National University of Singapore, DSA 5102

Foundations of Machine Learning

Qianxiao Li

2020 Semester 1

Preface

These lecture notes are a work-in-progress for *DSA5102: Foundations of Machine Learning* that I am teaching at National University of Singapore in Semester 1 of AY 2020-2021. This document will be progressively updated throughout the semester.

The notes are meant to be a gentle introduction to the theory and algorithms of machine learning. We emphasize on simplicity and breadth, and so depth is necessarily compromised for certain topics. Interested students may consult the many textbooks and papers on the subject referenced throughout the notes for further study.

This document is typeset using LaTeXwith a modified theme based on

https://www.overleaf.com/latex/templates/lecture-note-template/dwyrjrnthdcz

If you see any mistakes or typos in the notes, please send me an email at qianxiao@nus.edu.sg so that I can incorporate the relevant corrections.

Contents

1 Introduction

1.1 Overview

In 1950, English mathematician Alan Turing proposed to consider the following question [Tur50]

> *Can machines think?*

In the paper's subsequent discussions it becomes apparent that the question Turing is really asking is

> *Can machines do what thinking beings do?*

He goes on the introduce the idea of the *imitation game*, in which the goal of the machine is to fool a human being into thinking that it is in fact a human – giving rise to the so-called *Turing test* for the artificial intelligence.

Machine learning is the study of methodologies that *may* lead to an answer of the second question. In machine learning we pose a much more conservative version of question 2: *How can machines* ***learn*** *to do* ***some things*** *that thinking beings do?* The learning process can be formalized as [Mit97]: " *A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P if its performance at tasks in T, as measured by P, improves with experience E.* " In this sense, machine learning is the study of algorithmic approaches to learning, and is in general a subset of the broad field of artificial intelligence.

These notes serve as an introduction to the theory and algorithms in modern machine learning. The various material are organized into three components, *supervised learning*, *unsupervised learning* and *reinforcement learning*. These represent major topics in modern machine learning, although in reality their intersections are non-empty and their union is not exhaustive. Nevertheless, for the purpose of introduction, they do represent a sufficient variety of problems to base discussions on.

The viewpoint adopted in these notes is not limited to a statistical one. Instead, an attempt is made to also discuss some related lines of work that is seeing growing importance in the study of machine learning today. These include approximation theory in high dimensions, dynamical systems, stochastic processes and so on. To keep the scope sufficiently broad and the material sufficiently accessible, some very interesting machine learning topics have to be omitted. For example, the statistical treatment of various topics are quite limited – little is covered in terms of Bayesian statistics, Gaussian processes and Bayesian optimization, to name a few. Moreover, rigorous proofs of theoretical results are kept at a minimum. However, appropriate references will be given and the interested readers can dive deeper into the relevant results. One can also

consult more comprehensive textbooks on the subject, e.g. [BO06, FHT01, MRT18]. Lastly, what is also missing is extensive case studies on modern datasets and problems – this is intentional as the goal of this course is to develop foundations of machine learning methodologies, on top of which the readers can explore further into the exciting world of machine learning and data science.

1.2 Data

Contrary to traditional programming, the output of machine learning is typically a program – or called a *model* with which inference about a new data point can be made. This is different from traditional programming, in the sense that we wish to infer a "program" from its inputs and outputs. Therefore, data is one of the most important aspects of machine learning (this is the experience E in Mitchell's definition of machine learning). In fact, the growing rate of data collection is one of key enabling factors of modern machine learning applications. Data comes in many forms, such as images, text, and time-series etc., see Figure 1.1. Invariably, for machine learning they will have to be represented on a computer and thus we will need some mapping scheme into floating numbers (Euclidean space). This is natural for images (using RGB values) and many other numerical data, but for more discrete ones such as text data, this is less obvious. In fact, some work must be done to map words into vectors in Euclidean spaces [MCCD13].

(a) Image (b) Text (c) Time-series

Figure 1.1: Different types of data. (a) Images of handwritten digits from the MNIST dataset [LCB10]. (b) Word cloud generated from this document. (c) Time-series using FMRI dataset from `http://seaborn.pydata.org/`.

An important consideration when embedding discrete data into Euclidean spaces is the following distinction between data types: *ordinal* vs *nominal* data. Ordinal data, as the root word "order" suggests, describe data with a natural sequence. For example, the ratings of a E-commerce product may be discrete (1 to 5 stars), but they are have a natural order. On the other hand, the category of images, e.g. "cats, dogs, rats", have no natural order to them for most purposes. Such discrete data are called nominal, or *categorical* data. One important remark is that order is not an intrinsic property of data, but relies on the application use-case. For example, the

numbers $1-5$ obviously have an order to them, but when they are interpreted as the labels of images of hand-written digits, it is more appropriate to treat them as distinct nominal objects.

To embed ordinal data, we usually do so by mapping the data into a subset of the real numbers that preserve such order. For example, 1-5 as number of stars in product ratings can be embedded as

$$\{\bigstar, \bigstar\bigstar, \bigstar\bigstar\bigstar, \bigstar\bigstar\bigstar\bigstar, \bigstar\bigstar\bigstar\bigstar\bigstar\} \to \{1, 2, 3, 4, 5\}\,, \tag{1.1}$$

whereas the same numbers when represented as labels of hand-written digits can be represented as the *one-hot encoding*

$$\{\,1, 2, 3, 4, 5\,\} \to \left\{ \begin{pmatrix}1\\0\\0\\0\\0\end{pmatrix}, \begin{pmatrix}0\\1\\0\\0\\0\end{pmatrix}, \begin{pmatrix}0\\0\\1\\0\\0\end{pmatrix}, \begin{pmatrix}0\\0\\0\\1\\0\end{pmatrix}, \begin{pmatrix}0\\0\\0\\0\\1\end{pmatrix} \right\}. \tag{1.2}$$

This encoding has the advantage that the "nominality" is preserved in some sense. For example, when treated as categorical variables, the vector representation of 1 and 2 are no closer or further than that of 1 and 5, as should be the case. This is not so for the naive embedding into the numbers 1-5. Throughout the examples in these notes, we will encounter the usage of such embeddings.

1.3 Classes of Machine Learning Problems

There are many types of machine learning problems, and in these notes we categorize them broadly into three categories: *supervised learning*, *unsupervised learning* and *reinforcement learning*. In supervised learning, we are given dataset comprising inputs and outputs (or labels) of a particular predictive process, from which our goal is to infer the predictive relationship and to make good predictions on new data. In unsupervised learning, we are only given inputs but not outputs, so our goals are typically different in that we are trying to infer something about the distribution of the underlying input data. Finally, reinforcement learning involves training an agent to navigate an environment through performing some actions. Here, a direct output (e.g. the optimal action) is not given. Rather, we are given reward/penalty signals for each action we take, and from these signals our goal is to optimize our action or plan. In Table 1.1, we list some representative topics in each of these domains. Note that these categories are not disjoint and their intersections represent important areas of modern machine learning research. For example, semi-supervised learning (supervised$\cap$unsupervised) has important applications in datasets that are expensive to label; Large-scale reinforcement learning draws heavily on value function approximations using supervised learning (reinforcement$\cap$supervised). There are many more such examples. Our main rationale for such a classification is to organize groups of theoretical and practical questions that apply generally to some class of problems. For example, approximation theory generally apply to supervised learning problems.

Supervised Learning	Unsupervised Learning	Reinforcement Learning
Regression	Clustering	Value iteration methods
Classification	Dimensional reduction	Policy gradient methods
Function approximation	Generative models	Actor-critique methods
Inverse problems/design	Anomaly detection	Exploration
...	...	...

Table 1.1: A highly non-exhaustive list of topics of study in each class of machine learning problems.

1.4 Evaluating Machine Learning Model Performance

Recall in the definition of machine learning, there is a performance metric P that measures our learning process. This is of fundamental importance in guiding the optimization process of machine learning models. How is P computed? To do so, we typically define some loss function (resp. performance metric) that we can compute with data, given the model at hand. Our goal is train our model so as to minimize the loss (resp. maximize the performance metric). However, we must be careful what data we use to achieve this.

Our ultimate goal is to build machine learning models that perform well on *unseen* data. Therefore, our loss functions should be computed on unseen data. Since by definition this is not available, given a dataset $\mathcal{D}$ we usually split it into a *training set* $\mathcal{D}_{\text{train}}$ and a *testing set* $\mathcal{D}_{\text{test}}$. Our training algorithms shall only use information from $\mathcal{D}_{\text{train}}$ where as the model performance will be evaluated on the unseen $\mathcal{D}_{\text{test}}$, which the model would not have seen. Often, but not always, we will split the dataset randomly. Exceptions may arise depending on the application. For example, if we would like to predict the future price of a stock given some past information (say the previous months), we should probably use data in the later years as test sets so that we are less likely to learn confounding factors such as specific temporal trends. When our dataset labels are highly imbalanced, we should also consider other splitting schemes.

Sometimes, we need a further splitting $\mathcal{D} = \mathcal{D}_{\text{train}} \cup \mathcal{D}_{\text{valid}} \cup \mathcal{D}_{\text{test}}$. The set $\mathcal{D}_{\text{valid}}$ is called the *validation set*. As before, only $\mathcal{D}_{\text{train}}$ participates in the optimization of models. On the other hand, $\mathcal{D}_{\text{valid}}$ can be used to tune hyper-parameters (e.g. depth of decision trees, number of layers in a neural network) and in general perform model selection. Note that this can be improved using cross-validation, which we will briefly touch upon later. One can understand validation set as a proxy of the testing set, which we should always pretend is unavailable to us at any time other than the final evaluation of our model's performance.

Finally, we note that not all model evaluations need to be based on test data collected *a priori*. For example, when applying machine learning techniques to solve differential equations, the solution quality can be directly checked without resorting to a hold-out set by appealing to the equations that are available to us.

1.5 Notation

Finally, we outline some notation conventions throughout the notes. We will follow usual notations used in calculus and probability, e.g. ∇ for the gradient operator, $\mathbb{E}$ for expectation and so on. It is assumed that the reader is familiar with such basic notation. Specific to these notes, we will use lower-case symbols to represent scalars and vectors (i.e. vectors are not bold-faced). Matrices will be represented by capital letters and sets are generally given script capital letters (e.g. $\mathcal{D}, \mathcal{H}$). The usual Euclidean norm is denoted as $\|\cdot\|$ and $|\cdot|$ is the absolute value. For consistency, we will use N to denote the number of samples, and M to denote complexity measures of hypothesis spaces (e.g. in linear basis models, M is the number of basis functions). The identity matrix of d dimensions will be denoted by I_d, and we will drop the subscript when the context is clear and there is no need to explicitly mention the dimension. For any vector v, we write $v \geq 0$ to indicate that $v_j \geq 0$ for every coordinate j. Vectors are usually interpreted as column vectors, with (x, y, z) denoting a column vector with entries x, y and z. The composition of two functions is denoted by $\circ$, e.g. $(f \circ g)(x) = f[g(x)]$. Other specific notation will be defined in context.

2 Supervised Learning

2.1 Overview

Supervised learning is perhaps the most basic class of machine learning problems. Here, we are given a dataset $\mathcal{D} = \{x_i, y_i\}_{i=1}^N$ consisting of *inputs* x_i with their corresponding *labels* y_i and N is the size of the data. The underlying assumption is that each y_i is determined by x_i through some mapping f^*, i.e. $y_i = f^*(x_i)$. The function f^* is sometimes called the *oracle*, carrying the meaning that it can determine perfectly the label of any sample presented to it. More generally, one can take into account of noise and uncertainties by assuming that given x_i, y_i is a sample from some "oracle" conditional distribution $y_i \sim p^*(\cdot|x_i)$. The most common model for this case is when $y_i = f^*(x_i) + \epsilon_i$ with ϵ_i representing some random noise term. For the sake of simplicity, for now we shall discuss supervised learning in the deterministic context. When the labels y_i take values in a continuum, say in $\mathbb{R}$, we say that this is a *regression* problem. Otherwise, if y_i take discrete values, we say that this is a *classification* problem. For example, in a digits recognition problem using the MNIST dataset [Figure 1.1(a)], each x_i correspond to a 28×28 gray-scale image of a number from 0 to 9 and the label y_i is the identity of this number. This is a classification problem and the oracle f^* is a perfect classifier for these types digits that maps each image to its corresponding number.

Unfortunately, the oracle f^* is unknown to us except from the information contained in the dataset $\mathcal{D} = \{x_i, y_i = f^*(x_i)\}_{i=1}^N$. As such, the over-arching goal of supervised learning is to construct, using $\mathcal{D}$, a good approximation of the oracle. Going back to the digit recognition example, the supervised learning task is to come up with a predictive model that tells us the number an image represents, using only the information given in the set of image-number pairs. The word *supervised* means that in our dataset $\mathcal{D}$, the correct label is provided to us as a form of supervision by f^* in our learning process.

So, how can we go about constructing such a predictive model? Notice that without explicit knowledge of f^* and from mere observations of $\mathcal{D}$, it is not clear how we can even represent f^*, say on a computer. Consequently, this motivates the following approach: we take a collection of functions that we *can* represent on a computer or even a piece of paper; From this collection we *pick* one f that "best approximates" f^* in some sense – f is then taken as our learned predictive model. This collection of functions, which is our job to decide, is called the *hypothesis space* and we will denote it by $\mathcal{H}$. In the following lectures, we will see many examples of different hypothesis spaces giving rise to different machine learning models with many interesting properties.

What is missing from the above discussion is how we pick a particular function f to approximate

f^*. Clearly, it relies on a precise definition of "best approximation". This is where the concept of *loss functions* comes in. In abstract terms, we want to have a notion of how close any classifier f is to the oracle f^*. Let us again consider the digit recognition problem. Here f^* is a perfect classifier of digits, and so a reasonable measure of the closeness of another classifier f to the oracle is

$$R(f) = \frac{1}{N}\sum_{i=1}^{N} \mathbb{1}_{f(x_i)\neq f^*(x_i)}, \tag{2.1}$$

where $\mathbb{1}_c$ is the *indicator function* which equals 1 if condition c is true and 0 otherwise. Thus, the right-hand-side is the proportion of digit images from $\mathcal{D}$ that are correctly matched to their labels by f, i.e. the *accuracy* of f on the training dataset $\mathcal{D}$. In statistical language, $R(f)$ is also called the *risk* associated with the predictor f.

More generally, the closeness of f to f^* can be defined by a suitable *loss function* L, so that

$$R(f) = \frac{1}{N}\sum_{i=1}^{N} L(f(x_i), f^*(x_i)) = \frac{1}{N}\sum_{i=1}^{N} L(f(x_i), y_i) \tag{2.2}$$

and we want $L(y', y)$ to decrease as y and y' becomes closer. For the example given in (2.1), we have $L(y', y) = \mathbb{1}_{y\neq y'}$ and this is known as the *zero-one loss*. For various reasons which will become clear, the zero-one loss is not often used, and we will consider a variety of other loss functions in both classification and regression problems.

Once a suitable loss function (and hence a notion of distance) is defined, the supervised learning problem can now be formalized as an optimization problem

$$\min_{f\in\mathcal{H}} R(f), \tag{2.3}$$

and if we can solve this problem (we will discuss various methods to do so in later chapters), a (approximate) minimizer $\widehat{f} \in \mathcal{H}$ of (2.3) is then our obtained predictive model. The process of finding $\widehat{f}$ by minimizing $R(f)$ (which depends on the data $\mathcal{D}$) is called *training*, and $\widehat{f}$ is called a trained model. Hopefully, $\widehat{f}$ performs our task of predicting label from inputs adequately, in which case we have succeeded at this supervised learning task.

Empirical Risk Minimization vs Population Risk Minimization

The preceding discussion, in particular Eq. (2.3) appears to suggest that machine learning is in some sense equivalent to an optimization problem. However, this is not so. In fact, Eq. (2.3) is *not* the actual problem we want to solve. To see this, simply observe that we can easily come up with a $\widehat{f}$ that minimizes the zero-one loss in the digit recognition problem – we simply memorize the label associated with each image x_i, $i = 1, 2, \ldots, N$ found in the training data, i.e.

$$\widehat{f}(x) = \begin{cases} y_i & x = x_i \text{ for some } i = 1, 2, \ldots, N \\ \text{anything} & \text{otherwise} \end{cases}. \tag{2.4}$$

Obviously, $R(\widehat{f}) = 0$ but $\widehat{f}$ is not what we want. What we really want is $\widehat{f}$ to perform well on *new* examples not found in, but distributed identically as, the original training dataset. In mathematical terms, what we really want to solve is the *population risk minimization* problem

$$\min_{f \in \mathcal{H}} R_{\text{pop}}(f) = \mathbb{E}_{x \sim \mu} L(f(x), f^*(x)), \tag{2.5}$$

where μ denotes the probability distribution from which the samples $\{x_i\}_{i=1}^N$ are sampled from. In contrast, (2.3) is an *empirical risk minimization* problem, which we can rewrite as

$$\min_{f \in \mathcal{H}} R_{\text{emp}}(f) = \frac{1}{N} \sum_{i=1}^{N} L(f(x_i), f^*(x_i)) \qquad x_i \overset{\text{i.i.d.}}{\sim} \mu. \tag{2.6}$$

However, in practice we cannot represent the sample distribution μ, and hence we often resort to solving (2.6) in place of (2.5). Nevertheless, it must be stressed that a good solution of (2.5) is what we are really after. The difference between the solutions of these problems is the study of *generalization*, which sets learning problems apart from pure optimization problems. We will come back to this point when we venture into the basics of statistical learning theory.

Three Paradigms of Supervised Learning

Now that we have formalized the basic problem of supervised learning, it is natural to discuss what sort of questions can we ask in machine learning theory and practice. In a sense, these questions can be grouped into three large categories: *approximation*, *optimization* and *generalization*. Below we list some central questions in each of these aspects.

1. Approximation – *How large is our hypothesis space $\mathcal{H}$?*. In particular, does it include, or at least contain functions that are very close to our oracle f^*? This is in fact the study of *approximation theory* and some of *harmonic analysis* [DP07, Mal09], although there are also many modern developments, particularly in the area of deep learning.
2. Optimization – *How can we find or get close to an approximation $\widehat{f}$ of f^*?*. This is indeed the empirical risk minimization problem, and questions include the design of large-scale optimization algorithms, their convergence analysis and their efficient implementation. Many methods are extensions of classical methods in convex optimization [Nes04]. See [BCN18] for a modern review. We will cover some basics later in the course.
3. Generalization – *Can the $\widehat{f}$ found generalize to unseen examples?* This concerns the fundamental interaction between the size of the data and the complexity of our hypothesis space. In fact, this question is the focus of classical statistical learning theory [FHT01]. We will discuss the basics later in this course, although there are many modern developments as well that we cannot cover in this course.

Figure 2.1 gives an illustration of these questions. Of course, some of these concepts also apply beyond supervised learning framework.

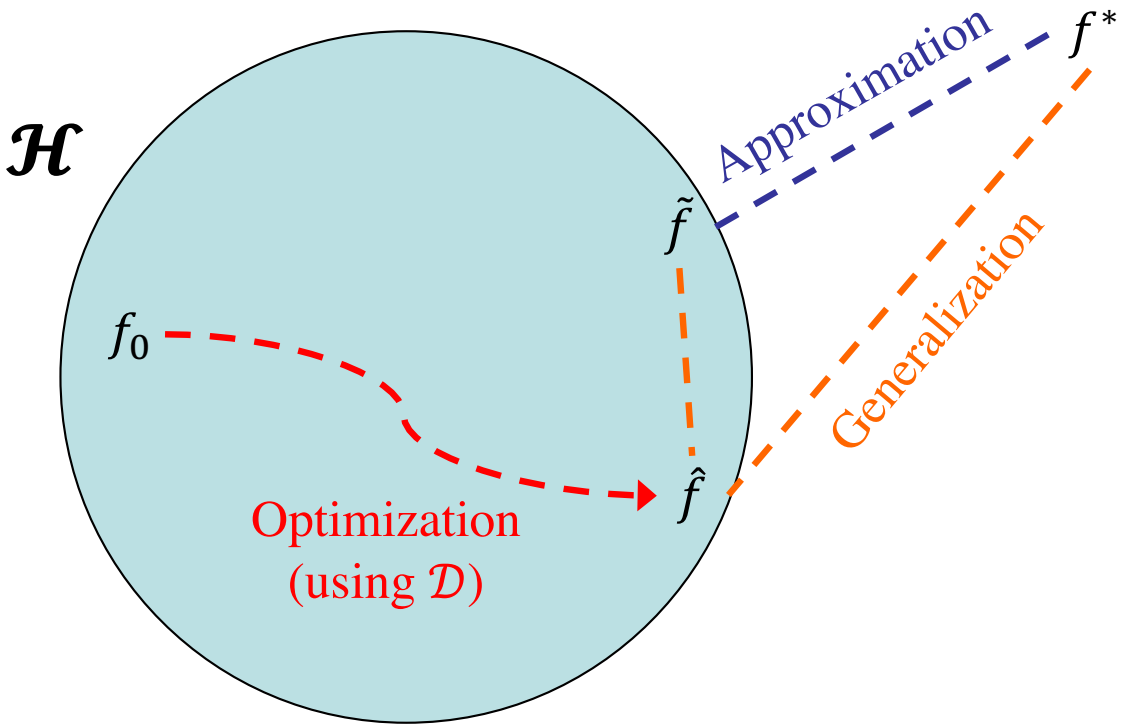


Figure 2.1: A schematic showing the three theoretical paradigms of supervised learning. *Approximation* studies the distance between the best approximator $\tilde{f}$ in our hypothesis space $\mathcal{H}$ and the oracle f^*. *Optimization* studies the process at which you arrive at or close to $\tilde{f}$, by using the training dataset $\mathcal{D}$ and starting from some initial guess f_0. *Generalization* problems arise because the dataset is finite and so the optimization on training set finds not $\tilde{f}$ but some other $\hat{f}$, and we must quantify the distance between them. In fact, what we are really interested in is the distance between $\hat{f}$ and f^* (or $\tilde{f}$) when it comes to generalization.

2.2 Linear Models

A linear model is perhaps the most basic choice of hypothesis space in machine learning. In this section, we introduce basic linear models for regression and classification problems, with the goal of illustrating the key concepts, questions and approaches in supervised learning that we discussed previously in more abstract terms. This should set a simple but useful baseline as many central issues in supervised learning can be very much understood using examples in linear models.

2.2.1 Simple Linear Regression

We begin with a review of simple linear regression in one dimension – an example we have probably all seen in school. However, we have mostly seen it in a statistical context. Here, our goal is to perceive various phenomenon encountered in linear regression in the machine learning view point (approximation, optimization and generalization), serving as a concrete preparation as we deal with more complex machine learning models in the subsequent chapters.

Now, suppose that we are given a bunch of points $\mathcal{D} = \{x_i, y_i\}_{i=1}^N$ (with x_i, y_i real numbers) and our goal is to fit a line to them – with the goal that if some new point x' arrives, we will be able to predict its corresponding y'. Importantly, note that we are not assuming that each (x_i, y_i) follow a linear relationship, but rather, we *choose a linear hypothesis space* and try to find the best function in this hypothesis space to approximate the relationship between (x_i, y_i), which may be due to an unknown oracle f^* so that $y_i = f^*(x_i)$ – and f^* need not be linear! In the language of supervised learning introduced earlier, the hypothesis space we choose in simple linear regression is the space of linear functions[1] in one variable, i.e.

$$\mathcal{H} = \{f : f(x) = w_0 + w_1 x, w_0 \in \mathbb{R}, w_1 \in \mathbb{R}\}. \tag{2.7}$$

How do we determine the best approximator from $\mathcal{H}$? Following the approach in 2.1, we first define an appropriate loss function. One of the most commonly used loss functions in linear regression is the mean-squared loss $L(y', y) = \frac{1}{2}(y - y')^2$, in which case we obtain the empirical risk minimization problem

$$\min_{f \in \mathcal{H}} R_{\text{emp}}(f) = \frac{1}{2N} \sum_{i=1}^{N} (f(x_i) - y_i)^2 \tag{2.8}$$

which we can write in explicit parametric form

$$\min_{(w_0, w_1) \in \mathbb{R}^2} R_{\text{emp}}(w_0, w_1) = \frac{1}{2N} \sum_{i=1}^{N} (w_0 + w_1 x_i - y_i)^2. \tag{2.9}$$

The minimization problem (2.9) can be solved by simple calculus, setting $\frac{\partial R_{\text{emp}}}{\partial w_0} = 0$ and $\frac{\partial R_{\text{emp}}}{\partial w_1} = 0$, which yields the solution $\widehat{f}(x) = \widehat{w}_0 + \widehat{w}_1 x$ with

$$\widehat{w}_0 = \overline{y} - \widehat{w}_1 \overline{x}, \qquad \widehat{w}_1 = \frac{\sum_{i=1}^N (x_i - \overline{x})(y_i - \overline{y})}{\sum_{i=1}^N (x_i - \overline{x})^2}, \qquad \overline{x} = \frac{1}{N} \sum_{i=1}^{N} x_i, \qquad \overline{y} = \frac{1}{N} \sum_{i=1}^{N} y_i. \tag{2.10}$$

This is also called *ordinary least squares* solution, owing to the fact that we have used the mean square error as the loss function. Now, let us discuss some of the aforementioned issues in Section 2.1 in the concrete context of linear regression.

[1] Strictly speaking, these functions are *affine* in x, but are linear in $(1, x)$. f

Exercise 2.1

Derive the ordinary least squares formula (2.10). This is a special case of a general least squares formula to be derived in Proposition 2.7.

Approximation. We first illustrate the approximation problem. Recall that this deals with the size of our hypothesis space – is it big enough to approximate the function f^* we are interested in? We will illustrate this with a simple yet instructive example below.

Example 2.2: The Approximation Problem

Let us consider inputs sampled from a grid on the interval $[0, 1]$, i.e. $x_i = i/N$, for $i = 1, 2, \ldots, N$. Here the x_i's are chosen deterministically so there are no issues with generalization – we simply want to fit $\{x_i, y_i\}_{i=1}^N$ where the labels y_i are produced by some target oracle function f^*, i.e. $y_i = f^*(x_i)$.

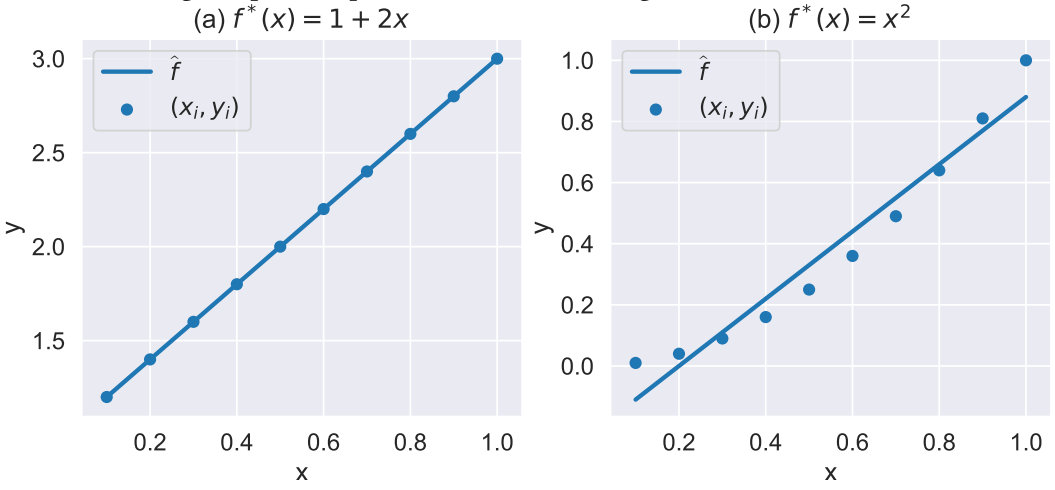
(a) First, consider the case where f^* is itself linear, e.g. $f^*(x) = 1 + 2x$, so that $y_i = 1 + 2i/N$. Verify that the least squares formula (2.10) indeed gives $\widehat{w}_0 = 1, \widehat{w}_1 = 2$, i.e. our hypothesis space is large enough that $f^* \in \mathcal{H}$ and $\min_{f \in \mathcal{H}} R_{\text{emp}}(f) = 0$. In this case we say that there is no approximation error.

(b) Now, consider the case where f^* is nonlinear, e.g. $f^*(x) = x^2$, in which case $y_i = i^2/N^2$. Verify that the least squares formula (2.10) gives

$$\widehat{w}_0 = -\frac{(N+1)(N+2)}{6N^2}, \qquad \widehat{w}_1 = \frac{1}{N} + 1. \tag{2.11}$$

For large N this is approximately $\widehat{w}_0 \approx -1/6, \widehat{w}_1 \approx 1$. In this case, we have an approximation error since the linear function $\widehat{f}(x) = \widehat{w}_0 + \widehat{w}_1 x$ is not a perfect fit for the data. In particular, $\min_{f \in \mathcal{H}} R_{\text{emp}}(f) > 0$, meaning that our linear hypothesis space is not big enough to approximate f^*.

In the following, we plot the performance of linear regression for the two cases above.



Optimization. There is not much to say about optimization since the least squares formula (2.10) is actually explicitly derived from calculus. In general cases (say, with different loss functions), the empirical risk minimization problem cannot be solved explicitly, and usually we use some iterative method. The simplest of which is *gradient descent*, where we perform the

following updates

$$\begin{aligned} w_0^{k+1} &= w_0^k - \eta \frac{\partial R_{\text{emp}}(w_0^k, w_1^k)}{\partial w_0} \\ w_1^{k+1} &= w_1^k - \eta \frac{\partial R_{\text{emp}}(w_0^k, w_1^k)}{\partial w_1} \end{aligned}. \tag{2.12}$$

In fact, for on-line least squares problems we also often resort to iterative updates simply because not all data is available to us at the same time (See Chapter 3.1.3 of [BO06]). We will discuss the basic properties of optimization problems in machine learning later.

Generalization. The simple linear regression problem can also demonstrate the problem of generalization. Let us now assume that the x_i's are actually random variables independently and identically distributed (i.i.d.) according to some probability density μ on $\mathbb{R}$. That is to say,

$$\mathbb{P}[a \le x_i \le b] = \int_a^b \mu(x)dx, \qquad a \in \mathbb{R}, b \in \mathbb{R}. \tag{2.13}$$

Then, the population risk minimization problem with mean square loss (recall (2.5)) is

$$\begin{aligned} \min_{(w_0, w_1) \in \mathbb{R}^2} R_{\text{pop}}(w_0, w_1) &= \mathbb{E}_{x \sim \mu} \frac{1}{2}(w_0 + w_1 x - f^*(x))^2 \\ &= \frac{1}{2} \int_{\mathbb{R}} (w_0 + w_1 x - f^*(x))^2 \mu(x) dx \end{aligned} \tag{2.14}$$

This is evidently different from the empirical risk minimization problem (2.9). So, how close are their solutions? We explore this in the following example.

Example 2.3: Population Risk Minimization

Let us take μ to be the uniform density on $[0, 1]$, i.e. $\mu(x) = 1$ for all $x \in [0, 1]$ and 0 otherwise. Take $f^*(x) = x^2$ as in case (b) in Example 2.2. Computing the integral in (2.14), we have

$$R_{\text{pop}}(w_0, w_1) = \frac{w_0^2}{2} + \frac{w_0 w_1}{2} - \frac{w_0}{3} + \frac{w_1^2}{6} - \frac{w_1}{4} + \frac{1}{10}, \tag{2.15}$$

whose minimizer is $\widetilde{w}_0 = -\frac{1}{6}$ and $\widetilde{w}_1 = 1$. Contrast with empirical risk minimization,which draws N random samples $\{x_i\}_{i=1}^N$ from μ instead. Do you expect $\widehat{w}_0$ and $\widehat{w}_1$ to be equal or close to $-\frac{1}{6}$ and 1 if N is small? What happens when N is increased?

We have seen from Example 2.2(b) that if our hypothesis space is too small, we cannot fit the relationship between x_i and y_i no matter how many data points of we have. This is in fact a case of *underfitting*, i.e. the complexity of our hypothesis space is too low. On the other hand,

the complexity of our hypothesis space can be too high relative to the number of samples, in this case we would experience *overfitting*. Below we give some examples of overfitting.

Example 2.4: Overfitting

Let us consider x_i's a grid in $[0, 1]$ as in Example 2.2, and the following two cases

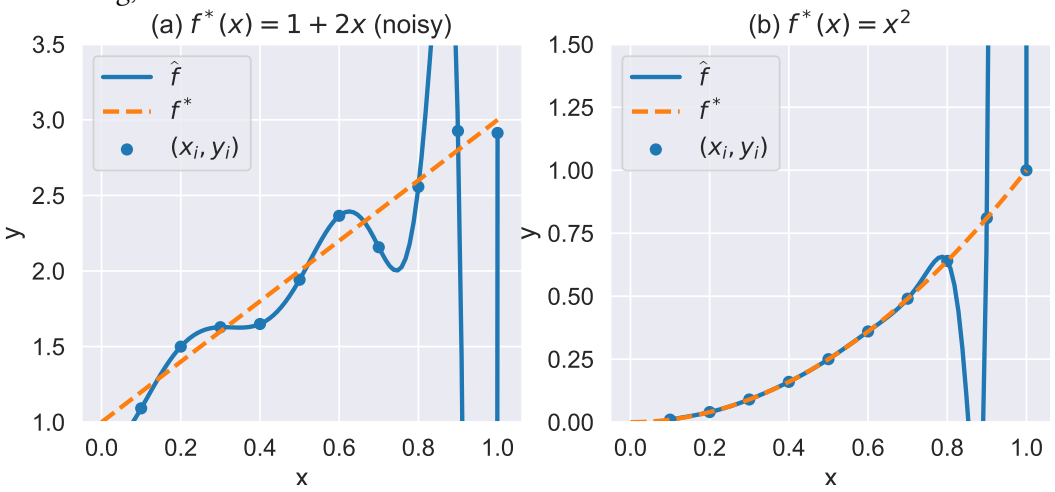
(a) $y_i = 1 + 2x_i + 0.1\epsilon_i$ where ϵ_i are i.i.d. standard normal variables.

(b) $y_i = x_i^2$.

We know that for (b), a linear hypothesis space is not enough, so we are going to do something drastic and consider a much larger hypothesis space, consisting of polynomials up to degree 99.

$$\mathcal{H} = \left\{ f : f(x) = \sum_{j=0}^{99} w_j x^j \right\} \tag{2.16}$$

Below we show the result of linear regression with mean square loss (least squares solution) with $N = 10$ on the two settings. Are these the fit that we want? This is demonstration of overfitting, which can occur both with or without noise.



As a side note, you may wonder if given arbitrarily high order polynomials, can we fit any function we want? The answer is yes if we consider a closed and bounded domain. This is the content of a classical result in approximation theory, called *Weierstrass approximation theorem*, which is later generalized by Stone [Sto48].

Exercise 2.5

Investigate the effect of overfitting in Example 2.4 by increasing N. We will come back to the relationship between overfitting and sample size when we introduce statistical learning theory.

Loss Functions. So far in our discussion for the simple least squares problem, we have always used the mean square loss $L(y', y) = \frac{1}{2}(y - y')^2$. This is also sometimes called the ℓ^2 loss. The ℓ^2 loss is often used for the simple reason that the ordinary least squares problem (and in fact, many other problems using the mean square loss) admits explicit solutions that simplify analysis. Moreover, least squares solution in general linear basis models have a nice interpretation as orthogonal projections onto the subspace spanned by the basis functions.

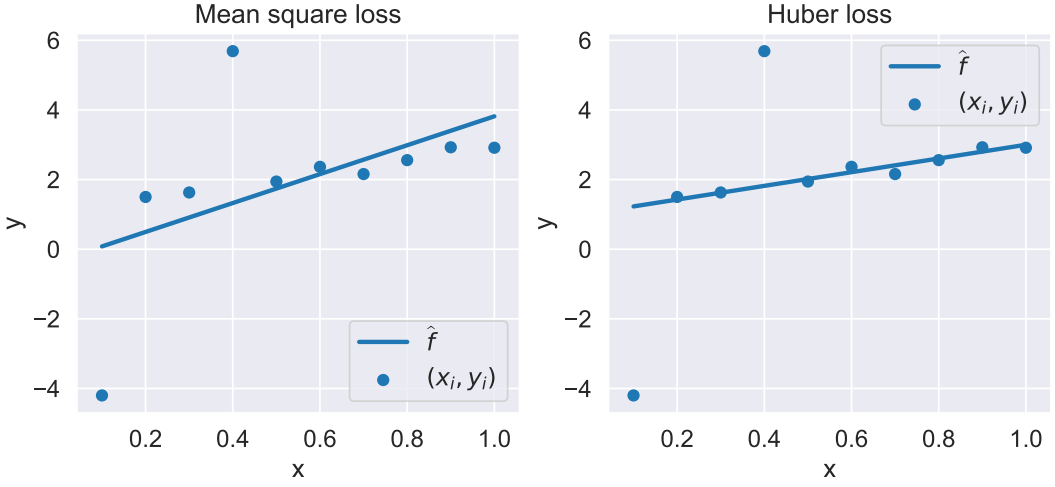
However, we can also consider other loss functions. For example, for classification problems we typically use the cross-entropy loss to be introduced later. Even for regression problems, there are also other choices. One example is the *Huber loss*, which combines the mean square ℓ^2 loss and the absolute (ℓ^1) loss

$$L(y', y) = \begin{cases} \frac{1}{2}(y - y')^2 & |y - y'| \le \delta \\ \delta|y - y'| - \frac{1}{2}\delta^2 & \text{otherwise} \end{cases}. \tag{2.17}$$

Here, δ controls the point of switching between mean square and absolute loss. In the example below we illustrate the effect of the choice of loss functions on linear regression.

Example 2.6: Loss Functions

We apply 1D linear regression to Example 2.4(a), but with some noise and outliers. We use different loss functions, namely mean-square loss and the Huber loss ($\delta = 1$). The results are shown below. Observe that the Huber loss is more robust to outliers. Why?



2.2.2 General Linear Basis Models

Often we want to apply linear regression to more general cases than those considered in Section 2.2.1. For instance, we want to have more than one input dimension (e.g. each $x_i \in \mathbb{R}^d$), or more complicated hypothesis spaces (e.g. the polynomials considered in Example 2.4). It

turns out that these cases can all be handled quite easily by formulating a generalization of the simple linear regression, called *linear basis models*.

Consider now the dataset $\mathcal{D} = \{x_i, y_i\}_{i=1}^N$ where each input $x_i \in \mathbb{R}^d$ is now a vector in d dimensions. Instead of simple linear hypothesis space, we consider the more general linear basis hypothesis space

$$\mathcal{H}_M = \left\{ f : f(x) = \sum_{j=0}^{M-1} w_j \phi_j(x) \right\} \tag{2.18}$$

where each ϕ_j is a function from $\mathbb{R}^d$ to $\mathbb{R}$. These are called *basis functions* or sometimes *feature maps*, for the reason that their role is to extract some feature from the input data x that is useful for predicting y. We have freedom to define what functions $\{\phi_j\}$ we use. The subscript M on $\mathcal{H}_M$ emphasizes the fact that the complexity or size of our hypothesis space depends on M and generally increases as M increases.

Observe that this is in fact a generalization of simple linear regression ($d = 1$), if we choose $M = 2$ and $\phi_0(x) = 1$, $\phi_1(x) = x$, but it also includes the polynomial hypothesis space in Example 2.4. There are many choices of ϕ_j's, and we give some examples below in the $d = 1$ case:

$$\text{Polynomial basis} \qquad \phi_j(x) = x^j \tag{2.19}$$

$$\text{Gaussian basis} \qquad \phi_j(x) = \exp\left(-\frac{(x - m_j)^2}{2s^2}\right) \tag{2.20}$$

$$\text{Sigmoid basis} \qquad \phi_j(x) = \sigma\left(\frac{x - m_j}{s}\right) \qquad \sigma(b) = \frac{1}{1 + e^{-b}} \tag{2.21}$$

There are in fact many more choices, include splines [DBRDB78], fourier basis, wavelet basis [Mal09] and many more – even neural networks as we will encounter later on.

Ordinary Least Squares. It turns out that the least squares formula (2.10) can be generalized for linear basis models. Recall that the empirical risk minimization problem we want to solve here is

$$\min_{w_0, \ldots, w_{M-1}} R_{\text{emp}}(w_0, \ldots, w_{M-1}) = \frac{1}{2N} \sum_{i=1}^{N} \left(\sum_{j=0}^{M-1} w_j \phi_j(x_i) - y_i \right)^2. \tag{2.22}$$

We can write the above much more compactly as

$$\min_{w \in \mathbb{R}^M} R_{\text{emp}}(w) = \frac{1}{2N} \|\Phi w - y\|^2, \tag{2.23}$$

where we have defined

$$w = \begin{pmatrix} w_0 \\ w_1 \\ \vdots \\ w_{M-1} \end{pmatrix}, \qquad y = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{pmatrix}, \qquad \Phi = \begin{pmatrix} \phi_0(x_1) & \phi_1(x_1) & \cdots & \phi_{M-1}(x_1) \\ \phi_0(x_2) & \phi_1(x_2) & \cdots & \phi_{M-1}(x_2) \\ \vdots & \vdots & \ddots & \vdots \\ \phi_0(x_N) & \phi_1(x_N) & \cdots & \phi_{M-1}(x_N) \end{pmatrix}, \tag{2.24}$$

and $\|\cdot\|$ denotes the usual Euclidean distance. This allows us to derive the ordinary least squares formula by solving $\nabla R_{\text{emp}}(\widehat{w}) = 0$ for $\widehat{w}$.

Proposition 2.7: Ordinary Least Squares Formula

Suppose $\Phi^\top\Phi$ is invertible, then the solution of (2.23) is

$$\widehat{w} = (\Phi^\top\Phi)^{-1}\Phi^\top y. \tag{2.25}$$

Proof: We have $\nabla R_{\text{emp}}(\widehat{w}) = \frac{1}{N}\Phi^\top(\Phi\widehat{w} - y)$. Setting the right hand side to 0 gives $\Phi^\top\Phi\widehat{w} = \Phi^\top y$. Solving for $\widehat{w}$ gives the required solution. □

Exercise 2.8

Show that Proposition 2.7 is consistent with the simple linear regression formula (2.10) when $d = 1, M = 2, \phi_0(x) = 1, \phi_1(x) = x$.

Singular Case and Regularization. Proposition 2.7 requires $\Phi^\top\Phi$ to be invertible, but what happens if this is not the case? Note that Φ is a $N \times M$ matrix whose rank is at most $\min(N, M)$. Suppose $N \geq M$, which is the case when the number of samples is greater than or equal to the number of features, or equivalently, the "complexity" of our hypothesis class. Since $\Phi^\top\Phi$ is a $M \times M$ matrix, it is invertible as long as the columns of Φ are linearly independent, which we should be able to satisfy by appropriate choices of the ϕ_j's. What happens if $N < M$? In this case the rank of $\Phi^\top\Phi$ is at most N which is smaller M and so it is non-invertible, or *singular*.

So, what is the solution of the least squares problem now? It turns out that there is not one, but an *infinite* number of solutions. They are given by

$$\{\widehat{w}(u) : u \in \mathbb{R}^M\} \quad \text{where} \quad \widehat{w}(u) = \Phi^\dagger y + (I - \Phi^\dagger\Phi)u. \tag{2.26}$$

The matrix $\Phi^\dagger$ denotes the *Moore–Penrose pseudoinverse* [GVL96] of Φ. Moreover, for any of these solutions, we have $R_{\text{emp}}(\widehat{w}(u)) = 0$, i.e. our training data is perfectly fit. Recall from Example 2.4 that this may not be good in machine learning scenarios.

Exercise 2.9

Show that if $\Phi^\top\Phi$ is invertible then $\widehat{w}(u)$ from (2.26) reduces to the ordinary least squares solution (2.25) for any u. (Hint: recall that for a matrix A, if $A^\top A$ is invertible then $A^\dagger = (A^\top A)^{-1}A^\top$. If you do not remember this, review the basics of pseudoinverse in any linear algebra reference, e.g. [GVL96]).

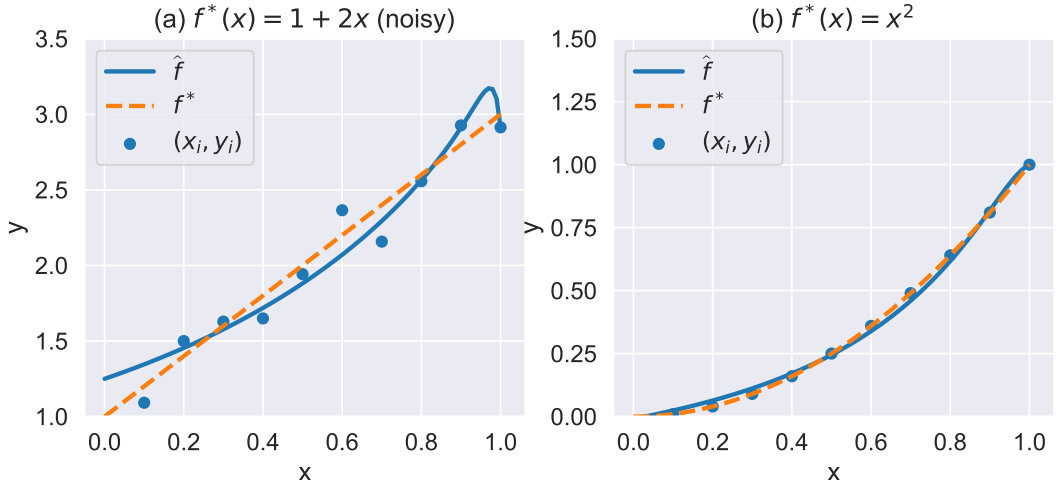
The fact that we have an infinite number of solutions is often not good as we typically need to pick one to perform our predictions. Which one do we pick? One choice is to pick one that has the smallest norm $\|\widehat{w}(u)\|$, which is $\widehat{w}(0) = \Phi^\dagger y$. More generally, we can consider adding to the empirical risk minimization problem a *regularization* term

$$\min_{w \in \mathbb{R}^M} \frac{1}{2N} \|\Phi w - y\|^2 + \lambda C(w) \tag{2.27}$$

where $C : \mathbb{R}^M \to \mathbb{R}_+$ is the regularization function and $\lambda > 0$ controls its strength. If we pick $C(w) = \|w\|^2$ (ℓ^2 regularization), then we get the unique solution $\widehat{w} = \Phi^\dagger y$ if $\Phi^\top \Phi$ is singular. This is also known as *ridge regression*. However, we can also use other types of regularization, such as $C(w) = \|w\|_1$ (ℓ^1 regularization). In this case, we actually find a *sparse* solution, i.e. many of entries $\widehat{w}$ are 0. This is called *lasso* [FHT01] in the statistics literature and is also related to the field of *compressed sensing* [Don06], which has many interesting applications, including fast Magnetic Resonance Imaging (MRI) [LDP07]. Moreover, regularization can also reduce overfitting, as the following example shows.

Example 2.10: Regularization and Overfitting

We apply the ℓ^2 regularization to the two cases in Example 2.4. The results are shown below. Observe that overfitting is reduced.



2.2.3 Classification using Linear Models

Thus far we have only discussed regression problems. Classification problems can be similarly handled by linear basis models. Recall that in classification problems, the labels y_i are categorical variables, meaning that they take a finite number of values which have no natural order. As discussed in Section 1.2, a common way to handle this case is to use the one-hot embedding – suppose that each label y_i can take values in the set of K possible classes $\{c_1, c_2, \ldots, c_K\}$, we

can then regard each y_i as a one-hot vector of length K, i.e. if the input x_i belongs to class c_k then

$$y_i = (0, \ldots, 0, \underbrace{1}_{\text{k}^{\text{th}} \text{ position}}, 0, \ldots, 0). \tag{2.28}$$

Note that in this case, the underlying oracle function f^* maps each x_i to a vertex of the K-dimensional hypercube. Consequently, we need to form a hypothesis space containing predictors that outputs a K-dimensional vector from a single input. That is, the range of every $f \in \mathcal{H}$ should be K-dimensional. Hence, it is natural to consider linear basis models of the form

$$\mathcal{H} = \left\{ f : f(x) = g\left(\sum_{j=0}^{M-1} w_j \phi_j(x) \right), w_j \in \mathbb{R}^K \right\}. \tag{2.29}$$

Comparing to (2.18), we see that each weight w_j is now a K-dimensional vector instead of a real number and moreover, we have an additional *activation function* $g : \mathbb{R}^K \to \mathbb{R}^K$ after the linear transformation. This is because we want to compare each f with f^* whose output is a one-hot vector, i.e. a vertex of the K-dimensional hypercube, so we want f to output values also on the vertex or something similar to enable comparison. The most popular choice of g is the *softmax* function, which is given by

$$g_{\text{sm}}(z)_k = \frac{\exp(z_k)}{\sum_{\ell=1}^{K} \exp(z_\ell)}. \tag{2.30}$$

It is called softmax because it is a smooth approximation to the maximum function which converts each z to an one-hot vector having 1 at the position of the maximal element of z, i.e.

$$g_{\max}(z)_k = \begin{cases} 1 & z_k > z_\ell \text{ for all } \ell \neq k \\ 0 & \text{otherwise} \end{cases}. \tag{2.31}$$

For example, if $z = (1.2, -3.5, 2.5)$ then $g_{\max}(z) = (0, 0, 1)$. The softmax function is an approximation to this function. One small technicality is that there may be no unique maximal element in z, in which case we can take the first maximal coordinate or a random maximal coordinate and set it to 1. Alternatively, we can divide 1 evenly over the set of maximal coordinates. In the following, we will restrict our attention to softmax activation functions. Notice that unlike regression problems, our functions $f \in \mathcal{H}$ are no longer linear functions of the weights $\{w_j\}$, due to the presence of the nonlinear activation function g.

Exercise 2.11

Show that for any y, $g_{\text{sm}}(z)_k > 0$ for all k and $\sum_{k=1}^{K} g_{\text{sm}}(z)_k = 1$. Thus, the softmax function is often used to model probabilities. Show also that for any z with a unique maximal coordinate, $g_{\text{sm}}(\lambda z) \to g_{\max}(z)$ as $\lambda \to \infty$, which justifies why the softmax function is an approximation of the maximal function.

Following the approach for regression problems, we can cast supervised learning for classification problems with linear basis models as the following empirical risk minimization problem

$$\min_{W \in \mathbb{R}^{M \times K}} R_{\text{emp}}(W) = \frac{1}{N} \sum_{i=1}^{N} L(g_{\text{sm}}(\Phi W)_i, y_i), \tag{2.32}$$

where now the trainable weights is written as a $M \times K$ matrix W, whose j^{th} row is w_j. Notice that we have not specified the loss function L, which is now a function from $\mathbb{R}^K \times \mathbb{R}^K$ to $\mathbb{R}$. One possibility is to take the mean square loss again, but it is often better to take the *cross-entropy* loss, given by

$$L(y', y) = -\sum_{k=1}^{K} y_k \log y'_k. \tag{2.33}$$

Observe that the definition of the cross-entropy loss above requires both arguments to be probability distributions over K elements, i.e. both y and y' should have non-negative entries smaller than or equal to 1, and their entries sum to 1. Moreover, notice that $L(y', y) \neq L(y, y')$ in general, meaning that the loss function is non-symmetric, unlike ℓ^p losses.

Exercise 2.12

Show that if y is a one-hot vector, then $L(\cdot, y)$ is minimized at $y' = y$.

Unlike least-squares regression, the empirical risk minimization problem for classification rarely admits explicit solutions and are often solved by iterative methods, such as (stochastic) gradient descent. We now illustrate the application of linear models for classification on a digit recognition problem on the MNIST dataset [LCB10].

Example 2.13: Digit Recognition on MNIST using Linear Models

The MNIST dataset [LCB10] contains 70000 hand-written digits from 0 to 9. The input images are 28×28 gray-scale images (which we flatten into a 784 dimensional vector and their respective numbers labeled. We will use the simplest linear model with $M = 28^2 + 1$, $\phi_0 = 1$, $\phi_j(x) = x_j$, together with softmax activation and cross-entropy loss. Note that this is also called multi-class logistic regression. Training the model gives about 92% test accuracy on a test set of 10000 samples. Can we do better with different choices of ϕ_j's?

2.2.4 Further Reading

In this section, we introduced linear models for regression and classification, with particular emphasis on explicit examples with simple linear regression to demonstrate some key issues

in machine learning that we will meet again and again later on in the course. These include approximation errors, optimization algorithms, generalization gaps, regularization, choice of loss functions etc.

Invariably, with the emphasis on simplicity and the big picture, we miss many interesting issues. These include but are not limited to, probabilistic and statistical interpretations of linear models (in particular, with noise and uncertainty), the issue of decision boundaries, and on-line linear regression and classification. The interested reader is encouraged to browse textbooks on the subject to explore these topics in greater detail, e.g. [BO06, MRT18]

2.3 Kernel Methods

In Section 2.2.2, we saw that general linear basis models work by transforming the inputs x_i, which live in $\mathbb{R}^d$, via a class of functions $\{\phi_j\}$ into $\mathbb{R}$. We called these functions *feature maps*, for the simple reason that they extract useful features from the input data and allow us to use a linear model on the resulting space, which in this case is $\mathbb{R}$. In this section, we will develop this idea in greater generality in the context of *kernel methods*.

2.3.1 Least Squares Revisited

Let us first motivate ideas by revisiting the least squares solution of general linear models presented in Section 2.2.2. We consider the ℓ^2-regularized least squares problem (also called ridge regression) with respect to the feature maps $\{\phi_j\}_{j=0}^{M-1}$, given by

$$R_{\text{emp}}(w) = \frac{1}{2N}\left(\|\Phi w - y\|^2 + \lambda\|w\|^2\right). \tag{2.34}$$

Recall the definitions of the design matrix Φ in (2.24). We assume that the regularization parameter λ is greater than 0. Following the same approach as in Proposition 2.7, we can show that the least squares solution is given by

$$\widehat{w} = (\Phi^\top\Phi + \lambda I_M)^{-1}\Phi^\top y, \tag{2.35}$$

and the predicted outputs is $\widehat{y} = \Phi\widehat{w}$. More generally, the best approximator function $\widehat{f}$ in our hypothesis space makes the following prediction on a new sample $x \in \mathbb{R}^d$

$$\widehat{f}(x) = \phi(x)^\top\widehat{w}, \tag{2.36}$$

where $\phi(x) = (\phi_0(x), \ldots, \phi_{M-1}(x))$. Notice that unlike in Proposition 2.7, we did not need any invertibility conditions here because the matrix $\Phi^\top\Phi + \lambda I_M$ is always positive definite for $\lambda > 0$, hence invertible.

Exercise 2.14

Derive the regularized ordinary least squares solution (2.35).

We now rewrite the regularized least squares solution in another way. First we claim that

$$\widehat{w} = (\Phi^\top \Phi + \lambda I_M)^{-1} \Phi^\top y = \Phi^\top (\Phi\Phi^\top + \lambda I_N)^{-1} y. \tag{2.37}$$

To show this, multiply $(\Phi^\top \Phi + \lambda I_M)$ to $\Phi^\top (\Phi\Phi^\top + \lambda I_N)^{-1} y$ to give

$$\begin{aligned}(\Phi^\top \Phi + \lambda I_M)\Phi^\top (\Phi\Phi^\top + \lambda I_N)^{-1} y &= (\Phi^\top \Phi\Phi^\top + \lambda \Phi^\top)(\Phi\Phi^\top + \lambda I_N)^{-1} y && (2.38)\\ &= \Phi^\top (\Phi\Phi^\top + \lambda I_N)(\Phi\Phi^\top + \lambda I_N)^{-1} y && (2.39)\\ &= \Phi^\top y && (2.40)\end{aligned}$$

and hence we can multiply $(\Phi^\top \Phi + \lambda I_M)^{-1}$ to both sides to obtain (2.37). Consequently, we can write $\widehat{f}$ as

$$\widehat{f}(x) = \phi(x)^\top \widehat{w} = \phi(x)^\top \Phi^\top (\Phi\Phi^\top + \lambda I_N)^{-1} y. \tag{2.41}$$

By defining $\alpha = (\Phi\Phi^\top + \lambda I_N)^{-1} y$, we can rewrite the above as

$$\widehat{f}(x) = \sum_{i=1}^{N} \alpha_i \phi(x)^\top \phi(x_i). \tag{2.42}$$

The formulae (2.42) and (2.36) are equivalent representations, so what have we achieved by going through all these trouble? Observe that the i, j component of the $N \times N$ matrix $\Phi\Phi^\top$ can be written as $\phi(x_i)^\top \phi(x_j)$. Consequently, unlike (2.36), the formula (2.42) only depends on our feature map ϕ through the function $(x, x') \mapsto \phi(x)^\top \phi(x')$. In other words, we may define a *kernel* function $k : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$ by

$$k(x, x') = \phi(x)^\top \phi(x'). \tag{2.43}$$

Then, $\widehat{f}$ only depends on the feature maps through the kernel k, which once known, allows us to *never* compute any feature maps $x \mapsto \phi(x)$ directly in order to make a prediction! To see this more clearly, we can further rewrite (2.42) as

$$\widehat{f}(x) = \sum_{i=1}^{N} [(G + \lambda I_N)^{-1} y]_i k(x, x_i), \qquad G_{ij} = k(x_i, x_j). \tag{2.44}$$

The matrix G is called the *Gram* matrix. What is the advantage that comes with not having to work with explicit feature maps $(\phi_0, \ldots, \phi_{M-1})$? First, if M is very large yet we have some good way of evaluating k, this would save cost in practice. More importantly, this allows us to consider perhaps the case $M = \infty$, as long as we can compute the kernel! This is the key idea that underlies kernel methods: we can often directly work with the kernel function k and not explicit feature maps $\{\phi_j\}$. This holds true for many cases beyond the regularized ordinary least squares problem. Now, let us discuss some mathematical basics behind kernel functions.

2.3.2 Basic Properties of Kernels

Previously, we saw that for any set of feature maps $\{\phi_j\}$, we can construct a kernel k given by

$$k(x, x') = \phi(x)^\top \phi(x') = \sum_{j=0}^{M-1} \phi_j(x)\phi_j(x'). \tag{2.45}$$

What does k look like? Let us show in the example below some correspondences between feature maps and kernel functions.

Example 2.15: Kernel and Feature Maps

Let $d = 1, M = 2$ and $\phi_0(x) = 1, \phi_1(x) = x$, so that we are in the case of 1D simple linear regression. Then we have

$$k(x, x') = 1 + xx'. \tag{2.46}$$

Let us now take things in reverse. Consider $d = 2$ and a function defined by

$$k(x, x') = (1 + x^\top x')^2. \tag{2.47}$$

Let us now show that k corresponds to a feature map. Observe that

$$\begin{aligned} k(x, x') &= (1 + x_1 x_1' + x_2 x_2')^2 \\ &= (1, \sqrt{2}x_1, \sqrt{2}x_2, \sqrt{2}x_1x_2, {x_1}^2, {x_2}^2) \cdot (1, \sqrt{2}x_1', \sqrt{2}x_2', \sqrt{2}x_1'x_2', {x_1'}^2, {x_2'}^2) \end{aligned} \tag{2.48}$$

Hence, the kernel function (2.47) corresponds to the feature map

$$\phi(x) = (1, \sqrt{2}x_1, \sqrt{2}x_2, \sqrt{2}x_1x_2, {x_1}^2, {x_2}^2). \tag{2.49}$$

Example 2.15 shows that not only can we derive corresponding kernel functions from feature maps, we seems to be able to do the reverse – specifying a kernel function *implicitly* defines a feature map. Now, does this hold for *any* function $k : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$?. The answer is false. To see this, observe that for any vector of feature maps ϕ, we have $\phi(x)^\top \phi(x) = \|\phi(x)\|^2 \geq 0$. Therefore, if $k(x, x) < 0$ for some x then it cannot be expressed as $k(x, x') = \phi(x)^\top \phi(x')$ for any ϕ. For example, $k(x, x') = x^\top x' - 1$ cannot correspond to a feature map since $k(0, 0) = -1 < 0$. Moreover, observe that for k to correspond to some ϕ, it must also be symmetric in its arguments, i.e. $k(x, x') = k(x', x)$. Therefore, to discuss kernel methods in general, we should place some restrictions on what functions we consider to be valid kernels that defines feature maps.

Definition 2.16: Symmetric Positive Definite Kernels

A function $k : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$ is called a *symmetric positive definite* (SPD) kernel if for any collection $\{x_1, \ldots, x_n\}$ of vectors in $\mathbb{R}^d$, the Gram matrix G with elements $G_{ij} = k(x_i, x_j)$ is symmetric positive semi-definite. Recall that a $n \times n$ matrix G is symmetric positive definite if the following are satisfied

1. (Symmetric) $G_{ij} = G_{ji}$ for any $i, j = 1, \ldots, n$.
2. (Positive Semi-definite) $c^\top G c \geq 0$ for any $c \in \mathbb{R}^n$.

Observe that any SPD kernel will avoid the obvious issues described before. In fact, if $k(x, x') = \phi(x)^\top \phi(x')$ then one can check that k is a SPD kernel. What is interesting is that the reverse is also true – for any SPD kernel k, there exists a Hilbert space (called *feature space*) $\mathbb{H}$ and a mapping $\phi : \mathbb{R}^d \to \mathbb{H}$ such that $k(x, x') = \langle \phi(x), \phi(x') \rangle$, where $\langle z, z' \rangle$ denotes the inner product in $\mathbb{H}$. In the second case in Example 2.15, we have $\mathbb{H} = \mathbb{R}^6$ and $\langle z, z' \rangle = z^\top z'$ is just the usual dot product. However, in general $\mathbb{H}$ may be infinite-dimensional. This existence result follows from Mercer's theorem [Mer09] on representations of symmetric positive definite functions. In fact, SPD kernels give rise to a nice theory on reproducing kernel Hilbert spaces (RKHS) via the Moore-Aronszajin theorem [Aro50]. These discussions are beyond the scope of these notes.

We now give some simple examples of SPD Kernels.

Example 2.17: SPD Kernels

1. Linear kernel: $k(x, x') = x^\top x'$.
2. Polynomial kernel: $k(x, x') = (1 + x^\top x')^m$, $m > 0$.
3. Gaussian (radial basis function) kernel: $k(x, x') = \exp\left(-\frac{\|x - x'\|^2}{2s^2}\right)$, $s > 0$
4. Laplacian kernel: $k(x, x') = \exp\left(-\lambda \|x - y\|\right)$, $\lambda > 0$.
5. ($\star$) We present a more interesting example. Let Ω be a finite set and consider the collection of subsets of Ω. For any $A, A' \subset \Omega$, define the kernel $k(A, A') = 2^{|A \cap A'|}$, where $\cap$ denotes intersection and $|A|$ denotes the number of elements in A. Then, $k : \mathcal{P}(\Omega) \times \mathcal{P}(\Omega) \to \mathbb{R}$ is a SPD kernel. This shows that kernels, and generally kernel methods can be applied to input spaces other than $\mathbb{R}^d$.

If this is the first time you have seen the kernels in Example 2.17, you may wonder how to show that they in fact satisfy the condition in Definition 2.16. Indeed, it is not immediately obvious if any Gram matrix formed by these kernels must be symmetric positive semi-definite. The following proposition is useful in verifying SPD kernels, and in fact is even more useful in building new kernels from known ones. Such techniques are very frequently employed in mathematics.

Proposition 2.18: Closure Properties of SPD Kernels

Suppose $k_1, k_2, \ldots$ is a collection of SPD kernels on $\mathbb{R}^d \times \mathbb{R}^d$. Then, the following are also SPD kernels

1. (Scaling) $k(x, x') = \lambda k_1(x, x')$ for any $\lambda > 0$
2. (Addition) $k(x, x') = k_1(x, x') + k_2(x, x')$
3. (Normalization) $k(x, x') = g(x)k_1(x, x')g(x')$ for any real-valued function g.
4. (Limit) $k(x, x') = \lim_{n\to\infty} k_n(x, x')$, provided the limit exists for all x, x'.
5. (Product) $k(x, x') = k_1(x, x')k_2(x, x')$.

Proof 2.18: Closure Properties of SPD Kernels

In each case, the symmetric condition is clear, so we will only prove the positive semi-definiteness of the Gram matrices. Let $\{x_1, \ldots, x_n\}$ be an arbitrary collection of points in $\mathbb{R}^d$ and let G_m be the Gram matrix associated with the kernel k_m, i.e. $G_{m,ij} = k_m(x_i, x_j)$. Similarly, we set G and G' to be the Gram matrix associated with k and k' respectively. Let c be an arbitrary vector in $\mathbb{R}^n$.

1. $c^\top G c = c^\top(\lambda G_1)c = \lambda(c^\top G_1 c) \geq 0$.
2. $c^\top G c = c^\top(G_1 + G_2)c = c^\top G_1 c + c^\top G_2 c \geq 0$.
3. Define the vector b to have components $b_i = c_i g(x_i)$. Then, $c^\top G c = b^\top G_1 b \geq 0$.
4. By assumption $c^\top G_m c \geq 0$ for all m and $\lim_{m\to\infty} G_m = G$, and thus $c^\top G c = c^\top(\lim_{m\to\infty} G_m)c = \lim_{m\to\infty} c^\top G_m c \geq 0$.
5. Let $X_1 \sim \mathcal{N}(0, G_1)$ and $X_2 \sim \mathcal{N}(0, G_2)$ be independent random Gaussian vectors in $\mathbb{R}^n$. Set $Y = X_1 \circ X_2$ be their Hadamard product, i.e. $Y_i = X_{1,i}X_{2,i}$ for each coordinate i. Then,

$$\begin{aligned} \mathrm{Cov}(Y_i, Y_j) &= \mathbb{E}Y_i Y_j = \mathbb{E}X_{1,i}X_{2,i}X_{1,j}X_{2,j} \\ &= \mathbb{E}X_{1,i}X_{1,j}\mathbb{E}X_{2,i}X_{2,j} \qquad \text{(By independence)} \\ &= \mathrm{Cov}(X_{1,i}, X_{1,j})\mathrm{Cov}(X_{2,i}, X_{2,j}) \\ &= G_{1,ij}G_{2,ij} \\ &= G_{ij}. \end{aligned} \tag{2.50}$$

   Since G is the covariance matrix of Y, it must be positive semi-definite.

□

Using Proposition 2.18, we can show easily that the polynomial kernel, for example, is SPD. First, notice that the linear kernel is SPD. We can show this directly by $c^\top G c = \sum_{i,j} c_i c_j G_{ij} = \sum_{i,j} c_i c_j x_i^\top x_j = \sum_{i,j,m} c_i c_j x_{i,m} x_{j,m} = \sum_m (\sum_i c_i x_{i,m})^2 \geq 0$. Moreover, the constant kernel $k(x, x') = 1$ is trivially SPD. Hence, using property 2 of Proposition 2.18, the kernel $k(x, x') = 1 + x^\top x'$ is SPD. Finally, using property 5 of Proposition 2.18 consecutively for $m - 1$ times, we get $k(x, x') = (1 + x^\top x')^m = (1 + x^\top x') \times \cdots \times (1 + x^\top x')$ is SPD.

Exercise 2.19

Show that the Gaussian or radial basis function (RBF) kernel defined in Example 2.17 is a SPD kernel. (Hint: Observe that $\|x - x'\|^2 = \|x\|^2 - 2x^\top x' + \|x'\|^2$. Now set $g(x) = \exp(-\|x\|^2/2s^2)$ and use property 3 in Proposition 2.18. Finally use the other properties via a Taylor expansion of the exponential function to deduce the result.)

Now, to highlight the fact using kernels is much more flexible than using explicit feature maps, we will derive a explicit feature map corresponding to the Gaussian (RBF) kernel in one dimension ($d = 1$). Recall that in this case, the Gaussian kernel is given by

$$\begin{aligned} k(x, x') &= \exp\left(-\frac{1}{2s^2}(x - x')^2\right) \\ &= \exp\left(-\frac{1}{2s^2}x^2\right)\exp\left(\frac{1}{s^2}xx'\right)\exp\left(-\frac{1}{2s^2}x'^2\right) \\ &= \exp\left(-\frac{1}{2s^2}x^2\right)\left(\sum_{m=0}^{\infty}\frac{(xx')^m}{s^{2m}m!}\right)\exp\left(-\frac{1}{2s^2}x'^2\right) \\ &= \exp\left(-\frac{1}{2s^2}x^2\right)\left(1 + \sqrt{\frac{1}{s^2 1!}}x \cdot \sqrt{\frac{1}{s^2 1!}}x' + \sqrt{\frac{1}{s^4 2!}}x^2 \cdot \sqrt{\frac{1}{s^4 2!}}x'^2 + \dots\right)\exp\left(-\frac{1}{2s^2}x'^2\right) \\ &= \phi(x)^\top \phi(x), \end{aligned} \tag{2.51}$$

where

$$\phi(x) = \exp\left(-\frac{1}{2s^2}x^2\right)\left(1, \sqrt{\frac{1}{s^2 1!}}x, \sqrt{\frac{1}{s^4 2!}}x^2, \dots\right) \tag{2.52}$$

Formula (2.52) gives an explicit feature map which corresponds to the Gaussian kernel in 1D. Notice now that $\phi(x)$ is an infinite dimensional feature map! In other words, computing the explicit features $x \mapsto \phi(x)$ is impossible as there are infinite number of terms, but the kernel can be computed easily. This is an instance of the well-known *kernel trick* that is ubiquitous in kernel methods.

2.3.3 Kernel Ridge Regression

Having introduced some basic properties of kernels, let us return to the regularized regression problem motivated in Section 2.3.1. Using formula (2.44), we see that for each SPD kernel k, we have a corresponding regularized least squares formula corresponding to a general linear basis model generated by a feature map corresponding to k. More concretely, given any SPD kernel

k, the hypothesis space we are considering is

$$\mathcal{H}(k) = \left\{ f : f(x) = \sum_{j=0}^{\infty} w_j \phi_j(x), \quad w_j \in \mathbb{R} \text{ with } k(x, x') = \sum_{j=0}^{\infty} \phi_j(x)\phi_j(x') \right\}. \tag{2.53}$$

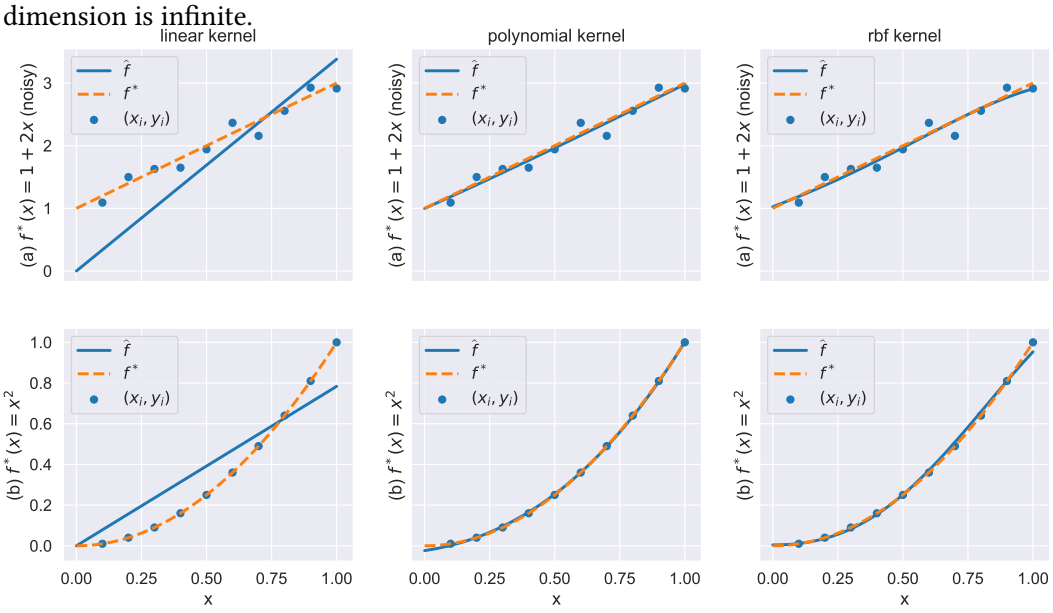
In this space, the regularized empirical risk minimization has the solution

$$\widehat{f}(x) = \sum_{i=1}^{N} [(G + \lambda I_N)^{-1} y]_i k(x, x_i), \qquad G_{ij} = k(x_i, x_j), \tag{2.54}$$

which can be computed without explicit knowledge of $\{\phi_j\}$. This is known as *kernel ridge regression*, and is one of the most basic applications of kernel methods and in particular, the kernel trick.

Example 2.20: Kernel Ridge Regression

We apply kernel ridge regression to the examples considered in Example 2.4 with a small regularization parameter. The results are shown below. We observe that polynomial and Gaussian/RBF kernels work reasonably well. Note that in the RBF case, the feature space dimension is infinite.



2.3.4 Further Reading

We have only scratched the surface of kernel methods. Although in the next section we will introduce a well-known application of kernel methods, namely the support vector machine, we have invariably missed a large chunk of interesting theory and applications of the kernel trick. For example, Gaussian processes [Ras03] makes great use of kernels. There are also applications

of kernels to unsupervised learning, e.g. kernel PCA [SSM98] and kernel mean embedding of distributions [MFSS17], just to name a few. For comprehensive exposition on kernel methods, the reader is referred to [SC04, HSS08].

2.4 Support Vector Machines

We saw in kernel ridge regression that although the kernel trick allows us to implicitly handle large or infinite feature space dimensions, in computing the Gram matrix we have to evaluate $k(x_i, x_j)$ for all pairs of data points. If we have N data points, there amounts to N^2 number of operations for each inference. This becomes very expensive when N is large. It is then natural to ask if there are kernel methods that scale better with large datasets. This is the study of *sparse kernel methods*, a prime example of which is the kernel support vector machine (kernel SVM). To begin with, we first introduce the SVM in the linear setting, without reference to kernels.

2.4.1 Linear Support Vector Machines

Consider a binary classification dataset $\mathcal{D} = \{x_i, y_i\}_{i=1}^N$ where each $x_i \in \mathbb{R}^d$ and each $y_i = 1$ (class +) or $y_i = -1$ (class −). An important concept concerning binary classification is that of *linear separability*. Let us define and illustrate it below.

Definition 2.21: Linear Separability

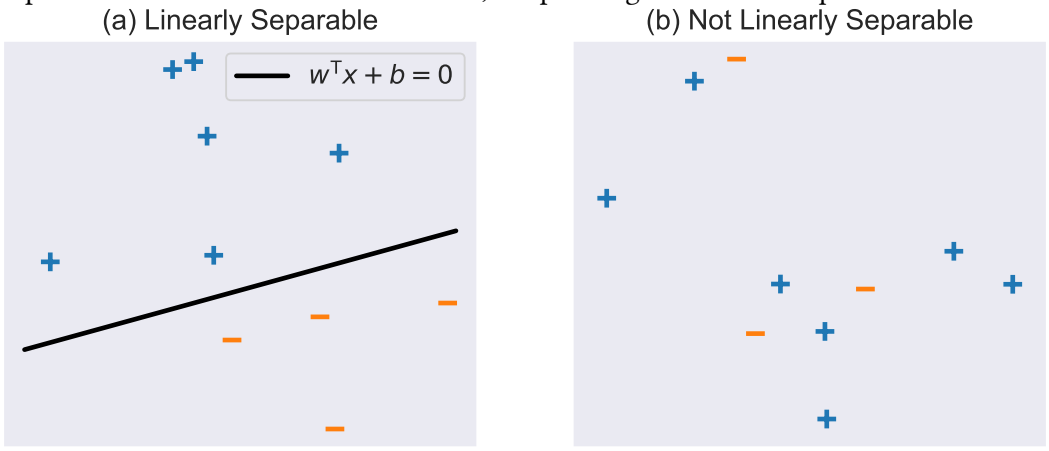
We say that $\mathcal{D}$ is *linearly separable* if there exists $w \in \mathbb{R}^d$ and $b \in \mathbb{R}$ such that

$$w^\top x_i + b > 0 \text{ if } y_i = +1 \text{ and } w^\top x_i + b < 0 \text{ if } y_i = -1, \tag{2.55}$$

for each $i = 1, \ldots, N$. We can rewrite the above compactly as the following condition

$$y_i(w^\top x_i + b) > 0. \tag{2.56}$$

The following figure illustrates a linearly separable situation versus one that is not linearly separable. Notice that in the former case, a separating line is not unique.



Margin and Maximum Margin Solution. Let us hereafter assume that our dataset $\mathcal{D}$ is linearly separable. As seen previously, the separating line (or hyperplane in higher dimensions) need not be unique. Hence, we want to find one that has desirable properties, such as having good generalization. Support vector machines approach this problem via introducing the concept of *margin*.

Let $w \in \mathbb{R}^d$ and $b \in \mathbb{R}$ define a separating hyperplane

$$\{x \in \mathbb{R}^d : w^\top x + b = 0\}. \tag{2.57}$$

This is also called the decision boundary of the associated decision function (approximation of the oracle)

$$f(x) = \begin{cases} +1 & w^\top x + b > 0, \\ -1 & w^\top x + b < 0. \end{cases} \tag{2.58}$$

We can write this compactly as

$$f(x) = \text{Sign}(w^\top x + b), \tag{2.59}$$

where $\mathrm{Sign}(z) = +1$ if $z > 0$ and -1 if $z < 0$. The margin of the decision function is the minimum distance of points in $\mathcal{D}$ to the separating hyperplane $w^\top x + b = 0$. In precise terms, the margin is given by

$$\min_{i=1,\ldots,N} \frac{|w^\top x_i + b|}{\|w\|} \tag{2.60}$$

It turns out that to achieve the best generalization we should look for *maximum margin solutions*. This can be formalized in statistical learning theory, but is also intuitively clear: the larger the margin, the more sampling noise we can accommodate. The maximum margin solution can be obtained by the following optimization problem

$$\max_{w\in\mathbb{R}^d, b\in\mathbb{R}} \left\{ \frac{1}{\|w\|} \min_{i=1,\ldots,N} |w^\top x_i + b| \right\} \text{ subject to } y_i(w^\top x_i + b) > 0 \text{ for all } i = 1,\ldots,N. \tag{2.61}$$

At the moment, (2.61) does not seem easy to handle. Now, we will show via a sequence of transformations that this problem is in fact a standard constrained convex optimization problem.

First, notice that $|w^\top x_i + b| = y_i(w^\top x_i + b)$. Moreover, the margin distance $y_i(w^\top x_i + b)/\|w\|$ is invariant with respect to the transformation $w \mapsto \kappa w$, $b \mapsto \kappa b$ for any $\kappa > 0$. Thus, without loss of generality we can assume $y_i(w^\top x_i + b) \geq 1$ for all i by taking $\kappa^{-1} = \min_i y_i(w^\top x_i + b)$. Then, $\min_i |w^\top x_i + b| = \min_i y_i(w^\top x_i + b) = 1$ and for the closest points x_j to the decision boundary, we have $y_j(w^\top x_j + b) = 1$. Note that there is at least one such point, but there may be many. Consequently, the optimization problem reduces to maximizing $1/\|w\|$, which is equivalent to minimizing $\|w\|^2/2$ (factor of $1/2$ is for convenience). To summarize, an equivalent formulation of (2.61) is

$$\min_{w\in\mathbb{R}^d, b\in\mathbb{R}} \frac{1}{2}\|w\|^2 \text{ subject to } y_i(w^\top x_i + b) \geq 1 \text{ for all } i = 1,\ldots,N. \tag{2.62}$$

The optimization problem (2.62) is a constrained optimization problem and an effective way to analyze it is via the method of *Lagrange multipliers*. Let us briefly review it in an informal way.

Method of Lagrange Multipliers. Minimizing a differentiable function $F : \mathbb{R}^m \to \mathbb{R}$ is often done by setting its derivative ∇F to 0. This obtains a stationary point, which one can then check if it is indeed a minimum. To achieve this, we can either use higher order derivatives (e.g. the Hessian), or deduce minimality from the properties of the function F. For example, if F is convex, then any stationary point is a (global) minimum.

Let us now consider the constrained optimization problem

$$\min_{z\in\mathbb{R}^m} F(z) \text{ subject to } G(z) = 0, \tag{2.63}$$

for some differentiable constraint function $G : \mathbb{R}^m \to \mathbb{R}$. How do we solve this problem? The crucial observation is that at an optimal point $\widehat{z}$, ∇F and ∇G must be parallel, otherwise we can

move along the curve $G(z) = 0$ (which is locally perpendicular to ∇G) to decrease the function value of F. This can be expressed as the condition

$$\nabla F(\widehat{z}) + \mu \nabla G(\widehat{z}) = 0. \tag{2.64}$$

for some $\mu \neq 0$. Together with the condition $G(\widehat{z}) = 0$, we see that $\widehat{z}$ must be a stationary point of

$$\mathcal{L}(z, \mu) = F(z) + \mu G(z). \tag{2.65}$$

The function $\mathcal{L} : \mathbb{R}^m \times \mathbb{R} \to \mathbb{R}$ is called the *Lagrangian* and μ is called a *Lagrange multiplier*.

What if we replace the equality constraint $G(z) = 0$ by an inequality constraint $G(z) \leq 0$? There are two cases: if $\widehat{z}$ lies in the interior of the constraint set, i.e. if $G(\widehat{z}) < 0$ then the constraint is *inactive*, and so the condition is simply $\nabla F(\widehat{z}) = 0$. On the other hand, if $\widehat{z}$ lies on the boundary of the constraint set, i.e. if $G(\widehat{z}) = 0$, then we are in the equality constraint case considered previously. The only caveat is that in the inequality case, we need ∇F and ∇G to point in different directions (See Figure 2.2), otherwise we can move into the interior of the constraint set and decrease F in the process, contradicting the optimality of $\widehat{z}$. Thus, we should have $\mu \geq 0$ and $\mu g(\widehat{z}) = 0$.

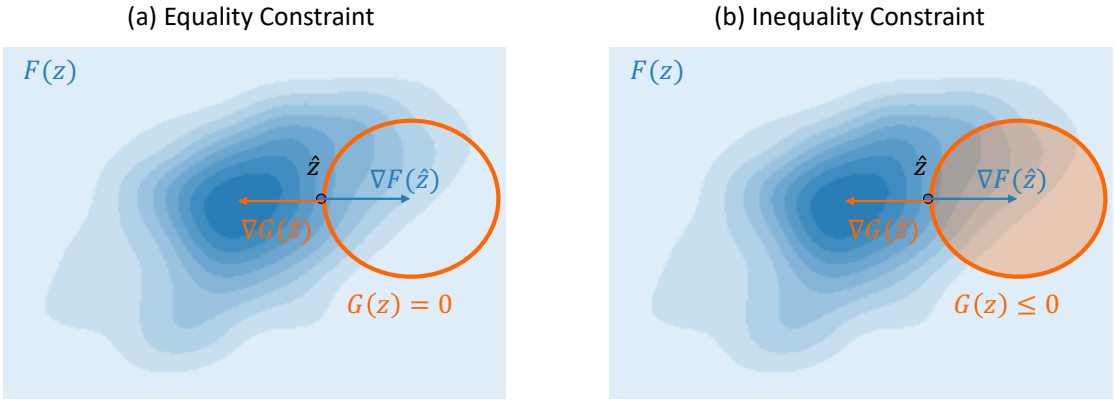


Figure 2.2: Illustration of the idea behind Lagrange multipliers.

In general, we can have many equality constraints and inequality constraints. Observe that each equality constraint $G_j(x) = 0$ can be written as two inequality constraints $G_j(x) \leq 0$ and $-G_j(x) \leq 0$, thus it suffices to consider only systems of inequality constraints. Therefore, we have the following constrained optimization problem

$$\begin{aligned} &\min_{z \in \mathbb{R}^m} F(z) \text{ subject to} \\ &G_j(z) \leq 0, j = 1, \ldots, n, \end{aligned} \tag{2.66}$$

Following the approach previously, we introduce a vector of Lagrange multipliers $\mu = (\mu_1, \ldots, \mu_n)$, with the Lagrangian

$$\mathcal{L}(z, \mu) = F(z) + \sum_{j=1}^{n} \mu_j G_j(z). \tag{2.67}$$

Then, following the approach previously for the single-constraint case, we can derive the following necessary conditions. These conditions are known as the Karush-Kuhn-Tucker (KKT) conditions [KT51]. They can be proven to be necessary for optimality under some technical assumptions known as constraints qualifications.

Theorem 2.22: KKT Conditions (Informal)

Under technical assumptions, for each solution $\widehat{z}$ of (2.66), there exists $\widehat{\mu} \in \mathbb{R}^n$ for which the following conditions hold:

$$\begin{aligned} \text{(Stationarity)} \quad & \nabla_z \mathcal{L}(\widehat{z}, \widehat{\mu}) = 0 \\ \text{(Primal Feasibility)} \quad & G_j(\widehat{z}) \leq 0, \quad j = 1, \dots, n \\ \text{(Dual Feasibility)} \quad & \widehat{\mu}_j \geq 0, \quad j = 1, \dots, n \\ \text{(Complementary Slackness)} \quad & \widehat{\mu}_j G_j(\widehat{z}) = 0, \quad j = 1, \dots, n \end{aligned} \tag{2.68}$$

Moreover, these conditions are also sufficient for optimality if (2.66) is further assumed to be convex.

Finally, we may consider the *dual* of the problem (2.66) as

$$\begin{aligned} &\max_{\mu \in \mathbb{R}^n} \tilde{F}(\mu), \qquad \text{where } \tilde{F}(\mu) = \inf_{z \in \mathbb{R}^m} \mathcal{L}(z, \mu) \\ &\text{Subject to } \mu \geq 0. \end{aligned} \tag{2.69}$$

Under convexity assumption of (2.66) and constraint qualification, *strong duality* holds in the sense that the dual and primal problem have same optimal objective values. A solution $\widehat{\mu}$ of the KKT conditions (Theorem 2.22) solves the dual problem (2.69). For more details on constrained optimization and the KKT conditions, the reader is referred to [KT51, NWNW06] and also Appendix B.3 of [MRT18].

KKT Conditions for Linear SVM Let us now return the constrained optimization problem (2.62) for the maximum margin solution of linear SVM. Applying the KKT conditions (Theorem 2.22) with $z = (w, b)$ and $G_j(z) = 1 - y_j(w^\top x_j + b)$, we obtain the Lagrangian

$$\mathcal{L}(w, b, \mu) = \frac{1}{2}\|w\|^2 + \sum_{j=1}^{N} \mu_j (1 - y_j(w^\top x_j + b)). \tag{2.70}$$

and stationary conditions

$$\widehat{w} = \sum_{j=1}^{N} \widehat{\mu}_j y_j x_j, \qquad 0 = \sum_{j=1}^{N} \widehat{\mu}_j y_j. \tag{2.71}$$

Furthermore, the complementary slackness condition implies

$$0 = \widehat{\mu}_j (1 - y_j(\widehat{w}^\top x_j + \widehat{b})) \tag{2.72}$$

Now comes some crucial observations. First, suppose for the moment that $\widehat{\mu}$ has been found. Then, our maximal margin decision function (i.e. best approximator of the oracle classifier) is

$$\widehat{f}(x) = \text{Sign}\left(\sum_{j=1}^{N} \widehat{\mu}_j y_j x_j^\top x + \widehat{b}\right). \tag{2.73}$$

In the above sum, only data points (x_j, y_j) for which $\widehat{\mu}_j \neq 0$ contributes. Owing to the complementary slackness condition (2.72), these are precisely the vectors satisfying $y_j(\widehat{w}^\top x_j + \widehat{b}) = 1$, i.e. those that are closest to the maximum margin decision boundary $\widehat{w}^\top x_j + \widehat{b} = 0$. These are called *support vectors*. In this sense, the prediction step in linear SVM is sparse in the data points, unlike kernel ridge regression.

Dual Problem for Linear SVM. We can go on to derive the dual problem of (2.62), which if solved gives us $\widehat{\mu}$. Following (2.69), we obtain the dual problem

$$\max_{\mu \in \mathbb{R}^N} \sum_{j=1}^{N} \mu_j - \frac{1}{2} \sum_{i,j}^{N} \mu_i \mu_j y_i y_j x_i^\top x_j \tag{2.74}$$

$$\text{Subject to: } \mu_j \geq 0, j = 1, \ldots, N \qquad \sum_{j=1}^{N} \mu_j y_j = 0. \tag{2.75}$$

Once solved, we can plug into expression (2.71) to find $\widehat{w}$. The constant $\widehat{b}$ can be found by solving $1 = \widehat{w}^\top x_j + \widehat{b}$ for any support vector x_j. Consequently, this fixes our learned classifier (2.73) with which we can make predictions.

2.4.2 Kernel Support Vector Machines

As in Section 2.3.3, the advantage of the dual formulation is that now, all that is required to predict a new data point is the evaluation of $x^\top x'$ for any inputs x, x'. We can then make the SVM nonlinear by replacing x by feature maps $\phi(x)$. Finally, using the kernel trick we can replace the inner product $\phi(x)^\top \phi(x')$ by kernels $k(x, x')$. The resulting kernel SVM dual problem reads

$$\max_{\mu \in \mathbb{R}^N} \sum_{j=1}^{N} \mu_j - \frac{1}{2} \sum_{i,j}^{N} \mu_i \mu_j y_i y_j k(x_i, x_j) \tag{2.76}$$

$$\text{Subject to: } \mu_j \geq 0, j = 1, \ldots, N \qquad \sum_{j=1}^{N} \mu_j y_j = 0. \tag{2.77}$$

Once solved to obtain $\widehat{\mu}$, we would have the predictive function

$$\widehat{f}(x) = \text{Sign}\left(\sum_{j=1}^{N} \widehat{\mu}_j y_j k(x_j, x) + \widehat{b}\right). \tag{2.78}$$

$$\text{where } \widehat{b} = 1 - \sum_{j=1}^{N} \widehat{\mu}_j y_j k(x_j, x_i) \text{ for some support vector } x_i. \tag{2.79}$$

2.4.3 Further Reading

In the previous sections we introduced the most basic SVM setting. There are a number of extensions. The most important one has to do with the fact that we assumed separability. Although it is plausible that a feature map induced by a kernel choice may make our data linearly separable in feature space, it is useful to derive a theory without such assumptions. It turns out that this is quite simple, as we just consider a relaxed version of (2.62) with additional slack variables $\xi \in \mathbb{R}^N$,

$$\min_{w \in \mathbb{R}^d, \xi \in \mathbb{R}^N, b \in \mathbb{R}} \frac{1}{2}\|w\|^2 + C \sum_{i=1}^{N} \xi_i^p \tag{2.80}$$

$$\text{subject to: } y_i(w^\top \phi(x_i) + b) \geq 1 - \xi_i \text{ and } \xi_i \geq 0 \text{ for all } i = 1, \ldots, N. \tag{2.81}$$

Here $p \geq 1$ ensures convexity and all the theory before can be analogously extended to this case. This also immediately allows for us to remove the constraint by observing that the solution must be obtained when equality $y_i(w^\top \phi(x_i) + b) = 1 - \xi_i$ holds. Therefore, we have the equivalent, but unconstrained version of (2.80)

$$\min_{w \in \mathbb{R}^d, \xi \in \mathbb{R}^N, b \in \mathbb{R}} \frac{1}{2}\|w\|^2 + C \sum_{i=1}^{N} \max(0, 1 - y_i(w^\top \phi(x_i) + b))^p. \tag{2.82}$$

The unconstrained formulation (2.82) may be useful for direct optimization of the primal SVM problem. Other extensions includes those to multi-class classification, regression and Bayesian inference in the form of relevance vector machines, see [MRT18, BO06] and also in-depth discussions on SVM and kernel methods in [CV95, SC04, CS00, HSS08].

2.5 Decision Trees

So far we have looked at linear or linear basis models and their kernel variants. In this section, we are going to consider a different sort of classification or regression method, in which the approximators are piece-wise constant functions. The simplest of such methods is *decision trees*, in which input space is *stratified* or *partitioned* into simple regular regions and a constant prediction is assigned to each region.

Decision trees are very natural models of decision making, and consists a class of easily interpretable machine learning models, in the sense that we can readily deduce how the model arrives at the prediction. Regression and classification models using decision trees are called *CART*, which stands for *classification and regression trees*.

2.5.1 Decision Trees for Regression

Mathematically, a (directed) tree is a *directed acyclic graph* (See Figure 2.3). The base vertex of the tree is called the *root node*. The terminal vertices are called *terminal nodes*. All other vertices are called *internal nodes*. The edges that connects the nodes are called *branches*.

Based on a decision tree, how can we build a model capable of making predictions? Let us first discuss this in terms of regression in one dimension.

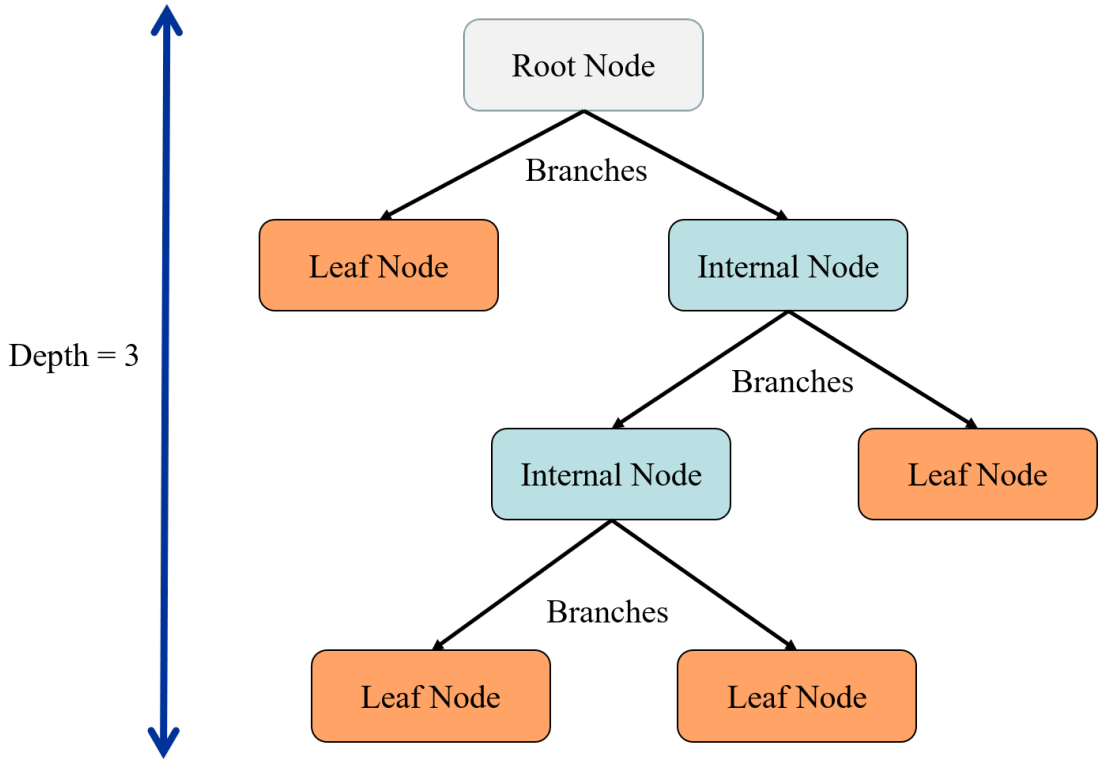


Figure 2.3: The structure of a decision tree.

Suppose we want to approximate some oracle function $f^* : [0, 1] \to \mathbb{R}$. The simplest decision

tree is formed by selecting some $\theta_0 \in (0, 1)$ and defining the piece-wise constant function

$$f(x) = \begin{cases} a & x > \theta_0 \\ b & x \leq \theta_0 \end{cases}. \tag{2.83}$$

What values should we pick for a and b? One simple proposal is to pick them to be the average of f^* in these regions, i.e.

$$a = \frac{1}{\theta_0} \int_0^{\theta_0} f^*(z)dz, \qquad b = \frac{1}{1-\theta_0} \int_{\theta_0}^{1} f^*(z)dz. \tag{2.84}$$

This is a binary decision tree of depth 1, since there are only two nodes connected to the root node. We can also form deeper decision trees by further splits. See Figure 2.4.

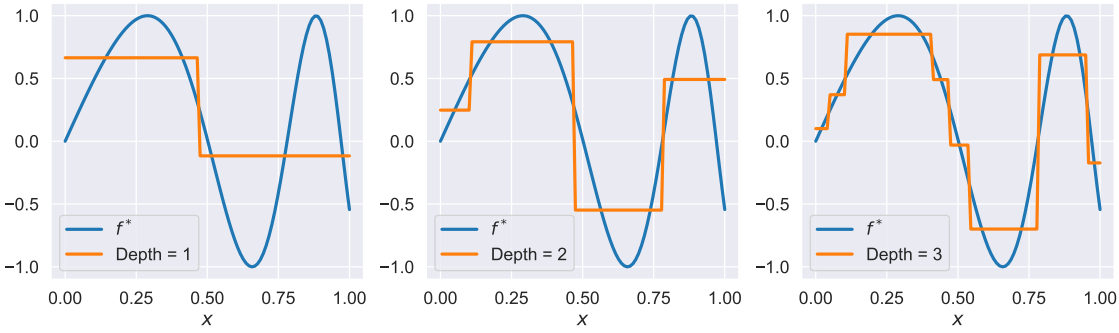


Figure 2.4: Decision trees of different depths for regression.

More generally, a decision tree based regressor is formed by dividing the input domain $\mathcal{X}$ into J distinct and non-overlapping regions $\mathcal{R}_1, \mathcal{R}_2, \ldots, \mathcal{R}_J$ with $\cup_{j=1}^{N} \mathcal{R}_j = \mathcal{X}$. We then assign a constant value a_j for all input points in $\mathcal{R}_j$. More precisely, a decision tree regression hypothesis space consists of functions of the form

$$f(x) = \sum_{j=1}^{J} a_j \mathbb{1}_{\mathcal{R}_j}(x), \qquad \{\mathcal{R}_j\} \text{ a partition of } \mathcal{X}. \tag{2.85}$$

The regions $\mathcal{R}_j$ can theoretically be of any shape, but often we want to keep the results interpretable, and hence we restrict them to high dimensional rectangles (Figure 2.5).

Optimization How do we optimize in this hypothesis space of regression trees? Given a dataset $\mathcal{D} = \{x_i, y_i\}_{i=1}^{N}$, we can simply set a_j to be the average of y_i's for which $x_i \in \mathcal{R}_j$, i.e.

$$a_j = \overline{y}_j = \frac{\sum_i y_i \mathbb{1}_{\mathcal{R}_j}(x_i)}{\sum_i \mathbb{1}_{\mathcal{R}_j}(x_i)} \tag{2.86}$$

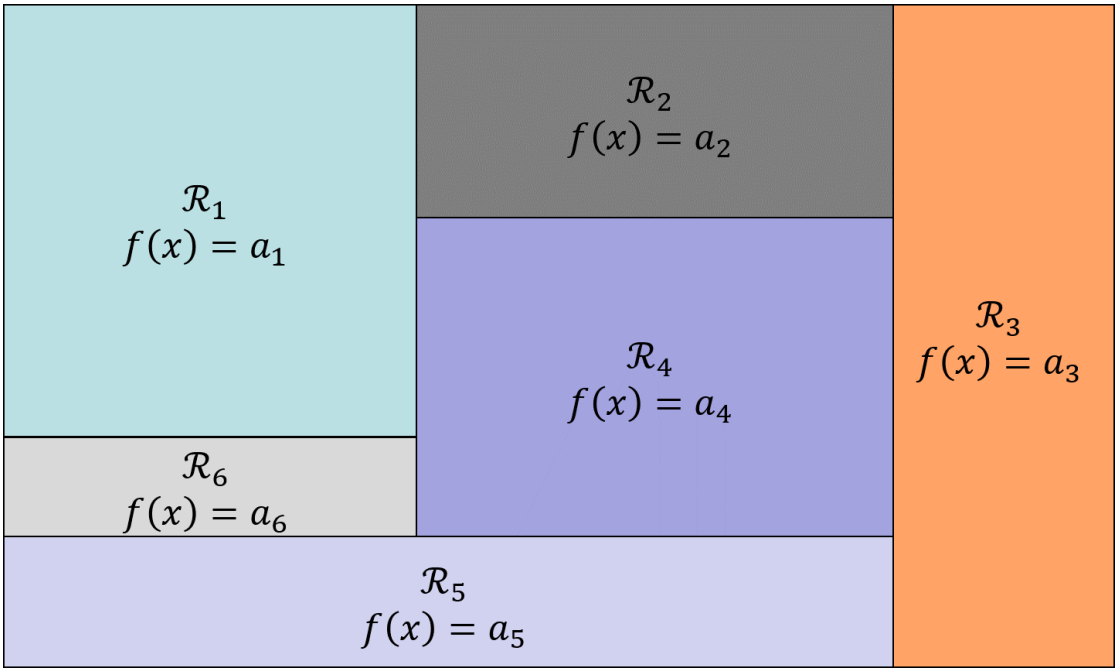


Figure 2.5: Piecewise constant function with rectangular regions.

Then, it remains to determine the regions $\{\mathcal{R}_j\}$. Hence, the empirical risk minimization with square loss involves the following optimization problem

$$\min_{\{\mathcal{R}_j\}} \frac{1}{2} \sum_{i=1}^{N} \left(\sum_{j=1}^{J} \overline{y}_j \mathbb{1}_{\mathcal{R}_j}(x_i) - y_i \right)^2 . \tag{2.87}$$

In other words, we are minimizing over all possible rectangular partitions $\{\mathcal{R}_j\}$ of the input space $\mathcal{X}$. It turns out that this problem is very difficult to solve exactly. In the language of complexity theory, it is NP-hard.

Instead of solving (2.87) exactly, we can proceed in a *greedy* manner. This approach is called *recursive binary splitting*. We start with the root node and successively split the input space in one of its dimensions into two parts. This corresponds to adding two leaf nodes into the tree by attaching them to a chosen leaf node, which now becomes an internal node. The split is chosen so that the loss function (e.g. square loss) is minimized at the current step, regardless if future splits may become suboptimal. This is why it is called a greedy method, since it only cares about minimizing the present error and not future ones. In Figure 2.6, we illustrate using a one dimensional example where greedy method gives a sub-optimal solution. A common stopping criterion is when each split region contains only a few number of sample points.

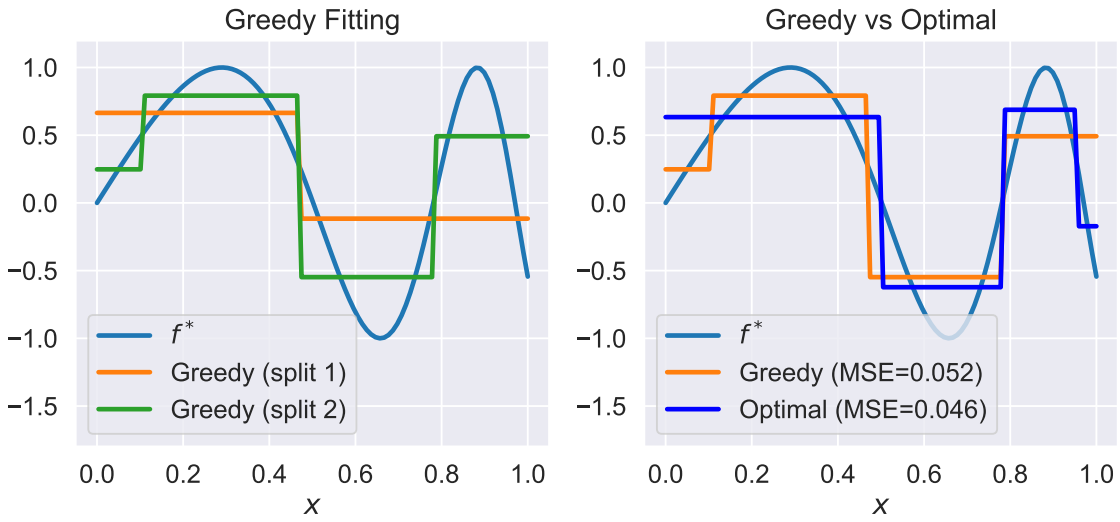


Figure 2.6: Comparison of greedy solution vs optimal solution.

2.5.2 Decision Trees for Classification

Classification problems can be similarly handled by decision trees. In this case, we can still consider the hypothesis space (2.85), except that we have to pick the values $\{a_j\}$ differently during optimization. Instead of using the average as in (2.86), we usually take a_j to be the majority vote: suppose we have a K-class classification problem where the labels can take values in $\{c_1, c_2, \ldots, c_K\}$, then we can set $a_j = \bar{c}_j$ where $\bar{c}_j$ is the most frequently occurring class for inputs in $\mathcal{R}_j$. Ties can be broken randomly.

To carry out the recursive binary splitting algorithm as before, we need a performance metric for classification so that a greedy selection of the split can be chosen at every step. One natural choice is the classification error rate, but it is not very sensitive to splits and hence not suitable for building good decision trees. Instead, let us define p_{jk} to be the proportion of samples in $\mathcal{R}_j$ belonging to class k, then we can use the following loss functions, also known as measures of *impurity*, as basis for greedy splitting:

$$\textbf{Entropy:} \qquad \sum_{j=1}^{J} \sum_{k=1}^{K} -p_{jk} \log p_{jk}, \tag{2.88}$$

$$\textbf{Gini Index:} \qquad \sum_{j=1}^{J} \sum_{k=1}^{K} p_{jk}(1 - p_{jk}). \tag{2.89}$$

The recursive binary splitting algorithm can be carried out analogously by minimizing the losses/impurities at each split.

2.5.3 Advantages and Disadvantages of Decision Trees

Decision trees are quite different from other types of models we have seen before. So, what are some advantages and disadvantages of using decision trees? Below we give a non-exhaustive list.

Advantages:

- Can readily visualize and understand predictions
- Implicit feature selection via analyzing contribution of splits to reduction of error/impurity
- Robust to data types, supervised learning tasks and nonlinear relationships

Disadvantages:

- Prone to overfitting
- Sensitive to data variation and balancing
- Greedy algorithms may find sub-optimal solutions to empirical risk minimization

2.5.4 Further Reading

The main issue that we have not discussed here is that of pruning. It is found that greedy methods typically obtain trees of poor generalization properties. To overcome this, we usually devise a pruning procedure: we first grow (using greedy methods) a large tree and then prune it to obtain a smaller sub-tree that has better generalization properties. Such pruning procedures usually involves minimizing a loss function (e.g. (2.88) or (2.89)) plus a term that penalizes the complexity of a tree, e.g. number of leaf nodes, or other complexity measures motivated by learning theory. For details, see [BO06, MRT18].

2.6 Model Ensembling

The most glaring drawback of decision trees is that they are prone to overfitting: if the depth is low, the approximation power is low, but when the depth is large, there often overfit the data. Figure 2.7 gives an illustration of this. However, there is a class of methods that greatly overcomes this drawback and they enable significant broader application of decision trees. These are known as *ensembling methods*, where we combine a collection of *weak learners* (models from a small hypothesis space) to form a *strong learner* that has good approximation and generalization properties. The role of weak learners are usually played by decision trees. Nevertheless, we can discuss these methods with respect to general weak learners. We will introduce two different methods to combine models, namely *bagging* and *boosting*.

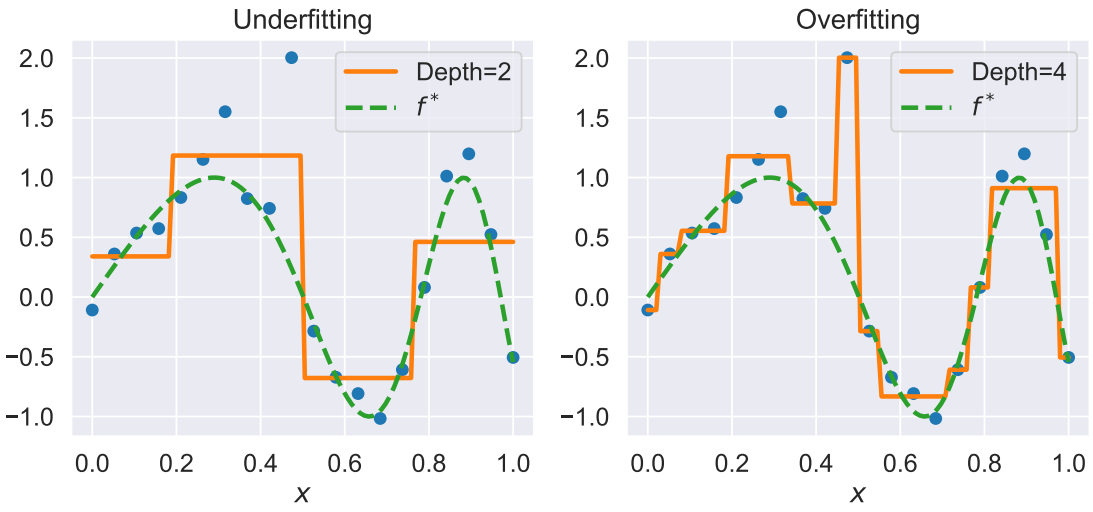


Figure 2.7: Overfitting phenomenon of decision trees. Shallow trees have limited expressive power and deep trees tend to overfit.

2.6.1 Bagging

The first method we discuss for combining models is also perhaps the most obvious: we aggregate the predictions due to a collection of models trained on different sub-samples of the training set in the obvious way – For regression problems, we simply take the their average prediction; For classification problems, we take their modal prediction via a majority vote. This is broadly known as the method of *bootstrap aggregating*, or *bagging*. The regression case is summarized in Algorithm 1. The classification case is similar.

Algorithm 1: Bootstrap Aggregating (Bagging)

Data: $\mathcal{D} = \{x_i, y_i\}_{i=1}^N$
for $j = 1, \ldots, m$ **do**
  Draw a random, independent size N' subset $\mathcal{D}_j$ from $\mathcal{D}$, with replacement;
  Obtain f_j from empirical risk minimization on $\mathcal{D}_j$;
end
return $\overline{f}(x) = \frac{1}{m} \sum_{j=1}^m f_j(x)$

What does bagging achieve? To see this, we can consider a simple model of bagging for regression. Assume that we have a collection of trained models $\{f_i\}_{i=1}^m$ from some fixed hypothesis space $\mathcal{H}$. We assume that each f_i is a random approximation of the oracle function f^*, in the sense that

$$f_i(x) = f^*(x) + \epsilon_i(x) \tag{2.90}$$

for some random function ϵ_i of mean 0 and variance σ^2. The aggregated prediction function is

$$\overline{f}(x) = \frac{1}{m}\sum_{i=1}^{m} f_i(x). \tag{2.91}$$

We will now show that the aggregated function reduces the mean squared error, under the assumption that for $i \neq j$, ϵ_i and ϵ_j are uncorrelated functions. To see this, notice that the mean squared error of the prediction of each f_i is given by

$$\begin{aligned} E(x) &= \mathbb{E}\frac{1}{m}\sum_{i=1}^{m}(f^*(x) - f_i(x))^2 \\ &= \mathbb{E}\frac{1}{m}\sum_{i=1}^{m}\epsilon_i(x)^2 \\ &= \frac{1}{m}\sum_{i=1}^{m}\mathbb{E}\epsilon_i(x)^2 \\ &= \sigma(x)^2. \end{aligned} \tag{2.92}$$

On the other hand, the squared error of the aggregated prediction is

$$\begin{aligned} \overline{E}(x) &= \mathbb{E}\left(f^*(x) - \overline{f}(x)\right)^2 \\ &= \mathbb{E}\left(f^*(x) - \frac{1}{m}\sum_{i=1}^{m} f_i(x)\right)^2 \\ &= \mathbb{E}\left(\frac{1}{m}\sum_{i=1}^{m}[f^*(x) - f_i(x)]\right)^2 \\ &= \mathbb{E}\left(\frac{1}{m}\sum_{i=1}^{m}\epsilon_i(x)\right)^2 \\ &= \mathbb{E}\frac{1}{m^2}\sum_{i=1}^{m}\epsilon_i(x)^2 \qquad \text{(uncorrelated)} \\ &= \frac{1}{m}\sigma(x)^2 = \frac{1}{m}E(x). \end{aligned} \tag{2.93}$$

What if there is a bias, such that each $\mathbb{E}\epsilon_i(x) = b(x) \neq 0$? Then, one can see that $\mathbb{E}\overline{f} = \mathbb{E}f_i = b$, i.e. the aggregate model does not change the bias. However, a similar calculation as the above will show that it reduces the variance by m times. More generally, this result can be deduced by the central limit theorem, or concentration/large-deviation estimates.

The apparent significant variance reduction is rarely observed in practice. This is because the errors ϵ_i are formed by sub-sampling of the training dataset: if a small number of samples are taken then the variance σ^2 is large; if a large number of samples are taken then different models become highly correlated. More importantly, bagging does not effectively reduce the bias of

the model. In other words, it does not improve the approximation power of the underlying hypothesis space. In the following, we will introduce another way of combining models that in fact increases the approximation capacity.

2.6.2 Boosting

Unlike bagging which combines models in parallel, *boosting* combines models in a sequential way. The most basic form of boosting is adaptive boosting, or *AdaBoost* [FS96]. The main idea in AdaBoost is to adaptively re-weight the training sample after training each weak learner so that the next weak learner can focus on correcting the mistakes made by the previous learners. Finally, the weak learners are combined together with appropriate weights to reduce the overall error. The Adaboost algorithm is given in Algorithm 2.

Algorithm 2: AdaBoost

Data: $\mathcal{D} = \{x_i, y_i\}_{i=1}^N$

Initialize: $w_j^{(1)} = 1/N$ for all $j = 1, \ldots, N$

for $j = 1, \ldots, m$ **do**

Obtain f_j from

$$f_j = \arg\min_{f \in \mathcal{H}} \sum_{i=1}^{N} w_i^{(j)} \mathbb{1}_{y_i \neq f(x_i)} \tag{2.94}$$

Compute combination coefficients:

$$\delta_j = \frac{\sum_{i=1}^N w_i^{(j)} \mathbb{1}_{y_i \neq f_j(x_i)}}{\sum_{i=1}^N w_i^{(j)}} \qquad \alpha_j = \log\left(\frac{1-\delta_j}{\delta_j}\right) \tag{2.95}$$

Update weights:

$$w_i^{(j+1)} = w_i^{(j)} \exp(\alpha_j \mathbb{1}_{y_i \neq f_j(x_i)}) \tag{2.96}$$

end

return $\overline{f}(x) = \text{Sign}\left(\sum_{j=1}^m \alpha_j f_j(x)\right)$

From Algorithm 2, we observe that f_1 is trained with uniform weights $w_i = 1/N$, i.e. in the usual way. Subsequently, the weights are increased for misclassified points and decreased for correctly classified points. Therefore, subsequent classifiers are encouraged to focus on these misclassified points. The weighting coefficients α_j gives greater weight to more accurate classifiers and hence improves overall classification accuracy.

Exponential Loss Interpretation The original AdaBoost algorithm is inspired by statistical learning, but Friedman et al [FHT00] showed that it can also be viewed as a sequential minimization of the exponential loss. Let us briefly account this interpretation below.

Define the exponential loss for binary outputs as

$$L(y', y) = e^{-yy'}. \tag{2.97}$$

Then, the empirical risk minimization problem is

$$R_{\text{emp}}(f) = \sum_{i=1}^{N} e^{-y_i f(x_i)}. \tag{2.98}$$

We now take $f = \overline{f}_m$ to be a linear combination of a sequence of m base classifiers (weak learners) f_j, $j = 1, \ldots, m$, so that

$$\overline{f}_m(x) = \frac{1}{2} \sum_{j=1}^{m} \alpha_j f_j(x). \tag{2.99}$$

Note that the factor $1/2$, which makes subsequent calculations convenient, does not affect the sign of the output that is used to make predictions. Training involves minimizing the empirical risk with respect to both the base classifiers $\{f_j\}$ and the combination coefficients $\{\alpha_j\}$.

We shall proceed sequentially. Suppose that we have learned $f_1, \ldots, f_{m-1}$ and $\alpha_1, \ldots, \alpha_{m-1}$. Our goal is to determine f_m, α_m. We have

$$\begin{aligned} R_{\text{emp}}(\overline{f}_m) &= \sum_{i=1}^{N} \exp\left(-y_i \overline{f}_{m-1}(x_i) - \frac{1}{2} y_i \alpha_m f_m(x_i)\right) \\ &= \sum_{i=1}^{N} w_i^{(m)} \exp\left(-\frac{1}{2} y_i \alpha_m f_m(x_i)\right), \end{aligned} \tag{2.100}$$

where the coefficients $w_i^{(m)} = \exp(-y_i \overline{f}_{m-1}(x_i))$ can be treated as constants since we are only minimizing over f_m and α_m. Now, we have

$$\begin{aligned} R_{\text{emp}}(\overline{f}_m) &= e^{-\alpha_m/2} \sum_{i=1}^{N} \mathbb{1}_{f_m(x_i)=y_i} w_i^{(m)} + e^{\alpha_m/2} \sum_{i=1}^{N} \mathbb{1}_{f_m(x_i)\neq y_i} w_i^{(m)} \\ &= (e^{\alpha_m/2} - e^{-\alpha_m/2}) \sum_{i=1}^{N} \mathbb{1}_{f_m(x_i)\neq y_i} w_i^{(m)} + e^{-\alpha_m/2} \sum_{i=1}^{N} w_i^{(m)}. \end{aligned} \tag{2.101}$$

Observe that the last term and the factor in the first term is independent of f_m, so minimizing with respect to f_m is equivalent to (2.94). Moreover, minimizing with respect to α_m above corresponds to the update rule (2.95). Lastly, with the choice of f_m and α_m, the weights on the data points is updated as

$$w_i^{(m+1)} = w_i^{(m)} e^{-\alpha_m y_i f_m(x_i)/2}. \tag{2.102}$$

Using the fact that $y_i f_m(x_i) = 1 - 2\mathbb{1}_{y_i \neq f_m(x_i)}$, we see that the update rule for weights is

$$w_i^{(m+1)} = w_i^{(m)} e^{-\alpha_m/2} e^{\alpha_m \mathbb{1}_{y_i \neq f_m(x_i)}}. \tag{2.103}$$

Discarding the constant factor $e^{-\alpha_m/2}$ which is independent of i, we obtain the weight update scheme (2.96).

2.6.3 Further Reading

There is a vast literature on model ensembling. For bagging, one of the primary algorithms used in practice is random forests [Bre01], which combines bootstrap aggregation with decision trees and random feature selection. For boosting algorithms, there are numerous extensions to AdaBoost not discussed here, including multi-class classification and regression. The underlying idea of these extensions are based on the exponential loss interpretation presented earlier. Another popular boosting method is *gradient boosting*, where each subsequent regressor is added in the gradient direction (in function space) of the loss function. See [FHT00, FHT01] for an exposition on these methods and beyond.

2.7 Neural Networks

We started the discussion on supervised learning models on linear basis models and their variants, including kernel machines and the SVM. Common to all these models is the fact that the basis functions, or feature maps, are chosen *a priori* and does not depend on the training dataset. In Section 2.5, we introduced decision trees, which is the first example where the basis functions are chosen at training time according to the dataset. Concretely, the decision tree basis function $\phi_j(x) = \mathbb{1}_{x \in \mathcal{R}_j}$ depends on the partition $\{\mathcal{R}_j\}$, which is learned from the data, e.g. using the recursive splitting algorithm.

In this section, we are going to introduce another class of such models where the basis functions are learned from the data: *neural networks*. Machine learning models based on neural networks, in particular *deep learning*, have become increasingly popular with the increasing amount of data and computational power we have. The term "neural networks" originated from the fact that these models were first developed as an attempt to model, in a mathematically precise way, neural interactions in the human brain [MP43, WH60]. However, it became clear quickly that such models are over-simplified and lack complexity required to understand human physiology. Nevertheless, they form a class of powerful machine learning models that admit unique properties that are worth studying. In the following, we start with the basics of shallow neural networks.

2.7.1 Shallow Neural Networks

For simplicity, let us consider a regression problem in d dimensions, so that the oracle function is $f^* : \mathbb{R}^d \to \mathbb{R}$. A shallow neural network corresponds to the following hypothesis space

$$\mathcal{H}_{\text{nn},M} = \left\{ f : f(x) = \sum_{j=1}^{M} v_j \sigma(w_j^\top x + b_j), w_j \in \mathbb{R}^d, v_j \in \mathbb{R}, b_j \in \mathbb{R} \right\} \tag{2.104}$$

Let us now introduce some nomenclature that is customary in the study of neural networks. The function $\sigma : \mathbb{R} \to \mathbb{R}$ is called an *activation function*. Popular choices include

ReLU (Rectified Linear Unit)	$\sigma(z) = \max(0, z)$	(2.105)
Leaky ReLU	$\sigma(z) = \max(0, z) + \delta \min(0, z)$	(2.106)
Tanh	$\sigma(z) = \tanh(z)$	(2.107)
Sigmoid	$\sigma(z) = \dfrac{1}{1 + e^{-z}}$	(2.108)
Soft-plus	$\sigma(z) = \log(1 + e^z)$	(2.109)

but this list is of course not exhaustive. Next, the parameters w_j are often called *weights* and b_j are called *biases*. Recall that in linear models, we tend to combine them by appending "1" to the input state x. However, here we will write out the bias term explicitly to conform with popular notation. We will refer to v_j as *coefficients*, but in deeper models they can also be regarded as weights. Finally, the number M is the dimensional of the *hidden layer* and this controls the complexity of the model. Often, we refer to $h_j = w_j^\top x + b_j$ as the value on the j^{th} *hidden node*. Thus, M is the number of hidden nodes in the neural network. Figure 2.8 gives an illustration of a shallow neural network model for prediction.

Linear vs Nonlinear Approximation Let us now discuss the approximation properties of shallow neural networks. We begin by stressing again the difference between (2.104) and linear basis models. For (2.104), we are allowed to choose w_j, b_j after seeing the data or the oracle function f^*. On the other hand, the linear basis equivalent of (2.104) is

$$\mathcal{H}_{\text{linear}} = \left\{ f : f(x) = \sum_{j=1}^{M} v_j \sigma(w_j^\top x + b_j), v_j \in \mathbb{R} \right\} \tag{2.110}$$

with a fixed collection of $w_j \in \mathbb{R}^d$, $b_j \in \mathbb{R}$ for $j = 1, \ldots, M$. This is linear because for any $f_1, f_2 \in \mathcal{H}_{\text{linear}}$ and $\lambda_1, \lambda_2 \in \mathbb{R}$, $\lambda_1 f_1 + \lambda_2 f_2 \in \mathcal{H}_{\text{linear}}$. The same is *not* true for $\mathcal{H}_{\text{nn},M}$ since this would require in general $2M$ terms in the summation. So, what is the difference between linear and nonlinear hypothesis spaces? We give a simple illustration in Example 2.23. In general, nonlinear hypothesis space offers a more efficient representation of functions, but we pay an price of obtaining a often harder optimization problem.

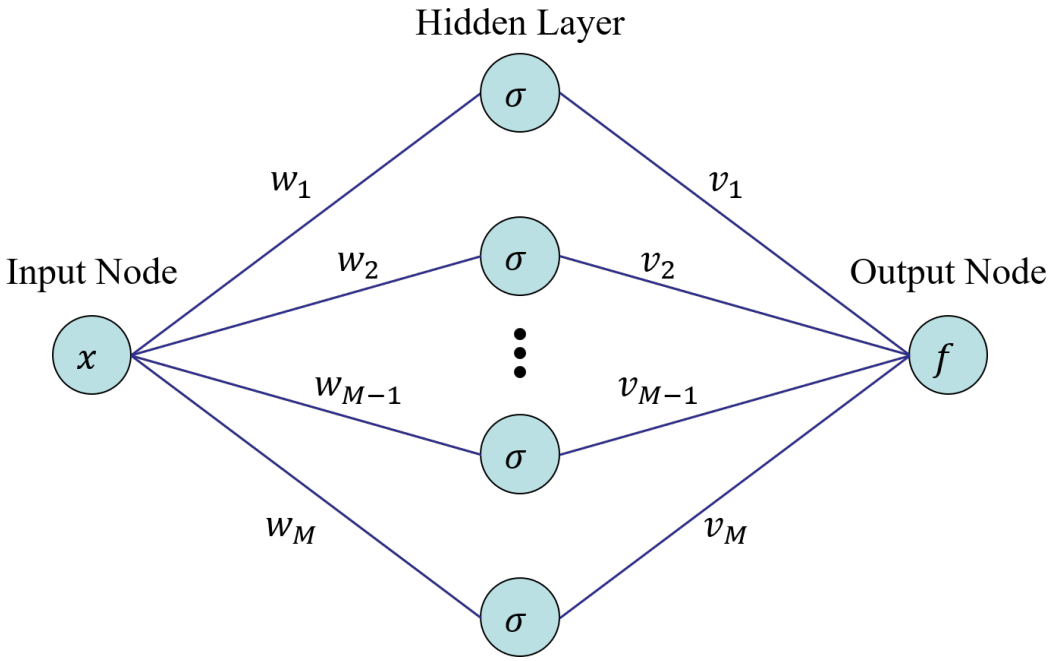


Figure 2.8: Illustration of a function parameterized by a shallow neural network with one hidden layer. For convenience we ignore the biases.

Example 2.23: Cosine Series Approximation

Let us consider a 1D problem. Suppose we are given an oracle function $f^*(x) = \cos(x/2)$, and we attempt to approximate it using a cosine series. Consider the following two choices:

$$\mathcal{H}_{\text{linear}} = \{f : f(x) = \textstyle\sum_{j=0}^{M} a_j \cos(jx)\} \tag{2.111}$$

$$\mathcal{H}_{\text{nonlinear}} = \{f : f(x) = \textstyle\sum_{j=0}^{M} a_j \cos(w_j x), w_j \in \mathbb{R}\} \tag{2.112}$$

It is clear that in the nonlinear case, we could simply take $M = 0$ and obtain a perfect approximation by picking $a_0 = 1$, $w_0 = 1/2$. However, in the linear case we would not have perfect approximation for any finite M; In fact, the best $(M+1)$-term linear approximation is the cosine series

$$\hat{f}(x) = \textstyle\sum_{j=0}^{M} \frac{2(-1)^j}{\pi(1-4j^2)} \cos(jx). \tag{2.113}$$

Universal Approximation Theorem To further demonstrate its approximation power, let us now discuss a foundational result in approximation theory of neural networks. This is known as the *universal approximation theorem*. In words, it says that given enough hidden nodes, a neural network can approximate *any* function to *arbitrary* accuracy. Let us give the precise

statement of this result below.

Theorem 2.24: Universal Approximation Theorem for Neural Networks

Let $K \subset \mathbb{R}^d$ be closed and bounded and $f^* : K \to \mathbb{R}$ be continuous. Assume that the activation function σ is *sigmoidal*, i.e. σ is continuous and $\lim_{z\to\infty} \sigma(z) = 1$, $\lim_{z\to-\infty} \sigma(z) = 0$. Then, for every $\epsilon > 0$ there exists $f \in \cup_{M\geq 1} \mathcal{H}_{\text{nn},M}$ such that

$$\|f - f^*\|_{C(K)} = \max_{x\in K} |f(x) - f^*(x)| < \epsilon \tag{2.114}$$

A general proof of Theorem 2.24 follows from an application of the Hahn-Banach theorem and the Riesz-Markov representation theorem, together with an argument based on the non-degenerate properties of σ [Cyb89], although there are also many other proofs using different techniques (see further reading at the end of this section). Since understanding the proof requires some knowledge of functional analysis that is not covered in these notes, we will not present the details.

2.7.2 Optimizing Neural Networks

The universal approximation theorem (Theorem 2.24) ensures that a neural network can be built to approximate any continuous function f^* on a compact domain. However, it does not tell us *how* to build it. In other words, while the approximation problem (c.f. Section 2.1 and Figure 2.1) is settled, the optimization problem remains. In this section, we shall discuss the *gradient descent* method for optimizing machine learning models, including but not limited to neural networks.

As demonstrated in many previous cases, in practice the empirical risk minimization problem can be written as an optimization problem over certain trainable parameters. In other words, we assume that the Hypothesis space admits a *parameterization*, so that $\mathcal{H} = \{f : f(x) = f_\theta(x) : \theta \in \Theta\}$. The set Θ is the allowed set of trainable parameters. For example, in the case of shallow neural networks (2.104), $\theta = (v, w, b) \in \Theta = \mathbb{R}^{(2+d)M}$. In most applications we can let $\Theta = \mathbb{R}^p$ be a Euclidean space, but there of course exists important exceptions (e.g. quantized networks). In the following, we will assume $\Theta = \mathbb{R}^p$. Consequently, the empirical risk minimization can be written as

$$\min_{f\in\mathcal{H}} R_{\text{emp}}(f) = \min_{\theta\in\in\mathbb{R}^p} R_{\text{emp}}(f(\theta)) = \min_{\theta\in\in\mathbb{R}^p} \frac{1}{N}\sum_{i=1}^{N} L(f_\theta(x_i), y_i). \tag{2.115}$$

Recall that L is the loss function. To introduce optimization methods, it is convenient to introduce the following shorthand

$$\Phi_i(\theta) = L(f_\theta(x_i), y_i), \qquad \Phi(\theta) = \frac{1}{N}\sum_{i=1}^{N} \Phi_i(\theta). \tag{2.116}$$

Then, the empirical risk minimization problem aims to solve $\min_{\theta \in \mathbb{R}^p} \Phi(\theta)$.

Gradient Descent. Except for simple cases (e.g. least squares), minimizing $\Phi(\theta)$ does not admit an explicit solution, and we often resort to iterative approximation methods that are implementable on a computer. We now introduce the simplest of them all, the gradient descent (GD) algorithm. Notice that the gradient vector $\nabla\Phi(\theta)$ always points in the steepest ascent direction on the surface defined by $z = \Phi(\theta)$, and hence to decrease $\Phi(\theta)$ we should go in the *opposite* direction, the steepest descent direction given by $-\nabla\Phi(\theta)$ (Figure 2.9). How long a step should we take? This is controlled by a parameter called the *learning rate* or *step size*, $\eta > 0$. It is typically taken to be small to ensure stability of the algorithm. The algorithm is summarized in 3 and also illustrated on Figure 2.9.

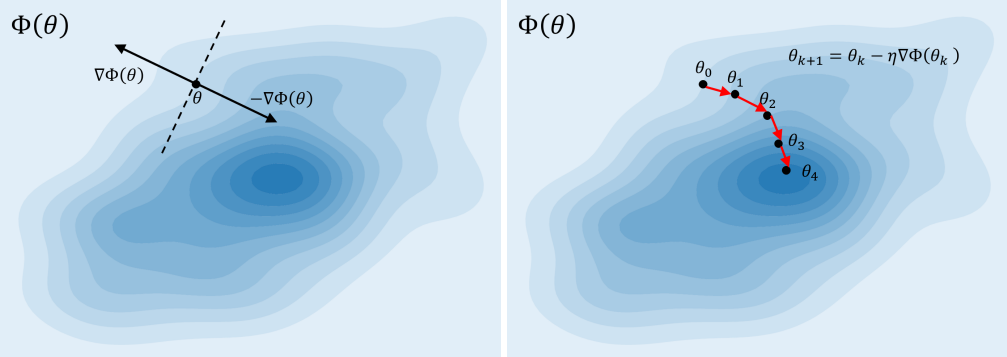


Figure 2.9: Illustration of gradient descent for function minimization. Left: $\nabla\Phi$ always points in the steepest descent direction. Right: the path of the gradient descent algorithm.

Algorithm 3: Gradient Descent

Hyperparameters: K (# iterations), η (learning rate)
Initialize: $\theta_0 \in \mathbb{R}^p$
for $k = 0, 1, \ldots, K-1$ **do**
  $\theta_{k+1} = \theta_k - \eta\nabla\Phi(\theta_k)$
end
return θ_K

It can be shown that if Φ is sufficiently well-behaved, e.g. if $\nabla\Phi$ is globally Lipschitz, then $\|\nabla\Phi(\theta_k)\| \to 0$ as $k \to \infty$ for η sufficiently small [Nes04]. In other words, there is at least a subsequence of GD iterates that converge to a stationary point. However, we want to to *minimize* Φ and not just find a stationary point. Does GD converge to a minimum? To discuss this question, we need to differentiate between two kinds of minima: local and global.

Local vs Global Minima. We begin with concrete definitions.

Definition 2.25: Local and Global Minima

Let $\Phi : \mathbb{R}^p \to \mathbb{R}$ be a function. We say that θ^* is *local minimum* of Φ if there exists $\delta > 0$ such that

$$\Phi(\theta^*) \leq \Phi(\theta) \text{ for all } \theta \in \mathbb{R}^p \text{ satisfying } \|\theta - \theta^*\| \leq \delta. \tag{2.117}$$

We say that it is a *global minimum* if

$$\Phi(\theta^*) \leq \Phi(\theta) \text{ for all } \theta \in \mathbb{R}^p. \tag{2.118}$$

It turns out that under general conditions, one can show that gradient descent almost always converges to a local minimum. However, in general it does not converge to a global minimum unless we assume additional conditions on the function Φ. Let us give an example of such a condition that is commonly encountered in machine learning.

Definition 2.26: Convex Functions

We say that a function $\Phi : \mathbb{R}^p \to \mathbb{R}$ is *convex* if

$$\Phi(\lambda\theta + (1-\lambda)\theta') \leq \lambda\Phi(\theta) + (1-\lambda)\Phi(\theta') \tag{2.119}$$

for all $\theta, \theta' \in \mathbb{R}^p$ and $\lambda \in [0, 1]$.

It turns out that if Φ is convex, then one can show that all local minima are automatically global minima. We state and prove this result in Proposition 2.27.

Proposition 2.27

Let $\Phi : \mathbb{R}^p \to \mathbb{R}$ be convex. Suppose θ^* is a local minimum of Φ, then it is a global minimum of Φ.

Proof. Without loss of generality we assume $\Phi(\theta^*) = 0$ (otherwise just replace $\Phi(\theta)$ by $\Phi(\theta) - \Phi(\theta^*)$). Suppose for the sake of contradiction that exists $s \in \mathbb{R}^p$ such that $\Phi(s) < 0$. Define $u(\lambda) = \lambda s + (1-\lambda)\theta^*$ for $\lambda \in [0, 1]$. By the definition of convexity (2.119), we have

$$\Phi(u(\lambda)) \leq \lambda\Phi(s) + (1-\lambda)\Phi(\theta^*) = \lambda\Phi(s), \tag{2.120}$$

or $\Phi(s) \geq \Phi(u(\lambda))/\lambda$ for all $\lambda \in (0, 1]$. But, $\|u(\lambda) - \theta^*\| = \lambda\|s - \theta^*\|$. Picking $\lambda = \min(1, \delta/\|s - \theta^*\|)$ gives $\|u(\lambda) - \theta^*\| \leq \delta$. By definition of local minimum, $\Phi(u(\lambda)) \geq 0$ and so $\Phi(s) \geq \Phi(u(\lambda))/\lambda \geq 0$, contradicting our premise that $\Phi(s) < 0$.

Consequently, as long as Φ is convex, GD can solve the empirical risk minimization problem.

Furthermore, for convex functions one can actually give a *rate* at which GD converges: to reach an error of ϵ in the function value, we roughly require at most $O(\epsilon^{-1})$ GD iterations. If stronger conditions are assumed on Φ (e.g. strong convexity), then this rate can be faster.

Stochastic Gradient Descent. So far we have discussed GD in quite general optimization problems. However, there is some structure to the objective function Φ in empirical risk minimization problems: it is written as an average over objectives due to each sample [c.f. (2.116)]

$$\Phi(\theta) = \frac{1}{N}\sum_{i=1}^{N} \Phi_i(\theta). \tag{2.121}$$

By linearity of the gradient operation, we also have

$$\nabla\Phi(\theta) = \frac{1}{N}\sum_{i=1}^{N} \nabla\Phi_i(\theta). \tag{2.122}$$

When the number of samples N is large, computing the gradient above, which is necessary for every step of the GD algorithm (Algorithm 3), becomes prohibitively expensive.

To tackle this problem, we use instead stochastic gradient descent [RM51] Here, instead of computing the full gradient $\nabla\Phi(\theta)$, we can sample a sub-sample of objectives corresponding to the training samples and compute a random gradient estimate for use in the descent algorithm. Algorithm 4 summarizes the stochastic gradient descent algorithm. Unlike GD, in each step of SGD we only need to compute a sum of B gradients instead of N gradients. In large-scale applications, N can be millions yet B can typically be picked to be less than 100, offering tremendous savings. Of course, the price we pay here is that the random gradients g_k introduce *random fluctuations* and complicates the analysis of the descent algorithm: although $\mathbb{E}[g_k|\theta_k] = \nabla\Phi(\theta_k)$, its variance can play a large role in determining the behavior of the algorithm.

Algorithm 4: Stochastic Gradient Descent

Hyperparameters: K (# iterations), η (learning rate), B (batch size)
Initialize: $\theta_0 \in \mathbb{R}^p$
for $k = 0, 1, \ldots, K-1$ **do**
  Sample $\{i_1, \ldots, i_B\} \subset \{1, \ldots, N\}$ uniformly at random;
  Compute $g_k = \frac{1}{B}\sum_{j=1}^{B} \nabla\Phi_{i_j}(\theta_k)$;
  Update $\theta_{k+1} = \theta_k - \eta g_k$;
end
return θ_K

2.7.3 Deep Neural Networks and Back-Propagation

So far, we have introduced shallow neural networks and discussed their optimization and approximation properties. In this section, we discuss the extension of shallow neural networks

to *deep* neural networks (DNN), forming the basis of the deep learning revolution we are witnessing today.

The simplest type of DNN are the so-called deep fully-connected networks, where we simply iterate the structure of shallow neural networks T times. T is the *depth* of the DNN. Concretely, deep neural networks make up the following hypothesis space

$$\begin{aligned}&\mathcal{H}_{\text{dnn}} = \left\{f : f(x) = v^\top f_T(x), v \in \mathbb{R}^{d_T}\right\}\\ &\text{where}\\ &f_{t+1}(x) = \sigma(W_t f_t(x) + b_t), \quad W_t \in \mathbb{R}^{d_{t+1}\times d_t}, \quad b_t \in \mathbb{R}^{d_{t+1}}, \quad t = 0, \dots, T-1\\ &\text{with} \quad d_0 = d, \quad f_0(x) = x.\end{aligned} \tag{2.123}$$

The trainable parameters are the weights $\{W_0, \dots, W_{T-1}\}$, biases $\{b_0, \dots, b_{T-1}\}$ and the final combination (final layer) weights v. The activation function is applied element-wise to each vector, i.e. $\sigma(z)_i = \sigma(z_i)$. Figure 2.10 shows a standard fully connected DNN. Compared to shallow neural networks, deep neural networks has the advantage that it can represent hierarchical features naturally, owing to the compositional nature of the hypothesis space. One can show that just like shallow networks, DNNs have the universal approximation property: if the number of layers is large enough, provided that the width of each layer $\{d_t\}$ are not too small [Han17, LPW+17].

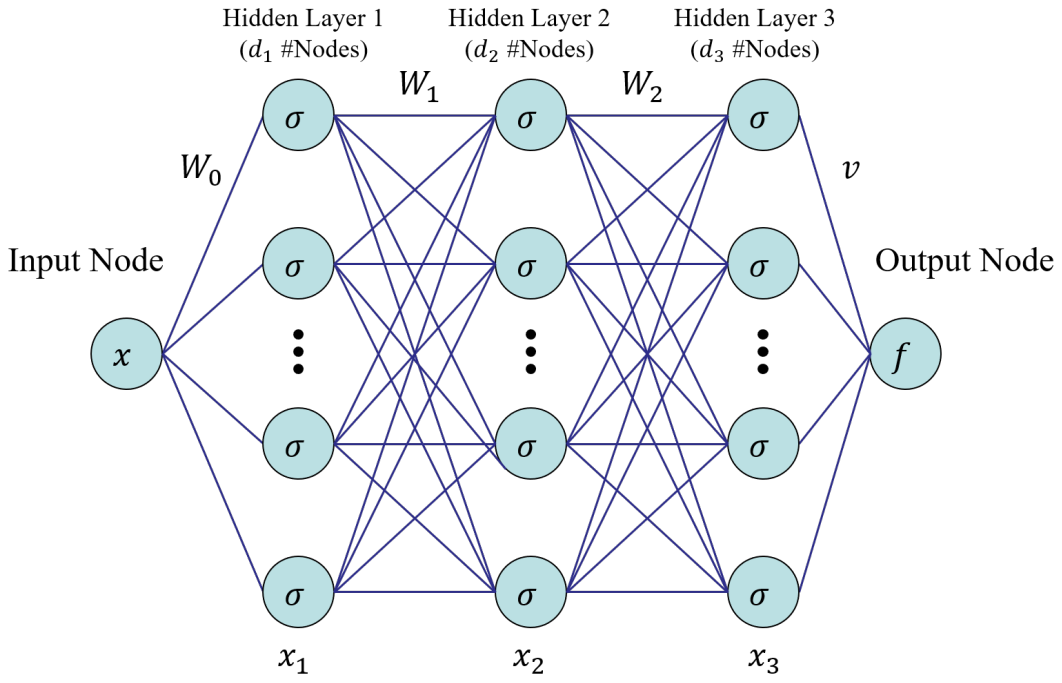


Figure 2.10: Illustration of a function parameterized by a deep neural network with three hidden layers. As before, we ignore the biases.

Back-propagation Algorithm. Just like shallow networks, DNNs can be trained using GD or SGD. The only complication is, since there are many trainable parameters, can we have a efficient way to compute the (stochastic) gradient? Recall that we have the sample-wise objective

$$\Phi_i(\theta) = L(v^\top f_T(x_i), y_i), \tag{2.124}$$

with f_T computed using the iteration rule (2.123). Thus, trainable parameters are of the form

$$\theta = \{W_0, \dots, W_{T-1}; b_0, \dots, b_{T-1}; v\} \tag{2.125}$$

and we have to compute the derivative of Φ_i with respect to each of these parameters. For simplicity, we shall drop the biases (they can be absorbed into the weights by adding a 1 to the state vector).

On first look, this may seem daunting, but it turns out that we can use the chain rule to simply this computation, and it is rather efficient to compute the required gradient vectors sequentially. This is the content of the back-propagation algorithm [LeC88**?** , BH75]. To describe this algorithm, it is instructive to rewrite the iteration rule in (2.123) (without biases) for a specific sample (x, y) as follows:

$$\begin{aligned} &x_{t+1} = g_t(x_t, W_t) \qquad t = 0, \dots, T \\ &\text{where } x_0 = x, \end{aligned} \tag{2.126}$$

For the specific case of DNNs, the final layer is absorbed into layer $t = T$, with $v = W_T$. Moreover, the feed-forward function $g_t(x, W) = \sigma(Wx)$ for all layers other than $t = T$, where $g_T(x, W) = Wx$. The loss function thus has the form $\Phi(\theta) = L(x_{T+1}, y)$. However, the formulation above will include any recursive architecture since g_t is assumed to be a general feed-forward function. We will derive the back-propagation formula for this general scenario.

To compute the derivative $\nabla_{W_t}\Phi(\theta)$, we make the crucial observation: due to the feed-forward nature of the DNN, given x_{t+1}, x_{T+1} does not depend on W_s for $s \le t$. Thus, we can write

$$\Phi(\theta) = L(x_{T+1}(x_{t+1}(x, W_0, \dots, W_t); W_{t+1}, \dots, W_T), y), \tag{2.127}$$

and so by the chain rule we have

$$\begin{aligned} \nabla_{W_t}\Phi(\theta) &= [\nabla_{W_t} x_{t+1}]^\top \nabla_{x_{t+1}} L(x_{T+1}, y) \\ &= [\nabla_{W_t} g_t(x_t, W_t)]^\top \nabla_{x_{t+1}} L(x_{T+1}, y). \end{aligned} \tag{2.128}$$

Let us define $p_t = \nabla_{x_t} L(x_{T+1}(x_t; W_t, \dots, W_T), y)$, then we have

$$\nabla_{W_t}\Phi(\theta) = [\nabla_{W_t} g_t(x_t, W_t)]^\top p_{t+1}. \tag{2.129}$$

Therefore, as long as we can compute $\{p_t\}$, we can compute the gradients readily. Observe that p_t can in fact be computed via a backward pass, which is again a consequence of the chain rule

$$p_t = [\nabla_{x_t} g_t(x_t, W_t)]^\top p_{t+1}, \qquad p_{T+1} = \nabla_{x_{T+1}} L(x_{T+1}, W_T). \tag{2.130}$$

Hence, the back-propagation algorithm can be carried out as follows: feed-forward to compute $\{x_t\}$, feed-backward to compute $\{p_t\}$ along with the derivatives according to (2.129). The back-propagation algorithm to compute parameter gradients are summarized in Algorithm 5. Once gradients are computed, standard GD/SGD algorithms can be applied to optimize DNNs.

Remark. The above presentation of the back-propagation algorithm may appear different from many machine learning references, but it is more succinct. More importantly, it highlights an important origin from optimal control theory [Ber00, BH75, Pon18]. In fact, this is a special case of the *sweeping algorithm*, or *method of successive approximations* [CL82] – one of the most basic algorithms in optimal control theory, and the algorithm is based on the solution of the Pontryagin's maximum principle [BGP61]. The variables $\{p_t\}$ are called *co-states* and can be interpreted as Lagrange multipliers. There are also important connections between (2.130) and the *variational equation* in the theory of ordinary differential equations [Arn12]. In fact, (2.130) is the discrete analogue of the variational equation.

Algorithm 5: back-propagation Algorithm

Initialize: $x_0 = x \in \mathbb{R}^d$
for $t = 0, 1, \ldots, T$ **do**
  $x_{t+1} = g_t(x_t, W_t) = \sigma(W_t x_t)$;
end
Set $p_{T+1} = \nabla_{x_{T+1}} L(x_{T+1}, y)$;
for $t = T, T-1, \ldots, 1$ **do**
  $\nabla_{W_t} \Phi = p_{t+1}^\top \nabla_{W_t} g_t(x_t, W_t)$;
  $p_t = [\nabla_{x_t} g_t(x_t, W_t)]^\top p_{t+1}$;
end
return $\{\nabla_{W_t} \Phi : t = 0, \ldots, T\}$

2.7.4 Further Reading

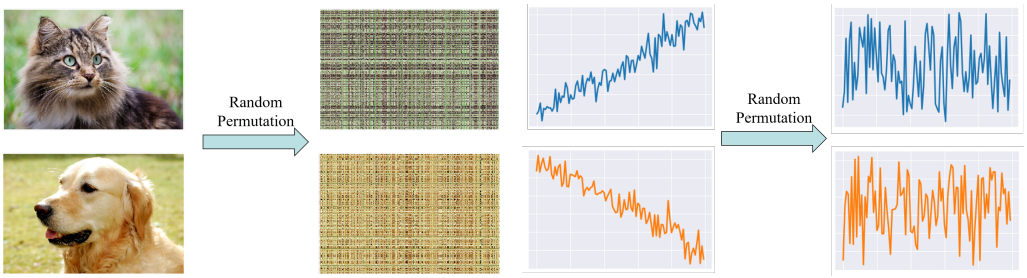
First, the topic of nonlinear approximation and its differences from linear fixed-terms approximation is reviewed in [DeV98]. The universal approximation theorem for shallow networks has been the subject of a lot of attention in the earlier developments in neural networks, with many theoretical works proving it under various assumptions and conditions, see e.g. [Cyb89, HSW89, SW89, SW90, HSW90, Hor91, Hor93, Bar93, Bar94]. More recently, such approximation theorems have been extended to deep neural networks [Han17, LPW+17, LJ18, Zho19, SYZ19, EMW19].

Next, on the topic of optimization in machine learning, there are many important extensions to the simple (S)GD algorithm. A recent review is [BCN18], although there are many new developments as well. One point of view is that the SGD and its variants can be viewed as a noisy gradient flow corresponding to a stochastic differential equation, leading to connections between large-scale optimization, diffusion processes, optimal control and beyond [LTE17, LTE19].

Finally, with regard to deep learning, as alluded to earlier there is also a connection between deep neural network architecture itself to differential equations. The mathematical theory for this viewpoint is laid out in [LCTE17, EHL+19]. Finally, we have only introduced the simplest of deep learning architectures. In the next section, we will introduce two classes of deep learning architectures for structured data.

2.8 Deep Learning for Data with Structures

In principle, almost any input data (image, time series, tabular values) can be flattened and concatenated in to a vector in $\mathbb{R}^d$, after which we can apply any of the machine learning methods introduced in the previous sections, including fully-connected NNs. However, there are good reasons not to perform such flattening transformations. One reason is that there is often no canonical means of flattening, and a arbitrarily picked one may destroy some structured information that was present in the input data. For example, two distinctly different classes of images, when flattened, may become much harder to distinguish (See Figure 2.11). For these types of data with intrinsic structures and symmetries, it is desirable to build models that directly work on these data and respect their structures and symmetries. In this section, we discuss several deep learning architectures that exploit this fact.



(a) Image (b) Time Series

Figure 2.11: Illustration of the effect of permutation on data with spatial or temporal structure. Applying a permutation on the spatial or temporal indices destroys such structure, but fully connected neural networks treats these two cases equivalently, thus unsuitable to process such data types.

2.8.1 Convolutional Neural Networks

In this section, we introduce a popular architecture for image and time series applications, known as the *convolutional neural network* (CNN). To motivate ideas, we begin with a quick introduction to the idea of convolution, which is ubiquitous in signal processing and harmonic analysis.

Convolutions. Let us consider two real valued functions $x, w : \mathbb{R} \to \mathbb{R}$. The *convolution* of w and x, which we denote by $w * x$, is given by

$$(w * x)(\tau) = \int_{-\infty}^{\infty} w(t)x(\tau - t)dt \tag{2.131}$$

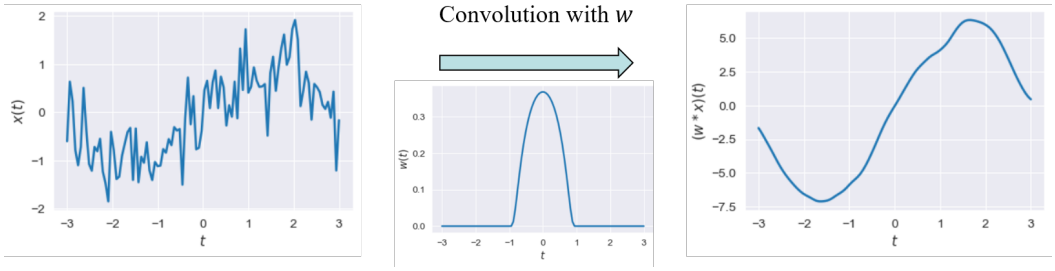
We shall assume that w, x are such that the above integral exists for all τ. The following basic properties are immediate from the definition of convolutions:

1. Commutativity: $w * x = x * w$
2. Linearity: $w * (\lambda_1 x_1 + \lambda_2 x_2) = \lambda_1 w * x_1 + \lambda_2 w * x_2$

So, what does the convolution do? Let us give some intuitions below.

Example 2.28: Some Intuitions Behind Convolution

Let us consider two real valued random variables U, V with probability densities $w(u)$ and $x(v)$ respectively. What is the probability density of $U + V$? Observe that for any value τ, there are infinite number of possibilities such that $u + v = \tau$. In fact, for any u, as long as we set $v = \tau - u$ we satisfy $u + v = \tau$. Hence, the probability density of $U + V$ evaluated at some value τ must be obtained by summing, or integrating, the product of the probability density of U at u and that of V at $\tau - u$ over all of u. This is precisely the integral (2.131), and so the density of $U + V$ is $w * x$. In this sense, the convolution operation can be thought of as a *fuzzy* version of addition, just like how random variables are *fuzzy* versions of numbers.
Another more visual illustration is following. Suppose that w is a smooth bump function and x is some rough function of its argument. Then $w * x$ smooths out x, kind of like how a myopic person sees the world. This is illustrated below. In fact, as long as w is smooth, one can show that its Fourier transform decays quickly, and so it has the same smoothing properties by filtering out the high frequency components of x when convolved with it. This is a consequence of the *convolution theorem*, which says that $\mathcal{F}(w * x) = \mathcal{F}(w)\mathcal{F}(x)$, where $\mathcal{F}$ denotes the Fourier transform.



Discrete Convolutions. While the theory of convolutions of continuous functions plays an important role in mathematics, particularly in the field of probability theory and harmonic analysis, very often for applications we are faced with discrete signals. They can be discrete in a generic sense, or more often, they are discrete samples from a continuous signal. One may think that such a sampling procedure always loses information, but a classical result due to Shannon [Sha49] shows that if a function is band-limited (i.e. its Fourier transform is compactly supported), then it can be recovered from discrete measurements.

In machine learning, signals (pictures, time series) are often represented as vectors or tensors, which may come from discrete samplings from continuous data. To apply the concept of convolution just like in the continuous case, we need to define a discrete analogue. Given

two *infinitely-long* vectors $w = \{w(i) : i \in \mathbb{Z}\}$ and $x = \{x(i) : i \in Z\}$, we define their *discrete convolution* as

$$(w * x)(k) = \sum_{i=-\infty}^{\infty} w(i)x(k-i). \tag{2.132}$$

Notice that the usual properties, such as commutativity and linearity, and in fact many other properties from the continuous case, are retained. In particular, observe that if w is a discrete bump function, then w also smooths a potentially rough signal x.

Notice that if we were to flip the kernel w so that $w(n) \mapsto w(-n)$, we would have

$$(w * x)(k) = \sum_{i=-\infty}^{\infty} w(i)x(k+i). \tag{2.133}$$

Technically, this is not a convolution but rather a *cross-correlation*, but often in machine learning there is no need to make such a distinction, since it only involves flipping the definition of w. We shall hereafter abuse notation, as is common in machine learning literature and programming libraries, to refer to the above also as a convolution.

Padding for Finite Convolutions. In practice, however, we can only represent finite values, meaning that w and x are finite vectors. Let us say that their lengths are m and n respectively. In this case, we need to take care of the *boundary conditions*. There are a few possible ways to do so, leading to different types of convolutions. In the following, we shall assume that $m < n$, and we will call w the *kernel*[2] or *filter* and x the *signal*.

- Circular convolution:

$$(w * x)(k) = \sum_{i=0}^{m-1} w(i)x(k+i), \qquad k = 0, \dots, n-1, \tag{2.134}$$

  where x is periodically extended so that $x(j) = x(j-n)$ for $j \geq n$.

- Valid convolution:

$$(w * x)(k) = \sum_{i=0}^{m-1} w(i)x(k+i), \qquad k = 0, \dots, n-m, \tag{2.135}$$

  in which case no periodic extension is needed.

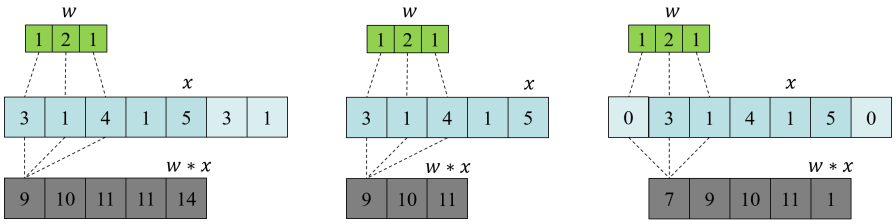
- Convolution with same zero padding:

$$(w * x)(k) = \sum_{i=0}^{m-1} w(i)x(k+i-\lfloor m/2 \rfloor), \qquad k = 0, \dots, n-1, \tag{2.136}$$

[2] This is not the same as the kernel as in kernel methods in Section 2.3.

where x is padded with zeros, so that $x(j) = 0$ for $j \notin \{0, \dots, n-1\}$. The symbol $\lfloor z \rfloor$ denotes the biggest integer smaller or equal to z. This is to balance the zero padding on both sides. Some implementations ignore this balancing and just have $x(k+i)$ in the above.

Different types of convolutions are illustrated in Figure 2.12.



(a) Circular (b) Valid (c) Same

Figure 2.12: Illustration of different kind of boundary conditions for finite discrete convolutions. [TODO:Fix mistake in plot]

2D Convolutions Very often, we want the discrete convolution to deal with images and so it is useful to define convolutions in two dimensions. For two dimensional w, x (now matrices), we define

$$(w * x)(k, l) = \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} w(i, j) x(k+i, l+j) \tag{2.137}$$

or with a "$-$" sign above, if we do not flip the filters. As before, for finite signals this can be padded or periodically extended in the same way as in (2.134) to (2.136).

Input Data and Convolution Layer We will now focus on image data, to which convolutional neural networks are most commonly applied. Let us consider an input image x. In the unstructured case, we typically assume that $x \in \mathbb{R}^d$. In the case of images, x represents discrete samples from a continuous image, so $x \in \mathbb{R}^{d \times d}$, where $d \times d$ is the number of pixel samples. The intensity of the image at the $(i, j)^{\text{th}}$ pixel is denoted as $x(i, j)$, and is typically a value from 0 to 255 inclusive. This can be further normalized to the range $[0, 1]$.

If we are dealing with monochrome images this is sufficient. However, for colored images we usually have an additional dimension known as *channels*. In the case of RGB images, we would have a $d \times d$ matrix for each color, red, green and blue. Hence, the resulting image should be represented as a rank 3 tensor

$$x_k(i, j), \qquad i, j = 0, \dots, d-1, \qquad k = 0, \dots, c-1 \quad (c \text{ is \# channels}). \tag{2.138}$$

For monochrome or black-and-white images, we have $c = 1$.

A convolution layer is very much like a fully connected layer. Instead of defining a matrix weights to apply the linear transformation, we have a "matrix" W of convolution filters, i.e. each W_{lk} is a 2D convolution filter of size $m \times m$[3]. A simple convolution layer applies the following transformation to an input image $x_k(i, j)$

$$\sigma\left(\sum_k W_{lk} * x_k + b_l\right). \tag{2.139}$$

As before σ is a point-wise nonlinearity applied to each element in the tensor and b_l are the biases. What is the output dimension of this transformation? We can see that this is again a collection of images arranged in channels, where the number of channels is the number of values the index l can take. For each output channel l, the dimension of the output image corresponding to that channel depends on the mode of convolution: for circular and same padded convolutions, we again have a $d \times d$ image. For valid convolutions, the image size will decrease. Eq. (2.139) is the most basic form of a convolution layer, and there are many variants and generalizations of this, although they mostly comprise of basic building blocks of transformations such as above.

Why Convolutions? There are in fact many reasons one would want to use convolution layers instead of fully connected layers for image processing and related applications. In the following, we will discuss some of them.

First, recall that the convolution operation is a linear operation. Let us look at the 1D case: suppose that we have length 3 filter w and we apply it to a signal $x \in \mathbb{R}^5$ using the circular convolution boundary conditions. By writing out the product we can see that this is really a matrix multiplication:

$$w * x = \underbrace{\begin{pmatrix} w_0 & w_1 & w_2 & 0 & 0 \\ 0 & w_0 & w_1 & w_2 & 0 \\ 0 & 0 & w_0 & w_1 & w_2 \\ w_2 & 0 & 0 & w_0 & w_1 \\ w_1 & w_2 & 0 & 0 & w_0 \end{pmatrix}}_{C_w} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} \tag{2.140}$$

The matrix C_w is known as a Toeplitz matrix. The matrix representation of convolutions highlights an important property of convolution networks: *weight sharing*. Unlike fully connected networks where every element of the weight matrix can be varied, in a convolution the matrix representation shares weights across different points of space. To put it in another way, the structure of C_w ensures that the transformation applied to each spatial coordinate of x are not independent. This is sensible when working on images. Furthermore, notice that in the case where x is a very long vector where as the filter w has short length, then we will get a very sparse Toeplitz matrix. Together with weight sharing, we can see that although convolution can

[3] We can also have varying sizes for each filter

be represented as a matrix multiplication, it has a much smaller number of trainable variables. This is good for both learning and for storage.

Lastly, and perhaps most importantly, the convolution operation allows us to place a prior restriction on our hypothesis space. To understand this point, Let us take the example in (2.140), but we now translate the input signal to $x' = (x_4, x_0, x_1, x_2, x_3)$, i.e. shift it by one position downwards. By applying the same convolution with w, we see that $w * x'$ is just $w * x$, but also shifted one position down. More generally, let T be any translation operation, then we have

$$w * (Tx) = T(w * x). \tag{2.141}$$

Due to this fact, we say that convolutions are *equivariant* with translations. Equivalently, convolutions and translations commute. Lastly, observe that the point-wise nonlinear operation $z \mapsto \sigma(z)$ is also equivariant to translations. This means that the entire convolution layer (2.139) is equivariant to translations.

To see the significance of this, let us consider another concept called *invariance*. Let T be any transformation operator on the signal x (e.g. translation), we say that a function f is invariant under T if

$$f(Tx) = f(x) \qquad \text{for all } x. \tag{2.142}$$

An example of translation invariant mapping is

$$f(x) = \sum_i x(i), \tag{2.143}$$

but this is of course too simple to be used to model practical relationships. The equivariance property of convolutions allows us to easily construct more complicated models by the following observation: Suppose that a function g is equivariant and f is invariant with respect to a transformation T, then $f \circ g$ is invariant with respect to T. This is immediate since

$$(f \circ g)(Tx) = f(g(Tx)) = f(Tg(x)) = f(g(x)) = (f \circ g)(x). \tag{2.144}$$

Carrying on this line of reasoning, for any invariant f and any sequence of equivariant $\{g_j\}$, the function $f \circ g_1 \circ \cdots \circ g_J$ is invariant with respect to T.

Now, for many image applications, our oracle model is in fact translation invariant. Suppose we have an image x of a object, whose class label is given by $f^*(x)$, then, if we translate the image x we would not expect its class to change: a translated cat is still a cat. Mathematically, $f^*(T(x)) = f^*(x)$ for any translation T. The equivariance property of convolutional layers enables us to easily build models that are also invariant under translations. This highlights a important rule in choosing hypothesis spaces: if we know f^* satisfies some symmetry or invariance properties, we want to pick a hypothesis space so that every function in our hypothesis space also satisfies the same symmetry or invariance. This way, we can effectively decrease the size of the hypothesis space and improve approximation and optimization errors. This is visualized in Figure 2.13.

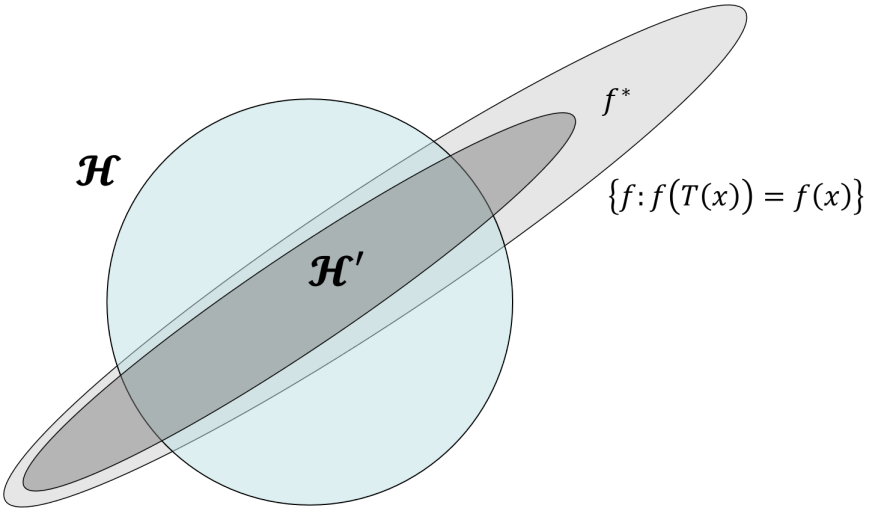


Figure 2.13: Schematic showing that it is desirable to shrink hypothesis space using information from f^*. If we know that f^* is invariant under T, then it is desirable to choose a hypothesis space $\mathcal{H}'$ whose member functions also respect such invariance.

Pooling. As another illustration of building invariance, we briefly discuss the idea of pooling. This is another type of transformation on signals x that builds invariance. The most commonly used version is *max pooling* with *stride* p, which corresponds to the following operation in 1D:

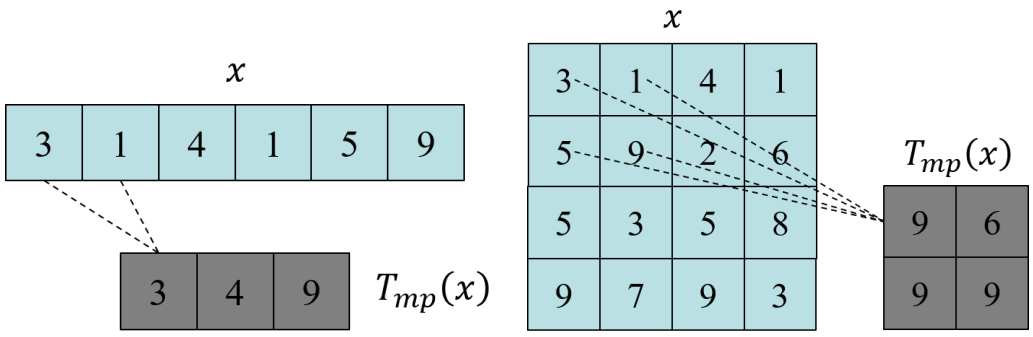
$$(T_{\text{mp}}x)(k) = \max_{i=kp,\ldots,(k+1)p} x(i). \tag{2.145}$$

The stride p indicates the length of the pooling window. Figure 2.14 illustrates the pooling operation in 1D and 2D. Notice that this transformation operation decreases the signal size by p times. Moreover, pooling has invariance to local deformations: if within each pooling window the signal x is deformed by permuting its pixel values, then the output remains invariant. This is another way to build another type of invariance that is useful in many image applications. We note here that there are many other variants of pooling, including average pooling and un-strided pooling, to name a few [GBC16].

Deep CNN Architecture. We conclude this section by giving an example of a basic deep convolutional neural network (CNN) architecture. We start with some multi-channel image $x \in \mathbb{R}^{d\times d\times c}$. Just like in deep fully connected neural networks, we recursive apply the following operations:

$$\begin{aligned} x_0 &= x \\ x_{t+1} &= T_{\text{mp}}T_{\text{conv}}x_t, \qquad t = 0,\ldots,T-1 \\ f(x) &= T_{\text{fc}}x_T, \end{aligned} \tag{2.146}$$

where T_{conv} refers to the simple convolution layer (2.139) and T_{mp} is the max pooling layer defined in (2.145). After each transformation, we may call x_t a feature map, which is a result



(a) 1D (b) 2D

Figure 2.14: Max pooling operation in 1 and 2 dimensions.

of the convolution and pooling operations. In the final layer, we flatten the image into a long vector and apply a simple (usually 1-2 hidden layers) fully connected NN, T_{fc} to yield our model predictions. Just as in Section 2.7.2, the network can be trained with GD/SGD with backpropagation. The trainable weights are the convolution weights and biases, together with the weights in the fully connected layers at the end. We illustrate this simple CNN in Figure 2.15. Note that there are many variants to the basic architecture and the reader is referred to [GBC16] and also current research literature for further exploration.

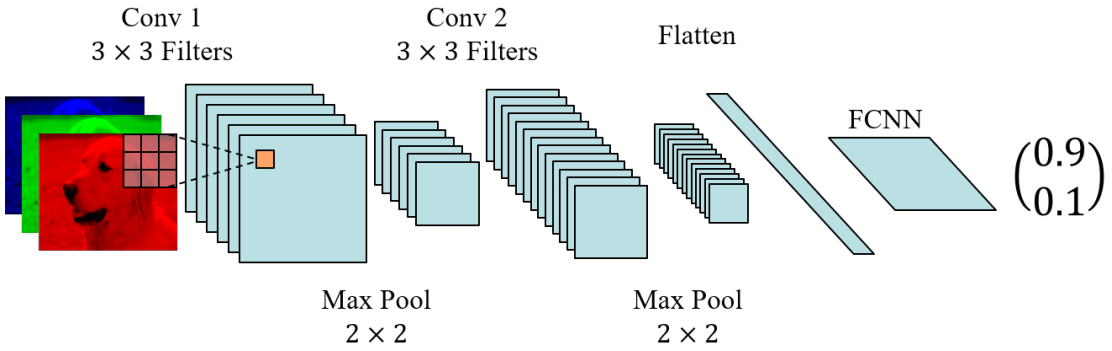


Figure 2.15: A basic deep CNN architecture.

2.8.2 Recurrent Neural Networks

Just like image data, time series data also exhibit some structure. Instead of spatial information, we must now take into account temporal information and correlations. One neural network architecture suited for such applications is recurrent neural networks. In this section, we will briefly introduce the basics of recurrent neural networks.

Time Series Data. We will denote an input data from a time series as

$$x = \{x(\tau) : \tau = 1, \dots, \tau_{\max}\}. \tag{2.147}$$

Here, τ denotes time and $\tau_{\max}$ is the terminal length and can be ∞ for some applications. Each x_τ may also contain additional structure (e.g. an image, in which x represents a video sequence), but for simplicity of exposition we will assume $x_\tau \in \mathbb{R}^d$ is a vector for each τ. Corresponding to an input time series x is an output. There are two types of outputs that one can consider. In the first type, y is just a scalar number or a class that corresponds to the entire time series input x. In the second type, the output y itself can also be a time series, in which case we need to make a prediction for every time step τ.

Recurrent Neural Networks and Parameter Sharing in Time Our goal is to build a relationship between x and y, which is again related by some oracle f^*, except that this oracle now works on the temporal sequence and may also produce a temporal sequence as output.

Recurrent neural networks (RNN) attempt to model the relationship between x and y by learning a *hidden* dynamical system. The essence of the model is the following difference equation

$$\begin{aligned} h_{\tau+1} &= g(h_\tau, x_{\tau+1}; \theta) \\ o_\tau &= u(h_\tau, \phi). \end{aligned} \tag{2.148}$$

The trainable parameters are θ, ϕ and the sequence of outputs $\{o_\tau : \tau = 1, \dots, \tau_{\max}\}$ is the prediction from our RNN model. The parameters θ, ϕ are trained so that $\{o_\tau\}$ closely approximates y_τ. If only a single input is required, then only $o_{\tau_{\max}}$ is used for prediction. Just as in traditional neural networks, the simplest RNN can be formed by taking g and u to be 1-layer neural networks, i.e.

$$\begin{aligned} g(h, x, \theta) &= \sigma(W_h h + W_x x + b), \qquad \theta = (W_h, W_x, b), \\ u(h, \phi) &= \sigma(W_o h + b_o), \qquad \theta = (W_o, b_o). \end{aligned} \tag{2.149}$$

There are of course many variants to this basic architecture, e.g. g, u can be deep, or even convolutional neural networks.

The most important point to observe is that unlike deep neural networks we have seen before, although the forward inference of RNNs requires repeated application of transformations, in each "layer" in the τ direction, we always use the *same* set of parameters. In other words, we *share* parameters in the time-wise (τ) direction, just like how we share parameters in CNN in the space direction. For this reason, we differentiate this type of networks from classical deep networks, which we usually call *feed-forward* neural networks. In the case of RNNs, there is actually a loop in the computational graph, as illustrated in figure 2.16.

Training Recurrent Neural Networks. It turns out that training RNNs is almost the same as in feed-forward networks. We can simply *unroll* the graph in Figure 2.16 to a feed-forward-like structure, as shown in Figure 2.17. Again, notice that different from truly feed-forward

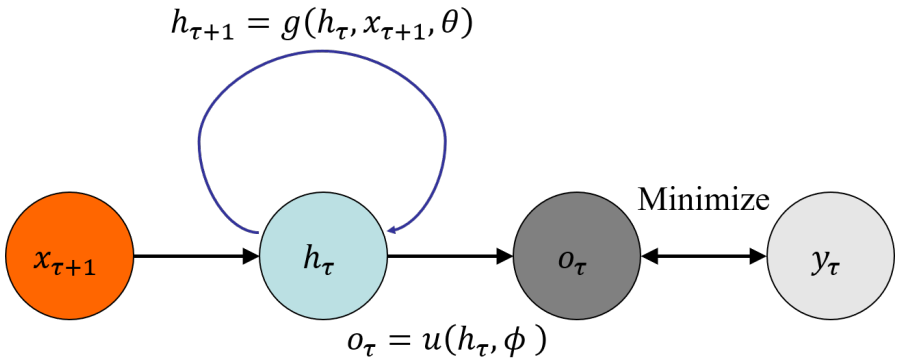


Figure 2.16: A basic RNN architecture. Note that unlike feed-forward networks, the graph has a cyclic node modelling the evolution of the hidden states $\{h_\tau\}$.

networks, the weights at every unrolled layer are tied, or shared. After unrolling, we can proceed exactly as before and use backpropagation to compute the gradients with respect to all parameters. This is known as *backpropagation through time* (BPTT) [Wer90].

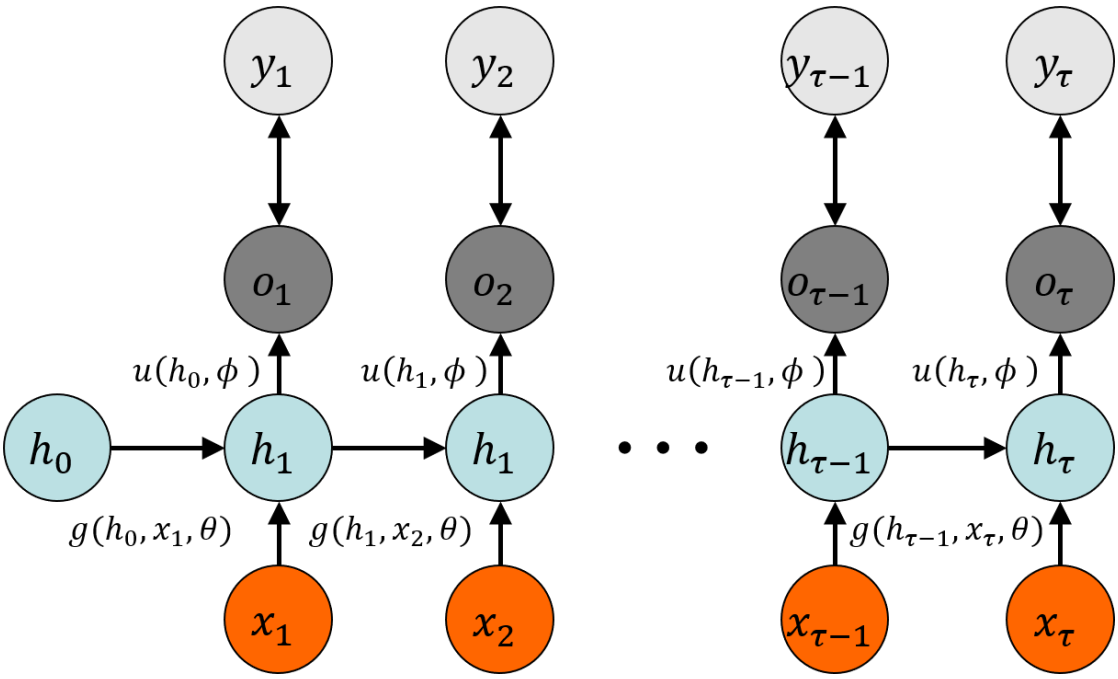


Figure 2.17: An unrolled RNN. It is important to note that the same parameters θ, ϕ are shared for every unrolled layer. This is unlike feed-forward networks.

2.8.3 Further Reading

As CNNs and RNNs saw a ton of development in recent years, it is impossible to give a full account of the detailed network architectures and theoretical underpinnings. The interested reader is referred to a modern reference [GBC16] as well as up-to-date literature on *ArXiV* for more thorough exposition. We will name some interesting topics in this area that is missing in these notes. First, we only discussed the simplest CNN components, but there are many important variants and improvements, including graph CNNs (see [WPC+19]) and attention mechanisms [VSP+17], just to name a few. Besides CNNs, there are also other methods to build invariance, such as scattering transform neural networks [Mal16]. On the side of *RNNs*, there are many more complex architectures to handle time series data, particularly those with long time dependence. These include recursive neural networks [Pol90, Bot14] and gated networks (LSTM, GRU) [HS97, CVMG+14]. They are very powerful models for time series data, particularly in the area of language modelling.

2.9 Basic Learning Theory

So far have we introduced various supervised learning models and discussed their approximation and optimization properties. We have also made some primarily empirical observation of their generalization performance. In this section, we introduce some basic concepts in learning theory that concretizes, in a mathematical sense, the problem of generalization and present some basic generalization bounds for simple learning scenarios. We begin with a review of relevant concepts in probability theory.

2.9.1 Review: Basic Concepts in Probability

In this section, we quickly review some basic concepts and notions in probability that are relevant to the rest of this section on learning theory.

Sets. The study of sets forms the basis of mathematics. This is especially so in probability and statistics, where sets and operations amongst sets play a fundamental role. Let Ω be set, which is a collection of "stuff". We recall the following notions

1. An *element* of Ω, ω is an object that is contained in Ω. We denote this as $\omega \in \Omega$. If ω is not contained in Ω, we write $\omega \notin \Omega$.
2. A set Ω is *empty* if there are no ω such that $\omega \in \Omega$. In that case we write $\Omega = \varnothing$ and call it an *empty set*.
3. A *subset* A of Ω, written as $A \subset \Omega$, is a set such that every $\omega \in A$ satisfies $\omega \in \Omega$.
4. Consider two subsets $A, B \subset \Omega$. Then, the *intersection* $A \cap B$ is the set $\{\omega : \omega \in A \text{ and } \omega \in B\}$. Their *union*, $A \cup B$ is $\{\omega : \omega \in A \text{ or } \omega \in B\}$.

5. The set $A \setminus B$ is the set $\{\omega : \omega \in A \text{ and } \omega \notin B\}$.
6. A and B are *disjoint* if $A \cap B = \emptyset$.
7. If Ω contains only a finite number of elements, we define $|\Omega|$ to be the number of elements. This can be extended to the general notion of *cardinality*, which applies to both finite and infinite sets.

Probability. An axiomatic study of probability started with the seminal work of Andrey Kolmogorov in the 1930s, building on the work of Émile Borel, Henri Lebesgue and others on set theory and measure theory. We will only review the main concepts without going into the abstract mathematical underpinnings:

1. Sample space: Ω (all possible events that can happen)
2. An event: $A \subset \Omega$ (some combination of things that happen)
3. Probability measure: A probability measure $\mathbb{P}$ assigns each combination of things happening a probability of it happening, i.e. $\mathbb{P}(A)$ is the probability that A happens. The probability measure satisfies the following properties:
   a) $\mathbb{P}(A) \in [0, 1]$ for any A.
   b) $\mathbb{P}(\Omega) = 1$.
   c) Let A_j, $j = 1, 2, \ldots$ be disjoint, then $\mathbb{P}(\cup_j A_j) = \sum_j \mathbb{P}(A_j)$.

   These properties are enough to derive all properties that you may know, such as law of large numbers and central limit theorems! This is the primary contribution of Kolmogorov's work, which gave the theory of probability a concrete axiomatic basis. Let us list some properties that we will need later below.
4. If $A \subset B$, then $\mathbb{P}(A) \leq \mathbb{P}(B)$
5. $\mathbb{P}(A \cup B) \leq \mathbb{P}(A) + \mathbb{P}(B)$. Note that by induction, this implies $\mathbb{P}(\cup_{j=1}^m A_j) \leq \sum_{j=1}^m \mathbb{P}(A_j)$ for all $m \geq 1$. This is known as the *union bound*, which will become useful later.
6. Random variables: random variables are functions from Ω into $\mathbb{R}^d$. The simplest way to think of them is to just treat $\Omega = \mathbb{R}^d$, in this case a random variable x is just a random vector in $\mathbb{R}^d$.
7. Distribution: for each random variable x, we can identify $\mathbb{P}$ with a probability measure μ on subsets of $\mathbb{R}^d$. We call it the *distribution* of x and write $x \sim \mu$. It has the property that the probability of $x \in A$ for some $A \subset \mathbb{R}^d$ is given by $\mu(A)$. We have the following integral representation

$$\mathbb{P}_{x\sim\mu}[x \in A] \equiv \mu(A) = \int_A d\mu \equiv \int_{\mathbb{R}^d} \mathbb{1}_{y\in A} d\mu(y) \tag{2.150}$$

8. Densities: If for any zero-volume set A, $\mu(A)$ is also 0, then we say that μ is *absolutely continuous with respect to the Lebesgue measure.* Essentially, this means that μ assigns small probabilities to sets of small volume, and contains no "lumps". In this case, we can show that μ has a representation via a *probability density* $\rho : \mathbb{R}^d \to \mathbb{R}_+$, such that

$$\mu(A) = \int_{\mathbb{R}^d} \mathbb{1}_{y \in A} \rho(y) dy. \tag{2.151}$$

   This is a consequence of the *Radon-Nikodym theorem* and basically says that $\mu(A)$ is like a weighted version of the volume of A, applying a weight to each point y equal to $\rho(y)$.

9. Expectation: Let $u : \mathbb{R}^d \to \mathbb{R}$ be any function. Then, the expectation of $u(x)$, denoted by $\mathbb{E}_{x \sim \mu}[u(x)]$ is the average value of $u(x)$ when x has distribution μ. Again, it has the integral representation

$$\mathbb{E}_{x \sim \mu}[u(x)] = \int_{\mathbb{R}^d} u(y) \mu(dy) \overset{\text{(if no lumps)}}{=} \int_{\mathbb{R}^d} u(y) \rho(y) dy. \tag{2.152}$$

   Notice that using this definition and point 8 above, we can see that

$$\mathbb{P}_{x \sim \mu}[x \in A] \equiv \mu(A) \equiv \mathbb{E}_{x \sim \mu}[\mathbb{1}_{x \in A}] \tag{2.153}$$

   In short, probabilities operates on sets, whereas expectations operates on functions. Probability can be seen as an expectation operating on the function which is the indicator function of the set.

10. Independence: two events A and B are independent if $\mathbb{P}(A \cap B) = \mathbb{P}(A)\mathbb{P}(B)$. Phrased in terms of random variables, x and x' are independently distributed if $\mathbb{P}[x \in A, x' \in B] = \mathbb{P}[x \in A]\mathbb{P}[x' \in B]$ for all $A, B \subset \mathbb{R}^d$. In words, the outcome of one of them does not affect that of the other in any way.

11. independently, Identically Distributed (i.i.d.): We say a collection of random variables $\{x_1, \ldots, x_N\}$ is i.i.d. with distribution μ if each $x_i \sim \mu$.

2.9.2 The PAC Framework

We now introduce a basic setting for developing learning theory, known as the *probably approximately correct* (PAC) framework. The main intuition behind developing this framework is the following conundrum: we want some precise measure of the performance of our model, say accuracy, on a test set, but

1. The training set is usually randomly sampled from some distribution, so our trained model is going to be random, and furthermore, the performance of a randomly generated test set is going to be random.

2. The accuracy for the worst case tends to be bad, but this is not representative. In fact, almost no model can achieve perfect accuracy.

The PAC framework resolves these issues by allowing us to ascertain that a model is performing well if it is correct up to some accuracy allowance (approximately) with high probability (probably). Let us now formalize this idea.

We denote by $\mathcal{X}$ the space of inputs and $\mathcal{Y}$ the space of labels. Recall that $f^* : \mathcal{X} \to \mathcal{Y}$ is an oracle function we would like to approximate using functions from a chosen hypothesis space $\mathcal{H}$. For simplicity, in this section we assume that $\mathcal{Y} = \{0, 1\}$, meaning that we have a binary classification problem, although the results presented can be extended to more general scenarios. We consider the slightly more general case where f^* itself belongs to a space of *concepts* C. In other words, we want to define learning notions that holds not for just one, but for a collection of target oracle functions.

Let us now consider a dataset $\mathcal{D} = \{x_i, y_i \equiv f^*(x_i)\}_{i=1}^N$ where each x_i is independent, identically distributed (i.i.d.) according to some distribution μ on $\mathcal{X}$. Recall that for each $f \in \mathcal{H}$, we can define the following risks using the zero-one loss

$$\text{Empirical Risk} \qquad R_{\text{emp}}(f) = \frac{1}{N} \sum_{i=1}^{N} \mathbb{1}_{f(x_i) \neq f^*(x_i)} \tag{2.154}$$

$$\text{Population/Expected Risk} \qquad R_{\text{pop}}(f) = \mathbb{E}_{x \sim \mu} \mathbb{1}_{f(x) \neq f^*(x)} \tag{2.155}$$

In practical learning, we can minimize the empirical risk, while in reality the expected risk controls the actual generalization capabilities of our model on unseen data. In the following, we shall use the notation $\mathbb{E}_{\mathcal{D} \sim \mu}$ to denote the expectation taken with respect to the dataset $\mathcal{D}$ where each x_i is i.i.d. with distribution μ.

Exercise 2.29

Show that for a *fixed* function f, $\mathbb{E}_{\mathcal{D} \sim \mu} R_{\text{emp}}(f) = R_{\text{pop}}(f)$. This shows that the empirical risk $R_{\text{emp}}(f)$, which is random, is an *unbiased* estimator of $R_{\text{pop}}(f)$, the population/expected risk. Is the i.i.d. assumption on $\{x_i\}$ necessary for this to hold?

Let us now define a notion of *learnability* under the PAC framework.

Definition 2.30: PAC-Learnability

We say a concept space C is PAC-learnable if there exists an algorithm $\mathcal{A}$ and a polynomial function $\text{poly}(\cdot,\cdot)$ such that for any $\epsilon, \delta > 0$, any distributions μ and any oracle $f^* \in C$, $\mathcal{A}$ returns an approximator $\widehat{f}$ such that as long as $N \geq \text{poly}(1/\epsilon, 1/\delta)$, we have

$$\mathbb{P}_{\mathcal{D}\sim\mu}[R_{\text{pop}}(\widehat{f}) \leq \epsilon] \geq 1 - \delta. \tag{2.156}$$

If further that $\mathcal{A}$ runs in $\text{poly}(1/\epsilon, 1/\delta)$, then C is said to be efficiently PAC-learnable and $\mathcal{A}$ is called a PAC-learning algorithm for C.

Remark. In a more general definition, the poly function can also depend on other properties pertaining to the concept space C and the input space $\mathcal{X}$, e.g. their complexities.

The most important point in the above is formula (2.156), which is the mathematical way of saying that our returned approximator, $\widehat{f}$, is probably (with probability $1-\delta$) and approximately (with error ϵ) correct when evaluated in terms of the population risk.

Let us emphasize certain key aspects of this definition:

1. There are no specific assumptions on the input distribution μ.
2. Training and testing samples are drawn from the *same* distribution.
3. PAC-learnability is a property of a concept space C.

2.9.3 Examples of PAC-learnability and PAC-learning Algorithms

In this section, we discussion some examples of PAC-learnable concept spaces as well as PAC-learning algorithms that can be constructed, together with how a formula like (2.30) may arise.

Learning Concentric Circles Under the Uniform Distribution. We begin with an example that is not quite what we want to achieve in the end. This is because we are going to assume that μ is a particular distribution, namely the uniform distribution. However, recall that we emphasized previously that PAC learnability statement should be distribution independent – it should hold for any μ. Nevertheless, this example will serve highlight the key insight that shows why we would expect a PAC-learning guarantee to hold for the current problem to be introduced, and similar problems.

We consider $\mathcal{X} = \mathbb{R}^2$ so that we are performing binary classification in the 2D plane. We shall assume that μ is the uniform distribution on the unit square $[-1/2, +1/2]^2$. This is convenient since for any set $A \subset [-1/2, +1/2]$ that we can ascribe a notion of area to (i.e. it is *Lebesgue*

measurable), we must have $\mathbb{P}_{x\sim\mu}[x \in A] = \text{Area}(A) = \int_A dy$. In simpler words, probability is just area under the uniform distribution – this fact will help us visualize things more easily.

Let us consider the concept space C to be indicator functions of disks in the unit square, i.e.

$$C = \left\{ f : f(x) = \mathbb{1}_{\|x\| \le r}, 0 \le r < 1/2 \right\}. \tag{2.157}$$

From the concept space we shall pick some oracle function f^* that generates our dataset $\mathcal{D} = \{x_i, y_i = f^*(x_i)\}_{i=1}^N$ with $x_i \sim \mu$. Figure 2.18 shows the oracle classifier $f^*(x) = \mathbb{1}_{\|x\| \le r^*}$ and its predictions on our dataset. Our goal is to find some $\widehat{f}$ from $\mathcal{H}$ (which we take to also be $= C$ in this example) that approximates f^* not just on the training dataset but also on a testing set produced from μ.

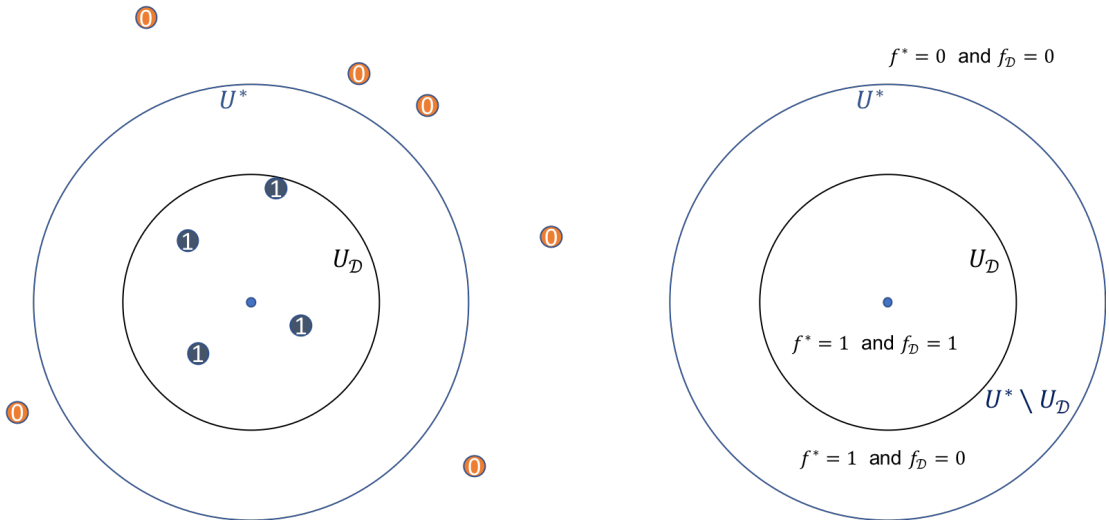


Figure 2.18: Illustration of the problem of learning concentric circles. f^* predicts everything within the disk U^* to be 1, and 0 otherwise. On the other hand, $\widehat{f}$ predicts everything in $U_\mathcal{D}$ to be 1, where $U_\mathcal{D}$ is the smallest circumscribing circle of the positive examples in the training set. On the right, we see that f^* and $\widehat{f}$ agree on all predictions except the annulus $U^* \setminus U_\mathcal{D}$.

Let us consider the following algorithm that gives us a candidate $\widehat{f}$: we find the smallest circle that circumscribes the training data for which $y_i = f^*(x_i) = 1$. Mathematically, we can write

$$\widehat{f}(x) = \mathbb{1}_{\|x\| \le r_\mathcal{D}}, \qquad r_\mathcal{D} = \max\{\|x_i\| : y_i = 1, i = 1, \ldots, N\}. \tag{2.158}$$

and in Figure 2.18 we also plot $\widehat{f}$. Notice the following important properties:

1. We can define $U^* = \{x : \|x\| \le r^*\}$ as the disk of radius r^* and $U_\mathcal{D}$ as the disk of radius $r_\mathcal{D}$. Then $f^* = \mathbb{1}_{U^*}$ and $\widehat{f} = \mathbb{1}_{U_\mathcal{D}}$.
2. $\widehat{f}$ always has zero training error, $R_{\text{emp}}(\widehat{f}) = 0$, since it classifies all training points correctly.

3. $\widehat{f}(x) \leq f^*(x)$ for all x, since $r_{\mathcal{D}} \leq r^*$.
4. The only region where $f^*(x) \neq \widehat{f}(x)$ is when $r_{\mathcal{D}} < \|x\| \leq r^*$, i.e. x belongs to the annulus $U^* \setminus U_{\mathcal{D}}$. Hence, we have

$$R_{\text{pop}}(\widehat{f}) = \mathbb{E}_{x\sim\mu} \mathbb{1}_{f^*(x)\neq\widehat{f}(x)} = \mathbb{P}_{x\sim\mu}[x \in U^* \setminus U_{\mathcal{D}}] = \text{Area}(U^* \setminus U_{\mathcal{D}}). \tag{2.159}$$

Now, let us derive a PAC learning statement for this problem. This amounts to finding some N large enough so that $R_{\text{pop}}(\widehat{f}) = \text{Area}(U^* \setminus U_{\mathcal{D}}) \leq \epsilon$ with probability $1 - \delta$, for any fixed $\epsilon, \delta > 0$. Note that this probability is with respect to the randomness in $\mathcal{D}$. Putting it another way, we can try to derive an upper-bound for $\mathbb{P}_{\mathcal{D}\sim\mu}[\text{Area}(U^* \setminus U_{\mathcal{D}}) > \epsilon]$.

Let us proceed in the following steps, which is also illustrated in Figure 2.19:

1. When is $\text{Area}(U^* \setminus U_{\mathcal{D}}) > \epsilon$?
   a) If $\text{Area}(U^*) \leq \epsilon$, then $\text{Area}(U^* \setminus U_{\mathcal{D}}) \leq \epsilon$. Thus, $\text{Area}(U^* \setminus U_{\mathcal{D}}) > \epsilon$ only if $\text{Area}(U^*) > \epsilon$.
   b) Now suppose $\text{Area}(U^*) > \epsilon$. Then, by continuity of the area, we can shrink U^* to a smaller disk U' of radius r' such that $\text{Area}(U^* \setminus U') = \epsilon$. Suppose that $U_{\mathcal{D}}$ contains U', then it is clear that the annulus $U^* \setminus U_{\mathcal{D}}$ is contained in $U^* \setminus U'$, which then implies $\text{Area}(U^* \setminus U_{\mathcal{D}}) \leq \text{Area}(U^* \setminus U') = \epsilon$. Hence, $\text{Area}(U^* \setminus U_{\mathcal{D}}) > \epsilon$ only if $U_{\mathcal{D}} \subset U'$. This happens only if $U_{\mathcal{D}} \cap (U^* \setminus U') = \emptyset$, i.e. $U_{\mathcal{D}}$ cannot intersect with an annulus with an area equal to ϵ.
2. Hence, we have

$$\begin{aligned}
&\mathbb{P}_{\mathcal{D}\sim\mu}[R_{\text{pop}}(\widehat{f}) > \epsilon] \\
&= \mathbb{P}_{\mathcal{D}\sim\mu}[\text{Area}(U^* \setminus U_{\mathcal{D}}) > \epsilon] \\
&\leq \mathbb{P}_{\mathcal{D}\sim\mu}[U_{\mathcal{D}} \cap (U^* \setminus U') = \emptyset] \\
&= \mathbb{P}_{x\sim\mu}[x \notin U^* \setminus U']^N && \text{[i.i.d. assumption]} \\
&= (1-\epsilon)^N && \text{[Area}(U^* \setminus U') = \epsilon\text{]} \\
&\leq \exp(-\epsilon N)\,. && \text{[}1 - z \leq \exp(-z)\text{]}
\end{aligned} \tag{2.160}$$

3. We want the right hand side of the above to be smaller than or equal to δ, so we can find a condition for N for this to be true by solving $\exp(-\epsilon N) \leq \delta$, which gives $N \geq \frac{1}{\epsilon} \log \frac{1}{\delta}$. Hence we have shown that for any $\delta, \epsilon > 0$, as long as $N \geq \frac{1}{\epsilon} \log \frac{1}{\delta}$, we have

$$R_{\text{pop}}(\widehat{f}) \leq \epsilon \text{ with probability of at least } 1 - \delta. \tag{2.161}$$

   This is almost a PAC-learning guarantee, except that we have assumed a specific distribution μ.

4. In fact, we can write the result (2.161) in another way. Let us just choose $N = \frac{1}{\epsilon}\log\frac{1}{\delta}$ so that (2.161) holds. Then, we can eliminate ϵ and obtain

$$|R_{\text{pop}}(\widehat{f}) - \underbrace{R_{\text{emp}}(\widehat{f})}_{=0}| \leq \frac{1}{N}\log\frac{1}{\delta}, \tag{2.162}$$

which holds with probability at least $1-\delta$. This is known as a bound on the *generalization gap*, or a *generalization bound*. These bounds are extremely useful in learning theory, because it tells us the effect of increasing the sample size on generalization performance. In words, it says that the generalization error of our model $\widehat{f}$, which always minimizes the empirical risk, decreases at a rate $O(N^{-1})$ as the number of samples N increases.

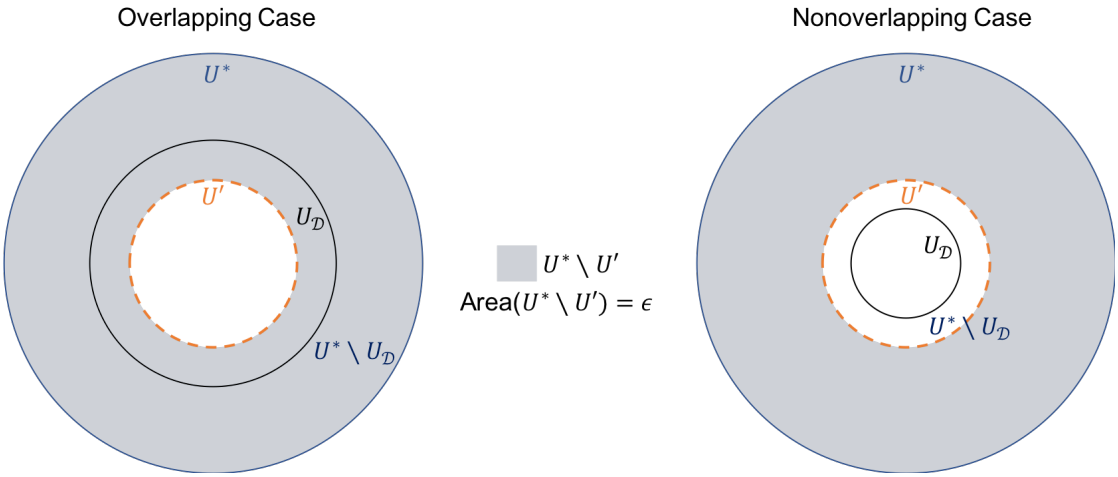


Figure 2.19: Illustration of the construction of U' for overlapping and non-overlapping cases.

Learning Concentric Circles in General The main reason why the previous does not constitute a strict PAC learning result is the fact that we assumed μ is the uniform distribution. In particular, PAC learning results should not depend on a specific choice of μ.

It turns out that this assumption is not at all necessary in the general case, by the following crucial observation: the uniform distribution allows us to associate probabilities with areas. In general, distributions are "weighted" areas, and obeys essentially the same rules. Hence, the previous arguments work exactly the same, when we just replace $\text{Area}(\cdot) = \mathbb{P}_{x\sim\mu}[x \in \cdot]$.

To simplify things, we shall slightly restrict μ to have no "lumps". In other words, we assume μ is such that $\mathbb{P}_{x\sim\mu}(x \in A) = 0$ if $\text{Area}(A) = 0$. Mathematically, we say that μ is *absolutely continuous with respect to the Lebesgue measure* (See Section 2.9.1). Not all distributions have probability densities, but they can all be approximated by those that do. We will come back to this point later to see how to relax this assumption to show the statement in entire generality.

We proceed as before and replace $\text{Area}(A)$ by $\mathbb{P}_{x\sim\mu}[x \in A]$, which we can denote by $\mu(A)$. The reason why we can do this is that since μ is absolutely continuous, $\mu(A)$ changes in a continuous

way as we shrink A, just like the area, so all previous arguments follow through without any changes.

This allows us to show

$$\begin{aligned}
&\mathbb{P}_{\mathcal{D}\sim\mu}[R_{\text{pop}}(\widehat{f}) > \epsilon] \\
&= \mathbb{P}_{\mathcal{D}\sim\mu}[\mu(U^* \setminus U_{\mathcal{D}}) > \epsilon] \\
&\leq \mathbb{P}_{\mathcal{D}\sim\mu}[U_{\mathcal{D}} \cap (U^* \setminus U') = \varnothing] \\
&= \mathbb{P}_{x\sim\mu}[x \notin U^* \setminus U']^N && \text{[i.i.d. assumption]} \\
&= (1-\epsilon)^N && [\mu(U^* \setminus U') = \epsilon] \\
&\leq \exp(-\epsilon N)\,. && [1 - z \leq \exp(-z)]
\end{aligned} \tag{2.163}$$

which implies the general PAC-learning result for learning concentric circles

$$R_{\text{pop}}(\widehat{f}) \leq \epsilon \text{ with probability of at least } 1-\delta, \tag{2.164}$$

as long as $N \geq \frac{1}{\epsilon}\log\frac{1}{\delta}$.

Remark. Finally, we discuss how the no-lumps assumption on μ can be relaxed. Clearly, if μ only gives non-zero probability to a finite number of points, then this assumption is not going to hold. However, there are two ways to extend the results above

1. We can replace the definition of r' not by the value exactly at which $\mu(D^* \setminus D') = \epsilon$ but by

$$r' = \sup\{r : \mathbb{P}_{x\sim\mu}[r \leq \|x\| \leq r^*] \geq \epsilon\} \tag{2.165}$$

   This is the same if μ has no lumps, but this supremum always exists whether or not μ has lumps. This allows us to carry out all the calculations as before exactly in the same way.

2. Alternatively, we can approximate μ by a sequence of μ_j which are lump-less, and $\mu_j \to \mu$ weakly, or in distribution. Since PAC learning bounds hold for any μ_j, it must also hold in the limit.

These generalizations require some technical details but introduces no new ideas over the simplified situation, and hence we can ignore them in these notes.

Learning Rectangles. As another example, let us consider the problem of learning rectangles instead of circular disks for classification. The setting is mostly identical as before, except now that f^* is an indicator function of a axes-aligned rectangle $[a,b]\times[c,d]$ (See Figure 2.20). As before, given a dataset $\mathcal{D}$ we pick the smallest rectangle that contains all the data and set the approximator $\widehat{f}$ to be the indicator function on the rectangle. The rectangle corresponding to f^* is denoted U^*, and that corresponding to $\widehat{f}$ is denoted $U_{\mathcal{D}}$.

We proceed in the same way as before, again assuming μ has no lumps and noting that this can be relaxed by sup/inf type arguments, or by limiting arguments. We upper-bound $\mathbb{P}_{\mathcal{D}\sim\mu}[R_{\text{pop}}(\widehat{f}) > \epsilon] = \mathbb{P}_{\mathcal{D}\sim\mu}[\mu(U^* \setminus U_{\mathcal{D}})]$ by the following steps:

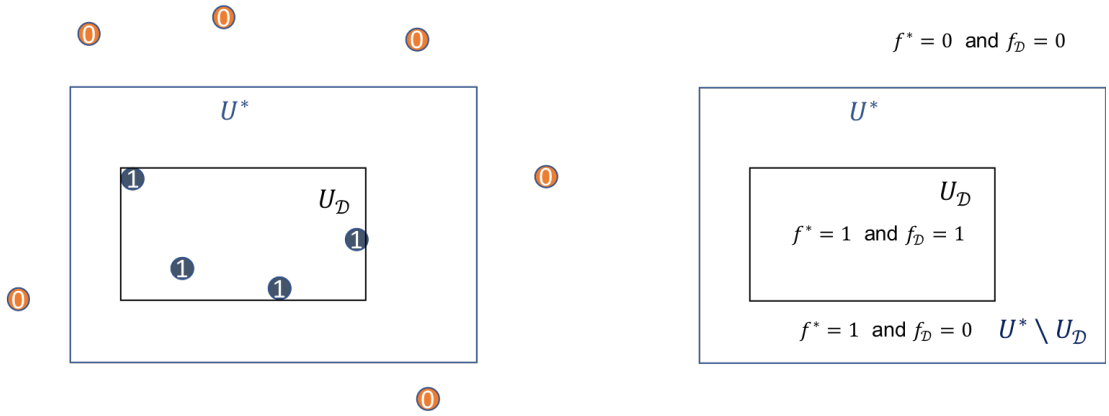


Figure 2.20: Illustration of the problem of learning rectangles. As before, f^* predicts everything within U^* to be 1, and 0 otherwise. On the other hand, $\widehat{f}$ predicts everything in $U_{\mathcal{D}}$ to be 1, where $U_{\mathcal{D}}$ is the smallest circumscribing square of the positive examples in the training set. On the right, we see that f^* and $\widehat{f}$ agree on all predictions except in the gap $U^* \setminus U_{\mathcal{D}}$.

1. When is $\mu(U^* \setminus U_{\mathcal{D}}) > \epsilon$?
   a) If $\mu(U^*) \leq \epsilon$, then $\mu(U^* \setminus U_{\mathcal{D}}) \leq \epsilon$. Thus, $\mu(U^* \setminus U_{\mathcal{D}}) > \epsilon$ only if $\mu(U^*) > \epsilon$.
   b) Now suppose $\mu(U^*) > \epsilon$. Then, by continuity, we can shrink U^* to a smaller rectangle, by shrinking all of its sides by forming 4 rectangles A_1, A_2, A_3, A_4 such that each of them has $\mu(A_j) = \epsilon/4$ (See Figure 2.21). If U^* overlaps with all of $A_1, \ldots, A_4$, then $U^* \setminus U_{\mathcal{D}}$ is contained in $\cup_j A_j$, and hence cannot have probability more than ϵ, since each of them only has probability $\epsilon/4$. This is a form of union bound. Hence, $\mu(U^* \setminus U_{\mathcal{D}}) > \epsilon$ only if $U_{\mathcal{D}}$ does not overlap with at least one of A_j.

2. Hence, we have

$$\begin{aligned}
&\mathbb{P}_{\mathcal{D}\sim\mu}[R_{\text{pop}}(\widehat{f}) > \epsilon] \\
&= \mathbb{P}_{\mathcal{D}\sim\mu}[\mu(U^* \setminus U_{\mathcal{D}}) > \epsilon] \\
&\leq \mathbb{P}_{\mathcal{D}\sim\mu}[\cup_{j=1}^{4}\{U_{\mathcal{D}} \cap A_j = \varnothing\}] \\
&\leq \sum_{j=1}^{4} \mathbb{P}_{\mathcal{D}\sim\mu}[\{U_{\mathcal{D}} \cap A_j = \varnothing\}] && \text{[Union bound]} \\
&\leq \sum_{j=1}^{4} \mathbb{P}_{\mathcal{D}\sim\mu}[x_i \notin A_j \text{ for all } i = 1, \ldots, N] \\
&= \sum_{j=1}^{4} \left(\mathbb{P}_{x\sim\mu}[x \notin A_j]\right)^N && \text{[i.i.d. assumption]} \\
&\leq 4\,(1-\epsilon/4)^N && [\mu(A_j) \geq \epsilon/4] \\
&\leq 4\exp\left(-\epsilon N/4\right). && [1 - z \leq \exp(-z)] \qquad (2.166)
\end{aligned}$$

3. We want the right hand side of the above to be smaller than or equal to δ, so we can find a condition for N for this to be true by solving $4\exp(-\epsilon N/4) \leq \delta$, which gives $N \geq \frac{4}{\epsilon}\log\frac{4}{\delta}$. Hence we have shown that for any $\delta, \epsilon > 0$, as long as $N \geq \frac{4}{\epsilon}\log\frac{4}{\delta}$, we have

$$R_{\text{pop}}(\widehat{f}) \leq \epsilon \text{ with probability of at least } 1-\delta. \qquad (2.167)$$

This is again a PAC-learning guarantee

4. As before, we can write (2.167) in another way by choosing $N = \frac{4}{\epsilon}\log\frac{4}{\delta}$, we can eliminate ϵ and obtain the generalization bound

$$|R_{\text{pop}}(\widehat{f}) - \underbrace{R_{\text{emp}}(\widehat{f})}_{=0}| \leq \frac{4}{N}\log\frac{4}{\delta}, \qquad (2.168)$$

which holds with probability at least $1-\delta$.

Exercise 2.31

Derive the PAC learning results for the concentric circles case, when instead of choosing $\widehat{f}$ as the smallest circle consistent with the data, we choose it as the largest one, i.e.

$$r_{\mathcal{D}} = \min\{\|x_i\| : y_i = 0, i = 1, \ldots, N\}. \qquad (2.169)$$

2.9.4 Generalization Bounds for Finite Hypothesis Space

We saw with previous examples that PAC-learning results and generalization bounds can be derived for simple cases where we chose a particular space of concepts C. It begs the question

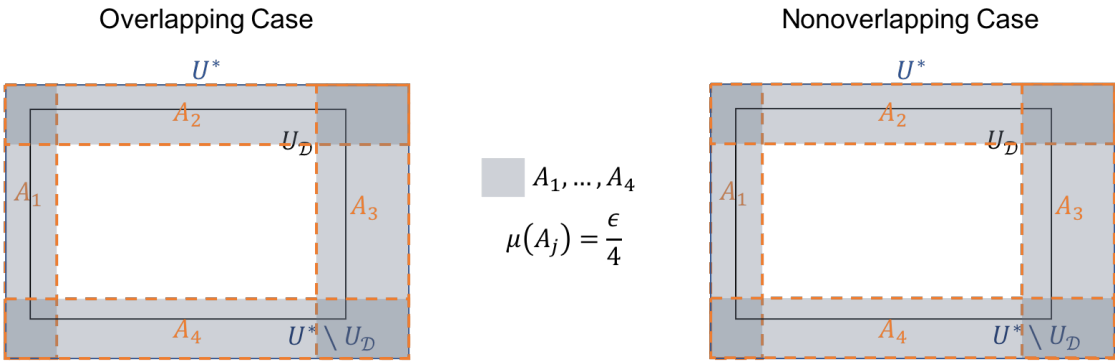


Figure 2.21: Illustration of the construction of $A_1, \ldots, A_4$ and the bounds for the overlapping and non-overlapping cases.

of whether we can do this for general concept spaces. In its full generality, this requires some more sophisticated tools from probability theory that is out of the scope of the course, but here we discuss a simple one that can be proven that relies on the following assumptions:

1. The hypothesis space $\mathcal{H}$ and the concept space C coincide, i.e. f^* always lies in our hypothesis space $\mathcal{H}$. In this case, the approximation error is always 0.
2. The hypothesis space $\mathcal{H}$ is finite, i.e. $|\mathcal{H}| < \infty$.
3. For any dataset $\mathcal{D}$, the algorithm $\mathcal{A}$ returns a *consistent* hypothesis, i.e. $R_{\text{emp}}(\widehat{f}) = 0$.

Under these assumptions, we can now show the following general result.

Theorem 2.32: PAC-Learning Guarantee

Let the assumptions above be satisfied. Then, for any $\delta, \epsilon > 0$, we have

$$\mathbb{P}_{\mathcal{D}\sim\mu}[R_{\text{pop}}(\widehat{f}) \leq \epsilon] \geq 1 - \delta \text{ if } N \geq \frac{1}{\epsilon}\left(\log|\mathcal{H}| + \log\frac{1}{\delta}\right). \tag{2.170}$$

Proof 2.32: PAC-Learning Guarantee

Define $\mathcal{H}_\epsilon := \{f \in \mathcal{H} : R_{\text{pop}}(f) > \epsilon\}$. By definition, for any $f \in \mathcal{H}_\epsilon$, we have

$$\begin{aligned}
\mathbb{P}_{\mathcal{D}\sim\mu}[R_{\text{emp}}(f) = 0] &= \mathbb{P}_{\mathcal{D}\sim\mu}[f(x_i) = f^*(x_i) \text{ for all } i = 1, \dots, N] \\
&= \left(\mathbb{P}_{x\sim\mu}[f(x) = f^*(x)]\right)^N \qquad \text{[i.i.d. assumption]} \\
&= \left(1 - \mathbb{P}_{x\sim\mu}[f(x) \neq f^*(x)]\right)^N \\
&< (1-\epsilon)^N \qquad \text{[Since } \mathbb{P}_{x\sim\mu}[f(x) \neq f^*(x)] = R_{\text{pop}}(f) > \epsilon\text{]} \qquad (2.171)
\end{aligned}$$

Observe that $R_{\text{pop}}(\widehat{f}) > \epsilon$ only if there exists a consistent $f \in \mathcal{H}_\epsilon$, i.e. $R_{\text{emp}}(f) = 0$. Hence, we have

$$\begin{aligned}
\mathbb{P}_{\mathcal{D}\sim\mu}[R_{\text{pop}}(\widehat{f}) > \epsilon] &\leq \mathbb{P}_{\mathcal{D}\sim\mu}[\cup_{f\in\mathcal{H}_\epsilon}\{R_{\text{emp}}(f) = 0\}] \\
&\leq \sum_{f\in\mathcal{H}_\epsilon} \mathbb{P}_{\mathcal{D}\sim\mu}[R_{\text{emp}}(f) = 0] \qquad \text{[Union bound]} \\
&\leq |\mathcal{H}_\epsilon|(1-\epsilon)^N \qquad \text{[Using (2.171)].} \\
&\leq |\mathcal{H}|(1-\epsilon)^N \qquad (2.172) \\
&\leq |\mathcal{H}|e^{-N\epsilon} \qquad \text{[}1 - z \leq \exp(-z)\text{].} \qquad (2.173)
\end{aligned}$$

Solving $|\mathcal{H}|e^{-N\epsilon} \leq \delta$ for N completes the proof. □

Remark. Equivalently, Theorem 2.32 gives the generalization bound

$$R_{\text{pop}}(\widehat{f}) \leq \frac{1}{N}\left(\log|\mathcal{H}| + \log\frac{1}{\delta}\right) \qquad (2.174)$$

which holds with probability at least $1 - \delta$. This is interesting: if we ignore the δ term, then the generalization error is controlled by $\log|\mathcal{H}|/N$. As before, if we increase the sample size, the generalization error drops. On the other hand, the general case says something more: suppose we have a fixed sample size, then increasing the complexity of our hypothesis space $\mathcal{H}$, meaning that we increase $|\mathcal{H}|$, then this error bound becomes bigger and bigger. Importantly, since this is an upper-bound, the bound itself being large does not imply that the actual generalization error is large. Instead, it means that this bound can tell us very little about the generalization error in the case where our hypothesis space is highly complex. In this case, we say that the bound is *vacuous*. The study of the generalization properties of models where classical bounds are vacuous is an intense area of research in machine learning theory today.

2.9.5 Further Reading

Again, we have only discussed the very basics of learning theory, which is a large subject warranting its own course(s). The interested reader may refer to reference [MRT18], Chapter

2 and 3, for a more thorough treatment. Note that much of the materials in this chapter is adapted from [MRT18], albeit with some simplifications. It is particularly useful to read on the generalizations of the basic results here to inconsistent cases, as well as infinite-dimensional hypothesis spaces. Modern research on dealing with cases where the bounds are vacuous can be found in [ZBH+16, DZPS18, ALS18, ALS18, LL18, ACH18, ADH+19, Coo18, OS19, CG19, OS18] and new papers on this topic come out very often.

3 Unsupervised Learning

3.1 Overview

Unsupervised learning represents a large set of problem statements in machine learning that does not involve approximating some input-label relationship. In particular, we are given a dataset $\mathcal{D} = \{x_i\}_{i=1}^N$ and our goal is to learn some often task-agnostic properties of the dataset. In some sense, unsupervised learning is any learning problem that is not supervised learning, so it contains a much richer class of problems. Below, we give some examples of unsupervised learning tasks:

1. *Dimensionality Reduction:* Suppose that each data point x_i is very high-dimensional, e.g. it represents a picture of very high resolution. Can we find a reduced representation of the data that still retains its key features, but lives a much lower dimensional space?
2. *Clustering:* Can we find clusters within the dataset, so that data points within clusters share higher similarities than across clusters?
3. *Density Estimation:* Suppose that each data point x_i is sampled from some underlying distribution μ^* which is unknown to us. Can we model μ^*, e.g. estimate a $\hat{\mu}$ from the data so that $\mu^* \approx \hat{\mu}$?
4. *Generative Models:* Similar to density estimation, we suppose $x_i \sim \mu^*$. However, instead of modelling μ^* explicitly, can we build a model that produces fresh samples x_i' that are approximately distributed according to μ^*?

These are of course not exhaustive, but they represent some main class of unsupervised learning formulations that are useful in practice. As in the supervised case, we shall start with linear models and work our way towards more complex and modern methodologies.

3.2 Principal Component Analysis

Principal component analysis, or PCA for short, is an incredibly useful method for dimensional reduction and compression of data. It is essentially an eigenvalue decomposition of a sample covariance matrix. For this reason, we will first review some associated concepts in linear algebra.

3.2.1 Review: Eigenvalues and Eigenvectors

In this section we will use the following basic properties of eigenvalues and eigenvectors. Readers unfamiliar with any of these properties should refer to standard linear algebra references, e.g. [GVL96].

1. For a real square matrix $(d \times d)$ A, an *eigenvector* of A with associated *eigenvalue* is a pair (u, λ) such that

$$Au = \lambda u. \tag{3.1}$$

   Here, u is a non-zero, d-dimensional vector and λ is a scalar.

2. We say that A is *diagonalizable* if there exists a diagonal matrix Λ and an invertible matrix P such that $A = P\Lambda P^{-1}$.

3. A is *symmetric* if $A^\top = A$.

4. A is *orthogonal* if $A^\top A = AA^\top = I$. Equivalently, $A^\top = A^{-1}$.

5. A well-known result is that if A is symmetric, then A is diagonalizable by orthogonal matrices with real eigenvalues, i.e. there exists an orthogonal U and real diagonal matrix Λ such that

$$A = U\Lambda U^\top. \tag{3.2}$$

   In particular, Λ is a diagonal matrix of eigenvalues and columns $\{u_j\}$ of U are the corresponding eigenvectors. By scaling, $\{u_j\}$ can be further made *orthonormal*, in the sense that

$$u_i^\top u_j = \delta_{ij} = \begin{cases} 1 & i = j, \\ 0 & \text{otherwise.} \end{cases} \tag{3.3}$$

6. A symmetric matrix A is *positive semi-definite* if $x^\top Ax \geq 0$ for all $x \in \mathbb{R}^d$. Equivalently, A is positive semi-definite if all eigenvalues of A non-negative. A is *positive definite* if $x^\top Ax > 0$, or equivalently if all eigenvalues are strictly positive.

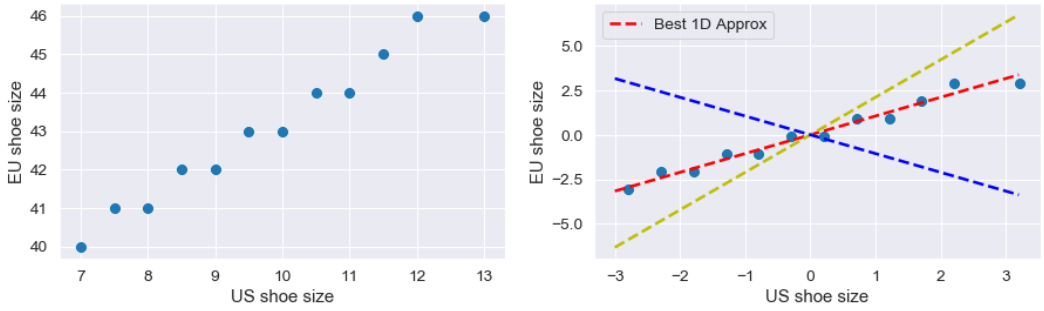
7. For a positive semi-definite matrix with eigen-decomposition $A = U\Lambda U^\top$, where the diagonal matrix of eigenvalues is $\Lambda = \text{diag}(\lambda_1, \ldots, \lambda_d)$, we will assume that the eigenvalues are arranged in decreasing order, i.e. $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_d \geq 0$.

3.2.2 Two Formulations of PCA

The primary problem of PCA is to discover, via a linear transformation, a reduced representation of a set of data $\mathcal{D} = \{x_i\}_{i=1}^N$. To begin with, let us consider an example of how to summarize a two-dimensional dataset in one dimension.

Example 3.1: Compressing a 2D Dataset

Let us look at a simple dataset with two variables comparing US and EU shoe sizes. Although there are two variables, it is clear even without looking at the graphs that there should be only one main direction of variation, the direction of increasing size!



Let us try to compress the centered data (right plot) to a one-dimensional subspace, meaning projecting it to a line passing through the origin. We can think of two criteria for choosing a line: we can either maximize the variation (variance) of the data along the direction of the line, or we can minimize the error incurred by projecting the data to the line. It appears in the plot that the red line is optimal in both senses.

As Example 3.1 shows, there are two considerations for how to compress/project the data: one to maximize the variance and one to minimize the error. In the example, they led to the same answer. It turns out that in general, this is also true. We now introduce these formulations and see how they relate to the eigenvalues and eigenvectors.

Maximize Variance Formulation. Suppose that $x_i \in \mathbb{R}^d$ for $i = 1, \ldots, N$ and we want to find a linear transformation from $\mathbb{R}^d$ to $\mathbb{R}^m$ with $m \leq d$ (and often much smaller). Without loss of generality, we shall assume that the data is centered as follows: For each $j = 1, \ldots, d$, define

$$\overline{x}_j = \frac{1}{N} \sum_{i=1}^{N} x_{ij} \tag{3.4}$$

to be the sample mean of the j^{th} dimension of the data. We write $\overline{x} \in \mathbb{R}^d$ as the vector of samples means with its j^{th} coordinate as the above. We assume that $\overline{x} = 0$. This can always be achieved by subtracting $\overline{x}$ from each x_i. Now, the sample covariance matrix of this dataset is

$$S = \frac{1}{N} \sum_{i=1}^{N} x_i x_i^\top \qquad \text{i.e.} \qquad S_{jk} = \frac{1}{N} \sum_{i=1}^{N} x_{ij} x_{ik}, \qquad j, k = 1, \ldots, d, \tag{3.5}$$

Now, we start by considering the case $m = 1$, i.e. we are transforming to a one-dimensional space. Then, a general linear transformation of x_i to a scalar is given by

$$z_i = u^\top x_i, \qquad u \in \mathbb{R}^d, \|u\| = 1. \tag{3.6}$$

Note that we assumed $\|u\| = 1$, i.e. u is a unit vector, so that this corresponds to a projection map. We want to choose u so that the sample variance of $\{z_i\}_{i=1}^N$ is maximized. We can compute the sample variance as follows

$$\text{Var}(\{z_i\}) = \frac{1}{N}\sum_{i=1}^{N}(z_i - \overline{z})^2 = \frac{1}{N}\sum_{i=1}^{N} u^\top x_i u^\top x_i = u^\top S u \tag{3.7}$$

Hence, the optimal transformation u_1 is given by the optimization problem

$$u_1 = \underset{u\in\mathbb{R}^d,\,\|u\|=1}{\arg\max}\; u^\top S u. \tag{3.8}$$

How do we solve this problem? It should be intuitively clear from the interpretation of eigenvalues that u_1 should be the the normalized eigenvector corresponding to the largest eigenvalue. We can check that this is in fact the case using the method of Lagrange multipliers.

Writing the constraint $\|u\|^2 = u^\top u = 1$ in terms of Lagrange multipliers, we have an unconstrained maximization problem

$$u^\top S u^\top + \lambda(1 - u^\top u) \tag{3.9}$$

Differentiating with respect to u gives

$$Su = \lambda u, \tag{3.10}$$

which means that any stationary point of our problem must be an eigenvector with eigenvalue λ. To see which is a maximum, note that if $Su = \lambda u$, then

$$u^\top S u = \lambda u^\top u = \lambda. \tag{3.11}$$

Thus, the variance is maximized if $\lambda = \lambda_1$, the largest eigenvalue of S. Recall that for a symmetric positive definite matrix S, λ_1 denotes the largest eigenvalue. Hence, the projection map is $x \mapsto u_1^\top x$ with u_1 the eigenvector corresponding to the eigenvalue λ_1. We call $\{z_i = u_1^\top x_i\}$ the *first principal component score* of the dataset $\mathcal{D}$, with u_1 the *first principal component axis*.

The preceding argument generalizes to higher output dimensions $m > 1$ as follows. For $m = 2$, we want to first find the principal component u_1 that maximize variance as before, and then we can try to find a u_2 such that $u_2^\top x$ has maximal variance and that $u_2^\top u_1 = 0$. That is, u_2 is a direction that is *independent* from the first principal component axis and has maximal variance. Following similar reasoning we can show that u_2 is the eigenvector corresponding to λ_2, the second largest eigenvalue of S. The quantities $\{u_2^\top x_i\}$ are called the second principal component score of $\mathcal{D}$, with second principal component axes u_2.

In general, the first m principal component scores are given by a matrix $Z_m \in \mathbb{R}^{N\times m}$ given by

$$Z_m = XU_m, \tag{3.12}$$

with the i^{th} row of X (data matrix) being x_i and $U_m \in \mathbb{R}^{d\times m}$, with its j^{th} column equal to u_j, the eigenvector corresponding to the j^{th} largest eigenvalue λ_j of the covariance matrix S of the data. This is a matrix of principal component axes.

Minimize Error Formulation. Alternatively, we can formulate PCA as a "compression" algorithm, in which we want to find a projection map to a lower dimensional space while minimizing the error incurred by the projection. Let us consider an orthonormal basis $\{u_j : j = 1, \dots, d\}$ for $\mathbb{R}^d$. Then, each data point x_i can be represented as a linear combination

$$x_i = \sum_{j=1}^{d} \alpha_{ij} u_j = \sum_{j=1}^{d} (u_j^\top x_i) u_j. \tag{3.13}$$

Suppose now that we want to project x_i onto a m-dimensional space spanned by the first m basis vectors $\{u_1, \dots, u_m\}$, so that

$$z_i = \sum_{j=1}^{m} \beta_{ij} u_j \tag{3.14}$$

Our goal is to choose $\{\beta_{ij}\}$ and $\{u_j\}$ to minimize the error

$$\frac{1}{N} \sum_{i=1}^{N} \|x_i - z_i\|^2, \tag{3.15}$$

which we can rewrite, due to the orthonormal condition $u_j^\top u_k = \delta_{jk}$, as

$$\frac{1}{N} \sum_{i=1}^{N} \|x_i - z_i\|^2 = \frac{1}{N} \sum_{i=1}^{N} \left[\sum_{j=1}^{m} (\alpha_{ij} - \beta_{ij})^2 + \sum_{j=m+1}^{d} \alpha_{ij}^2 \right] \tag{3.16}$$

Hence, we can minimize the first term by choosing $\beta_{ij} = \alpha_{ij} = u_j^\top x_i$, making it 0. Then, the error is measured by

$$\frac{1}{N} \sum_{i=1}^{N} \sum_{j=m+1}^{d} \alpha_{ij}^2 = \frac{1}{N} \sum_{i=1}^{N} \sum_{j=m+1}^{d} (u_j^\top x_i)^2 = \sum_{j=m+1}^{d} u_j^\top S u_j. \tag{3.17}$$

Finally, we need to choose $\{u_j\}$ in such a way that $\sum_{j=m+1}^{d} u_j^\top S u_j$ is minimized, while preserving orthonormality. It is not hard to see that $\{u_j\}$ should be chosen as the set of eigenvectors corresponding to the eigenvalues of S, ordered in decreasing eigenvalue.

For example, let us consider the case when $d = 2$ and $m = 1$. Then, we have the Lagrange's problem

$$\min_{u \in \mathbb{R}^2} u^\top S u + \lambda(1 - u^\top u), \tag{3.18}$$

whose solution gives us u_2. Differentiating gives

$$Su = \lambda u \tag{3.19}$$

and so u is an eigenvector of S with eigenvalue λ, and

$$u^\top S u = \lambda. \tag{3.20}$$

This is minimized if we pick $\lambda = \lambda_2$, the minimal eigenvalue of S and $u = u_2$ is the corresponding eigenvector. Consequently, our projection map is onto the complement of the subspace spanned by u_2, which by orthogonality is u_1, the eigenvector corresponding to eigenvalue λ_1.

In general, we can show that the minimum error is achieved if we pick $\{u_j\}$ to be the set of orthonormal eigenvectors of the covariance matrix S, ordered in decreasing eigenvalues. Consequently, the transformation that incur the minimal error is given by the mapping

$$x \mapsto \sum_{j=1}^{m} (u_j^\top x) u_j \tag{3.21}$$

Applying this transformation to each data x_i, in matrix form we have the mapping

$$X \mapsto X U_m U_m^\top, \tag{3.22}$$

and the score $Z_m = XU_m$ is the coefficients corresponding to the principal components, as in (3.12).

3.2.3 The PCA Algorithm

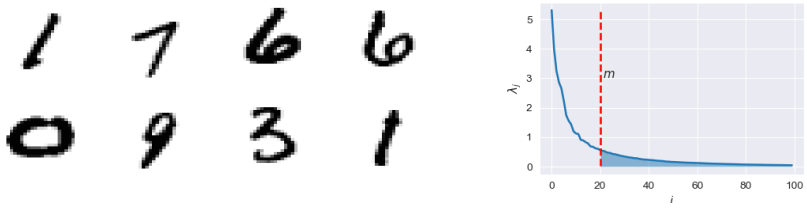
With these interpretations in mind, we summarize the standard PCA algorithm for dimensionality reduction in Alg. 6. One point of discussion that remains is how to choose m. In turns out that one can usually compute a full eigen-decomposition of the covariance matrix S and observe the decay of the eigenvalues. A cut-off can be suitably determined by studying the eigenvalues. The following example illustrates this.

Algorithm 6: Principal Component Analysis

Data: $\mathcal{D} = \{x_i\}_{i=1}^N$, $x_i \in \mathbb{R}^d$ for all i
Hyperparameters: m (reduction dimension)
Compute sample covariance matrix $S = \frac{1}{N} \sum_{i=1}^N x_i x_i^\top$;
Compute the first m eigenvectors $\{u_1, \ldots, u_m\}$ and eigenvalues $\{\lambda_1, \ldots, \lambda_m\}$;
Form $d \times m$ matrix U_m whose j^{th} column is u_j ;
Compute $Z_m = XU_m$;
return *Principal component scores Z_m. Eigenvalues and eigenvectors λ_j, u_j for $j = 1, \ldots, m$*

Example 3.2: Determining m in PCA

Let us consider the MNIST dataset of hand-written digits. Performing a PCA on the dataset ($d = 784$) we obtain a eigenvalue distribution as shown in the following figure. Notice the decay of the eigenvalues of the sample covariance matrix (right plot). We can choose m to be sufficiently large so that the projection error $\sum_{j=m+1}^{d} \lambda_j$ (shaded area) is small enough.



Numerical Methods for PCA. In Algorithm 6, it is required to compute the top m eigenvalues and eigenvectors for the $d \times d$ covariance matrix S. One way to do it numerically is to compute a full eigen-decomposition of S, which requires $O(d^3)$ operations. For large d and the case that we know m, a better way is to only compute the top m eigenvalues and eigenvectors. This can be achieved by the power method and its variants [TBI97], which have complexity $O(md^2)$, which represents a large amount of savings if $m \ll d$.

Another case of interest is when $N \ll d$, i.e. where we are dealing with extremely high dimensional problems. Then, observe that eigen-decompositions are extremely expensive. Let us see how we can rewrite the eigenvalue problem

$$X^\top X u_i = S u_i = \lambda_i u_i \tag{3.23}$$

Multiply X to both sides of the above, we get

$$X X^\top (X u_i) = \lambda_i (X u_i) \tag{3.24}$$

We can then define $v_i = X u_i$. Notice that each $v_i \in \mathbb{R}^N$ and we obtain an eigenvalue problem in $\mathbb{R}^N$ instead of $\mathbb{R}^d$, as follows

$$X X^\top v_i = \lambda_i v_i \tag{3.25}$$

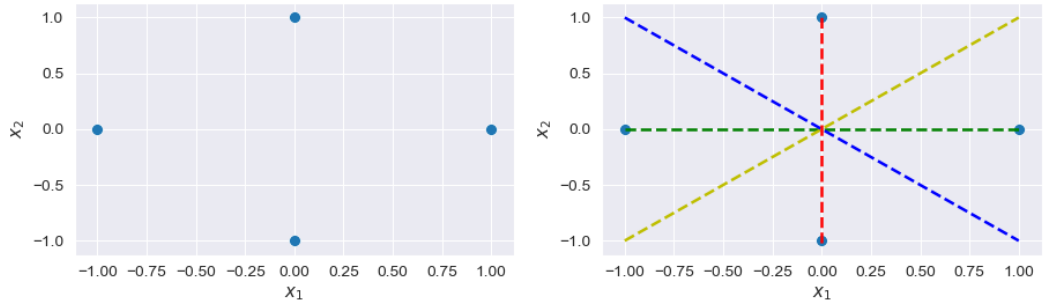
Furthermore, the eigenvalue problems (3.23) and (3.25) have the same eigenvalues. Thus, we can solve (3.25) in place of (3.23). The former can work in N dimensions instead of d, and is useful if $d \gg N$.

3.2.4 PCA in Feature Space

To motivate an extension of the usual PCA, we first give an example of a dataset where a naïve PCA is insufficient in discovering low-dimensional embeddings of the dataset.

Example 3.3: Circular Clusters

Consider the dataset plotted below in the left plot.



A direct application of PCA cannot reduce this dataset to one effective dimension. Indeed, the sample covariance matrix is $S = \text{diag}(1, 1)/2$. Giving principal directions aligned with the coordinate axes, but no dominant eigenvalues. See right plot.

How do we come up with an extension of PCA to handle such cases? As with passing from linear models to linear basis models, the trick here is that we can work with feature maps. Just like in Section 2.2.2, we define a vector of feature maps

$$\phi(x) = (\phi_1(x), \dots, \phi_M(x)) \tag{3.26}$$

Then, we can perform PCA in feature space, in which case the design matrix $\Phi_{ij} = \phi_j(x_i)$ then plays the role of the data matrix X. More concretely, let us assume without loss of generality that Φ is again centered with 0 sum for each column (otherwise, center it before proceeding). Then, define the sample covariance matrix in feature space as

$$S_\phi = \frac{1}{N} \sum_{i=1}^{N} \phi(x_i)\phi(x_i)^\top. \tag{3.27}$$

Note that S_ϕ is a $M \times M$ matrix, whereas S was a $d \times d$ matrix. Then, we can perform PCA entirely as before, by performing the eigen-decomposition

$$S_\phi = U \Lambda U^\top, \tag{3.28}$$

where U is a $M \times M$ matrix whose columns are the orthonormal eigenvectors of S_ϕ and Λ is a diagonal matrix of eigenvalues. The nonlinear principal component scores are given by

$$Z = \Phi U \tag{3.29}$$

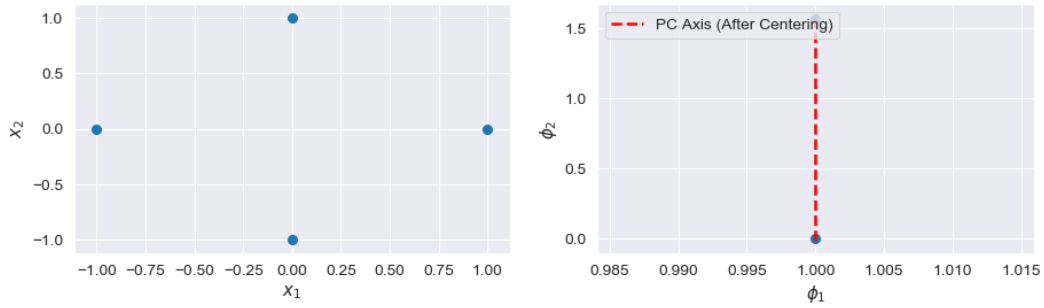
We summarize this algorithm in Alg. 7.

Algorithm 7: Principal Component Analysis in Feature Space

Data: $\mathcal{D} = \{x_i\}_{i=1}^N$, $x_i \in \mathbb{R}^d$ for all i
Hyperparameters: m (reduction dimension), ϕ (feature maps)
Compute design matrix $\Phi_{ij} = \phi_j(x_i)$;
Center design matrix $\Phi_{ij} \leftarrow \Phi_{ij} - \sum_i \Phi_{ij}/N$;
Compute sample covariance matrix $S_\phi = \frac{1}{N}\Phi^\top\Phi$;
Compute the first m eigenvectors $\{u_1, \ldots, u_m\}$ and eigenvalues $\{\lambda_1, \ldots, \lambda_m\}$ of S_ϕ ;
Form $d \times m$ matrix U_m whose j^{th} column is u_j ;
Compute $Z_m = \Phi U_m$;
return *Principal component score* Z_m. *Eigenvalues and eigenvectors* λ_j, u_j *for* $j = 1, \ldots, m$

Example 3.4: Circular Clusters Revisited

Let us return to Example 3.3, but consider the feature maps $\phi_1(x) = (x_1^2 + x_2^2)^{1/2}$ and $\phi_2 = |\tan^{-1}(x_2/x_1)|$. Then, we have $S_\phi = \text{diag}(0, \pi^2/16)$ after centering, which gives a principal component in the second coordinate axis in (ϕ_1, ϕ_2) plane that perfectly summarizes the data (see right plot).



Remark. Just like in Section 2.3, once we begin to work with feature maps, we can then attempt to write the PCA formulation entirely in terms of dot products $\phi(x)^\top\phi(x)$, upon which we can then turn to a kernel formulation. This is in fact possible with PCAs, which allows us to work only implicitly with feature maps. This is the well-known kernel PCA (KPCA) algorithm [SSM97].

3.2.5 PCA as a Form of Whitening

Finally, we discuss how PCA can be used as a form of normalization for data, known as *whitening*. Let us stick with the original space and not feature space, so we have the data matrix X whose i^{th} row is x_i. Recall that the principal components are given by

$$Z = XU, \tag{3.30}$$

where U is the eigenvector matrix of the sample covariance matrix S. As before, let us denote by Λ the diagonal matrix of eigenvalues. Let us assume that the data is such that S is invertible, and hence the eigenvalues are all positive. This is usually the case if $N \geq d$ and the data are linearly independent. Now, define the transformation

$$X \mapsto X' = XU\Lambda^{-1/2} = Z\Lambda^{-1/2}. \tag{3.31}$$

Note that this is a linear transformation on the dataset, so clearly the sample mean of X' is still 0. Let us check its covariance:

$$\begin{aligned} \frac{1}{N}\sum_{i=1}^{N} x_i' x_i'^{\top} &= \frac{1}{N} X'^{\top} X' = \frac{1}{N}(XU\Lambda^{-1/2})^{\top}(XU\Lambda^{-1/2}) \\ &= \frac{1}{N}(XU\Lambda^{-1/2})^{\top}(XU\Lambda^{-1/2}) = \Lambda^{-1/2}U^{\top}SU\Lambda^{-1/2}. \end{aligned} \tag{3.32}$$

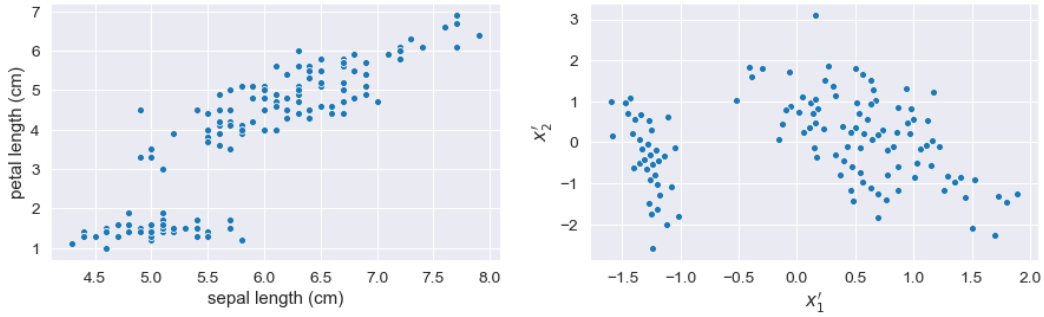
But $S = U\Lambda U^{\top}$ and so $U^{\top}SU = \Lambda$, therefore

$$\frac{1}{N}\sum_{i=1}^{N} x_i' x_i'^{\top} = \Lambda^{-1/2}\Lambda\Lambda^{-1/2} = I \tag{3.33}$$

Hence, the transformed dataset X' has mean 0 and covariance I, i.e. it is transformed into independent and normalized components. This is known as *whitening* or *sphering*.

Example 3.5: Data Whitening

We consider the Iris Petals dataset [Fis36] shown on the left, which plots sepal and petal lengths of three iris species, *Iris setosa*, *Iris virginica* and *Iris versicolor*. We can clearly see a correlation between the two variables. Now, we apply the whitening transformation based on PCA described above and the results are shown on the right plot. Observe that the two components x_1', x_2' are now uncorrelated.



3.2.6 Autoencoders

To motivate autoencoders, we start by interpreting PCA as a lossy encoding-decoding algorithm. Recall upon computing eigenvectors and eigenvalues of the sample covariance matrix, we have

the principal component scores

$$Z = XU. \tag{3.34}$$

Since U is orthogonal, we can invert this to obtain

$$X = ZU^\top \tag{3.35}$$

Now, if we insist on a dimensionality reduction, then we cannot use the full eigenvector matrix U, but only U_m, the first m columns of U. Then, the first m principal components is the matrix

$$Z_m = XU_m \tag{3.36}$$

We can approximately invert this by

$$X_m = Z_m U_m^\top. \tag{3.37}$$

Just like our original data matrix X, $X_m \in \mathbb{R}^{N\times d}$. However, if $m < d$ then it is in general not true that $X_m = X$. In particular, observe that if $N \geq d$ and if X has independent columns, then X has rank d. On the other hand, X_m is of rank at most m, and hence cannot be equal to X. Nevertheless, X_m represents an approximation of X, and we know that the error of the approximation is the sum of the omitted eigenvalues $\sum_{i=m+1}^{d} \lambda_i$. The approximation is good if the eigenvalues have disparate magnitudes so that the first m eigenvalues dominate the rest.

The low-dimensional representation Z_m of X is known as a *latent representation*, simply *latents*. That is, each row of Z_m is a reduced representation of each row of X, which corresponds to an input data. In this case, the latent space has dimension m. The PCA maps each input $x \in \mathbb{R}^d$ into the latent space $\mathbb{R}^m$ by the encoding map

$$T_{\text{enc}} : \mathbb{R}^d \to \mathbb{R}^m \qquad T_{\text{enc}}(x)_j = x^\top u_j = (U_m^\top x)_j, \qquad j = 1, \ldots, m. \tag{3.38}$$

For each latent $z \in \mathbb{R}^m$, we can approximately recover its original representation by the decoding map

$$T_{\text{dec}} : \mathbb{R}^m \to \mathbb{R}^d \qquad T_{\text{dec}}(z)_j = (U_m z)_j, \qquad j = 1, \ldots, m. \tag{3.39}$$

Figure 3.1 illustrates this encoding-decoding procedure.

It is clear from Example 3.3 that linear transformations may not be sufficient in finding efficient latent representations. *Autoencoders* builds on this idea to derive a nonlinear analogue. Here, we consider two general parameterized mappings

$$T_{\text{enc}}(\cdot;\theta) : \mathbb{R}^d \to \mathbb{R}^m \qquad T_{\text{dec}}(\cdot;\phi) : \mathbb{R}^m \to \mathbb{R}^d \tag{3.40}$$

representing a general, possibly nonlinear encoding and decoding map. Under these mappings, for each input x we can obtain the latents $z = T_{\text{enc}}(x;\theta)$, which we can then decode to obtain $x' = T_{\text{dec}}(z;\phi)$.

These maps depend on adjustable parameters θ, ϕ, which can be optimized by solving the empirical risk minimization problem which aims to minimize the distance between the reconstruction x_i' of each sample x_i

$$\min_{\theta,\phi} \frac{1}{N} \sum_{i=1}^{N} \|x_i - T_{\text{dec}}(T_{\text{enc}}(x_i;\theta);\phi)\|^2 \tag{3.41}$$

It remains to discuss how to parameterize the encoder and decoder functions. One popular choice is to use neural networks. For example, we can use a one-hidden-layer shallow neural network with q hidden units, we then obtain the encoder function

$$T_{\text{enc}}(x;\theta) = A\sigma(Wx + b), \qquad \theta = (A, W, b) \in \mathbb{R}^{m\times q} \times \mathbb{R}^{q\times d} \times \mathbb{R}^q. \tag{3.42}$$

Similarly, the decoder function is

$$T_{\text{dec}}(x;\theta) = B\sigma(Vx + c), \qquad \theta = (B, V, c) \in \mathbb{R}^{d\times q} \times \mathbb{R}^{q\times m} \times \mathbb{R}^q. \tag{3.43}$$

The empirical risk minimization problem (3.41) can be solved by stochastic gradient descent, which trains both the encoder parameters θ and the decoder parameters ϕ concurrently. This is known as the autoencoder and its architecture is illustrated in Figure 3.1. Of course, in general we can replace the encoder and decoder functions by any other function approximators, including deep neural networks.

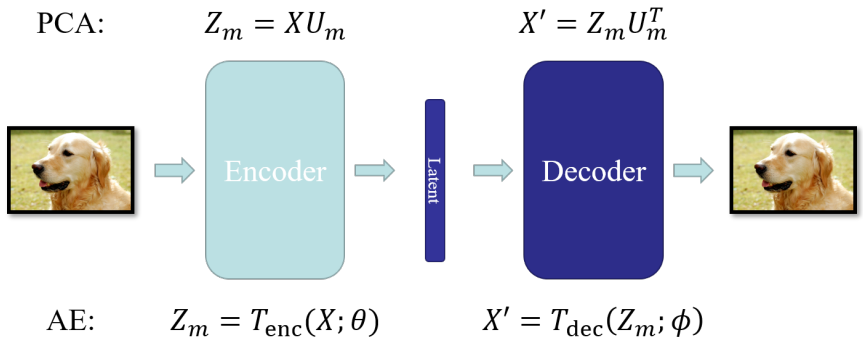


Figure 3.1: Illustration of both PCA and AE as compression-decompression algorithms involving latent representations.

3.2.7 Further Reading

In the simple formulation discussed here we focused on the eigen-decomposition of the covariance matrix. In fact, another perhaps more natural way to interpret PCA is that it is an singular value decomposition (SVD) of the data matrix X. In fact, most state-of-the-art numerical algorithms for PCA performs efficient SVD on X without necessarily computing the covariance matrix [GVL96]. Note that PCA has been independently discovered in many different fields.

Hence, they may come in many names. For example, in the dynamical systems literature, PCA is known as the *Karhunen–Loève transform* (KLT) [Kar47] or the *proper orthogonal decomposition* (POD) [Cha00]. In introducing the PCA in feature space, we have alluded to the kernel version of the PCA. This is useful when a good kernel can be identified on physical grounds. More information on kernel PCA can be found in [SSM97]. There are also many probabilistic variants of PCA, such as maximum likelihood PCA, Bayesian PCA, etc. See Chapter 12 of [BO06]. Finally, there are also many variants on the autoencoder architecture, including regularized or sparse autoencoders, as well as their probabilistic extensions. See Chapter 14 of [GBC16].

3.3 Clustering and Gaussian Mixture Models

Cluster analysis, or simply clustering, is a commonly encountered task in unsupervised learning. Here our goal is to partition the input data into several groups, such that samples within the same group exhibit higher similarity than samples across groups. Applications of this is abound in practice, ranging from feature extraction, data compression to image segmentation. In this section, we first introduce a simple clustering algorithm called K-means clustering, and then discuss a probabilistic extension of it using Gaussian mixture models.

3.3.1 K-means Clustering

Clustering is a widely used technique in unsupervised learning to identify groupings of data based on some similarity measure. Suppose we have a dataset $\mathcal{D} = \{x_i\}_{i=1}^N$ of vectors in $\mathbb{R}^d$. Our goal is to partition $\mathcal{D}$ into K groups, and we will assume that K is unknown to us. We proceed in the following steps

1. Identify representative of the clusters $z_1, \dots, z_K \in \mathbb{R}^d$.
2. Assign each data point x_i to the k^{th} cluster whose representative is z_k.

To formalize the assignment step, we introduce the notation

$$r_{ik} = \begin{cases} 1 & x_i \text{ is assigned to cluster } k, \\ 0 & \text{otherwise.} \end{cases} \tag{3.44}$$

We denote by R the matrix with entries $\{r_{ij}\}$. This is known as the *1-of-K coding scheme*. Now, we can introduce a measure of *distortion* of our assignment scheme:

$$J(R, Z) = \frac{1}{2N} \sum_{i=1}^{N} \sum_{k=1}^{K} r_{ik} \|x_i - z_k\|^2. \qquad Z \in \mathbb{R}^{K \times d}, \quad Z_{kj} = z_{kj} \tag{3.45}$$

This is simply the sum of squares of distances from each sample point x_i to the representative z_k that it is assigned to. It remains to minimize the distortion or loss J with respect to both the representatives or centers Z and the assignment scheme R. It is easier to proceed iteratively as follows:

1. Suppose we know Z. Then, R is determined by assigning each x_i to the nearest z_k. That is

$$r_{ik} = \begin{cases} 1 & k = \arg\min_j \|x_i - z_j\|^2 \\ 0 & \text{otherwise.} \end{cases} \tag{3.46}$$

   Ties are broken arbitrarily (e.g. take k to be the smallest index value of the minimizing indices).

2. Suppose we know R. Then, the loss function J is a quadratic function of Z, which we can find its minimum by setting its derivative to zero. We have

$$\nabla_z J(R, z) = \frac{1}{N} \sum_{i=1}^{N} r_{ik}(x_i - z_k) = 0, \tag{3.47}$$

   giving

$$z_k = \frac{\sum_{i=1}^{N} r_{ik} x_i}{\sum_{i=1}^{N} r_{ik}}. \tag{3.48}$$

   In other words, z_k is the average value of x_i that has been assigned to the cluster represented by z_k. This motivates the name *K-means* algorithm and we summarize the algorithm in Alg. 8. There are many possible stopping criteria, e.g. check convergence of the loss function J, or the convergence of the cluster centers Z.

Algorithm 8: K-means Clustering Algorithm

Data: $\mathcal{D} = \{x_i\}_{i=1}^N$, $x_i \in \mathbb{R}^d$ for all i
Hyperparameters: K (number of clusters); stopping criterion
Initialize: $Z \in \mathbb{R}^{K \times d}$
while *stopping criterion not reached* **do**
  Update R: $r_{ik} = \begin{cases} 1 & k = \arg\min_j \|x_i - z_j\|^2 \\ 0 & \text{otherwise.} \end{cases}$, $i = 1, \ldots, N$;
  Update Z: $z_k = \frac{\sum_{i=1}^N r_{ik} x_i}{\sum_{i=1}^N r_{ik}}$, $k = 1, \ldots, K$;
end
return *cluster centers Z, cluster assignments R*

Let us say a few words about the convergence properties of the K-means algorithm. Notice in both steps of Alg. 8, the loss J cannot increase. Moreover, $J \geq 0$ and hence we know that

$$J(R_k, x_k) \to \hat{J} \text{ as } k \to \infty \tag{3.49}$$

for some $\hat{J} \geq 0$ starting with any R_0, x_0. In fact, we can show that the algorithm converges in a *finite* number of steps. To see this, observe that since the data is finite, there are at most K^N possible partitions (in fact, much less due to repetitions in the count). Hence the sequence

$\{R_k\}$ must end in cycles, but if R changes then the loss must decrease, thus the cycle must be of length 1.

However, we do not know Alg. 8 will converge to a global optimum. In fact, it is generally not true that the algorithm finds a global optimum, as the following example illustrates.

Example 3.6: Local Optimum vs Global Optimum in K-means

Let us consider K-means clustering in two dimensions ($d = 2$) with $N = 4$ data points, located at $x_1 = (2, 1), x_2 = (2, -1), x_3 = (-2, 1), x_4 = (-2, -1)$. Let us apply the K-means algorithm with $K = 2$. We initialize the cluster assignments so that $r_1 = r_2 = (1, 0)$ (cluster 1) and $r_3 = r_4 = (0, 1)$ (cluster 2). Observe that the K-means algorithm converges in one iteration, finding centers $z_1 = (1, 0)$, $z_2 = (-1, 0)$ and retaining this cluster assignment with loss $1/2$. However, if we initialize with a different cluster assignment $r_1 = r_3 = (1, 0)$ and $r_2 = r_4 = (0, 1)$, then the K-means algorithm again converges in one iteration, finding centers $z_1 = (0, 1)$, $z_2 = (0, -1)$ and retaining the assignment. However, this time we have loss 2, meaning that this is only a sub-optimal solution.

3.3.2 Gaussian Mixture Models

One drawback of the K-means algorithm is that the cluster identification is *hard*, in that each sample x_i belongs to one and only one cluster. This makes sense for problems where the data is clustered tightly, but this hard assignment is not stable if there are many data lying roughly in-between two classes.

In this section, we introduce Gaussian mixture models which can be interpreted both as a density estimation method, as well as a soft version of the K-means clustering.

Review: Conditional Distribution, Maximum Likelihood. Let A, B be events and suppose $\mathbb{P}(B) \neq 0$. The conditional probability of A given B is

$$\mathbb{P}(A|B) = \frac{\mathbb{P}(A \cap B)}{\mathbb{P}(B)}. \tag{3.50}$$

The Bayes rule (or Bayes theorem) follows from the definition of conditional probabilities

Proposition 3.7: Bayes Rule

We have $\mathbb{P}(A|B) = \frac{\mathbb{P}(B|A)\mathbb{P}(A)}{\mathbb{P}(B)}$ provided $\mathbb{P}(B) \neq 0$.

Let X, Y be random variables taking values in a discrete and finite set. The conditional distribution of X given Y is

$$p(x|y) = \mathbb{P}(X = x|Y = y) = \frac{\mathbb{P}(X = x \text{ and } Y = y)}{\mathbb{P}(Y = y)}. \tag{3.51}$$

For continuous random variables the definition is similar by replacing the terms on the right hand side with densities. Denoting $p(x) = \mathbb{P}(X = x)$ and $p(y) = \mathbb{P}(Y = y)$ as the marginal distributions and $p(x, y) = \mathbb{P}(X = x \text{ and } Y = y)$ as the joint distribution, it follows from Bayes rule that

$$p(x|y) = p(y|x)p(x)/p(y), \qquad p(y) \neq 0. \tag{3.52}$$

Often, we encounter problems of estimating some parameters r of a probabilistic model that explains some observed data $\{x_i\}$. That is, we assume that $\{x_i\}$ is i.i.d. according to some formal conditional distribution $p(x|r)$, and we want to estimate r from $\{x_i\}$. One way to do this is called *maximum likelihood estimation*, which solves the problem

$$r = \arg\max_s \log \prod_i p(x_i|s) = \arg\max_s \sum_i \log p(x_i|s). \tag{3.53}$$

Example 3.8: Fitting a Gaussian in 1D

Consider a dataset $\mathcal{D} = \{x_i\}_{i=1}^N$, which we model as i.i.d. samples from a standard normal distribution with mean z and variance σ^2 to be determined. Then, the log-likelihood of the dataset given the mean and variance is

$$l(\mathcal{D}, z, \sigma) = \log \prod_{i=1}^{N} (2\pi\sigma^2)^{-1/2} \exp\left(-\frac{1}{2\sigma^2}(x_i - z)^2\right). \tag{3.54}$$

$$= -\frac{N}{2}\log(2\pi\sigma^2) - \frac{1}{2\sigma^2}\sum_{i=1}^{N}(x_i - z)^2 \tag{3.55}$$

We now perform maximum likelihood estimation by maximizing $l(\mathcal{D}, z, \sigma)$ with respect to z, σ. Taking derivative with respect to z and setting it to 0 gives

$$\sum_{i=1}^{N}(x_i - z) = 0 \quad \Rightarrow \quad z = \frac{1}{N}\sum_{i=1}^{N} x_i. \tag{3.56}$$

Taking derivative with respect to σ and setting it to 0 gives

$$-\frac{N}{\sigma} + \frac{1}{\sigma^3}\sum_{i=1}^{N}(x_i - z)^2 = 0 \quad \Rightarrow \quad \sigma^2 = \frac{1}{N}\sum_{i=1}^{N}(x_i - z)^2. \tag{3.57}$$

In other words, the maximum likelihood estimators for z, σ^2 are the sample mean and variance respectively. These are known as *sufficient statistics* for the Gaussian distribution. The same holds for higher dimensions if one replaces the sample variance by the sample covariance matrix.

Gaussian Mixtures. In the K-means algorithm, each data point x_i is assigned to the cluster represented by the closest z_k, as decided by the i^{th} row of the 1-to-K encoding matrix, r_i, which equals 1 at the k^{th} position and 0 otherwise. We now consider a soft version of this.

In essence, we model $\{x_i\}$ as samples from some unknown probability distribution with K clusters, and our goal is to approximate this distribution. We will use the Gaussian mixture model to parameterize this distribution. A Gaussian mixture distribution has the following probability density function (PDF):

$$p(x) = \sum_{k=1}^{K} \pi_k p_g(x; z_k, \Sigma_k), \tag{3.58}$$

where $z_k \in \mathbb{R}^d$ and $\Sigma_k \in \mathbb{R}^{d\times d}$, $k = 1, \ldots, K$ are a collection of vectors and positive definite matrices, representing the mean and covariance of each Gaussian component, and p_g denotes

the Gaussian PDF, i.e.

$$p_g(x; z, \Sigma) = (2\pi)^{-d/2}|\Sigma|^{-1/2}e^{(x-z)^\top \Sigma^{-1}(x-z)}. \tag{3.59}$$

Here, $pi_k \geq 0$ and $\sum_k \pi_k = 1$, i.e. $\pi = (\pi_1, \ldots, \pi_K)$ is a probability vector. The π_k's are known as the *mixture weights* or *mixture coefficients*.

We now show that the Gaussian mixture distribution can be interpreted using conditional probabilities. The idea is simple: we can generate samples from (3.58) by first generating a one-hot vector $r \in \{0, 1\}^K$ which has a one at position k with probabilities π_k. Then, given r which has one at position k, we consider x which has a Gaussian distribution with mean m_k and Σ_k. More precisely,

$$p(r) = \prod_{k=1}^{K} \pi_k^{r_k}. \tag{3.60}$$

$$p(x|r) = \prod_{k=1}^{K} (p_g(x; z_k, \Sigma_k))^{r_k}. \tag{3.61}$$

Then, the marginal distribution for x is given by

$$p(x) = \sum_r p(x|r)p(r) = \sum_{k=1}^{K} \pi_k p_g(x; z_k, \Sigma_k), \tag{3.62}$$

which agrees with the Gaussian mixture distribution (3.58). Here, the vector r plays the role of a *latent* variable, whose existence helps represent the distribution of x more easily. We will see that the problem formulated this way allows us to come up with an algorithm to determine z_k, Σ_k and π_k from data.

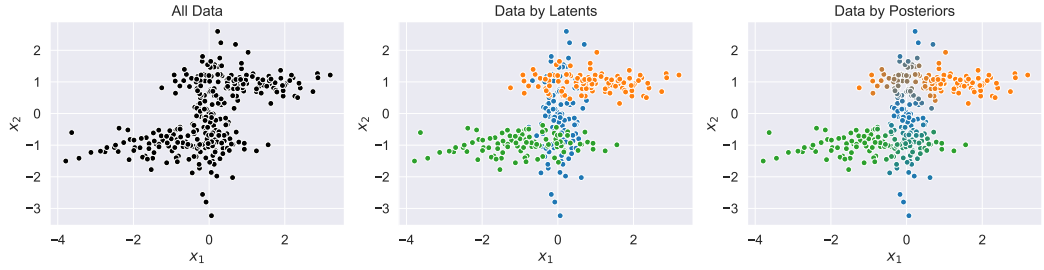
Before proceeding, we also use the Bayes rule to derive the conditional probability that r has a 1 at the k^{th} position, given x:

$$p(r_k = 1|x) = \frac{p(x|r_k = 1)p(r_k = 1)}{p(x)} = \frac{\pi_k p_g(x; z_k, \Sigma_k))}{\sum_{\ell=1}^{K} \pi_\ell p_g(x; z_\ell, \Sigma_\ell)}. \tag{3.63}$$

The interpretation is as follows: if we regard π_k as the *prior* probability for $r_k = 1$, then $p(r_k = 1|x)$ is the *posterior* probability that $r_k = 1$, after observing a realization x. It can be viewed as a measure of "responsibility" of the component k for explaining the outcome x. This viewpoint is illustrated in the example below.

Example 3.9: Posterior Distribution

We consider a 2D dataset generated from a mixture of three Gaussians shown in the left figure below. In the middle figure, we mark the dataset according to which Gaussian the data is generated from in the underlying latent generation process $\{r_{ik}\}$. In the far right figure, we plot the same dataset, but with colors representing the posteriors computed via (3.63), which shows which cluster explains each data point. We can see that in the overlaps of the 3 Gaussians, the posteriors take intermediate values.



Maximum Likelihood. Our task is to model $\mathcal{D} = \{x_i\}_{i=1}^N$ i.i.d. samples from a Gaussian mixture distribution of the form (3.58). This amounts to estimating $\{\pi_k, z_k, \Sigma_k\}$ from data. To proceed, let us represent $\mathcal{D}$ as the usual data matrix $X \in \mathbb{R}^{N\times d}$. For data point we can associate a latent variable $r_i \in \{0,1\}^K$ i.i.d. according to $\{\pi_k\}$. We can similarly write the latent variables as a matrix $R \in \mathbb{R}^{N\times K}$ whose i^{th} row is r_i.

Then, the log-likelihood is given by

$$\log p(X|\{\pi_k, z_k, \Sigma_k\}) = \sum_{i=1}^{N} \log \left[\sum_{k=1}^{K} \pi_k p_g(x_i; z_k, \Sigma_k) \right]. \tag{3.64}$$

Our goal is to now maximize this log-likelihood.

Remark. There is some technical issues with directly maximizing the log-likelihood without any regularization when $K \geq 2$ since the Gaussians can collapse onto a point (covariance goes to 0). See [BO06], Chapter 9 for a discussion on this.

An Iterative Algorithm. Let us now introduce a powerful algorithm for maximizing the likelihood in (3.64).

We begin by writing down the conditions that a maximum of the log-likelihood must satisfy. First, setting the derivative of (3.64) with respect to z_k to zero, we obtain

$$0 = -\sum_{i=1}^{N} \frac{\pi_k p_g(x_i; z_k, \Sigma_k)}{\sum_{\ell=1}^{K} \pi_\ell p_g(x_i; z_\ell, \Sigma_\ell)} \Sigma_k^{-1}(x_i - z_k) \tag{3.65}$$

Notice that the first term in the summation over i is the "responsibility" of the k^{th} component for x_i, $p(r_{ik} = 1|x_i)$ (See (3.63)). Let us denote this quantity by $\gamma_{ik} = p(r_{ik} = 1|x_i)$. Then, (3.66) implies

$$z_k = \frac{1}{N_k} \sum_{i=1}^{N} \gamma_{ik} x_i, \qquad N_k = \sum_{i=1}^{N} \gamma_{ik}. \tag{3.66}$$

In a sense, z_k is a weighted average of sample points x_i, whose weight is determined by the responsibility of the k'th cluster in explaining x_i. We can interpret N_k as the "effective" number of points assigned to cluster k.

Next, we set the derivative of (3.64) with respect to Σ_k to 0 to obtain

$$\Sigma_k = \frac{1}{N_k} \sum_{i=1}^{N} \gamma_{ik} (x_i - z_k)(x_i - z_k)^\top, \tag{3.67}$$

which is again the form of a weighted average of covariance matrices. Deriving this result in one dimension is similar to what is done in Example 3.8. In higher dimensions, one needs to use some properties of derivatives of log-determinants. However the idea is the same.

Exercise 3.10

Derive the result (3.67) in one dimension. Using the fact that $\nabla_A \log(\det(A)) = (A^{-1})^\top$ for any positive definite matrix A, derive the results in general.

Finally, we maximize (3.64) with respect to the mixture weights π_k. This is a constrained optimization since we require $\sum_k \pi_k = 1$. Using the Lagrange multiplier method, we have the Lagrangian

$$\log p(X|\{\pi_k, z_k, \Sigma_k\}) + \lambda \left(\sum_k \pi_k - 1 \right). \tag{3.68}$$

Taking derivative with respect to π_k and setting it to 0, we get

$$\sum_{i=1}^{N} \frac{p_g(x_i; z_k, \Sigma_k)}{\sum_{\ell=1}^{K} \pi_\ell p_g(x_i; z_\ell, \Sigma_\ell)} + \lambda = 0. \tag{3.69}$$

Multiply both sides by π_k and summing over k yields $\lambda = -N$. Substitute this back to the above gives (noting that the first term is nothing by γ_{ik})

$$\pi_k = \frac{N_k}{N}. \tag{3.70}$$

That is, the mixture coefficients are just the proportion of the effective number of data points assigned to the k^{th} cluster.

Importantly, notice that expressions (3.66), (3.67) and (3.70) are not explicit solutions of the maximum likelihood problem, since they are inter-dependent. For example, to compute them we need $\{\gamma_{ik}\}$, which depends non-trivially on $\{z_k, \pi_k, \sigma_k\}$. However, just like in the K-means algorithm, these expressions point us to a way to an iterative algorithm:

1. Suppose we know $\{z_k, \pi_k, \sigma_k\}$, then we can compute $\{\gamma_{ik}\}$.
2. Suppose we know $\{\gamma_{ik}\}$, then we can compute $\{z_k, \pi_k, \sigma_k\}$.

Hence, we simply iterate the above two steps until some error criterion is reached. The algorithm is summarized in Alg. 9, again with many possible stopping criteria. Just as in Alg. 8, we can show that each iteration must increase the log-likelihood (3.64) and thus the algorithm converges.

Algorithm 9: Maximum Likelihood for Gaussian Mixture Models

Data: $\mathcal{D} = \{x_i\}_{i=1}^N$, $x_i \in \mathbb{R}^d$ for all i
Hyperparameters: K (number of clusters); stopping criterion
Initialize: $\pi_k = 1/K$, $z_k \in \mathbb{R}^d$, $\Sigma_k \in \mathbb{R}^{d\times d}$ for $k = 1, \ldots, K$
while *stopping criterion not reached* **do**
  Update $\{\gamma_{ij}\}$: $\gamma_{ik} = \frac{\pi_k p_g(x; z_k, \Sigma_k))}{\sum_{\ell=1}^K \pi_\ell p_g(x; z_\ell, \Sigma_\ell)}$., $i = 1, \ldots, N$, $k = 1, \ldots, K$;
  Compute $\{N_k = \sum_{i=1}^N \gamma_{ik}\}$ and Update $\{\pi_k, z_k, \Sigma_k\}$:

$$z_k = \frac{1}{N_k}\sum_{i=1}^N \gamma_{ik} x_i, \quad \Sigma_k = \frac{1}{N_k}\sum_{i=1}^N \gamma_{ik}(x_i - z_k)(x_i - z_k)^\top, \quad \pi_k = \frac{N_k}{N}. \tag{3.71}$$

  for $k = 1, \ldots, K$;
end
return *cluster centers, covariances, and mixture coefficients* $\{z_k, \Sigma_k, \pi_k\}$*, cluster responsibilities or soft assignments* $\{\gamma_{ik}\}$,

Relationship between Gaussian Mixture Models and K-means Clustering. One may guess from the similarities in Alg. 8 and Alg. 9 that there should be some relationship between the two methods. In fact, we motivated the Gaussian mixture model as a soft version of the K-means algorithm. Let us now make this precise.

We consider fitting a Gaussian mixture model (3.58), but with fixed common covariance matrices $\Sigma_k = \epsilon I_d$, where $\epsilon > 0$ and I_d is the $d \times d$ covariance matrix. This gives the PDF

$$p_g(x; z_k, \Sigma_k) = \frac{1}{(2\pi\epsilon)^{d/2}} \exp\left(-\frac{1}{2\epsilon}\|x - z_k\|^2\right) \tag{3.72}$$

We will regard ϵ (and hence the covariance matrices) as given and not estimate it from data. From Alg. 9, given any $\{\pi_k, z_k, \Sigma_k = \epsilon I\}$, we estimate the posteriors (responsibilities) as

$$\gamma_{ik} = \frac{\pi_k \exp\left(-\frac{1}{2\epsilon}\|x_i - z_k\|^2\right)}{\sum_{\ell=1}^K \pi_\ell \exp\left(-\frac{1}{2\epsilon}\|x_i - z_\ell\|^2\right)} \tag{3.73}$$

Let us assume that $\pi_k \neq 0$ for all k and $x_i \neq z_k$, then as $\epsilon \longrightarrow 0$, we can see that

$$\gamma_{ik} \longrightarrow \begin{cases} 1 & k = \arg\min_\ell \|x_i - z_\ell\|^2, \\ 0 & \text{otherwise}, \end{cases} \tag{3.74}$$

assuming that the arg min is unique. This is identical to the hard assignment r_{ik} defined in (3.46) in K-means clustering. Furthermore, the estimation for z_k in (3.66) reduces to the K-means estimation (3.48), since $\gamma_{ik} \longrightarrow r_{ik}$. The estimation of $\{\pi_k\}$ in (3.70) is inconsequential here. In summary, the Gaussian mixture model reduces to the K-means clustering algorithm in the limit of zero covariance matrix. In this sense, the former in the case of finite covariance is a soft, or probabilistic extension of the K-means clustering algorithm.

3.3.3 Further Reading

There are a number of extensions of the basic K-means algorithm introduced here. First, with regards to computational complexity, the estimation step for the assignment matrix $\{r_{ij}\}$ requires computation of a large number of Euclidean norms. In high dimensions this is inefficient. Consequently, approaches based on precomputing a good data structure [RP89, Moo00] or using triangle inequality to exclude redundant computations have been used to improve the computational efficiency of the K-means algorithm [Hod88]. Another way to reduce computational burden is to use mini-batch algorithms to update the cluster centers z_k, i.e. using stochastic gradient descent [Mac67]. Furthermore, we have only considered loss functions of the mean-squared type. One can extend this to arbitrary loss functions, giving rise to the so-called *K-medoids* algorithm [PJ09].

Algorithm 9 is in fact a special case of a large class of algorithms known as *expectation maximization* (EM) algorithms, which are commonly applied to solve maximum likelihood or maximum *a posterior* estimation for problems involving latent random variables. The interested reader may refer to [BO06] chapter 9, upon which most of the material in this chapter is based.

4 Reinforcement Learning

4.1 Overview

So far we have covered supervised learning problems where we are given labelled data, as well as unsupervised learning where we are only given inputs. Reinforcement learning (RL) presents another paradigm somewhere in between. In reinforcement learning problems, we are given some reward signal that promotes desired outcomes, and through maximizing the reward we can build learners that accomplishes very complex tasks.

Some distinguishing features of reinforcement learning problems are as follows: First, they involve some form of long-term planning, as maximizing the immediate reward is often not the desired tasks. Next, they typically are *closed-loop* systems in the sense that the learner's actions also affect the environment. Finally, they involve accumulating experience and presents a general problem of exploration vs exploitation.

Let us give some examples of problems RL can be applied to in order to make the above points clearer.

1. Playing chess: this involves long term planning, as reward in the form of win-loss only comes at the end of the game. We get better at chess by playing chess.
2. A robotic vacuum cleaner finding the best path to clean a room with unknown geometry, and deciding when to go back for charging or continues cleaning. Here exploration is key, but it should also perform planning so as to not run of battery before reaching the charging station.
3. A newly-born gazelle learning how to walk and run. Exploration and processing of constant reward signal allows it to learn quickly.

There are of course many more examples and the reader is encouraged to browse [Sut99] for other instances.

Of course, as always we should formalize the heuristic notions we described before. In reinforcement learning, three key entities are involved:

1. The *agent*
2. The *environment*
3. The *interpreter*

The agent is the learner, which can perform *actions* that affect the environment. Through the interpreter, the environment then tells the agent the associated *reward* for that action at the current *state* of the agent-environment system, as well as the next *state* as a result of that action and other extrinsic factors. The interpreter is present for the agent may not always be able to fully observe the environment. Figure 4.1 illustrates these concepts.

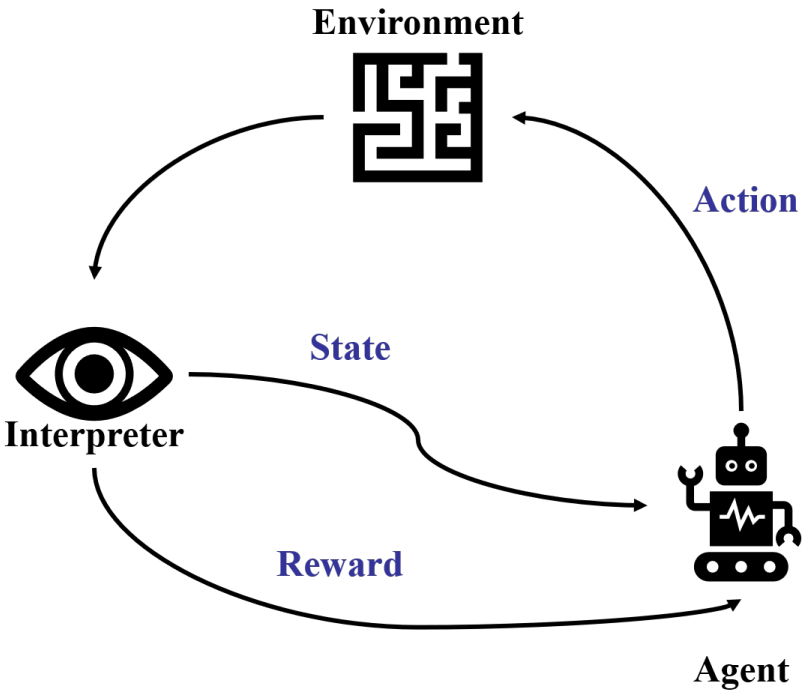


Figure 4.1: The main components of a RL system.

Example 4.1: The RL Components for a Robotic Vacuum Cleaner

For the robotic cleaner example above, the agent is the robot. The environment is the entire room together with the full status of the robot. The robot cannot observe the entire environment, and its perception of the environment is through its own charted map of the room, the location of the charging hub, as well as its own battery level. Possible actions of the robot include to clean, to move, or to go back to the charging hub. Positive rewards should be given when the robot cleans up trash, and a large negative reward is given when the robot runs out of battery before reaching the charging hub.

Let us now emphasize the key difference between RL and supervised learning as well as unsupervised learning: unlike fully unsupervised learning where no information other than inputs are given, in RL we have a form of reward signal that guides our learning. However,

unlike supervised learning, these reward signals are not some predictions we wish to make, but rather just guidance that steers us towards the right path. For example, applying RL to chess, we only need to give a positive reward for winning, but unlike supervised learning, we do not need a supervisor to tell us what is the best move to take at each board position. This highlights the flexibility of the RL framework.

4.2 Markov Decision Processes

To set the stage for RL algorithms and applications, we first introduce the basic mathematical framework that underlies RL, known as *Markov decision processes* (MDP).

Let some agent interact with an environment in a sequence of time steps $t = 0, 1, 2, \ldots$. At each time t, the agent receives through the interpreter some state $S_t \in \mathcal{S}$ where $\mathcal{S}$ is the set of possible states that the agent can observe. On the basis of S_t, the agent performs an action $A_t \in \mathcal{A}(S_t)$, with $\mathcal{A}(S_t)$ the space of actions that the agent can perform at state S_t. One time step later, the agent receives a numerical reward $R_{t+1} \in \mathcal{R} \subset \mathbb{R}$ and finds itself in a new state S_{t+1}. Figure 4.2 illustrates these essential components.

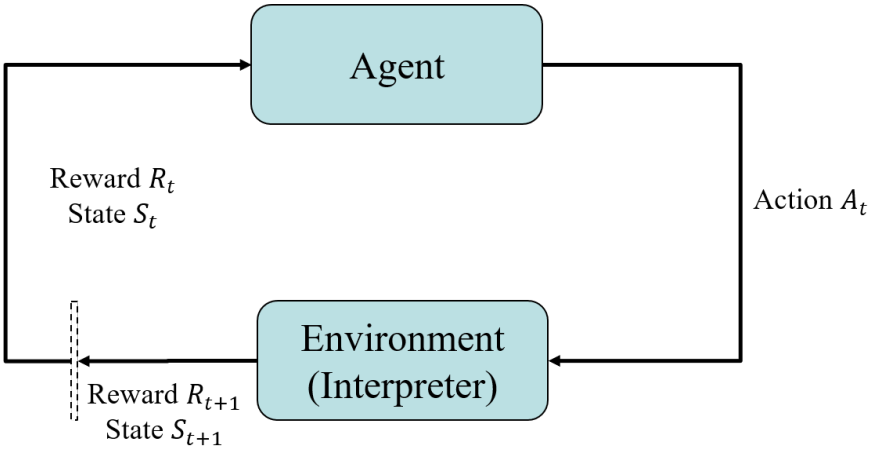


Figure 4.2: Components of a Markov decision process.

The Markov decision process models the inter-dependence of the quantities $\{S_t, A_t, R_t\}$. We define the *policy*, denoted by π, as

$$\pi(a|s) = \mathbb{P}[A_t = a|S_t = s]. \tag{4.1}$$

The agent will select actions stochastically based on the current state and the policy function, i.e.

$$A_t \sim \pi(\cdot|S_t). \tag{4.2}$$

The agent's goal is to maximize long term reward by adjusting its policy π through experience. We will also consider deterministic policies where $\pi(a|s) = \mathbb{1}_{a=a(s)}$. In this case, we can simply interpret π as a mapping

$$\pi : \mathcal{S} \to \mathcal{A} \qquad \pi(s) = a(s). \tag{4.3}$$

Of course, the reward R_{t+1} is going to depend on the current action and state. The same holds for the next state S_{t+1}, which should also depend on on the current action and state. Just like the policy function, we will also assume that they are stochastic, i.e. they are sampled according to

$$p(s', r|s, a) = \mathbb{P}[S_{t+1} = s', R_{t+1}|S_t = s, A_t = a]. \tag{4.4}$$

The primary assumption in (4.4) is that the system is *Markovian*, meaning that the distribution of the reward and next state depends *only* on the current state and action taken, and not on their histories. This is the primary assumption we take hereafter. Unlike the policy function, (4.4) is a property of the underlying system and not something that the agent has control over.

Definition 4.2: Markov Decision Processes

A Markov Decision Process (MDP) is specified by its state space $\mathcal{S}$, action space $\mathcal{A}(s)$, $s \in \mathcal{S}$ and the one-step stochastic dynamics in (4.4). We say that the MDP is finite if $\mathcal{S}$ is finite and $\mathcal{A}(s)$ is finite for each $s \in \mathcal{S}$.

The goal of RL is not to greedily maximize the reward at the present, but a long term reward. We will call this the *return*, which is defined by

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}. \tag{4.5}$$

The number $\gamma \in (0, 1]$ is known as the *discount factor*. For infinite horizon problems, we should take $\gamma < 1$ so that G_t remains finite. In this sense, we weight the immediate rewards to be more significant than future rewards, but the latter is still taken into consideration. For finite-horizon problems where $R_t = 0$ for all $t > T$ (T is the time horizon), then we may take $\gamma = 1$.

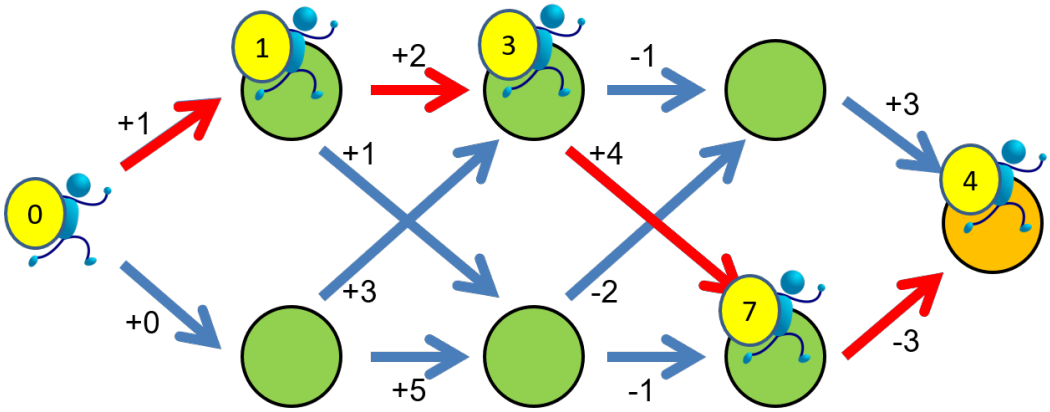
Our goal in RL is to maximize the expected return starting from some initial state $S_0 = s_0 \in \mathcal{S}.$, i.e. the quantity

$$\mathbb{E}_\pi[G_0|S_0 = s]. \tag{4.6}$$

Dynamic Programming. Before we introduce the concept of *value functions* which are central objects of study in reinforcement learning, let us first motivate its usefulness by considering a simple example. In doing so, we will illustrate a general recursive principle known as dynamic programming.

Example 4.3: A Toy Maze

Consider the following maze where we want to get to the orange circle while maximizing the reward obtained along the way. When we cross each arrow, we gain a reward equal to the number attached to that arrow. The red path shows an example path with a final reward of 4.



Suppose that there are N circles to choose from per step and T steps in total. Then, the total number of paths is N^T and grows exponentially with T. This is known as the *curse of dimensionality* [Bel66].

Instead of a brute force search over all paths, we can use the principle of dynamic programming to find a solution much more efficiently. To do this, let us introduce some notation. We will index each time step in the maze by $t = 0, 1, \ldots, T$. Also, we denote by S_t the circle we step on at the t^{th} step, and R_t the reward we obtain at the t^{th} step.

Define the function

$$v_t(s) = \max\left\{\sum_{s=t+1}^{T} R_s : S_t = s\right\}. \tag{4.7}$$

In other words, $v_t(s)$ is the best possible reward we can get starting from state s at time t. Then, we can work backwards easily!

Let us just consider the case in Example 4.3, where $S_t = 1$ or 2 for $t = 1, 2, 3$. Here, $S_t = 1$ denotes the top circle and $S_t = 2$ is the bottom circle. The initial state is $S_0 = 0$. Then, clearly we have

$$v_3(1) = +3, \qquad v_3(2) = -3, \tag{4.8}$$

since both cases we only have one choice – and this is the best we can do. Now, let us consider $t = 2$. Given we are at $S_2 = 1$, then there are two choices, either we go to $S_3 = 1$ or $S_3 = -1$. If we go to $S_3 = 1$ we get a reward of -1 and then, the best we can do from there would be

$v_3(1) = +3$. Similarly, if we take $S_3 = 2$ then we get +4 reward and the best we can do from $S_3 = 2$ is $v_3(2) = -3$. Hence,

$$v_2(1) = \max\{-1 + v_3(1), +4 + v_3(2)\} = +2. \tag{4.9}$$

A similar calculation shows that $v_2(2) = +1$. Once we know these values we can then compute $v_1(\cdot)$ and so on. This allows us to calculate backwards to obtain $v_0(0) = +6$. This is the best possible reward we can get, and we have obtained it without resorting to brute force search over all the paths! Moreover, once we have solved for $v_t(s)$ for all t, s, we can also easily find the optimal policy to navigate this maze. We simply proceed *greedily* with respect to the value function: at time t we always go the circle in the next step with the highest $v_{t+1}(s)$ plus the current immediate reward.

In fact, the above methodology is known as *dynamic programming* [Bel66]. Let us look at the computational complexity of dynamic programming versus a brute force search, which takes N^T steps. In dynamic programming, we simply have to traverse the time steps once, starting the from the end. For each time step, we have to compute N values of $v_t(s)$, each depends on a linear combination of $v_{t+1}(s)$. Hence, for each time step we incur a computation overhead of N^2. Therefore, the entire dynamical programming procedure solves the problem in N^2T steps. This is *much* less than N^T!.

The key idea behind dynamic programming is defining the so called cost-to-go $v_t(s)$ (4.7), which allows us to derive a recursion in $v_t(s)$ that gives a solution to our original problem. The function $v_t(s)$ is also known as the *value function*, emphasizing the fact that it represents the "value" of a given state.

We will see that in reinforcement learning, we can also define a value function that is similar in spirit to the one we defined above. Consequently, this allows us to derive both theoretical guarantees and practical algorithms that can effectively solve reinforcement learning problems.

4.2.1 Value Function and the Bellman's Equation

Recall that in RL, we want to maximize the expected return (4.6). To find this, we follow the dynamic programming principle and define the *value function*

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s\right], \tag{4.10}$$

where $\mathbb{E}_\pi$ denotes the expected value of a random variable given that the agent follows policy π. We will specifically call $v_\pi(s)$ the *state value function*, emphasizing that it represents the value of a particular state.

On the other hand, we can also define the *action value function*, corresponding to the value of an action taken under a state, which is

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s, A_t = a\right]. \tag{4.11}$$

One can immediately see that from definition, we have the following identities

$$q_\pi(s,a) = \sum_{s',r} p(s',r|s,a)[r + \gamma v_\pi(s')] \tag{4.12}$$

$$v_\pi(s) = \sum_a \pi(a|s) q_\pi(s,a). \tag{4.13}$$

As is shown in Example 4.3, the key to using value functions is to derive some recursion. We now do this for both the state and action value functions. First, for the state value function we have

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi\left[G_t|S_t = s\right] \\ &= \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s\right] \\ &= \mathbb{E}_\pi\left[R_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k R_{t+k+2}|S_t = s\right] \\ &= \sum_a \pi(a|s) \sum_{s'} \sum_r p\left(s',r|s,a\right)\left[r + \gamma \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+2}|S_{t+1} = s'\right]\right] \\ &= \sum_a \pi(a|s) \sum_{s',r} p\left(s',r|s,a\right)\left[r + \gamma v_\pi\left(s'\right)\right]. \end{aligned} \tag{4.14}$$

The above gives the following recursion for $v_\pi(s)$

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p\left(s',r|s,a\right)\left[r + \gamma v_\pi\left(s'\right)\right]. \tag{4.15}$$

This is known as the *Bellman equation.*

Equation (4.15) is a recursion, in fact a *linear* one. To see this, define

$$p(s'|s,a) := \sum_r p(s',r|s,a), \tag{4.16}$$

$$P(\pi)_{ss'} := \sum_a \pi(a|s) p\left(s'|s,a\right), \tag{4.17}$$

$$b(\pi)_s := \sum_a \pi(a|s) \sum_r p\left(s',r|s,a\right) r, \tag{4.18}$$

where we denote by $p(s'|s,a) = \sum_r p(s',r|s,a)$ the marginal conditional distribution of s'. Now, denote by $P(\pi)$ the $|\mathcal{S}| \times |\mathcal{S}|$ matrix with entries $\{P(\pi)_{ss'} : s,s' \in \mathcal{S}\}$ and $b(\pi)$, v_π the $|\mathcal{S}|$ dimensional vectors with entries $\{b(\pi)_s : s \in \mathcal{S}\}$ and $\{v_\pi(s) : s \in \mathcal{S}\}$ respectively, then we can rewrite (4.15) as

$$v_\pi = \gamma P(\pi) v_\pi + b(\pi). \tag{4.19}$$

One can then show that this equation has a unique solution.

Proposition 4.4: Existence and Uniqueness of Solution to Bellman's Equation

There exists a unique solution $\{v_\pi(s) : s \in \mathcal{S}\}$ to the Bellman's equation (4.19) given by

$$v_\pi = (I - \gamma P(\pi))^{-1} b(\pi).$$

Proof. To show this it is enough to show that $(I - \gamma P(\pi))$ is invertible, for which it is sufficient to show $\|P(\pi)\|_\infty = \max_s \sum_{s'} |P(\pi)_{ss'}| \leq 1$. In fact, we have the equality

$$\begin{aligned}
\max_s \sum_{s'} |P(\pi)_{ss'}| &= \max_s \sum_{s'} \left| \sum_a \pi(a|s) p(s'|s,a) \right| \\
&= \max_s \sum_a \pi(a|s) \sum_{s'} p(s'|s,a) \\
&= \max_s \sum_a \pi(a|s) = 1.
\end{aligned}$$

This implies $I - \gamma P(\pi)$ has no zero eigenvalues and thus invertible.

4.2.2 Optimal Policy and Bellman's Optimality Equation

The value function induces a partial order on the space of policies. For two policies π and π', we say that $\pi \geq \pi'$ if $v_\pi(s) \geq v_{\pi'}(s)$ for all $s \in \mathcal{S}$. Then, it makes sense to ask whether there exists an *optimal policy* π_* such that $\pi_* \geq \pi$ for all π. It turns out that this is true, but it is not obvious. This is because we will need $v_{\pi_*}(s) \geq v_\pi(s)$ for all s and all π simultaneously for the same v_*.

We will give a derivation on why this is true, and in doing so we will introduce the important concept of the *Bellman's optimality equation* that characterizes the optimal value function. This will become important when designing algorithms. We first state the formal definition of an optimal policy as discussed above.

Definition 4.5: Optimal Policy

A policy π_* is optimal if its value is maximal for every state $s \in \mathcal{S}$, i.e. for any other policy π, we have $v_{\pi_*}(s) \geq v_\pi(s)$ for all $s \in \mathcal{S}$.

At this point, it is not clear if an optimal policy exists, and if it does, it is also not clear if its unique. We will show that existence holds, but uniqueness does not.

Recall the definition of the action-value function q_π as defined in (4.11). We first show the following result on policy improvement.

Proposition 4.6: Policy Improvement

For any two policies π, π', if

$$\sum_a \pi'(a|s) q_\pi(s,a) \geq \sum_a \pi(a|s) q_\pi(s,a) \qquad \forall s \in \mathcal{S},$$

then we must have

$$v_{\pi'}(s) \geq v_\pi(s) \qquad \forall s \in \mathcal{S}.$$

Furthermore, a strict inequality for one s in the first implies a strict inequality for at least one s in the second.

Proof. Let π, π' be as assumed above, then

$$\begin{aligned}
v_\pi(s) &= \sum_a \pi(a|s) q_\pi(s,a) && \text{[Using (4.13)]} \\
&\leq \sum_a \pi'(a|s) q_\pi(s,a) \\
&= \sum_a \pi'(a|s) \sum_{s',r} p(s',r|s,a) \left[r + \gamma v_\pi(s') \right] && \text{[Using (4.12)]} \\
&= \mathbb{E}_{\pi'} \left[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s \right].
\end{aligned}$$

Repeating this process on $v_\pi(S_{t+1})$, we have

$$\begin{aligned}
v_\pi(s) &\leq \mathbb{E}_{\pi'} \left[R_{t+1} + \gamma R_{t+2} + \gamma^2 v_\pi(S_{t+2}) | S_t = s \right] \\
&\leq \dots \\
&\leq \mathbb{E}_{\pi'} \left[\sum_{k=0}^{K} \gamma^k R_{t+k+1} + \gamma^{K+1} v_\pi(S_{K+1}) | S_t = s \right],
\end{aligned}$$

for any $K \geq 0$. Taking limit $K \to \infty$ and noting that v_π is bounded, $0 < \gamma < 1$, we have

$$v_\pi(s) \leq v_{\pi'}(s).$$

Finally, notice that if there is a strict inequality in the assumption, there is at least one strict inequality for the result.

Intuitively, Proposition 4.6 says that if we were concentrate the policy on actions with large action values, then we tend to have a policy that is at least as good as the current one. This naturally suggests that an optimal policy should already maximize $q_\pi(s,a)$, otherwise it can be improved, which leads to a contradiction. The following result makes this precise.

Theorem 4.7: Bellman Optimality Condition (Necessary)

Let π_* be an optimal policy. Then, for any pair of state action $(s, a) \in \mathcal{S} \times \mathcal{A}$ such that $\pi_*(a|s) > 0$, the following holds:

$$a \in \arg\max_{a'} q_{\pi_*}(s, a'). \tag{4.20}$$

In particular, the value function of π_* satisfies the *Bellman's optimality equation*

$$v_{\pi_*}(s) = \max_a q_{\pi_*}(s, a) \equiv \max_a \sum_{s',r} p(s', r|s, a)[r + \gamma v_{\pi_*}(s')] \tag{4.21}$$

Proof. By Prop 4.6, if (4.20) does not hold for some s then the policy π is not optimal, since π can be improved by defining $\pi'(\cdot|s') = \pi(\cdot|s')$ for all $s' \neq s$, and $\pi'(a|s) = 1$ for some $a \in \arg\max_{a'} q_\pi(s, a')$ and 0 otherwise.
Now, π_* must satisfy the Bellman's equation, i.e.

$$\begin{aligned} v_{\pi_*}(s) &= \sum_a \pi_*(a|s) \underbrace{\sum_{s',r} p(s', r|s, a)[r + \gamma v_{\pi_*}(s')]}_{q_{\pi_*}(s,a)} \\ &= \max_a \sum_{s',r} p(s', r|s, a)[r + \gamma v_{\pi_*}(s')], \end{aligned}$$

where the last step follows from (4.20).

In fact, the converse of Thm. 4.7 also holds. To establish this, we first show that the Bellman's optimality equation admits a unique solution. This relies on the following well-known result in analysis.

Theorem 4.8: Contraction Mapping Theorem

Let V be a complete normed space and $F : V \to V$ be a contraction, i.e.

$$\|F(v) - F(u)\| \leq \gamma \|v - u\|, \qquad \forall v, u \in V \qquad 0 \leq \gamma < 1.$$

Then, there exists a unique $v_* \in V$ such that such that $F(v_*) = v_*$. Moreover,

$$v_* = \lim_{k \to \infty} v_k$$

for any sequence $\{v_k\}$ such that $v_{k+1} = F(v_k)$ and $v_0 \in V$ is arbitrary.

Proposition 4.9: Uniqueness of Solutions of Bellman's Optimality Equation

There exists a unique v_* that satisfies the Bellman's optimality equation

$$v_*(s) = \max_a \sum_{s',r} p(s', r|s, a)[r + \gamma v_*(s')]$$

Proof. Define $F : \mathbb{R}^n \to \mathbb{R}^n$ (with $n = |\mathcal{S}|$)

$$F(v)(s) := \max_a \sum_{s',r} p(s', r|s, a)[r + \gamma v(s')].$$

Then, the Bellman's optimality equation can be written as $v_* = F(v_*)$, i.e. it is a fixed point of F. Using Thm. 4.8, it is enough to show that F is a contraction under the infinity norm $\|v\|_\infty = \max_s |v(s)|$.
Let $v, u \in \mathbb{R}^n$ be arbitrary and set

$$\pi_v(s) = \arg\max_a \sum_{s',r} p(s', r|s, a)[r + \gamma v(s')],$$

which is the maximizing action corresponding to state s given value v. Then,

$$\begin{aligned} F(v)(s) - F(u)(s) &\le F(v)(s) - \sum_{s',r} p(s', r|s, \pi_v(s))[r + \gamma u(s')] \\ &= \gamma \sum_{s',r} p(s', r|s, \pi_v(s))[v(s') - u(s')] \\ &\le \gamma \|v - u\|_\infty \sum_{s',r} p(s', r|s, \pi_v(s)) \\ &= \gamma \|v - u\|_\infty. \end{aligned}$$

By swapping u, v in the above argument we also have $F(u)(s) - F(v)(s) \le \gamma \|u - v\|_\infty$. Thus we have shown

$$\|F(v) - F(u)\|_\infty \le \gamma \|v - u\|_\infty,$$

and thus the result follows from Thm. 4.8.

Now that we have shown that there exists a unique solution to the Bellman's optimality equation, we can deduce that the Bellman's optimality condition (4.20) is a necessary *and* sufficient condition for optimality. We summarize this main result in the following theorem.

Theorem 4.10: Bellman Optimality Condition

A policy π is optimal if and only if for any pair of state action $(s, a) \in \mathcal{S} \times \mathcal{A}$ such that $\pi(a|s) > 0$, the following holds:

$$a \in \arg\max_{a'} q_\pi(s, a'). \tag{4.22}$$

In particular, v_π is the unique solution of the Bellman's optimality equation

$$v_\pi(s) = \max_a q_\pi(s, a) \equiv \max_a \sum_{s', r} p(s', r|s, a)[r + \gamma v_\pi(s')] \tag{4.23}$$

Proof. The "only if" part follows from Thm. 4.7. The "if" part can be deduced by the observation that if (4.22) holds, then v_π satisfies (4.23), and by uniqueness of its solution (Thm. 4.9) we deduce that $v_\pi = v_*$ and hence $v_\pi(s) = v_*(s) \geq v_{\pi'}(s)$ for any s and any other policy π'. Thus, it is an optimal policy.

Now, we show how Prop. 4.6 and Thm. 4.10 implies the existence of an optimal policy π_* (as defined in Def. 4.5). In fact, we can show that there exists an optimal *deterministic* policy.

Theorem 4.11: Existence of Optimal Deterministic Policy

Any finite MDP admits an optimal deterministic policy.

Proof. Let π_* be a deterministic policy that maximizes $\sum_{s \in \mathcal{S}} v_\pi(s)$. The existence of such a policy is guaranteed since there are only finite many deterministic policies. If π_* is not an optimal policy, then by Theorem 4.10 there exists a state s such that

$$\pi_*(s) \notin \arg\max_{a'} q_{\pi_*}(s, a')$$

By Prop. 4.6, we can then improve π_* by a modified policy such that

$$\hat{\pi}(s') = \pi_*(s') \qquad \forall s' \neq s \qquad \text{and} \qquad \hat{\pi}(s) \in \arg\max_{a'} q_{\pi_*}(s, a').$$

This gives a strict inequality for at least one s and so $\sum_s v_{\hat{\pi}}(s) > \sum_s v_{\pi_*}(s)$, which contradicts the fact that π_* maximizes $\sum_s v_\pi(s)$.

Thm. 4.11 suggests that for the purpose of analyzing finite MDPs, it is enough to consider only deterministic policies. In the following, we shall denote by π_* such an optimal deterministic policy. Then, we can denote by

$$v_*(s) := v_{\pi_*}(s) = \max_\pi v_\pi(s) \qquad \text{and} \qquad q_*(s, a) := q_{\pi_*}(s, a) = \max_\pi q_\pi(s, a) \tag{4.24}$$

the corresponding optimal value function and action value function respectively. By Thm. 4.10,

we have the formula

$$\pi_*(s) \in \arg\max_a q_*(s,a). \tag{4.25}$$

Therefore, knowing the optimal action value function is enough to determine an optimal policy. Note that the analogues of (4.12), (4.13) are

$$q_*(s,a) = \sum_{s',r} p(s',r|s,a)[r + \gamma v_*(s')] \tag{4.26}$$

$$v_*(s) = q_*(s,\pi_*(s)) = \max_a q_*(s,a). \tag{4.27}$$

Combining, we have the *Bellman's optimality equations* for the optimal value function already discussed before

$$v_*(s) = \max_a \sum_{s',r} p(s',r|s,a)[r + \gamma v_*(s')] \tag{4.28}$$

and the action value function

$$q_*(s,a) = \sum_{s',r} p(s',r|s,a)[r + \gamma \max_{a'} q_*(s',a')]. \tag{4.29}$$

The preceding argument shows the following important points:

1. For finite MDPs, there always exists an optimal policy π_* such that $\pi_* \geq \pi$ for all policies π.
2. The optimal value function v_* associated with π_* is unique. However, an optimal policy need not be unique.
3. There always exists an optimal policy that is deterministic (See (4.25)). In fact, for every stochastic policy there always exists a deterministic policy that is at least as good.

In the next sections, we will investigate algorithms to solve for the optimal (action) value functions in order to find optimal policies. We will investigate a variety of algorithms for different settings.

4.3 Numerical Algorithms for Reinforcement Learning

In this section, we introduce some basic algorithms to solve reinforcement learning problems posed as finite MDPs. First, we distinguish between the concept of *model-free* and *model-based* algorithms.

- **Model-based:** Here we assume that we know the underlying probabilistic model for the interaction between the environment and the agent is known to us. More precisely, we have access to the function $p(s,r'|s,a)$ that defines the MDP.

- **Model-free:** In model-free algorithms, we do not know the function $p(s, r'|s, a)$ and we can only obtain samples from p.

In fact, recent progress in reinforcement learning is precisely for model-free situations, which greatly enhances the range of applicability. However, to introduce algorithms we first focus on the model-based approach.

4.3.1 Model-based Algorithms

As seen in Section 4.2.2, a solution of the RL problem can be obtained as long as we solve the Bellman's optimality equation (4.28). In the model-based scenario, the terms in the equation are completely known to us. Let us restate the equation here here:

$$v_*(s) = \max_a \sum_{s',r} p\left(s', r|s, a\right)\left[r + \gamma v_*\left(s'\right)\right]. \tag{4.30}$$

Also, given v_*, the optimal action value function is

$$q_*(s, a) = \sum_{s',r} p(s', r|s, a)[r + \gamma v_*(s')], \tag{4.31}$$

from which the optimal value function and a deterministic optimal policy can be formed by

$$v_*(s) = \max_a q_*(s, a), \qquad \pi_*(s) \in \arg\max_a q_*(s, a). \tag{4.32}$$

Thus, this shows that we can find an optimal policy, thereby solving the reinforcement learning problem, if we first solve the Bellman's optimality equation (4.30). This forms the first class of model-based algorithms known as *value iteration*.

Value Iteration. Let us now introduce the value iteration algorithm for model-based reinforcement learning. Observe that we can rewrite the Bellman's optimality equation (4.30) as

$$v_* = F(v_*), \tag{4.33}$$

where the $F : \mathbb{R}^n \to \mathbb{R}^n$ ($n = |\mathcal{S}|$) is defined as

$$F(v)(s) = \max_a \sum_{s',r} p(s', r|s, a)[r + \gamma v(s')]. \tag{4.34}$$

Note that using the previous matrix notations, we can rewrite this as

$$F(v) = \max_\pi [\gamma P(\pi)v + b(\pi)], \tag{4.35}$$

where the maximization is taken over all deterministic policies $\pi : \mathcal{S} \to \mathcal{A}$ and

$$\begin{aligned} P(\pi)_{ss'} &= \sum_r p(s', r|s, \pi(s)) \\ b(\pi)_s &= \sum_r r p(s', r|s, \pi(s)) \end{aligned} \tag{4.36}$$

Owing to (4.33), we can find v_* by solving the fixed point problem, via iterating $v_{k+1} = F(v_k)$, $k = 0, 1, 2, \ldots$. Suppose that the limit of v_k as $k \to \infty$ exists, then it satisfies $F(v_\infty) = v_\infty$, and by uniqueness of optimal value function it implies $v_\infty = v_*$. The algorithm is summarized in Alg. 10. A simple stopping criterion is $\|v - F(v)\| < \epsilon$ for some small error tolerance ϵ.

Algorithm 10: Value Iteration Algorithm

Model Parameters: MDP transition probability $p(s', r|s, a)$, discount rate $\gamma < 1$
Input: Stopping criterion
Initialize: $v \in \mathbb{R}^n$
while *stopping criterion not reached* **do**
  $v \leftarrow F(v) = \max_\pi \{\gamma P(\pi)v + b(\pi)\}$
end
return v

We now show that Alg. 10 converges. The proof relies the contraction mapping theorem (Thm. 4.8) introduced earlier.

Theorem 4.12: Convergence of Value Iteration

Let $v_0 \in \mathbb{R}^n$ be arbitrary and define $v_{k+1} = F(v_k)$. Then, we have $v_* = \lim_{k\to} v_k$.

Proof. This follows from the fact that F is a contraction, as proved in Thm. 4.9.

Thm. 4.12 shows that the value iteration algorithm 10 converges. In fact, one can analyze how fast the convergence happens. Given an error tolerance $\epsilon > 0$, how many iterations is required so that $\|v - v_*\| \leq \epsilon$? To answer this, observe that

$$\|v_{k+1} - v_*\|_\infty = \|F(v_k) - F(v_*)\|_\infty \leq \gamma \|v_k - v_*\|_\infty.$$

Hence, $\|v_k - v_*\|_\infty \leq \gamma^k \|v_0 - v_*\|_\infty$. Setting $\|v_k - v_*\|_\infty \leq \epsilon$ gives $k = O\left(\frac{\log(1/\epsilon)}{\log(1/\gamma)}\right)$. In other words, the convergence is exponentially fast. In the optimization literature, this is also called *linear convergence*.

Policy Iteration. Although value iteration converges very quickly, it still theoretically takes an infinite number of iterations to achieve optimality. However, recall that for finite MDPs, there are finite number of deterministic policies, at least one of which must be optimal. Hence,

it is natural to ask whether there is an algorithm that solves the reinforcement learning problem in a *finite* number of steps. The alternative solution method, *policy iteration*, achieves this.

The basis for policy iteration is the policy improvement result in Prop. 4.6, which implies that for any policy π with action value function $q_\pi(s,a)$, the new policy defined by

$$\pi'(s) \in \arg\max_a q_\pi(s,a) \tag{4.37}$$

must have equal or better value, i.e. $v'_\pi \geq v_\pi$. Moreover:

- If there exists at least one s for which $\pi(s) \notin \arg\max_a q_\pi(s,a)$, then π' so obtained is a strictly better policy,
- If for all s, $\pi(s) \in \arg\max_a q_\pi(s,a)$, then π is optimal.

Hence, by iterating the above policy improvement (4.37), we can then achieve optimality in a finite number of steps. The algorithm is summarized in Alg. 11, which essentially weave policy evaluation (E) and improvement (I) steps

$$\pi_0 \xrightarrow{\text{E}} v_{\pi_0} \xrightarrow{\text{I}} \pi_1 \xrightarrow{\text{E}} v_{\pi_1} \xrightarrow{\text{I}} \pi_2 \xrightarrow{\text{E}} \cdots \xrightarrow{\text{I}} \pi_* \xrightarrow{\text{E}} v_* \tag{4.38}$$

We prove its convergence in Thm. 4.13.

Algorithm 11: Policy Iteration Algorithm

Model Parameters: MDP transition probability $p(s', r|s, a)$, discount rate $\gamma < 1$
Initialize: Deterministic policy $\pi \leftarrow \pi_0$
while $\pi \neq \pi'$ **do**
  $\pi \leftarrow \pi'$;
  $v \leftarrow (I - \gamma P(\pi))^{-1} b(\pi)$ (Evaluate v_π, the value of policy π);
  $\pi' \leftarrow \arg\max_\pi \{b(\pi) + \gamma P(\pi) v\}$ (The term inside the max is just q_π);
end
return v

Theorem 4.13: Convergence of Policy Iteration

Let π_k, $k = 0, 1, 2, \ldots$ be a sequence of policies which are updated according to Alg. 11. Then, we have $v_{\pi_k} \leq v_{\pi_{k+1}} \leq v_*$. Moreover, for any π_0, $v_{\pi_k} = v_*$ for all sufficiently large k.

Proof. By the policy improvement result (Prop. 4.6), we know that $v_{\pi_k} \leq v_{\pi_{k+1}} \leq v_*$ for any k. Moreover, if $\pi_k = \pi_{k+1}$ then $v_{\pi_k} = v_*$, and if the last equality does not hold then $\pi_k \neq \pi_{k+1}$, and the value strictly increases. Since there are a finite number of deterministic policies, we must have $v_{\pi_k} = v_*$ after a finite number of steps.

In general, value iteration may converge quickly in terms of the number of iterations, but each iteration is in general more expensive than value iteration, due to the fact that a linear equation of the form $(I - \gamma P(\pi))v_\pi = b(\pi)$ must be solved.

4.3.2 Model-free Algorithms

In previous algorithms on policy evaluation and iteration, it is assumed that $p(r, s'|s, a)$ is a known function. This is reasonable for some small toy systems, but in reality p is often not known to us, due to a variety of reasons:

- Computing/estimating p is intractable due to system size and complexity.
- Reward and state evolutions are from some black-box system.

In this case, it is desirable to ask for model-free algorithms, where we do not need to specify the functional form of p. However, we will still need information from the evolution of states and rewards. Thus, we will assume that we can sample from p, i.e., given a state s and an action a, there exists a possibly black-box simulator that outputs the next state s' and the reward r, which are distributed according to $p(r, s'|s, a)$. We do not know the latter, but we can query samples from this simulator. It turns out that in the model-free context, we can make use of Monte-Carlo simulations to evaluate and improve policies. Our goal is to present various useful algorithms in the model free context, without recourse to the proof of their convergence. The latter requires some advanced tools in probability theory (e.g. Martingales) which are out of scope of these notes. The interested readers can refer to textbooks, e.g. [MRT18], Ch. 17.

Monte-Carlo Policy Evaluation. First, we discuss model-free policy evaluation, i.e. estimating the value of a particular given policy. Instead of solving the Bellman's equation for the value function, we will directly make use of the definition of the value function, namely

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s\right]. \tag{4.39}$$

We will assume time homogeneity and take $t = 0$ in the above. This is written as an expectation, and hence we can estimate it by average over samples. More precisely, using our black-box simulator, we can draw N samples

$$S_0^{(n)} = s, S_1^{(n)}, S_2^{(n)}, \dots \quad \text{and} \quad R_1^{(n)}, R_2^{(n)}, \dots, \quad \text{for} \quad n = 1, 2, \dots, N, \tag{4.40}$$

where at each $S_t^{(n)}$ the action A_t is chosen according to the policy π. Once the action $A_t^{(n)}$, the next states $S_{t+1}^{(n)}$ and reward $R_{t+1}^{(n)}$ are sampled according to our black-box simulator. Each sample trajectory of states and rewards $\{S_t^{(n)}, R_t^{(n)} : t = 0, 1, \dots\}$ is called an *episode*.

By appealing to the law of large numbers, we can approximate the expectation by the sample mean over the returns of these N episodes:

$$v_\pi(s) \approx \frac{1}{N}\sum_{n=1}^{N}[G_0^{(n)}|S_0^{(n)}=s] = \frac{1}{N}\sum_{n=1}^{N}[G_0^{(n)}|S_0^{(n)}=s] = \frac{1}{N}\sum_{n=1}^{N}\left[\sum_{k=0}^{\infty}\gamma^k R_{k+1}^{(n)}\right]. \tag{4.41}$$

Monte-Carlo Policy Improvement. The Monte-Carlo policy improvement method works similarly. Looking at the previous section on model-based methods, it is clear that as long as we can estimate the action value function (4.11). Just like the state value function, we can use sample means over episodes to get

$$q_\pi(s,a) \approx \frac{1}{N}\sum_{n=1}^{N}\left[\sum_{k=0}^{\infty}\gamma^k R_{k+1}^{(n)}\right]. \qquad S_0^{(n)}=s,\ A_0^{(n)}=a \tag{4.42}$$

The policy improvement may then be set as

$$\pi'(s) = \arg\max_a q_\pi(s,a). \tag{4.43}$$

We summarize the model-free policy iteration in Algorithm 12.

Remark. Note that in Algorithm 12, we have resorted to sampling every $s \in \mathcal{S}$ to compute the action value function. In practical scenarios, this may be inefficient since $\mathcal{S}$ may be large. Rather, we can run a very long trajectory, and take any sub-trajectory that starts from s as another independent trajectory in the average. There are two choices: 1) for each unique s_j in the trajectory, we take the first time s_j appears as the starting point and take the path that originates from this starting point as an independent path. This is called *every visit* Monte Carlo. 2) alternatively, we can take every s_j in the long trajectory as a starting point of a sub-trajectory, and this is called called *every visit* Monte Carlo. Both can be shown to also give a good estimate of the value functions, although they have different bias-variance tradeoffs.

Temporal Differencing and Q-Learning. While the previous policy iteration method is easy to implement, one can see that for large state/action spaces, it may not be very efficient. This is because the policy evaluation step can be very expensive: for a larger state/action space, the variance of the Monte-Carlo estimate will be very large. Thus, one may ask if there's a more iterative method, such as the value iteration method that we discussed before, which avoids the computation of value functions given a policy. Rather, we want to iteratively update some estimate of the value function.

The first algorithm along this line of thought is the TD(0) algorithm, which is used to estimate the value function corresponding to a particular π iteratively. In this algorithm, we do not use Monte-Carlo to compute the value function in one go. Instead, suppose at the k^{th} iteration we have an estimate $v_\pi^{(k)}$ of the value function corresponding to policy π.

Algorithm 12: Model-free Policy Iteration Algorithm

Input: State-reward simulator, Sample size N, Stopping criterion
Initialize: Initial policy π, Initial action value function $q(s, a)$
while *stopping criterion not reached* **do**
  Policy evaluation:
  for s *in* $\mathcal{S}$ **do**
    for a *in* $\mathcal{A}(s)$ **do**
      Sample episodes according to π:
      $S_0^{(n)} = s, S_1^{(n)}, S_2^{(n)}, \ldots$ and $R_1^{(n)}, R_2^{(n)}, \ldots$ for $n = 1, 2, \ldots, N$, Set action-value function:
      $q(s, a) \leftarrow \frac{1}{N} \sum_{n=1}^{N} \left[\sum_{k=0}^{\infty} \gamma^k R_{k+1}^{(n)} | S_0^{(n)} = s, A_0^{(n)} = a \right]$.
    end
  end
  Policy improvement:
  for s *in* $\mathcal{S}$ **do**
    $\pi(s) \leftarrow \arg\max_a q(a, s)$
  end
end
return $\pi \approx \pi_*, q \approx q_*$

Then, we sample the next state s' and obtain the reward r from a possibly black-box environment. Then, we set the next iterate as

$$v_\pi^{(k+1)}(s) = (1 - \alpha) v_\pi^{(k)}(s) + \alpha [r + \gamma v_\pi^{(k)}(s')]. \tag{4.44}$$

If we take expectation with respect to s', r and suppose that $v_\pi^{(k)}$ converges, then it is not hard to see that the limit must be the unique solution of the Bellman's equation, i.e. the value function corresponding to π. This can be made rigorous, see [MRT18] Ch 17.5.2.

The reason this is called *temporal differencing* is because we can rewrite (4.44) as

$$v_\pi^{(k+1)}(s) = v_\pi^{(k)}(s) + \alpha [r + \gamma v_\pi^{(k)}(s') - v^{(k)}(s)]. \tag{4.45}$$

Then, we can see that the algorithm works as follows:

- If the current iterate "under-values" s, i.e. the value assigned to s is lower than the immediate reward plus the estimated value of the next state, then we increase its estimated value
- If the current iterate "over-values" s, i.e. the value assigned to s is higher than the immediate reward plus the estimated value of the next state, then we decrease its estimated value
- The last term that decides whether "over-valuing" or "under-valuing" occurs is a temporal difference.

The "0" in TD(0) is to make a distinction with the more general TD(λ) algorithm, which considers the temporal difference of rewards collected for multiple steps. We omit these here for simplicity. The TD(0) algorithm for iterative policy evaluation is summarized in Alg. 13.

Algorithm 13: TD(0) Algorithm for Policy Evaluation

Input: State-reward simulator, initial state sampler, Stopping criterion, Policy π
Initialize: Initial value v_0, Episode length T
$v \leftarrow v_0$;
while *stopping criterion not reached* **do**
  $s \leftarrow$ InitialStateSampler();
  for $t = 0$ *to* $T - 1$ **do**
    $a \leftarrow a \sim \pi(\cdot|s)$;
    $s', r \leftarrow$ StateRewardSimulator(s, a);
    $v(s) \leftarrow (1 - \alpha)v(s) + \alpha[r + \gamma v(s')]$;
    $s \leftarrow s'$
  end
end
return v

In a similar vein, we can form an iterative method that finds an optimal policy via updating estimations of the action value function. This is based on the result we have derived earlier that if q_* is the optimal action value function, then an optimal policy can simply be read off as

$$\pi_*(s) \in \arg\max_a q_*(s, a). \tag{4.46}$$

Moreover, the optimal value function satisfies the Bellman's optimality equation

$$q_*(s, a) = \sum_{s', r} p(s', r|s, a)[r + \gamma \max_{a'} q_*(s', a')]. \tag{4.47}$$

As before, we can replace the expectation over s', r via a sample, and derive the update

$$q(s, a) \leftarrow (1 - \alpha)q(s, a) + \alpha[r + \gamma \max_{a'} q(s', a')], \tag{4.48}$$

where s', r are sampled according to the black-box environment simulator. This is known as *Q-learning*, and is in fact one of the primary ideas driving the success of Alpha-Go. We summarize the Q-learning algorithm in Alg. 14.

4.4 Further Reading

The topic of reinforcement learning is a huge one deserving at least one entire course. For this reason, we have only introduced the bare basics in this chapter. Key topics missing is the class of methods called policy gradient and actor critic methods. These directly optimizes

Algorithm 14: Q-learning Algorithm

Input: State-reward simulator, initial state sampler, Stopping criterion, Policy π
Initialize: Initial value q_0, Episode length T
$q \leftarrow q_0$;
while *stopping criterion not reached* **do**
  $s \leftarrow$ InitialStateSampler();
  for $t = 0$ *to* $T - 1$ **do**
    $a \leftarrow a \sim \pi(\cdot|s)$;
    $s', r \leftarrow$ StateRewardSimulator(s, a);
    $q(s, a) \leftarrow (1 - \alpha)q(s, a) + \alpha[r + \gamma \max_{a'} q(s', a')]$;
    $s \leftarrow s'$;
  end
end
return q

the policy function π, with value function computations as support. Another aspect driving modern deep reinforcement learning is using deep neural networks to estimate value functions and action value functions, which is an interesting combination of supervised learning and reinforcement learning. Nevertheless, these methods inevitably builds on the basic ideas we introduced in this chapter. Instead of specifying a number of references, the reader is referred to the comprehensive texts [Sut99] on reinforcement learning methods and applications. For more theoretical treatment on Markov decision processes and reinforcement learning, the reader may refer to [Put14, MRT18].

Bibliography

[ACH18] Sanjeev Arora, Nadav Cohen, and Elad Hazan. On the optimization of deep networks: Implicit acceleration by overparameterization. *arXiv preprint arXiv:1802.06509*, 2018.

[ADH+19] Sanjeev Arora, Simon S Du, Wei Hu, Zhiyuan Li, and Ruosong Wang. Fine-grained analysis of optimization and generalization for overparameterized two-layer neural networks. *arXiv preprint arXiv:1901.08584*, 2019.

[ALS18] Zeyuan Allen-Zhu, Yuanzhi Li, and Zhao Song. A convergence theory for deep learning via over-parameterization. *arXiv preprint arXiv:1811.03962*, 2018.

[Arn12] Vladimir Igorevich Arnold. *Geometrical Methods in the Theory of Ordinary Differential Equations*, volume 250. Springer Science & Business Media, 2012.

[Aro50] Nachman Aronszajn. Theory of reproducing kernels. *Transactions of the American mathematical society*, 68(3):337–404, 1950.

[Bar93] Andrew R Barron. Universal approximation bounds for superpositions of a sigmoidal function. *IEEE Transactions on Information theory*, 39(3):930–945, 1993.

[Bar94] Andrew R Barron. Approximation and estimation bounds for artificial neural networks. *Machine learning*, 14(1):115–133, 1994.

[BCN18] Léon Bottou, Frank E. Curtis, and Jorge Nocedal. Optimization Methods for Large-Scale Machine Learning. *SIAM Review*, 2018.

[Bel66] Richard Bellman. Dynamic programming. *Science*, 153(3731):34–37, 1966.

[Ber00] Dimitri P Bertsekas. *Dynamic Programming and Optimal Control*, volume 1. Athena Scientific, 2nd edition, 2000.

[BGP61] V G Boltyanskiy, Revaz V Gamkrelidze, and L S Pontryagin. Theory of optimal processes. Technical report, JOINT PUBLICATIONS RESEARCH SERVICE ARLINGTON VA, 1961.

[BH75] A. E. Bryson and Y. Ho. Applied Optimal Control (Bryson).pdf. 1975.

[BO06] Christopher M Bishop and Others. *Pattern Recognition and Machine Learning*, volume 4. springer New York, 2006.

[Bot14] Léon Bottou. From machine learning to machine reasoning. *Machine learning*, 94(2):133–149, 2014.

[Bre01] Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[CG19] Yuan Cao and Quanquan Gu. A generalization theory of gradient descent for learning over-parameterized deep relu networks. *arXiv preprint arXiv:1902.01384*, 2019.

[Cha00] Anindya Chatterjee. An introduction to the proper orthogonal decomposition. *Current science*, pages 808–817, 2000.

[CL82] F L Chernousko and A A Lyubushin. Method of successive approximations for solution of optimal control problems. *Optimal Control Applications and Methods*, 3(2):101–114, 1982.

[Coo18] Yaim Cooper. The loss landscape of overparameterized neural networks. *arXiv preprint arXiv:1804.10200*, 2018.

[CS00] Nello Cristianini and John Shawe-Taylor. *An Introduction to Support Vector Machines and Other Kernel-Based Learning Methods*. Cambridge university press, 2000.

[CV95] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine learning*, 20(3):273–297, 1995.

[CVMG+14] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[Cyb89] George Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314, 1989.

[DBRDB78] Carl De Boor, J. R., and Carl De Boor. A practical guide to splines. In *Mathematics of Computation*, volume 27. springer-verlag New York, 1978.

[DeV98] Ronald A DeVore. Nonlinear approximation. *Acta numerica*, 7:51–150, 1998.

[Don06] David L. Donoho. Compressed sensing. *IEEE Transactions on Information Theory*, pages 1–34, 2006.

[DP07] Alan J. Davies and M. J. D. Powell. Approximation Theory and Methods. In *The Mathematical Gazette*. 2007.

[DZPS18] Simon S Du, Xiyu Zhai, Barnabas Poczos, and Aarti Singh. Gradient descent provably optimizes over-parameterized neural networks. *arXiv preprint arXiv:1810.02054*, 2018.

[EHL+19] Weinan E, Jiequn Han, Qianxiao Li, E Weinan, Jiequn Han, Qianxiao Li, Weinan E, Jiequn Han, and Qianxiao Li. A mean-field optimal control formulation of deep learning. *Research in the Mathematical Sciences*, 6(1):10, 2019.

[EMW19] Weinan E, Chao Ma, and Lei Wu. Barron Spaces and the Compositional Function Spaces for Neural Network Models. *arXiv preprint arXiv:1906.08039*, 2019.

[FHT00] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. Additive logistic regression: A statistical view of boosting (with discussion and a rejoinder by the authors). *The annals of statistics*, 28(2):337–407, 2000.

[FHT01] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. *The Elements of Statistical Learning*, volume 1. Springer series in statistics New York, 2001.

[Fis36] Ronald A Fisher. The use of multiple measurements in taxonomic problems. *Annals of eugenics*, 7(2):179–188, 1936.

[FS96] Yoav Freund and Robert E Schapire. Experiments with a new boosting algorithm. In *Icml*, volume 96, pages 148–156. Citeseer, 1996.

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT press, 2016.

[GVL96] G Golub and C Van Loan. Matrix computations, 3rd edn. J. *Hopkins, London*, 1996.

[Han17] Boris Hanin. Universal Function Approximation by Deep Neural Nets with Bounded Width and ReLU Activations. (1):1–9, 2017.

[Hod88] Michael E. Hodgson. Reducing the computational requirements of the minimum-distance classifier. *Remote sensing of Environment*, 25(1):117–128, 1988.

[Hor91] Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural networks*, 4(2):251–257, 1991.

[Hor93] Kurt Hornik. Some new results on neural network approximation. *Neural networks*, 6(8):1069–1072, 1993.

[HS97] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[HSS08] Thomas Hofmann, Bernhard Schölkopf, and Alexander J. Smola. Kernel methods in machine learning. *Annals of Statistics*, 2008.

[HSW89] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

[HSW90] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Universal approximation of an unknown mapping and its derivatives using multilayer feedforward networks. *Neural networks*, 3(5):551–560, 1990.

[Kar47] Kari Karhunen. Under Lineare Methoden in der Wahr Scheinlichkeitsrechnung. *Annales Academiae Scientiarun Fennicae Series A1: Mathematia Physica*, 47, 1947.

[KT51] Harold W Kuhn and Albert W Tucker. Nonlinear programming, in (J. Neyman, ed.) Proceedings of the Second Berkeley Symposium on Mathematical Statistics and Probability. 1951.

[LCB10] Yann LeCun, Corinna Cortes, and C J Burges. MNIST handwritten digit database. *AT&T Labs [Online]. Available: http://yann. lecun. com/exdb/mnist*, 2:18, 2010.

[LCTE17] Qianxiao Li, Long Chen, Cheng Tai, and Weinan E. Maximum principle based algorithms for deep learning. *The Journal of Machine Learning Research*, 18(1):5998–6026, 2017.

[LDP07] Michael Lustig, David Donoho, and John M. Pauly. Sparse MRI: The application of compressed sensing for rapid MR imaging. *Magnetic Resonance in Medicine*, 2007.

[LeC88] Yann LeCun. A theoretical framework for back-propagation. In *The Connectionist Models Summer School*, volume 1, pages 21–28, 1988.

[LJ18] Hongzhou Lin and Stefanie Jegelka. Resnet with one-neuron hidden layers is a universal approximator. In *Advances in Neural Information Processing Systems*, pages 6169–6178, 2018.

[LL18] Yuanzhi Li and Yingyu Liang. Learning overparameterized neural networks via stochastic gradient descent on structured data. In *Advances in Neural Information Processing Systems*, pages 8157–8166, 2018.

[LPW+17] Zhou Lu, Hongming Pu, Feicheng Wang, Zhiqiang Hu, and Liwei Wang. The expressive power of neural networks: A view from the width. In *Advances in Neural Information Processing Systems*, pages 6231–6239, 2017.

[LTE17] Qianxiao Li, Cheng Tai, and Weinan E. Stochastic modified equations and adaptive stochastic gradient algorithms. In *International Conference on Machine Learning*, 2017.

[LTE19] Qianxiao Li, Cheng Tai, and Weinan E. Stochastic Modified Equations and Dynamics of Stochastic Gradient Algorithms I: Mathematical Foundations. *Journal of Machine Learning Research*, 20(40):1–47, 2019.

[Mac67] James MacQueen. Some methods for classification and analysis of multivariate observations. In *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability*, volume 1, pages 281–297. Oakland, CA, USA, 1967.

[Mal09] Stephane Mallat. A Wavelet Tour of Signal Processing. In *A Wavelet Tour of Signal Processing*. 2009.

[Mal16] Stéphane Mallat. Understanding deep convolutional networks. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 374(2065):20150203, 2016.

[MCCD13] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[Mer09] J. Mercer. Functions of Positive and Negative Type, and their Connection with the Theory of Integral Equations. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 1909.

[MFSS17] Krikamol Muandet, Kenji Fukumizu, Bharath Sriperumbudur, and Bernhard Schölkopf. Kernel mean embedding of distributions: A review and beyond. *Foundations and Trends® in Machine Learning*, 10(1-2):1–141, 2017.

[Mit97] Thomas M Mitchell. *Machine Learning*. McGraw-Hill, Inc., New York, NY, USA, 1 edition, 1997.

[Moo00] Andrew W. Moore. The anchors hierarchy: Using the triangle inequality to survive high dimensional data. In *Proceedings of the Sixteenth Conference on Uncertainty in Artificial Intelligence*, pages 397–405. Morgan Kaufmann Publishers Inc., 2000.

[MP43] Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943.

[MRT18] Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar. *Foundations of Machine Learning*. MIT press, 2018.

[Nes04] Yurii Nesterov. *Introductory Lectures on Convex Optimization*. 2004.

[NWNW06] Jorge Nocedal, Stephen J Wright, Jorge Nocedal, and Stephen J Wright. *Numerical Optimization*, volume 2. Springer Science & Business Media, 2006.

[OS18] Samet Oymak and Mahdi Soltanolkotabi. Overparameterized Nonlinear Learning: Gradient Descent Takes the Shortest Path? *arXiv preprint arXiv:1812.10004*, 2018.

[OS19] Samet Oymak and Mahdi Soltanolkotabi. Towards moderate overparameterization: Global convergence guarantees for training shallow neural networks. *arXiv preprint arXiv:1902.04674*, 2019.

[PJ09] Hae-Sang Park and Chi-Hyuck Jun. A simple and fast algorithm for K-medoids clustering. *Expert systems with applications*, 36(2):3336–3341, 2009.

[Pol90] Jordan B Pollack. Recursive distributed representations. *Artificial Intelligence*, 46(1-2):77–105, 1990.

[Pon18] Lev Semenovich Pontryagin. *Mathematical Theory of Optimal Processes*. Routledge, 2018.

[Put14] Martin L Puterman. *Markov Decision Processes.: Discrete Stochastic Dynamic Programming*. John Wiley & Sons, 2014.

[Ras03] Carl Edward Rasmussen. Gaussian processes in machine learning. In *Summer School on Machine Learning*, pages 63–71. Springer, 2003.

[RM51] Herbert Robbins and Sutton Monro. A stochastic approximation method. *The annals of mathematical statistics*, pages 400–407, 1951.

[RP89] V. Ramasubramanian and K. K. Paliwal. A generalized optimization of the kd tree for fast nearest-neighbour search. In *Fourth IEEE Region 10 International Conference TENCON*, pages 565–568. IEEE, 1989.

[SC04] John Shawe-Taylor and Nello Cristianini. *Kernel Methods for Pattern Analysis*. Cambridge university press, 2004.

[Sha49] Claude Elwood Shannon. Communication in the presence of noise. *Proceedings of the IRE*, 37(1):10–21, 1949.

[SSM97] Bernhard Schölkopf, Alexander Smola, and Klaus-Robert Müller. Kernel principal component analysis. In *International Conference on Artificial Neural Networks*, pages 583–588. Springer, 1997.

[SSM98] Bernhard Schölkopf, Alexander Smola, and Klaus-Robert Müller. Nonlinear component analysis as a kernel eigenvalue problem. *Neural computation*, 10(5):1299–1319, 1998.

[Sto48] Marshall H. Stone. The Generalized Weierstrass Approximation Theorem. *Mathematics Magazine*, pages 237–254, 1948.

[Sut99] A. G. Sutton, R.S and Barto. Reinforcement Learning: An Introduction, by Sutton, R.S. and Barto, A.G. *Trends in Cognitive Sciences*, 3(9):360, 1999.

[SW89] Maxwell Stinchcombe and Halbert White. Universal approximation using feedforward networks with non-sigmoid hidden layer activation functions. In *IJCNN International Joint Conference on Neural Networks*, 1989.

[SW90] Maxwell Stinchcombe and Halbert White. Approximating and learning unknown mappings using multilayer feedforward networks with bounded weights. In *1990 IJCNN International Joint Conference on Neural Networks*, pages 7–16. IEEE, 1990.

[SYZ19] Zuowei Shen, Haizhao Yang, and Shijun Zhang. Nonlinear approximation via compositions. *arXiv preprint arXiv:1902.10170*, 2019.

[TBI97] Lloyd N Trefethen and David Bau III. *Numerical Linear Algebra*, volume 50. Siam, 1997.

[Tur50] Alan M Turing. Computing machinery and intelligence. *Mind*, 49:23–65, 1950.

[VSP+17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, pages 5998–6008, 2017.

[Wer90] Paul J Werbos. Backpropagation through time: What it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, 1990.

[WH60] Bernard Widrow and Marcian E Hoff. Adaptive switching circuits. Technical report, Stanford Univ Ca Stanford Electronics Labs, 1960.

[WPC+19] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S Yu. A comprehensive survey on graph neural networks. *arXiv preprint arXiv:1901.00596*, 2019.

[ZBH+16] Chiyuan Zhang, Samy Bengio, Moritz Hardt, Benjamin Recht, and Oriol Vinyals. Understanding deep learning requires rethinking generalization. *arXiv preprint arXiv:1611.03530*, 2016.

[Zho19] Ding-Xuan Zhou. Universality of deep convolutional neural networks. *Applied and Computational Harmonic Analysis*, 2019.