Mathematical Foundations of Machine Learning

Lecture notes by:
Maria Han Veiga
François Gaston Ged

v0.0.1

November 19, 2022

These notes are based on the class MATH498: Modern Topics in Mathematics – Mathematical Foundations of Machine Learning at the University of Michigan, Fall 2021 taught by Maria Han Veiga.

Contents

Chapter 1

Introduction

These notes are aimed at being an introduction to machine learning, with **a stronger focus on the mathematics behind a lot of the algorithms and techniques**. While these are based on a math course, I still want to give you the opportunity to do stuff hands-on, so it's important that we take a computer and also learn how to solve specific machine learning problems. I am not sure what is it that you want to do in your career or in the future, so I want that these notes give you the tools to be able to understand machine learning, read ML papers, both theoretical and applied, as well as use ML as another tool that you can use to solve problems, similarly to approximation theory, calculus, analysis.

I hope these notes leaves you well equipped to speak about ML, inspires you to start doing research in ML, or to apply to jobs that ask for ML. We will focus both on theory, as well as applied machine learning.

The second iteration of these notes are work in progress and a collaboration between Maria Han Veiga (PhD Applied Mathematics) and François Ged (PhD Probability Theory). We write these notes in a way that we believe is helpful for mathematicians to understand the fundamental principles of Machine Learning. Because these notes are aimed at senior undergraduates and early graduate students of Mathematics, some content will be under

$\star$ **Remark 1. Star remarks**, which denote remarks on possibly relevant mathematical information/more formal theory, which is not part of the scope of the course.

and

Hand Wavy Remark 1. which denote remarks which are more intuition / informal observations.

You can imagine who is more likely to use which type of remark.

Chapter 2

Probability review

Contents

In our journey to understand machine learning, we will encounter several sources of randomness, such as those coming from the collected data, which are usually random observations (i.e. samples) of some unknown probability distribution, the initial parameters of the model we will use to make predictions, or the intrinsic randomness of some training algorithms. To build a solid theory, we need some knowledge of probability theory and this is what this chapter is about.

2.1 Motivation

Let us consider the task of *binary classification*, where we wish to learn a mapping from inputs $x \in \mathcal{X}$ (we also call these *features*) to outputs $\mathcal{Y} = \{-1, +1\}$ [1]. We can formalise the problem as a function approximation problem. Given a labeled training set $\{(x_i, y_i); i = 1 \cdots m\}$, we assume that there is some unknown function $f : \mathcal{X} \to \mathcal{Y}$ such that $f(x_i) = y_i$ for all $i = 1 \cdots m$, and the goal of learning is to approximate the function f by a function $\hat{f} : \mathcal{X} \to \{-1, 1\}$. Then, for any input $x \in \mathcal{X}$, one can make a prediction of its label using $\hat{y} = \hat{f}(x)$. This is what it is called a *discriminative model*, as its goal is to discriminate between different classes.

One can think of spam detection, in this case, x is an email (or representation of an email) and -1 denotes spam, $+1$ denotes not spam.

However, to set a milder decision rule, one might prefer to estimate the probability that the email x is a spam, and only warn the user the email is potentially a spam if this probability is larger than some chosen threshold. Having a probability estimate of class belonging is even more important when the number of classes is larger than two.

Instead of discriminating whether x belongs to some class y, one might want to create an object x that belongs to a given class y. This is the goal of a *generative model* that tries to learn the conditional probability $p(x|y)$ instead.

Besides the possibility of making stochastic predictors and stochastic generators, other sources of randomness are more broadly encountered when practicing Machine Learning. Namely, it is commonly assumed that the dataset is randomly generated by some unknown probability distribution. On the other hand, by using parametric models, one needs to set a value for the initial parameters before training, and it is common to initialise them at random values. Finally, the training procedure (i.e. the update of the parameters to fit the data) itself can include some randomness. Hence, a good understanding of machine learning requires some basic knowledge of probability theory.

[1] We will define more rigorously what $\mathcal{X}$ is later on

2.2 Probability space

Probability theory is a mathematical framework that allows us to reason about phenomena or experiments under uncertainty. A probabilistic model is a mathematical model of a probabilistic experiment that satisfies the axioms of probability theory, and allows us to calculate probabilities as well as to reason about the outcomes of an experiment.

Definition 2.2.1. A *probability space* is a triplet $(\Omega, \mathcal{F}, P)$, where Ω is an arbitrary set, $\mathcal{F}$ is a σ-field of subsets of Ω, and P is a measure on $\mathcal{F}$ such that

$$P(\Omega) = 1,$$

called probability measure (or in short, probability).

This means that:

- Ω is called the *sample space*, it is the set of all possible outcomes,
- $\mathcal{F}$ is a collection of subsets of Ω, $\sigma-$field[2],
- the probability P maps elements of $\mathcal{F}$ (subsets of Ω) onto the real interval $[0, 1]$,
- an element $A \in \mathcal{F}$ is called an event, and $P(A) \in [0, 1]$ is the probability that A occurs.

Example 2.2.1. Suppose we toss a fair coin twice and we observe the outcome (the two tosses are independent). We have

- $\Omega = \{HH, TT, HT, TH\}$
- $\mathcal{F} = \{\{\}\{HH\}, \{TT\}, \{HT\}, \{TH\}, \{HH, TT\}, \{HH, HT\}, ..., \Omega\}$, $|\mathcal{F}| = 2^{|\Omega|} = 2^4 = 16$

[2]It must contain Ω, and it must be closed by complementation and by countable unions to be a σ-field; we will not manipulate σ-algebras in this class and therefore we do not dwell on it to keep the focus on the necessary concepts

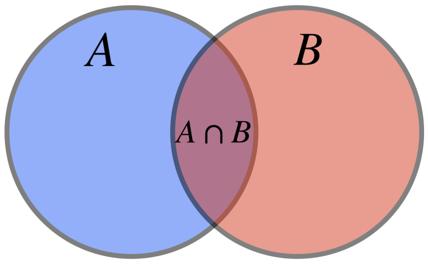


Figure 2.1: Schematic of sample space Ω, event A and probability of A, $P(A)$.

Events that depend on the outcome of the experiment can be written as elements of the σ-field $\mathcal{F}$. For example, the event "having exactly one head during those two tosses" is the element $\{HT, TH\}$. Probabilities of events in $\mathcal{F}$ are assigned by the probability measure P. We have for example $P(\{\}) = 0$ (always true for any probability measure by definition), and because all outcomes are equally likely (the coin is fair), we have

$$P(\text{“getting exactly one head”}) = \frac{\#\text{ outcomes with exactly one head}}{\#\text{ number of possible outcomes}} = \frac{2}{4} = \frac{1}{2}$$

.

2.3 Independence and Conditioning

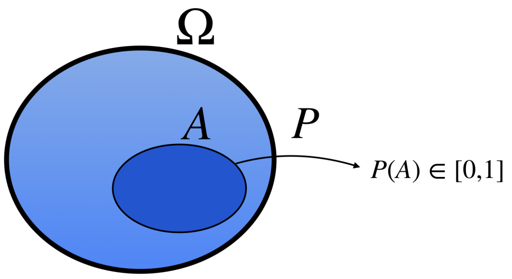
We say that two events $A, B \in \mathcal{F}$ are independent if the occurrence or non-occurrence of one does not affect the probability assigned to the other. More formally, we can state

- A and B are independent if $P(A \cap B) = P(A)P(B)$.
- The events in a set $\{A_s | s \in S\} \subset \mathcal{F}$ are independent if for every finite subset $S_0 \subset S$ we have

$$P\left(\bigcap_{s \in S_0} A_s\right) = \prod_{s \in S_0} P(A_s). \tag{2.1}$$

To be more precise, the last point is the property of *mutual independence* for a family of events. It is strictly stronger than the property of pairwise independence that A_s and $A_{s'}$ must be independent for all $s \neq s'$ in S.

Example 2.3.1. I tossed a fair coin 8 times (tosses are assumed independent) and I got head 8 times, what's the probability I get tails in my next throw?



Let $B \in \mathcal{F}$ with $P(B) > 0$. For every event $A \in \mathcal{F}$, the conditional probability that A occurs conditionally given that B occurs is denoted by $P(A|B)$ and defined by

$$P(A|B) = \frac{P(A \cap B)}{P(B)}. \tag{2.2}$$

Example 2.3.2. Given I threw a fair dice and I got an even number, what is the probability it was 2? What about 3?

- A: {"Getting 2"}
- B: {"Getting even"}
- $A \cap B$: {"Getting 2"} and {"Getting even"} = {"Getting 2"}

$$P(A|B) = \frac{P(A \cap B)}{P(B)} = \frac{1/6}{1/2} = 1/3$$

The following properties can also be proven:

- If $P(B) > 0$ and $\{A_i\}_{i\geq 1}$ are pairwise disjoint events, then

$$P\left(\bigcup_{i=1}^{\infty} A_i | B\right) = \sum_{i=1}^{\infty} P(A_i|B). \tag{2.3}$$

- If $B \in \mathcal{F}$ is such that $P(B) > 0$ and $P(B^c) > 0$, where B^c denotes the complement $B^c = \Omega \setminus B$, then

$$P(A) = P(A|B)P(B) + P(A|B^c)P(B^c), \tag{2.4}$$

  for all $A \in \mathcal{F}$.

- If $(B_i)_{i\geq 1} \subset \mathcal{F}$ form a partition of Ω (i.e. they are pairwise disjoint and cover Ω) and $P(B_i) > 0$ for all $i \geq 1$, then for all $A \in \mathcal{F}$

$$P(A) = \sum_{i=1}^{\infty} P(A|B_i)P(B_i). \tag{2.5}$$

2.4 Random Variables

A real random variable provides us with a numerical value that is dependent on the outcome of an experiment. It is a convenient way to express the elements of Ω as numbers rather than abstract elements of sets.

Throughout this course, we will only consider **real** random variables or **multivariate real** random variables, that is to say, random variables with values in $\mathbb{R}$ or $\mathbb{R}^d$ for $d \geq 2$.

Definition 2.4.1. Real Random Variable Let $\mathcal{G}$ be a σ-field on $\mathbb{R}$. A real random variable X is a function $X : \Omega \to \mathbb{R}$, such that for all $B \in \mathcal{G}$, it holds that $\{\omega \in \Omega : X(\omega) \in B\} \in \mathcal{F}$.

$\star$ **Remark 2.** Note that defining a real random variable makes sense only given a probability space $(\Omega, \mathcal{F}, P)$ and a σ-field $\mathcal{G}$ on $\mathbb{R}$. If not specified, it is common to assume that $\mathcal{G}$ is the Borel σ-field on $\mathbb{R}$. (Recall that the Borel σ-field is generated by all sub-intervals of the form $(a, b]$ for all $a, b \in \mathbb{R}$.)

The event $\{\omega \in \Omega : X(\omega) \in B\}$ in the definition is simply the preimage of B by X, also denoted by $X^{-1}(B)$.

The above definition naturally extends to $X : \Omega \mapsto \mathbb{R}^d$ for all $d \geq 2$.

Example 2.4.1. Toss independently n fair coins and observe the resulting sequence. The state space consists of the set of all 2^n possible coin sequences. Let our random variable X be the number of heads. [3] For $k \geq 0$, what is $P(X = k)$? Meaning, in n coin tosses, what is the probability we get exactly k heads?

Example 2.4.2. Toss 2 independent and fair coins. Our random variable X is the number of heads.

- $X(\text{“}HH\text{”}) = 2$
- $X(\text{“}HT\text{”}) = 1$
- $X(\text{“}TH\text{”}) = 1$
- $X(\text{“}TT\text{”}) = 0$

Perhaps in more practical terms, a real random variable transforms an element ω from the original sample space (which could be abstract or difficult to work with directly) into a numerical quantity $X(\omega)$ (real number) that is more convenient or tangible to work with (e.g. quantities that we may measure in a laboratory).

Once we are mapped by X from Ω to $\mathbb{R}$, we may choose to view $\mathbb{R}$ as the new “sample space” and create a new probability triple for itself. Then, on this new measurable space $(\mathbb{R}, \mathcal{B})$, we can assign probabilities to the events in $\mathcal{B}$, and we call that the *probability law* for random variable X, denoted by P_X. The new probability space associated with the random variable X is then $(\mathbb{R}, \mathcal{B}, P_X)$. We can also map from $\mathbb{R}$ back to the original sample space via X^{-1} (note this is the “preimage” inverse, not the usual inverse function). Hence, we can write, for some event B in the new σ-field $\mathcal{B}$,

$$P_X(B) = P(X^{-1}(B)) = P(\{\omega : X(\omega) \in B\}). \tag{2.6}$$

[3]Convince yourself that X is indeed a random variable.

2.5 Discrete Random Variables

2.5.1 Definition

A *discrete* random variable is one whose range $X(\Omega)$ (i.e., the set of values it can take) is countable. The *probability mass function (PMF)* of a discrete random variable is defined as

$$p_X(x) = P(X = x), \qquad \forall x \in \mathbb{R}, \tag{2.7}$$

and in particular,

$$\sum_{x \in \mathbb{R}} p_X(x) = 1. \tag{2.8}$$

(In the above sum, only countably many terms are non-null.) With a slight abuse of language, we say that a random variable X has distribution, or law, p_X [4].

2.5.2 Examples of discrete random variables

- ($X \sim \mathcal{U}(\{a, \ldots, b\})$) Discrete uniform with integer parameters $a < b$, (e.g.: throwing a fair dice):

$$p_X(x) = \frac{1}{b-a} \text{ if } x \in \{a, \ldots, b\} \text{ and } p_X(x) = 0 \text{ otherwise.}$$

- ($X \sim \text{Ber}(p)$) Bernoulli with parameter $0 \le p \le 1$; (e.g.: yes or no):

$$p_X(k) = p^k (1-p)^{1-k}, \quad k \in \{0,1\} \text{ and } p_X(k) = 0 \text{ otherwise}$$

- ($X \sim \text{Bin}(n, p)$) Binomial with parameters $n \in \mathbb{N}$ and $p \in [0,1]$; (e.g.: number of heads after n independent coin tosses with a biased coin):

$$p_X(k) = \binom{n}{k} p^k (1-p)^{n-k} \text{ for } k = 0, 1, \ldots, n \text{ and } p_X(k) = 0 \text{ otherwise}$$

[4] Even though technically its law is P_X defined in the previous section (that is because P_X characterizes p_X).

- ($X \sim \text{Pois}(\lambda)$) Poisson with parameter $\lambda > 0$:

  $$p_X(k) = e^{-\lambda}\lambda^k/k! \text{ for } k = 0, 1, \dots \text{ and } p_X(k) = 0 \text{ otherwise.}$$

  The Poisson distribution is typically used to model the number of times an event occurs in a unit amount of time when these occurrence are thought to be independent and when the rate of occurrence is $1/\lambda$, so that the average number of occurrence per unit of time is λ (e.g. earthquakes, victory of France in the soccer world cup[5], ...).

2.5.3 Multiple random variables (marginal, joint, conditional, independence)

Consider two discrete random variables X and Y associated with the same experiment. The probability law of each one of them is described by its respective PMF p_X or p_Y, called the *marginal PMFs* of the couple (X, Y). The marginal PMFs do not provide any information on possible relations between these two random variables.

The statistical properties of two random variables X and Y are captured by their *joint PMF*:

$$p_{X,Y}(x, y) = P(X = x, Y = y). \tag{2.9}$$

Example 2.5.1. Toss a fair coin and let $X = 1$ if the result is head, $X = 0$ if it is tail. Let $Y = X$ and $Y' = 1 - X$. Show that (X, Y) and (X, Y') have the same marginal PMFs but not the same joint PMFs.

We may also concatenate multiple random variables together into a random vector $X = [X_1, \dots, X_n]$ and still use the notation $p_X(x)$ but with the understanding that it is the joint PMF (in this case, $x = (x_1, \dots, x_n)$). From the joint PMF, we can recover the marginals via

$$p_X(x) = \sum_y p_{X,Y}(x, y), \qquad p_Y(y) = \sum_x p_{X,Y}(x, y). \tag{2.10}$$

[5]in this case λ is very large...

The *conditional PMF* of X given Y is

$$p_{X|Y}(x|y) = P(X = x|Y = y), \tag{2.11}$$

which is well defined whenever $p_Y(y) > 0$. Using the definition of conditional probabilities, we obtain

$$p_{X|Y}(x|y) = \frac{p_{X,Y}(x, y)}{p_Y(y)}. \tag{2.12}$$

Visually, if we fix y, then the conditional PMF $p_{X|Y}(x|y)$ can be viewed as a "slice" of the joint PMF $p_{X,Y}$, but normalized so that it sums to one.

Random variables X and Y are said to be *independent* if for all $x, y \in \mathbb{R}$, it holds that

$$p_{X,Y}(x, y) = p_X(x)p_Y(y), \tag{2.13}$$

or equivalently for all $x, y \in \mathbb{R}$ such that $p_Y(y) > 0$,

$$p_{X|Y}(x|y) = p_X(x). \tag{2.14}$$

Furthermore, if X and Y are independent, then functions of these random variables $h(X)$ and $g(Y)$ are also independent, provided that $h(X)$ and $g(Y)$ are random variables. (One indeed needs to check that they satisfy Definition 2.4.1; for example, it is always the case when h and g are continuous).

2.5.4 Sequence of discrete random variables

We just saw how the joint PMF encodes the distribution of a couple of discrete random variables. Sometimes we will want to consider more than two random variables at the same time. A *sequence of random variables* $(X_i)_{i\geq 1}$ is a sequence such that for all $i \geq 1$, X_i is a random variable.

We say that the random variables in $(X_i)_{i\geq 1}$ are independent if and only if for all $k \geq 1$ and all $i_1, \dots, i_k \in \mathbb{N}$ pairwise distinct, it holds that

$$p_{(X_{i_1},\dots,X_{i_k})}(x_1, \dots, x_k) = \prod_{\ell=1}^{k} p_{X_{i_\ell}}(x_\ell), \quad \forall x_1, \dots, x_k \in \mathbb{R}.$$

This is sometimes called *mutual independence* to stress that it is strictly stronger that *pairwise independence*, the latter being the weaker property that for all $i \neq j$, the two random variables X_i, X_j are independent

Example 2.5.2. Let X be a random variable on $\{0, 1\}$ with the following law: $P(X = 1) = 1/2 = P(X = 0)$. Let Y have the same law and be independent from X. Let Z be a random variable such that $Z = X$ if $Y = 1$, and $Z = 1 - X$ if $Y = 0$. One can show that the triplet (X, Y, Z) is composed of pairwise independent variables, but not mutually independent.

2.6 Continuous Random Variables

Most of the properties and concepts for continuous random variables will be the same or analogous to its discrete counterpart (by swapping summation with integration).

2.6.1 Definitions

When X takes real continuous values it is more natural to specify the probability of X being inside some interval $\mathbb{P}(a \leq X \leq b),\ a < b$. By convention, we specify $\mathbb{P}(X \leq x)$ for all $x \in \mathbb{R}$, which is known as the *cumulative distribution function* (CDF) of X, denoted by $F_X(x)$.

A *continuous* (real) random variable is one that has a *probability density function (PDF)* $f_X(x)$ such that

$$F_X(x) = P(X \leq x) = \int_{-\infty}^{x} f_X(t)\, dt. \tag{2.15}$$

Conversely, if the CDF is differentiable (not always true), then

$$f_X(x) = \frac{\partial F_X(x)}{\partial x}. \tag{2.16}$$

Since the CDF is monotonically increasing, then $f_X(x) \geq 0$; and since

$\lim_{x\to+\infty} F_X(x) = 1$, then

$$\int_{-\infty}^{+\infty} f_X(t)\,dt = 1. \tag{2.17}$$

Using the PDF of a continuous random variable, we can compute the probability of various subsets of the real line:

$$P(a < X < b) = P(a \leq X \leq b) = \int_a^b f_X(t)\,dt, \tag{2.18}$$

$$P(X \in B) = \int_B f_X(x)\,dx. \tag{2.19}$$

$\star$ **Remark 3.** From measure theory, for the last equation to make sense, we need B to be *Lebesgue measurable*. Since it is the case of all Borel sets, we can use this formula for all B that can be constructed from intervals. We shall always work with such measurable sets throughout the class, without necessary recalling it.

$\star$ **Remark 4.** Any random variable can be decomposed into a continuous part and a singular part (that does not need to be discrete but could be). For example, $X = 0$ with probability $1/2$ and $X = U \sim \mathcal{U}(0,1)$ (the uniform distribution on $(0,1)$) with probability $1/2$, then X is neither continuous nor discrete.

2.6.2 Some examples of continuous random variables

- ($X \sim \mathcal{U}(a,b)$) Uniform with parameters a and b (and $a < b$);

  $$f_X(x) = 1/(b-a) \text{ for } x \in [a,b]$$

  The probability law of a uniform random variable on $[a,b]$ is the Lebesgue measure on $[a,b]$ divided by $b-a$.

- ($X \sim \operatorname{Exp}(\lambda)$) Exponential with $\lambda > 0$,

  $$f_X(x) = \lambda e^{-\lambda x}.$$

  The exponential distribution is memoryless, that is

  $$\mathbb{P}(X > x+t | X > x) = \mathbb{P}(X > t),$$

  for all $x, t > 0$.

- ($X \sim \mathcal{N}(\mu, \sigma^2)$) Normal (Gaussian) with mean $\mu \in \mathbb{R}$ and standard deviation $\sigma > 0$;

$$f_X(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}. \tag{2.20}$$

  The distribution $\mathcal{N}(0,1)$ is called the *standard normal.*

2.6.3 Multiple random variables (marginal, joint, conditional, independence)

The joint CDF between X and Y is

$$F_{X,Y}(x,y) = P(X \le x, Y \le y) = \int_{-\infty}^{x} \int_{-\infty}^{y} f_{X,Y}(u,v)\, du\, dv \tag{2.21}$$

and $f_{X,Y}$ is called the joint PDF. If the CDF is differentiable (not always true), then

$$\frac{\partial^2 F_{X,Y}}{\partial x \partial y}(x,y) = f_{X,Y}(x,y). \tag{2.22}$$

Similar to the univariate case, we can compute the probability of an event B with

$$P((X,Y) \in B) = \int_B f_{X,Y}(x,y)\, dx\, dy. \tag{2.23}$$

The marginal PDF of X is then

$$f_X(x) = \int_{-\infty}^{\infty} f_{X,Y}(x,y)\, dy. \tag{2.24}$$

The conditional CDF of X given Y is

$$F_{X|Y}(x|y) = \int_{-\infty}^{x} \frac{f_{X,Y}(u,y)}{f_Y(y)} du \tag{2.25}$$

where f_Y is the marginal PDF of Y and we assumed $f_Y(y) > 0$. The conditional PDF is then

$$f_{X|Y}(x|y) = \frac{f_{X,Y}(x,y)}{f_Y(y)}. \tag{2.26}$$

We say that X and Y are independent if and only if their joint CDF, equivalently joint PDF, can be factored:

$$F_{X,Y}(x,y) = F_X(x)F_Y(y) \tag{2.27}$$

$$f_{X,Y}(x,y) = f_X(x)f_Y(y), \qquad \forall x,y \in \mathbb{R}. \tag{2.28}$$

Equivalently, for all $x,y \in \mathbb{R}$ such that $f_Y(y) > 0$,

$$F_{X|Y}(x|y) = F_X(x) \tag{2.29}$$

$$f_{X|Y}(x|y) = f_X(x). \tag{2.30}$$

2.6.4 Sequence of continuous random variables

Sequences of continuous random variables are defined identically to sequences of discrete random variables. The mutual independence property translates similarly with the PDFs and CDFs as follows: we say that a sequence of continuous random variables are mutually independent if and only if for all $k \geq 1$ and all $i_1, \ldots, i_k \in \mathbb{N}$ pairwise distinct, it holds that

$$F_{X_{i_1},\ldots,X_{i_k}}(x_1,\ldots,x_k) = \prod_{\ell=1}^{k} F_{X_{i_\ell}}(x_\ell),$$

$$\text{or equivalently} \quad f_{X_{i_1},\ldots,X_{i_k}}(x_1,\ldots,x_k) = \prod_{\ell=1}^{k} F_f X_{i_\ell}(x_\ell),$$

for all $x_1, \ldots, x_k$.

2.7 Moments

The probability density function of a continuous or probability mass function of a discrete random variable X provides us the probabilities of all the possible values of X. It is often desirable to summarize this information in a single representative number.

In order to do so, one can look at the average value of X, if one were to sample it many times. This value (that we call the *expectation* of X and that we define below) requires that the variable does not take extremely large values too often, otherwise this average may explode and thus be ill defined. We formalise the property of a variable being non-extreme as *integrability.*

2.7.1 Non-negative random variables

To compute a mean under a probability distribution P, we need to integrate a map against P. We first properly define integrals on functions taking non-negative values. The case of non-positive functions is identical. Then, for general functions, we will naturally obtain an integrability condition to define its integral as the difference between its positive and its negative parts.

We have seen that a real random variable X is a function from a probability space $(\Omega, \mathcal{F}, P)$ to $(\mathbb{R}, \mathcal{G})$ (where, often, $\mathcal{G}$ is chosen as the Borel σ-field on $\mathbb{R}$). We have seen that this defines a probability measure p_X on $\mathbb{R}$ that is called the law (or distribution) of X. Hence, using p_X, we can directly compute integrals on $\mathbb{R}$ without explicitly defining $(\Omega, \mathcal{F}, P)$, e.g. to compute the average value that X takes.

Definition 2.7.1. Suppose X is a non-negative random variable, that is $P(X \geq 0) = 1$. The *expectation* of X is defined by

$$\mathbb{E}[X] = \sum_x x\, p_X(x), \tag{2.31}$$

if X is discrete, and

$$\mathbb{E}[X] = \int_{-\infty}^{\infty} x f_X(x) dx, \tag{2.32}$$

if X is continuous.

Remark 1. The case of random variables that are neither discrete nor continuous is out of the scope of this class.

Remark 2. Note that $\mathbb{E}[X]$ does not have to be finite, in which case $\mathbb{E}[X] = +\infty$ so that $\mathbb{E}[X]$ is always well defined (as long as X is non-negative).

Instead of integrating X, one can integrate $g(X)$ where $g : \mathbb{R} \to \mathbb{R}_+$. The only restriction is that $g(X)$ is a random variable; this depends on g and this ios the case for all the maps we will consider in this class. (For example, it is true as soon as g is piecewise continuous.) Hence, for **any** real random variable X and g such that $g(X)$ is a random variable, the following is well defined:

$$\mathbb{E}[g(X)] = \sum_x g(x)p_X(x),$$

if X is discrete and

$$\mathbb{E}[g(X)] = \int_{-\infty}^{\infty} g(x)f_X(x)dx$$

if X is continuous.

$\star$ **Remark 5.** The expectation $\mathbb{E}[g(X)]$ can equivalently be written on the canonical probability space $(\Omega, \mathcal{F}, P)$ as follows:

$$\mathbb{E}[g(X)] = \int_\Omega g(X(\omega))P(d\omega).$$

2.7.2 The case of signed random variables

In the previous section, we introduced the expectation of a non-negative random variable and assigned it a well-defined value. Identical arguments can be made to define the expectation of non-positive variables. In this section, we aim at defining $\mathbb{E}[X]$ for a random variable X that can take signed values, that is $P[X > 0] > 0$ and $P[X < 0] > 0$.

If we try to define the expectation as in the previous section, a problem will arise with infinite values. Let us see an example. Let X be such that for all $k \in \mathbb{N}$, $p_X(k) = p_X(-k) = \frac{3}{\pi^2} \cdot \frac{1}{k^2}$ and $p_X(x) = 0$ for all $x \notin \mathbb{N} \cup -\mathbb{N}$. One can check that $\sum_{k \in \mathbb{N} \cup -\mathbb{N}} p_X(k) = 1$ so that p_X is indeed a probability measure. However, if we try to define $\mathbb{E}[X]$ as $\sum_{k\in\mathbb{N}\cup NN} kp_X(k)$, the positive and negative parts will be

$$\frac{3}{\pi^2}\sum_{k\in\mathbb{N}}\frac{k}{k^2} = +\infty \tag{2.33}$$

$$\text{and} \qquad \frac{3}{\pi^2}\sum_{k\in-\mathbb{N}}\frac{k}{k^2} = -\infty. \tag{2.34}$$

Since $+\infty - \infty$ is ill defined, we cannot make sense of $\mathbb{E}[X]$ in that case.

In order to avoid this issue, we need to restrict ourselves to *integrable* random variables: Let $X_+ = X\mathbf{1}_{X\geq 0}$ and $X_- = |X|\mathbf{1}_{X<0}$ so that $X = X_+ - X_-$ (note that both X_+ and X_- are non-negative random variables).

Definition 2.7.2. We say that a real random variable X is integrable if $\mathbb{E}[X_+], \mathbb{E}[X_-] < +\infty$. In this case, we define

$$\mathbb{E}[X] = \mathbb{E}[X_+] - \mathbb{E}[X_-]. \tag{2.35}$$

The integrability condition is often written $\mathbb{E}[|X|] < \infty$ since $\mathbb{E}[|X|] = \mathbb{E}[X_+] + \mathbb{E}[X_-]$, by linearity of the expectation (which follows from the linearity of the integral).

Remark 3. When exactly one of $\mathbb{E}[X_+]$ and $\mathbb{E}[X_-]$ is infinite, then it is possible to define $\mathbb{E}[X] = +\infty$ or $-\infty$ according to whether X_+ or X_- is not integrable. In this case, even though $\mathbb{E}[X]$ is well defined (infinite), X is not integrable.

Below we list the basic most important properties for expectations, where X, Y are integrable:

- If $X \geq 0$ then $\mathbb{E}[X] \geq 0$ (Monotonicity)
- $|\mathbb{E}[X]| \leq \mathbb{E}[|X|]$ (triangle inequality)
- If X and Y are integrable, then $\mathbb{E}[aX + bY] = a\mathbb{E}[X] + b\mathbb{E}[Y]$ for all $a, b \in \mathbb{R}$. (Linearity)
- If $X = c$ then $\mathbb{E}[X] = c$.

- For any event $A \in \mathcal{F}$, we have $P(A) = \mathbb{E}[1_A]$.
- If X and Y are square integrable, then $\mathbb{E}[|XY|] \le \sqrt{\mathbb{E}[X^2]\mathbb{E}[Y^2]}$, with equality if and only if $X = aY$ for some $a \in \mathbb{R}$. (Cauchy-Schwarz Inequality)

For a random variable X with $\mathbb{E}[X^2] < \infty$, that is, X is square integrable, we can define its variance as

$$var(X) = \mathbb{E}[(X - \mathbb{E}[X])^2] = \mathbb{E}[X^2] - \mathbb{E}[X]^2.$$

- The square root of the variance is the *standard deviation*, often denoted by σ_X or just σ.
- $var(aX) = a^2 var(X)$.
- If X and Y are independent and square integrable, then $\mathbb{E}[XY] = \mathbb{E}[X]\mathbb{E}[Y]$ and $var(X + Y) = var(X) + var(Y)$.

The *covariance* of two square integrable variables X and Y is defined as

$$Cov(X, Y) = \mathbb{E}\left[(X - \mathbb{E}[X])(Y - \mathbb{E}[Y])\right] = \mathbb{E}[XY] - \mathbb{E}[X]\mathbb{E}[Y]. \qquad (2.36)$$

When $Cov(X, Y) = 0$, we say that X and Y are *uncorrelated.*

Independence implies uncorrelated, but uncorrelated does not imply independence. By rescaling the covariance, we obtain the *correlation* (assuming that none of X, Y is deterministic):

$$\rho(X, Y) = \frac{Cov(X, Y)}{\sqrt{Var(X)Var(Y)}} \quad \in [-1, 1]. \qquad (2.37)$$

The correlation $\rho(X, Y)$ can be thought of as a measure of the linear dependence between X and Y.

Remark 4. The fact that the correlation always belongs to $[-1, 1]$ is proven using Cauchy-Schwarz Inequality. In particular, the equality case of this inequality entails that $\rho(X, Y) = 1$ (resp. -1) if and only if $Y = aX$ for some real $a > 0$ (resp. $a < 0$).

In general, with two discrete integrable random variables, we can form the joint expectation

$$\mathbb{E}[g(X,Y)] = \sum_x \sum_y g(x,y)\, p_{X,Y}(x,y). \tag{2.38}$$

The *conditional expectation* of X given Y is defined to be

$$\mathbb{E}[X|Y=y] = \sum_x x\, p_{X|Y}(x|y). \tag{2.39}$$

The *total expectation theorem* can be derived as well:

$$\sum_y \mathbb{E}[X|Y=y]\, p_Y(y) = \mathbb{E}[X]. \tag{2.40}$$

For the case of two continuous random variables, we have the joint expectation

$$\mathbb{E}[g(X,Y)] = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} g(x,y)\, f_{X,Y}(x,y)dxdy. \tag{2.41}$$

The *conditional expectation* of X given Y is defined to be

$$\mathbb{E}[X|Y=y] = \int_{-\infty}^{\infty} x\, f_{X|Y}(x|y)dx \tag{2.42}$$

The case where X is discrete and Y is continuous is similar with the integral over the values of X replaced by a sum.

For a real $p > 1$, if X^p is integrable, its p-th moment is defined as

$$\mathbb{E}[X^p]. \tag{2.43}$$

For any $p > q \geq 1$, it holds that if X^p is integrable, then X^q is integrable. Hence, the $p \geq 1$ such that X^p is integrable is an interval (possibly empty).

$\star$ **Remark 6.** The last statement about the integrability of X^q implied by that of X^p is a consequence of the following more general fact: Let $L^p = L^p(\Omega, \mathcal{F}, m)$ be the space of functions $f : \Omega \to \mathbb{R}$ such that $\int_\Omega |f(\omega)|^p m(d\omega)$ for some measure m. If m is finite (i.e. $m(\Omega) < \infty$), then $L^q \subset L^p$.

Informally, for an integral to be ill-defined, one needs either the integrand $f(\omega)$ to explode or the measure m to assign an infinite mass on a region where the function is not close to zero. When m is bounded, only the large values of $f(\omega)$ can be problematic. Therefore, if the values of $|f(\omega)|^p$ are not too large, it is also the case of $|f(\omega)|^q$ since $x^q < x^p$ when $x > 1$.

Since we work with a probability space ($m = P$), we can just apply this fact and obtain the above-mentioned property of the moments of a random variable.

2.8 Samples from an unknown distribution

2.8.1 Law of large numbers and central limit theorem

Whereas Probability Theory addresses general questions related to $(\Omega, \mathcal{F}, P)$, the field of Statistics is concerned with questions where the distribution of a random variable p_X is unknown. Intuitively, the more observations from p_X, the more we should know about it. This intuition is formalised in the *Law of Large Numbers*:

Theorem 1. *Let $(X_i)_{i\geq 1}$ be a sequence of i.i.d. and integrable random variables. For all $m \in \mathbb{N}$, define $S_m := \sum_{i=1}^m X_i$. It holds that*

$$\lim_{m\to\infty} \frac{S_m}{m} = \mathbb{E}[X_1],$$

with probability 1.

When a statement holds with probability 1 under P, we say that the statement holds P-*almost surely.* What the above theorem tells us is that the empirical mean of i.i.d. and integrable random variables converges to the true mean (i.e. the expectation). In other words, if we don't know p_X but have access to an infinite number of independent observations, we can retrieve the expectation of p_X.

In practice, of course, we don't have access to infinitely many observations. The law of large numbers does not tell us at what speed the empirical mean converges. It turns out that by adding the assumption of finite second moment, the *Central Limit Theorem* gives us the order of magnitude of the distance between the empirical mean and the true mean.

Theorem 2. *Let $(X_i)_{i\geq 1}$ be a sequence of i.i.d. square integrable random variables. For all $m \in \mathbb{N}$, define $S_m = \sum_{i=1}^m X_i$. For any real numbers $a < b$, it holds that*

$$\lim_{m\to\infty} P\left[\sqrt{\frac{m}{\mathrm{Var}[X_1]}}\left(\frac{S_m}{m} - \mathbb{E}[X_1]\right) \in (a,b)\right] = P[a < Y < b],$$

where $Y \sim \mathcal{N}(0,1)$.

In the central limit theorem, we have the term $\frac{S_m}{m} - \mathbb{E}[X_1]$, which converges almost surely to 0 as $m \to \infty$ by the law of large numbers. Multiplying this difference by the factor $\sqrt{\frac{m}{\mathrm{Var}[X_1]}}$, we get a random number with normal distribution with mean 0 and variance 1. This factor grows at the speed $\sqrt{m}$ (while $\mathrm{Var}[X_1]$ remains constant in m). This means that in the law of large numbers, the random variable $\frac{S_m}{m}$ converges to $\mathbb{E}[X_1]$ **exactly** at speed $\frac{1}{\sqrt{m}}$. The constant factor simply normalises the Gaussian limit to have variance 1.

2.8.2 Estimating an unknown probability

In this section, we see on a concrete example how the theorems from the previous section can be used to estimate an unknown probability from a finite sample.

The frequentist way

Suppose that we have a fair coin. The fact that we specify the coin is fair implicitly poses the probability P, such that $P(\text{“}heads'') = P(\text{“}tails'') = 1/2$. If instead we are given a coin and we do not know whether it is fair, then we can only say that there exists $p \in [0,1]$ such that $P(\text{“}heads'') = 1 - P(\text{“}tails'') = p$. We can, however, say more about the likely values of p by *estimating* it through repeated experiments, as follows:

We can see the outcome of a coin toss as that of a Bernoulli random variable with parameter p, where the variable is equal to 1 if the coin yields "heads", and 0 if "tails". Suppose that we toss the coin $m \in \mathbb{N}$ times and let X_i be the outcome of the i-th toss. Note that $(X_i)_{i=1}^m$ are i.i.d. Using the law of large numbers, we know that $S_m := \sum_{i=1}^m X_i$ is such that $\frac{S_m}{m} \to p$ as $m \to \infty$. Hence, for a large enough m, we can estimate p by

$$\hat{p} := \frac{\text{\# of heads}}{\text{\# of tosses}} = \frac{S_m}{m}.$$

Naturally, if $m = 3$, it is unlikely that our estimate is satisfactory. Suppose that we want to have an error of order $1/100$. Then we can use the central limit theorem that tells us that the error of the estimate, that is $\hat{p} - p$, is of order $1/\sqrt{m}$ up to a random number that looks like a standard centered Gaussian for large enough m. Hence, to obtain an error of order $1/100$, we need

$$\frac{1}{\sqrt{m}} \leq \frac{1}{100} \;\Leftrightarrow\; m \geq 100^2 = 10000.$$

That is, by tossing ten thousand times the coin, we have that $\hat{p} - p \approx \frac{1}{100} Y$ where $Y \sim \mathcal{N}(0, 1)$.

Take-home message: In this class, "learning" means approximating a function or a distribution given a dataset. From the above example, the law of large numbers and the central limit theorems may be seen as first results on learning. Assuming that the data are indeed i.i.d., the more data, the better.

The Bayesian way

The example of the coin toss above is the *frequentist* paradigm of Statistics. It typically requires a large amount of data to be accurate. The other paradigm is *Bayesian* Statistics, that yields effective estimate with few data, but often requires more computations and an *a priori* distribution. It is based on Bayes' Theorem 3, that we now state:

Theorem 3. *(Bayes' theorem) Let $A, B \in \mathcal{F}$ such that $P(A), P(B) > 0$ The*

Bayes' theorem *states that*

$$P(B|A) = \frac{P(B)P(A|B)}{P(A)} \tag{2.44}$$

The informal interpretation of Bayes' Theorem in the context of learning is the following: suppose B represents your *a priori* beliefs of the world, and A is some observations that are linked to these beliefs. Ideally, you want to update your beliefs according to A. That is exactly what this theorem tells us: the probability of our beliefs B **a posteriori** (that is, after having observed A), is given by the right-hand side of the equation of the statement. Note that it depends on three terms: the prior probability we attribute to our beliefs $P(B)$, the probability of the observations given our prior beliefs $P(A|B)$, and a last term more difficult to interpret $P(A)$. This last term can further be decomposed as

$$P(A) = P(A|B)P(B) + P(A|B^c)P(B^c),$$

so that we can compute $P(A)$ according to whether our beliefs are true or not, and the prior probability we assign to our beliefs.

Let us see how this works with an example.

Example 2.8.1. **Naive Bayes classifier:**

This is a simple "probabilistic classifier" based on applying Bayes' theorem with strong (naive) independence assumptions between the features.

Let us consider again a binary classification. The naive Bayes classifier is a conditional probability model: given an input x to be classified, represented by a vector $x = (x_1, ..., x_m)$ representing m features, it assigns conditional probabilities

$$p(y = +1|x_1, \cdots, x_m), \quad p(y = -1|x_1, \cdots, x_m).$$

Using Bayes' Theorem, we can write

$$p(y = +1|x_1, \cdots, x_m) = \frac{p(y = +1)p(x|y = +1)}{p(x)}.$$

The numerator is equivalent to the joint probability

$$p(y = +1, x_1, \cdots, x_m)$$

which can be rewritten as follows, using the chain rule for repeated applications of the definition of conditional probability:

$$\begin{aligned} p(y=+1, x_1, \cdots, x_m) &= p(x_1|x_2 \cdots, x_m, y=+1)p(x_2, \cdots, x_m, y=+1) \\ &= p(x_1|x_2 \cdots, x_m, y=+1)p(x_2|x_3 \cdots, x_m, y=+1) \\ &\quad p(x_3, \cdots, x_m, y=+1) \\ &= \cdots \\ &= p(x_1|x_2 \cdots, x_m, y=+1)p(x_2|x_3 \cdots, x_m, y=+1) \cdots \\ &\quad p(x_{m-1}|x_m, y=+1)p(x_m|y=+1)p(y=+1). \end{aligned}$$

Suppose now that the so-called **naive conditional independence assumption** holds, which tells us that all features in x are mutually independent, conditional on the label (e.g. $y=+1$ or $y=-1$). Under this assumption:

$$p(x_i|x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_m, y=+1) = p(x_i|y=+1).$$

Then, with this assumption, the original probability can be re-written as:

$$\begin{aligned} p(y=+1|x_1, \cdots, x_m) &\propto p(y=+1, x_1, \cdots, x_m) \\ &\propto p(y=+1)p(x_1|y=+1) \cdots p(x_m|y=+1). \end{aligned}$$

For a new example x, we can compute our best guess as the true label, using:

$$\hat{y} = \arg\max_c p(y=c|x).$$

This is called the MAP estimate (maximum a posteriori).

Example 2.8.2. **Application of naive Bayes classifier:** We now use the naive Bayes classifier to answer the following question: Suppose that there is an infectious disease for which we have a test with false positive probability 0.01 and false negative probability 0.2. This means that a test on a non-infected patient returns a positive result ("infected") with probability 0.01 and a test on an infected patient returns a negative result with probability 0.2. Suppose moreover that we know that 10% of the population is infected.

More formally, let $y \in \{-1, 1\}$ be such that $y=1$ if the patient is infected, -1 if they are non-infected. Suppose that the patient was tested three times (let's ignore the timing of the tests for this exercise) and the results were $x_1 = 1, x_2 = -1, x_3 = 1$, where 1 means that the test was positive and -1 negative.

The naive conditional independence assumption holds in this case, since given y, the results of the tests x_1, x_2, x_3 are independent. We can therefore compute

$$\begin{aligned}
&p(y = +1|x_1 = 1, x_2 = -1, x_3 = 1) \\
&\qquad \propto p(y = +1)p(x_1 = 1|y = +1)p(x_2 = -1|y = +1)p(x_3 = 1|y = +1) \\
&\qquad \propto 0.1 \times (1 - 0.2) \times 0.2 \times (1 - 0.2) = 0.0128.
\end{aligned}$$

Similar computations yield $p(y = -1|x_1 = 1, x_2 = -1, x_3 = 1) \propto 0.0000891$. In particular, we deduce that $p(y = +1|x_1 = 1, x_2 = -1, x_3 = 1) > p(y = -1|x_1 = 1, x_2 = -1, x_3 = 1)$, and the naive Bayes classifier classifies the tested individual as "infected".

2.9 Resources

- Measure, Integral and Probability, Marek Capinski and Ekkehard Kopp

Exercises/things to dwell on:

- We have a fair dice, we want to estimate the probability of getting 6 after throwing it once.
- Suppose we have two dice, we want to estimate the probability of getting at least one 6 after throwing it twice.
- I have two dice and throw them, what is the probability I get the sum of both of them to be larger than 6?
- What is the probability I get 6 on my second throw, after I rolled a 3? And a 6?
- Suppose I have a target and I throw a dart randomly, what is the probability I hit the center of the board?

- Suppose I have this list of words, sampling a word uniformly at random, can you tell me the probability of the word occurrence?
- What happens if I want to compute the probability of a word I have not seen in the list?
- Using Bayes theorem, show that $P(A) > 0$ and $P(B) > 0$, then

$$P(B|A) = \frac{P(A \cap B)}{P(A)} = \frac{P(B)P(A|B)}{P(A)}. \tag{2.45}$$

Chapter 3

Introduction to Machine Learning

Contents

Machine learning aims at building algorithms that autonomously learn how to perform a task from examples. This definition is rather vague on purpose, but to make it slightly clearer: by "autonomously", we mean that no expert is teaching (or coding by hand) the solution; by "learn" we mean that we have a measure of performance of the algorithm's output on the task. In this chapter, we wish to give a general and accessible picture of Machine Learning. We will introduce the notation, set-up and essential concepts of the field, without dwelling on details. The current chapter should provide enough formalism and intuition to make the forthcoming chapters appear natural to the reader.

3.1 Different paradigms

There are three main paradigms in machine learning that sometimes share similar ideas while having very specific techniques. Namely, they are

- *supervised learning* – we have access to labelled examples: spam detection using a dataset of emails, some of which we know are spam, the others we know are not spam;
- *unsupervised learning* – examples are not labelled: the dataset is composed of paintings, the algorithm must group them by guessing which come from the same artist;
- *reinforcement learning* – examples are generated from interacting with the environment: the algorithm controls a drone and learns how to navigate by trial and error.

These different types of learning are not exclusive.

In these notes, we mostly focus on supervised learning; much of what we treat can then be transferred to unsupervised and reinforcement learning. We also concentrate on specific and common techniques of unsupervised learning in Chapter **??**. We hope that the reader is then able to learn autonomously from the literature.

3.2 Supervised learning

In the previous section, we said that supervised learning is learning through **labelled** examples. The problems that can be solved are divided into two types: classification problems when the goal is to predict a *categorical* variable, and regression problems when the prediction takes on *numerical* values that are compared with a distance map.

Example 3.2.1. Recognizing handwritten digits is a classification task: each digit belongs to one of the ten classes from “zero” to “nine”. If a model reads a 2 instead of a 5, it is as wrong as if it has read a 1.

Example 3.2.2. On the other hand, predicting tomorrow's weather (say temperature) is a regression task: if a model predicts a temperature of 24°F tomorrow in Ann Arbor and it happens to be 27°F, although not exact, this prediction is more satisfactory than another of 12°F.

The term *supervised* refers to the fact that the examples used in building the predictor come with labels, that is, learning how to distinguish handwritten digits is done by presenting images <u>and</u> the right answers to the algorithm. This is in contrast to *unsupervised learning* where no labels are provided and the main goal is to find *structure* in the data (for example, possible clusters, lower dimensional representations, etc...)

3.2.1 Set-up

The supervised learning set-up is as follows:

- An input space $\mathcal{X} \subset \mathbb{R}^{n_{\text{in}}}$ with $n_{\text{in}} \geq 1$ and an output space $\mathcal{Y} \subset \mathbb{R}^{n_{\text{out}}}$ with $n_{\text{out}} \geq 1$,
- An unknown mapping $f : \mathcal{X} \to \mathcal{Y}$ we want to approximate,
- A probability distribution D on $\mathcal{X}$,
- A dataset $S = \{(x_i, y_i); i = 1..m\}$ such that $f(x_i) = y_i$ for all $i = 1..m$ and the x_i's are independent and identically distributed (i.i.d.) with common law D,
- A *hypothesis class* $\mathcal{H}$, that is a set of functions mapping $\mathcal{X}$ to $\mathcal{Y}$, supposedly containing the unknown f (or good approximations of it),
- A loss function $\ell : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}_+$ such that for $h \in \mathcal{H}$, $\ell(h(x_i), y_i)$ measures the error of the prediction of h at x_i from its true label $y_i = f(x_i)$. It is thus standard to assume that $\ell(y, y') = \ell(y', y)$ and $\ell(y, y) = 0$ for all $y, y' \in \mathcal{Y}$.
- A training algorithm $\mathcal{A}$, as defined in the forthcoming definition 3.2.4.

Remark 5. The dataset above supposes that the observations are perfect. Often in practice, this is not the case (e.g. temperature measurements). Such a dataset is said to be *noisy* and this noise is included in the model such that $y_i = f(x_i) + \epsilon_i$ where $(\epsilon_1, \cdots, \epsilon_n)$ is a random vector, often (but not always) assumed to be Gaussian with mean 0 and independent coordinates. More on that in the next chapter.

Remark 6. Choosing a parametric model corresponds to choosing a specific hypothesis class $\mathcal{H}$ – hence we will interchangeably use *model* and *hypothesis class*. For instance, if $\mathcal{X} \subset \mathbb{R}$, one could choose $\mathcal{H} = \{h : x \mapsto w_1 \cos(x) + w_2 \sin(x); w_1, w_2 \in \mathbb{R}\}$. The numbers w_1, w_2 are called the *parameters* of the model. Throughout these notes, we will aggregate them in a parameters vector $w \in \mathbb{R}^P$ where P will usually denote the number of parameters of the model (here $P = 2$). Henceforth, we call an element of $\mathcal{H}$ a *predictor* and we write f_w instead of h for a function in $\mathcal{H}$ when we want to make the dependency of the predictor on the parameters explicit.

Remark 7. Choosing a hypothesis class $\mathcal{H}$ is often an engineering choice, we might choose $\mathcal{H}$ according to simplicity, expressiveness, prior knowledge of our problem, etc. In this class, we will see for example; perceptrons, support vector machines, kernel methods, ensemble methods, neural networks, etc...)

Then, we can define more concretely classification and regression problems.

Definition 3.2.1. A problem is said to be a classification problem if the labels are categorical, or more formally, if $\mathcal{Y}$ is discrete and $\ell : (y, y') = 1_{y \neq y'}$. If there are $k \in \mathbb{N}$ classes, we usually encode them such that $\mathcal{Y} = 1, \cdots, k$.

Definition 3.2.2. A regression task is a learning task where $\mathcal{Y}$ takes numerical values, i.e. $\mathcal{Y} \in \mathbb{R}^{n_{\text{out}}}$ and such that predictions are evaluated by a loss function $\ell : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}_+$ that does more than simply discriminated whether the prediction is right or wrong, but also provides a magnitude of error. This is the key difference between classification and regression.

Definition 3.2.3. For a predictor $h \in \mathcal{H}$, the *generalisation error* of h is defined by

$$R_D(h) = \mathbb{E}_{x \sim D}[\ell(h(x), f(x))]. \tag{3.1}$$

The goal of supervised learning is to solve the optimization problem

$$\min_{h \in \mathcal{H}} R_D(h). \tag{3.2}$$

In general, the generalisation error of a predictor is not directly accessible. We instead can minimise the following quantity:

$$\min_{h \in \mathcal{H}} L(h) = \min_{h \in \mathcal{H}} \frac{1}{m} \sum_{i=1}^{m} \ell(h(x_i), y_i). \tag{3.3}$$

where the mapping $L : \mathcal{H} \to \mathbb{R}_+$ is called the *empirical error* (or *training error*, *empirical/training loss*), as it depends on the dataset. What we described above is the **empirical risk minimization** (ERM) framework, and it assumes that a predictor which minimises (3.3), that we call h_*, is close to minimising (3.1). Even though we may not manage to find h_*, the hope is that a predictor h that performs decently well on the dataset is good at predicting labels for inputs outside of the dataset. Informally and more concisely, we expect $L(h)$ small $\Rightarrow R_D(h)$ small. We will see, however, in the forthcoming section 3.3 that for some h, $L(h)$ can be small (even zero) but $R_D(h)$ very large; this is called *overfitting*. Avoiding overfitting is both a practical and theoretical topic of interest.

3.2.2 Parameters and training

In remark 6, we explained that a parametric model defines a hypothesis class. That is, denoting by $\mathcal{W} \subset \mathbb{R}^P$ the parameters space of the model, we have $\mathcal{H} = \{f_w; w \in \mathcal{W}\}$. The parameters w can be trained to search over predictors in $\mathcal{H}$, hence they are called *trainable parameters*. The training procedure simply refers to the following:

Definition 3.2.4 (informal)**.** Given a dataset S and a hypothesis class $\mathcal{H} = \{f_w; w \in \mathcal{W}\}$ with $\mathcal{W} \subset \mathbb{R}^P$ the parameter space, we say that a map $\mathcal{A} = \mathcal{A}_{\mathcal{H},S} : \mathcal{W} \to \mathcal{W}$ is a *training algorithm*.

The idea behind this definition is that the purpose of a training algorithm $\mathcal{A}$ is to send initial parameters $w_0 \in \mathcal{W}$ to trained parameters $\mathcal{A}(w_0) \in \mathcal{W}$ using the dataset S such that $f_{\mathcal{A}(w_0)}$ performs well at minimizing (3.3). In this context, the scheme used to choose w_0 gives an initial predictor f_{w_0} that needs not perform well. One can choose w_0 deterministically or randomly according to this scheme, that we call *the initialization* of the parameters. The training algorithm can itself use randomness.

On the other hand, a parametric model can have non-trainable parameters

Definition 3.2.5. All parameters of $\mathcal{H}$ and $\mathcal{A}$ that are not modified by $\mathcal{A}$ are called *hyperparameters.*

Changing the hyperparameters corresponds to changing the hypothesis class or the algorithm $\mathcal{A}$.

Example 3.2.3. Let $\mathcal{X} = \mathbb{R}$ and $\mathcal{Y} = \mathbb{R}$. Let $\mathcal{H}_k$ be the set of polynomials of degree at most $k \in \mathbb{N}$. Given a dataset $S = \{(x_i, y_i); i = 1..m\}$, one can try to learn the task in $\mathcal{H}_1$ if one believes that the relationship between inputs and outputs is linear, i.e. there exists $a, b \in \mathbb{R}$ such that $y_i = ax_i + b$ and more generally, $f(x) = ax + b$ for all $x \in \mathbb{R}$. A training algorithm $\mathcal{A} : \mathbb{R}^2 \to \mathbb{R}^2$ that finds the best (a, b) does not modify the value of k (equal to 1 here), hence it is a hyperparameter.

3.3 Model selection

In this section, we present a method that addresses the following question:

Given a learning task, how do we choose a good model ?

Indeed, choosing a simple model, i.e. a parametric hypothesis class with few parameters, may result in a predictor that is unable to fit the data, whereas a complex model with too many parameters will fit the data but may not be able to predict reasonable values outside of the dataset. These issues are called *underfitting* and *overfitting* and are (slightly) more formally defined below. We will see how splitting the dataset into a training set and a test set helps avoiding it.

Furthermore, having finitely many data samples prevents us from trying out all possible models we have at hand, as it would cause a problem of overfitting **the test set**. The *cross-validation* technique fixes this problem by smartly making use of the data samples, as we will describe.

3.3.1 Underfitting and overfitting

Recall that in the ERM framework, in hope to minimise (3.1), we seek to minimise (3.3).

Definition 3.3.1. Suppose we try to solve a task with the ERM framework and let $h \in \mathcal{H}$ be a predictor. The difference between the generalisation loss of h and its empirical loss is called the *generalisation gap* of h, that is

$$R_D(h) - L(h) = \mathbb{E}_{x \sim D}\left[\ell(h(x), f(x))\right] - \frac{1}{m}\sum_{i=1}^{m} \ell(h(x_i), y_i).$$

The generalisation gap is in general inaccessible to the practitioner, as D and f are unknown. However, there are ways one can estimate it, by splitting the dataset into two disjoint training set and test set. Indeed, let S_{train} and S_{test} be two disjoint subsets of S. To ease the notation, let us assume that $S_{\text{train}} = \{(x_i, y_i) : i \in \{1, \ldots, m'\}\}$ and $S_{\text{test}} = \{(x_i, y_i) : i \in \{m'+1, \ldots, m\}\}$ for some $m' < m$. Then, it suffices to train a model on S_{train} to fit it, and to estimate the generalisation error $\mathbb{E}_{x \sim D}\left[\ell(h(x), f(x))\right]$ by

$$\frac{1}{m - m'}\sum_{i=m'+1}^{m} \ell(h(x_i), y_i).$$

Note that if $m - m'$ tends to infinity, this becomes the exact generalisation error by the law of large numbers (Theorem 1) since the data samples are assumed i.i.d.

Definition 3.3.2. (Underfitting and overfitting)

- We say that *underfitting* occurs when a hypothesis class is too simple to fit the data properly, that is when $\inf_{h \in \mathcal{H}} L(h)$ is large.

- We say that *overfitting* occurs when a predictor h fits the data well but is too complex to generalise outside the dataset, that is when $L(h) \approx 0$ but $R_D(h)$ is large.

In general, the training error decreases as we increase the complexity or flexibility of our model (e.g., in polynomial fitting, as we use higher degrees of polynomial functions). The generalisation error tends to also decrease initially as complexity increases, but then increases as the model overfits the training set. [1]

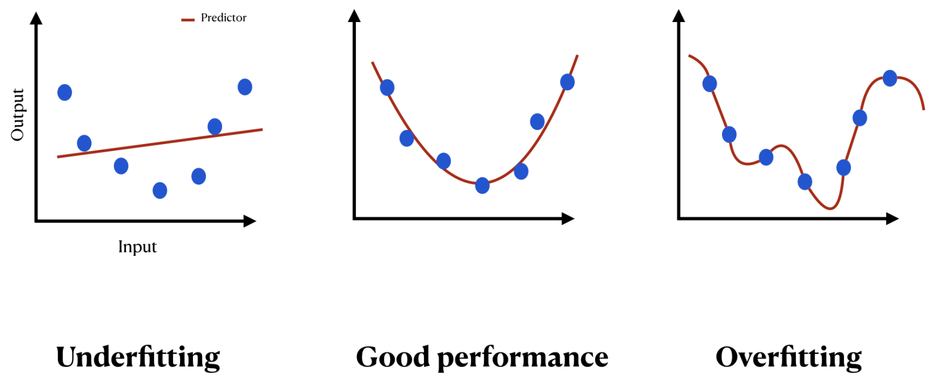


Figure 3.1: Example of *underfitting*, optimal fitting and *overfitting*.

⋆ **Remark 7.** In Figure 3.1, overfitting occurs because the predictor is too complex and fits the noise in the data. Even without noise, an overly complex model can feature overfitting, e.g.: for a dataset of n points on a line, a polynomial of degree $n + k$ can perfectly fit all the datapoints and look very different from a line.

[1] This is not the complete story, as you will see further on.

3.3.2 Validation set and cross-validation

On a given task, to assess the efficacy of a model $\mathcal{H}$, after having split the dataset into a training set and a test set, we can simply look at the test error, as this is an estimate for the generalisation loss.

Thanks to this split, we have a way to detect overfitting. For example, if the training error is very low, but the test error is very high. If our model overfits, we can simply change to another model. However, that is a naive way of choosing a model that could lead to **overfitting the test set**.

Suppose that for a given task, you have the choice between several models $\mathcal{H}^{(1)}, \dots, \mathcal{H}^{(n)}$, and that you have no a priori reason to favour one over the others. How do we select the best model? Suppose that we train all of them on the training set, and compare the predictors thus obtained on the test set. We then select the predictor that had the lower test error.

By proceeding that way, we expose ourselves to selecting a predictor which, **by chance**, overfits S_{test}. Note that the predictor does not have to be trained on S_{test} to overfit it.

Indeed, suppose that the number of models $n \to \infty$, then one can convince oneself that it becomes more and more likely that one of the predictors is such that $h(x_i) \approx y_i$ for all $(x_i, y_i) \in S_{\text{test}}$. This means that we cannot assess whether the chosen hypothesis class is well chosen.

Validation set

One way to deal with that issue is to split the dataset into three disjoint subsets:

- the training set S_{train},
- the validation set S_{val},
- the test set S_{test}.

Now the procedure becomes the following: train the n models on S_{train}, compare their performances on S_{val}, select the best model and retrain it on $S_{\text{train}} \cup S_{\text{val}}$, then assess its performance on S_{test}.

One analogy is to view the training set as textbook lectures and examples from which we learn a new concept, and we encountered some confusing topics which we have multiple possible interpretations; validation set entails practice problems and previous years' exams, to help us choose which interpretation is the best; and the test set is the final exam of the course.

Cross-validation

Although this procedure seems satisfactory, it can greatly be improved. Indeed, note that for each class $\mathcal{H}^{(i)}$, we select **only one** predictor $h^{(i)}$ to assess how good of a choice $\mathcal{H}^{(i)}$ is. This in turns make the randomness of the finite sampling and split of the dataset too important in the model selection; ideally we want to choose the best model for a distribution D and a labelling function f, not the best model for a given dataset split $S_{\text{train}}, S_{\text{val}}, S_{\text{set}}$.

In order to solve this issue, one can use *cross-validation*. Let $\mathcal{H}$ be fixed and let us use cross-validation to assess how good of a choice it is for a given task. Let split the dataset $S = \{(x_i, y_i) : i \in \{1, \ldots, m\}\}$ into a training set S_{train} and test set S_{test} as before. We now randomly partition the dataset $S_{train} = \{(x_i, y_i) : i \in \{1, \ldots, m'\}\}$ into k disjoint and covering subsets $S_1, \ldots, S_k$ of roughly the same size. For $i = 1$ to k, we call S_i the i-th fold. We denote by m_i the size of the i-th fold such that $\sum_{i=1}^{k} m_i = m'$. We now proceed as follows: for all $i \in \{1, \ldots, k\}$, train the model on

$$\bigcup_{\substack{j=1 \\ j \neq i}}^{k} S_j,$$

and denote by $h_i \in \mathcal{H}$ the predictor thus obtained. We evaluate the performance of h_i on the i-th fold, that are the only samples from the training dataset that the predictor has not trained on, that is we define

$$L_i(h_i) := \frac{1}{m_i} \sum_{(x,y) \in S_i} \ell(h_i(x), y).$$

We now have a collection of k predictors belonging to $\mathcal{H}$, each trained on a different subset of the training dataset, and for each we have an estimate of their generalisation error computed on the corresponding i-th fold they have not seen. We now evaluate the quality of the model $\mathcal{H}$ for the task by

$$\mathrm{CV}(\mathcal{H}) := \frac{1}{k}\sum_{i=1}^{k} L_i(h_i).$$

Coming back to our initial question of choosing the best model among $\mathcal{H}^{(1)}, \ldots, \mathcal{H}^{(n)}$, we can simply choose the one that minimises the cross-validation error, that is the smallest $\mathrm{CV}(\mathcal{H}^{(i)})$. Then, we can retrain that model on the whole training dataset S_{train}, and estimate its generalisation error on S_{test}.

Choosing k

Recall that k is the number of folds used during cross-validation (CV). What value of k should we choose?

We can ask ourselves, what is:

- the influence of k when estimating the expected generalisation error?
- the influence of k on the size of the training set and therefore, attained approximators $h_1, ..., h_k$?
- the computational complexity of the training algorithm for different k?

Consider a dataset of fixed size m. On the one extreme, if we choose $k = m'$, then the k-fold CV becomes the *leave-one-out cross validation (LOO-CV).* With respect to the estimator of the generalisation error, most experiments show that there's a monotonically decreasing or constant variance of the generalisation error estimator with increasing k[2]. We also note that this creates the largest number of predictors $h_1, \ldots, h_k$, which might be very similar to one another, with training sets of size $(m'' = m' - 1)$ (after leaving

[2]with some exceptions on the stability of the algorithm

one out). Furthermore, it can be quite computationally expensive since it requires solving m' slightly smaller $(m'-1)$-sized subproblems of the same type.

On the other extreme, a small k (such as $k = 2$) provides an estimator for the generalisation error with a higher variance and higher bias. This is due to the fact that the estimators are trained on distinct and smaller datasets $(m'' = m'/2)$. However, the computational complexity of the training algorithm is small as well (as we only train h_1 and h_2).

In practice, the number of folds k will depend on the dataset size, as one must balance the training dataset size $m'' = (k-1)m'/k$ and the computational complexity of training k models. For example, Figure 3.2 shows a hypothetical learning curve for a classifier, plotting $1 - R_D(h)$, the true prediction error for a given estimator h using a 5-fold CV. In this case, if we have a dataset of 200 points, then a 5-fold CV would generate training sets of size $m'' = 160$, which would reach an error that is fairly close to if the full $m' = 200$ were used and the trained predictors would not suffer from much more bias. On the other hand, if we had a dataset set of $m' = 50$ points, then using a 5-fold CV would lead to training sets of size $m'' = 40$, leading to a substantially increased error of the predictor. The precise trade-off thus is problem dependent and dataset size dependent.

As a compromise among bias, variance and computational cost, k of 5 or 10 are commonly used choices for many applied problems.

3.4 No free lunch theorem

The no free lunch theorem is often talked about informally, one of the reasons being that many similar – but not equivalent – versions of the theorem exist in the literature. The theorem is often stated as follows:

“All optimization algorithms perform equally well when their performance is averaged across all possible problems.”

The term “averaged” here does not have a formal meaning, but using the

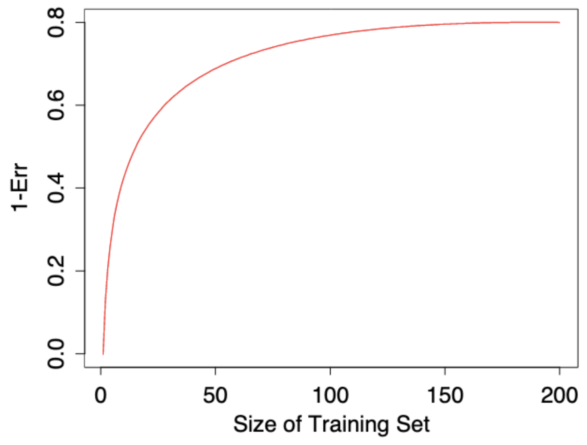


Figure 3.2: A hypothetical learning curve for a classifier on a given task: a plot of 1-Err versus the size of the training set, considering a 5-fold CV. [?]

Machine Learning formalism we introduced in this chapter, the no free lunch theorem can be understood as follows: without any prior knowledge on the data distribution D and the labelling function f, there is no way to guess whether a pair of hypothesis class and training algorithm $(\mathcal{H}_1, \mathcal{A}_1)$ is likely to perform better or worse than another pair $(\mathcal{H}_2, \mathcal{A}_2)$.

It means that there is no single best machine learning algorithm across all possible prediction problems. This, in turn, motivates the development of many different types of models, to cover the wide variety of data that appears in the real world.

3.5 Optimisation

Optimisation is the field of Mathematics that is concerned with finding inputs that optimise – i.e. minimise or maximise – a given function. This is precisely what we aim for in the ERM framework! In this section, we present the basic algorithm for training a parametric model.

3.5.1 Gradient descent

In the set-up of supervised learning, we informally defined the notion of training algorithms in Definition 3.2.4. Many training algorithms that are used in practice are inspired (sometimes mere variants) of the simple *gradient descent algorithm* used to solve minimization problems.

Let $C : \mathbb{R}^P \to \mathbb{R}$ be a convex, differentiable map. Recall that this means that

$$C(v) \geq C(u) + \nabla C(u) \cdot (v - u),$$

where $\cdot$ denotes the dot product in $\mathbb{R}^P$. Because it is convex, every critical point of C is a global minimum, i.e. $\nabla C(u) = 0$ if and only if $u = \arg\min_{w \in \mathbb{R}^P} C(w)$.

The gradient descent algorithm

The gradient descent algorithm is a first-order iterative optimisation algorithm for finding a local minimum of a differentiable function.

Recall that $S = \{(x_i, y_i); i = 1..m\}$ is our dataset and $\mathcal{H}$ is our hypothesis class.

Definition 3.5.1. Let $w_0 \in \mathbb{R}^P$ and fix $\eta > 0$. We then define w_k, $k \geq 0$ recursively as

$$w_{k+1} = w_k - \eta \nabla C(w_k).$$

One step of the above recursion is called a *gradient descent step* or *gradient descent update.*

Given a number of steps $K \in \mathbb{N}$, gradient descent is the following training algorithm:

$$\mathcal{A}_{\text{GD}} : w_0 \mapsto w_K,$$

for all $w_0 \in \mathbb{R}^P$.

Remark 8. The positive real number η is called the *learning rate*, as it governs the size of the updates (see influence of stepsize in Figure 3.3). Both the learning rate η and the number of steps K are left unchanged when applying $\mathcal{A}_{\text{GD}}$ to w_0. Hence, they are hyperparameters according to Definition 3.2.5.

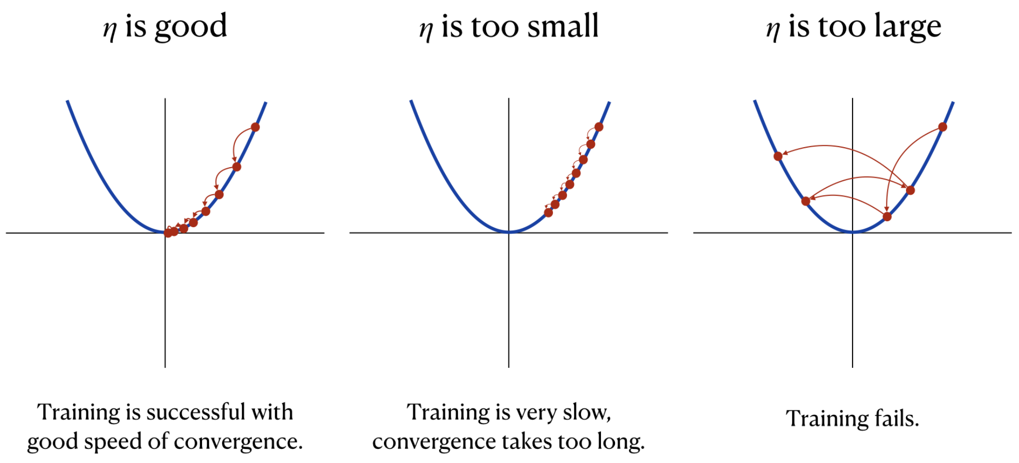

η is good
η is too small
η is too large
Training is successful with good speed of convergence.
Training is very slow, convergence takes too long.
Training fails.

Figure 3.3: Learning rate η

Remark 9. In the ERM framework, the gradient descent update equation reads as

$$w_{k+1} = w_k - \eta \nabla_w L(f_{w_k}) = w_k - \eta \sum_{i=1}^{m} \nabla_w \ell\left(f_{w_k}(x_i), y_i\right).$$

3.5.2 Gradient flow*

Closely related to gradient descent is the *gradient flow*, which can be seen as a continuous version of gradient descent when the learning rate goes to 0, as we shall see. The interest of gradient flow is purely theoretical: it is often easier to **prove** theorems by assuming that training is done under gradient flow, and then argue that these theorems should hold true under gradient

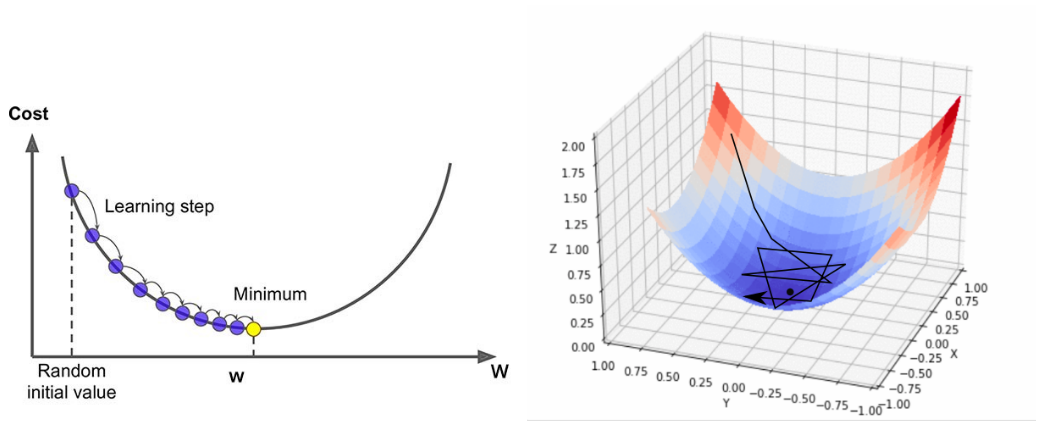


Figure 3.4: Example of Gradient Descent in 1-dimensional problem (left) and 2-dimensional problem (right)

descent for small enough learning rates (though rigorous results can also be proven under gradient descent directly).

Definition 3.5.2. We say that a family $(u(t))_{t\in\mathbb{R}_+} \subset \mathbb{R}^P$ follows the (negative of the) gradient flow of C if it is solution of the following differential equation:

$$\partial_t u(t) = -\nabla_u C(u(t)).$$

Because $u(t)$ follows the negative of the gradient of C, one can show that $t \mapsto C(u(t))$ is decreasing and reaches its minimal value as $t \to \infty$. Assuming that $u(t) \to u_* \in \mathbb{R}^P$ as $t \to \infty$, one sees that $\nabla C(u_*) = 0$ and the convexity of C ensures that $C(u_*)$ is a global minimum.

To see that gradient descent is a discrete approximation of gradient flow, let $k \in \mathbb{N}$ and use Definition 3.5.1 to write

$$w_k = -\eta \sum_{\ell=0}^{k} \nabla C(w_\ell).$$

By choosing $k = \lfloor t/\eta \rfloor$, where $\lfloor\cdot\rfloor$ denotes the integer part, we can take the limit as $\eta \to 0+$ and (assuming that ∇C is continuous to define the Riemann integral) there exists $w : \mathbb{R}_+ \to \mathbb{R}^P$ such that

$$\lim_{\eta \to 0+} w_{t/\eta} = \lim_{\eta \to 0+} -\eta \sum_{\ell=0}^{\lfloor t/\eta \rfloor - 1} \nabla C(w_\ell) = \int_0^t -\nabla C(w(s))\mathrm{d}s,$$

and therefore rescaled time gradient descent converges to gradient flow.

3.6 ML pipeline in practice

For most machine learning problems (supervised), the pipeline will be similar:

- Given a dataset, split it into: training, validation and test sets.
- Choose a hypothesis class (or several) $\mathcal{H}$.
- Define a metric of error ℓ.
- Use cross-validation to find the best suited hypothesis class $\mathcal{H}$ in a set of possible hypothesis classes
- Once the best suited hypothesis class $\mathcal{H}$ has been chosen, train a prediction in $\mathcal{H}$ and valuate the performance on test set to judge the generalisation error.

3.7 List of tasks

As presented in this chapter, the machine learning formalism abstractly seems to be suited for a wide variety of tasks as long as they can be phrased as a function approximation problem. Below you will find a list of tasks to illustrate how to use this formalism. For each of them, we encourage the reader to answer the following questions:

Q1. Into what paradigm does the task fall? (Supervised/unsupervised)

Q2. If supervised, is it a regression or a classification task?

Throughout the notes, we will learn about many different models (or hypothesis classes). In order to get familiar with them and understand what makes them different, we encourage the reader to think about the following question for every encounter with a new $\mathcal{H}$:

Q3. Is $\mathcal{H}$ well suited for this task? (For all the tasks below.)

For the tasks where it is provided, f denotes the labelling function (generating the dataset) and cannot be used to trained the algorithm, as it is in general not known. We provide it to the reader so they may evaluate the performances of their models by computing the exact generalisation error and generalisation gap.

Task 1: Let $\mathcal{X} = \mathbb{R}^2$ and $\mathcal{Y} = 0, 1$. Suppose $f : x \mapsto \mathbb{1}_{\{x_1 < x_2\}}$. You are given the dataset $S = \{(-1, 0, 1), (-0.5, -1, 0), (0, 0.5, 1), (0, 1, 1), (1, 0.5, 0), (2, 0, 0)\}$.

Task 2: Let $\mathcal{X} = \mathbb{R}^2$ and $\mathcal{Y} = 0, 1$. Let f be 0 inside the closed unit disk, and 1 outside of it.

Task 3: Let $\mathcal{X} = \mathbb{R}$ and $\mathcal{Y} = \mathbb{R}$. Take some points in $\mathbb{R}$ and label them by $f(x) = -x + 3$.

Task 4: Let $\mathcal{X} = \mathbb{R}$ and $\mathcal{Y} = \mathbb{R}$. Take some points in $\mathbb{R}$ and label them by $f(x) = x^2 - 1$.

Task 5: You find a text handwritten on a tablet in an unknown alphabet. Before trying to decipher the text, you want to group the identical characters together.

Task 6: You have access to all American literature and its translation in French. You want your algorithm to learn how to translate a text from American English to French.

Task 7: You know the rules of chess, i.e. how to legally move the pieces on a chessboard and how to asses the result of a chess game. Besides that, you have no knowledge of what is a good or bad move. You want to build a chess engine that surpasses top human level.

Task 8: You are given a dataset of pictures of cats and you want to generate new realistic images of cats.

Task 9: You work for Nitflex, a streaming service company, and want to give good recommendations of movies to new subscribers, based on the data of the older subscribers (movies they liked, categories, age, etc)

Chapter 4

Statistical learning theory

Contents

In this chapter, we take the point of view of Statistics to study learning theory. In short, Statistics is concerned with estimating an unknown distribution μ from observations of it. Typical questions can be: how many data points should one collect to ensure that one's estimator $\hat{\mu}$ is close enough (in some sense) to μ? How complex should a model be to have a small empirical error and a small generalisation error? More generally, we will be interested in deriving approximation bounds of hypothesis classes.

$\star$ **Remark 8.** If you are familiar with numerical analysis, you can think of this chapter as techniques to come up with *a priori error estimates* (i.e. before we sample the dataset), whereas what was shown in the previous chapter, we were computing *a posteriori error estimates* (we measure the error for a specific model and a specific dataset).

4.1 Learnability

In this chapter, except if explicitly stated, we restrict ourselves to the case of binary classification, that is $\mathcal{Y} = \{0, 1\}$, as it simplifies the presentation. As usual, we assume that the dataset $S = \{(x_i, y_i) : i \in \{1, \dots, m\}\}$ is such that the x_i's are i.i.d. with common law D in $\mathcal{X}$ and that $y_i = f(x_i)$ for some labeling function f. Furthermore, we assume the 0-1 loss function $1_{h(x) \neq y}$.

Since for a trained predictor h, the empirical error $L(h)$ depends on the training set S, which is generated through random samplings under D, there will be randomness on the trained predictor h, and therefore, in $R_D(h)$. Thus, we can see $R_D(h)$ as a random variable. We can't expect that S will suffice to direct the learner toward a good classifier (w.r.t. to all of D), in case S is not representative of D.

Example 4.1.1. Suppose we have a urn with 30% black and 70% white balls, and we take a sample where we get "W W W W W". In this case, our sample does not represent the underlying distribution of the balls. (Note that the probability of sampling this dataset is $0.7^5 \approx 0.16$, which is far from negligible.)

From the law of large numbers (Theorem 1), we know that more data samples will ensure that the dataset is representative enough and avoid situations like in this example. But the finiteness of the dataset can hinder learning. In the previous chapter, we saw ways to assess the quality of a model, and how to select a good model among a collection of models to solve a given task. In this chapter, we are instead concerned with studying the *learnability* of a given hypothesis class $\mathcal{H}$ from a finite dataset.

4.1.1 Realisability assumption

Definition 4.1.1. (Realisability assumption) For given hypothesis class $\mathcal{H}$, data distribution D and labeling function f, we say that the *realisability assumption* holds for $\mathcal{H}, D, f$ if $R_{D,f}(h) = 0$.

Informally, this can also be seen as the labeling function f can be represented by elements of $\mathcal{H}$, at least on the support of D. Alternatively, if $f \in \mathcal{H}$, then clearly the realisability assumption holds true.

Example 4.1.2. It is easy to construct a case where the realisability assumption does not hold: let $\mathcal{X} = \mathbb{R}$, $D = \text{Unif}(-1, 1)$, $f : x \mapsto \mathbb{1}_{\{|x|>1/2\}}$ and $\mathcal{H} = \{h_w : x \mapsto \mathbb{1}_{\{x>w\}}\}$, where $w \in \mathbb{R}$ denotes the only parameter of the model. One can check that the realisability assumption does not hold. (Note that m being fixed, there is a positive probability on the dataset S that there exists $h \in \mathcal{H}$ such that $L(h) = 0$, but this probability is not 1.)

4.1.2 PAC learnability

The realisability assumption ensures that there is a predictor in the hypothesis class that fits the data perfectly. This guarantee, however, does not mean that we are able to find this predictor. We need stronger properties for the hypothesis class. This is why the *Probably Approximately Correct learning* (PAC learning) framework was introduced.

Definition 4.1.2. (PAC learnability) A hypothesis class $\mathcal{H}$ is PAC learnable if there exists a function $m_{\mathcal{H}} : (0, 1)^2 \to \mathbb{N}$ and a learning algorithm $\mathcal{A}$ with the following properly: for every $\epsilon, \delta \in (0, 1)$, for every distribution $\mathcal{D}$ over $\mathcal{X}$, and for every labeling function $f : \mathcal{X} \to \{0, 1\}$, if the realisability assumption holds with respect to $\mathcal{H}, D, f$, then when running $\mathcal{A}$ on $m \geq m_{\mathcal{H}}(\epsilon, \delta)$ i.i.d. examples generated by D labeled by f, $\mathcal{A}$ returns a hypothesis $h_{\mathcal{A}}$ such that, with probability of at least $1 - \delta$, $R_D(h_{\mathcal{A}}) \leq \epsilon$.

We have two parameters in the PAC learnability. The accuracy parameter ϵ determines how far we allow our predictor h to be from the optimal predictor ("approximately correct"), and a confidence parameter δ that indicates how likely h is to meet the accuracy requirement ("probably").

Remark 10. The definition above does not imply anything on the computational aspects of learnability. Nonetheless:

(i) some definitions require $m(\epsilon, \delta)$ to grow polynomially with its parameters as they tend to 0, which for example, does not allow $m(\epsilon, \delta) = \epsilon^{-1}2^{1/\delta}$;

(ii) some definitions include the number of computations of $\mathcal{A}$ as a parameter of m (and ask for polynomial growth).

Remark 11. The PAC learning definition is not easy to digest at first. After some time, one may even wonder if the definition is not vacuous. Indeed, since m can grow at any pace in terms of its parameters and there is no restriction either on $\mathcal{A}$, one can set $m(\epsilon, \delta)$ extremely large (think of $10^{(\epsilon\delta)^{-1000000}}$). By the law of large numbers and the central limit theorem (Theorems 1 and 2), we know that $L(h) \to R_D(h)$ as $m \to \infty$ with an approximation error of order $O(1/\sqrt{m})$, and since the realisability assumption holds, it feels like we should always be likely to find a good enough predictor. However this intuition does not take into account the following fact: $m(\epsilon, \delta)$ and $\mathcal{H}$ are fixed **before** choosing f and D. In Corollary 6.4 and Theorem 5.1 in [12], the interested reader will find a construction of a non-PAC learnable hypothesis class. We **informally** explain the main idea on a particular case:

Let $\mathcal{H}$ be the set of all functions from $\mathbb{R}$ to $\{0, 1\}$; I claim that it is not PAC learnable and to demonstrate it, as you choose $m(\epsilon, \delta)$, I will adversarially construct D and f. After you fix $m(\epsilon, \delta)$, I choose arbitrary $2m(\epsilon, \delta)$ pairwise distinct points in $\mathbb{R}$. I let D be the uniform discrete distribution on these $2m(\epsilon, \delta)$ points. Because $\mathcal{H}$ contains all $\{0, 1\}$-binary functions, I can label them in any possible way with an $f \in \mathcal{H}$ so that the realisability assumption holds. Because (at least) half of the $2m(\epsilon, \delta)$ points will not be in the dataset S (recall that it is sampled with D), whatever your algorithm $\mathcal{A}$ is, even if it finds a predictor h with $L(h) = 0$, it will not have learned anything on half of the points, and therefore will be likely to not be ϵ-close to f. Conclusion: $\mathcal{H}$ is not PAC learnable.

4.1.3 Agnostic PAC learnability

We saw that PAC learning offers guarantees on a hypothesis class that allows to retrieve a labeling function **within that class**, since Definition 4.1.2 assumes the realisability assumption. However, the datasets a practitioner meets are not, in general, labeled by a function that belongs to the hypothesis class they use. Even in the case of a truly linear relationship between x and y, it is enough, for example, to have noise in the samples so that the realisability assumption does not hold for the set of linear predictors. This means that we can't guarantee that the generalisation error $R_D(h) = 0$; we still want to find $h \in \mathcal{H}$ for which the generalisation error is nevertheless low.

It turns out that the realisability assumption can be removed and the PAC learning formalism can be extended:

Definition 4.1.3. (Agnostic PAC learnability) A hypothesis class $\mathcal{H}$ is agnostic PAC learnable if there exists a function $m_{\mathcal{H}} : (0,1)^2 \to \mathbb{N}$ and a learning algorithm $\mathcal{A}$ with the following property: for every ϵ,$\delta \in (0,1)$ and for every distribution $\mathcal{D}$ over $\mathcal{X}$ and labeling function $f : \mathcal{X} \to \{0,1\}$, when running the learning algorithm on $m \geq m_{\mathcal{H}}(\epsilon, \delta)$ i.i.d. samples, the algorithm returns a hypothesis $h_{\mathcal{A}}$ such that, with probability at least $1 - \delta$,

$$R_D(h_{\mathcal{A}}) \leq \min_{h^* \in \mathcal{H}} R_D(h^*) + \epsilon.$$

Remark 12. When the realisability assumption does not hold, no learner can guarantee an arbitrarily small error ϵ. Under the definition of agnostic PAC learning, a learner can still declare success if its error is not much larger than the best error achievable by a predictor from the class $\mathcal{H}$. This is in contrast to PAC learning, in which the learner is required to achieve a small error in absolute terms and not relative to the best error achievable by the hypothesis class. In particular, for a given task, a hypothesis class $\mathcal{H}$ could be a very poor choice, and still be agnostic PAC learnable. More informally and concisely: agnostic PAC learnable does **not** imply good model choice.

4.2 Finite-sized hypothesis classes

The PAC learning formalism tells us that if $\mathcal{H}$ is too complex, we may not be able to find good predictors in $\mathcal{H}$ from finitely many data samples. The first restriction one may want to look at is when $\mathcal{H}$ is finite, that is, $\mathcal{H}$ contains only a finite number of functions. Is $\mathcal{H}$ simple enough to be PAC learnable? Throughout this section, we work under the assumption that $\mathcal{H}$ is finite.

4.2.1 PAC learnability

It turns out that any finite hypothesis class is PAC learnable:

Theorem 4. *If $\mathcal{H}$ is finite, then it is PAC learnable. Moreover, the map $m : (0,1)^2 \to \mathbb{N}$ in Definition 4.1.2 can be chosen as*

$$m(\epsilon, \delta) = \left\lceil \frac{\log(|\mathcal{H}|/\delta)}{\epsilon} \right\rceil,$$

where $\lceil x \rceil$ denotes the smallest integer greater or equal to $x \in \mathbb{R}$.

Proof. Let $h_{\mathcal{A}}$ denote the resulting predictor after training. Note that $h_{\mathcal{A}}$ depends on the dataset S; if we sample m i.i.d. data points according to D, we look at S as a random variable with law D^m.

Fix $\epsilon, \delta \in (0,1)$ and let the map $m(\cdot,\cdot)$ be defined as in the Theorem. Proving that $\mathcal{H}$ is PAC learnable amounts to proving that

$$D^{m(\epsilon,\delta)}(S : R_D(h_{\mathcal{A}}) > \epsilon) < \delta. \tag{4.1}$$

We will make use of the three following basic facts that we admit without proof:

(i) For all $x \in \mathbb{R}$, it holds that $1 + x \leq e^x$.

(ii) For all sets $A, B \in \mathcal{F}$ and probability P, it holds that $P(A \cup B) \leq P(A) + P(B)$.

(iii) If $A \subset B$, then $P(A) \leq P(B)$

Fix $m \in \mathbb{N}$, let us bound $D^m(S : R_D(h_{\mathcal{A}}) > \epsilon)$. Define the set of bad predictors

$$\mathcal{H}_b := \{h \in \mathcal{H} : R_D(h) > \epsilon\},$$

and define the set of misleading datasets of size m

$$M := \{S : \exists h \in \mathcal{H}_b, L(h) = 0\}.$$

(We used "misleading" to stress that even though the hypothesis class contains a predictor with null empirical error, this predictor fails to achieve a generalisation smaller than our threshold ϵ.) Finally, for all $h \in \mathcal{H}$, define the set of misleading datasets **for** h by

$$M(h) := \{S : h \in \mathcal{H}_b, L(h) = 0\}.$$

Note that by definition, we can write

$$M = \bigcup_{h \in \mathcal{H}_b} M(h).$$

In particular, thanks to the fact (ii) above, we have that

$$D^m(M) \leq \sum_{h \in \mathcal{H}_b} D^m\left(M(h)\right).$$

Because the m samples of the dataset are independent, for all $h \in \mathcal{H}_b$, we have that

$$D^m\left(M(h)\right) = D^m\left(S : h(x_i) = y_i, \forall i \in \{1, \dots, m\}\right) = \prod_{i=1}^{m} D\left(x_i : h(x_i) = y_i\right),$$

and by definition of the generalisation error,

$$\mathbb{E}_{x \sim D}(1_{h(x) \neq y}) = D\left(x_i : h(x_i) \neq y_i\right)$$

we moreover have that for each data point,

$$D\left(x_i : h(x_i) = y_i\right) = 1 - R_D(h) \leq 1 - \epsilon,$$

since we chose h in the bad hypothesis set $\mathcal{H}_b$. In particular, using the basic fact (i), we see that

$$D^m\left(M(h)\right) \leq (1 - \epsilon)^m \leq e^{-\epsilon m}.$$

Putting everything together, we have shown that

$$D^m(M) \leq \sum_{h \in \mathcal{H}_b} e^{-\epsilon m} \leq |\mathcal{H}| e^{-\epsilon m}.$$

Recall that our goal is to establish (4.1), which is not equivalent to the above bound. Indeed, we just upper bounded **the sum over bad hypotheses** of the probability to sample a misleading dataset. Nonetheless, the probability of sampling a misleading dataset for the trained predictor (as in (4.1)) is upper bounded by the left hand-side above, thanks to the basic fact (iii): since $\{S : R_D(h_{\mathcal{A}}) > \epsilon\} \subset \bigcup_{h \in \mathcal{H}_b} \{S : R_D(h)\} = M$, we have that

$$D^m(S : R_D(h_{\mathcal{A}}) > \epsilon) \leq D^m(M) \leq |\mathcal{H}| e^{-\epsilon m}.$$

It suffices now to check that for $m > m(\epsilon, \delta) = \frac{\log(|\mathcal{H}|/\delta)}{\epsilon}$, the right hand-side above is smaller than δ, which is the case, hence concluding the proof. $\square$

Given a finite hypothesis class and two numbers $\epsilon, \delta \in (0, 1)$, Theorem 4 provides a sufficient number of data points to learn a good predictor for binary classification, that is to say with generalisation error smaller than ϵ, with a probability greater than $1-\delta$, even in the worst case scenario where the labeling function (in $\mathcal{H}$) and the data distribution are chosen adversarially.

We can analyse the bound of Theorem 4 and note that:

- as $m \to \infty$, the probability of picking a misleading set for which $h \in \mathcal{H}_b$ yields $L(h) = 0$ decreases
- as $|\mathcal{H}|$ increases, so does the (adversarial) probability of finding bad hypothesis.
- the smalle we choose ϵ (i.e.: we want better accuracy), the greater we need to choose m.

Remark 13. It is important to note that Theorem 4 does **not** say that if $m < m(\epsilon, \delta)$, then training is likely to fail. It says that if $m < m(\epsilon, \delta)$, then there could exist a labeling function $f \in \mathcal{H}$ and a data distribution D that could make training fail.

4.2.2 Agnostic PAC learnability

Recall Definition 4.1.3, where agnostic PAC learnability was defined. We proved in Theorem 4 that any finite hypothesis class is PAC learnable, and derived a bound for the number of data samples that guarantee success of training. We are now interested in removing the realisability assumption:

Theorem 5. *If $\mathcal{H}$ is finite, then it is agnostic PAC learnable. Moreover, the map $m : (0,1)^2 \to \mathbb{N}$ in Definition 4.1.3 can be chosen as*

$$m(\epsilon, \delta) := \left\lceil \frac{2\log(2|\mathcal{H}|/\delta)}{\epsilon^2} \right\rceil .$$

Before we prove Theorem 5, we need to introduce the uniform convergence property:

Definition 4.2.1. We say that a hypothesis class $\mathcal{H}$ has the uniform convergence property if there is a function $m^{UC} : (0,1)^2 \to \mathbb{N}$ such that for every $\epsilon, \delta \in (0,1)$ and for every probability distribution D over $\mathcal{X}$, if $m \geq m^{UC}(\epsilon, \delta)$ and the m data samples in S are sampled i.i.d. with common law D, it holds that

$$D^m \left(\exists h \in \mathcal{H} : |R_D(h) - L(h)| > \epsilon \right) < \delta.$$

In a sentence, the uniform convergence property states that provided that m is large enough, the generalisation gap is likely to be small across all data distribution D and labeling function f.

Proof of Theorem 5. The strategy is the following:

- we show that every finite $\mathcal{H}$ has the uniform convergence property of Definition 4.2.1;
- we show that if $\mathcal{H}$ has the uniform convergence property, then it is agnostic PAC learnable with $m(\epsilon, \delta) = m^{UC}(\epsilon/2, \delta)$.

Let $\mathcal{H}$ be a finite hypothesis class. To show that it has the uniform convergence property, we admit without proof that

$$D^m \left(|R_D(h) - L(h)| > \epsilon \right) \leq 2e^{-2m\epsilon^2}, \quad \forall h \in \mathcal{H}.$$

It is a simple application of Hoeffding's inequality (that can easily be found online with its proof), that provides a bound on the probability that an empirical mean is far from its expectation. (Note that the expectation of

$L(h)$ integrated over the datasets under D^m is indeed given by $R(h)$.) We now use the basic fact (ii) in the proof of Theorem 4 to write

$$D^m\left(\bigcup_{h\in\mathcal{H}}\{|R_D(h)-L(h)|>\epsilon\}\right)\leq\sum_{h\in\mathcal{H}}2e^{-2m\epsilon^2}\leq|\mathcal{H}|2e^{-2m\epsilon^2}.$$

Hence, we can choose $m^{UC}(\epsilon,\delta)=\frac{\log(2|\mathcal{H}|/\delta)}{2\epsilon^2}$ and we see that $\mathcal{H}$ has the uniform convergence property.

We now show that the uniform convergence property implies the agnostic PAC learning property. Let $h_{\mathcal{A}}$ denote the predictor obtained by the training algorithm and let h^* be the optimal predictor within the class, that is

$$h^*:=\arg\min_{h\in\mathcal{H}}R_D(h).$$

Define the event $E_{\text{unif}}:=\{\forall h\in\mathcal{H}:|R_D(h)-L(h)|<\epsilon\}$. On the event E_{unif}, we have that

$$R_D(h_{\mathcal{A}})\leq L(h_{\mathcal{A}})+\epsilon\leq L(h^*)+\epsilon\leq R_D(h^*)+2\epsilon.$$

Note that the uniform convergence property ensures that $D^m(E_{\text{unif}})>1-\delta$, which means in particular that the above inequalities hold true with a probability greater than $1-\delta$.

This shows that by setting $m(\epsilon,\delta):=m^{UC}(\epsilon/2,\delta)$, then $\mathcal{H}$ satisfies the agnostic PAC learnability property. More explicitely,

$$m(\epsilon,\delta)=\frac{2\log(2|\mathcal{H}|/\delta)}{\epsilon^2},$$

as claimed, which concludes the proof. □

4.3 Infinite sized hypothesis classes *

So far, we have assumed that $|\mathcal{H}|$ is finite. We showed that finite classes are learnable and that the sample complexity of a hypothesis class is upper bounded by an expression that involves the log of its size. Is there something similar to this, when we consider $|\mathcal{H}| = \infty$? Namely, we want to say something about the expressiveness of a set of functions.

Example: linear classification in 2-d, 2 points, 3 points, 4 points.

We say linear functions are expressive enough to *shatter* 3 points.

Definition 4.3.1. A set S of examples is shattered by a set of functions $\mathcal{H}$ if for every partition of the examples in S into positive and negative examples, there is a function $f_w \in \mathcal{H}$ that gives exactly these labels to the examples.

Definition 4.3.2. (Vapnik–Chervonenkis dimension) The VC-dimension of a hypothesis class $\mathcal{H}$, denoted VCdim($\mathcal{H}$), is the maximal size of a set $C \subset \mathcal{X}$ that can be shattered by $\mathcal{H}$. If $\mathcal{H}$ can shatter sets of arbitrarily large size, we say $\mathcal{H}$ has infinite VC-dimension.

$\mathcal{H}$	VC-dim
Half intervals	1
Intervals	2
Half-spaces in the plane	3
Neural networks	number of parameters

For infinite hypothesis sets, VCdim($\mathcal{H}$) takes the role of $\log(|\mathcal{H}|)$ for finite hypothesis sets. For example: Given a sample S with m examples, find some $h \in \mathcal{H}$ that with at least probability $1 - \delta$, the hypothesis f_w has error less than ϵ if

$$m \geq \frac{1}{\epsilon}\left(8\text{VCdim}(\mathcal{H})\log\frac{13}{\epsilon} + 4\log\frac{2}{\delta}\right).$$

4.4 Bias-complexity tradeoff and Bias-variance tradeoff

4.4.1 Existence of noise

Consider again the setting of classification, with $\mathcal{Y} = \{-1, 1\}$. Let us introduce the notion of noisy labels.

Suppose we have a dataset $S = \{(x_1, y_1), \dots, (x_n, y_n)\}$. We now suppose that each data point and its corresponding label are independently drawn from an unknown data distribution, i.e., $(x_i, y_i) \sim \mathcal{D}(X, Y)$. Note the difference with the setup where the labels are given by a deterministic labeling function f: here the same x can have a positive probability to be labeled by y or y', with $y \neq y'$.

This lack of a perfect labeling function f accounts for the noise on the label. For example, you can think of this as, for example, the features do not contain all the information needed to attribute the label in a deterministic way. This setting is a bit closer to reality – because of maybe lack of information, noise, or other source of uncertainty, the labelling function might not be deterministic.

Example 4.4.1. Suppose we have a model for tastiness of a papaya given the colour and softness. Let's say most soft papayas with bright colour are tasty. However, we can have the situation that the papaya is soft and bright, and still not tasty (e.g.: bad climate?), even if it's unlikely.

Under this new assumption that there's also noise on the labels, we can write the theoretical optimal classifier:

Definition 4.4.1. Bayes Optimal predictor: Given a probability distribution $\mathcal{D}$ over $\mathcal{X} \times \mathcal{Y}$, the predictor is defined as

$$f_{bayes} = \begin{cases} 1 & \text{if } \mathcal{D}(y = 1|x) \geq 1/2 \\ -1 & \text{otherwise} \end{cases}$$

- In the deterministic case, $\mathcal{D}(y = 1|x)$ is either 1 or 0, because we have a deterministic map f.

- Under uncertainty, on average, this predictor is optimal

$$R_D(f_{bayes})) \leq R_D(h) \quad \forall h \in \mathcal{H}$$

- We rarely have access to this classifier, because it implies we can evaluate the probability $\mathcal{D}(y = 1|x)$. Typically,

$$f_{bayes} \notin \mathcal{H}$$

The error made by f_{bayes} is, by definition,

$$R_D(f_{bayes}) = \mathbb{E}_x \left[1 - \max\{(\mathcal{D}(-1|x), \mathcal{D}(1|x))\}\right],$$

and this is the minimal theoretical error possible. This leads us to us defining *noise* as:

Definition 4.4.2. Given a distribution $\mathcal{D}$ over $\mathcal{X} \times \mathcal{Y}$, the noise at point $x \in \mathcal{X}$ is defined as:

$$\text{noise}(x) = 1 - \max\{(\mathcal{D}(-1|x), \mathcal{D}(1|x))\}.$$

The noise is a characteristic of the learning task and it's indicative of it's level of difficulty. For example, for a point x if the noise is $1/2$, it will be challenge to predict its label correctly. On the contrary, a noise of 0 means that there exists a labeling function.

4.4.2 Bias-complexity trade-off

In learning theory, we talk about bias-complexity trade-off. The generalisation error can be decomposed in two components (three components if we include noise):

$$R_D(h) = \epsilon_{approx} + \epsilon_{est} \tag{4.2}$$

where $\epsilon_{approx} = \min_{h' \in \mathcal{H}} R_D(h')$ and $\epsilon_{est} = R_D(h) - \epsilon_{approx}$

Approximation error: this is the minimum error achievable by a predictor in the hypothesis class $\mathcal{H}$. This term measures how much error we have because we restrict ourselves to a specific class i.e. how much *inductive bias* we have.

This error does not depend on the sample size and it's determined by the hypothesis class chosen. Enlarging the hypothesis class (e.g. making it more complicated) can decrease the approximation error.

Note that under the **realisability assumption**, the error is zero. In the agnostic case, the approximation error can be large.

Example 4.4.2. Using a finite polynomial basis to represent a non-polynomial function, there is an inherent approximation error.

Estimation error: Error between the approximation error and the error achieved by the ERM predictor. This is in general non-null because the empirical error is only an estimate of the generalisation error, therefore we do not necessarily reach the minimal error over the hypothesis set. This quantity depends on the **training set size** and on **the size and complexity of the hypothesis class**. (namely, ϵ_{est} increases logarithmically with size of $\mathcal{H}$ and decreases with m increasing).

Since we want to minimize the total error, we have a trade-off, called **bias-complexity trade off**. A rich $\mathcal{H}$ reduces the approximation error but might lead to high estimation error (overfitting), whereas a simple $\mathcal{H}$ reduces the estimation error, but increases the approximation error (underfitting).

4.4.3 Bias-Variance tradeoff

The *Bias-Variance trade-off* is typically referenced in computational statistics and it does not use the learning framework we have been describing (PAC). Nevertheless, it's used quite often to describe the notion of managing the complexity of a chosen model class with its different sources of errors.

In this section, we leave the binary classification setup to consider a regression task with the mean squared error. More precisely:

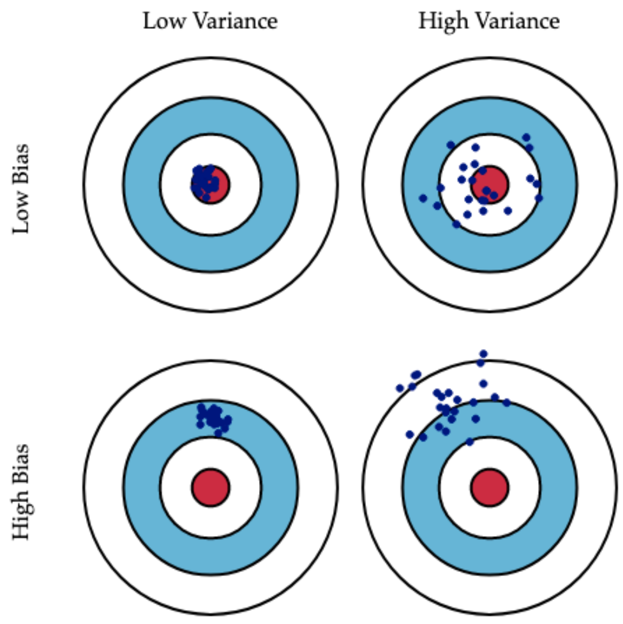


Figure 4.1: Graphical representation of bias and variance (precision and accuracy).

Lemma 6. Given a function $f = f(x)$ to be approximated, a dataset for training $S = \{(x_i, y_i),\ i = 1, \cdots, m\}$ of fixed size, with $y_i = f(x_i) + \epsilon$, and a predictor h that aims to approximate f, the expected mean-squared error for the predictor h_S, obtained after training on the dataset S with the ERM framework, can be decomposed as:

$$\underbrace{\mathbb{E}_{x,y,S}\left[(h_S(x) - y)^2\right]}_{\text{Expected Test Error}} = \underbrace{\mathbb{E}_{x,S}\left[\left(h_S(x) - \bar{h}(x)\right)^2\right]}_{\text{Variance}} + \underbrace{\mathbb{E}_{x,y}\left[(\bar{y}(x) - y)^2\right]}_{\text{Noise}} \tag{4.3}$$

$$+ \underbrace{\mathbb{E}_{x}\left[\left(\bar{h}(x) - \bar{y}(x)\right)^2\right]}_{\text{Bias}^2}, \tag{4.4}$$

where

$$\bar{h}(\cdot) = \mathbb{E}_{S \sim \mathcal{D}^m}\left[h_S(\cdot)\right],$$

denotes the expected predictor when sampling different datasets S from $\mathbb{D}^m$, and

$$\bar{y}(x) = \mathbb{E}_{y|x}\left[f(x) + \epsilon\right],$$

the expected value for y given x (as we consider y being noisy).

Proof. We consider the **expected error** for the predictor h_S that we obtained after training on the dataset S with the ERM framework, which can be written as:

$$\mathbb{E}_{(x,y)\sim\mathcal{D}}\left[(h_S(x) - y)^2\right] = \int\limits_x \int\limits_y \left(h_S(x) - y\right)^2 \mathcal{D}(x, y)\mathrm{d}y\mathrm{d}x.$$

We can write the expected Test Error (given the ERM framework and $\mathcal{H}$):

$$\mathbb{E}_{\substack{(x,y)\sim\mathcal{D} \\ S\sim\mathcal{D}^m}}\left[(h_S(x) - y)^2\right] = \int_S \int_x \int_y \left(h_S(x) - y\right)^2 \mathcal{D}(x, y)\mathcal{D}^m(S)\mathrm{d}x\mathrm{d}y\mathrm{d}S$$

The expected test error can then be decomposed as

$$\begin{aligned}\mathbb{E}_{\substack{(x,y)\sim\mathcal{D}\\ S\sim\mathcal{D}^m}}\left[(h_S(x)-y)^2\right] &= \mathbb{E}\left[\left[\left(h_S(x)-\bar{h}(x)\right)+\left(\bar{h}(x)-y\right)\right]^2\right] \\ &= \mathbb{E}\left[(h_S(x)-\bar{h}(x))^2\right] \\ &\quad + 2\mathbb{E}\left[\left(h_S(x)-\bar{h}(x)\right)\left(\bar{h}(x)-y\right)\right]+\mathbb{E}\left[\left(\bar{h}(x)-y\right)^2\right]\end{aligned} \tag{4.5}$$

The middle term of the above equation is 0:

$$\begin{aligned}\mathbb{E}_{\substack{(x,y)\sim\mathcal{D}\\ S\sim\mathcal{D}^m}}\left[\left(h_S(x)-\bar{h}(x)\right)\left(\bar{h}(x)-y\right)\right] &= \mathbb{E}_{x,y}\left[\mathbb{E}_S\left[h_S(x)-\bar{h}(x)\right]\left(\bar{h}(x)-y\right)\right] \\ &= \mathbb{E}_{x,y}\left[\left(\mathbb{E}_S\left[h_S(x)\right]-\bar{h}(x)\right)\left(\bar{h}(x)-y\right)\right] \\ &= \mathbb{E}_{x,y}\left[\left(\bar{h}(x)-\bar{h}(x)\right)\left(\bar{h}(x)-y\right)\right] \\ &= \mathbb{E}_{x,y}\left[0\right] \\ &= 0\end{aligned}$$

Returning to (4.5), we're left with the variance and another term

$$\mathbb{E}_{\substack{(x,y)\sim\mathcal{D}\\ S\sim\mathcal{D}^m}}\left[(h_S(x)-y)^2\right] = \underbrace{\mathbb{E}_{x,S}\left[\left(h_S(x)-\bar{h}(x)\right)^2\right]}_{\text{Variance}} + \mathbb{E}_{x,y}\left[\left(\bar{h}(x)-y\right)^2\right] \tag{4.6}$$

Expanding the second term in the above equation:

$$\begin{aligned}\mathbb{E}_{x,y}\left[\left(\bar{h}(x)-y\right)^2\right] &= \underbrace{\mathbb{E}_{x,y}\left[(\bar{y}(x)-y)^2\right]}_{\text{Noise}} + \underbrace{\mathbb{E}_x\left[\left(\bar{h}(x)-\bar{y}(x)\right)^2\right]}_{\text{Bias}^2} \\ &\quad + 2\,\mathbb{E}_{x,y}\left[\left(\bar{h}(x)-\bar{y}(x)\right)(\bar{y}(x)-y)\right].\end{aligned}$$

The third term in the equation above is 0 by [1]:

$$\begin{aligned}
\mathbb{E}_{x,y}\left[\left(\bar{h}(x)-\bar{y}(x)\right)\left(\bar{y}(x)-y\right)\right] &= \mathbb{E}_x\left[\mathbb{E}_{y|x}\left[\bar{y}(x)-y\right]\left(\bar{h}(x)-\bar{y}(x)\right)\right] \\
&= \mathbb{E}_x\left[\mathbb{E}_{y|x}\left[\bar{y}(x)-y\right]\left(\bar{h}(x)-\bar{y}(x)\right)\right] \\
&= \mathbb{E}_x\left[\left(\bar{y}(x)-\mathbb{E}_{y|x}\left[y\right]\right)\left(\bar{h}(x)-\bar{y}(x)\right)\right] \\
&= \mathbb{E}_x\left[\left(\bar{y}(x)-\bar{y}(x)\right)\left(\bar{h}(x)-\bar{y}(x)\right)\right] \\
&= \mathbb{E}_x\left[0\right] \\
&= 0.
\end{aligned}$$

This gives us the decomposition of expected test error as follows

$$\begin{aligned}
\underbrace{\mathbb{E}_{x,y,S}\left[\left(h_S(x)-y\right)^2\right]}_{\text{Expected Test Error}} = \underbrace{\mathbb{E}_{x,S}\left[\left(h_S(x)-\bar{h}(x)\right)^2\right]}_{\text{Variance}} + \underbrace{\mathbb{E}_{x,y}\left[\left(\bar{y}(x)-y\right)^2\right]}_{\text{Noise}} \\
+ \underbrace{\mathbb{E}_x\left[\left(\bar{h}(x)-\bar{y}(x)\right)^2\right]}_{\text{Bias}^2}
\end{aligned}$$

□

Variance: Captures how much your model changes if you train on a different training set. How "over-specialized" is your model to a particular training set (overfitting)? If we have the best possible model for our training data, how far off are we from the average model?

Bias: What is the inherent error that you obtain from your model even with infinite training data? This is due to your model being "biased" to a particular kind of solution (e.g. linear function). In other words, bias is inherent to your model.

[1]By the property of conditional expectation, we have:

$$E_{\mathbf{x},y}(f(y)) = \int_x\int_y f(y)D(x,y)dxdy = \int_x\int_y f(y)D(y|x)D(x)dxdy \tag{4.7}$$

$$= E_x[\int_y f(y)D(y|x)dy] = E_x[E_{y|x}[f(y)]] \tag{4.8}$$

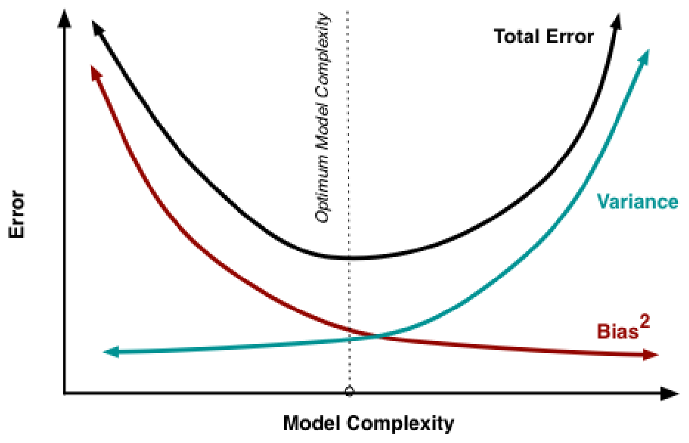


Figure 4.2: Representation of bias and variance contribution to error.

Noise: How big is the data-intrinsic noise? This error measures ambiguity due to your data distribution and feature representation. You can never beat this, it is an aspect of the data.

⋆ **Remark 9.** What about neural networks? Don't we have millions of parameters and still models generalise?

The bias–variance trade-off implies that a model should balance underfitting and overfitting: Rich enough to express underlying structure in data and simple enough to avoid fitting spurious patterns. However, in modern practice, very rich models such as neural networks are trained to exactly fit (i.e., interpolate) the data. Classically, such models would be considered overfitted, and yet they often obtain high accuracy on test data.

In [1][2], authors show a "double-descent" curve that includes the textbook U-shaped bias–variance trade-off curve by showing how increasing model capacity beyond the point of interpolation results in improved performance.

[2]Link to paper: `https://www.pnas.org/content/116/32/15849`

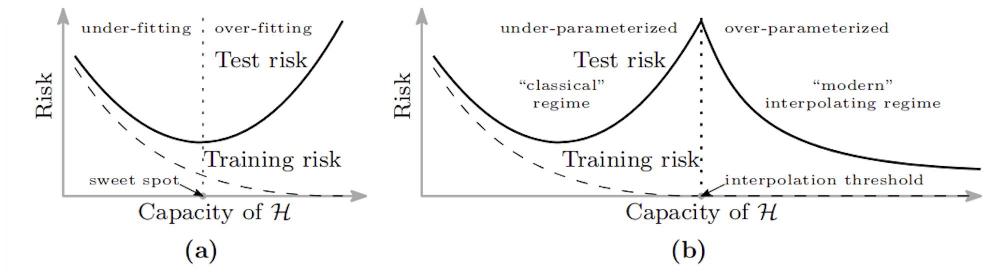


Figure 4.3: Curves for training risk and test risk. (A) The classical U-shaped risk curve arising from the bias-variance tradeoff. (B) The double-descent risk curve, which incorporates the U-shaped risk curve (i.e. the *classical* regime) together with the observed behaviour from using high-expressivity function classes (i.e.: the *modern* interpolating regime), separated by the interpolation threshold. The Predictors to the right of the interpolation threshold have zero training risk. Current topic of discussion.

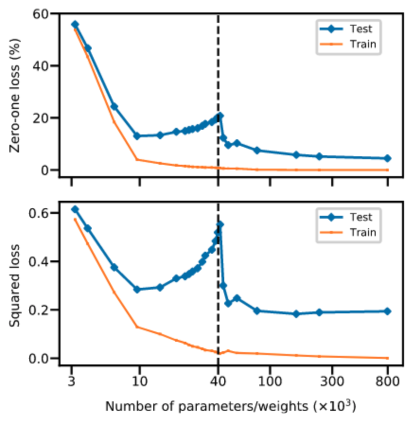


Figure 4.4: Double-descent risk curve for a fully connected neural network on MNIST. Training and test errors are shown for different losses. The dataset considered has 4000 datapoints, with feature dimension $d = 784$ and $K = 10$ classes. The number of parameters of the. network is given by $(d + 1)H + (H + 1)K$. The interpolation threshold (black dashed line) is observed at $n \cdot K$

Chapter 5

Linear Models

Contents

In this chapter, we study the family of *linear predictors* - a very useful family of hypothesis class. Linear predictors are intuitive, easy to interpret, theoretically sound, and fit the data reasonable well in many natural learning problems.

Let $\mathcal{X} = \mathbb{R}^d$ and $\mathcal{Y} = \mathbb{R}$ and consider the parametric class of affine functions

$$\mathcal{H} = \{f_w : x \to f_w(x) = \vec{w}^T\vec{x} + b : \vec{w} \in \mathbb{R}^d, b \in \mathbb{R}\}, \tag{5.1}$$

each function is parametrised by $\vec{w}$ and b, and takes as input a vector $\vec{x} \in \mathbb{R}^d$ and returns a scalar $\vec{w}^T\vec{x} + b$. One could consider the output space to be $\mathbb{R}^{d'}$ for $d' \geq 2$ without changing the theory, but making the presentation slightly more difficult. We will stick to $\mathcal{Y} = \mathbb{R}$ throughout the chapter.

We call the **homogeneous representation** as including b in $\vec{w}$ such that $\vec{w} = (w_1, ..., w_d, b)$. The class above is then equivalent to taking the input space to be $\mathbb{R}^d \times \{1\}$ and $f_w(\vec{x}) = \vec{w}^T\vec{x}$.

For binary classification, we can compose an element of $\mathcal{H}$ with the sign function which returns the sign of a real number (positive or negative).

5.1 Linear Regression

5.1.1 Cost function choice

Linear predictors are nice because they are linear, and thus preserve nice properties of the loss function $\ell : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}_+$ to the (empirical) *cost function* $C : \mathbb{R}^d \to \mathbb{R}_+$, defined on the parameter space by

$$C(w) := \frac{1}{m}\sum_{i=1}^{m} \ell(f_w(x_i), y_i). \tag{5.2}$$

For example, since the composition of differentiable and convex maps is differentiable and convex, it holds that if ℓ is convex in its first argument, then so is C on the parameter space. In particular, **training on $\mathcal{H}$ with gradient flow on the parameters is guaranteed to converge to a global minimum of the cost function** (admitted without proof, see Section 3.5).

Common choices of loss functions are

- the squared error loss (or L_2 loss) $\ell : (y, y') := \frac{1}{2}(y - y')^2$,

- the absolute error loss (or L_1 loss) $\ell : (y, y') \mapsto |y - y'|$,
- the Huber loss

$$\ell(y, y', \delta) = \begin{cases} \frac{1}{2}(y - y')^2 & \text{for } |y - y'| < \delta \\ \delta(|y - y'| - \frac{1}{2}\delta) & \text{otherwise.} \end{cases}$$

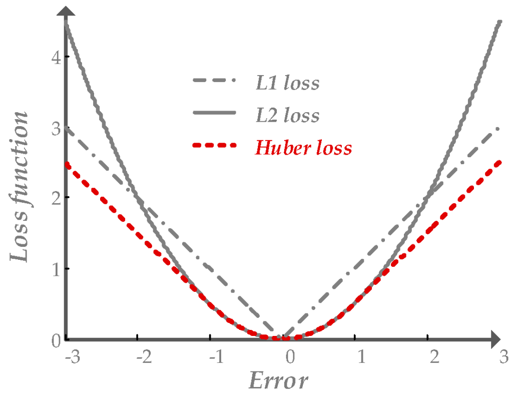


Figure 5.1: Different loss functions shapes.

The squared error loss is nice for theoretical reasons: convex, differentiable, no hyperparameters, but it is not robust to outliers (i.e.: your total error might be dominated by a point which is an outlier (the difference grows squared)). On the contrary, the absolute error loss is less prone to potential outliers (see 5.2[1]), and is convex too. However, it is not differentiable at 0. The Huber loss is a compromise between squared and absolute error losses: it is convex and both differentiable and robust to outliers. The price to pay is that it has a hyper parameter δ that has to be tuned.

[1]Notebook: `https://colab.research.google.com/drive/1ucl_aC8Q_q8Y5DC4uPpBqi5DyiFbSIBi?usp=sharing`

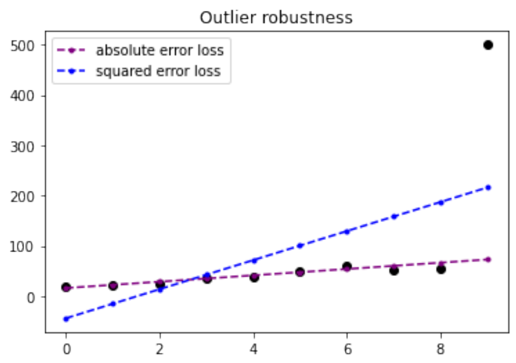


Figure 5.2: Example of optimizing the squared-error loss or absolute error loss, with presence of outlier.

5.1.2 Explicit solution

Least squares is the method that solves the empirical risk minimization problem for the hypothesis class (5.1) with respect to the squared loss. We want to find w that minimizes

$$\arg\min_w C(w) = \arg\min_w L(f_w) = \arg\min_w \frac{1}{2m}\sum_{i=1}^{m}(w^T x_i - y_i)^2.$$

(Note that here we are using the homogeneous notation: $w = (w_1, ..., w_n, b)$, $x_i = (x_{i1}, .., x_{in}, 1)^T$.) We will use the more compact notation and equivalent formulation

$$\arg\min_w C(w) = \frac{1}{2m}\arg\min_w ||Xw - Y||^2, \tag{5.3}$$

where $X = (x_{ij})_{ij} \in \mathbb{R}^{m\times n}$, $Y = (y_1, \ldots, y_m)^T$, and $||\cdot||$ is the Euclidean norm in $\mathbb{R}^m$.

Theorem 7. *The linear regression problem with square loss as in* (5.3) *satisfies the following:*

(i) if $X^T X$ is invertible, then the solution is unique, given by

$$w = (X^T X)^{-1} X^T Y;$$

(ii) *if X^TX is not invertible, then there are infinitely many solutions. Moreover, the minimal L_2 norm solution is given by*

$$w = X^+Y,$$

where X^+ denotes the pseudo inverse[2] *of X.*

Remark 14. Note that if we assume that all datapoints are distinct (i.e. X is full-rank), it holds that X^TX is invertible if we are in the situation $m \geq n$. (at least as many datapoints m than degrees of freedom n) [3] Similarly, if we are in the situation of $m < n$, then X^TX is not invertible.[4]

Proof. We prove (i), as the proof of (ii) is an exercise of homework 2.

To solve this problem, we calculate the gradient of the cost function C defined in (5.2) and set it to zero:

$$\begin{aligned}
\nabla_w C(w) &= \frac{1}{2m}\nabla_w ||Xw - Y||^2 = \frac{1}{2m}\nabla_w (Xw - Y)^T(Xw - Y) \\
&= \frac{1}{2m}\nabla_w \left((Xw)^T(Xw) - (Xw)^Ty - y^T(Xw) + Y^TY\right) \\
&= \frac{1}{2m}\nabla_w \left((Xw)^T(Xw) - 2(Xw)^Ty\right)
\end{aligned}$$

Using the lemma you proved in the homework, we write:

$$\begin{aligned}
\nabla_w (Xw)^T(Xw) &= \nabla_w (w^T(X^TXw)) \\
&= ((X^TX) + (X^TX)^T)w = (X^TX + X^TX)w \\
&= 2(X^TX)w,
\end{aligned}$$

[2]The pseudo inverse of a matrix generalises the inverse to non-invertible matrices. The geometric interpretation is the following: for any matrix $X \in \mathbb{R}^{m\times n}$, denote by $\text{Im}(X) \subset \mathbb{R}^m$ its image. Then for all element $v \in \text{Im}(X)$, there exists a unique element $u \in \mathbb{R}^n$ such that $Xu = v$. The pseudo inverse $X^+ \in \mathbb{R}^{n\times m}$ returns the following: for any $v \in \mathbb{R}^m$, consider its unique orthogonal projection $\pi(v)$ onto $\text{Im}(X)$, then X^+v is the unique $u \in \mathbb{R}^n$ such that $Xu = \pi(v)$.

[3]Why? Consider the dimension and rank of X^TX.

[4]Why? When $m < n$, X has at most rank m. As such, $X^TX \in \mathbb{R}^{n\times n}$ has also at most rank m.

and

$$\nabla_w 2(Xw)^T y = \nabla_w 2w^T X^T y = \partial_{wi}(X^T y)_i = 2X^T y.$$

This yields

$$\nabla C(w) = \frac{1}{2m}(2X^T Xw - 2X^T y) = 0. \tag{5.4}$$

Given that the Hessian of C is a semipositive definite matrix, we have a convex function. Hence, w^* is a global minimum of $C(w)$ if and only if $\nabla C(w^*) = 0$; we want to find w^* such that $\nabla C(w^*) = 0$. This means w^* is a minimum if and only if it satisfies the normal equation

$$X^T X w^* = X^T Y$$

Since we assume that $X^T X$ is **invertible**, we have

$$w^* = (X^T X)^{-1} X^T Y, \tag{5.5}$$

as claimed. □

Note that to prove (ii), we can proceed as in (i) up to Equation (5.4), but then we cannot invert $X^T X$. We can still seek for a solution to $Xw = Y$, but it is not unique. This means there is an infinite number of w^* that achieve the same minimal square error on the training data. This is called an over-determined problem, we have too many degrees of freedom in the problem and not enough constraints (data).

Which solution to seek for in the over-determined case? We can solve the following minimization problem instead:

$$w^* = \arg\min_w ||w||^2 \text{ subject to } \min_w ||Xw - y||^2$$

i.e. we find a solution with minimal L_2 norm *in the parameters space.*

To do so, one uses some linear algebra tools. Recall that if $A = U\Sigma V^T$ is the Singular Value Decomposition[5] of A, the pseudo-inverse of A is defined as $A^+ = V\Sigma^+ U^T$. Note that A is $n \times m$ and Σ^+ is an $n \times m$ diagonal matrix $\Sigma^+ = diag[1/\sigma_1, 1/\sigma_2, \cdots, 1/\sigma_r, 0, \cdots, 0]$, where r is the rank of A.

[5]

If A is invertible, the pseudo-inverse is the same as the inverse. If A is not invertible, A^+A yields a $n \times n$ diagonal matrix with the first r diagonal entries to be 1, and 0 for the others. AA^+ will be a similar diagonal matrix $m \times m$.

Note: if we plug in $X = U\Sigma V^T$ into the solution for the $m > n$ case, we see that

$$X^+ = (X^T X)^{-1} X^T$$

Remark 15. **(No closed form solution)** In generality, one might not have access to the closed form solution of a learning task. Then, we can use optimisation techniques, such as **gradient descent**, as introduced in Section3.5.

But even when a closed-form solution is available for linear models, we might use gradient descent. This is because building $(X^T X)^{-1}$ can be expensive, whereas computing the gradient of C and iterating is cheap. Furthermore, because C is convex, we can find the optimal solution (for appropriate learning rate).

5.1.3 Regularization

In the overdetermined case $n > m$, in Theorem 7, we modified the objective to include some norm of w to achieve a unique solution. Indeed, (infinitely) many solutions may have zero empirical loss, but what we hope for is to have a small generalisation error as well. The practice of adding something extra to the minimization is one way to prevent overfitting, as well as to attain a unique solution.

Regularized Loss Minimization (RLM) is a learning rule in which we jointly minimize the empirical risk (ERM) and a regularization function.

Theorem 8. *(Singular Value Decomposition (SVD)). Given a real $m \times n$ matrix A, it can be decomposed as*

$$A = U\Sigma V^T,$$

where $U \in \mathbb{R}^{m\times m}$, $V \in \mathbb{R}^{n\times n}$ are orthogonal ($U^TU = UU^T = I, V^TV = VV^T = I$) and $\Sigma \in \mathbb{R}^{m\times n}$ is a diagonal matrix with non-negative real numbers in the diagonal. We denote $\Sigma = diag[\sigma_1, \cdots, \sigma_r, 0, \cdots, 0]$, each $\sigma_i > 0$ and r is the rank of A.

A regularization function is a mapping $\mathcal{R} : \mathbb{R}^P \to \mathbb{R}_+$, and the regularized loss minimization rule outputs a hypothesis in

$$\arg\min_w (L(w) + \mathcal{R}(w)).$$

As we mentioned, it helps preventing overfitting. Not only are we minimizing the empirical risk, but we are also minimizing some penalisation on the model. Intuitively, the regularization function measures the complexity of hypotheses. Regularization can also be seen as a stabilizer of the learning algorithm. An algorithm is considered **robust** if a slight change of its input does not change its output much (e.g. continuously).

The L_2 regularisation is standard and is given by:

$$\mathcal{R}(w) = \lambda ||w||^2, \quad \lambda > 0,$$

which in the context of linear regression leads to the **Ridge regression**, defined by the following minimisation problem:

$$\min_w \frac{1}{2m} \sum_{i=1}^{m} (y_i - w^T x_i)^2 + \lambda ||w||^2, \quad \lambda > 0, ||w||^2 = \sum_{j=1}^{n} w_j^2 \tag{5.6}$$

Theorem 9. *Consider the ridge regression problem* (5.6)*. If $-m\lambda$ is not an eigenvalue of $X^T X$, then the solution is unique and is given by*

$$w^* = (X^T X + \lambda m I)^{-1} X^T Y.$$

The proof can be done as in the non-regularised case by setting the gradient of the cost to 0.

Another type of regularisation is the L_1 norm, namely

$$\mathcal{R}(w) = \lambda |w|_1, \quad |w|_1 = \sum_{i=1}^{n} |w_i|,$$

which defines the **Lasso regression**, leading to the optimisation problem:

$$\min_w \frac{1}{2m} \sum_{i=1}^{m} (y_i - w^T x_i)^2 + \lambda |w|_1, \quad \lambda > 0.$$

However, there's no closed form solution for the general case.

Remark 16. We note that a minimal norm solution (as seen in Theorem 7) **is not** a solution of the norm-regularised problem. Nevertheless, it can be shown that as $\lambda \to 0$, the solution of the L_2 regularized problem converges to the minimal L_2 norm solution of the original (non-regularized) problem.

The example below compares minimal L_1 and L_2 norm solutions, as well as L_2 regularized solutions.

Example 5.1.1. Let $x_1 = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$, $x_2 = \begin{pmatrix} 2 \\ 2 \end{pmatrix}$, $y_1 = 1$ and $y_2 = 2$. One can see that the solutions of $\min_{w \in \mathbb{R}^2} \sum_{i=1}^{2} (w^T x_i - y_i)^2$ are given by the line $w = \begin{pmatrix} a \\ 1-a \end{pmatrix}$, $a \in \mathbb{R}$.

The minimal L_2 norm solution is obtained for $a = 1/2$, whereas the segment $a \in [0,1]$ contains all minimal L_1 norm solution of the problem.

However, if we regularise with an L_2 penalty on the parameters, i.e. we minimise $\min_{w \in \mathbb{R}^2} \frac{1}{4} \sum_{i=1}^{2} (w^T x_i - y_i)^2 + \lambda ||w||_2^2$ for some $\lambda > 0$, then $w = \begin{pmatrix} 1/2 \\ 1/2 \end{pmatrix}$ is not a solution (one can convince oneself by taking the gradient and noting that it is non zero).

The conclusion is similar for the L_1-regularised problem[6].

Remark 17. Note that now the scale of the vectors in X matters. Why? We want $\vec{w}$ to be small, if features span different magnitudes, the contribution of a large feature will dominate the regression. So it's necessary, when considering Ridge or Lasso regression, to center and normalise the data X.

5.1.4 Representing nonlinear functions using basis functions

As we have seen, linear models have nice theoretical guarantees. What if the relationship between inputs and outputs isn't linear? It turns out we

[6]An example of a regression weights being optimised under different regularisation strategies. `https://developers.google.com/machine-learning/crash-course/regularization-for-sparsity/l1-regularization`

can use linear models to express non-linear relationships. Indeed, suppose we have data given in the form (see figure 5.3). So, we can understand that

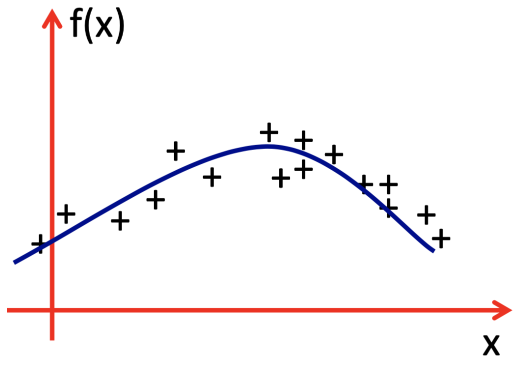


Figure 5.3: Quadratic relation

our approximator could benefit from having non-linear features. Namely,

$$f_w(x) = w_1 x^2 + w_0 x + b$$

Our linear coefficients are $w = [b, w_0, w_1]$, and our "features" become $[1, x, x^2]$. We are finding a linear model on "*nonlinear features*", namely, given by x^2.

In general, one can fit non-linear functions via linear regression using a transformation ϕ which applies nonlinear transformations on our features:

$$f_w(x) = \sum_{i=1}^{d} w_i \phi_i(x)$$

For example, if we consider polynomial transformations of our feature space, ϕ can be given as:

- Feature space is is 1-dim: $\phi(x) \rightarrow (1, x, x^2, ..., x^k)$
- Feature space is is 2-dim: $\phi(x) \rightarrow (1, x_1, x_2, x_1^2, x_2^2, x_1x_2, ..., x_1^k, x_2^k)$
- Feature space is is d-dim: vector of all monomials in x_1 to x_p of degree up to k.

5.2 Classification

Let us turn again to the classification problem, where we are considering a dataset with binary labels, i.e.: $S = \{(x_i, y_i) : i = 1..m\} \subset \mathbb{R}^n \times \{-1, 1\}$.

Assumption: We suppose that the dataset $S \subset \mathbb{R}^n \times \{-1, 1\}$ is *linearly separable*, i.e. there exists a hyperplane $P = \{x \in \mathbb{R}^n : w^T x + b = 0\}$ for some fixed $\in \mathbb{R}^n_*, b \in \mathbb{R}$, that separates the data. Formally, for all $i = 1..m$, $y_i = 1$ if and only if $w^T x_i + b > 0$ and similarly, $y_i = -1$ if and only if $w^T x_i + b < 0$.

The class of half-spaces is defined as follows:

$$\mathcal{H} = \{f_w : x \to f_w(x) = \operatorname{sign}(\vec{w}^T \vec{x} + b) : \vec{w} \in \mathbb{R}^n, b \in \mathbb{R}\}.$$

Let us illustrate this hypothesis class geometrically, considering the case $n = 2$. Each hypothesis forms a hyperplane that is perpendicular to the vector $\vec{w} = (w_1, w_2)$ and intersects the vertical axis at the point $(0, -b/w_2)$. The instances that are "above" the hyperplane, that is, share an acute angle with $\vec{w}$, are labeled positively. Instances that are "below" the hyperplane, that is, share an obtuse angle with $\vec{w}$, are labeled negatively.

How do we find a good $\vec{w}$? We can consider the following minimization:

$$\arg\min_{w} C(w) = \arg\min_{\vec{w}, b} \sum_{i=1}^{m} \mathbb{1}_{\{y_i \neq \operatorname{sign}(\vec{w}^T \vec{x_i} + b)\}}.$$

However, the loss function $\ell(y, y') = \mathbb{1}_{\{y \neq y'\}}$, called **0-1 loss** is not convex nor continuous. Thus, to make the optimisation problem easier, we introduce a surrogate loss, called the **Perceptron loss**[7]:

$$\ell_{perc}(x, y) = \max(0, -y(\vec{w}^T \vec{x} + b)).$$

[7]Note that if y and $w^T x - b$ have the same sign, then the second term is negative and thus the $\ell_{perc}(x, y)$ is zero. If the signs are opposite, then error quantity will be positive nonzero.

5.2.1 Perceptron algorithm

This ℓ_{perc} loss is useful for the sake of solving the optimisation problem[8].

As previously seen, we compute the gradient of the cost:

$$\begin{aligned}
C(w) &= \frac{1}{m}\sum_{i=1}^{m} \max(0, -y_i(w^T x_i + b)) \\
\nabla C(w) &= \frac{1}{m}\sum_{i=1}^{m} \nabla_w \max(0, -y_i(w^T x_i + b)) \\
\nabla C(w) &= \frac{1}{m}\sum_{i=1}^{m} \begin{cases} 0 & \text{if } y_i w^T x_i + b \geq 0 \text{ (correctly classified)} \\ -y_i x_i & \text{otherwise (misclassified).} \end{cases}
\end{aligned}$$

Then we can use the gradient descent algorithm to find a plane that linearly separates the data (if that's possible)[9]. Then, the gradient descent update becomes:

$$w_{t+1} = w_t - \eta_t \frac{1}{m} \sum_{i: y_i(w^T x_i + b) < 0} -y_i x_i.$$

The **Perceptron algorithm** does not give us guarantees except that it separates the data if the data is linearly separable (i.e.: if gradient descent converges, it finds a local minimum which by convexity is a global minimum). There are, however, infinitely many planes that separate the data.

[8] What about at $x = 0$? The derivative is not unique, but it can be approached with sub-gradients.

[9] Consider the code in: `https://colab.research.google.com/drive/1xDC8vlx6Eepl34xmcN6Cm-GhwFStM5KG?usp=sharing`, which shows the linearly separable case and a nonlinear separable case, and what happens with the perceptron algorithm.

5.2.2 Support Vector Machine

The Support Vector Machine (SVM) is a linear classifier that can be viewed as an extension of the Perceptron algorithm. In the context of binary classification in a linearly separable dataset, the Perceptron guarantees that you find a separating hyperplane, while the SVM finds the **maximum-margin separating hyperplane**. Refer to figure 5.4 for a comparison.

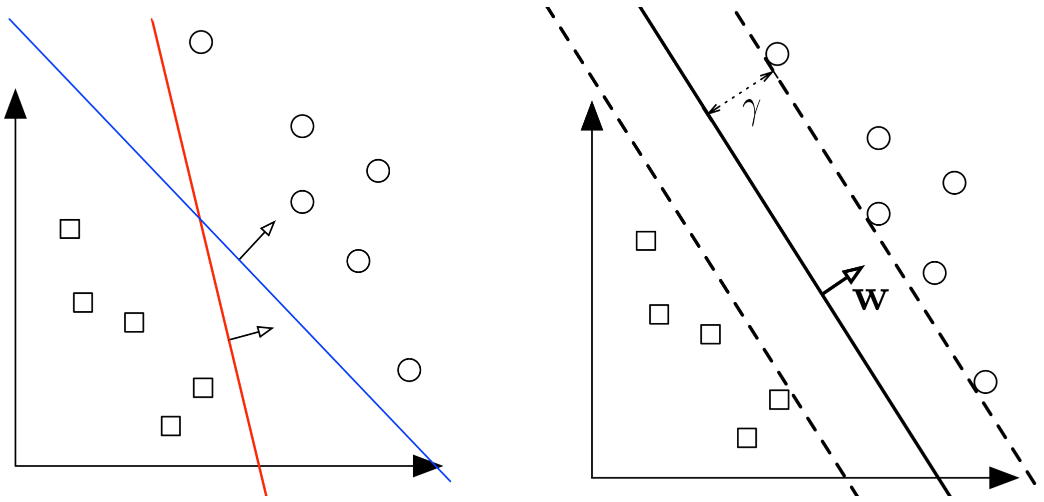


Figure 5.4: Two different separating hyperplanes for the same data set. (Right:) The maximum margin hyperplane. The margin, γ, is the distance from the hyperplane (solid line) to the closest points in either class (which touch the parallel dotted lines).

Definition 5.2.1. (**Margin**) Consider a separating hyperplane defined through w, b as the set of points $P = \{x \in \mathbb{R}^n : w^T x + b = 0\}$. The margin $\gamma(w, b)$ is defined as the distance from the hyperplane to the closest point across both classes.

Then, we can state the **SVM objective**:

$$\max_{w,b} \gamma(w, b) \;\; \text{s.t. } \forall i = 1..m, \; y_i(w^T x_i + b) \geq 0. \tag{5.7}$$

The SVM objective states that we want to find the hyperplane defined by w, b such that the distance of that plane to points in the dataset is maximized, and that the plane correctly classifies points.

The expression above is not yet computable. Namely, we need an explicit expression for the margin γ in order to solve this optimisation problem, however, you will see that it leads us to a max min problem (5.7), which is not trivial to solve. Luckily, turns out that the SVM objective can be reformulated in a much *nicer* formulation.

Proposition 1. The solution (w_*, b_*) of (5.7) is also the solution of

$$\begin{aligned} &\min_w w^T w \\ &\text{s.t. } \forall i \in \{1, \ldots, m\}, \quad y_i(w^T x_i + b) \geq 1. \end{aligned} \tag{5.8}$$

Remark 18. This new formulation (5.8) is a quadratic optimization problem. The objective is quadratic and the constraints are all linear. One can solve it efficiently with a Quadratically Constrained Quadratic Program solver. It has a unique solution whenever a separating hyper plane exists.

Before we prove Proposition 1, we need to establish some preliminary results such as finding a convenient expression for the margin of a separating hyperplane. Namely, how do we find the margin or, how do we compute the distance of an arbitrary point x to a hyperplane P?

Lemma 10. Let P be the hyperplane induced by a nonzero $w \in \mathbb{R}^n$ and $b \in \mathbb{R}$ and let $x \in \mathbb{R}^n$. We have that

$$\operatorname{dist}(P, x) = \frac{|w^T x + b|}{||w||}.$$

In particular, if $S = \{(x_1, y_1), \ldots, (x_m, y_m)\}$, then the margin of P with respect to S is given by

$$\gamma(w, b) = \min_{x \in S} \frac{|w^T x + b|}{||w||}.$$

Furthermore, we can rescale the parameters w and b, and thus the margin, without changing the hyperplane (i.e. invariant to rescaling)

$$\gamma(\beta w, \beta b) = \gamma(w, b) \quad \forall \beta \in \mathbb{R}, \beta \neq 0.$$

Proof. Let $d := x - x_P \in \mathbb{R}^n$, where x_P is the orthogonal projection of x onto hyperplane P. We know that $w^T x_P + b = 0$ by definition, which entails that

$$w^T(x - d) + b = 0.$$

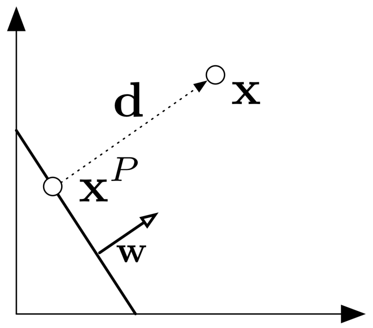


Figure 5.5: Distance between an arbitrary point x and hyperplane P.

What is d? It's the distance vector resulting from subtracting x from its projection x_P, and its norm is the minimum distance between x and any point on the hyperplane. Furthermore, note that d is colinear with w, so we can write $d = \alpha w$ for some $\alpha \in \mathbb{R}$. Then,

$$w^T(x - d) + b = 0$$
$$w^T(x - \alpha w) + b = 0$$
$$\alpha = \frac{w^T x + b}{w^T w} = \frac{w^T x + b}{||w||^2}$$

Now that we know α, we can compute the length of $d = \alpha w$, i.e. the distance of x to P as

$$||d|| = \sqrt{d^T d} = \sqrt{\alpha^2 w^T w} = \frac{|w^T x + b|}{||w||},$$

as claimed.

The margin of the hyperplane P to a dataset S follows by definition, i.e., we are finding the point in S for which the distance to the hyperplane is the smallest. From the expression of the margin, one then deduces its scale-invariance property, which concludes the proof. □

Remark 19. Symmetry of the margin: If the hyperplane is such that the margin γ to a labeled dataset is maximized, it must lie right in the middle of the two classes. In other words, γ must be the distance to the closest point within both classes. (If not, you could move the hyperplane towards data points of the class that is further away and increase γ, which contradicts that γ is maximized.)

Proof of Proposition 1. By plugging the expression of γ from Lemma 10 in the objective (5.7), we get

$$\max_{w,b} \left[\min_{x \in S} \frac{|w^T x + b|}{||w||} \right] \text{such that } \forall i,\ y_i(w^T x_i + b) \geq 0$$

We can pull the denominator outside of the minimization because it does not depend on x. Because the hyperplane is scale invariant, we can fix the scale of w, b any way we want. Let's be clever about it, and choose it such that

$$\min_{x \in S} |w^T x + b| = 1$$

Then, we can simplify the optimisation:

$$\max_{w} \frac{1}{||w||} \cdot 1 = \min_{w} ||w||,$$

and note that the w that (satisfies the constraints and) minimizes $||w||$ is the same that minimizes $||w||^2$. This is because $f(x) = x^2$ is monotonically increasing for $x \geq 0$ and $||w|| \geq 0$.

Then, the new constrained optimization problem becomes:

$$\begin{aligned} &\min_{w} w^T w \\ &\forall i, y_i(w^T x_i + b) \geq 0 \\ &\text{s.t.} \min_{i} |w^T x_i + b| = 1. \end{aligned}$$

These constraints are still hard to deal with, however luckily we can show that (for the optimal solution) they are equivalent to the much simpler (5.8).

$(\implies)$ We can write the constraints with the absolute value

$$|w^T x_i + b| = 1$$

as

$$y_i(w^T x_i + b) = 1$$

because we are in the separable case. And if the minimum is 1 then all other points are ≥ 1.

($\Longleftarrow$) Assume that for all $i \in \{1, \ldots, m\}$, $y_i(w^T x_i + b) \geq 1$. (e.g. larger than 1) is true, how can we guarantee that $|w^T x_i + b| = 1$ for at least one point x_i? Suppose it is not the case, then w is not minimized, and we can minimize it further. (Divide w, b by $||w||$.)

This concludes the proof. $\square$

Now that we have established the simpler formulation of the SVM problem (5.8), we can numerically find a solution to it, as mentioned in Remark 18. Although we do not have a general closed-form solution, more can be said about the maximal margin hyperplane.

5.2.3 Detour: Duality theory of constrained optimisation

In this section, we see how to handle constraints when dealing with an optimisation problem. We will then apply the method to SVM to express the solution of the SVM problem (5.8).

Equality constraints

Consider the following general constrained optimisation problem:

$$\begin{aligned} &\min_w \quad C(w) \\ &\text{s.t. } \ h_i(w) = 0, i = 1, ..., r, \end{aligned} \tag{5.9}$$

where the h_i's are $\mathcal{C}^1$ constraints.

One can use the method of *Lagrange multiplier* to solve (5.9), by turning a constrained optimisation into an unconstrained optimisation and introducing penalties on the violation of the constraints.

Definition 5.2.2. The *Lagrangian function* of (5.9) is defined for all $w \in \mathbb{R}^n$ and $\alpha \in \mathbb{R}^r$ by

$$\mathcal{L}(w, \alpha) = C(w) - \sum_{i=1}^{r} \alpha_i h_i(w). \tag{5.10}$$

The coefficients α_i's are called the *Lagrange multipliers*.

Intuition (hand-wavy). Suppose that w^* is the solution of (5.9). In particular, the constraints are satisfied and for each $i \in \{1, \dots, r\}$, if we move in a direction w orthogonal to $\nabla_w h_i(w^*)$, we locally have $h_i(w^* + \epsilon w) \approx h_i(w^*) + O(\epsilon^2)$. Now this reasoning applies if w is orthogonal to $\nabla_w h_i(w^*)$ simultaneously for all $i \in \{1, \cdots, r\}$. If $w^T \nabla_w C(w^*) \neq 0$, then we found a direction that decreases the cost in a neighborhood of w_*, while respecting the constraints, which contradicts that w^* is optimal. This means that, necessarily, $w^T \nabla_w C(w^*) = 0$. This means that the gradient of the cost at w^* is orthogonal to any w orthogonal to the gradient of the constraints at w^*: it holds that $\nabla_w C(w^*) \in \mathrm{Span}(\{\nabla_w h_i(w^*); i = 1, \dots, r\})$, that is, there exist $\alpha_1^*, \dots, \alpha_r^*$ such that

$$\nabla_w C(w^*) = \sum_{i=1}^{r} \alpha_i^* \nabla_w h_i(w^*). \tag{5.11}$$

The above intuition justifies the definition of the Lagrangian function, as it helps us write (5.11) in a concise manner as $\nabla_w \mathcal{L}(w^*, \alpha^*) = 0$. By computing the gradient of $\mathcal{L}(w, \alpha)$ with respect to w, α, and set it to 0, we can solve for the unknowns. Whether we are minimizing of maximizing $\mathcal{L}$, the α_i's play a role of penalising the solution if the constraints are violated. Consider $\nabla_{w,\alpha} \mathcal{L} = 0$ and write

$$\nabla_{w,\alpha} C(w) - \nabla_{w,\alpha} \sum_i \alpha_i h_i(w) = 0$$

Expanding this out, we have a vector where the first n entries with respect to w lead to:

$$\nabla_w C(w) - \sum_i \alpha_i \nabla_w h_i(w) = 0,$$

and the last r entries lead to

$$h_i(w) = 0, i = 1, ..., r.$$

Define $g_{\text{prim}} : w \mapsto \max_\alpha \mathcal{L}(w, \alpha)$. The solution to (5.11)[10] is given, in terms of the Lagrangian, as:

$$p^* = \min_w g_{\text{prim}}(w). \tag{5.12}$$

How does this solution relate to that of the original problem (5.9)? When considering $g_{\text{prim}}(w)$, we are fixing a w and then maximize over α. We see that as soon as one constraint is violated, say $h_i(w) \neq 0$, then we can let α_i go to plus or minus infinity (depending on the sign of $h_i(w)$) to make the Lagrangian blow up, therefore $g_{\text{prim}}(w) = +\infty$. In particular, $g_{\text{prim}}(w)$ is finite if and only if w satisfy the constraints (provided that $C(w)$ is finite). Hence, the solution w_* of (5.9) is the same as that of (5.12).

Inequality constraints

Similarly, we can write the Lagrangian for the following constrained optimisation problem (instead of equality in the constraints, we seek for $\leq$ constraints):

$$\begin{aligned} \min_w \quad & C(w), \\ \text{s.t.} \;\; & h_i(w) \leq 0, i = 1, ..., r. \end{aligned} \tag{5.13}$$

Definition 5.2.3. The Lagrangian function of (5.13) is defined for all $w \in \mathbb{R}^n$ and $\alpha \in [0, \infty)^r$ by

$$\mathcal{L}(w, \alpha) = L(w) + \sum_{i=1}^{r} \alpha_i h_i(w), \tag{5.14}$$

where the α_i's are called the Lagrange multipliers.

[10]Also called as the *primal problem/primal solution.*

Again, note that if the constraint on $h_i(w)$ is violated (i.e. if $h_i(w) > 0$), then if $\alpha_i > 0$, this will lead to an increase in the Lagrangian.

As in the equality constraints case, the solution to (5.13) is given, in terms of the Lagrangian, as:

$$p^* = \min_w g_{\text{prim}}(w) = \min_w \max_{\alpha \geq 0} \mathcal{L}(w, \alpha). \tag{5.15}$$

The primal problem, however, does not seem easier to solve than the original problem itself. Indeed, in the minimisation-maximisation problem (5.15), because we maximize over α after having chosen a w, when minimizing over w, we are restricted to candidates that satisfy the constraints (otherwise, the Lagrangian blow up as explained before). What we would like to do is to first choose α and only then minimize over w. This is what is called the *dual optimisation problem*, which turns the constrained min-max optimisation problem into a penalised max/min optimisation problem where we search for an optimal solution over α after minimizing over w. [11]

First, we write the dual function:

$$g_{\text{dual}}(\alpha) = \min_w \mathcal{L}(w, \alpha),$$

and then the *dual optimisation problem*:

$$d^* = \max_{\alpha \geq 0} g_{\text{dual}}(\alpha) = \max_{\alpha \geq 0} \min_w \mathcal{L}(w, \alpha). \tag{5.16}$$

How does solving this relate to solving (5.15)? We can note that the solution to (5.16) returns a lower bound to (5.15): first note that by definition, for every w, α, we have

$$\mathcal{L}(w, \alpha) \leq g_{\text{prim}}(w).$$

Taking the minimum over w yields

$$g_{\text{dual}}(\alpha) \leq p_*,$$

[11] If you are interested in this, start with the notion of duality in Linear Programming problems.

and then the maximum over α to get

$$d^* \leq p^*.$$

In some cases, the solution to the original optimisation problem (primal) is the same as the solution to the dual problem, i.e.:

$$d^* = p^*.$$

This is called *strong duality condition.* Spoiler alert: The strong duality condition holds for the SVM optimisation problem.

To ensure strong duality, we need some conditions on the candidate solutions, namely the **Karush-Kuhn-Tucker (KKT) conditions**[12]: we say that w_* and α_* fulfil the KKT conditions if and only if

$$\begin{aligned}
\frac{\partial}{\partial w_i}\mathcal{L}(w^*, \alpha^*) &= 0, i = 1, ..., n \\
\frac{\partial}{\partial \alpha_i}\mathcal{L}(w^*, \alpha^*) &= 0, i = 1, ..., r \text{ (stationarity} \quad \nabla\mathcal{L} = 0) \\
h_i(w^*) &\leq 0, i = 1, ..., r \text{ (primal feasibility)} \\
\alpha_i^* &\geq 0, i = 1, ..., r \text{ (dual feasibility)} \\
\alpha_i^* h_i(w^*) &= 0, i = 1, ..., r \text{ (complementary slackness)}
\end{aligned}$$

We admit the following result:

Theorem 11. *For an optimisation problem for which the strong duality condition holds, any primal optimal solution w^* and dual optimal solution α^* respect the KKT conditions. Conversely, if f and h_i are affine for all i, then the KKT conditions are sufficient for duality.*

Proof. Refer to book: "Convex Optimization" by Stephen P. Boyd. □

Cool, now what? Two facts:

[12]Subject to differentiability and convexity requirements.

- the SVM optimisation satisfies the strong duality condition.
- this means that the optimal w^* for the SVM problem (5.8) and α^* satisfy the KKT conditions.

SVM optimisation problem:

Recall the SVM optimisation problem

$$\begin{aligned}&\min_w w^T w\\ &\text{s.t. } \forall i:\ \ y_i(w^T x_i + b) - 1 \geq 0.\end{aligned}$$

The Lagrangian for the SVM problem[13]:

$$\mathcal{L}(w, b, \alpha) = w^T w - \sum_{i=1}^{m} \alpha_i (y_i(w^T x_i + b) - 1)$$

Observation 1: The SVM optimisation satisfies the strong duality condition, we can solve either the dual problem (finding α_i) or the primal (finding w).

Observation 2: The KKT conditions are obtained by setting the gradient of the Lagrangian with respect to the primal variables w and α to zero and plugging in the optimal solutions w^* and α^*.

Then, let's use the KKT conditions to say something about our solutions.

Using the stationarity condition:

$$\nabla_w \mathcal{L} = w^* - \sum_{i=1}^{m} \alpha_i^* y_i x_i = 0 \implies w^* = \sum_{i=1}^{m} \alpha_i^* y_i x_i. \tag{5.17}$$

The optimal weight vector w^*, which defines the normal to the hyperplane that maximizes the margin, is a linear combination of the training vectors $x_1, ..., x_m$.

[13]Sometimes you will see this multiplied by 1/2 to *simplify* the gradient computation...

Using the complementary slackness condition:

$$\forall i, \alpha_i^*[y_i(w^T x_i + b) - 1] = 0 \implies \alpha_i^* = 0 \text{ or } y_i(w^T x_i + b) = 1.$$

There are some vectors which lie on the margin, and for those, the corresponding α_i^* is nonzero.

Combining with (5.17), a vector x_i appears in the expansion of w^* if and only if $\alpha_i^* \neq 0$. The datapoints which appear in the expansion of w^* are called ‘‘*support vector*”. The support vectors define the maximum margin hyperplane. [14]

Note: While w is unique for the SVM problem, the support vectors are not. E.g in a hyperplane in N dimensions, we need $N+1$ points to define a hyperplane. If there are more support vectors than $N+1$, then we can choose different support vectors to specify the hyperplane.

- If you were to move one of the support vectors and retrain the SVM, the resulting hyperplane would change.
- If we move non-support vectors, the SVM hyperplane would not change (provided you don't move them too much, or they could turn into support vectors themselves).

We thus have the following:

Proposition 2. The SVM classifier f_* with parameters (w^*, b^*) that maximize the maximal margin hyperplane for a separable dataset $\{(x_i, y_i); i = 1, \ldots, m\}$ is of the form

$$f_*(x) = sign((w^*)^T x + b^*) = sign\left(\left(\sum_{i=1}^{m} \alpha_i^* y_i x_i\right)^T x + b^*\right).$$

[14]To find **b**, we can note that

$$y_j(w^T x_j + b) = 1$$

for some j (one of the support vectors). Then $b = y_j - w^T x_j$

Note that our approximating function is given by an inner product between support vectors and new datapoint x, which we will exploit in the future to learn non-linear hyper-planes.

5.2.4 Non-separable case

So far we have assumed that the data was linearly separable. This is often not the case, we can't find a hyperplane that separates between the two classes. In this case, there is no solution to the optimization problems stated above.

We can fix this by allowing the constraints to be violated ever so slightly with the introduction of slack

$$\begin{array}{c} \min_{w,b,\xi} w^T w + \lambda \sum_{i=1}^{n} \xi_i \\ s.t.\ \forall i\ y_i(w^T x_i + b) \geq 1 - \xi_i \quad C \geq 0 \\ \forall i\ \xi_i \geq 0 \end{array}$$

The slack variable ξ_i allows the input x_i to be closer to the hyperplane (or even be on the wrong side), but there is a penalty in the objective function for such "slack".

If λ is very large, the SVM becomes very strict and tries to get all points to be on the right side of the hyperplane. If λ is very small, the SVM becomes very loose and may "sacrifice" some points to obtain a simpler (i.e. lower $\|w\|_2^2$) solution.

Unconstrained Formulation: Let us consider the value of ξ_i for the case of $\lambda \neq 0$. Because the objective will always try to minimize ξ_i as much as possible, the equation must hold as an equality and we have:

$$\xi_i = \begin{cases} 1 - y_i(w^T x_i + b) & \text{if } y_i(w^T x_i + b) < 1 \\ 0 & \text{if } y_i(w^T x_i + b) \geq 1 \end{cases}$$

This is equivalent to the following closed form:

$$\xi_i = \max(1 - y_i(w^T x_i + b), 0).$$

If we plug this closed form into the objective of our SVM optimization problem, we obtain the following unconstrained version as loss function and regularizer:

$$\min_{w,b} \underbrace{w^T w}_{l_2-regularizer} + \lambda \sum_{i=1}^{n} \underbrace{\max\left[1 - y_i(w^T x_i + b), 0\right]}_{hinge-loss}.$$

Chapter 6

Kernel methods

Linear classifiers are great, but what if there exists no linear decision boundary? As it turns out, there is an elegant way to incorporate non-linearities into most linear classifiers.

We can make linear classifiers non-linear by applying a nonlinear mapping ϕ on the input feature vectors $\mathcal{X}$ to a higher-dimensional space $\mathcal{X}_H$, where linear separation is possible.

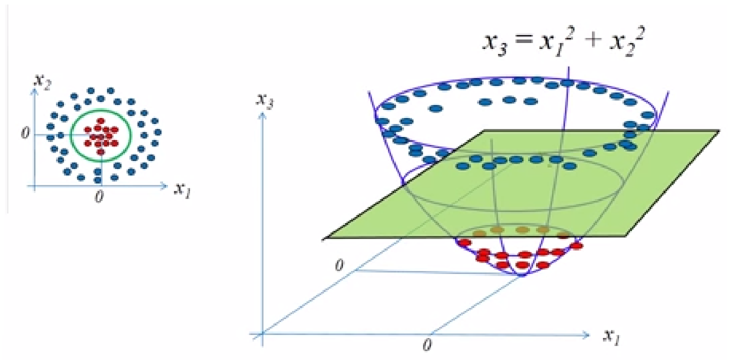


Figure 6.1: Nonlinear transformation on the input. Source: Scikit-learn. A cool visualisation: `https://www.youtube.com/watch?v=OdlNM96sHio&t=0s`

Some disadvantages:

- $\phi(x)$ might be very high dimensional.
- Do we have to build $\phi(x)$ from scratch?

In this chapter, we will talk about methods which use this idea of sending elements of $x \in \mathcal{X}$ onto some higher dimensional space $\mathcal{X}_H$ which we don't necessarily have to know much about, where our problem becomes easier to solve. All we need to know about this new space is that it is a so-called *Reproducible kernel Hilbert space.*

Before we can dive into what this means, we start with a brief review and introduction to key concepts.

Note: throughout this chapter, we always assume that the output space $\mathcal{Y} = \mathbb{R}$, to ease the notation.

6.1 Inner products and kernels

In $\mathbb{R}^n$, the dot product $x \cdot x' = \sum_{i=1}^n x_i x_i'$ can be seen as a way to measure the similarity between two elements x, x' (e.g. $x \cdot x' = 0$ if and only if x and x' are orthogonal, i.e. nothing of x can be used to represent x'). In an arbitrary vector space, this notion can be generalised:

Definition 6.1.1. An *inner product* in a real vector space H is a map $\langle \cdot, \cdot \rangle : H \times H \to \mathbb{R}$ that satisfies the following properties: for all $u, v, w \in H$ and $\alpha \in \mathbb{R}$,

1. $\langle u, v \rangle = \langle v, u \rangle$, (symmetry)
2. $\langle u + \alpha v, w \rangle = \langle u, w \rangle + \alpha \langle v, w \rangle$, (bilinearity)
3. $\langle v, v \rangle \geq 0$ with equality if and only if $v = 0$. (positive-definiteness)

Note that by symmetry, we only need to check the linearity in the first variable to get the bilinearity.

We will sometimes write $\langle \cdot, \cdot \rangle_H$ to specify on which space we consider the inner product, in order to avoid ambiguity.

Example 6.1.1. **Inner product examples:**

- The Euclidean space $\mathbb{R}^n$, where the inner product is given by the dot product:

$$\langle (x_1, \cdots, x_n), (y_1, \cdots, y_n) \rangle = x_1 y_1 + \cdots + x_n y_n.$$

- The vector space of real functions whose domain is an closed interval $[a, b]$ with inner product:

$$\langle f, g \rangle = \int_a^b f(x) g(x) dx.$$

The inner product induces a norm on H:

$$||u||_H := \sqrt{\langle u, u \rangle_H}.$$

In particular, once we have a norm, we have a (induced) metric ($d(u, v) := ||u - v||_H$), and we can make sense of the notion of *completeness* of a space.

Remark 20. **Reminder.** A space H is said to be *complete* (w.r.t. a norm $||\cdot||_H$) if and only if for any sequence $(u_n)_{n \geq 1}$ of elements of H that converges to some u_* (w.r.t. to norm $||\cdot||_H$), it holds that $u_* \in H$.
An inner product space that is complete w.r.t. the norm induced by its inner product is called a *Hilbert space.*

Theorem 12. *(Riesz Representation theorem) If T is a linear bounded operator on a Hilbert space H, then there exists some $g \in H$ such that*

$$T(f) = \langle f, g \rangle_H \quad \forall f \in H.$$

This means every (bounded) linear operator $T : H \to \mathbb{R}$, on a Hilbert space can be written as an inner product of some $g \in H$.

Definition 6.1.2. Let $\mathcal{X}$ be an non empty set. We say that a symmetric[1] function $K : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ is a *positive definite kernel* (pd kernel) if and only if for any fixed $n \in \mathbb{N}$ and $c_1, \cdots, c_n \in \mathbb{R}$, it holds that

$$\sum_{i,j=1}^{n} c_i c_j K(x_i, x_j) \geq 0, \quad \forall x_1, \cdots, x_n \in \mathcal{X}.$$

Remark 21. Caveat: Sometimes[2], the above is called a *positive semidefinite kernel*, because it allows the sum in the definition to be 0 even for non-null c_i's. In the Machine Learning literature, "positive definite kernel" is understood as defined in Definition 6.1.2 and this is the convention we will use in this manuscript.

Note there's no restrictions on $\mathcal{X}$ (except it's non-empty). The property that defines pd kernels can be rephrased as: for all $n \in \mathbb{N}$ and $x_1, \cdots, x_n \in \mathcal{X}$, the matrix $\underline{K}$ that is given by

$$\underline{K}_{i,j} = K(x_i, x_j) \quad i, j = 1..n$$

is (symmetric) positive semidefinite. This matrix is called *Gram matrix.*

Exercise: Show that it is equivalent for a symmetric matrix to be positive semidefinite and to have all its eigenvalues non-negative.

6.2 Reproducible kernel Hilbert spaces

We introduced the two notions of inner products and pd kernels in the previous section. The link between them may not seem straightforward. However, as mentioned in the previous section, pd kernels share some similarities with pd symmetric matrices, which satisfy the following:

- if $A \in \mathbb{R}^{n \times n}$ is a pd symmetric matrix, then one can check that the function $b : (x, x') \mapsto x^T A x'$, $x, x' \in \mathbb{R}^n$, defines an inner product.

[1]In this manuscript, it is implicitly assumed that a kernel is symmetric.

[2]In Probability Theory

Viewing pd kernels as the infinite-dimensional analogue of pd symmetric matrices, one can hope that a pd kernel is always linked to some inner product. It is indeed the case but before we see it, we need to introduce some concepts.

Let $\mathcal{H}$ be a Hilbert space of functions from $\mathcal{X}$ to $\mathbb{R}$.

$\star$ **Remark 10.** We are given a Hilbert space $\mathcal{H}$ of real functions on $\mathcal{X}$. Suppose that for all $x \in \mathcal{X}$, the functional $L_x : \mathcal{H} \to \mathbb{R}$ defined by $L_x(f) = f(x)$, is a bounded operator on $\mathcal{H}$, i.e.

$$\forall x \in \mathcal{X},\ \exists M_x > 0 \text{ s.t. } \forall f \in \mathcal{H} :\ |f(x)| \leq M_x ||f||_{\mathcal{H}}. \tag{6.1}$$

Then, using the Riesz representation theorem, there exists a unique $K_x \in \mathcal{H}$ such that $L_x(f) = \langle f, K_x \rangle_{\mathcal{H}}$, so that one can define the kernel $K : (x, x') := \langle K_x, K_{x'} \rangle_{\mathcal{H}}$ on $\mathcal{X} \times \mathcal{X}$ such that $\mathcal{H}$ is a RKHS.

Definition 6.2.1. We say that a kernel K on $\mathcal{H}$ satisfies the *reproducing property* if and only if for all $x \in \mathcal{X}$ and all $f \in \mathcal{H}$, it holds that

$$\langle f, K_x \rangle_{\mathcal{H}} = f(x),$$

where $K_x := K(x, \cdot) \in \mathcal{H}$. In this case, we say that $\mathcal{H}$ is a *reproducible kernel Hilbert space* (RKHS) with reproducing kernel K.

We note in particular that for any $x, y \in \mathcal{X}$,

$$K(x, y) = \langle K(\cdot, x), K(\cdot, y) \rangle_{\mathcal{H}}.$$

Example 6.2.1. **On the reproducing property:**

Let feature map $\phi : \mathbb{R}^2 \to \mathcal{H}$, where $\mathcal{H}$ is the space of functions from $\mathbb{R}^2 \to \mathbb{R}$, be defined for all $x \in \mathbb{R}^2$ by

$$\begin{aligned} \phi_x \in \mathcal{H} : y &\mapsto x_1 y_1 + x_2 y_2 + x_1 x_2 y_1 y_2, \\ &\mathbb{R}^2 \to \mathbb{R}. \end{aligned}$$

Define the kernel K on $\mathbb{R}^2 \times \mathbb{R}^2$ by

$$K(x, x') = \langle \phi_x, \phi'_x \rangle_{\mathcal{H}} := x_1 x'_1 + x_2 x'_2 + x_1 x_2 x'_1 x'_2$$

Fix $u \in \mathbb{R}^3$ and define $f_u : \mathbb{R}^2 \to \mathbb{R}$ by

$$f_u(x) = u_1 x_1 + u_2 x_2 + u_3 x_1 x_2.$$

Note that $f_u \in \mathcal{H}$, since we can write $f_u = \phi_{(u_3,1)} + \phi_{(u_1-u_3,0)} + \phi_{(0,u_2-1)}$, which is a linear combination of elements of $\mathcal{H}$. We thus have a RKHS $\mathcal{H}$ with feature map ϕ, and we can check that K enjoys the reproducing property: since $K_x = \phi_x$, we have that

$$\begin{aligned}\langle f_u, K_x \rangle_{\mathcal{H}} &= \left\langle \phi_{(u_3,1)}, \phi_x \right\rangle + \left\langle \phi_{(u_1-u_3,0)}, \phi_x \right\rangle + \left\langle \phi_{(0,u_2-1)}, \phi_x \right\rangle \\ &= f_u(x),\end{aligned}$$

as claimed.

Now we can ask ourselves, given a pd kernel K on $\mathcal{X} \times \mathcal{X}$, is there a RKHS of real functions on $\mathcal{X}$ associated with K? The answer is positive and is known as the *Moore-Aronszajn theorem*:

Theorem 13. ***Moore-Aronszajn theorem:*** *Let $K : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ be a positive definite kernel. There exists a unique RKHS $\mathcal{H} \subset \{f : \mathcal{X} \to \mathbb{R}\}$ with reproducing kernel K. In particular, there exists a mapping $\phi : \mathcal{X} \to \mathcal{H}$ such that for all $x, x' \in \mathcal{X}$,*

$$K(x, x') = \langle \phi(x), \phi(x') \rangle_{\mathcal{H}}.$$

Note that Theorem 13 shows that a pd kernel induces a *unique* RKHS and vice versa. However, for a given $RKHS$, the feature map ϕ is not unique.

In view of the above theorem, why do we introduce a kernel instead of computing $\phi(x)$, and then the inner product in $\mathcal{H}$? Because evaluating the kernel is computationally more tractable. Furthermore, we can define a kernel K without explicitly knowing what the space $\mathcal{H}$ is. Thanks to the above theorem, a pd kernel allows us to measure the similarity of two points $x, x' \in \mathcal{X}$ through the implicit inner product of $\phi(x)$ and $\phi(x')$ in a (unknown) RKHS. This is sometimes referred to as the *kernel trick* in machine learning: we send the input space to a higher dimensional space where the elements can better be compared.

Equipped with those theoretical tools, we now look at some examples where introducing a kernel can be useful.

Example 6.2.2. **Polynomial kernel:** For any constant $C > 0$, a polynomial kernel of degree $d \in \mathbb{R}$ is the kernel K defined over $\mathcal{X} \subset \mathbb{R}^N$ by:

$$\forall x, x' \in \mathcal{X}, K(x, x') = (x \cdot x' + c)^d$$

Let an input space be of dimension $N = 2$, a second degree polynomial $d = 2$ corresponds to the following inner product in dimension 6:

$$\forall x, x' \in \mathcal{X}, K(x, x') = (x_1 x_1' + x_2 x_2' + c)^2 = \begin{bmatrix} x_1^2 \\ x_2^2 \\ \sqrt{2} x_1 x_2 \\ \sqrt{2c} x_1 \\ \sqrt{2c} x_2 \\ c \end{bmatrix} \cdot \begin{bmatrix} x_1'^2 \\ x_2'^2 \\ \sqrt{2} x_1' x_2' \\ \sqrt{2c} x_1' \\ \sqrt{2c} x_2' \\ c \end{bmatrix}$$

The features corresponding to a second-degree polynomial are the original features (x_1, x_2) as well as products of these features, and the constant feature.[3]

Example 6.2.3. **Gaussian kernel:** For any constant $\sigma > 0$, a Gaussian kernel or radial basis function (RBF) is the kernel K defined over $\mathcal{X} \subset \mathbb{R}^N$ by:

$$\forall x, x' \in \mathcal{X}, K(x, x') = \exp\left(-\frac{||x' - x'||^2}{2\sigma^2}\right).$$

What mapping ϕ would lead to this kernel? Let us consider a simplification, where $\sigma = 1$ and let the original instance space be $\mathbb{R}$ and consider the mapping $\phi(x) := \left\langle (\frac{1}{\sqrt{n!}} e^{-\frac{x^2}{2}} x^n)_{n \geq 0}, \cdot \right\rangle$. Then, $K_x(\cdot) := \left\langle (\frac{1}{\sqrt{n!}} e^{-\frac{x^2}{2}} x^n)_{n \geq 0}, \cdot \right\rangle$.

[3]A cool visualisation using Kernel SVM, with a polynomial kernel: `https://www.youtube.com/watch?v=OdlNM96sHio&t=0s`

$$\begin{aligned}
\langle K_x, K_{x'} \rangle &= \sum_{n=0}^{\infty} \left(\frac{1}{\sqrt{n!}} e^{-\frac{x^2}{2}} x^n \right) \left(\frac{1}{\sqrt{n!}} e^{-\frac{x'^2}{2}} x'^n \right) \\
&= e^{-\frac{x^2+x'^2}{2}} \sum_{n=0}^{\infty} \left(\frac{(xx')^n}{n!} \right) \\
&= e^{-\frac{(x-x')^2}{2}} = e^{-\frac{||x-x'||^2}{2}}.
\end{aligned}$$

Intuitively, the Gaussian kernel sets the inner product in the feature space between x, x' to be close to zero if the instances are far away from each other (in the original domain), and close to 1 if they are close.

Gaussian kernels are among the most frequently used kernels in applications.

Example 6.2.4. **Kernelised SVM:** Recall that using the Lagrange multipliers to solve a linearly separable classification task with SVM, the solution (w_*, b_*) has the following form

$$f_{w_*}(x) = sign(w_*^T x + b_*) = sign\left(\sum_{i=1}^{m} \alpha_i y_i ((x_i)^T x) + b_* \right).$$

The "kernelised" SVM (an example can be seen in figure 9.4) yields:

$$f_{w_*}(x) = sign\left(\sum_{i=1}^{m} \alpha_i y_i K(x_i, x) + b_* \right)$$

We simply replace the dot product $(x_i)^T x$ by $K(x_i, x)$, i.e. we compare the features through K, or equivalently, we compare the features in some implicit higher dimensional space where the similarity is measured through an inner product.

See figure 9.4 for an example of Kernel SVM using a Gaussian Kernel (radial basis functions RBF), and again the following video `https://www.youtube.com/watch?v=OdlNM96sHio&t=0s` for an example of the SVM using a polynomial kernel.

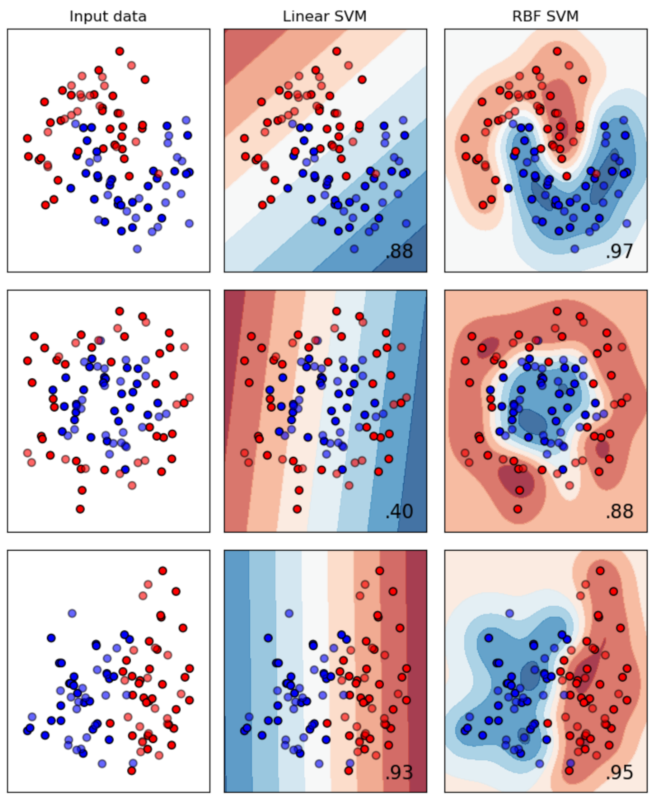


Figure 6.2: Linear SVM and SVM using a RBF kernel. **Source:** `https://scikit-learn.org/stable/auto_examples/classification/plot_classifier_comparison.html`

Example 6.2.5. Consider the function $K : \mathbb{R}^2 \times \mathbb{R}^2 \to \mathbb{R}$ given as:

$$K(x, y) = x_1 y_1 + x_2 y_2.$$

Is this a valid kernel? I.e. Is it a symmetric, positive definite kernel?

1. Check symmetry
2. Consider for any $n \in \mathbb{N}$, the set of points $x_1, ..., x_n \in \mathbb{R}^2$. Verify that

the matrix $\underline{K}$, given by

$$\underline{K}_{i,j} = K(x_i, x_j) \quad i, j = 1, \cdots, n,$$

is symmetric positive semidefinite.

Now that we got some intuition for what kernels are and how they work, let us prove the main theorem (Moore-Aronszajn Theorem) of this section.

Proof. (Proof of theorem 13) *Sketch of proof:*

We are given a pd kernel K and we want to construct the RKHS $\mathcal{H}$. We build $\mathcal{H}$ up in the following way:

- Let $G_1 := \{K_x : x \in \mathcal{X}\}$, where we recall that $K_x = K(x, \cdot)$.
- Let G_2 be the set of finite linear combinations of elements of G_1, i.e.

  $$G_2 := \left\{ \sum_{i=1}^{r} \alpha_i K_{x_i} : \; r \in \mathbb{N}, \alpha_i \in \mathbb{R}, x_i \in \mathcal{X}, \forall i = 1..r \right\}$$

  One can check that G_2 is a vector space.
- We define an inner product in G_2 as follows: for all $x, y \in \mathcal{X}$, define $b : G_1 \times G_1$ by $b(K_x, K_y) := K(x, y)$.

  Then, for all $f, g \in G_2$, there exist $r_f, r_g \in \mathbb{N}$, $\alpha_i, \beta_j \in \mathbb{R}$ and $x_i, y_j \in \mathcal{X}$ for all $i = 1..r_f$, $j = 1..r_g$ such that we can write f, g as:

  $$f = \sum_{i=1}^{r_f} \alpha_i K_{x_i}, \qquad g = \sum_{i=1}^{r_g} \beta_i K_{y_i}.$$

  We then extend b on $G_2 \times G_2$ as

  $$b(f, g) = \sum_{i=1}^{r_f} \sum_{j=1}^{r_g} \alpha_i \beta_j K(x_i, y_j).$$

  We readily see by construction that b is symmetric and bilinear.

Moreover, since K is a pd kernel, it holds that $b(f,f) \geq 0$ for all $f \in G_2$. To show that b is positive definite, it remains to show that $b(f,f) > 0$ when $f \neq 0$.
Exercise: *show that* $|b(f,K_x)| \leq \sqrt{b(f,f)K(x,x)}$, $x \in \mathcal{X}$, $f \in G_2$ *(Cauchy-Schwarz Inequality). Hint: look at* $b(f+K_x, f+K_x)$ *and* $b(f-K_x, f-K_x)$ *then use that for all* $c, d \in \mathbb{R}$, $c^2 + d^2 \geq 2|cd|$
We choose $x \in \mathcal{X}$ such that $f(x) \neq 0$ and we use the result of the above exercise: $0 < f(x)^2 = b(f,K_x)^2 \leq b(f,f)K(x,x)$. We thus have that b is positive definite, which entails that we b is an inner product on G_2. We then define $\langle \cdot, \cdot \rangle_{G_2} := b(\cdot,\cdot)$ and $||f||_{G_2} := \sqrt{\langle f, f \rangle}$.

- There is a last property that G_2 lacks to be a Hilbert space: it is not complete. One can define the space

$$\mathcal{H} := \{\lim_{n\to\infty} f_n;\ (f_n)_{n\geq 1} \text{ Cauchy sequence in } (G_2, ||\cdot||_{G_2})\}$$

where the limit is the pointwise limit. For $f, g \in \mathcal{H}$, it is possible to define $\langle f, g\rangle_{\mathcal{H}} := \lim_{n\to\infty} \langle f_n, g_n\rangle_{G_2}$ that makes $\mathcal{H}$ a Hilbert space.

□

⋆ **Remark 11.** The proof of this last point turns out to be quite technical and is beyond the scope of this notes. For our purpose, the intuition given by the above sketch should be enough. For the curious and motivated reader, we give a rough plan of how to complete the proof of the last point:

(i) Show that $(f_n)_{n\geq 1}$ $||\cdot||_{G_2}$-Cauchy sequence $\Rightarrow$ $f_n(x)$ Cauchy sequence in $\mathbb{R}$ for all $x \in \mathbb{R}$ (use reproducing kernel then Cauchy-Schwarz inequality). In particular, $(f_n)_{n\geq 1}$ converges pointwise.

(ii) Show that if a Cauchy sequence $f_n \to 0$ pointwise as $n \to \infty$ then $||f_n||_{G_2} \to 0$, (fix $N \in \mathbb{N}$ large enough, write $\langle f_n, f_n\rangle_{G_2} = \langle f_n - f_N, f_n\rangle_{G_2} + \langle f_N, f_n\rangle_{G_2}$ and bound the two terms).

(iii) Show that for two Cauchy sequences $(f_n)_{n\geq 1}, (g_n)_{n\geq 1}$ in G_2, $\left(\langle f_n, g_n\rangle_{G_2}\right)_{n\geq 1}$ is a Cauchy sequence in $\mathbb{R}$. (Use the Cauchy-Schwarz inequality)

(iv) For $f, g \in \mathcal{H}$, define $\langle f, g\rangle_{\mathcal{H}} := \lim_{n\to\infty} \langle f_n, g_n\rangle_{G_2}$ and show using (ii) that it does not depend on the choice of the Cauchy sequences $(f_n)_{n\geq 1}, (g_n)_{n\geq 1}$ that converge pointwisely to f and g.

(v) Show that $\langle \cdot, \cdot \rangle_{\mathcal{H}}$ is indeed an inner product.

(vi) Show that G_2 is dense in $\mathcal{H}$.

(vii) Show that $\mathcal{H}$ is complete: take a Cauchy sequence $(f_n)_{n\geq 1}$ $\underline{\text{in } \mathcal{H}}$ and use (vi) to define a sequence $(g_n)_{n\geq 1}$ $\underline{\text{in } G_2}$ such that $\lim_{n\to\infty} ||f_n - g_n||_{\mathcal{H}} = 0$; check that $(g_n)_{n\geq 1}$ is a Cauchy sequence in G_2 that pointwisely converges to a function $g \in \mathcal{H}$ and show that f_n converges to g in $\mathcal{H}$.

(viii) Check that K is the reproducing kernel of $\mathcal{H}$.

6.3 Mercer's Theorem

In this section, we will state results on pd kernels without proofs. We will see that a kernel can be represented as a sum of its *eigenfunctions*, similar to the eigendecomposition of a symmetric matrix. Thanks to this representation, the inner product in the associated RKHS can be seen as an inner product in $L^2(\mu)$, the set of square integrable functions against some measure μ with compact support in $\mathcal{X}$. We will then use this representation to see how the inner product in the RKHS corresponds, for some specific examples, to a dot product in $\mathbb{R}^n$.

Definition 6.3.1. Let μ be a finite measure on a compact subset $B \subset \mathcal{X}$. The integral operator $T_K : L^2(\mu) \to L^2(\mu)$ induced by a pd kernel K and the measure μ is defined by

$$\begin{aligned} T_K f : \mathcal{X} &\to \mathbb{R}, \\ x &\mapsto T_K f(x) := \int_{\mathcal{X}} K(x, x') f(x') \mu(\mathrm{d}x'). \end{aligned}$$

We say that $e : \mathcal{X} \to \mathbb{R}$ is an *eigenfunction* of T_K with *eigenvalue* λ if and only if $T_K e = \lambda e$.

Pd kernels can be seen as infinite-dimensional generalization of positive definite matrices. We admit the well known following fact:

A real matrix $M \in \mathbb{R}^{n\times n}$ is semi-positive definite with rank $k \leq n$ if and only if $M = B^T B$ for some matrix $B^{k\times n}$ of rank k, and if M is positive definite, then $k = n$. In particular, the columns $b_1, \dots, b_n \in \mathbb{R}^k$ of B are such that $M_{i,j} = \langle b_i, b_j\rangle$, where the inner product denotes the dot product.

It turns out that we can decompose the kernel K using the eigenfunctions of the operator T_K, and get the analogue of the above fact for pd kernels. For a measure μ on a set B, let $L^2(B,\mu) := \left\{f : B \to \mathbb{R} : \int_B f(x)^2\mu(\mathrm{d}x) < \infty\right\}$.

Theorem 14 (Mercer's Theorem)**.** *Let K be a continuous pd kernel and μ be a finite measure supported on a compact subset $B \subset \mathcal{X}$. There exists an orthonormal basis $(e_i)_{i\geq 1}$ of $L^2(B,\mu)$ consisting of eigenfunctions of T_K with non-negative eigenvalues $(\lambda_i)_{i\geq 1}$. Furthermore, for all $i \geq 1$, if $\lambda_i > 0$, then e_i is continuous and for all $x, x' \in B$, it holds that*

$$K(x,x') = \sum_{i\geq 1} \lambda_i e_i(x) e_i(x'),$$

where the series converges uniformly on B.

Suppose that T_K has finitely many non-zero eigenvalues, say $n \in \mathbb{N}$. Then with Mercer's Theorem, we can write K as a dot product:

$$K(x,x') = \sum_{i=1}^{n} \lambda_i e_i(x) e_i(x') = \langle e_\lambda(x), e_\lambda(x')\rangle,$$

where we defined $e_\lambda(x) := (\sqrt{\lambda_1}e_1(x), \dots, \sqrt{\lambda_n}e_n(x)) \in \mathbb{R}^n$. Now by Theorem 13, this means that

$$\langle \phi(x), \phi(x')\rangle_{\mathcal{H}} = \langle e_\lambda(x), e_\lambda(x')\rangle,$$

that is, the inner product in the RKHS $\mathcal{H}$ actually corresponds to a dot product in $\mathbb{R}^n$!

Let's come back to Example 6.2.1 to illustrate what this means.

Exercise.

Let $K(x,x') := x_1x_1' + x_2x_2' + x_1x_2x_1'x_2'$. Let $B := [-c,c]^2 \subset \mathbb{R}^2$ for a positive real number c and let $\mu(\mathrm{d}x) := \mathbb{1}_B(x)\mathrm{d}x$

(i) Write the induced integral operator T_K.

(ii) Show that $e_1 : \mathbb{R}^2 \to \mathbb{R}, x \mapsto x_1$ is an eigenfunction of T_K and find its eigenvalue.

(iii) Find all the other eigenfunctions with nonzero eigenvalues.

(iv) Write an orthonormal basis of $L^2(B, \mu)$.

(v) Deduce the expression of $K(x, x')$ as a dot product in $\mathbb{R}^n$ for some $n \in \mathbb{N}$.

Example 6.3.1. In Example 6.2.1, we started from a feature map ϕ_x to define the kernel $K(x, x') = x_1x_1' + x_2x_2' + x_1x_2x_1'x_2'$. Visually, we recognized the dot product in $\mathbb{R}^3$ of the vectors $(x_1, x_2, x_1x_2)^T$ and $(x_1', x_2', x_1'x_2')^T$. We now show that this is exactly what can be derived from Mercer's Theorem.

Let $B := [-c, c]^2 \subset \mathbb{R}^2$ with $c > 0$ arbitrary. Let μ be the Lebesgue measure on B. The associated integral operator reads as

$$T_K f(x) = \int_{[-c,c]^2} (x_1x_1' + x_2x_2' + x_1x_2x_1'x_2') f(x')\mathrm{d}x'.$$

We look for the eigenfunctions of T_K:

$$\begin{aligned} & T_K f(x) = \lambda f(x), \quad \in B \\ \Leftrightarrow \quad & a_1x_1 + a_2x_2 + a_3x_1x_2 = \lambda f(x), \end{aligned} \tag{6.2}$$

where

$$\begin{aligned} a_1 &:= \int_{[-c,c]^2} x_1' f(x')\mathrm{d}x', \\ a_2 &:= \int_{[-c,c]^2} x_2' f(x')\mathrm{d}x', \\ a_3 &:= \int_{[-c,c]^2} x_1'x_2' f(x')\mathrm{d}x'. \end{aligned}$$

One can check that $e_1(x) = \frac{x_1}{\lambda_1^{1/2}}$, $e_2(x) = \frac{x_2}{\lambda_2^{1/2}}$ and $e_3(x) = \frac{x_1x_2}{\lambda_3^{1/2}}$ are eigenfunctions with respective eigenvalues $\lambda_1 = \lambda_2 = \frac{2}{3}c^3$ and $\lambda_3 = \frac{4}{9}c^6$ Let us do it only for the first eigenfunction/eigenvector: we plug e_1 in the left-hand side of (6.2) and we get

$$\frac{1}{\lambda_1^{1/2}}\left(x_1\int_{[-c,c]^2}(x'_1)^2\mathrm{d}x' + x_2\int_{[-c,c]^2}x'_1x'_2\mathrm{d}x' + x_1x_2\int_{[-c,c]^2}(x'_1)^2x'_2\mathrm{d}x'\right)$$

$$e_1(x)\left[\frac{(x'_1)^3}{3}\right]_{-c}^{c} + 0 + 0$$

$$\lambda_1 e_1(x),$$

as claimed.

The reader can also check that e_1, e_2 and e_3 are orthonormal and it is clear from (6.2) that T_K has no other eigenfunction with non-zero eigenvalue (that is not a combination of these three).

Let $e_\lambda(x) := (\lambda_1^{1/2}e_1(x), \lambda_2^{1/2}e_2(x), \lambda_3^{1/2}e_3(x))^T$, then for any $x, x' \in B$, we thus see that

$$K(x,x') = \langle \phi_x, \phi_{x'}\rangle_{\mathcal{H}} = \langle e_\lambda(x), e_\lambda(x')\rangle\,.$$

Conclusion: For a simple kernel, we used Mercer's Theorem to express it as a dot product in $\mathbb{R}^3$ instead of an abstract inner product in the RKHS of its canonical feature maps $\phi_x, \phi_{x'}$. One thus speak of *non-canonical feature maps* $\varphi(x) = (x_1, x_2, x_1x_2)^T$ which define a Hilbert space that is not the RKHS, but for which the inner product is equivalent.

Note that we made an arbitrary choice for the compact set $B = [-c,c]^2$ and the finite measure $\mu(\mathrm{d}x) = \mathbb{1}_B(x)\mathrm{d}x$. Changing c does not change the eigenfunctions (inside the smallest B), but does change the eigenvalues. Choosing a different shape than a square for B would more profoundly change the operator and then the eigenfunctions, which would lead to other non-canonical feature maps and another Hilbert space, but with, again, an equivalent inner space.

6.4 Representer theorem

The representer theorem plays an large role in a large class of learning problems. It provides the means to reduce a infinite dimensional optimization problem to tractable finite dimensional one.

Theorem 15. *Let $\mathcal{X}$ be a set. K is a positive definite kernel on $\mathcal{X}$ and $\mathcal{H}$ is its corresponding RKHS. Furthermore, let $S = \{(x_1, y_1), ..., (x_m, y_m)\}$ be a finite set of points in $\mathcal{X} \times \mathcal{Y}$.*

Let us consider $h \in \mathcal{X}$ as a candidate model, $\lambda > 0$ and ℓ an arbitrary loss function. Then, the solution to the optimisation problem:

$$\min_{h \in \mathcal{H}} \sum_{i=1}^{m} \ell(h(x_i), y_i) + \lambda ||h||_{\mathcal{H}}^2$$

admits a representation of the form:

$$\forall x \in \mathcal{X}, f_w(x) = \sum_{i=1}^{m} w_i K(x_i, x) = \sum_{i=1}^{m} w_i K_{x_i}(x).$$

Proof. Let $\mathcal{H}_s$ be the sub-space spanned by the training data:

$$\mathcal{H}_s = \{f_w \in \mathcal{H} : f_w(x) = \sum_{i=1}^{m} w_i K_{x_i}(x), (w_1, \cdots, w_m) \in \mathbb{R}^m\}$$

$\mathcal{H}_s$ is a finite dimensional subspace of $\mathcal{H}$.

Then, $\exists \mathcal{H}_s^{\perp} = \{u \in \mathcal{H} : \langle u, v \rangle = 0, \forall v \in \mathcal{H}_s\}$, such that

$$\forall f_w \in \mathcal{H}, f_w = f_{w,s} + f_w^{\perp} \text{ (orthogonal decomposition)}$$

The part which is orthogonal has a property which is:

$$\forall i = 1, \cdots, m, \quad f_w^{\perp}(x_i) = \left\langle f_w^{\perp}, K_{x_i} \right\rangle = 0$$

by the reproducing property of RKHS (because $K_{x_i} \in \mathcal{H}_s$).

Therefore,

$$\forall i = 1, \cdots, m \quad f_w(x_i) = f_{w,s}(x_i)$$

i.e the orthogonal part does not influence f_w at points x_i.

Lastly, we must show that for a minimum f_w, $f_w^{\perp}$ does not enter the last term of the objective function, given by $\lambda ||f_w||^2_{\mathcal{H}}$. The last term is given by the norm of f_w in $\mathcal{H}$.

$$||f_w||^2_{\mathcal{H}} = ||f_{w,s}||^2_{\mathcal{H}} + ||f_w^{\perp}||^2_{\mathcal{H}}.$$

As the objective function is strictly increasing in the last variable (the norm), then

$$||f_w^{\perp}||^2_{\mathcal{H}} = 0$$

minimizes the objective function. Thus, minimum $f_w \in \mathcal{H}_s$. $\square$

6.5 Kernel (ridge) regression

In chapter 5, we studied linear models and obtain the analytical solution of linear regression in Theorem 7 and that of linear ridge regression in Theorem 9, when considering the squared error loss. In the current chapter, we saw how kernel methods can express non-linear relationships between inputs and outputs as linear combinations of elements of a RKHS, see in particular Theorem 15.

Question: Can we derive an analytical solution with kernel methods?

It turns out that by considering the squared error loss, the answer is yes. We define respectively the objective function and the regularised objective function for all $h \in \mathcal{H}$ as

$$L(h) := \frac{1}{m} \sum_{i=1}^{m} (h(x_i) - y_i)^2,$$
$$L_{\mathrm{r}}(h) := L(h) + \mathcal{R}(h),$$

where $\mathcal{R}(h) := \lambda||h||^2_{\mathcal{H}}$ for some $\lambda > 0$. The kernel regression and kernel ridge regression problems then read as

$$\min_{h\in\mathcal{H}} L(h), \tag{6.3}$$

$$\text{and } \min_{h\in\mathcal{H}} L_{\mathrm{r}}(h). \tag{6.4}$$

Theorem 16. *Let $S = \{(x_i, y_i) \in \mathcal{X} \times \mathcal{Y}; i = 1..m\}$ and recall that $\underline{K} = (K(x_i, x_j))_{1\leq i,j\leq m}$ denotes the Gram matrix of the dataset. Let $K(x, X) = (K(x, x_1), \ldots, K(x, x_m))$.*

(i) If $\underline{K}$ is invertible, then a solution to (6.3) *is h_* given by*

$$h_*(x) = K(x, X)\underline{K}^{-1}Y.$$

(ii) If $-m\lambda$ is not an eigenvalue of $\underline{K}$, then (6.4) *is minimised at h_* given by*

$$h_*(x) = K(x, X)\left(\underline{K} + m\lambda I_m\right)^{-1} Y,$$

where I_m is the $m \times m$ identity matrix.

Chapter 7

Gaussian processes

Contents

So far, we have considered models which have a clear functional structure, meaning, we consider a class of functions (for example, linear functions). Another approach to tackle the learning problem (both in regression and classification), is to give a prior probability to every possible function, where higher probabilities are given to functions that we consider to be more likely, for example, because they are smoother than other functions.

The first mentioned approach has an obvious problem, in that we have to decide upon the richness of the class of functions considered; if we are using a model based on a certain class of functions (e.g. linear functions) and the target function is not well modelled by this class, then the predictions will be poor. We can increase the flexibility of the class of functions, but this runs into the danger of overfitting, where we can obtain a good fit to the training data, but perform badly when making test predictions.

The second approach appears to have a serious problem, in that surely there are an uncountably infinite set of possible functions, and how are we going to compute with this set in finite time? A *Gaussian process* is a generalization of a Gaussian random variable. So far we have seen random variables which are scalars or vectors, similarly a stochastic process is a random function.

Example 7.0.1. Consider a 1-d regression problem. In 7.1(a) we show a number of functions drawn at random from the prior distribution over functions specified by a particular *Gaussian process*, which favours smooth functions. This prior is taken to represent our prior beliefs over the kinds of functions we expect to observe, before seeing any data.

Suppose now we see two datapoints (x_1, y_1) and (x_2, y_2). Then, we wish to consider only functions which pass by those two points. In 7.1(b), we see functions which are consistent with the observed data (dashed lines), and the solid line depicts the mean of all functions consistent with those observations. Notice how uncertainty is reduced close to the observations (this is because we have the prior that the functions are smooth).

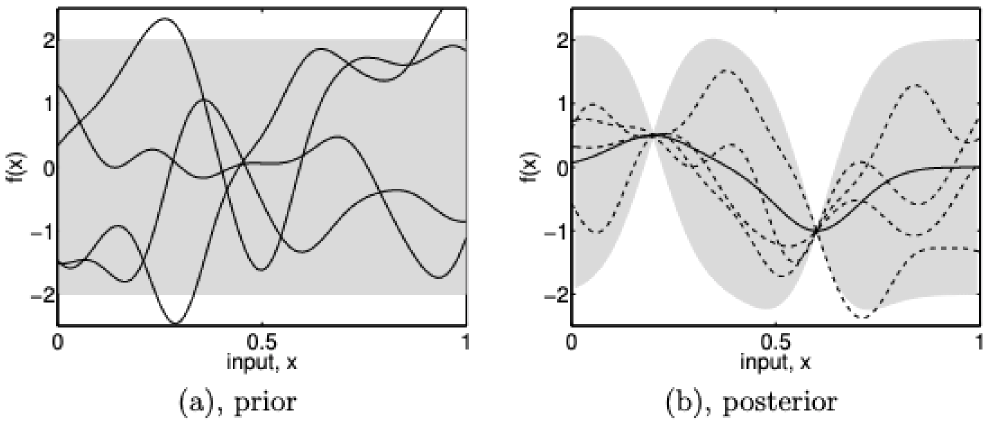


Figure 7.1: Panel (a) shows four samples drawn from the prior distribution. Panel (b) shows the situation after two datapoints have been observed. The mean prediction is shown as the solid line and four samples from the posterior are shown as dashed lines. In both plots, the shaded region denotes twice the standard deviation at each input value x.

7.1 Formal definition

Definition 7.1.1. Let $\mathcal{X}$ be a nonempty set, $K : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ be a positive definite kernel and $\mu : \mathcal{X} \to \mathbb{R}$ be any real-valued function. Then a random function $f : \mathcal{X} \to \mathbb{R}$ is said to be a Gaussian Process (GP) with mean function μ and covariance kernel K, denoted by $GP(\mu, K)$, if the following holds: For any finite set $X = (x_1, \cdots, x_m) \in \mathcal{X}$ of any size $m \in \mathbb{N}$, the random vector

$$f_X = (f(x_1), \cdots, f(x_m))^T \in \mathbb{R}^m$$

follows the multivariate normal distribution $\mathcal{N}(\mu_X, \underline{K}_{XX})$ with covariance matrix $\underline{K}_{XX} = (K(x_i, x_j))_{i,j=1}^m \in \mathbb{R}^{m \times m}$ and mean vector

$$\mu_X = (\mu(x_1), \cdots, \mu(x_m))^T \in \mathbb{R}^m.$$

Remark 22. The positive definite kernel K is equivalent to the covariance kernel/covariance function.

Remark 23. This definition implies that if f is a Gaussian process, then there exists a mean function $\mu : \mathcal{X} \to \mathbb{R}$ and a covariance kernel $K : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$. On the other hand, it is also true that for any positive definite kernel K and mean function μ, there exists a corresponding Gaussian process $f \sim GP(\mu, K)$. There exists a one-to-one correspondence between Gaussian processes $f \sim GP(\mu, K)$ and pairs (μ, K) of mean function μ and positive definite kernel K.

Remark 24. Since K is the covariance function of a Gaussian process, by definition it can be written as

$$K(x, x') = \mathbb{E}_{f \sim GP(\mu, K)} \left[(f(x) - \mu(x))(f(x') - \mu(x')) \right], \quad x, x' \in \mathcal{X},$$

where the expectation is with respect to the random function $f \sim GP(\mu, K)$.

Example 7.1.1. **(A concrete GP)** The most common choices for $\mu(x)$ and $K(x, x')$ are $\mu(x) = 0$ and $K(x, x') = \exp(-(x - x')^2/2)$.

7.1.1 Kernel (covariance functions)

The (covariance function) kernel is a crucial ingredient in a Gaussian process predictor, as it encodes our assumptions about the function which we wish to learn. From a slightly different viewpoint it is clear that in supervised learning the notion of similarity between data points is crucial; it is a basic similarity assumption that points with inputs x which are close are likely to have similar target values y, and thus training points that are near to a test point should be more informative about the prediction at that point. Under the Gaussian process view it is the covariance function that defines nearness or similarity.

What is the effect of choosing a kernel to define the $GP(\mu, K)$?

7.1.2 Squared exponential covariance function

The squared exponential (SE) covariance function has the form, for $r = x - x', \quad x, x' \in \mathcal{X}$:

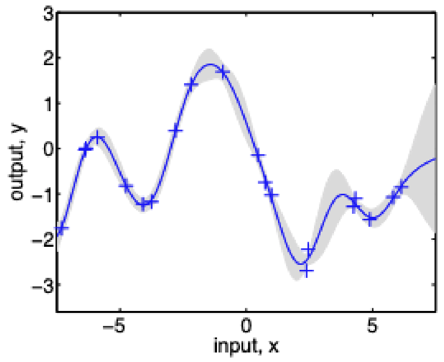
$$K_{SE}(r) = \exp(-\frac{r^2}{2\ell^2})$$

with parameter $\ell \in \mathbb{R}$ defining the *characteristic length-scale*, which modifies the behaviour of the Gaussian Process (see figure 7.2.

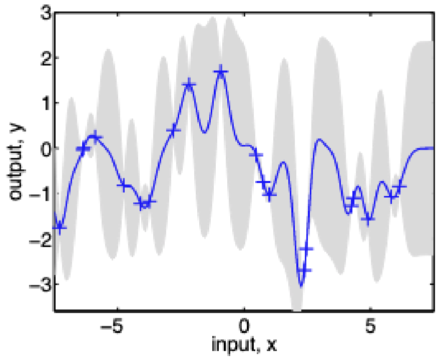
Definition 7.1.2. Mean square derivative: A random process X_t is mean-square differentiable at time t_0 if there is a random process X'_{t_0} such that

$$\lim_{t \to t_0} \mathbb{E}\left[\left(X'_{t_0} - \frac{X_t - X_0}{t - t_0}\right)^2\right] = 0.$$

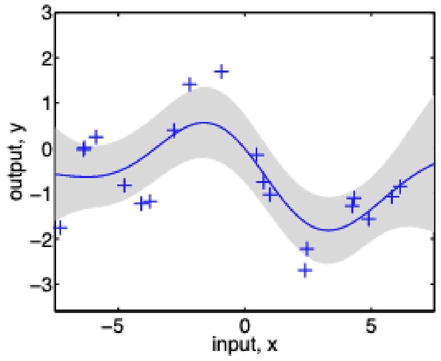
This covariance function is infinitely differentiable which means that the GP with this covariance function has Mean Square derivatives of all orders, and is thus very smooth. This can be seen as a disadvantage, however, the squared exponential is probably the most widely-used kernel within the kernel machines field.



(a), $\ell = 1$



(b), $\ell = 0.3$



(c), $\ell = 3$

Figure 7.2: Varying the hyperparameter ℓ regulates the influence of neighbouring points.

7.1.3 Matérn class of covariance functions

The Matérn class of covariance functions is given by:

$$K_{Matern}(r) = \frac{2^{1-\nu}}{\Gamma(\nu)} \left(\frac{\sqrt{2\nu} r}{\ell} \right)^{\nu} K_{\nu} \left(\frac{\sqrt{2\nu} r}{\ell} \right),$$

with positive parameters ν and ℓ, where K_ν is a modified Bessel function [**?**]. To see the influence of ν in the resulting GP, refer to figure 7.3. For the Matérn class the process $f(x)$ is k-times mean square differentiable if and only if $\nu > k$.

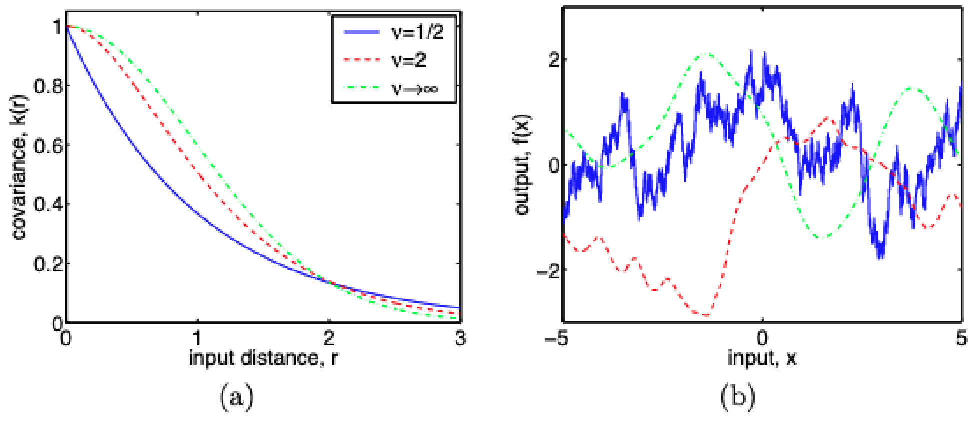


Figure 7.3: Varying the hyperparameter ν regulates regularity of the functions in the GP. Note if $\nu \to \infty$, we obtain the SE covariance function.

Both covariance functions obtained from the the SE and the Matérn kernels are so called *stationary covariance functions*, as they are a function of $x - x'$, and thus, invariant to translations in the input space. The covariance functions given above decay monotonically with r and are always positive. However, this is not a necessary condition for a covariance function [1]

7.1.4 Dot product covariance functions

The kernel

$$K(x, x') = \sigma_0^2 + x \cdot x',$$

constitutes a non-stationary covariance function.

7.2 Gaussian processes and kernel methods

In this short introduction to Gaussian processes, we have seen some familiar objects that were already introduced in the chapter about kernel methods.

[1] E.g. a valid covariance function can have the form of a damped oscillation.

Is this a coincidence, or is there some more obvious relation between GPs and kernel methods?

Recall that the ridge regression adds a regularisation penalty (scaled by λ) to the cost term:

$$\frac{1}{m}\sum_{i=1}^{m}(y_i - f(x_i))^2 + \lambda||f||_{\mathcal{H}}^2.$$

By Theorem 16, the solution for the optimisation above is [2]

$$f(x) = y^T(K_{XX} + m\lambda I)^{-1}k_{Xx},$$

with $(K_{XX})_{i,j} = k(x_i, x_j)$, $y = (y_1, ..., y_m)^T$ and $k_{Xx} = (k(x_1, x), \cdots, k(x_m, x))^T$ the vector of inner products between the data and the new point x.

We will see that a prediction using kernel ridge regression is equivalent to the mean prediction of a *GP regression*.

Recall that we discussed the Bayesian approach based on Theorem 3 for estimating an unknown probability distribution in Section 2.8.2: we fix a *prior distribution* encoding our beliefs, we observe data samples, we update our beliefs accordingly to obtain the *posterior distribution*.

The *Gaussian process regression* (also known as Kriging) is a Bayesian non parametric method for regression. Being a Bayesian approach, the GP-regression produces a posterior distribution of the unknown regression function f, provided by the training data (X, Y), a prior distribution Π_0 on f and a likelihood function denoted by $\ell_{X,Y}(f)$.

More specifically, the prior Π_0 is defined as a $GP(\mu, k)$ with mean function μ and covariance kernel k.

Note: since this GP serves as a prior, the mean function μ and the kernel k should be chosen so that they reflect one's prior knowledge or belief about the regression function f.

[2]You can verify by considering the Lagrangian.

The likelihood function is defined by a probabilistic model $p(y_i|f(x_i))$ for the noise variables $\xi_1, \cdots, \xi_m$, since this determines the distributions of the observations Y with the additive noise model:

$$y_i = f(x_i) + \xi_i, \quad i = 1, ..., m$$

It is common to assume ξ_i are i.i.d centered Gaussian random variables with variance $\sigma_2 > 0$.

$$\xi_i \sim \mathcal{N}(0, \sigma^2), \quad i = 1, ..., m.$$

Theorem 17. *Assume the following*

- $y_i = f(x_i) + \xi_i, \quad i = 1, ..., m$
- $\xi_i \sim \mathcal{N}(0, \sigma^2), \quad i = 1, ..., m.$
- $f \sim GP(\mu, k)$

and let $X = (x_1, ..., x_m) \in \mathcal{X}^n$ *and* $Y = (y_1, ..., y_m)^T \in \mathbb{R}^m$*. Then, we have*

$$f|Y \sim GP(\bar{\mu}, \bar{k}),$$

where $\bar{\mu} : \mathcal{X} \to \mathbb{R}$ *and* $\bar{k} : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ *are given by:*

$$\bar{\mu}(x) = \mu(x) + k_{xX}(K_{XX} + \sigma^2 I_m)^{-1}(Y - \mu_X) \quad x \in \mathcal{X} \tag{7.1}$$

$$\bar{k}(x, x') = k(x, x') - k_{xX}(K_{XX} + \sigma^2 I_n)^{-1} k_{Xx'}, \quad x, x' \in \mathcal{X} \tag{7.2}$$

$$\tag{7.3}$$

with $k_{Xx} = k_{xX}^T = (k(x_1, x), \cdots, k(x_m, x))^T$*, and* $(K_{XX})_{i,j} = k(x_i, x_j)$

Using the theorem above, the following equivalence holds for GP-regression and kernel ridge regression: We have $\bar{\mu} = f_{KRR}$ if $\sigma^2 = m\lambda$, where

1. $\bar{\mu}$ is the posterior mean function of GP-regression based on (X, Y), the GP prior $f \sim GP(0, k)$ and the modelling assumption where the noise is i.i.d. $\mathcal{N}(0, \sigma^2)$

2. f_{KRR} is the solution to kernel ridge regression based on (X, Y), the RKHS $\mathcal{H}$ and regularisation constant $\lambda > 0$.

Remark 25. One of the disadvantages of Gaussian processes is the fact that the covariance matrix size will scale with relation m^2 for dataset of size m, and furthermore, to invert the covariance matrix takes approximately $O(m^3)$ operations.[3]

[3] Current best asymptotic complexity is $O(m^2.376)$ [?].

Chapter 8

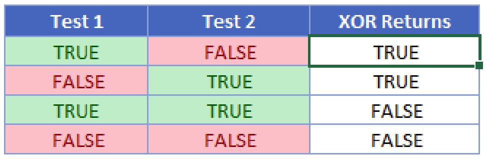
Deep learning

Contents

In the recent years, Machine Learning as become often identified as *Neural Networks* or *Deep Learning*, so we could not leave this class of models out of our exposition about Machine Learning.

One prototypical example often used to motivate the need of neural networks is the function XOR (Figure 8.1), which is a non-linear function that a *simple* neural network can approximate.

In this chapter, we will start with the simplest type of deep neural networks: the fully connected dense neural networks (section 8.1). We will see that some notions of learning theory that we've encountered will be challenged, and new mathematical theory is necessary to understand why neural networks seem to work so well in practice.



Test 1	Test 2	XOR Returns
TRUE	FALSE	TRUE
FALSE	TRUE	TRUE
TRUE	TRUE	FALSE
FALSE	FALSE	FALSE

Figure 8.1: XOR function

Furthermore, neural networks are (by far) the models presented in these lecture notes which require more care when being set-up and trained. You will see there are more hyper-parameters, engineering choices and ways to successfully/unsuccessfully train a neural network. We consolidate some practical advice, *tricks of the trade*, in section 8.6.

8.1 Fully connected dense neural networks

8.1.1 Definitions

Definition 8.1.1. Let us introduce the function (a unit):

$$a = \sigma(w^T x + b),$$

where $x \in \mathbb{R}^n$ are inputs, $w \in \mathbb{R}^n$ weights, $b \in \mathbb{R}$ the bias and $\sigma : \mathbb{R} \to \mathbb{R}$ an activation function.

We can write different models we have studied by considering different activation functions:

- linear regression: $\sigma(z) = z$
- binary classification: $\sigma(z) = sign(z)$
- logistic regression: $\sigma(z) = \frac{1}{1+e^{-z}}$

A neural network is a *combination* of a lot of these units. For example, a 1-layer neural network $f : \mathbb{R}^d \to \mathbb{R}^{d_1}$ is given by:

$$f(x) = \sigma_1(W_1 x + b_1)$$

for $x \in \mathbb{R}^d$, W_1 a matrix $\in \mathbb{R}^{d_1 \times d}$ and $b_1 \in \mathbb{R}^{d_1}$.

What is $f(x)$ doing? It's taking a vector x and applying a linear transformation to it (by W_1, b), this returns a vector of dimension d_1. σ is applied to this vector (typically component-wise).

A 2-layer neural network $f : \mathbb{R}^d \to \mathbb{R}^{d_2}$ is given as:

$$f(x) = \sigma_2(W_2 \sigma_1(W_1 x + b_1) + b_2),$$

where W_2 is a matrix $\in \mathbb{R}^{d_2 \times d_1}$ and $b_2 \in \mathbb{R}^{d_2}$.

And for a general L-layer neural network $f : \mathbb{R}^d \to \mathbb{R}^{d_L}$ we can write the definition:

Definition 8.1.2. A fully connected feedforward neural network is given by its **architecture**, namely, by hyper-parameters:

- $L \in \mathbb{N}$ the number of layers
- $\sigma_i : \mathbb{R} \to \mathbb{R}$, activation functions $(i = 1, ..., N)$
- $d_0, ..., d_L$, specifying the number of neurons in the input, output and l-th hidden layer.

and it has the form:

$$f(x) = \sigma_L \left(W_L \sigma_{L-1} \left(\cdots \left(W_2 \sigma_1 \left(W_1 x + b_1 \right) + b_2 \right) \cdots \right) + b_L \right),$$

where the σ_i's are applied component-wise, that is for $z \in \mathbb{R}^{d_i}$, we write $\sigma_i \begin{pmatrix} z_1 \\ \vdots \\ z_{d_i} \end{pmatrix} = \begin{pmatrix} \sigma_i(z_1) \\ \vdots \\ \sigma_i(z_{d_i}) \end{pmatrix}$. Alternatively, one can write recursively

$$\begin{aligned} x^{(0)} &= x, \\ x^{(k+1)} &= \sigma_{k+1}(W_{k+1} x^{(k)} + b_{k+1}), \\ \text{and} \quad f(x) &= x^{(L)}. \end{aligned}$$

A few properties of this definition are:

- One can both do classification and regression, depending on the activation function.
- Feature selection is "built-in": if σ_L is the identity, then the neural network output is $W_L x^{(L-1)} + b_L$, which is a linear combination of the penultimate layer's output.
- Number of degrees of freedom for a neural network is given by:

$$\text{dof} = \sum_{l=1}^{L} d_l d_{l-1} + d_l$$

Remark 26. There are some engineering choices: L, σ_i, d_i. This fixes the number of degrees of freedom we have to find for W_i, b_i.

Example 8.1.1. What if all σ are the identity functions?

$$\begin{aligned} & W_L(\cdots(W_1 x + b_1)\cdots) + b_L \\ =& (W_L \cdots W_1)x + (b_L + W_L b_{L-1} + \cdots W_L \cdots W_2 + b_1. \end{aligned}$$

This is just a linear predictor.

8.1.2 Loss functions

Similarly to other supervised learning models we have seen, neural networks are trained in the ERM / Regularised ERM framework.

When considering *regression type problems*, a typical loss function to be considered is the Mean Squared Error (or Mean absolute error). For a neural network f_w, we get

$$C(w) = L(f_w) = \frac{1}{2m} \sum_{i=1}^{m} ||f_w(x_i) - y_i||_2^2.$$

From the definition of fully-connected neural networks, we see that (as soon as there is a non-linear activation) the map $w \mapsto f_w$ is non-linear. As a consequence, even when the chosen loss L on the function's space is convex, the cost C on the parameter's space is not. In particular, for neural networks, there is no guarantee that local minima are global minima, which makes the success of training through gradient descent not obvious.

For a **classification task**, like previously, we do not want to optimise over non-differentiable functions (or at least, hard differentiable). So instead of the $0-1$ loss $\ell(y,y') = \mathbb{1}_{\{y\neq y'\}}$, we consider a surrogate loss, similar to what we did with the perceptron.

Consider a fully connected neural network $f_w(x) = W_L x^{(L-1)} + b_L$. To perform binary classification, one can choose the predictor to be $\text{sign}(f_w(x))$. A commonly used surrogate loss is the logistic loss, given by:

$$\ell(f_w(x), y) = \log\left(1 + \exp^{-yf_w(x)}\right).$$

Why? If $y = 1$ and $f_w(x) > 0$ (or $y = -1$ and $f_w(x) < 0$), the value $exp(-yf_w(x))$ is small, and then we have $\ell(f_w(x), y) \approx 0$. Otherwise, in case of misclassification, we have: $\log(1 + e^{-f_w(x)y}) > \log 2$.

Another common way to do binary classification is to choose $d_{L-1} = 1$ and $\sigma_L(z) = \text{sigmoid}(z) = \frac{1}{1+\exp(-z)}$, which returns a number in $[0,1]$, that can be interpreted as probability.

Consider the two classes 0 and 1. To predict the class of an input x, one can then choose 0 if $f_w(x) \leq 0.5$ and 1 otherwise. The loss function to minimize can be the cross-entropy loss defined by

$$L(f_w) = \frac{1}{m}\sum_{i=1}^{m}(y_i \log f_w(x) + (1-y_i)\log(1 - f_w(x))).$$

Note if $y = 0$, then we want $f_w(x)_{y_i}$ to be close to 0, and if $y = 1$, we want $f_w(x))$ to be close to 1. Note that a perfect classifier achieves 0 loss.

Multi-class classification (i.e. a datapoint x can have a label $0, ..., n-1$) can be easily performed by considering $d_{L-1} = n$ and $\sigma_L(z) = \text{softmax}$, which returns a probability on a finite set. For example, $d_{L-1} = n$ we get the output vector:

$$f_w(x) = \frac{1}{\sum_{i=1}^{n} \exp(x_i^{(L-1)})} \begin{pmatrix} \exp(x_1^{(L-1)}) \\ \cdots \\ \exp(x_n^{(L-1)}) \end{pmatrix}.$$

Then, the loss function is the categorical cross-entropy:

$$L(f_w) = \frac{1}{m} \sum_{i=1}^{m} \sum_{j=1}^{n} (y_{i,j} \log f_w(x_i)_j + (1 - y_{i,j}) \log(1 - f_w(x_i)_j)),$$

where (with a slight abuse of notation) we write, for the i-th datapoint, $y_i = (y_{i,1}, \cdots, y_{i,n})$. Observe that datapoint x_i can only belong to one class, so y_i is zero everywhere, except on the class that x_i belongs to. Again, a perfect classifier achieves 0 loss.

The non-convexity issues we talked about for regression are similar for classification. The softmax map and the cross-entropy loss can be generalised to tackle multi-class classification.

8.2 Back Propagation

8.2.1 Definition

You can note that through gradient based optimisation methods, we must find a way to update each parameter of the model. We want an efficient way to write the updates for each degree of freedom we have in the neural network.

The back-propagation algorithm can be thought of as a table-filling algorithm that takes advantage of storing intermediate results.

Example 8.2.1. Note that our cost function, for a generic L-layer neural network, is given as:

$$\begin{aligned} C(w) = L(f_w) &= \sum_{i=1}^{m} \ell(y_i, f_w(x_i)) \\ &= \sum_{i=1}^{m} \ell(y_i, \sigma_L\left(W_L \sigma_{L-1}\left(\cdots\left(W_2 \sigma_1\left(W_1 x_i + b_1\right) + b_2\right)\cdots\right) + b_L\right) \end{aligned}$$

For the weights w_{ij}^L in weights matrix W_L, for example, we want to compute $\partial_{w_{ij}^L} C(w)$. This is given, through the chain rule, by:

$$\partial_{w_{ij}^L} C(w) = \frac{\partial z_L}{\partial w_{ij}^L} \frac{\partial \sigma_L}{\partial z_L} \frac{\partial C(w)}{\partial \sigma_L}$$

with $z_L = W_L \sigma_{L-1}$, so $\frac{\partial z_L}{\partial w_{ij}^L} = \sigma_{L-1,j}$.

For the weight w_{ij}^{L-1}, we have

$$\partial_{w_{ij}^{L-1}} C(w) = \frac{\partial z_{L-1}}{\partial w_{ij}^{L-1}} \frac{\partial \sigma_{L-1}}{\partial z_{L-1}} \frac{\partial z_L}{\partial \sigma_{L-1}} \frac{\partial \sigma_L}{\partial z_L} \frac{\partial C(w)}{\partial \sigma_L}$$

with $z_{L-1} = W_{L-1} \sigma_{L-2}$.

Note that some terms of the derivative have already been computed when we wrote the update for w_{ij}^L.

We can write a neural network recursively, as:

$$\begin{aligned} Z_n &= W_n X_{n-1} \\ X_n &= \sigma_n(Z_n), \end{aligned}$$

where Z denotes the vector of z_i and W the matrix of weights w_{ij}. Furthermore, X_0 denotes the input vector.

Suppose we have computed $\frac{\partial C}{\partial X_{n,i}}$, for $i = 1, ..., d_n$ (the width of the layer n).

Then we can compute recursively the following gradients:

$$\begin{aligned}\frac{\partial C}{\partial z_{n,i}} &= \left.\frac{\partial \sigma_n(z)}{\partial z}\right|_{z=z_{n,i}} \frac{\partial C}{\partial x_{n,i}} \\ \frac{\partial C}{\partial w_{ij}^n} &= x_{n-1,j}\frac{\partial C}{\partial z_{n,i}} \\ \frac{\partial C}{\partial x_{n-1,j}} &= \sum_i w_{ij}^n \frac{\partial C}{\partial z_{n,i}}.\end{aligned}$$

This can also be written in matrix-vector notation, namely,

$$\begin{aligned}\frac{\partial C}{\partial Z_n} &= \left.\frac{\partial \sigma_n(z)}{\partial z}\right|_{z=Z_n} \circ \frac{\partial C}{\partial X_n} \\ \frac{\partial C}{\partial W_n} &= \left(\frac{\partial C}{\partial Z_n}\right)(X_{n-1})^T \\ \frac{\partial C}{\partial X_{n-1}} &= W_n^T \frac{\partial C}{\partial Z_n},\end{aligned}$$

note that now $\left.\frac{\partial \sigma_n(z)}{\partial z}\right|_{z=Z_n}$ produces a vector of the activation function evaluated at Z_n, and $\circ$ gives the Hadarmard product between two vectors. [1]

To make things more clear for the next exposition, we can write

$$\begin{aligned}\frac{\partial C}{\partial W_n} &= \left(\frac{\partial C}{\partial Z_n}\right)(X_{n-1})^T \\ &= \left(\left.\frac{\partial \sigma_n(z)}{\partial z}\right|_{z=Z_n} \circ \frac{\partial C}{\partial X_n}\right)(X_{n-1})^T \\ &= \left(\left.\frac{\partial \sigma_n(z)}{\partial z}\right|_{z=Z_n} \circ W_{n+1}^T \frac{\partial C}{\partial Z_{n+1}}\right)(X_{n-1})^T \\ &= \left(\left.\frac{\partial \sigma_n(z)}{\partial z}\right|_{z=Z_n} \circ W_{n+1}^T \left.\frac{\partial \sigma_{n+1}(z)}{\partial z}\right|_{z=Z_{n+1}} \circ W_{n+2}^T \ldots \left.\frac{\partial \sigma_L(z)}{\partial z}\right|_{z=Z_L} \circ \frac{\partial C}{\partial X_L}\right)(X_{n-1})^T.\end{aligned}$$

[1] $a = (a_1, ..., a_n), b = (b_1, ..., b_n)$ then $a \circ b = (a_1 b_1, ..., a_n b_n)$

We can call the term inside the parenthesis as

$$\delta^n := \frac{\partial \sigma_n(z)}{\partial z}\bigg|_{z=Z_n} \circ W_{n+1}^T \frac{\partial \sigma_{n+1}(z)}{\partial z}\bigg|_{z=Z_{n+1}} \circ W_{n+2}^T \cdots \frac{\partial \sigma_L(z)}{\partial z}\bigg|_{z=Z_L} \circ \frac{\partial C}{\partial X_L},$$

then the gradient reads:

$$\frac{\partial C}{\partial W_n} = \delta^n X_{n-1}^T. \tag{8.1}$$

We can compute δ^{n-1} recursively, by

$$\delta^L = \frac{\partial \sigma_L(z)}{\partial z}\bigg|_{z=Z_L} \circ \frac{\partial C}{\partial X_L}, \tag{8.2}$$

$$\delta^{n-1} = \frac{\partial \sigma_{n-1}(z)}{\partial z}\bigg|_{z=Z_{n-1}} \circ W_{n-1}^T \delta^n, \quad n = 1, ..., L. \tag{8.3}$$

8.2.2 Exploding and vanishing gradients

Neural networks are usually trained with gradient based algorithms, the gradient being computed with backpropagation. Since they are composition of functions, this may cause gradients to explode or vanish at early layers. Let's first look at the chain rule (for simplicity in the scalar case $w \in \mathbb{R}$) for the derivative of a map $f_L(w) = g_L(g_{L-1}(\dots g_1(w))\cdots)$: let $f_\ell(w) = g_\ell(g_{\ell-1}(\dots g_1(w))\cdots)$ for all $\ell = 1, \cdots, L$ and $f_0(w) \equiv w$. We have

$$\begin{aligned} f_L'(w) &= g_L'(f_{L-1}(w)) f_{L-1}'(w) \\ &= \cdots \\ &= \prod_{i=0}^{L-1} g_{L-i}'(f_{L-i-1}(w)). \end{aligned}$$

We see that the derivative of the composition of L maps is a product of L derivatives. In particular, if each of them is of order, say, $a \in (0, \infty)$, then $f_L'(w)$ is of order a^L. For large L, if $a < 1$, then we get a very small gradient, whereas if $a > 1$, it can grow very large. Gradients that are too small or too large hinder for training, for similar reasons as that of the learning rate we discussed in Section 3.5, see Figure 3.3. Loosely speaking, with small gradients, training gets stuck and takes too long to converge, with large gradients, training jumps over minima and gets away from good solutions.

Recall that the backpropagation update at layer $n \in \{1, \dots, L\}$ in a fully connected neural network is given in Equation (8.1) in terms of δ^n, which is recursively defined from the last layer to the previous ones in (8.2). In particular, the further we go backwards, the more terms in the product, which, as we said, can be the source of training instability.

Note that since the derivatives of the activation functions σ_ℓ's appear in the product (8.2), the phenomena of exploding and vanishing gradients are linked to the choices of these functions. For example, the sigmoid function $\sigma : x \mapsto (1 + e^{-x})^{-1}$ has a vanishing derivative away from some interval centered at 0. On the other hand, the derivative of a polynomial activation function of degree at least 2 explodes far enough from 0.

Can we even train then? Fortunately there are heuristics that lead to practical techniques allowing stable training. We mention two of them in the forthcoming section 8.6.2.

Specific weights initialisation schemes are based on heuristics aiming at stabilising the gradients.

8.2.3 Common initialization schemes

The way a neural network's weights are initialized prior to training has a crucial effect on the success of training. Indeed, suppose that weights are all initialized to be zero. Then, the updates for the weights are given as:

$$\frac{\partial C}{\partial W_n} = \delta^n X_{n-1}^T = 0$$

as δ^n yields a zero vector, meaning that the weights will not change during training.

More generally, if the weights are all initialized to a constant value c, then the update yields:

$$\begin{aligned}\frac{\partial C}{\partial W_n} &= \delta^n X_{n-1}^T = \delta^n \sigma_{n-1}(c\mathbf{1}^T X_{n-2}\mathbf{1})^T = \vec{\alpha}\vec{\beta}^T \\ &= c_n \mathbf{1}_{d_n \times d_{n-1}},\end{aligned}$$

where $\vec{\alpha}, \vec{\beta}$ are two constant vectors. In particular, all weights at the same layer n receive the same update $c_n \in \mathbb{R}$.

In general, it is best to initialize the weights at random (independent) values, thus avoiding this problem. Commonly used initialization schemes were born from heuristics to avoid the problems of exploding and vanishing gradients we discussed before.

Intuition for choosing the variance of the weights at initialization. Initialize the biases at 0. From (8.1) and (8.2), in order for the magnitude of the gradient to be of the same order across all layers, we want the activations X_n (or the preactivations Z_n) to have the same mean and the same variance at every layer. Let's see how to ensure it at initialization.

For simplicity, we assume:

- components of W_n are i.i.d. with mean 0 and variance a (with some unspecified distribution for now)
- activation function σ is tanh.

Let's write the variance of X_n in terms of that of X_{n-1}. We have

$$\begin{aligned}\mathrm{Var}\,[X_n] &= \mathrm{Var}\,[\sigma_n(W_n X_{n-1})] \\ &= \left(\mathrm{Var}\left[\sigma_n\left(\sum_{j=1}^{d_{n-1}} w_{ij}^n x_{n-1,j}\right)\right]\right)_{i=1,\dots,d_n}.\end{aligned}$$

Fix $i \in \{1, \dots, d_n\}$. By independence of $x_{n-1,j}$ and w_{ij}^n, the mean of $\sum_{j=1}^{d_{n-1}} w_{ij}^n x_{n-1,j}$ is 0, and if that sum is close enough to its mean, then we (roughly speaking) have the first order Taylor approximation

$$\sigma_n\left(\sum_{j=1}^{d_{n-1}} w_{ij}^n x_{n-1,j}\right) \approx \sigma_n'(0)\sum_{j=1}^{d_{n-1}} w_{ij}^n x_{n-1,j}.$$

This is, of course, very informal but we need not make a more formal claim for understanding the intuition. Now we get that

$$\begin{aligned}\operatorname{Var}\left[X_n\right] &\approx \left(\sigma_n'(0)^2 \operatorname{Var}\left[\sum_{j=1}^{d_{n-1}} w_{ij}^n x_{n-1,j}\right]\right)_{i=1,\dots,d_n} \\ &= \left(\sigma_n'(0)^2 \sum_{j=1}^{d_{n-1}} \operatorname{Var}\left[w_{ij}^n x_{n-1,j}\right]\right)_{i=1,\dots,d_n}, \qquad (8.4)\end{aligned}$$

where we used the **independence** of the $w_{ij}^n x_{n-1,j}$ for distinct j to take the sum out of the variance. Note that the w_{ij}^n's and the $x_{n-1,j}$'s are independent, since $x_{n-1,j}$ only depends on the inputs and the weights of the previous layers, which at initialization, are assumed to be independent from those of layer n. Note that for two independent random variables U, V, it holds that

$$\begin{aligned}\operatorname{Var}(UV) &= \mathbb{E}[U^2]\mathbb{E}[V^2] - \mathbb{E}[U]^2\mathbb{E}[V]^2 \\ &= \left(\mathbb{E}[U^2] - \mathbb{E}[U]^2\right)\left(\mathbb{E}[V^2] - \mathbb{E}[V]^2\right) + \mathbb{E}[U]^2\mathbb{E}[V^2] + \mathbb{E}[U^2]\mathbb{E}[V]^2 - 2\mathbb{E}[U]^2\mathbb{E}[V]^2 \\ &= \operatorname{Var}(U)\operatorname{Var}(V) + \operatorname{Var}(U)\mathbb{E}[V]^2 + \operatorname{Var}(V)\mathbb{E}[U]^2.\end{aligned}$$

If moreover U and V have mean 0, then $\operatorname{Var}(UV) = \operatorname{Var}(U)\operatorname{Var}(V)$. Assume that the inputs are centered, then $x_{n-1,j}$ is centered too. Hence, coming back to (8.4), we have "shown" that

$$\operatorname{Var}\left[X_n\right] \approx \sigma_n'(0)\operatorname{Var}\left[W_n\right]^T \operatorname{Var}\left[X_{n-1}\right].$$

The i-th component of the vector $\operatorname{Var}\left[W_n\right]^T \operatorname{Var}\left[X_{n-1}\right]$ is a sum of d_{n-1} elements $w_{ij}x_{n-1,j}$. One can choose $\operatorname{Var}[w_{ij}^n] = \frac{1}{d_{n-1}}$, such that provided that $\sigma_n'(0) = 1$ (e.g. for the tanh activation function), the variance of X_n is constant across layers.

Remark 27. This heuristic is valid only for the first step of gradient descent. After that, weights are, for example, no longer independent.

Remark 28. Similar derivation can be done for other activation functions, but some terms don't cancel out so nicely, and this will change the initialisation.

Some common initializations. The following is a list of commonly used initialization schemes. Biases are initialized at 0, all weights are mutually independent:

- Xavier: $w_{ij}^n \sim \mathrm{Unif}(0, 1/\sqrt{d_{n-1}})$. This is heuristically justified as above for the tanh activation function.
- Le Cun: $w_{ij}^n \sim \mathcal{N}(0, 1/d_{n-1})$.
- He: $w_{ij}^n \sim \mathcal{N}(0, 2/d_{n-1})$. It can be motivated similarly as above for the ReLU activation.

Other initialization schemes, such as Glorot initialization, scale the variance by $d_{n-1} + d_n$.

Active research provides insights on the effect of the variance at initialization on the training dynamics and generalization of neural networks; more on this in the forthcoming section 8.4.

8.3 Approximation Theorems

In this section we give a brief introduction to some of the approximation results using neural networks. This is a very active research area, but we will focus on one fundamental result which is often cited in talks/literature as one of the firsts of the kind. [5] has a good introduction, focusing also on the necessary tools from analysis to understand the proofs. This section is not examinable, it's a very brief introduction to *modern* advances in the theory of neural networks.

The theorem below, by Cybenko (1989) [3] is often called the universal approximation theorem for neural networks. We state a weaker version of the theorem proved in the linked paper [2].

[2] In the original paper, the theorem needs only σ to be a discriminatory function.

Theorem 18. *Let σ be the sigmoid function, that is $\sigma(x) = \frac{1}{1+e^{-x}}$. Then finite sums of the form*

$$G(x) = \sum_{j=1}^{N} \alpha_j \sigma(w_j^T x + b_j),$$

are dense in $C(I_n)$ with respect to the supremum norm, the space of continuous functions on I_n, where I_n denotes the n-dimensional unit cube $[0,1]^n$. Meaning, given any $f \in C(I_n)$ and $\epsilon > 0$, there is a sum, $G(x)$, of the above form, for which

$$|G(x) - f(x)| < \epsilon \quad \forall x \in I_n.$$

$\star$ **Remark 12.** The following proof requires some familiarity with functional analysis.

Proof. Assume without proof that the sigmoid has the following property (called discriminatory property): Let μ be a finite regular signed Borel measure, if

$$\int_{I_n} \sigma(w_j^T x + \theta) d\mu(x) = 0 \quad \forall (\vec{y}, \theta) \in \mathbb{R}^d \times \mathbb{R},$$

then $\mu = 0$.

Let $S = \{f(x) = \sum_j^N \alpha_j \sigma(w_j^T x + b_j) : N \in \mathbb{N}, \alpha_j, w_j, b_j \in \mathbb{R}, j = 1, ..., N\}$.

This is a linear subspace of $C(I_h)$. If closure of S, $\overline{S} = C(I_n)$, we are done.

Assume that $\overline{S} \neq C(I_n)$. By the Hahn-Banach theorem, there exists a bounded linear form $L \neq 0$ on $C(I_n)$ such that $L(\overline{S}) = 0$.

By the Riesz representation theorem, there exists a signed regular Borel measure μ that is nonzero (as $L \neq 0$) such that:

$$L(h) = \int_{I_n} h(x) d\mu(x) \quad \forall h \in C(I_n).$$

Taking $h \in S$, we have $L(h) = 0$ which implies $\mu = 0$ because of the discriminatory property, which yields a contradiction. $\square$

What the above theorem tells us is that neural networks can approximate any arbitrary continuous function, similar to the Stone-Weierstrass theorem for polynomials. It is worth noting that this fact is true for deeper networks as well: conditioning on the output of layer $L-2$ and considering it as the input layer, the layers $L-2, L-1$ and L can be seen as a two-layer neural network and one can directly apply the above theorem.

Similar result by Kurt Hornik (1991) [7]. The result above also extends to classification tasks. The same result also holds for more general sigmoidal functions and the ReLU function.[3]

8.4 * Infinitely wide neural networks

Neural networks are often used largely overparametrized. As a consequence, for a given architecture, there may be many neural networks that perfectly fit a given dataset. Yet, neural networks trained with (variations of) gradient descent sometimes seem to generalize well or at least, better than expected, for example when compared to overparametrised linear regression.

Constructing a general theory to explain their success seems out of reach. Indeed, we will see in this section that different training regimes occur by only changing the scale of the weights' initialization.

The content of this section concerns recent development on the theory of neural networks. It is far from exhaustive and its goal is to briefly present two successful approaches attempting to explain the surprising success of (overparametrized) neural networks.

8.4.1 Initialization scale

To simplify the notation, consider a neural network f_w with scalar input and ouput. Suppose moreover that it has a single hidden layer of width $d \in \mathbb{N}$ and with the identity activation function between the hidden and output layer,

[3]Similar type of statement as Weierstrass approximation theorem.

that is for an input $x \in \mathbb{R}$, the network predicts

$$f_w(x) := \frac{1}{d^\gamma} w_2 \sigma(w_1 x + b),$$

where $w_1, b \in \mathbb{R}^{d\times 1}, w_2 \in \mathbb{R}^{1\times d}$ are the learnable parameters of the network, and $\gamma > 0$ is a hyperparameter. We assume that the components of w_1, w_2 and b are i.i.d. standard Gaussian variables $\mathcal{N}(0,1)$ at initialization so that γ governs the initialization scale (recall that if $X \sim \mathcal{N}(0,1)$, then $aX \sim \mathcal{N}(0,a^2)$ for $a \in \mathbb{R}$).

Instead of the above equation, we will prefer to (equivalently) write the network as the sum of its hidden neurons, that is

$$f_w(x) := \frac{1}{d^\gamma} \sum_{k=1}^{d} w_{2,k} \sigma(w_{1,k} x + b_k), \tag{8.5}$$

In the Bayesian learning paradigm, the prior distribution of a model (initialization) is crucial as it is supposed to encode our prior beliefs and radically influences the learning (training). It is not clear that neural networks trained by gradient descent perform Bayesian learning (in fact, they do not, see the discussion at the end of Section 8.4.2). Nonetheless, Neal in [9] observed the following result for 2-layer neural networks (later generalized to deeper networks):

Theorem 19. *Suppose that f_w is a neural network at initialization defined as in (8.5) with $\gamma = 1/2$. Suppose moreover that σ is a Lipschitz map. Then it holds that the law of f_w converges as $d \to \infty$ towards $GP(0, K^{(2)})$, where the kernel $K^{(2)}$ is given by*

$$\begin{aligned} &K^{(2)}(x,x') := \mathbb{E}_{g\sim GP(0,K^{(1)})}\left[\sigma(g(x))\sigma(g(x'))\right] \\ \textit{where} \quad &K^{(1)}(x,x') := x^T x' + 1. \end{aligned}$$

Sketch of proof. The main idea is to use the central limit theorem 2. Indeed, one notes that in (8.5), the output $f_w(x)$ corresponds to a sum of d i.i.d. variables and the convergence to a Gaussian distribution follows from the central limit theorem. (One needs to make sure that $\sigma(X)$ has a second moment for X a Gaussian variable, which is the case since σ is Lipschitz.)

For the covariance, we first observe that $x \mapsto w_1 x + b$ is a Gaussian processes with covariance kernel $K^{(1)}$ as in the statement. Then, one conditions on the values of the hidden layers $\{x_k^{(1)}; k = 1, \dots, d\}$, and checks that $f_w(x)$ given $x^{(1)}$ is a Gaussian process (as a linear combination of independent Gaussian processes) with covariance kernel $\frac{1}{d}\sum_{k=1}^{d} x_k^{(1)}(x')_k^{(1)}$. One gets a limit as $d \to \infty$ that does not depend on x^1 nor $(x')^{(1)}$, and getting rid of the conditioning yields the claim. $\square$

Knowing that infinitely wide neural networks with the above initialization are Gaussian processes is informative about the prior knowledge we inject in our model, but does not provide any insight on what happens during training. In the next two sections, we present two recent techniques that will allow us to say more about training.

8.4.2 The Neural Tangent Kernel

In this section, we consider a neural network defined as in (8.5) with $\gamma = 1/2$. Since we study training, we denote by w_t the network's parameters at time t of training.

From Theorem 19, we know that the network at initialization f_{w_0} is a Gaussian process with explicit kernel. Recall that gradient descent steps are given by

$$w_{t+1} = w_t - \eta \frac{1}{m} \sum_{i=1}^{m} \partial_{\hat{y}} \ell(y_i, \hat{y})\big|_{\hat{y} = f_{w_t}(x_i)} \nabla_w f_{w_t}(x_i), \tag{8.6}$$

where $\eta > 0$ is the learning rate. We are interested in the dynamics of training, meaning that we would like to know more about how f_{w_t} evolves in time (and in particular what it looks like at convergence, when $t \to \infty$ (provided that it converges)). We do a first-order Taylor expansion of $f_{w_{t+1}}$ to read:

$$f_{w_{t+1}}(x) \approx f_{w_t}(x) + \partial_t w_t \cdot \nabla_w f_{w_t}(x)$$

Approximating $\partial_t w_t$ with a finite difference, we can replace it with (8.6) and get:

$$f_{w_{t+1}}(x) - f_{w_t}(x) \approx= -\eta \frac{1}{m} \sum_{i=1}^{m} \partial_{\hat{y}} \ell(y_i, \hat{y})\big|_{\hat{y}=f_{w_t}(x_i)} \nabla_w f_{w_t}(x_i) \cdot \nabla_w f_{w_t}(x). \tag{8.7}$$

It turns out that the dot product in the right-hand side defines a time-dependent kernel

$$\Theta_t^{(d)}(x, x') := \nabla_w f_{w_t}(x) \cdot \nabla_w f_{w_t}(x'),$$

where we made the dependency on the width d explicit. The kernel $\Theta_t^{(d)}$ is called the *Neural Tangent Kernel* (NTK) of the neural network at time t. Because the initialization is random, the NTK at initialization $\Theta_0^{(d)}$ is itself a random kernel. Moreover, the fact that its dynamics in time depends on the training makes the exact dynamics of the network f_{w_t} intractable.

However, there is a very important result from Jacot et al [8] then generalized in many ways (e.g. in Yang [13]) that gives us what we want:

Theorem 20. *There exists a deterministic kernel $\Theta^{(\infty)} : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ such that, in the setup above, for σ Lipschitz, twice differentiable with bounded second derivative, for all $x, x' \in \mathbb{R}$ and $t \in \mathbb{R}_+$, it holds with probability 1 that*

$$\lim_{d\to\infty} |\Theta_t^{(d)}(x, x') - \Theta^{(\infty)}(x, x')| = 0.$$

Furthermore, the limiting NTK has the following expression

$$\Theta^{(\infty)}(x, x') = K^{(1)}(x, x')\dot{K}^{(2)}(x, x') + K^{(2)}(x, x'),$$

where $K^{(1)}$ and $K^{(2)}$ are the kernels defined in Theorem 19, and $\dot{K}^{(2)}$ is defined as $K^{(2)}$ with the derivative σ' instead of σ in its definition.

Remark 29. The assumption that the neural network has a single hidden layer is superfluous and we make it solely for the sake of presentation.

$\star$ **Remark 13.** One can show that $\Theta^{(\infty)}$ is positive semi-definite when the input data $\{x_i : i = 1, \ldots, m\}$ lie on the sphere. (In our case – scalar inputs – it does not make much sense, however it does in the general d-dimensional case.)

$\star$ **Remark 14.** The assumptions on σ can be greatly relaxed, as it is enough to assume that the second order weak derivative of σ is polynomially bounded. (It is the case, for example, of the ReLU activation.)

Many consequences can be drawn from Theorem 20; we now loosely discuss the most straightforward.

Training follows kernel gradient descent. Ignoring the second order term in Equation (8.7), considering the limit as $d \to \infty$, the NTK is fixed and training has the following dynamics:

$$f_{w_{t+1}}(x) - f_{w_t}(x) = -\eta \frac{1}{m} \sum_{i=1}^{m} \partial_{\hat{y}} \ell(y_i, \hat{y})\big|_{\hat{y}=f_{w_t}(x_i)} \Theta^{(\infty)}(x_i, x).$$

Convergence. Suppose that the loss is the mean square loss $\ell(y, y') = \frac{1}{2}(y - y')^2$. For a vanishing learning rate η and as the number of steps of gradient descent tends to ∞ as $1/\eta$ [4], the training trajectory becomes

$$\partial_t f_{w_t}(x) = -\frac{1}{m} \sum_{j=1}^{m} \Theta^{(\infty)}(x, x_j)(f_{w_t}(x_j) - y_j).$$

Suppose that $\Theta^{(\infty)}$ is positive semi-definite. This differential equation has an explicit solution and in particular, it can be shown that f_{w_t} converges as $t \to \infty$ to

$$f_{w_\infty}(x) = f_{w_0}(x) - \sum_{i,j=1}^{m} \Theta^{(\infty)}(x, x_i)\Theta^{(\infty),-1}(x_i, x_j)(f_{w_0}(x_j) - y_j), \tag{8.8}$$

where $\Theta^{(\infty),-1}$ is the inverse of $\Theta^{(\infty)}$. We see that choosing x in the dataset, i.e. equal to some x_i, then $f_{w_\infty}(x_i) = y_i$ so the network perfectly fits the data and the empirical loss is zero.

[4] This corresponds to gradient flow as discussed in the optional section 3.5.2, which can be seen as the continuous version of gradient descent.

Gaussian process. Since f_{w_0} is a Gaussian process by Theorem 19 and since we apply a linear map to the Gaussian vector $(f_{w_0}(x_j))_{j=1,\ldots,m}$ in the expression of f_{w_∞} in (8.8), the neural network at convergence f_{w_∞} is itself a Gaussian process.

It is worth noting that this Gaussian process does not correspond to the Bayesian posterior given the prior f_{w_0} unless one subtracts the initial output function f_{w_0}; see e.g. [6] for more details on this.

Some limitations. The NTK regime (also called *kernel regime*, or *lazy regime*) does not fully describe neural networks behavior for several reasons. The first one is that neural networks used in practice do not contain an infinite number of neurons... But we won't be too concerned by that objection. Another critic made to the NTK regime is that the kernel is fixed as a result of the fact that individual weights asymptotically don't move during training: the map f_{w_t} evolves during training as a result of infinitely infinitesimal changes in the weights of the neurons. Loosely speaking, this causes the network to not learn any features in the data, akin to a kernel method with given kernel which fits the data as a linear combination of feature map of the kernel, without trying to learn these features at any point. However, feature learning can be a crucial characteristic of successful models in practice; see e.g. [4] for a language representation model using pre-training to learn important features of the data.

8.4.3 Mean Field regime

In this section we stay very informal to present another training regime where theoretical guarantees can be obtained.

We now assume that our 2-layer neural network is initialized as in (8.5) with $\gamma = 1$. Compared to the previous section where we supposed $\gamma = 1/2$, this initialization is small.

Let $\mu_d = \mu_d(w_1, w_2, b)$ be the measure on the weights space consisting of the average of Dirac masses on the neurons weights, that is

$$\mu_d = \frac{1}{d}\sum_{k=1}^{d} \delta_{(w_{1,k}, w_{2,k}, b_k)}.$$

Equation (8.5) with $\gamma = 1$ can now be rewritten in the integral form

$$f_w(x) = \int_{\mathbb{R}^3} w_2 \sigma(w_1 x + b)\mu_d(\mathrm{d}w_1, \mathrm{d}w_2, \mathrm{d}b).$$

We first note that in the infinite width limit $d \to \infty$, the infinite network at initialization is null because the weights are i.i.d. Gaussian and the law of large numbers tells us that for Z_1, Z_2, Z_3 i.i.d. $\mathcal{N}(0, 1)$,

$$\begin{aligned} f_w(x) &= \int_{\mathbb{R}^3} w_2 \sigma(w_1 x + b)\mu(\mathrm{d}w_1, \mathrm{d}w_2, \mathrm{d}b) \\ &= \mathbb{E}\left[Z_1 \sigma(Z_2 x + Z_3)\right] = \mathbb{E}\left[Z_1\right]\mathbb{E}\left[\sigma(Z_2 x + Z_3)\right] = 0, \end{aligned}$$

where $\mu := \lim_{d\to\infty} \mu_d$.

With an abuse of notation, we denote by μ_t the measure of the infinite-width neural network at time t of training. The advantage of writing the neural network in integral form and taking the limit is that the dynamics of the measure μ_t during training can be studied instead of that of the function. We refrain ourselves to do so as it involves many tools that are out of the scope of this class (from optimal transport theory such as Wassertein gradients). We refer the interesting reader to the paper [2]. One of the main results of this paper can be informally stated as

Theorem 21 (Informal)**.** *Under some assumption on the support of the initial measure μ_0 and if $\sigma =$ ReLU, then if μ_t converges to some μ_∞ as $t \to \infty$, then μ_∞ is optimal in the sense that the induced network f_{w_∞} achieves zero loss.*

Remark 30. Such a result is called a *global convergence result*: it does not claim that the network converges, but if it does, then its limit perfectly fits the data. It is important to note that the results in the mean-field regime require the assumption of having a single hidden layer. We chose $\sigma =$ ReLU, the result holds for more general activations under some homogeneity assumption of the network.

⋆ **Remark 15.** As opposed to the NTK regime, in the mean-field regime, feature learning occurs and we can move from one to the other regime simply by changing the initialization scale. For details on the impact of initialization scale and feature learning, see the paper [14].

8.5 Beyond feed forward neural networks

8.5.1 Convolutional neural networks

Convolutional Neural Networks (CNNs) are designed in such a way, that they can take into account spatial structure of the input. They were inspired by mice visual system and were originally designed to work with images. Compared to standard neural networks, CNNs have much fewer parameters which makes it possible to efficiently train very deep architectures.

The Convolutional neural network typically has the following structure (we consider 2-D CNN (images), there are 1-D (e.g.: speech) and 3-D (e.g.: medical imaging), possibly higher dimensions):

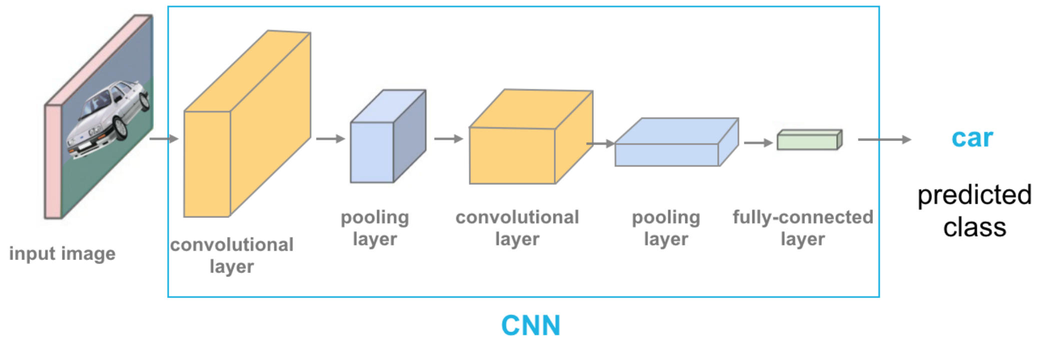


Figure 8.2: Diagram of CNN [?]

Formulation of each component:

Input: matrix $R^{n \times n}$

- Convolution layer: # filters, size of filters. (often $3x3$, $4x4$, $5x5$) It performs a discrete convolution between a filter and the (input or intermediary matrix). Below is the 2-d discrete convolution between matrix f and filter g

$$(f * g)(x, y) = \sum_{m=-M}^{M} \sum_{n=-N}^{N} f(x - n, y - m) g(n, m).$$

Several filters are formed.

- Pooling layer: maxpool (define the size of the max pool), returns the maximum value for the size of the maxpool.

  This operation performs downsampling and selects for the "dominant" pixel (i.e. the pixel with largest value). We note that similar images (say one is a slight shift of another) yield similar down-sampled images.

- Fully-connected layer: flatten input, map that to desired output size (e.g. multiclass classification)

- Output activation function: softmax (return probabilities of belonging to class i).

8.5.2 Generative Adversarial Neural Networks

The original Generative Adversarial Neural Network (GAN) was introduced as a new generative framework from training data sets. It addressed the question: "if you are given a data set of objects with a certain degree of consistency, can we artificially generate similar objects?"

Mathematically, objects in the dataset are samples generated from a common probability distribution D on the input space $\mathcal{X}$ and thus have this consistency between them.

In practice, we have observations of data points sampled from distribution D, and we want to approximate the underlying distribution with some approximation G. Suppose $\mathcal{X} \subset \mathbb{R}^n$. We start with an initial distribution γ defined in $\mathbb{R}^d$ and we define the mapping $G : \mathbb{R}^d \to \mathbb{R}^n$ such that if a random variable $z \in \mathbb{R}^d$ has distribution γ, then $G(z)$ has distribution D.

But how do we find a good approximation to D through G? We introduce another function, f_{disc}, called the discriminator function, which is a classifier tasked to predict whether a particular sample x is sampled from the original distribution D or from the approximation G, namely, if $D(x)$ is high, then $x \sim D$, if $D(x)$ is low, then $x \sim G(z)$.

Both the mapping G, called generator, and discriminator f_{disc} are neural networks (with their own set of parameters) which we will train.

The target loss function is given by:

$$\ell(G, f_{disc}) := \mathbb{E}_{x\sim D}\left[\log f_{disc}(x)\right] + \mathbb{E}_{z\sim\gamma}\left[\log(1 - f_{disc}(G(z)))\right].$$

The GAN solves the minimax problem

$$\min_G \max_{f_{disc}} \ell(G, f_{disc}).$$

For a given generator G, $\max_{f_{disc}} \ell(G, f_{disc})$ will optimise the discriminator f_{disc} to reject samples $G(z)$, by assigning high values to samples from the distribution D and low values to the generated samples $G(z)$. Whereas for a given discriminator f_{disc}, $\min_G \ell(G, f_{disc})$ optimises G so that the generated samples $G(z)$ will attempt to fool the discriminator f_{disc} into assigning high values.

Note that we can't actually compute these expectations, so we must approximate them:

$$\begin{aligned}\mathbb{E}_{x\sim D}\left[\log f_{disc}(x)\right] &\approx \frac{1}{m_1}\sum_{i=1}^{m_1} \log f_{disc}(x_i)\\ \mathbb{E}_{z\sim\gamma}\left[\log(1 - f_{disc}(G(z)))\right] &\approx \frac{1}{m_2}\sum_{i=1}^{m_2} \log\left(1 - f_{disc}(G(z_i))\right).\end{aligned}$$

where $x_1, ..., x_{m_1}$ are coming from the training set $\mathcal{X}$ and $z_1, ..., z_{m_2}$ from a dummy distribution γ.

The optimisation is done in two steps: first update the discriminator f_{disc} taking the gradient with respect to parameters of f_{disc} and computing the gradient ascent update, then, update the generator G taking the gradient with respect to parameters of G and computing the gradient descent update.

GANs have become very powerful tools for generative models, where we want to generate similar samples to a given dataset, for example, new faces, new rooms, etc[5]. Not only this, GANs can be used to solve problems in which we can't easily formalise a loss function, or in situations where we don't have access to a loss function, or we are unable to compute gradients on the loss function.

8.6 Tricks of the trade

This section is loosely based on the lecture series [10], advice that has come from many years of theory and experimentation that have lead to substantial differences in terms of speed, ease of implementation and accuracy when it comes to putting algorithms to work in practice. A lot of the information on these lecture series is already implemented (by default) on machine learning libraries (e.g. tensorflow, keras, torch).

In this section, I will give a summary and justification/reasoning to why these "tricks of the trade" are important in practice. Many of the advice are quite general and apply to most neural networks.

What are some things that can go wrong when using neural network models[6]?

- Poor performance (overfitting, stuck in local minima, etc...)
- Inability to train network (exploding gradients (NaN values) and flat gradients (refer to section 8.2.2), diverging solutions, slow convergence)
- Very large hyper-parameter space

[5]Check out `https://thisxdoesnotexist.com/` for many examples of GANs used to generate a variety of things!

[6]This is of course not an exclusive problem of neural networks.

8.6.1 Input regularisation

We've seen in Chapter 5 that sometimes it is necessary to normalise the input (linear regression with regularisation). An input X is typically centered by subtracting the empirical mean $\hat{X}$ and dividing by the empirical variance σ_X^2:

$$X' = \frac{X - \hat{X}}{\sigma_X^2}.$$

For neural networks, normalising the input is also a common practice, namely, such that:

- Average of each input over the training set should be close to zero;
- Scale input variables so that their covariances are about the same;
- input variables should be uncorrelated if possible.

Shifting and scaling is quite simple. However, decorrelating the inputs might be tricky. Sometimes, *Principal component analysis* is applied to the feature matrix to remove linear correlations in input.

It is also observed that convergence is usually faster if the average of each input over the training set is close to zero, meaning there are both positive and negative values. Let us consider an example of the extreme case:

Example 8.6.1. Suppose all the components of an input vector are positive (or all negative).

The gradient update for any weight i, j that sits on layer n, w_{ij}^n, is given as[7]:

$$w_{ij}^{t+1} = w_{ij}^t - \eta \frac{\partial L}{\partial w_{ij}} \quad \text{where} \quad \frac{\partial L}{\partial w_{ij}^n} = x_j^{n-1} \frac{\partial L}{\partial z_i^n}$$

[7]You can verify this by chain rule

Let us focus on the first layer, $n = 1$. The update for the weights corresponding to a particular output node i (corresponding to row i of the weights matrix W^1) are proportional to $x_j^0 \frac{\partial L}{\partial z_i^1}$. If all components of an input vector are positive, all the updates of the weights that feed into node i will have the same sign ($sign(\frac{\partial L}{\partial z_i^1})$). This means, those weights can only all decrease or increase together for a given input. If the weight vector must change direction, it can only do so by zigzagging, which is ineficient and slow.

Normalising the inputs is not only a concern for the speed of convergence but really for the trainability of the network. Due to finite numerical precision, numerical overflow can occur, turning gradient updates into NaN updates.

8.6.2 Stabilising the gradients

We saw in Section 8.2.3 the problems of vanishing and exploding gradients when training a neural network. We discussed how avoiding these problems motivated different weights initialization schemes. We have, however, no guarantee that these initializations prevent these two phenomena to occur beyond the first step of gradient descent. We present here two techniques that have been introduced to keep the training stable.

Gradient clipping. What can we do if the gradient becomes too large? Simply normalize it! The following technique is called *gradient clipping* and is quite intuitive: let $\epsilon > 0$, then

- if $||\nabla_w C(w)|| > \epsilon$, update the parameters with $\epsilon \frac{\nabla_w C(w)}{||\nabla_w C(w)||}$,
- otherwise, use the usual update $\nabla_w C(w)$.

If some components of the gradient dominate, the others, after clipping, might end up too small. Another practice is to clip each component of the gradient independently if it crosses a threshold ϵ.

Batch normalization. Batch normalization is a way to guarantee that all layers' activations are centered and normalized with respect to the batch of data samples the network is training on. We can formally define it as a new type of layer of the neural network. A batch normalization layer, denoted by BN, does the following:

- $\{x^{(i)}; i = 1, \ldots, m\} \subset \mathbb{R}^d$ is some batch of input.
- $\mu := \frac{1}{d} \sum_{i=1}^{d} x^{(i)}$ is the empirical mean (vector) of the input batch.
- $v := \frac{1}{d} \sum_{i=1}^{d} (x^{(i)} - \mu)^2$ is the empirical variance (vector) of the input batch, where the square function is applied component-wise.
- $\overline{x}^{(i)} := \frac{x^{(i)} - \mu}{\sqrt{v + \epsilon}}$ for all $i = 1, \ldots, m$, where $\epsilon > 0$ is a hyperparameter that prevents the denominator to be null.
- $\text{BN}(x^{(i)}) := (\gamma_k \overline{x}_k^{(i)} + \beta_k)_{k=1,\ldots,d}$, where γ_k, β_k are parameters that can be **learnable**.

A neural network can then be composed of many layers, some of them being BN.

Despite the several heuristics and the empirical success of batch normalization, there is, for now, no very robust theory nor consensus on the different effects of batch normalization.

8.6.3 Preventing overfitting

Avoiding overfitting is a common desire in Machine Learning, to have better hope that the model generalises to unseen data. For neural networks, there are different strategies to prevent overfitting.

Regularisation

Similarly to previously seen models, we can also add a regularisation term on the loss function, penalising the model parameters through a L_1 or L_2 norm.

Early stopping

The rational about early stopping is that when we decide on the number of epochs to train a network, it's not usually a very well informed decision.

monitoring the training and a validation loss, stopping when the validation loss no longer decresese / increases.

1. Split the training data into a training set and a validation set, e.g. in a 2-to-1 proportion.
2. Train only on the training set and evaluate the per-example error on the validation set once in a while, e.g. after every fifth epoch.
3. Stop training as soon as the error on the validation set is higher than it was the last time it was checked.
4. Use the weights the network had in that previous step as the result of the training run.

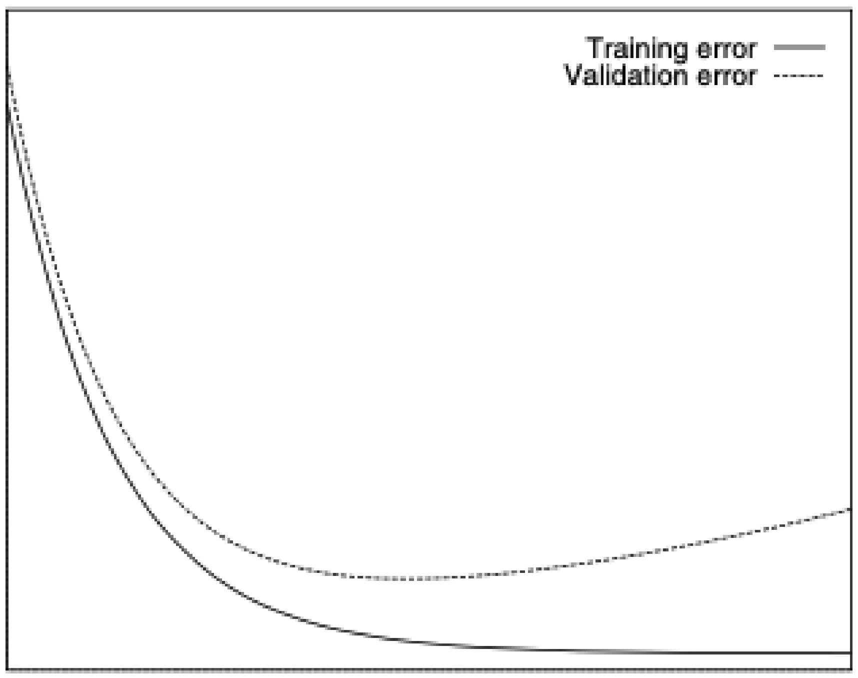
in reality, the curves don't behave so nicely.

As we see, choosing a stopping criterion predominantly involves a tradeoff between training time and generalization error. However, some stopping criteria may typically find better tradeoffs that others.

Empirically, leads to faster to train networks, prevents overfitting, can be 'tricky' to get right because the behaviour of the errors is not straightforward, tunable parameters (when is the validation error too large with respect to the training error? what is the number of successive s (number of epochs)?).

8.6.4 Dropout

Even though overparametrized neural networks seem to generalize better than expected, they are not completely immune to overfitting. A efficient way of regularizing them and enhance their generalization performances, one



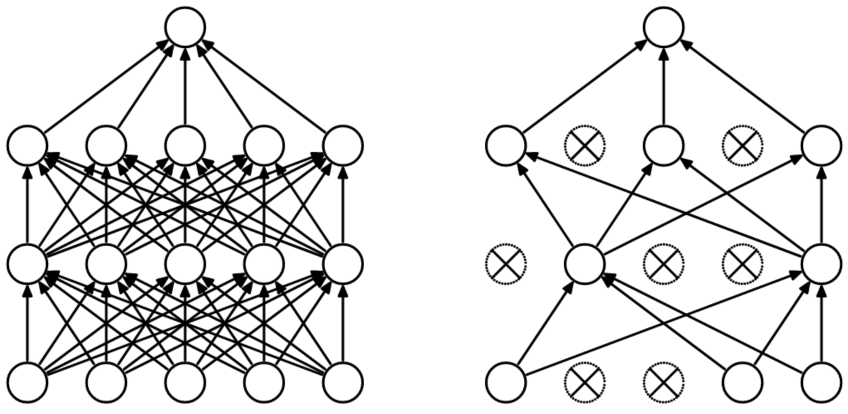
can turn off at random neurons while training. This method is called *dropout*. It works as follows:

- For each neuron, independently turn it off with a probability $1 - p \in (0, 1)$; p is called the *keep probability*, equivalently $1 - p$ is called the *dropout rate*. It is one more hyperparameter to the model
- The sub-network consisting of neurons that have been kept is trained for one step of gradient descent (or any optimization algorithm at use). Only the weights of the neurons that have been kept are updated during this step
- Repeat the two previous steps until the end of training.
- To make predictions with the trained neural network – or to evaluate it on a test dataset – use the full neural network with weights rescaled by the keep probability p.

The reason why we rescale the weights when predicting is because when the weights are updated, the sub-networks used contain an average proportion p of neurons, hence their weights tend to be bigger than they should when using all of them together. Think about the sum of two perfect predictors: the obtained predictor performs poorly unless we divide its output by 2.

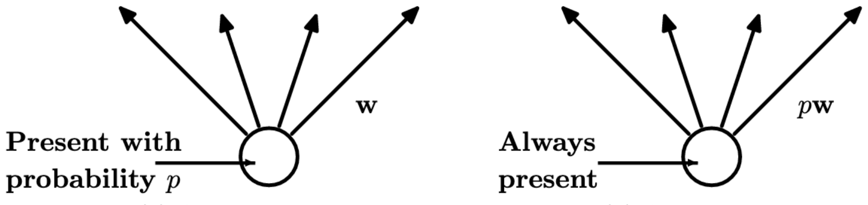
Why is it a good idea to use dropout? As we said, this method trains smaller networks inside the network, thus encouraging learning more sparse functions (i.e. simpler, needing less parameters to be expressed). Each sub-network hence has less opportunities to overfit. Another informal reason is that the full network resembles an average of smaller networks. Averaging simple predictors to construct a good predictor can be a very good idea to enhance training and performance; in the next chapter, we will see a model that is based on that idea, where the simple predictors do not even need to perform that well.

Dropout may also refer to other similar methods where individual weights are dropped out instead of individual neurons.


(a) Standard Neural Net (b) After applying dropout.

During training, we compare a neural network without dropout:

$$\begin{aligned} \widetilde{x}_i^{(l+1)} &= w_i^{(l+1)} x^{(l)} + b_i^{(l+1)}, \\ x_i^{(l+1)} &= \sigma(\widetilde{x}_i^{(l+1)}), \end{aligned}$$



(a) At training time (b) At test time

and with dropout:

$$\begin{aligned}
r_j^{(l)} &\sim \text{Bernoulli}(p), \\
z_i^{(l)} &= r_i^{(l)} x_i^{(l)}, \\
\widetilde{x}_i^{(l+1)} &= w_i^{(l+1)} z_i^{(l)} + b_i^{(l+1)}, \\
x_i^{(l+1)} &= \sigma(z_i^{(l+1)}).
\end{aligned}$$

8.6.5 Dealing with hyper-parameters

Architecture hyperparameters define the function space $\mathcal{H}$ where our approximator lives in. Some hyperparameters are: number of layers, width of each layer, activation functions, other types of layers (Dropout, normalising layers, etc...).

Then we also have *training hyperparameters*, that specify how the training algorithm $\mathcal{A}$ behave: number of epochs, batchsize, learning rate (momentum, etc..), training set / validation set split, loss function (type, l1-l2 penalties)

Many times, good hyperparameters come from experience, trial and error, theoretical justifications or empirical results. There are also computational frameworks that can help explore this large hyperparameter space, such as hyperopt [?].

Chapter 9

Ensemble methods

Contents

Ensemble methods are usually reserved for methods that generate a model using an aggregate of *base learners*. There are different ways to approach the construction of ensemble methods. In this chapter, we will focus on **boosting**.

Boosting is an algorithmic paradigm that grew out of a theoretical question and became a very practical machine learning tool. The boosting approach uses a generalisation of linear predictors to address two major issues:

- The first is the **bias-complexity tradeoff**. We have seen that the generalisation error of an ERM learner can be decomposed into a sum of **approximation error** and **estimation error**, as described in equation (4.2). The more expressive the hypothesis class the learner is searching over, the smaller the approximation error is, but the larger the estimation error becomes. A learner is thus faced with the problem of picking a good trade-off between these two considerations. The boosting paradigm allows the learner to have smooth control over this trade-off. **The learning starts with a basic class (that might have a large approximation error), and as it progresses the class that the predictor may belong to grows richer.**

- The second issue that boosting addresses is the computational complexity of learning. A boosting algorithm amplifies the accuracy of weak learners. Intuitively, one can think of a weak learner as an algorithm that uses a simple "rule of thumb" to output a hypothesis that comes from an easy-to-learn hypothesis class and performs just slightly better than a random guess. When a weak learner can be implemented efficiently, boosting provides a tool for aggregating such weak hypotheses to approximate gradually good predictors for larger, and harder to learn, classes.

In this chapter, we start again by considering binary classification for the sake of exposition, though ensemble methods apply to general classification tasks and regression.

Figure 9.1 illustrates how properly aggregating base learners can solve a task on which a base learners taken individually does not perform extremely well.

9.1 Weak learner

Let us define the concept of γ-weak-learnability first. If we were to randomly guess labels by tossing a fair coin each time, then in average, we would be right half of the time and wrong just as much. A γ-weak-learner is a learner that merely does slightly better than that.

Definition 9.1.1. A learning algorithm A is a γ-weak-learner for a class $\mathcal{H}$ if there exists a function $m_{\mathcal{H}} : (0,1) \to \mathbb{N}$ such that for every $\delta \in (0,1)$, for every distribution $\mathcal{D}$ over $\mathcal{X}$, and for every labeling function $f : \mathcal{X} \to \{\pm 1\}$,

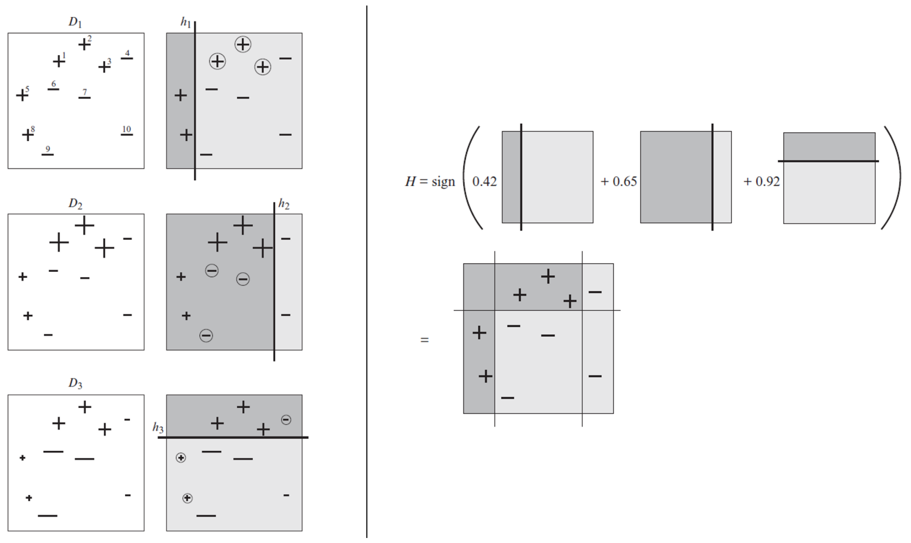


Figure 9.1: An illustration of how AdaBoost behaves on a tiny toy problem with $m = 10$ examples. Credit: [11] Figures 1.1 and 1.2, the reader will find greater details on this example therein.
Left: Each row depicts one round, for $t = 1, 2, 3$. The left box in each row represents the distribution $D^{(t)}$, with the size of each example scaled in proportion to its weight under that distribution. Each box on the right shows the weak hypothesis h_t, where darker shading indicates the region of the domain predicted to be positive. Examples that are misclassified by h_t have been circled.
Right: The combined classifier for the toy example is computed as the sign of the weighted sum of the three weak hypotheses, $w_1h_1 + w_2h_2 + w_3h_3$ as shown at the top. This is equivalent to the classifier shown at the bottom.

if the realizable assumption holds with respect to $\mathcal{H}$, $\mathcal{D}$, and f, then when running the learning algorithm on $m \geq m_{\mathcal{H}}(\delta)$ i.i.d. examples generated by $\mathcal{D}$ and labeled by f, the algorithm returns a hypothesis f_w such that, with probability of at least $1 - \delta$, $R_{(\mathcal{D},f)}(f_w) \leq 1/2 - \gamma$.

Definition 9.1.2. A hypothesis class $\mathcal{H}$ is γ-weak-learnable if there exists a γ-weak-learner for that class.

Note that this definition is almost identical to the definition of PAC learning (here we will call strong learning) shown in chapter 3.7, with one crucial difference: strong learnability implies the ability to find an arbitrarily good classifier, with error rate at most ϵ for an arbitrarily small ϵ, when considering the non-agnostic case. In weak learnability, however, we only need to output a hypothesis whose error rate is at most $1/2 - \gamma$ for a fixed $\gamma > 0$, namely, whose error rate is slightly better than what a random labeling would give us. The hope is that it may be easier to come up with efficient weak learners than with efficient (full) PAC learners.

One possible approach is to take a "simple" hypothesis class, denoted $\mathcal{B}$, and to apply ERM with respect to $\mathcal{B}$ (stands for base class) as the weak learning algorithm. For this to work, we need that $\mathcal{B}$ will satisfy two requirements:

- $ERM_{\mathcal{B}}$ is efficiently implementable.
- For every sample that is labeled by some hypothesis from $\mathcal{H}$, any $ERM_{\mathcal{B}}$ hypothesis will have an error of at most $1/2 - \gamma$.

Remark 31. What can be considered as weak learners?

- Decision stumps (A one-level decision tree.)

$$\mathcal{H}_{DS} = \{x \rightarrow sign(\theta - x_j)b : \theta \in \mathbb{R}, j \in [d], b \in \{\pm 1\}\}$$

- Decision trees. These are typically defined recursively.
- While traditionally decision stumps/trees were the *defacto* weak learners, but you can also use linear functions, splines, SVMs, shallow networks...

Let us see an example of how to find an optimal $h \in \mathcal{H}_{DS}$, in the class of Decision stumps.

Example 9.1.1. Let $\mathcal{X} = \mathbb{R}^d$ and consider the base hypothesis class over $\mathbb{R}^d$ to be:

$$\mathcal{H}_{DS} = \{x \mapsto sign(\theta - x_j)b : \theta \in \mathbb{R}, j \in [d], b \in \{\pm 1\}\}.$$

For simplicity, let us assume $b = 1$ (this flips the label ± 1). Let $S = \{(x_1, y_1), ..., (x_m, y_m)\}$ be a training set. We will show how to implement an ERM rule, namely, how to find a decision stump that minimizes $L(h)$.

Let us introduce the vector D, a probability vector in $\mathbb{R}^m$ (that is, all elements of D are non-negative and $\sum_i D_i = 1$). The weak learner we describe receives D and training set S and outputs a decision stump $h : \mathcal{X} \to \mathcal{Y}$ that minimizes the risk w.r.t. D:

$$L_D(f_w) = \sum_{i=1}^{m} D_i 1_{f_w(x_i) \neq y_i}.$$

Note that if $D = (1/m, ..., 1/m)$ then $L_D(f_w) = L(f_w)$.

Each decision stump is parametrised by an index $j \in [d]$ (selects which dimension of the feature vector to split over), and a threshold θ. Therefore, minimizing $L_D(h)$ amounts to solving the minimization problem:

$$\min_{j \in [d]} \min_{\theta \in \mathbb{R}} \left(\sum_{i: y_i = 1} D_i 1_{x_{i,j} > \theta} + \sum_{i: y_i = -1} D_i 1_{x_{i,j} \leq \theta} \right). \tag{9.1}$$

Note here we just wrote $L_D(f_w)$ in such a way that we show the mislabelling when the true label is positive, and the prediction is negative, and when the true label is negative, and the prediction is positive. What we want to do now is to show that this can be further simplified, eventually yielding to an easy minimization problem.

Fix $j \in [d]$ and sort the training examples such that $x_{1,j} \leq x_{2,j} \leq \cdots \leq x_{m,j}$. Let us define the set $\Theta_j = \{\frac{x_{i,j} + x_{i+1,j}}{2} : i \in [m-1]\} \cup \{x_{1,j} - 1, x_{m,j} + 1\}$. This is essentially setting up a grid for which, for any $\theta \in \mathbb{R}$, there exists

$\theta' \in \Theta_j$ that yields the same predictions for the sample S. Then, we can minimize θ over Θ_j.

This gives us an efficient procedure: choose $j \in [d]$ and $\theta \in \Theta_j$ that minimize the objective value in (9.1). This yields a runtime complexity of $\mathcal{O}(dm^2)$, but it's possible to minimize the objective in $\mathcal{O}(dm)$. Refer to [**?**] if interested.

9.2 Adaboost

AdaBoost (short for Adaptive Boosting) is an algorithm that has access to a weak learner and finds a hypothesis with a low empirical risk. The AdaBoost algorithm receives as input a training set of examples $S = \{(x_1, y_1), ..., (x_m, y_m)\}$ and the boosting process proceeds in a sequence of consecutive rounds. At round t, the booster algorithm first defines a probability over the samples S, denoted by $D^{(t)}$[1]. Then, the booster algorithm passes the probability vector $D^{(t)}$ and the sample S to the weak learner. The weak learner is assumed to return a "weak" hypothesis, h_t, whose error is given by,

$$\epsilon_t = R_D^{(t)}(h_t) = \sum_{i=1}^{m} D_i^{(t)} 1_{h_t(x_i) \neq y_i},$$

which is at most $1/2 - \gamma$ for a fixed $\gamma \in (0, 1/2)$[2] that does not depend on t.

Then, AdaBoost assigns a weight w_t to h_t, given by

$$w_t = \frac{1}{2} \log\left(\frac{1}{\epsilon_t} - 1\right),$$

that is, the smaller the error, the larger the weight.

Remark 32. What is the weight w_t? In the end, AdaBoost returns a classifier that aggregates all the weak learners that were obtained after each round t, of the form:

$$f_w(x) = \sum_{t=1}^{T} w_t h_t(x).$$

[1]Represented as a probability vector, as defined previously.

[2]There is a probability of at most δ (from definition 9.1.1) that the weak learner fails to have an error smaller than $1/2 - \gamma$

This can be seen as a linear combination of the weak learners h_t. We will see in the proof of the forthcoming theorem 22 that this choice of w_t is "optimal" in some sense.

At the end of the round, AdaBoost updates the probability vector $D^{(t)}$ so that examples on which h_t is wrong will get a higher probability mass while examples on which h_t is correct will get a lower probability mass. This gives more *importance* to the points that h_t misclassifies, for the next weak learner to focus on.[3]

The algorithm reads:

```
AdaBoost

input:
    training set $S = (\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_m, y_m)$
    weak learner WL
    number of rounds $T$
initialize $\mathbf{D}^{(1)} = (\frac{1}{m}, \ldots, \frac{1}{m})$.
for $t = 1, \ldots, T$:
  invoke weak learner $h_t = \mathrm{WL}(\mathbf{D}^{(t)}, S)$
  compute $\epsilon_t = \sum_{i=1}^{m} D_i^{(t)} \mathbb{1}_{[y_i \neq h_t(\mathbf{x}_i)]}$
  let $w_t = \frac{1}{2} \log \left( \frac{1}{\epsilon_t} - 1 \right)$
  update $D_i^{(t+1)} = \frac{D_i^{(t)} \exp(-w_t y_i h_t(\mathbf{x}_i))}{\sum_{j=1}^{m} D_j^{(t)} \exp(-w_t y_j h_t(\mathbf{x}_j))}$ for all $i = 1, \ldots, m$
output the hypothesis $h_s(\mathbf{x}) = \mathrm{sign} \left( \sum_{t=1}^{T} w_t h_t(\mathbf{x}) \right)$.
```

Figure 9.2: Algorithm for AdaBoost (from [11])

Remark 33. There are two quantities which play a role of *weight*. One is w_t, which gives the weights the contribution of a weak learner h_t in the final model f_w. Another weight, given by the probability vector $D^{(t)}$, gives a weight to each data point.

In the next theorem, we suppose that the dataset $S = \{(x_i, y_i); i = 1..m\}$ is such that the labels y_i's are generated by a map f in a hypothesis class $\mathcal{H}$ that is γ-weak-learnable.

[3]A video that shows the Adaboost algorithm. `https://www.youtube.com/watch?v=k4G2VCuOMMg`

Theorem 22. *Let S be a training set and assume that at each iteration of AdaBoost, the weak learner returns a hypothesis for which $\epsilon_t \leq 1/2 - \gamma$. Then, the training error of the output hypothesis of AdaBoost is at most*

$$L(h_s) = \frac{1}{m} \sum_{i=1}^{m} 1_{h_s(x_i) \neq y_i} \leq \exp(-2\gamma^2 T).$$

Proof. For each round t, let $f_t = \sum_{k \leq t} w_k h_k$, so that the output of Adaboost is $H_T := \text{sign}(f_T)$. In addition, let

$$Z_t = \sum_{i=1}^{m} D_i^{(t)} e^{-y_i w_t h_t(x_i)},$$

which is the normalisation factor so that $D_i^{(t+1)}$, as defined in the AdaBoost algorithm described in figure 9.2, is indeed a probability distribution.

Unrolling the recurrence to update $D^{(T+1)}$, for all $i = 1..m$, we can write

$$\begin{aligned} D_i^{(T+1)} &= D_i^{(1)} \prod_{t=1}^{T} \frac{\exp\big(-w_t y_i h_t(x_i)\big)}{Z_t} \\ &= \frac{1}{m} \times \frac{\exp\Big(-y_i \sum_{t=1}^{T} w_t h_t(x_i)\Big)}{\prod_{t=1}^{T} Z_t} \\ &= \frac{1}{m} \times \frac{\exp\left(-y_i f_T(x_i)\right)}{\prod_{t=1}^{T} Z_t}. \end{aligned}$$

Note that $1_{H_T(x) \neq y} \leq e^{-y F_T(x)}$, since x is misclassified by H_T if and only if $f_T(x)$ and y have opposite signs. Therefore,

$$\begin{aligned} R_S(H_T) &= \frac{1}{m}\sum_{i=1}^{m} 1_{H_T(x_i)\neq y_i} \\ &\leq \frac{1}{m}\sum_{i=1}^{m} \exp(-y_i f_T(x_i)) \\ &= \sum_{i=1}^{m} D_i^{(T+1)} \prod_{t=1}^{T} Z_t \\ &= \prod_{t=1}^{T} Z_t. \end{aligned}$$

We now rewrite Z_t as

$$\begin{aligned} Z_t &= \sum_{i=1}^{m} D_i^{(t)} \exp(-w_t y_i h_t(x_i)) \\ &= \sum_{i:y_i=h_t(x_i)} D_i^{(t)} \exp(-w_t) + \sum_{i:y_i\neq h_t(x_i)} D_i^{(t)} \exp(w_t) \\ &= (1-\epsilon_t)\exp(-w_t) + \epsilon_t \exp(w_t). \end{aligned}$$

By definition of ϵ_t. Furthermore, by definition of $w_t = \frac{1}{2}\log(1/\epsilon_t - 1)$, we get

$$\begin{aligned} Z_t &= (1-\epsilon_t)\sqrt{\frac{\epsilon_t}{1-\epsilon_t}} + \epsilon_t\sqrt{\frac{1-\epsilon_t}{\epsilon_t}} \\ &= \sqrt{4\epsilon_t(1-\epsilon_t)} \end{aligned}$$

By our assumption, we have that $\epsilon_t \leq 1/2 - \gamma$, then, we can bound the quantity above to obtain:

$$Z_t \leq \sqrt{1-4\gamma^2},$$

recalling that $g(x) = x(1-x)$ is monotonically increasing $[0, 1/2]$. We thus have proven that

$$R_S(H_T) \leq (1-4\gamma^2)^{T/2}.$$

To conclude, recall the fact that for all $x \in \mathbb{R}$, the exponential function satisfies $1+x \leq \exp(x)$ and use it so that $1-4\gamma^2 \leq \exp(-4\gamma^2)$, which entails that

$$R_S(H_T) \leq \exp(-2\gamma^2 T),$$

as claimed. □

Thanks to the above theorem, we see that even though each weak learner performs only slightly better than a purely (uniform) random guess, AdaBoost is able to choose a linear combination of them that yields a predictor whose error on the dataset decreases exponentially fast in the number of weak learners.

⋆ **Remark 16.** However, what we really care about is the true risk of the output hypothesis, i.e. the generalisation error. It turns out that

$$R_D(h) \leq R_S(h) + O\left(\sqrt{\frac{TVC(H)}{m}}\right).$$

$d = VC(H)$. We will not discuss this in this class.

9.2.1 * A sufficient condition for weak-learnability

Theorem 22 proves that the assumption of γ-weak learnability is sufficient to ensure that AdaBoost will drive down the training error very quickly. But when does this assumption of γ-weak learnability actually hold?

In this section, we provide a condition that implies the assumption of empirical weak learnability. This condition is only in terms of the functional relationship between the data instances and their labels, and does not involve distributions over examples.

Let all the weak hypothesis belong to some class of hypothesis $\mathcal{H}$. Suppose our training sample S is such that for some weak hypothesis $g_1, \cdots, g_k$ from $\mathcal{H}$, and for some nonnegative coefficients $a_1, \cdots, a_k$, with $\sum_{j=1}^k a_j = 1$, and for some $\theta > 0$, it holds that

$$y_i \sum_{j=1}^{k} a_j g_j(x_i) \geq \theta \tag{9.2}$$

for each example (x_i, y_i) in S. This condition implies that y_i can be computed by a weighted majority vote of the weak hypothesis:

$$y_i = \operatorname{sign}\left(\sum_{j=1}^{k} a_j g_j(x_i)\right),$$

namely, when it is strictly greater than 0. (9.2) demands that significantly more than a bare weighted majority to be correct. When this condition holds for all i, we say the sample S is linearly separable with margin θ.

9.2.2 Connections to other models

We saw, through AdaBoost, that boosting is essentially applying the weak classification algorithm to repeatedly modified versions of the data, producing a sequence of weak classifiers $h_t(x), \quad t = 1, \cdots, T$. The predictions from all of them are then combined through a weighted majority vote to produce the final prediction:

$$H_T(x) = \operatorname{sign}\left(\sum_{t=1}^{T} w_t h_t(x)\right),$$

where w_t are computed by the boosting algorithm and weight the contribution of each respective h_t. Their effect is to give higher influence to the more accurate classifiers in the sequence.

In this section, we will first show that AdaBoost fits an additive model in a base learner, optimising a novel exponential loss function. Then, we will develop a class of gradient boosted models (GBMs), for boosting weak learners for any loss function.

Boosting and additive models

Boosting is a way of fitting an additive expansion in a set of elementary "basis" functions. Here, the basis functions are individual classifiers $h_t(x) \in \{-1, +1\}$. More generally, basis function expansions take the form:

$$f(x) = \sum_{k=1}^{M} \beta_k b(x; \gamma_k),$$

where β are expansion coefficients and $b(x; \gamma) \in \mathbb{R}$ are usually simple functions of the multivariate argument x characterised by a set of parameters γ. E.g. decision stumps, γ parametrises the split variables and split points. For example, additive expansions like this can describe single-hidden-layer neural networks.

Models based on additive expansions are fit by minimizing a loss function averaged over the training data, such as the squared-error or likelihood-based loss function

$$\min_{\{\beta_k, \gamma_k\}} \sum_{i=1}^{m} L\left(y_i, \sum_{k=1}^{M} \beta_k b(x; \gamma_k)\right). \tag{9.3}$$

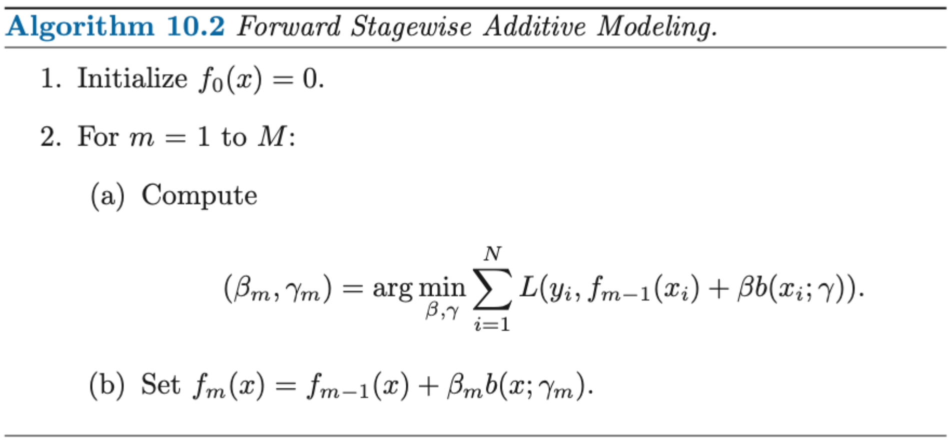
For many loss functions L or basis functions b, this requires computationally intensive numerical optimisation techniques. A simple alternative is to rapidly solve the sub-problem of fitting just a single basis function:

$$\min_{\{\beta, \gamma\}} \sum_{i=1}^{m} L\left(y_i, \beta b(x; \gamma)\right).$$

The *forward stagewise additive modelling* approximates the solution (9.3) by sequentially adding new basis functions to the expansion, without adjusting the parameters and coefficients of those that have been already added. This is outline in the algorithm shown in 9.3.

At each iteration m, one solves for the optimal basis function $b(x; \gamma_m)$ and corresponding $_m$ to add to the current expansion $f_{m-1}(x)$. This produces $f_m(x)$, and the process is repeated. Previously added terms are not modified.[4]

[4]It can be shown, for the squared-loss



```
Algorithm 10.2 Forward Stagewise Additive Modeling.
1. Initialize $f_0(x) = 0$.
2. For $m = 1$ to $M$:
    (a) Compute
        $(\beta_m, \gamma_m) = \arg\min_{\beta,\gamma} \sum_{i=1}^{N} L(y_i, f_{m-1}(x_i) + \beta b(x_i; \gamma))$.
    (b) Set $f_m(x) = f_{m-1}(x) + \beta_m b(x; \gamma_m)$.
```

Figure 9.3: Forward stagewise algorithm

AdaBoost is equivalent to forward stagewise additive modelling using the loss function:

$$L(y, f(x)) = \exp(-yf(x)).$$

This equivalence between AdaBoost and forward stagewise additive modelling was only discovered five years after AdaBoost's inception.

$$L(y, f(x)) = (y - f(x))^2$$

one has

$$\begin{aligned} L(y_i, f_{m-1}(x_i) + \beta b(x_i; \gamma)) &= (y_i - f_{m-1}(x_i) - b(x; \gamma))^2 \\ &= (r_{im} - \beta b(x_i; \gamma))^2 \end{aligned}$$

where $r_{im} = y_i - f_{m-1}(x_i)$ is simply the residual of the current model on the ith observation. Thus, for the squared-error loss, the term ${}_m b(x; \gamma_m)$ that best fits the current residuals is added to the expansion at each step.

9.2.3 * Gradient Boosting

Very informally speaking, gradient boosting is currently a very popular method that requires little *fiddling* to get a good classifier. Unlike neural networks, which require choosing more hyper-parameters, the gradient boosting algorithm produces predictive models with less engineering choices.

The main difference between AdaBoost and Gradient Boosting, is that the iterative corrections are not introduced by re-weighting the data, but through gradient updates. This has the advantage that the problem can use a general loss function, as well as being more computationally efficient.

9.3 Boosting regression

Although we do not go into detail in class about how this works, regression with trees is also possible, generating a hypothesis of the form:

$$h(x) = \sum_{t=1}^{T} w_t h_t(x).$$

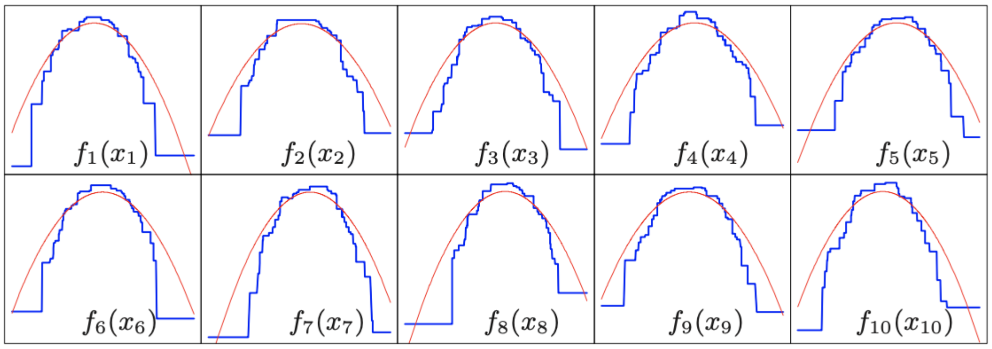


Figure 9.4: Example of Boosting for regression.

Bibliography

[1] Mikhail Belkin, Daniel Hsu, Siyuan Ma, and Soumik Mandal. Reconciling modern machine-learning practice and the classical bias–variance trade-off. *Proceedings of the National Academy of Sciences*, 116(32):15849–15854, 2019.

[2] Lénaïc Chizat and Francis Bach. On the global convergence of gradient descent for over-parameterized models using optimal transport. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018.

[3] G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems*, 2(4):303–314, 1989.

[4] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[5] Leonardo Ferreira Guilhoto. An overview of artificial neural networks for mathematicians. 2018.

[6] Bobby He, Balaji Lakshminarayanan, and Yee Whye Teh. Bayesian deep ensembles via the neural tangent kernel. *Advances in neural information processing systems*, 33:1010–1022, 2020.

[7] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366, 1989.

[8] Arthur Jacot, Clément Hongler, and Franck Gabriel. Neural tangent kernel: Convergence and generalization in neural networks. In Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett, editors, *NeurIPS*, pages 8580–8589, 2018.

[9] Radford M Neal. Priors for infinite networks. In *Bayesian Learning for Neural Networks*, pages 29–53. Springer, 1996.

[10] Yoav Freund Robert E. Schapire. *Boosting: Foundations and Algorithms.* The MIT Press, 2014.

[11] Yoav Freund Robert E. Schapire. *Boosting: Foundations and Algorithms.* The MIT Press, 2014.

[12] Shai Shalev-Shwartz and Shai Ben-David. *Understanding Machine Learning - From Theory to Algorithms.* Cambridge University Press, 2014.

[13] Greg Yang. Scaling limits of wide neural networks with weight sharing: Gaussian process behavior, gradient independence, and neural tangent kernel derivation. 2019.

[14] Greg Yang and Edward J Hu. Feature learning in infinite-width neural networks. *arXiv preprint arXiv:2011.14522*, 2020.