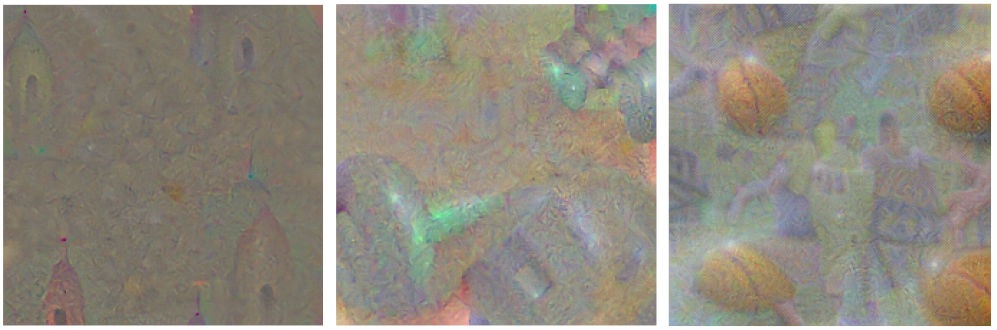


Figure 10.11: Three examples for prototypes of images for the three classes *church* (left), *dumbbell* (middle), and *basketball* (right) of the ImageNet data set. As model we used the pre-trained *GoogLeNet* [SLJ+15]. The prototypes have been computed by activation maximization with a total variation regularizer and gradient ascent. While the resulting prototype pictures do not show realistic scenarios, we can still observe the shapes of multiple church towers (left), dumbbells (middle), and basketballs (right).

For neural networks there exist specially designed interpretability methods like *layerwise relevance propagation* and *deep Taylor decomposition* [MBL+19]. Here, the output of the network is passed backwards through it in a specially weighted way depending on each neuron's activation. This can also be interpreted as a chain of Taylor expansions of first order in the neurons; see [MBL+19] for details.

More interpretability methods and the corresponding code packages can be found in [Mol20].

# Bibliography


[ACF22] M. Afsar, T. Crump, and B. Far. Reinforcement learning based recommender systems: A survey. *ACM Computing Surveys*, 55(7):1–38, 2022. (Cited on p. 139)

[ACJP20] E. Arias-Castro, A. Javanmard, and B. Pelletier. Perturbation bounds for Procrustes, classical scaling, and trilateration, with applications to manifold learning. *Journal of Machine Learning Research*, 21(15):1–37, 2020. (Cited on p. 71)

[ADHSK21] F. Abu-Dakka, Y. Huang, J. Silvério, and V. Kyrki. A probabilistic framework for learning geometry-based robot manipulation skills. *Robotics and Autonomous Systems*, 141:103761, 2021. (Cited on p. 169)

[ADLS10] G. Amaral, L. Dore, R. Lessa, and B. Stosic. K-means algorithm in statistical shape analysis. *Communications in Statistics—Simulation and Computation*, 39(5):1016–1026, 2010. (Cited on pp. 7, 77)

[Aea15] M. Abadi et al. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from https://www.tensorflow.org. (Cited on p. 105)

[Agg18] C. Aggarwal. *Neural Networks and Deep Learning*. Springer, Cham, Switzerland, 2018. (Cited on p. 87)

[AHK01] C. Aggarwal, A. Hinneburg, and D. Keim. On the surprising behavior of distance metrics in high dimensional space. In J. Van den Bussche and V. Vianu, editors, *Database Theory — ICDT 2001*, pages 420–434, Berlin, Heidelberg, Germany, 2001. Springer. (Cited on p. 13)

[And82] B. Anderson. Reverse-time diffusion equation models. *Stochastic Processes and Their Applications*, 12(3):313–326, 1982. (Cited on p. 135)

[AR20] M. Assran and M. Rabbat. On the convergence of Nesterov's accelerated gradient method in stochastic settings. In H. Daumé III and A. Singh, editors, *Proceedings of the 37th International Conference on Machine Learning, online*, volume 119 of *Proceedings of Machine Learning Research*, pages 410–420. PMLR, 2020. (Cited on p. 103)

[ARB19] M. Azaouzi, D. Rhouma, and L. Ben Romdhane. Community detection in large-scale social networks: State-of-the-art and future directions. *Social Network Analysis and Mining*, 9(1):23, 2019. (Cited on p. 7)

[Aro50] N. Aronszajn. Theory of reproducing kernels. *Transactions of the American Mathematical Society*, 68(3):337–404, 1950. (Cited on p. 47)

[ASB16] M. Arjovsky, A. Shah, and Y. Bengio. Unitary evolution recurrent neural networks. In M. Balcan and K. Weinberger, editors, *Proceedings of the 33rd International Conference on Machine Learning, New York, NY, USA*, volume 48 of *Proceedings of Machine Learning Research*, pages 1120–1128. PMLR, 2016. (Cited on p. 182)

[Au18] T. Au. Random forests, decision trees, and categorical predictors: The "absent levels" problem. *The Journal of Machine Learning Research*, 19(1):1737–1766, 2018. (Cited on p. 170)

[AZLS19] Z. Allen-Zhu, Y. Li, and Z. Song. A convergence theory for deep learning via over-parameterization. In K. Chaudhuri and R. Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning, Long Beach, CA, USA*, volume 97 of *Proceedings of Machine Learning Research*, pages 242–252. PMLR, 2019. (Cited on p. 129)

[Bal12] P. Baldi. Autoencoders, unsupervised learning, and deep architectures. In I. Guyon, G. Dror, V. Lemaire, G. Taylor, and D. Silver, editors, *Proceedings of ICML Workshop on Unsupervised and Transfer Learning, Bellevue, WA, USA*, volume 27 of *Proceedings of Machine Learning Research*, pages 37–49. PMLR, 2012. (Cited on p. 109)

[BB12] J. Bergstra and Y. Bengio. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13(10):281–305, 2012. (Cited on p. 49)

[BBS21] P. Bongini, M. Bianchini, and F. Scarselli. Molecular generative graph neural networks for drug discovery. *Neurocomputing*, 450:242–252, 2021. (Cited on p. 170)

[BCB15] D. Bahdanau, K. Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate. In *Proceedings of the 3rd International Conference on Learning Representations, San Diego, CA, USA*. ICLR, 2015. (Cited on p. 184)

[BCD97] M. Bardi and I. Capuzzo-Dolcetta. *Optimal Control and Viscosity Solutions of Hamilton-Jacobi-Bellman Equations*. Birkhäuser, Boston, MA, USA, 1997. (Cited on p. 139)

[BCN18] L. Bottou, F. Curtis, and J. Nocedal. Optimization methods for large-scale machine learning. *SIAM Review*, 60(2):223–311, 2018. (Cited on pp. 101, 102)

[BE21] J. Blechschmidt and O. Ernst. Three ways to solve partial differential equations with neural networks: A review. *GAMM Mitteilungen*, 44(2):e202100006, 2021. (Cited on p. 137)

[Bea20] T. Brown et al. Language models are few-shot learners. *Advances in Neural Information Processing Systems*, 33:1877–1901, 2020. (Cited on pp. 183, 190)

[Bel57] R. Bellman. *Dynamic Programming*. Princeton University Press, Princeton, NJ, USA, 1957. (Cited on pp. 144, 149)

[Bel61] R. Bellman. *Adaptive Control Processes: A Guided Tour*. Princeton University Press, Princeton, NJ, USA, 1961. (Cited on p. 12)

[Ber12] D. Bertsekas. *Dynamic Programming and Optimal Control: Approximate Dynamic Programming*, volume 2. Athena Scientific, Belmont, MA, USA, 4th edition, 2012. (Cited on pp. 144, 149, 151, 159, 165, 166)

[Ber17] D. Bertsekas. *Dynamic Programming and Optimal Control*, volume 1. Athena Scientific, Belmont, MA, USA, 4th edition, 2017. (Cited on pp. 144, 166)

[Ber19] D. Bertsekas. *Reinforcement Learning and Optimal Control*. Athena Scientific, Belmont, MA, USA, 2019. (Cited on pp. 139, 145, 149, 158, 159, 165, 166)

[Ber22] D. Bertsekas. *Abstract Dynamic Programming*. Athena Scientific, Belmont, MA, USA, 2022. (Cited on pp. 164, 165)

[Ber23] D. Bertsekas. *A Course in Reinforcement Learning*. Athena Scientific, Belmont, MA, USA, 2023. (Cited on p. 166)

[BG05] I. Borg and P. Groenen. *Modern Multidimensional Scaling: Theory and Applications*. Springer, New York, NY, USA, 2005. (Cited on p. 68)

[BGK22] B. Bohn, M. Griebel, and D. Kannan. Deep neural networks and PIDE discretizations. *SIAM Journal on Mathematics of Data Science*, 4(3):1145–1170, 2022. (Cited on p. 132)

[BGKP19] H. Bölcskei, P. Grohs, G. Kutyniok, and P. Petersen. Optimal approximation with sparsely connected deep neural networks. *SIAM Journal on Mathematics of Data Science*, 1(1):8–45, 2019. (Cited on p. 91)

[BGR19] B. Bohn, M. Griebel, and C. Rieger. A representer theorem for deep kernel learning. *Journal of Machine Learning Research*, 20(64):1–32, 2019. (Cited on p. 126)

[BHL93] G. Berkooz, P. Holmes, and J. Lumley. The proper orthogonal decomposition in the analysis of turbulent flows. *Annual Review of Fluid Mechanics*, 25(1):539–575, 1993. (Cited on p. 54)

[Bis06] C. Bishop. *Pattern Recognition and Machine Learning*. Springer, New York, NY, USA, 2006. (Cited on p. 16)

[BKH16] J. Ba, J. Kiros, and G. Hinton. Layer normalization. *arXiv preprint, arXiv:1607.06450*, 2016. (Cited on p. 188)

[BMDG05] A. Banerjee, S. Merugu, I. Dhillon, and J. Ghosh. Clustering with Bregman divergences. *Journal of Machine Learning Research*, 6(10):1705–1749, 2005. (Cited on p. 115)

[BMFB+22] L. Brogat-Motte, R. Flamary, C. Brouard, J. Rousu, and F. D'Alché-Buc. Learning to predict graphs with fused Gromov-Wasserstein barycenters. In K. Chaudhuri, S. Jegelka, L. Song, C. Szepesvari, G. Niu, and S. Sabato, editors, *Proceedings of the 39th International Conference on Machine Learning, Baltimore, MD, USA*, volume 162 of *Proceedings of Machine Learning Research*, pages 2321–2335. PMLR, 2022. (Cited on p. 170)

[BMR+17] T. Brown, D. Mané, A. Roy, M. Abadi, and J. Gilmer. Adversarial patch. *arXiv preprint, arXiv:1712.09665*, 2017. (Cited on p. 122)

[BMZ09] O. Bokanowski, S. Maroso, and H. Zidani. Some convergence results for Howard's algorithm. *SIAM Journal on Numerical Analysis*, 47(4):3001–3026, 2009. (Cited on p. 149)

[BNC+15] F. Buettner, K. Natarajan, F. Casale, V. Proserpio, A. Scialdone, F. Theis, S. Teichmann, J. Marioni, and O. Stegle. Computational analysis of cell-to-cell heterogeneity in single-cell RNA-sequencing data reveals hidden subpopulations of cells. *Nature Biotechnology*, 33(2):155–160, 2015. (Cited on pp. 81, 82)

[BOHS15] R. Benenson, M. Omran, J. Hosang, and B. Schiele. Ten years of pedestrian detection, what have we learned? In L. Agapito, M. Bronstein, and C. Rother, editors, *Computer Vision - ECCV 2014 Workshops*, pages 613–627, Cham, Switzerland, 2015. Springer. (Cited on pp. 62, 63)

[BPRS18] A. Baydin, B. Pearlmutter, A. Radul, and J. Siskind. Automatic differentiation in machine learning: A survey. *Journal of Machine Learning Research*, 18(153):1–43, 2018. (Cited on p. 95)

[Bre84] L. Breiman. *Classification and Regression Trees*. Routledge, New York, NY, USA, 1984. (Cited on pp. 176, 178)

[Bre96a] L. Breiman. Bagging predictors. *Machine Learning*, 24:123–140, 1996. (Cited on p. 175)

[Bre96b] L. Breiman. Stacked regressions. *Machine Learning*, 24:49–64, 1996. (Cited on p. 172)

[Bre01] L. Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001. (Cited on pp. 176, 178)

[BS96] D. Bertsekas and S. Shreve. *Stochastic Optimal Control: The Discrete-Time Case*. Athena Scientific, Belmont, MA, USA, 1996. (Cited on p. 150)

[BSZG18] A. Bojchevski, O. Shchur, D. Zügner, and S. Günnemann. NetGAN: Generating graphs via random walks. In J. Dy and A. Krause, editors, *Proceedings of the 35th International Conference on Machine Learning, Stockholm, Sweden*, volume 80 of *Proceedings of Machine Learning Research*, pages 610–619. PMLR, 2018. (Cited on p. 170)

[BV04] S. Boyd and L. Vandenberghe. *Convex Optimization*. Cambridge University Press, New York, NY, USA, 2004. (Cited on pp. 27, 35, 36, 37)

[BYF+19] M. Budninskiy, G. Yin, L. Feng, Y. Tong, and M. Desbrun. Parallel transport unfolding: A connection-based manifold learning approach. *SIAM Journal on Applied Algebra and Geometry*, 3(2):266–291, 2019. (Cited on p. 85)

[CAEHP+22] I. Chami, S. Abu-El-Haija, B. Perozzi, C. Ré, and K. Murphy. Machine learning on graphs: A model and comprehensive taxonomy. *Journal of Machine Learning Research*, 23(89):1–64, 2022. (Cited on p. 170)

[Cal20] O. Calin. *Deep Learning Architectures*. Springer, Cham, Switzerland, 2020. (Cited on pp. 16, 108)

[CBB17] M. Congedo, A. Barachant, and R. Bhatia. Riemannian geometry for EEG-based brain-computer interfaces; a primer and a review. *Brain-Computer Interfaces*, 4:1–20, 2017. (Cited on p. 169)

[CBMV20] R. Chakraborty, J. Bouza, J. Manton, and B. Vemuri. Manifoldnet: A deep neural network for manifold-valued data with applications. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 44(2):799–810, 2020. (Cited on p. 169)

[CC00] T. Cox and M. Cox. *Multidimensional Scaling*. Chapmann & Hall, New York, NY, USA, 2nd edition, 2000. (Cited on p. 59)

[CCHC19] B. Chang, M. Chen, E. Haber, and E. Chi. AntisymmetricRNN: A dynamical system view on recurrent neural networks. In *Proceedings of the 7th International Conference on Learning Representations, New Orleans, LA, USA*. ICLR, 2019. (Cited on p. 183)

[Cea15] F. Chollet et al. Keras. https://keras.io, 2015. (Cited on p. 105)

[CFV22] A. Calissano, A. Feragen, and S. Vantini. Graph-valued regression: Prediction of unlabelled networks in a non-Euclidean graph space. *Journal of Multivariate Analysis*, 190:104950, 2022. (Cited on p. 170)

[CGIT23] P. Climaco, J. Garcke, and R. Iza-Teran. Multi-resolution dynamic mode decomposition for damage detection in wind turbine gearboxes. *Data-Centric Engineering*, 4:e1, 2023. (Cited on p. 5)

[Cho17] F. Chollet. *Deep Learning with Python*. Manning Publications, Shelter Island, NY, USA, 2017. (Cited on p. 16)

[CKBM21] S. Chevallier, E. Kalunga, Q. Barthélemy, and E. Monacelli. Review of Riemannian distances and divergences, applied to SSVEP-based BCI. *Neuroinformatics*, 19(1):93–106, 2021. (Cited on p. 169)

[CL06] R. Coifman and S. Lafon. Diffusion maps. *Applied and Computational Harmonic Analysis*, 21(1):5–30, 2006. (Cited on pp. 72, 73, 75, 84)

[CL11] C.-C. Chang and C.-J. Lin. LIBSVM: A library for support vector machines. *ACM Trans. Intell. Syst. Technol.*, 2(3), 2011. (Cited on pp. 44, 51)

[COSJ17] Y. Costa, L. Oliveira, and C. Silla Jr. An evaluation of convolutional neural networks for music classification using spectrograms. *Applied Soft Computing*, 52:28–38, 2017. (Cited on p. 5)

[Cov65] T. Cover. Geometrical and statistical properties of systems of linear inequalities with applications in pattern recognition. *IEEE Transactions on Electronic Computers*, 3:326–334, 1965. (Cited on p. 44)

[CPK+18] L.-C. Chen, G. Papandreou, I. Kokkinos, K. Murphy, and A. Yuille. DeepLab: Semantic image segmentation with deep convolutional nets, atrous convolution, and fully connected CRFs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 40(4):834–848, 2018. (Cited on p. 130)

[CRBD18] R. Chen, Y. Rubanova, J. Bettencourt, and D. Duvenaud. Neural ordinary differential equations. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31, Red Hook, NY, USA, 2018. Curran Associates, Inc. (Cited on pp. 125, 132, 133)

[CV95] C. Cortes and V. Vapnik. Support-vector networks. *Machine Learning*, 20(3):273–297, 1995. (Cited on p. 38)

[CvMG+14] K. Cho, B. van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio. Learning phrase representations using RNN encoder–decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP), Doha, Qatar*, pages 1724–1734. Association for Computational Linguistics, 2014. (Cited on pp. 108, 179, 182)

[Cyb89] G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems*, 2(4):303–314, 1989. (Cited on p. 91)

[CZ07] F. Cucker and D. Zhou. *Learning Theory*. Cambridge University Press, Cambridge, UK, 2007. (Cited on pp. 16, 52)

[Dah22] W. Dahmen. Compositional sparsity, approximation classes, and parametric transport equations. *arXiv preprint, arXiv:2207.06128*, 2022. (Cited on p. 91)

[DBC+19] E. Dada, J. Bassi, H. Chiroma, S. Abdulhamid, A. Adetunmbi, and O. Ajibuwa. Machine learning for email spam filtering: Review, approaches and open research problems. *Heliyon*, 5(6):e01802, 2019. (Cited on p. 5)

[DBK+16] Y. Deng, F. Bao, Y. Kong, Z. Ren, and Q. Dai. Deep direct reinforcement learning for financial signal representation and trading. *IEEE Transactions on Neural Networks and Learning Systems*, 28(3):653–664, 2016. (Cited on p. 139)

[dC92] M. do Carmo. *Riemannian Geometry*. Birkhäuser, Boston, MA, USA, 1992. (Cited on p. 70)

[DCLT19] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Minneapolis, MN, USA*, pages 4171–4186. Association for Computational Linguistics, 2019. (Cited on pp. 183, 191)

[DD09] M. Deza and E. Deza. *Encyclopedia of Distances*. Springer, Berlin, Heidelberg, Germany, 2009. (Cited on pp. 7, 167)

[Dea22] J. Degrave et al. Magnetic control of tokamak plasmas through deep reinforcement learning. *Nature*, 602(7897):414–419, 2022. (Cited on pp. 139, 166)

[DFE+22] T. Dao, D. Fu, S. Ermon, A. Rudra, and C. Ré. FlashAttention: Fast and memory-efficient exact attention with IO-awareness. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems*, volume 35, pages 16344–16359, Red Hook, NY, USA, 2022. Curran Associates, Inc. (Cited on p. 190)

[DFO20] M. Deisenroth, A. Faisal, and C. Ong. *Mathematics for Machine Learning*. Cambridge University Press, Cambridge, UK, 2020. (Cited on p. 16)

[DG17] D. Dua and C. Graff. UCI machine learning repository. http://archive.ics.uci.edu/ml, 2017. (Cited on p. 25)

[DHP21] R. DeVore, B. Hanin, and G. Petrova. Neural network approximation. *Acta Numerica*, 30:327–444, 2021. (Cited on p. 91)

[DHS11] J. Duchi, E. Hazan, and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(61):2121–2159, 2011. (Cited on p. 104)

[Dij59] E. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959. (Cited on p. 71)

[Doe16] C. Doersch. Tutorial on variational autoencoders. *arXiv preprint, arXiv:1606.05908*, 2016. (Cited on pp. 109, 117)

[DOL03] M. De Oliveira and H. Levkowitz. From visual data exploration to visual data mining: A survey. *IEEE Transactions on Visualization and Computer Graphics*, 9(3):378–394, 2003. (Cited on p. 79)

[Doz16] T. Dozat. Incorporating Nesterov momentum into Adam. In *Proceedings of 4th International Conference on Learning Representations, Workshop Track, San Juan, Puerto Rico*, 2016. https://openreview.net/forum?id=OM0jvwB8jIp57ZJjtNEZ. (Cited on p. 105)

[DT05] N. Dalal and B. Triggs. Histograms of oriented gradients for human detection. In *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, San Diego, CA, USA*, volume 1, pages 886–893. IEEE, 2005. (Cited on p. 63)

[DTRF19] X. Dong, D. Thanou, M. Rabbat, and P. Frossard. Learning graphs from data: A signal representation perspective. *IEEE Signal Processing Magazine*, 36(3):44–63, 2019. (Cited on p. 170)

[DWSP12] P. Dollar, C. Wojek, B. Schiele, and P. Perona. Pedestrian detection: An evaluation of the state of the art. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 34(4):743–761, 2012. (Cited on pp. 62, 63)

[DWW22] W. Dahmen, M. Wang, and Z. Wang. Nonlinear reduced DNN models for state estimation. *Communications in Computational Physics*, 32(1):1–40, 2022. (Cited on p. 137)

[DYC+20] X. Dong, Z. Yu, W. Cao, Y. Shi, and Q. Ma. A survey on ensemble learning. *Frontiers of Computer Science*, 14(2):241–258, 2020. (Cited on p. 171)

[EAQ+21] B. Erichson, O. Azencot, A. Queiruga, L. Hodgkinson, and M. Mahoney. Lipschitz recurrent neural networks. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event*. OpenReview.net, 2021. (Cited on p. 183)

[EEF+18] K. Eykholt, I. Evtimov, E. Fernandes, B. Li, A. Rahmati, C. Xiao, A. Prakash, T. Kohno, and D. Song. Robust physical-world attacks on deep learning visual classification. In *Proceedings of the 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition, Salt Lake City, UT, USA*, pages 1625–1634. IEEE, 2018. (Cited on pp. 122, 192)

[EHJ21] W. E, J. Han, and A. Jentzen. Algorithms for solving high dimensional PDEs: From nonlinear Monte Carlo to machine learning. *Nonlinearity*, 35(1):278–310, 2021. (Cited on p. 137)

[EHN96] H. Engl, M. Hanke, and A. Neubauer. *Regularization of Inverse Problems*. Springer, Dordrecht, The Netherlands, 1996. (Cited on p. 11)

[Ein56] A. Einstein. *Investigations on the Theory of the Brownian Movement*. Dover Publications, Mineola, NY, USA, 1956. (Cited on p. 134)

[Elm90] J. Elman. Finding structure in time. *Cognitive Science*, 14(2):179–211, 1990. (Cited on p. 179)

[EMW20] W. E, C. Ma, and L. Wu. Machine learning from a continuous viewpoint, I. *Science China Mathematics*, 63(11):2233–2266, 2020. (Cited on p. 125)

[EMW22] W. E, C. Ma, and L. Wu. The Barron space and the flow-induced function spaces for neural network models. *Constructive Approximation*, 55(1):369–406, 2022. (Cited on p. 92)

[EW21] W. E and S. Wojtowytsch. Kolmogorov width decay and poor approximators in machine learning: Shallow neural networks, random feature models and neural tangent kernels. *Research in the Mathematical Sciences*, 8(1):1–28, 2021. (Cited on p. 129)

[FCH+08] R.-E. Fan, K.-W. Chang, C.-J. Hsieh, X.-R. Wang, and C.-J. Lin. LIBLINEAR: A library for large linear classification. *Journal of Machine Learning Research*, 9:1871–1874, 2008. (Cited on p. 51)

[Fea22] A. Fawzi et al. Discovering faster matrix multiplication algorithms with reinforcement learning. *Nature*, 610:47–53, 2022. (Cited on p. 139)

[FF13] M. Falcone and R. Ferretti. *Semi-Lagrangian Approximation Schemes for Linear and Hamilton–Jacobi Equations*. SIAM, Philadelphia, PA, USA, 2013. (Cited on p. 145)

[Fis36] R. Fisher. The use of multiple measurements in taxonomic problems. *Annals of Eugenics*, 7(2):179–188, 1936. (Cited on p. 25)

[Flo62] R. Floyd. Algorithm 97: Shortest path. *Communications of the ACM*, 5(6):345, 1962. (Cited on p. 71)

[Fri01] J. Friedman. Greedy function approximation: A gradient boosting machine. *Annals of Statistics*, 29(5):1189–1232, 2001. (Cited on p. 174)

[FS97] Y. Freund and R. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of Computer and System Sciences*, 55(1):119–139, 1997. (Cited on p. 174)

[FS06] W. Fleming and H. Soner. *Controlled Markov Processes and Viscosity Solutions*. Springer, New York, NY, USA, 2006. (Cited on p. 150)

[FZM+15] C. Frogner, C. Zhang, H. Mobahi, M. Araya, and T. Poggio. Learning with a Wasserstein loss. In C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 28, Red Hook, NY, USA, 2015. Curran Associates, Inc. (Cited on p. 167)

[GA11] M. Gönen and E. Alpaydın. Multiple kernel learning algorithms. *Journal of Machine Learning Research*, 12:2211–2268, 2011. (Cited on p. 52)

[GB10] X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. In Y. Teh and M. Titterington, editors, *Proceedings of the 13th International Conference on Artificial Intelligence and Statistics, Chia Laguna Resort, Sardinia, Italy*, volume 9 of *Proceedings of Machine Learning Research*, pages 249–256. PMLR, 2010. (Cited on p. 93)

[GBC16] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, Cambridge, MA, USA, 2016. http://www.deeplearningbook.org. (Cited on pp. 16, 87, 95, 96, 108, 109, 115, 178, 180, 182)

[GDG+17] P. Goyal, P. Dollár, R. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He. Accurate, large minibatch SGD: Training Imagenet in 1 hour. *arXiv preprint, arXiv:1706.02677*, 2017. (Cited on p. 108)

[Geo12] H.-O. Georgii. *Stochastics: Introduction to Probability and Statistics*. De Gruyter, Berlin, Germany; New York, NY, USA, 2012. (Cited on pp. 73, 84, 118)

[Gér19] A. Géron. *Hands-on Machine Learning with Scikit-Learn, Keras, and TensorFlow*. O'Reilly, Sebastopol, CA, USA, 2019. (Cited on p. 16)

[GGF09] R. Goni, P. García, and S. Foissac. The qPCR data statistical analysis. Technical report, Integromics SL, https://www.gene-quantification.com/integromics-qpcr-statistics-white-paper.pdf, 2009. (Cited on p. 82)

[GHR20] E. Gorbunov, F. Hanzely, and P. Richtárik. A unified theory of SGD: Variance reduction, sampling, quantization and coordinate descent. In S. Chiappa and R. Calandra, editors, *Proceedings of the 23rd International Conference on Artificial Intelligence and Statistics, online*, volume 108 of *Proceedings of Machine Learning Research*, pages 680–690. PMLR, 2020. (Cited on pp. 101, 102)

[GHT+10] G. Guo, M. Huss, G. Tong, C. Wang, L. Sun, N. Clarke, and P. Robson. Resolution of cell fate decisions revealed by single-cell gene expression analysis from zygote to blastocyst. *Developmental Cell*, 18(4):675–685, 2010. (Cited on pp. 81, 82)

[GK22] P. Grohs and G. Kutyniok, editors. *Mathematical Aspects of Deep Learning*. Cambridge University Press, Cambridge, UK, 2022. (Cited on p. 91)

[GLB+21] L. Girin, S. Leglaive, X. Bie, J. Diard, T. Hueber, and X. Alameda-Pineda. Dynamical variational autoencoders: A comprehensive review. *Foundations and Trends in Machine Learning*, 15(1–2):1–175, 2021. (Cited on p. 123)

[GP90] F. Girosi and T. Poggio. Networks and the best approximation property. *Biological Cybernetics*, 63(3):169–176, 1990. (Cited on p. 91)

[GP17] L. Grüne and J. Pannek. *Nonlinear Model Predictive Control*. Springer, Cham, Switzerland, 2017. (Cited on pp. 139, 165)

[GPAM+14] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. Generative adversarial nets. In Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 27, pages 2672–2680, Red Hook, NY, USA, 2014. Curran Associates, Inc. (Cited on p. 122)

[GR70] G. Golub and C. Reinsch. Singular value decomposition and least squares solutions. *Numerische Mathematik*, 14:403–420, 1970. (Cited on p. 54)

[GTGHS20] N. García Trillos, M. Gerlach, M. Hein, and D. Slepčev. Error estimates for spectral convergence of the graph Laplacian on random geometric graphs toward the Laplace–Beltrami operator. *Foundations of Computational Mathematics*, 20:827–887, 2020. (Cited on p. 73)

[GVL13] G. Golub and C. Van Loan. *Matrix Computations*, 4th edition. JHU Press, Baltimore, MD, USA, 2013. (Cited on pp. 18, 19, 20, 21, 57, 58)

[GW08] A. Griewank and A. Walther. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*, 2nd edition. SIAM, Philadelphia, PA, USA, 2008. (Cited on p. 95)

[HA04] V. Hodge and J. Austin. A survey of outlier detection methodologies. *Artificial Intelligence Review*, 22(2):85–126, 2004. (Cited on p. 37)

[Hac13] W. Hackbusch. *Multi-Grid Methods and Applications*. Springer Science & Business Media, Berlin, Germany, 2013. (Cited on p. 28)

[Hac16] W. Hackbusch. *Iterative Solution of Large Sparse Systems of Equations*, 2nd edition. Springer, Cham, Switzerland, 2016. (Cited on p. 28)

[HBT15] L. Haghverdi, F. Buettner, and F. Theis. Diffusion maps for high-dimensional single-cell analysis of differentiation data. *Bioinformatics*, 31(18):2989–2998, 2015. (Cited on p. 75)

[Hea23] L. Heumos et al. Best practices for single-cell analysis across modalities. *Nature Reviews Genetics*, 24:550–572, 2023. (Cited on p. 82)

[HGG+20] M. Hennig, M. Grafinger, D. Gerhard, S. Dumss, and P. Rosenberger. Comparison of time series clustering algorithms for machine state detection. In *Procedia CIRP*, volume 93, pages 1352–1357, Chicago, IL, USA, 2020. Elsevier B.V. (Cited on p. 7)

[Hig02] N. Higham. *Accuracy and Stability of Numerical Algorithms*, 2nd edition. SIAM, Philadelphia, PA, USA, 2002. (Cited on pp. 19, 20, 21)

[HJA20] J. Ho, A. Jain, and P. Abbeel. Denoising diffusion probabilistic models. In H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 6840–6851, Red Hook, NY, USA, 2020. Curran Associates, Inc. (Cited on pp. 125, 134, 136, 137, 138)

[HJKN20] M. Hutzenthaler, A. Jentzen, T. Kruse, and T. Nguyen. A proof that rectified deep neural networks overcome the curse of dimensionality in the numerical approximation of semilinear heat equations. *Partial Differential Equations and Applications*, 1(2):1–34, 2020. (Cited on p. 137)

[HK20] J. Hancock and T. Khoshgoftaar. Survey on categorical data for neural networks. *Journal of Big Data*, 7(28):1–41, 2020. (Cited on p. 170)

[HMvH+18] M. Hessel, J. Modayil, H. van Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. Azar, and D. Silver. Rainbow: Combining improvements in deep reinforcement learning. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence and Thirtieth Innovative Applications of Artificial Intelligence Conference and Eighth AAAI Symposium on Educational Advances in Artificial Intelligence, New Orleans, LA, USA*, AAAI'18/IAAI'18/EAAI'18. AAAI Press, 2018. (Cited on p. 165)

[Ho95] T. Ho. Random decision forests. In *Proceedings of 3rd International Conference on Document Analysis and Recognition, Montreal, Canada*, volume 1, pages 278–282. IEEE Computer Society, 1995. (Cited on p. 176)

[Hor91] K. Hornik. Approximation capabilities of multilayer feedforward networks. *Neural Networks*, 4(2):251–257, 1991. (Cited on p. 91)

[Hot33] H. Hotelling. Analysis of a complex of statistical variables into principal components. *Journal of Educational Psychology*, 24(6):417, 1933. (Cited on p. 54)

[How60] R. Howard. *Dynamic Programming and Markov Processes*. MIT Press, Cambridge, MA, USA, 1960. (Cited on p. 149)

[HPS12] H. Harbrecht, M. Peters, and R. Schneider. On the low-rank approximation by the pivoted Cholesky decomposition. *Applied Numerical Mathematics*, 62(4):428–440, 2012. (Cited on p. 57)

[HR18] E. Haber and L. Ruthotto. Stable architectures for deep neural networks. *Inverse Problems*, 34(1), 2018. 014004. (Cited on pp. 125, 131)

[HRZZ09] T. Hastie, S. Rosset, J. Zhu, and H. Zou. Multi-class AdaBoost. *Statistics and Its Interface*, 2(3):349–360, 2009. (Cited on p. 174)

[HS97] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Computation*, 9(8): 1735–1780, 1997. (Cited on pp. 108, 181, 182)

[HSK19] B. Henrique, V. Sobreiro, and H. Kimura. Literature review: Machine learning techniques applied to financial market prediction. *Expert Systems with Applications*, 124:226–251, 2019. (Cited on p. 5)

[HSL16] M. Henaff, A. Szlam, and Y. LeCun. Recurrent orthogonal networks and long-memory tasks. In M. Balcan and K. Weinberger, editors, *Proceedings of the 33rd International Conference on Machine Learning, New York, NY, USA*, volume 48 of *Proceedings of Machine Learning Research*, pages 2034–2042. PMLR, 2016. (Cited on p. 182)

[HTF09] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*. Springer Series in Statistics. Springer, New York, NY, USA, 2009. (Cited on pp. 11, 12, 16, 17, 30, 31, 49, 51, 77, 170, 171, 174)

[Hul14] J. Hull. *Options, Futures, and Other Derivatives*, 9th edition. Pearson, Boston, MA, USA, 2014. (Cited on p. 134)

[HWVG19] Z. Huang, J. Wu, and L. Van Gool. Manifold-valued image generation with Wasserstein generative adversarial nets. In *Proceedings of the 33rd AAAI Conference on Artificial Intelligence, Honolulu, HI, USA*, volume 33, pages 3886–3893. AAAI Press, 2019. (Cited on pp. 167, 169)

[HZRS16] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Las Vegas, NV, USA*, pages 770–778, 2016. (Cited on p. 130)

[Jea17] N. Jouppi et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture, Toronto, Canada*, pages 1–12, New York, NY, USA, 2017. Association for Computing Machinery. (Cited on p. 108)

[JGH18] A. Jacot, F. Gabriel, and C. Hongler. Neural tangent kernel: Convergence and generalization in neural networks. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31, pages 8580–8589, Red Hook, NY, USA, 2018. Curran Associates, Inc. (Cited on pp. 125, 126, 129)

[JHS+13] S. Jayasumana, R. Hartley, M. Salzmann, H. Li, and M. Harandi. Kernel methods on the Riemannian manifold of symmetric positive definite matrices. In *Proceedings of the 2013 IEEE Conference on Computer Vision and Pattern Recognition, Portland, OR, USA*, pages 73–80. IEEE Computer Society, 2013. (Cited on pp. 168, 169)

[Jol02] I. Jolliffe. *Principal Component Analysis*. Springer, New York, NY, USA, 2002. (Cited on pp. 54, 60)

[JRCC21] N. Jaquier, L. Rozo, D. Caldwell, and S. Calinon. Geometry-aware manipulability learning, tracking, and transfer. *The International Journal of Robotics Research*, 40(2-3):624–650, 2021. (Cited on p. 169)

[JRF12] G. Jurman, S. Riccadonna, and C. Furlanello. A comparison of MCC and CEN error measures in multi-class prediction. *PLOS ONE*, 7(8):1–8, 2012. (Cited on p. 24)

[JWHT21] G. James, D. Witten, T. Hastie, and R. Tibshirani. *An Introduction to Statistical Learning*, 2nd edition. Springer, New York, NY, USA, 2021. (Cited on pp. 16, 49, 51, 77, 171)

[KB14] D. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint, arXiv: 1412.6980*, 2014. (Cited on p. 104)

[KBP13] J. Kober, J. Bagnell, and J. Peters. Reinforcement learning in robotics: A survey. *The International Journal of Robotics Research*, 32(11):1238–1274, 2013. (Cited on p. 140)

[KBPA+19] P. Kidger, P. Bonnier, I. Perez Arribas, C. Salvi, and T. Lyons. Deep signature transforms. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32, Red Hook, NY, USA, 2019. Curran Associates, Inc. (Cited on p. 171)

[KDP+16] H. Kadri, E. Duflos, P. Preux, S. Canu, A. Rakotomamonjy, and J. Audiffren. Operator-valued kernels for learning from functional response data. *Journal of Machine Learning Research*, 17(20):1–54, 2016. (Cited on p. 170)

[KGPT+19] G. Kerg, K. Goyette, M. Puelma Touzel, G. Gidel, E. Vorontsov, Y. Bengio, and G. Lajoie. Non-normal recurrent neural network (NNRNN): Learning long time dependencies while improving expressivity with transient dynamics. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32, Red Hook, NY, USA, 2019. Curran Associates, Inc. (Cited on p. 182)

[Kid22] P. Kidger. *On Neural Differential Equations*. Doctoral thesis, Mathematical Institute, University of Oxford, Oxford, UK, 2022. (Cited on p. 133)

[KJM20] N. Kriege, F. Johansson, and C. Morris. A survey on graph kernels. *Applied Network Science*, 5(1):6, 2020. (Cited on p. 52)

[KKL20] N. Kitaev, Ł. Kaiser, and A. Levskaya. Reformer: The efficient transformer. *arXiv preprint, arXiv:2001.04451*, 2020. (Cited on p. 190)

[KLL+23] N. Kovachki, Z. Li, B. Liu, K. Azizzadenesheli, K. Bhattacharya, A. Stuart, and A. Anandkumar. Neural operator: Learning maps between function spaces with applications to PDEs. *Journal of Machine Learning Research*, 24(89):1–97, 2023. (Cited on pp. 138, 170)

[KLM21] N. Kovachki, S. Lanthaler, and S. Mishra. On universal approximation and error bounds for Fourier neural operators. *The Journal of Machine Learning Research*, 22(1):13237–13312, 2021. (Cited on pp. 138, 170)

[KP92] P. Kloeden and E. Platen. *Numerical Solution of Stochastic Differential Equations*. Springer, Berlin, Heidelberg, Germany, 1992. (Cited on pp. 134, 135)

[KS90] M. Kirby and L. Sirovich. Application of the Karhunen-Loeve procedure for the characterization of human faces. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12(1):103–108, 1990. (Cited on p. 54)

[KSH12] A. Krizhevsky, I. Sutskever, and G. Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. Burges, L. Bottou, and K. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25, pages 1097–1105, Red Hook, NY, USA, 2012. Curran Associates, Inc. (Cited on pp. 5, 96, 108)

[KSZ96] J. Klafter, M. Shlesinger, and G. Zumofen. Beyond Brownian motion. *Physics Today*, 49(2):33–39, 1996. (Cited on p. 134)

[KW70] G. Kimeldorf and G. Wahba. A correspondence between Bayesian estimation on stochastic processes and smoothing by splines. *The Annals of Mathematical Statistics*, 41(2):495–502, 1970. (Cited on p. 47)

[KW13] D. Kingma and M. Welling. Auto-encoding variational Bayes. *arXiv preprint, arXiv:1312.6114*, 2013. (Cited on pp. 109, 117)

[KW19] D. Kingma and M. Welling. An introduction to variational autoencoders. *Foundations and Trends in Machine Learning*, 12(4):307–392, 2019. (Cited on pp. 109, 117)

[KZK12] J. Kruppa, A. Ziegler, and I. König. Risk estimation and risk prediction using machine-learning methods. *Human Genetics*, 131(10):1639–1654, 2012. (Cited on p. 5)

[Lan12] S. Lang. *A First Course in Calculus*. Springer, New York, NY, USA, 2012. (Cited on p. 128)

[LeC85] Y. LeCun. Une procedure d'apprentissage pour reseau a seuil asymetrique. In *Proceedings of Cognitiva 85, Paris, France*, pages 599–604, 1985. (Cited on p. 91)

[Led01] M. Ledoux. *The Concentration of Measure Phenomenon*. American Mathematical Society, Providence, RI, USA, 2001. (Cited on p. 12)

[Lin76] S. Linnainmaa. Taylor expansion of the accumulated rounding error. *BIT Numerical Mathematics*, 16(2):146–160, 1976. (Cited on p. 91)

[LJP+21] L. Lu, P. Jin, G. Pang, Z. Zhang, and G. Karniadakis. Learning nonlinear operators via DeepONet based on the universal approximation theorem of operators. *Nature Machine Intelligence*, 3(3):218–229, 2021. (Cited on pp. 138, 170)

[LKA+20] Z. Li, N. Kovachki, K. Azizzadenesheli, B. Liu, K. Bhattacharya, A. Stuart, and A. Anandkumar. Fourier neural operator for parametric partial differential equations. *arXiv preprint, arXiv:2010.08895*, 2020. (Cited on pp. 138, 170)

[LL17] S. Lundberg and S.-I. Lee. A unified approach to interpreting model predictions. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 4765–4774. Curran Associates, Inc., Red Hook, NY, USA, 2017. (Cited on p. 194)

[LLHZ16] S. Lai, K. Liu, S. He, and J. Zhao. How to generate a good word embedding. *IEEE Intelligent Systems*, 31(6):5–14, 2016. (Cited on p. 170)

[Llo82] S. Lloyd. Least squares quantization in PCM. *IEEE Transactions on Information Theory*, 28(2):129–137, 1982. (Cited on p. 77)

[LLWT15] Z. Liu, P. Luo, X. Wang, and X. Tang. Deep learning face attributes in the wild. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV), Santiago, Chile*. IEEE, 2015. (Cited on p. 138)

[LMK22] S. Lanthaler, S. Mishra, and G. Karniadakis. Error estimates for DeepONets: A deep learning framework in infinite dimensions. *Transactions of Mathematics and Its Applications*, 6(1):tnac001, 2022. (Cited on pp. 138, 170)

[LMW+17] S. Liu, D. Maljovec, B. Wang, P.-T. Bremer, and V. Pascucci. Visualizing high-dimensional data: Advances in the past decade. *IEEE Transactions on Visualization and Computer Graphics*, 23(3):1249–1268, 2017. (Cited on p. 79)

[LOG+19] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov. RoBerta: A robustly optimized Bert pretraining approach. *arXiv preprint, arXiv:1907.11692*, 2019. (Cited on p. 191)

[Lor56] E. Lorenz. *Empirical orthogonal functions and statistical weather prediction*, volume 1. Massachusetts Institute of Technology, Department of Meteorology, Cambridge, MA, USA, 1956. (Cited on p. 54)

[LPM15] M.-T. Luong, H. Pham, and C. Manning. Effective approaches to attention-based neural machine translation. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing, Lisbon, Portugal*, pages 1412–1421. Association for Computational Linguistics, 2015. (Cited on p. 183)

[LSST+02] H. Lodhi, C. Saunders, J. Shawe-Taylor, N. Cristianini, and C. Watkins. Text classification using string kernels. *Journal of Machine Learning Research*, 2:419–444, 2002. (Cited on p. 170)

[Lum67] J. Lumley. The structure of inhomogeneous turbulence. In A. Yaglom and V. Tatarski, editors, *Atmospheric Turbulence and Wave Propagation*, pages 166–178, Moscow, Russia, 1967. (Cited on p. 54)

[LV07] J. Lee and M. Verleysen. *Nonlinear Dimensionality Reduction*. Springer Science & Business Media, New York, NY, USA, 2007. (Cited on pp. 16, 54, 59, 68, 69, 85)

[LXS+19] J. Lee, L. Xiao, S. Schoenholz, Y. Bahri, R. Novak, J. Sohl-Dickstein, and J. Pennington. Wide neural networks of any depth evolve as linear models under gradient descent. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32, pages 8572–8583, Red Hook, NY, USA, 2019. Curran Associates, Inc. (Cited on p. 129)

[MBL+19] G. Montavon, A. Binder, S. Lapuschkin, W. Samek, and K.-R. Müller. Layer-wise relevance propagation: An overview. In W. Samek, G. Montavon, A. Vedaldi, L. Hansen, and K.-R. Müller, editors, *Explainable AI: Interpreting, Explaining and Visualizing Deep Learning*, pages 193–209, Cham, Switzerland, 2019. Springer. (Cited on p. 195)

[MD93] C. Mooney and R. Duvall. *Bootstrapping: A Nonparametric Approach to Statistical Inference*, volume 95 of *Quantitative Applications in the Social Sciences*. SAGE, Newbury Park, CA, USA, 1993. (Cited on p. 175)

[Mea15] V. Mnih et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540): 529–533, 2015. (Cited on pp. 139, 150, 160)

[Mer09] J. Mercer. XVI. Functions of positive and negative type, and their connection with the theory of integral equations. *Philosophical Transactions of the Royal Society of London. Series A*, 209(441-458):415–446, 1909. (Cited on pp. 46, 47)

[Mha96] H. Mhaskar. Neural networks for optimal approximation of smooth and analytic functions. *Neural Computation*, 8(1):164–177, 1996. (Cited on p. 91)

[MHSG18] L. McInnes, J. Healy, N. Saul, and L. Großberger. UMAP: Uniform manifold approximation and projection. *Journal of Open Source Software*, 3(29):861, 2018. (Cited on p. 85)

[MJ21] M. Muehlebach and M. Jordan. Optimization with momentum: Dynamical, control-theoretic, and symplectic perspectives. *Journal of Machine Learning Research*, 22(73):1–50, 2021. (Cited on p. 138)

[MM15] D. Mishkin and J. Matas. All you need is a good init. *arXiv preprint, arXiv:1511.06422*, 2015. (Cited on p. 93)

[MM18] H. Minh and V. Murino. *Covariances in Computer Vision and Machine Learning*. Springer, Cham, Switzerland, 2018. (Cited on p. 168)

[MN15] J. Moudrík and R. Neruda. Evolving non-linear stacking ensembles for prediction of Go player attributes. In *2015 IEEE Symposium Series on Computational Intelligence, Cape Town, South Africa*, pages 1673–1680. IEEE, 2015. (Cited on p. 172)

[Mol20] C. Molnar. *Interpretable Machine Learning*. Self-published, 2020. available at https://christophmolnar.com/books/interpretable-machine-learning. (Cited on pp. 191, 192, 195)

[MP05] C. Micchelli and M. Pontil. On learning vector-valued functions. *Neural Computation*, 17(1):177–204, 2005. (Cited on p. 168)

[Mur22] K. Murphy. *Probabilistic Machine Learning: An Introduction*. MIT Press, Cambridge, MA, USA, 2022. (Cited on pp. 16, 17, 30, 31, 49, 51, 52, 77, 126, 171)

[Mur23] K. Murphy. *Probabilistic Machine Learning: Advanced Topics*. MIT Press, Cambridge, MA, USA, 2023. (Cited on pp. 16, 117, 134, 136, 137, 149)

[MWG+21] V. Makoviychuk, L. Wawrzyniak, Y. Guo, M. Lu, K. Storey, M. Macklin, D. Hoeller, N. Rudin, A. Allshire, A. Handa, and G. State. Isaac gym: High performance GPU based physics simulation for robot learning. In J. Vanschoren and S. Yeung, editors, *Neural Information Processing Systems Track on Datasets and Benchmarks*, volume 1. Curran Associates, Inc., 2021. (Cited on p. 166)

[MZ22] T. Mao and D.-X. Zhou. Approximation of functions from Korobov spaces by deep convolutional neural networks. *Advances in Computational Mathematics*, 48(6):84, 2022. (Cited on p. 91)

[Nau11] U. Naumann. *The Art of Differentiating Computer Programs: An Introduction to Algorithmic Differentiation*. SIAM, Philadelphia, PA, USA, 2011. (Cited on p. 95)

[NBC21] A. Narayan, B. Berger, and H. Cho. Assessing single-cell transcriptomic variability through density-preserving data visualization. *Nature Biotechnology*, 39(6):765–774, 2021. (Cited on p. 86)

[Nea20] E. Ntoutsi et al. Bias in data-driven artificial intelligence systems—an introductory survey. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 10(3):e1356, 2020. (Cited on p. 192)

[Nes83] Y. Nesterov. A method for solving the convex programming problem with convergence rate $O\left(\frac{1}{k^2}\right)$. *Soviet Mathematics Doklady*, 27(2):372–376, 1983. (Cited on p. 103)

[Ng11] A. Ng. Sparse autoencoder. *CS294A Lecture notes*, 72:1–19, 2011. available at https://web.stanford.edu/class/cs294a/sparseAutoencoder.pdf, Stanford University, Stanford, CA, USA. (Cited on p. 114)

[NJ02] A. Ng and M. Jordan. On discriminative vs. generative classifiers: A comparison of logistic regression and naive Bayes. In T. Dietterich, S. Becker, and Z. Ghahramani, editors, *Advances in Neural Information Processing Systems*, volume 14, pages 841–848, Cambridge, MA, USA, 2002. MIT Press. (Cited on p. 116)

[NW08] E. Novak and H. Woźniakowski. *Tractability of Multivariate Problems: Standard Information for Functionals*. European Mathematical Society, Zurich, Switzerland, 2008. (Cited on p. 13)

[NYC16] A. Nguyen, J. Yosinski, and J. Clune. Multifaceted feature visualization: Uncovering the different types of features learned by each neuron in deep neural networks. *arXiv preprint, arXiv:1602.03616*, 2016. (Cited on p. 194)

[NZM+19] H. Nies, Z. Zakaria, M. Mohamad, W. Chan, N. Zaki, R. Sinnott, S. Napis, P. Chamoso, S. Omatu, and J. Corchado. A review of computational methods for clustering genes with similar biological functions. *Processes*, 7(9):550, 2019. (Cited on p. 7)

[OFG97] E. Osuna, R. Freund, and F. Girosi. An improved training algorithm for support vector machines. In *Neural Networks for Signal Processing VII. Proceedings of the 1997 IEEE Signal Processing Society Workshop, Amelia Island, FL, USA*, pages 276–285. IEEE, 1997. (Cited on p. 41)

[Ope23] OpenAI. GPT-4 technical report. Technical report, 2023. https://cdn.openai.com/papers/gpt-4.pdf. (Cited on p. 190)

[OSZ22] J. Opschoor, C. Schwab, and J. Zech. Exponential ReLU DNN expression of holomorphic maps in high dimension. *Constructive Approximation*, 55(1):537–582, 2022. (Cited on p. 92)

[Pea01] K. Pearson. On lines and planes of closest fit to a system of points in space. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 6(2):559–571, 1901. (Cited on pp. 54, 65)

[Pea09] J. Pearl. *Causality*. Cambridge University Press, Cambridge, UK, 2009. (Cited on p. 192)

[PFA06] X. Pennec, P. Fillard, and N. Ayache. A Riemannian framework for tensor computing. *International Journal of Computer Vision*, 66:41–66, 2006. (Cited on p. 169)

[PG12] A. Paprotny and J. Garcke. On a connection between maximum variance unfolding, shortest path problems and Isomap. In *15th International Conference on Artificial Intelligence and Statistics (AISTATS 2012), La Palma, Canary Islands, Spain*, pages 859–867, 2012. (Cited on p. 85)

[Phi21] J. Phillips. *Mathematical Foundations for Data Analysis*. Springer, Cham, Switzerland, 2021. (Cited on p. 16)

[Pla98] J. Platt. Sequential minimal optimization: A fast algorithm for training support vector machines. Technical report, Microsoft Research Redmond Labs, Redmond, WA, USA, 1998. (Cited on pp. 40, 44)

[PMG+17] N. Papernot, P. McDaniel, I. Goodfellow, S. Jha, B. Celik, and A. Swami. Practical black-box attacks against machine learning. In *Proceedings of the 2017 Asia Conference on Computer and Communications Security, Abu Dhabi, UAE*, pages 506–519, 2017. (Cited on p. 122)

[PN17] A. Polydoros and L. Nalpantidis. Survey of model-based reinforcement learning: Applications on robotics. *Journal of Intelligent & Robotic Systems*, 86(2):153–173, 2017. (Cited on p. 140)

[Pow22] W. Powell. *Reinforcement Learning and Stochastic Optimization*. Wiley, Hoboken, NJ, USA, 2022. (Cited on pp. 139, 145, 166)

[Pro05] P. Protter. *Stochastic Integration and Differential Equations*. Springer, Berlin, Heidelberg, Germany, 2005. (Cited on p. 134)

[PSF19] X. Pennec, S. Sommer, and T. Fletcher. *Riemannian Geometric Statistics in Medical Image Analysis*. Academic Press, London, UK, 2019. (Cited on p. 169)

[PT13] A. Paprotny and M. Thess. *Realtime Data Mining*. Applied and Numerical Harmonic Analysis. Springer, Cham, Switzerland, 2013. (Cited on p. 139)

[PTVF07] W. Press, S. Teukolsky, W. Vetterling, and B. Flannery. *Numerical Recipes: The Art of Scientific Computing*, 3rd edition. Cambridge University Press, Cambridge, UK, 2007. (Cited on pp. 20, 57, 132)

[PV18] P. Petersen and F. Voigtlaender. Optimal approximation of piecewise smooth functions using deep ReLU neural networks. *Neural Networks*, 108:296–330, 2018. (Cited on p. 91)

[QSC+17] Y. Qin, D. Song, H. Cheng, W. Cheng, G. Jiang, and G. Cottrell. A dual-stage attention-based recurrent neural network for time series prediction. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence, Melbourne, Australia*, pages 2627–2633. AAAI Press, 2017. (Cited on p. 179)

[RBAF+19] M. Rhif, A. Ben Abbes, I. Farah, B. Martínez, and Y. Sang. Wavelet transform application for/in non-stationary time-series analysis: A review. *Applied Sciences*, 9(7):1345, 2019. (Cited on p. 171)

[RBDG20] R. Roscher, B. Bohn, M. Duarte, and J. Garcke. Explainable machine learning for scientific insights and discoveries. *IEEE Access*, 8(1):42200–42216, 2020. (Cited on pp. 191, 192)

[RBU+20] L. Rundo, L. Beer, S. Ursprung, P. Martin-Gonzalez, F. Markowetz, J. Brenton, M. Crispin-Ortuzar, E. Sala, and R. Woitek. Tissue-specific and interpretable sub-segmentation of whole tumour burden on CT images by unsupervised fuzzy clustering. *Computers in Biology and Medicine*, 120:103751, 2020. (Cited on p. 7)

[RBZ23] D. Ruiz-Balet and E. Zuazua. Neural ODE control for classification, approximation, and transport. *SIAM Review*, 65(3):735–773, 2023. (Cited on p. 133)

[RDGF16] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Las Vegas, NV, USA*, pages 779–788, 2016. (Cited on p. 62)

[Rec19] B. Recht. A tour of reinforcement learning: The view from continuous control. *Annual Review of Control, Robotics, and Autonomous Systems*, 2(1):253–279, 2019. (Cited on p. 139)

[RH20] L. Ruthotto and E. Haber. Deep neural networks motivated by partial differential equations. *Journal of Mathematical Imaging and Vision*, 62(3):352–364, 2020. (Cited on p. 132)

[RHGS15] S. Ren, K. He, R. Girshick, and J. Sun. Faster R-CNN: Towards real-time object detection with region proposal networks. In C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 28, pages 91–99, Red Hook, NY, USA, 2015. Curran Associates, Inc. (Cited on p. 130)

[RHW86] D. Rumelhart, G. Hinton, and R. Williams. Learning representations by back-propagating errors. *Nature*, 323:533–536, 1986. (Cited on pp. 91, 103)

[RM10] S. Rohanizadeh and M. Moghadam. A proposed data mining methodology and its application to industrial procedures. *Journal of Optimization in Industrial Engineering*, 2(4):37–50, 2010. (Cited on p. 15)

[RM21] K. Rusch and S. Mishra. Coupled oscillatory recurrent neural network (coRNN): An accurate and (gradient) stable architecture for learning long time dependencies. In *Proceedings of the 9th international Conference on Learning Representations, online*, 2021. (Cited on p. 183)

[RMEM21] K. Rusch, S. Mishra, B. Erichson, and M. Mahoney. Long expressive memory for sequence modeling. *arXiv preprint, arXiv:2110.04744*, 2021. (Cited on p. 183)

[Ros58] F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Reviews*, 65:386–408, 1958. (Cited on p. 88)

[RPK19] M. Raissi, P. Perdikaris, and G. Karniadakis. Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational Physics*, 378:686–707, 2019. (Cited on pp. 137, 192)

[RS05] J. Ramsay and B. Silverman. *Functional Data Analysis*. Springer, New York, NY, USA, 2005. (Cited on p. 170)

[RS07] J. Ramsay and B. Silverman. *Applied Functional Data Analysis: Methods and Case Studies*. Springer, New York, NY, USA, 2007. (Cited on p. 170)

[RSG16] M. Ribeiro, S. Singh, and C. Guestrin. "Why should I trust you?" Explaining the predictions of any classifier. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA*, pages 1135–1144. Association for Computing Machinery, 2016. (Cited on p. 194)

[RW06] C. Rasmussen and C. Williams. *Gaussian Processes for Machine Learning*. MIT Press, Cambridge, MA, USA, 2006. (Cited on pp. 46, 47, 52)

[RW17] W. Rawat and Z. Wang. Deep convolutional neural networks for image classification: A comprehensive review. *Neural Computation*, 29(9):2352–2449, 2017. (Cited on pp. 96, 108)

[RYH22] D. Roberts, S. Yaida, and B. Hanin. *The Principles of Deep Learning Theory*. Cambridge University Press, Cambridge, UK, 2022. (Cited on p. 126)

[Saa03] Y. Saad. *Iterative Methods for Sparse Linear Systems*, 2nd edition. SIAM, Philadelphia, PA, USA, 2003. (Cited on p. 28)

[Sag12] H. Sagan. *Space-Filling Curves*. Springer, New York, NY, USA, 2012. (Cited on p. 114)

[Sam65] P. Samuelson. Rational theory of warrant pricing. *Industrial Management Review*, 6(2):13–32, 1965. (Cited on p. 134)

[SB18] R. Sutton and A. Barto. *Reinforcement Learning: An Introduction*, 2nd edition. MIT Press, Cambridge, MA, USA, 2018. (Cited on pp. 139, 149, 150, 152, 153, 154, 155, 158, 162, 164, 165, 166)

[SBC16] W. Su, S. Boyd, and E. Candes. A differential equation for modeling Nesterov's accelerated gradient method: Theory and insights. *Journal of Machine Learning Research*, 17(153):1–43, 2016. (Cited on p. 138)

[Sch15] J. Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61:85–117, 2015. (Cited on p. 109)

[Sch22] B. Schölkopf. Causality for machine learning. In H. Geffner, R. Dechter, and J. Halpern, editors, *Probabilistic and Causal Inference: The Works of Judea Pearl*, pages 765–804. Association for Computing Machinery, New York, NY, USA, 2022. (Cited on p. 192)

[SDB16] L. Seversky, S. Davis, and M. Berger. On time-series topological data analysis: New data and opportunities. In *Proceedings of the 2016 IEEE Conference on Computer Vision and Pattern Recognition Workshops, Las Vegas, NV, USA*, pages 59–67, 2016. (Cited on p. 171)

[SDWMG15] J. Sohl-Dickstein, E. Weiss, N. Maheswaranathan, and S. Ganguli. Deep unsupervised learning using nonequilibrium thermodynamics. In F. Bach and D. Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning, Lille, France*, volume 37 of *Proceedings of Machine Learning Research*, pages 2256–2265. PMLR, 2015. (Cited on pp. 134, 136)

[Sea18] D. Silver et al. A general reinforcement learning algorithm that masters chess, Shogi, and Go through self-play. *Science*, 362(6419):1140–1144, 2018. (Cited on pp. 139, 160, 166)

[See04] M. Seeger. Gaussian processes for machine learning. *International Journal of Neural Systems*, 14(02):69–106, 2004. (Cited on p. 52)

[Set12] B. Settles. Active learning. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 6(1):1–114, 2012. (Cited on p. 6)

[SFSW12] D. Schnitzer, A. Flexer, M. Schedl, and G. Widmer. Local and global scaling reduce hubs in space. *Journal of Machine Learning Research*, 13(1):2871–2902, 2012. (Cited on p. 13)

[SHB16] R. Sennrich, B. Haddow, and A. Birch. Neural machine translation of rare words with subword units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), Berlin, Germany*, pages 1715–1725. Association for Computational Linguistics, 2016. (Cited on p. 185)

[She00] C. Shearer. The CRISP-DM model: The new blueprint for data mining. *Journal of Data Warehousing*, 5:13–22, 2000. (Cited on p. 14)

[She20] A. Sherstinsky. Fundamentals of recurrent neural network (RNN) and long short-term memory (LSTM) network. *Physica D: Nonlinear Phenomena*, 404:132306, 2020. (Cited on pp. 182, 183)

[SK87] L. Sirovich and M. Kirby. Low-dimensional procedure for the characterization of human faces. *Journal of the Optical Society of America A*, 4(3):519–524, 1987. (Cited on p. 62)

[SK12] M. Sugiyama and M. Kawanabe. *Machine Learning in Non-Stationary Environments: Introduction to Covariate Shift Adaptation*. MIT Press, Cambridge, MA, USA, 2012. (Cited on p. 22)

[SK18] M. Simonovsky and N. Komodakis. GraphVAE: Towards generation of small graphs using variational autoencoders. In V. Kůrková, Y. Manolopoulos, B. Hammer, L. Iliadis, and I. Maglogiannis, editors, *Artificial Neural Networks and Machine Learning – ICANN 2018*, pages 412–422, Cham, Switzerland, 2018. Springer. (Cited on p. 170)

[SLJ+15] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In *Proceedings of the 2015 IEEE Conference on Computer Vision and Pattern Recognition, Boston, MA, USA*, pages 1–9. IEEE, 2015. (Cited on pp. 193, 195)

[SMSM99] R. Sutton, D. McAllester, S. Singh, and Y. Mansour. Policy gradient methods for reinforcement learning with function approximation. In S. Solla, T. Leen, and K.-R. Müller, editors, *Advances in Neural Information Processing Systems*, volume 12, pages 1057–1063, Cambridge, MA, USA, 1999. MIT Press. (Cited on p. 166)

[SR04] M. Santos and J. Rust. Convergence properties of policy iteration. *SIAM Journal on Control and Optimization*, 42(6):2094–2115, 2004. (Cited on p. 149)

[SR18] O. Sagi and L. Rokach. Ensemble learning: A survey. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 8(4):e1249, 2018. (Cited on p. 171)

[SS96] S. Singh and R. Sutton. Reinforcement learning with replacing eligibility traces. *Machine Learning*, 22(1-3):123–158, 1996. (Cited on p. 152)

[SS02] B. Schölkopf and A. Smola. *Learning with Kernels – Support Vector Machines, Regularization, Optimization, and Beyond*. MIT Press, Cambridge, MA, USA, 2002. (Cited on pp. 38, 44, 46, 47, 48, 49, 51, 52, 126)

[SS16] S. Saitoh and Y. Sawano. *Theory of Reproducing Kernels and Applications*. Springer, Singapore, 2016. (Cited on p. 47)

[Sto48] M. Stone. The generalized Weierstrass approximation theorem. *Mathematics Magazine*, 21(5):237–254, 1948. (Cited on p. 91)

[Str19] G. Strang. *Linear Algebra and Learning From Data*. Wellesley-Cambridge Press, Wellesley, MA, USA, 2019. (Cited on p. 16)

[Sut88] R. Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, 3(1):9–44, 1988. (Cited on pp. 153, 154)

[SW06] R. Schaback and H. Wendland. Kernel techniques: From machine learning to meshless methods. *Acta Numerica*, 15:543–639, 2006. (Cited on p. 48)

[SWD+17] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms. *arXiv preprint, arXiv:1707.06347*, 2017. (Cited on p. 166)

[SX22] J. Siegel and J. Xu. High-order approximation rates for neural networks with $\mathrm{ReLU}^k$ activation functions. *Applied and Computational Harmonic Analysis*, 58:1–26, 2022. (Cited on p. 89)

[Sze10] C. Szepesvári. Algorithms for reinforcement learning. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 4(1):1–103, 2010. (Cited on p. 166)

[Tak81] F. Takens. Detecting strange attractors in turbulence. In D. Rand and L.-S. Young, editors, *Dynamical Systems and Turbulence, Warwick 1980*, pages 366–381, Berlin, Heidelberg, Germany, 1981. Springer. (Cited on p. 171)

[TdL00] J. Tenenbaum, V. de Silva, and J. Langford. A global geometric framework for nonlinear dimensionality reduction. *Science*, 290(5500):2319–2323, 2000. (Cited on pp. 70, 71)

[Tea23] M. Towers et al. Gymnasium. https://zenodo.org/record/8127025, 2023. (Cited on p. 146)

[Tes95] G. Tesauro. Temporal difference learning and TD-Gammon. *Communications of the ACM*, 38(3):58–68, 1995. (Cited on p. 139)

[Tik63] A. Tikhonov. Solution of incorrectly formulated problems and the regularization method. *Soviet Math. Dokl.*, 4:1035–1038, 1963. (Cited on p. 11)

[TK21] N. Takeishi and A. Kalousis. Physics-integrated variational autoencoders for robust and interpretable generative modeling. In M. Ranzato, A. Beygelzimer, Y. Dauphin, P. Liang, and J. Wortman Vaughan, editors, *Advances in Neural Information Processing Systems*, volume 34, pages 14809–14821, Red Hook, NY, USA, 2021. Curran Associates, Inc. (Cited on p. 123)

[TMB+16] D. Tomè, F. Monti, L. Baroffio, L. Bondi, M. Tagliasacchi, and S. Tubaro. Deep convolutional neural networks for pedestrian detection. *Signal Processing: Image Communication*, 47:482–489, 2016. (Cited on p. 62)

[TZ15] N. Tishby and N. Zaslavsky. Deep learning and the information bottleneck principle. In *Proceedings of the 2015 IEEE Information Theory Workshop, Jerusalem, Israel*, pages 1–5. IEEE, 2015. (Cited on p. 108)

[Uns19] M. Unser. A representer theorem for deep neural networks. *Journal of Machine Learning Research*, 20(110):1–30, 2019. (Cited on p. 92)

[VA06] S. Vassilvitskii and D. Arthur. k-means++: The advantages of careful seeding. In *Proceedings of the 18th Annual ACM-SIAM Symposium on Discrete Algorithms, New Orleans, LA, USA*, pages 1027–1035. SIAM, 2006. (Cited on p. 77)

[Vap99] V. Vapnik. *The Nature of Statistical Learning Theory*, 2nd edition. Springer, New York, NY, USA, 1999. (Cited on pp. 16, 47)

[VC19] L. Vanneschi and M. Castelli. Multilayer perceptrons. In S. Ranganathan, M. Gribskov, K. Nakai, and C. Schönbach, editors, *Encyclopedia of Bioinformatics and Computational Biology*, pages 612–620. Elsevier, Amsterdam, The Netherlands, 2019. (Cited on p. 87)

[VdMH08] L. Van der Maaten and G. Hinton. Visualizing data using t-SNE. *Journal of Machine Learning Research*, 9(11):2579–2605, 2008. (Cited on pp. 84, 85)

[vH10] H. van Hasselt. Double q-learning. In J. Lafferty, C. Williams, J. Shawe-Taylor, R. Zemel, and A. Culotta, editors, *Advances in Neural Information Processing Systems*, volume 23, pages 2613–2621, Red Hook, NY, USA, 2010. Curran Associates, Inc. (Cited on p. 164)

[Vin00] R. Vinter. *Optimal Control*. Birkhäuser, Boston, MA, USA, 2000. (Cited on p. 139)

[vL07] U. von Luxburg. A tutorial on spectral clustering. *Stat. Comput.*, 17(4):395–416, 2007. (Cited on p. 79)

[vRMB+23] L. von Rueden, S. Mayer, K. Beckh, B. Georgiev, S. Giesselbach, R. Heese, B. Kirsch, J. Pfrommer, A. Pick, R. Ramamurthy, M. Walczak, J. Garcke, C. Bauckhage, and J. Schuecker. Informed machine learning—A taxonomy and survey of integrating knowledge into learning systems. *IEEE Transactions on Knowledge and Data Engineering*, 35(1):614–633, 2023. (Cited on pp. 52, 137)

[vSMP+16] H. van Seijen, A. Mahmood, P. Pilarski, M. Machado, and R. Sutton. True online temporal-difference learning. *Journal of Machine Learning Research*, 17(145):1–40, 2016. (Cited on p. 165)

[VSP+17] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is all you need. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30, pages 6000–6010, Red Hook, NY, USA, 2017. Curran Associates, Inc. (Cited on pp. 183, 185, 186, 187)

[Wah90] G. Wahba. *Spline Models for Observational Data*. SIAM, Philadelphia, PA, USA, 1990. (Cited on p. 48)

[Wat89] C. Watkins. *Learning from Delayed Rewards*. PhD thesis, King's College, Cambridge, UK, 1989. (Cited on p. 158)

[Wen95] H. Wendland. Piecewise polynomial, positive definite and compactly supported radial functions of minimal degree. *Advances in Computational Mathematics*, 4(1):389–396, 1995. (Cited on p. 48)

[Wer82] P. Werbos. Applications of advances in nonlinear sensitivity analysis. In R. Drenick and F. Kozin, editors, *System Modeling and Optimization*, pages 762–770. Springer, Berlin, Heidelberg, Germany, 1982. (Cited on p. 91)

[WH96] G. Wanner and E. Hairer. *Solving Ordinary Differential Equations II: Stiff and Differential-Algebraic Problems*. Springer, Berlin, Heidelberg, Germany, 1996. (Cited on p. 127)

[WNH93] G. Wanner, S. Nørsett, and E. Hairer. *Solving Ordinary Differential Equations I: Nonstiff Problems*. Springer, Berlin, Heidelberg, Germany, 1993. (Cited on pp. 127, 131)

[Wol92] D. Wolpert. Stacked generalization. *Neural Networks*, 5(2):241–259, 1992. (Cited on p. 172)

[WR22] S. Wright and B. Recht. *Optimization for Data Analysis*. Cambridge University Press, Cambridge, UK, 2022. (Cited on p. 16)

[WWJ16] A. Wibisono, A. Wilson, and M. Jordan. A variational perspective on accelerated methods in optimization. *Proceedings of the National Academy of Sciences*, 113(47):E7351–E7358, 2016. (Cited on p. 138)

[WWS09] C. Wojek, S. Walk, and B. Schiele. Multi-cue onboard pedestrian detection. In *2009 IEEE Conference on Computer Vision and Pattern Recognition, Miami, FL, USA*, pages 794–801. IEEE, 2009. (Cited on pp. 62, 67)

[XTL09] H. Xie, H. Tang, and Y.-H. Liao. Time series prediction based on NARX neural networks: An advanced approach. In *International Conference on Machine Learning and Cybernetics, Baoding, Hebei, China*, volume 3, pages 1275–1279. IEEE, 2009. (Cited on p. 171)

[XW08] R. Xu and D. Wunsch. *Clustering*, volume 10 of *IEEE Press Series on Computational Intelligence*. John Wiley & Sons, Hoboken, NJ, USA, 2008. (Cited on pp. 7, 77, 86)

[Yar17] D. Yarotsky. Error bounds for approximations with deep ReLU networks. *Neural Networks*, 94:103–114, 2017. (Cited on p. 91)

[YH38] G. Young and A. Householder. Discussion of a set of points in terms of their mutual distances. *Psychometrika*, 3(1):19–22, 1938. (Cited on p. 59)

[YHW+07] J. Yang, D. Hubball, M. Ward, E. Rundensteiner, and W. Ribarsky. Value and relation display: Interactive visual exploration of large data sets with hundreds of dimensions. *IEEE Transactions on Visualization and Computer Graphics*, 13(3):494–507, 2007. (Cited on p. 7)

[YWV+22] J. Yu, Z. Wang, V. Vasudevan, L. Yeung, M. Seyedhosseini, and Y. Wu. CoCa: Contrastive captioners are image-text foundation models. *Transactions on Machine Learning Research*, 2022. https://openreview.net/forum?id=Ee277P3AYC. (Cited on p. 108)

[YYR+18] J. You, R. Ying, X. Ren, W. Hamilton, and J. Leskovec. GraphRNN: Generating realistic graphs with deep auto-regressive models. In *Proceedings of the 35th International Conference on Machine Learning, Stockholm, Sweden*, volume 80 of *Proceedings of Machine Learning Research*, pages 5708–5717. PMLR, 2018. (Cited on p. 170)

[ZA15] N. Zaitoun and M. Aqel. Survey on image segmentation techniques. *Procedia Computer Science*, 65:797–806, 2015. (Cited on p. 62)

[ZHS+17] M. Zeestraten, I. Havoutis, J. Silvério, S. Calinon, and D. Caldwell. An approach for imitation learning on Riemannian manifolds. *IEEE Robotics and Automation Letters*, 2(3):1240–1247, 2017. (Cited on p. 169)

[ZQW20] W. Zhao, J. Queralta, and T. Westerlund. Sim-to-real transfer in deep reinforcement learning for robotics: A survey. In *2020 IEEE Symposium Series on Computational Intelligence (SSCI)*, pages 737–744, 2020. (Cited on p. 166)

[ZTS+21] S. Zhai, W. Talbott, N. Srivastava, C. Huang, H. Goh, R. Zhang, and J. Susskind. An attention free transformer. *arXiv preprint, arXiv:2105.14103*, 2021. (Cited on p. 190)

# Index

Data Science Book Series

This unique book explores several well-known machine learning and data analysis algorithms from a mathematical and programming perspective. The authors

- present machine learning methods, review the underlying mathematics, and provide programming exercises to deepen the reader's understanding;
- accompany application areas with exercises that explore the unique characteristics of real-world data sets (e.g., image data for pedestrian detection, biological cell data); and
- provide new terminology and background information on mathematical concepts, as well as exercises, in "info-boxes" throughout the text.

*Algorithmic Mathematics in Machine Learning* is intended for mathematicians, computer scientists, and practitioners who have a basic mathematical background in analysis and linear algebra, but little or no knowledge of machine learning and related algorithms. Researchers in the natural sciences and engineers interested in acquiring the mathematics needed to apply the most popular machine learning algorithms will also find this book useful.

This book is appropriate for a practical lab or basic lecture course on machine learning within a mathematics curriculum.

**Bastian Bohn** is an *Akademischer Rat* at the Institute for Numerical Simulation, University of Bonn, Germany, where he was previously a postdoctoral researcher. His research interests include machine learning, the mathematics of data science, numerical algorithms in high dimensions, and approximation theory.

**Jochen Garcke** is a professor of numerics at the Institute for Numerical Simulation, University of Bonn, Germany, and department head at Fraunhofer SCAI (Institute for Algorithms and Scientific Computing), Sankt Augustin, Germany. His research interests include machine learning, scientific computing, reinforcement learning, and high-dimensional approximation.

**Michael Griebel** is a professor at the Institute for Numerical Simulation, University of Bonn, Germany, where he holds the Chair of Scientific Computing and Numerical Simulation. He is also director of Fraunhofer SCAI (Institute for Algorithms and Scientific Computing), Sankt Augustin, Germany. His research interests include numerical simulation, scientific computing, machine learning, and high-dimensional approximation.

For more information about SIAM books, journals, conferences, memberships, or activities, contact:

siam

Society for Industrial and Applied Mathematics
3600 Market Street, 6th Floor
Philadelphia, PA 19104-2688 USA
+1-215-382-9800
siam@siam.org • *siam.org*

ISBN: 978-1-61197-787-5

DI03

SIAM Textbooks