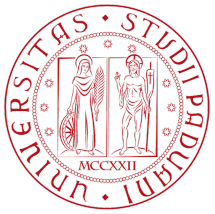


Università degli studi di Padova
Dipartimento di Fisica "Galileo Galilei"
Academic year 2019/2020

Master degree in Physics of Data

Notes of Machine Learning

Contents

What is Machine Learning?

The main subject o these notes is automated learning, or, as we will more often call it, Machine Learning (ML). Machine Learning is essentially "teaching to the computers so they can learn from inputs we give them". The input to a learning algorithm is training data, representing experience, and the output is some expertise, usually a new program that can independently perform a task.
A tipical example of a learning algorithm is a program that can distnguish between male and female people: it will be trained with a sample of different people (more heterogeneous as possible) to teach it the main differences of the two genres.
Another example is an algorithm that gives the probability of a person to have a certain height given, for example, the genre or the age.
A more common example is the algorithm that, between emails, can distinguish spam emails from the other ones.
Before starting the effective lectures, we should define a first small difference between the types of learning: since learning involves an interaction between the learner and the environment, one can divide learning tasks according to the nature of that interaction.

- *Supervised learning* describes a scenario in which the "experience", the training data, contains significant information that is missing in the unseen "test examples" to which the learned expertise is to be applied. In this setting, the acquired expertise is aimed to predict that missing information for the test data. In such cases, we can think of the environment as a teacher that "supervises" the learner by providing the extra information (labels). More abstractly, the algorithm can be seen as a process of "using experience to gain experties".

- In *Unsupervised learning*, instead, there is no distinction between training and test data: the learner processes input data with the goal of coming up with some summary, or compressed version of that data.

Chapter 1

Machine Learning Model

First of all, we give some definitions of the stuff the machine learnig algorithm has access to:

- **Domain set** (or **instance space** $\mathcal{X}$): is the set of all possible objects that we want to label, i.e. make prediction about. Usually this is a "vector of features".
- **Label set**: contains all possible prediction (labels). Usually correspond to the binary set $\{0, 1\}$ (that can mean $\{True, False\}$).
- **Training data**: $S = ((x_1, y_1), \ldots, (x_m, y_m))$ is a finite sequence of pairs in $\mathcal{X} \times \dagger$, and represent the input of the ML algorithm. The index m, instead, represent the dimension of the dataset. Despite the "set" notation, S is a sequence. In particular, the same example may appear twice in S and some algorithms can take into account the order of examples in S.
- **Prediction rule**: Is a function $h : \mathcal{X} \to \dagger$ (also called "predictor", "hypotheses" or "classifier") that can be used to predict the label of new domain points. We denote $A(S)$ the hypothesis that a learning algorithm A returns upon receiving the training sequence S.
- **Data-generation model**: We assume that the instances are generated by some probability distribution $\mathcal{D}$ (not known by the algorithm), and we assume that there is a "correct" labelling function $f : \mathcal{X} \to \dagger$ for which $y_i = f(x_i) \quad \forall i$. The labelling function isn't known by the algorithm: in fact, it is just what the program is trying to figure out. In summary, each pair in the training data is generated by first sampling a point x_i according to $\mathcal{D}$ and then labelling it by f.
- **Measure of success**: We define *error of the classifier* the probability that the algorithm does not predict the correct label on a random data point generated by distribution $\mathcal{D}$. In other words, the error of h i the probability to draw a random instance x, according to the distribution $\mathcal{D}$, such that $h(x) \neq f(x)$.

Usually, our sample will be a vector (so a set of numbers) $x \in \mathbb{R}^d$.
We must pay attention that what the algorithm learn depends on training data! So, if the training data are bad, the results will be bad aswell.

1.1 Measure of Success: Loss Function

Given a domain subset $A \subset \mathcal{X}$ and the probability distribution $\mathcal{D}$, $\mathcal{D}(A)$ represents the probability of observing a point $x \in A$.
In many cases, we refer to A as an *event* and express it using a function $\pi : \mathcal{X} \to \{0, 1\}$, that is $A = \{x \in \mathcal{X} : \pi(x) = 1\}$.
In this case denote $\mathcal{D}(A)$ as $\mathbb{P}_{x\sim\mathcal{D}}[\pi(x)]$, where $\sim$ indicates that x point is sampled according to $\mathcal{D}$.
Now, we define the **error of prediction rule**:

$$\ell_{\mathcal{D},f}(h) = \ell_{\mathcal{D}}(h) = \mathbb{P}_{x\sim\mathcal{D}}[h(x) \neq f(x)] = \mathcal{D}(\{x : h(x) \neq f(x)\}) \tag{1.1}$$

This is nothing but the probability of randomly choosing an example x for which $h(x) \neq f(x)$. It is also called *generalization error*, or **true error**.
It is important to understand that the algorithm doesn't know the probability distribution $\mathcal{D}$, so it is not able to compute the true error.

1.2 Empirical Risk Minimization

As mentioned earlier, a learning algorithm receives as input a training set S, sampled from an unknown distribution $\mathcal{D}$ and labeled by some target function f, and should output a predictor $h_S : \mathcal{X} \to \mathcal{Y}$ (the subscript S emphasizes the fact that the output predictor depends on the training set). The goal of the algorithm is to find h_S that minimizes the error with respect to the unknown $\mathcal{D}$ and f.
If, because of the true error can't be computed, we would like to try another estimate of it, we should define the error on the training data, called **training error**:

$$\ell_S(h) = \frac{|i : h(x_i) \neq y_i, i = 1, \ldots, m|}{m} \tag{1.2}$$

This division is simply the rate o correct prediction and the total samples (assuming classification problem and 0-1 loss), and it is also called *empirical error* or *empirical risk*.
This learning paradigm, which main goal is to find a predictor h that minimize ℓ_S, is called *Empirical Risk Minimization* or ERM, for short.

Although the ERM rule seems very natural, without being careful this approach may fail miserabily.
Let's consider, for example, the sample in figure n the left), assuming a probability distribution $\mathcal{D}$ for which instances x are taken uniformily at random in the square, and the labelling function f that assign the value 1 to the instances in the upper squares, and 0 to the others (essentially 0 to blue ones and 1 to red ones). If we collect the training set shown in figure n the right), to train the ML algorithm, a good predictor seems to be the function $h_S(x) = \begin{cases} 0 \text{ if x in left side} \\ 1 \text{ if x in right side} \end{cases}$, that effectively minimize the training loss ($L_S(h_S) = 0$, practically perfect). But is this a good predictor?
On the other hand, the true error of any classifier is, in this case $L_{\mathcal{D}}(h_S) = \frac{1}{2}$.
We have found a predictor whose performance on the training set is excellent, yet its performance on the true "world" is very poor. This phenomenon is called overfitting. Intuitively,

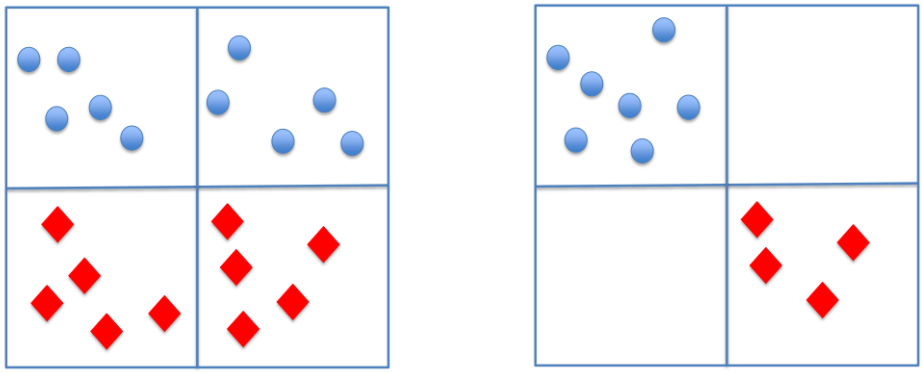


Figure 1.1: Example of overfitting

overfitting occurs when our hypothesis fits the training data "too well", but doesn't have the same performances on a generic dataset. We obviously want to avoid this overfitting, because algorithms should work in general cases. Overfitting is less probable when the training set is large. The bigger is a training set, the smaller will be the probability to have regular distribution of data.

1.3 Hypothesis class

Rather than giving up completely the ERM paradigm, we want to look at the conditions under which it is guarantee that ERM does not overfit.
A common solution is to apply the ERM learning rule over a restricted search space. Formally, the learner should choose in advance (before seeing the data) a set of predictors. This set is called a **hypothesis class** and is denoted by $\mathcal{H}$. Each $h \in \mathcal{H}$ is a function mapping from $\mathcal{X}$ to $\mathcal{Y}$. For a given class $\mathcal{H}$, and a training sample, S, the $ERM_{\mathcal{H}}$ learner uses the ERM rule to choose a predictor h, with the lowest possible error over S:

$$ERM_{\mathcal{H}} \in \underset{h \in \mathcal{H}`}{argmin}\, L_S(h) \tag{1.3}$$

where $argmin$ stands for the set of hypotheses in $\mathcal{H}$ that achieve the minimum value of $L_S(h)$. Since the choice of such a restriction is determined before the learner sees the training data, it should ideally be based on some prior knowledge about the problem to be learned. Intuitively, choosing a more restricted hypothesis class better protects us against overfitting but at the same time might cause us a stronger inductive bias: if the class is too small you will not find a good estimator.
Now we would like to prove that, if $\mathcal{H}$ is a finite class $ERM_{\mathcal{H}}$ is guarantee not to overfit, provided it is based on a sufficiently large training sample (depending on the size of $\mathcal{H}$). Limiting the dimension of $\mathcal{H}$ ($|\mathcal{H}| < \infty$) seems not to be very realistic, because we usually have to deal with parameters defined in $\mathbb{R}^d$: the correct assumption uses the finite precision with which computers manage real numbers.
Let us now analyze the performance of the $ERM_{\mathcal{H}}$ learning rule assuming that $\mathcal{H}$ is a finite class. For a training sample, S, labeled according to some $f : \mathcal{X} \to \mathcal{Y}$, let h_S denote a result of applying $ERM_{\mathcal{H}}$ to S, i.e. $h_s \in \underset{h \in \mathcal{H}}{argmin}\, L_S(h)$. Let's make some assumptions:

- (**Realizability**) There exist a $h^* \in \mathcal{H}$ such that $L_{\mathcal{D}}(h^*) = 0$: in practice we are asking for the existance of the perfect solution in our hypothesis class. Note that this assumption implies that with probability 1 over random samples, we have $L_S(h^*) = 0$.
- (**i.i.d.**) The examples in the training set are indipendently and identically distributed (i.i.d.) according to the distribution $\mathcal{D}$ and than labelled according the labeling function f. We denote this assumption by $S \sim \mathcal{D}^m$ (this notation represents the fact that we sample m times according to the distribution $\mathcal{D}$). This assumption seems to be unrealistic too: the sampling is usually biased on the situation; you will never be able to sample reproducing the real distribution and leave them indipendent between each other. Anyway, the larger the sample get, the more likely it is to reflect more accurately the distribution ad labelig used to generate it.

Since $L_{\mathcal{D}}(h_S)$ depends on the training set S, and that training set is picked up in a random process, there is a randomness in the choice of the predictor h_S and, consequently, in the risk $L_{\mathcal{D}}(h_S)$. It's never guarantee that the solution we find is the perfect one, because there is always some probability that the sampled training data happens to be very nonrepresentative of the underlying $\mathcal{D}$.
Usually we denote the probability o getting a nonrepresentative sample by δ, and call $(1-\delta)$ the **confidence parameter** of our prediction.
On the top of that, since we cannot guarantee a perfect label prediction, we introduce another parameter, the **accuracy parameter**, commonly denoted by ε.
So, we interpret the event $L_{\mathcal{D}}(h_S) > \varepsilon$ as a failure of the learner, while if $L_{\mathcal{D}}(h_S) \leq \varepsilon$ we view the output of the algorithm as an approximately correct predictor.

Theorem 1.1. *Let $\mathcal{H}$ be a finite hypothesis class. Let $\delta \in (0,1)$, $\varepsilon \in (0,1)$, and $m \in \mathbb{N}$ (size of the training set, i.e. S contains m i.i.d. samples) such that*

$$m \geq \frac{\log(|\mathcal{H}|/\delta)}{\varepsilon}$$

Then, for any f and any $\mathcal{D}$ for which the realizability assumption holds, with probability $\geq 1-\delta$ we have that for every ERM hypothesis h_S it holds that

$$L_{\mathcal{D},f}(h_S) \leq \varepsilon$$

Proof. Denote with P_{good} the probability to find a "good" h_S, such that $L_{\mathcal{D},f}(h_S) \leq \varepsilon$. We want to probe that $P_{good} \geq 1-\delta$, which is equivalent to have a probability $P_{bad} \leq \delta$ (with $P_{bad} = 1 - P_{good}$). The idea is to consider the set of all the possible m-dimensional training samples: into this set there are some samples that are "misleading", meaning that they result in a $L_{\mathcal{D},f}(h_S) \geq \varepsilon$, while the other lead to $L_{\mathcal{D},f}(h_S) \leq \varepsilon$ that we want. The essence of the proof relies in finding a bound on these "misleading samples" size.
Formally, let $S|_x = (x_1, \ldots, x_m)$ be the instances of the training set. We would like to upper bound the probability to sample from $\mathcal{D}$ a m-tuple which leads to a generalization error bigger than ε:

$$P_{bad} = \mathcal{D}^m(\{S|_x : L_{\mathcal{D},f}(h_S) > \varepsilon\})$$

Then, the set of "bad" hypothesis is

$$\mathcal{H}_B = \{h \in \mathcal{H} : L_{\mathcal{D},f}(h) > \varepsilon\}$$

The set of "misleading samples", that contains all the m-tuples which lead to a bad hypothesis after applying the ERM algorithm is:

$$M = \{S|_x : \exists h \in \mathcal{H}_B, L_S(h) = 0\} = \bigcup_{h \in \mathcal{H}_B} \{S|_x : L_S(h) = 0\}$$

Namely, for every $S|_x \in M$, there is a "bad" hypothesis, $h \in \mathcal{H}_B$, that looks like a "good" hypothesis on $S|_x$.
Now, let's note that

$$\mathcal{D}^m\left(\{S|_x : L_{\mathcal{D},f}(h_S) > \varepsilon\}\right) \leq \mathcal{D}^m(M) = \mathcal{D}^m\left(\bigcup_{h \in \mathcal{H}_B} \{S|_x : L_S(h) = 0\}\right)$$

Next, we upper bound the right-hand side of the preceding equation using the *union bound*:

$$\mathcal{D}(A \cup B) \leq \mathcal{D}(A) + \mathcal{D}(B) \tag{1.4}$$

In fact, if A and B where disjoint, then $\mathcal{D}(A \cup B) = \mathcal{D}(A) + \mathcal{D}(B)$. However, if $A \cap B \neq \emptyset$, then $\mathcal{D}(A \cap B) < \mathcal{D}(A) + \mathcal{D}(B)$. This can be proved more formally, but we will not do that here. Applying the union bound, we find the relation

$$\mathcal{D}^m\left(\{S|_x : L_{\mathcal{D},f}(h_S) > \varepsilon\}\right) \leq \sum_{h \in \mathcal{H}_B} \mathcal{D}^m\left(\{S|_x : L_S(h) = 0\}\right)$$

All the ERM solutions are "perfect" when evaluated on the training set, meaning that they correctly classify all the samples (that, we have to remember, are all i.i.d.):

$$\mathcal{D}^m\left(\{S|_x : L_S(h) = 0\}\right) = \mathcal{D}^m\left(\{S|_x : \forall i, h(x_i) = f(x_i)\}\right) = \prod_{i=1}^{m} \mathcal{D}^m\left(\{x_i : h(x_i) = f(x_i)\}\right)$$

For each individual sampling of an element of the training set we have

$$\mathcal{D}^m\left(\{S|_x : h(x_i) = f(x_i)\}\right) = 1 - L_{\mathcal{D},f}(h) \leq 1 - \varepsilon$$

where the last inequality follows from the fact that $h \in \mathcal{H}_B$ (so $L_{\mathcal{D},f}(h) > \varepsilon$). Combining the previous relations and using the inequality $1 - \varepsilon \leq e^{-\varepsilon}$ we obtain that for every $h \in \mathcal{H}_B$:

$$\mathcal{D}^m\left(\{S|_x : L_S(h) = 0\}\right) \leq \prod_{i=1}^{m} (1 - \varepsilon) = (1 - \varepsilon)^m \leq e^{-\varepsilon m}$$

So we can conclude that:

$$P_{bad} = \mathcal{D}^m(\{S|_x : L_{\mathcal{D},f}(h_S) > \varepsilon\}) \leq \sum_{h \in \mathcal{H}_B} e^{-\varepsilon m} = |\mathcal{H}_B| e^{-\varepsilon m} \underset{\mathcal{H}_B \subset \mathcal{H}}{\leq} |\mathcal{H}| e^{-\varepsilon m}$$

Finally, we arrived at the expression:

$$P_{bad} \leq |\mathcal{H}| e^{-\varepsilon m} \overset{!}{\leq} \delta$$

We then find a bound on m, by taking the *log* of both sides:

$$e^{-\varepsilon m} \leq \frac{\delta}{|\mathcal{H}|} \implies -\varepsilon m \leq \log\left(\frac{\delta}{|\mathcal{H}|}\right) \implies m \geq -\frac{1}{\varepsilon} \log\left(\frac{\delta}{|\mathcal{H}|}\right) \implies m \geq \frac{1}{\varepsilon} \log\left(\frac{|\mathcal{H}|}{\delta}\right)$$

□

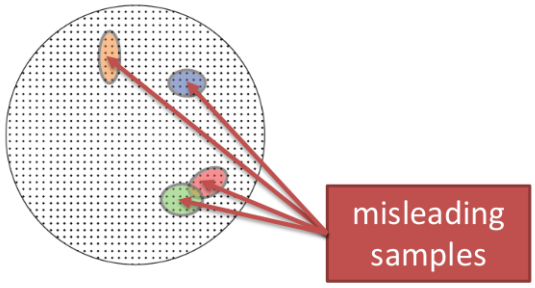


Figure 1.2: Each point in the large circle represents a possible m-tuple of instances. Each colored oval represents the set of "misleading" m-tuple of instances for some "bad" predictor $h \in \mathcal{H}_B$. The ERM can potentially overfit whenever it gets a misleading training set S. That is, for some $h \in \mathcal{H}_B$ we have $L_S(h) = 0$. The disequations found in the demostration of the theorem guarantee that for each individual bad hypothesis, $h \in \mathcal{H}_B$, atmost $(1-\varepsilon)^m$-fraction of the training sets would be misleading. In particular, the larger m is, the smaller each of these colored ovals becomes. The union bound formalizes the fact that the area representing the training sets that are misleading with respect to some $h \in \mathcal{H}_B$ (that is, the training sets in M) is atmostthe sum of the areas of the colored ovals. Therefore, it is bounded by $\mathcal{H}_B$ times the maximum size of a colored oval. Any sample S outside the colored ovals cannot cause the ERM rule to overfit.

This theorem tell us that, for a sufficiently large m, the $ERM_{\mathcal{H}}$ rule over a finite hypothesis class will be *probably* (with confidence $1-\delta$) *approximately* (up to an error of ε) correct.
Notes on the theorem:

- $|\mathcal{H}|$ is the cardinality (i.e. the dimension) of the hypothesis class
- The condition $m \geq \frac{\log(|\mathcal{H}|/\delta)}{\varepsilon}$ is just a sufficient (not necessary) condition. In fact it is a nearly weak request. So it is possible (and indeed happens) to have a good learner even with smaller (m lower than the bound) training datasets.
- m does not depend on f or on $\mathcal{D}$.

Chapter 2

PAC Learning

The title of this chapter stands for *Probably Approximately Correct* learning: since the data are sampled accordingly to $\mathcal{D}$, and since we can't find the perfect machine learning algorithm:

- we can only be approximately correct (accuracy parameter ε: we are satisfied with a good h_S for which $L_{\mathcal{D},f}(h_S) \leq \varepsilon$)
- we can only be probably correct (we want h_S to be a good hypothesis with probability $\geq 1 - \delta$)

Definition 2.1. ***(PAC learnability)***
A hypothesis class $\mathcal{H}$ is PAC learnable if there exist a function $m_{\mathcal{H}} : (0,1)^2 \to \mathbb{N}$ and a learning algorithm such that for every $\delta, \varepsilon \in (0,1)$, for every distribution $\mathcal{D}$ over $\mathcal{X}$, and for every labeling function $f : \mathcal{X} \to \{0,1\}$, if the realizability assumption holds with respect to $\mathcal{H}, \mathcal{D}, f$, then when running the learning algorithm on $m \geq m_{\mathcal{H}}(\varepsilon, \delta)$ i.i.d. examples generated by $\mathcal{D}$ and labeled by f, the algorithm returns a hypothesis h such that, with probability $\geq 1 - \delta$ (over the choice of examples): $\ell_{\mathcal{D},f}(h) \leq \varepsilon$.

Note: $m_{\mathcal{H}} : (0,1)^2 \to \mathbb{N}$ is the sample complexity of learning $\mathcal{H}$, in particular is the minimal integer that satisfies the requirements. This means that if we have an infinite number of possible hypotesis we cannot apply the theorem, but, at the same time, we can't be sure that the set is not learnable.

Corollary 2.0.1. *Every finite (sufficient condition, not necessary, as said before) hypothesis class is PAC learnable with sample complexity*

$$m_{\mathcal{H}}(\varepsilon, \delta) \leq \left\lceil \frac{\log(|\mathcal{H}|/\delta)}{\varepsilon} \right\rceil$$

The definition of PAC learnability contains two approximation parameters. The accuracy parameter ε determines how far the output classifier can be from the optimal one and a confidence parameter δ indicating how likely the classifier is tomeet that accuracy requirement.

Now we would like to generalize this model, and we can do it dropping some assumptions we have made before:

- We remove the realizability assumption: it was interesting from a theorical point of view but too strong in many real world applications. Rejecting this means that there exist $h^* \in \mathcal{H}$ such that $L_{\mathcal{D},f}(h) = 0$.

- In many application it is not too realistic that the labeling is fully determined by the features we measure; for this reason it is convenient to replace the function f with something more flexible, for example assuming that $\mathcal{D}$ is a probability distribution over $\mathcal{X} \times \mathcal{Y}$ (i.e. the joint distribution over domain points and labels). One can view such a distribution as being composed of two parts: a distribution $\mathcal{D}_x$ over unlabeled domain points (*marginal distribution*) and a *conditional* probability over labels for each domain point $D((x, y)|x)$.

2.1 The empirical and the true error

For a probability distribution $\mathcal{D}$ over $\mathcal{X} \times \mathcal{Y}$ we redefine the true error (or risk) of a prediction rule h to be

$$L_{\mathcal{D}}(h) = \mathbb{P}_{(x,y)\sim\mathcal{D}}[h(x) \neq y] = \mathcal{D}([(x, y) : h(x) \neq y]) \tag{2.1}$$

Note: in the previous definition e request was $[h(x) \neq f(x)]$ but, as we said before, the function $f(x)$ doesn't exist anymore because it was a too strong assumption asking that the labels were fully determined by a single function.
The definition of the empirical risk, instead, remains the same as before

$$L_S(h) = \frac{[i \in [m] : h(x_i) \neq y_i]}{m}$$

and still represent the probability that, for a pair (x_i, y_i) taken uniformily at random training data, the event "$h(x_i) \neq y_i$" holds.

2.2 The Bayes optimal predictor

Given any probability distribution $\mathcal{D}$ over $\mathcal{X} \times [0, 1]$, the best label predicting function (the one that minimize $L_{\mathcal{D}}(h)$) will be

$$f_{\mathcal{D}}(x) = \begin{cases} 1 & \text{if } \mathbb{P}[y = 1|x] \geq 1/2 \\ 0 & \text{otherwise} \end{cases} \tag{2.2}$$

It's easy to verify that the predictor $f_{\mathcal{D}}$ is optimal, in the sense that any other classifier $g : \mathcal{X} \to [0, 1]$, has a lower error, $L_{\mathcal{D}}(f_{\mathcal{D}}) \leq L_{\mathcal{D}}(g)$.
This optimal predictor is interensting from a theorical point of view, but unfortunately is not usable in practice because e don't know the distribution $\mathcal{D}$ and, consequently $\mathbb{P}[y = 1|x]$.

2.3 Agnostic PAC Learnability

Clearly, we cannot hope that the learning algorithm will find a hypothesis whose error is smaller than the minimal possible error, that of the Bayes predictor. Furthermore, as we shall prove later, once we make no prior assumptions about the data-generating distribution, no algorithm can be guaranteed to find a predictor that is as good as the Bayes optimal one. Instead, we require that the learning algorithm will find a predictor whose error is not much larger than the best possible error of a predictor in some given benchmark hypothesis class. Of course, the strength of such a requirement depends on the choice of that hypothesis class.

Definition 2.2. *(**Agnostic PAC Learnability**)*
A hypothesis class $\mathcal{H}$ is agnostic PAC learnable if there exist a function $m_{\mathcal{H}} : (0,1)^2 \to \mathbb{N}$ and a learning algorithm such that for every $\delta, \varepsilon \in (0,1)$ and for every distribution $\mathcal{D}$ over $\mathcal{X} \times \mathcal{Y}$, when running the algorithm on $m \geq m_{\mathcal{H}}(\varepsilon, \delta)$ i.i.d. examples generated by $\mathcal{D}$ the algorithm returns a hypothesis h such that, with probability $\geq 1 - \delta$ (over the choice of the m training examples):

$$L_{\mathcal{D}}(h) \leq \min_{h' \in \mathcal{H}} L_{\mathcal{D}}(h') + \varepsilon$$

Notes on the definition:

- The agnostic PAC learning generalizes the definition of PAC learning, in the sense that, if the realizability assumption holds, this new definition provides the same guarantee as the previous one.
- The realizability assumption would implies that the quantity $\min_{h' \in \mathcal{H}} L_{\mathcal{D}}(h')$, that represents the performance of the best possible classifier (for example the Bayes' one), is always equal to zero.

Anyway 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.

We now want to extend our model so that it can be applied to a wide variety of learning tasks. Let's consider 3 different possible problems:

- *Binary and Multiclass classification*: our training sample will be a finite sequence of (feature vector,label) pairs, the learner's output will be a function from the domain set to the label set, and, finally, for our measure of success, we can use the probability, over pairs, of the event that our predictor suggests a wrong label.
- *Regression*: in this task, one wishes to find some simple pattern in the data, a function between $\mathcal{X}$ and $\mathcal{Y} = \mathbb{R}$. For the loss function, we can't use the same of the previous case, we need a new one.

2.3.1 Generalized loss function

Given any set $\mathcal{H}$ (that plays the role of our hypotheses, or models) and some domain Z let ℓ be any function from $\mathcal{H} \times Z$ to the set of nonnegative real numbers, $\ell : \mathcal{H} \times Z \to \mathbb{R}+$. We call such functions loss functions.
We now define the **risk function** to be the expected loss of a classifier $h \in \mathcal{H}$, with respect to a probability distribution $\mathcal{D}$ over Z, namely:

$$L_{\mathcal{D}}(h) = \mathbb{E}_{z \sim \mathcal{D}} \left[\ell(h, z) \right] \tag{2.3}$$

Similarly, we define the **empirical risk** to be the expected loss over a given sample $S = (z_1, \ldots, z_m) \in Z^m$, namely:

$$L_S(h) = \frac{1}{m} \sum_{i=1}^{m} \ell(h, z_i) \tag{2.4}$$

We must keep in mind that the loss always depends on the problem, there isn't a universal one. Sometimes it is useful to create a "personalized" loss function.
Common loss functions:

- *0-1 loss*: commonly use in binary or multiclass classification (obviously is not the only possible solution, is just the simpler and most common one).

$$\ell_{0-1}(h,(x,y)) = \begin{cases} 0 & \text{if } h(x) = y \\ 1 & \text{if } h(x) \neq y \end{cases}$$

- *Squared loss L2*: commonly used in regression, penalize few large errors.

$$\ell_{sq}(h,(x,y)) = (h(x) - y)^2$$

- *Absolute value loss L1*: commonly used in regression, penalize many small errors:

$$\ell_{abs}(h,(x,y)) = |h(x) - y|$$

Example 2.1. *The loss function depends on our problem! Let's image a machine learning algorithm that verify fingerprints to guarantee the access to something. There are essentially two types of error that the computer can do:*

- *False accept: when accepts an unauthorized user.*
- *False reject: when doesn't accept an athorized user.*

If, for example, a supermarket implement the algorithm to give discounts:

- *False reject is costly; just the customer gets annoyed.*
- *False accept is minor, the supermarket lose just a discount and the intruder left his fingerprints.*

If, instead, a similar algorithm has been implemented by CIA for the security of a certain area:

- *False reject can be tolerade.*
- *False accept is a disaster!*

This example want to show that often the loss function need to be "calibrated" by our knowledge on the problem, to understand which errors are tolerable and which are not.

Definition 2.3. *(**Agnostic PAC Learnability for General Loss Functions**)*
Recalling the definition 2.2, for which the hypothesis generate by the algorithm (with probability $\geq 1 - \delta$ over the training set) has a true error

$$L_{\mathcal{D}}(h) \leq \min_{h' \in \mathcal{H}} L_{\mathcal{D}}(h') + \varepsilon$$

we can implement the definition we have written for the loss function deducing that:

$$L_{\mathcal{D}}(h) = \mathbb{E}_{z \sim \mathcal{D}} \left[\ell(h, z) \right]$$

Chapter 3

Learning from Uniform Convergence

Recall that, given a hypothesis class $\mathcal{H}$, the ERM learning paradigm works as follow: upon receiving a training sample S, the learner evaluates the error of each $h \in \mathcal{H}$ on the given sample and outputs a member of the class that minimizes this empirical risk. The hope is that an h that minimizes the empirical risk with respect to S is a risk minimizer with respect to the true data probability distribution aswell. For this reason we want that the empirical error is a good approximation of the true error for all the solutions, and not only for the best one ($L_S(h)$ is similar to $L_{\mathcal{D}}(h)$, $\forall\, h$).

Definition 3.1. *(**ε-representative**)*
A training set S is called ε-representative (with respect to domain Z, hypothesis class $\mathcal{H}$, loss function ℓ and distribution $\mathcal{D}$) if

$$\forall h \in \mathcal{H} \quad |L_S(h) - L_D(h)| \leq \varepsilon$$

This is a stronger request than previous ones because we do not more focus on the best hypothesis of the class.

Theorem 3.1. *Assume that the training set S is $\frac{\varepsilon}{2}$-representative. Then, any output of $ERM_{\mathcal{H}}(S)$ (i.e. any $h_S \in \underset{h\in\mathcal{H}}{argmin}\, L_S(h)$) satisfies:*

$$L_{\mathcal{D}}(h_S) \leq \min_{h\in\mathcal{H}} L_{\mathcal{D}}(h) + \varepsilon$$

Proof. For every $h \in \mathcal{H}$,

$$L_{\mathcal{D}}(h_S) \leq L_S(h_S) + \frac{\varepsilon}{2} \leq L_{\mathcal{D}}(h) + \frac{\varepsilon}{2} + \frac{\varepsilon}{2} = L_{\mathcal{D}}(h) + \varepsilon$$

where the first and the third inequalities are due to the assumption that S is $\frac{\varepsilon}{2}$- and the second inequality holds since h_S is an ERM predictor. □

The relation guarantee by the theorem is exactly the one required by the definition of agnostic PAC learnability: in fact, the consequence of this statement is that if, with probability at least $1-\delta$, a random training set S is ε-representative, then the ERM rule is an agnostic PAC learner.

Definition 3.2. *(**Uniform Convergence**)*
A hypothesis class $\mathcal{H}$ has the uniform (same m for all h and all $\mathcal{D}$) convergence property (with respect to a domain Z and a loss function ℓ) if there exists a function $m^{UC}_{\mathcal{H}} : (0,1)^2 \to \mathbb{N}$ such that for every $\varepsilon, \delta \in (0,1)$ and for every probability distribution $\mathcal{D}$ over Z, if S is a sample of $m \geq m^{UC}_{\mathcal{H}}(\varepsilon, \delta)$ i.i.d. examples drawn from $\mathcal{D}$, then with probability $\geq 1-\delta$, S is ε-representative.

This means that the function $m_{\mathcal{H}}^{UC}$ measures the (minimal) sample complexity of obtaining the uniform convergence property, namely, how many examples we need to ensure that with probability at least $1-\delta$ the sample would be ε-representative.

Proposition 3.1. *If a class $\mathcal{H}$ has the uniform convergence property with a function $m_{\mathcal{H}}^{UC}$ then the class is agnostically PAC learnable with the sample complexity $m_{\mathcal{H}}(\varepsilon,\delta) \leq m_{\mathcal{H}}^{UC}(\varepsilon/2,\delta)$. Furthermore, in that case the $ERM_{\mathcal{H}}$ paradigm is a successful agnostic PAC learner for $\mathcal{H}$.*

Proposition 3.2. *Let $\mathcal{H}$ be a finite hypothesis class, let Z be a domain, and let $\ell : \mathcal{H} \times Z \to [0,1]$ be a loss function. Then:*

- *$\mathcal{H}$ enjoys the uniform convergence property with sample complexity*

$$m_{\mathcal{H}}^{UC}(\varepsilon,\delta) \leq \left\lceil \frac{\log(2|\mathcal{H}|/\delta)}{2\varepsilon^2} \right\rceil$$

- *$\mathcal{H}$ is agnostically PAC learnable using the ERM algorithm with sample complexity*

$$m_{\mathcal{H}}(\varepsilon,\delta) \leq m_{\mathcal{H}}^{UC}(\varepsilon/2,\delta) \leq \left\lceil \frac{2\log(2|\mathcal{H}|/\delta)}{\varepsilon^2} \right\rceil$$

Proof. Fix some ε, δ. We need to find a sample size m that guarantes that for any $\mathcal{D}$, with probability at least $1-\delta$ of the choice of $S = (z_1, \ldots, z_m)$ sampled i.i.d. from $\mathcal{D}$ we have that for all $h \in \mathcal{H}$, $|L_S(h) - L_{\mathcal{D}}(h)| \leq \varepsilon$ (so we are asking S to be ε-representative for some m). Namely

$$\mathcal{D}^m\left(\{S : \forall h \in \mathcal{H}, |L_S(h) - L_{\mathcal{D}}(h)| \leq \varepsilon\}\right) \geq 1-\delta$$

Equivalently, we need to show that the probability of *not* having uniform convergence is

$$\mathcal{D}^m\left(\{S : \exists h \in \mathcal{H}, |L_S(h) - L_{\mathcal{D}}(h)| > \varepsilon\}\right) \leq \delta$$

In the expression above, I can rewrite the set as the union over h:

$$\{S : \exists h \in \mathcal{H}, |L_S(h) - L_{\mathcal{D}}(h)| > \varepsilon\} = \bigcup_{h \in \mathcal{H}} \{S : |L_S(h) - L_{\mathcal{D}}(h)| > \varepsilon\}$$

and, applying the Union Bound (1.4), we obtain

$$\mathcal{D}^m\left(\{S : \exists h \in \mathcal{H}, |L_S(h) - L_{\mathcal{D}}(h)| > \varepsilon\}\right) \leq \sum_{h \in \mathcal{H}} \mathcal{D}^m\{S : |L_S(h) - L_{\mathcal{D}}(h)| > \varepsilon\}$$

The next step will be to show that for any fixed hypothesis h, the gap between the true and the empirical risk is likely to be small (for a sufficiently large m).
Now, recall that $L_{\mathcal{D}}(h) = \mathbb{E}_{z\sim\mathcal{D}}\left[\ell(h,z)\right]$ (2.3) and that $L_S(h) = \frac{1}{m}\sum_{i=1}^m \ell(h,z_i)$ (2.4). Since each z_i is sampled i.i.d. from $\mathcal{D}$, the expected value of the random variable $\ell(h,z_i)$ is $L_{\mathcal{D}}(h)$. By the linearity of expectation, it follows that $L_{\mathcal{D}}(h)$ is also the expected value of $L_S(h)$. Hence, the quantity $|L_S(h) - L_{\mathcal{D}}|$ is the deviation of the random variable $L_S(h)$ from its expectation. A basic statistical fact, the law of large numbers, states that when m goes to infinity, empirical averages converge to their true expectation, and this is also valid for $L_S(h)$, since it is the empirical average of m i.i.d. samples.
However, this low is only an asymptotic result, and it provides no information about the gap between empirical averages and their expected value: to quantify this gap we will use a measure concentration inequality due to Hoeffding:

Lemma 3.1. ***(Hoeffding's Inequality)***
Let $\theta_1, dots, \theta_m$ be a sequence of i.i.d. random variables and assume that for all i, $\mathbb{E}[\theta_i] = \mu$ and $\mathbb{P}[a \leq \theta_i \leq b] = 1$. Then, for any $\varepsilon > 0$:

$$\mathbb{P}\left[\left|\frac{1}{m}\sum_{i=1}^{m}\theta_i - \mu\right| > \varepsilon\right] \leq 2\exp\left(-\frac{2m\varepsilon^2}{(b-a)^2}\right)$$

Getting back to our problem, let θ_i be the random variable $\ell(h, z_i)$. Since h is fixed and $z_1, \ldots, z_m$ are sample i.i.d., it follows that $\theta_1, \ldots, \theta_m$ are also i.i.d. random variables. Furthermore, $L_S(h) = \frac{1}{m}\sum_{i=1}^{m}\theta_i$ and $L_{\mathcal{D}}(h) = \mu$. Let us further assume that the range of ℓ is $[0, 1]$. We therefore obtain that

$$\mathcal{D}^m\left\{S : |L_S(h) - L_{\mathcal{D}}(h)| > \varepsilon\right\} = \mathbb{P}\left[\left|\frac{1}{m}\sum_{i=1}^{m}\theta_i - \mu\right| > \varepsilon\right] \leq 2\exp\left(-2m\varepsilon^2\right)$$

Applying this to the sum over h:

$$\sum_{h\in\mathcal{H}}\mathcal{D}^m\left\{S : |L_S(h) - L_{\mathcal{D}}(h)| > \varepsilon\right\} \leq |\mathcal{H}|2e^{-2m\varepsilon^2}$$

Now, we need to find m for which the term on the right above is smaller or equal than δ, so we fix

$$m \geq \frac{\log(2|\mathcal{H}|/\delta)}{2\varepsilon^2}$$

So that

$$\mathcal{D}^m\left\{S : |L_S(h) - L_{\mathcal{D}}(h)| > \varepsilon\right\} \leq \delta$$

□

3.0.1 The Discretization Trick

While the previous theorem only applies on finite hypothesis classes, there is a simple trick that allow us to get a very good estimate of the practical sample complexity of infinite hypothesis classes. In many real world application, in fact, we consider classes determined by a set of parameters in $\mathbb{R}$.
The solution arises when we use computers. Using a pc, in fact, we will probably maintain real numbers using floating point representation, say, of 64 bits. It follows that, in practice, our hypothesis class is parameterized by the set of scalar that can be represented using a 64 bits floating point number. There are at most 2^{64} such numbers; hence the actual size of our hypothesis class is at most 2^{64}. More generally, if our hypothesis class is parameterized by d numbers, in practice we learn an hypothesis class of size at most 2^{64d}.
Applying the proposition, we obtain that the sample complexity of such classes is bounded by

$$m_{\mathcal{H}}(\varepsilon, \delta) \leq m_{\mathcal{H}}^{UC}(\varepsilon/2, \delta) \leq \frac{2\log\left(2\frac{2^{64d}}{\delta}\right)}{\varepsilon^2}$$

This upper bound on the sample complexity has the deficiency of being dependent on the specific representation of real numbers used by our machine.

Chapter 4

The Bias-Complexity Trade-Off

In one of the previous chapters we saw that, unless one is careful, the training data can mislead the learner, and result in overfitting. To overcome this problem, we restricted the search space to some hypothesis class $\mathcal{H}$. Such an hypothesis class can be viewed as reflecting some prior knowledge that the learner has about the task – a belief that one of the members of the class $\mathcal{H}$ is a low-error model for the task.
Our main objective was, given a training set S and a loss function, to find a function $\hat{h}$ for which the error $L_{\mathcal{D}}(\hat{h})$ is small (that brought to the definition of Agnostic PAC learnability (2.2)). For this reason we needed a large hypothesis class to contain the best solution, but at the same time a good algorithm to find it.
But, is there some kind of universal learner, that is, a learner who has no prior knowledge about a certain task and is ready to be challenged by any task, that predicts the best $\hat{h}$ for any distribution $\mathcal{D}$?
So, what about using the set of *all* the functions from $\mathcal{X}$ to $\mathcal{Y}$ as the hypothesis class? With this assumption we would be sure that the solution (if the problem is solvable) is inside the class.

Theorem 4.1. ***(No-Free Lunch)***
Let A be a learning algorithm for the task of binary classification with respect to the 0-1 loss over a domain $\mathcal{X}$. Let m be any number smaller than $|\mathcal{X}|/2$, representing a training set size. Then, there exists a distribution $\mathcal{D}$ over $\mathcal{X} \times \{0,1\}$ such that:

- *there exists a function $f : \mathcal{X} \to \{0,1\}$ with $L_{\mathcal{D}}(f) = 0$ (so a perfect solution with 0 loss)*
- *with probability of at least $1/7$ over the choice $S \sim \mathcal{D}^m$ we have that $L_{\mathcal{D}}(A(S)) \geq 1/8$.*

This means that there isn't a universal learner: the algorithm can fail even if there is a perfect solution for the training set. In fact, the theorem guarantees the existance of this perfect solution (with 0 loss), but doesn't guarantee a good performance instead.
The key message is that for every ML algorithm there exist a task on which it fails even if another ML algorithm is able to solve it.

Idea of the proof: our training set is smaller than half of the domain. This means that we have no information on what happens on the other half, but we can suppose that there is some target function f that works on the other half in a way that contradicts our estimated labels.

Let's consider an ERM predictor over the hypothesis class of all the functions $f : \mathcal{X} \to \{0,1\}$. This class represent a lack of knowledge: every possible function is considered a good candidate.

According to the previous theorem, any algorithm that chooses its output from an hypothesis belonging to that class, and in particular the ERM predictor, will fail on some learning task. Therefore, the class is not PAC learnable, as formalized in the following corollary.

Corollary 4.1.1. *Let $\mathcal{X}$ be an infinite domain set and let $\mathcal{H}$ be the set of all the functions from $\mathcal{X}$ to $\{0,1\}$. Then, $\mathcal{H}$ is not PAC learnable.*

Proof. Let's assume, by way of contradiction, that the class is learnable and let's choose some $\varepsilon < 1/8$ and $\delta < 1/7$. By the definition of PAC learnability (2.1), it exists an algorithm A and an integer $m = m(\varepsilon, \delta)$, such that for any data-generating distribution over $\mathcal{X} \times \{0,1\}$, if for some function f in that domain, $L_{\mathcal{D}}(f) = 0$, then with probability $\geq 1 - \delta$ when A is applied to samples S of size m, generated i.i.d. by $\mathcal{D}$, $L_{\mathcal{D}}(A(S)) \leq \varepsilon$.
However, applying the Free-Lunch theorem, since $|\mathcal{X}| > 2m$, for every learning algorithm (and in particular for A), there exists a distribution $\mathcal{D}$ such that with probability greater than $1/7 > \delta$, $L_{\mathcal{D}}(A(S)) > 1/8 > \varepsilon$, which leads to the desired contradiction. □

But how can we prevent such failures? We should, for example, use our knowledge on a specific learning task to avoid distributions that will cause us to fail.
But how should we choose a good hypothesis class? On the other hand we want to believe that this class includes the hypothesis that has no error at all (in the PAC setting), or at least that the smallest error achievable by a hypothesis from this class is indeed rather small (in the agnostic setting). On the other hand, we have just seen that we cannot simply choose the richest class i.e. the class of all functions over the given domain.

- If we choose a *small* $\mathcal{H}$ we will deal with *low approximation capabilities* (large L_S) but *good generalization properties* ($L_{\mathcal{D}} \sim L_S$). So the approximation found will work similarly on real data as on the training set, but it's not guarantee that the best hypothesis is in the class.
- If we choose a *large* $\mathcal{H}$, instead, we will have *good approximation capabilities* (small L_S) but there is a bigger *risk of overfitting* ($L_{\mathcal{D}} \gg L_S$). Moreover the no free lunch theorem suggests to not have too larger hypothesis classes.

4.1 Error Decomposition

To answer these questions we decompose the true error of an $ERM_{\mathcal{H}}$ predictor, h_S into two components, as follows:

$$L_{\mathcal{D}}(h_S) = \varepsilon_{app} + \varepsilon_{est}$$

- **Approximation error**

  $$\varepsilon_{app} = \min_{h \in \mathcal{H}} L_{\mathcal{D}}(h)$$

  It is the minimum risk achievable by a predictor in the hypothesis class. It does not depend on the training set, or on the sample size, but only on the choice of the class. Enlarging the hypothesis class can decrease the approximation error. Under the realizability assumption, this error is zero.

- **Estimation error**

  $$\varepsilon_{est} = L_{\mathcal{D}}(h_S) - \min_{h \in \mathcal{H}}(h)$$

It is the difference between the approximation error and the error achieved by the ERM predictor. The estimation error results because the training error is only an estimate of the true risk, and so the predictor minimizing the empirical risk is only an estimate of the predictor minimizing the true risk. This type of error depends on the training set size and on the size, or complexity, of the hypothesis class. For a finite class, the estimation error increases (logarithmically) with $|\mathcal{H}|$ and decreases with m (so the training error becomes a good estimate of the true error).

The following pictures schematized what we just said regarding the two types of error.

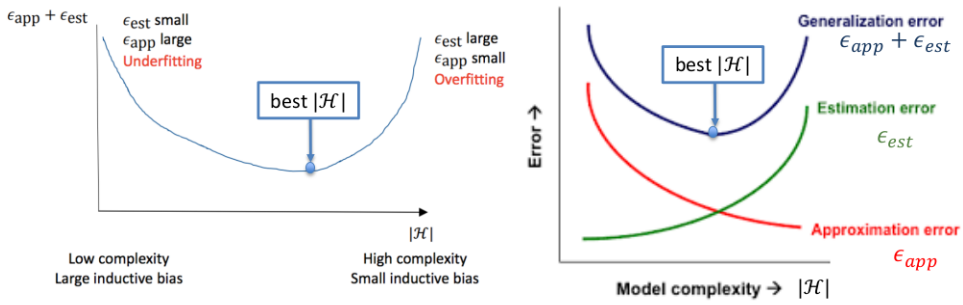


Figure 4.1: Error decomposition

Since our goal is to minimize the total risk, we face a *tradeoff*, called the **bias complexity tradeoff**. On one hand, choosing $\mathcal{H}$ to be a very rich class decreases the approximation error but at the same time might increase the estimation error, as a rich $\mathcal{H}$ might lead to *overfitting*. On the other hand, choosing $\mathcal{H}$ to be a very small set reduces the estimation error but might increase the approximation error or, in other words, might lead to *underfitting*.
The differences between overfitting and underfitting can be seen in picture 4.2.

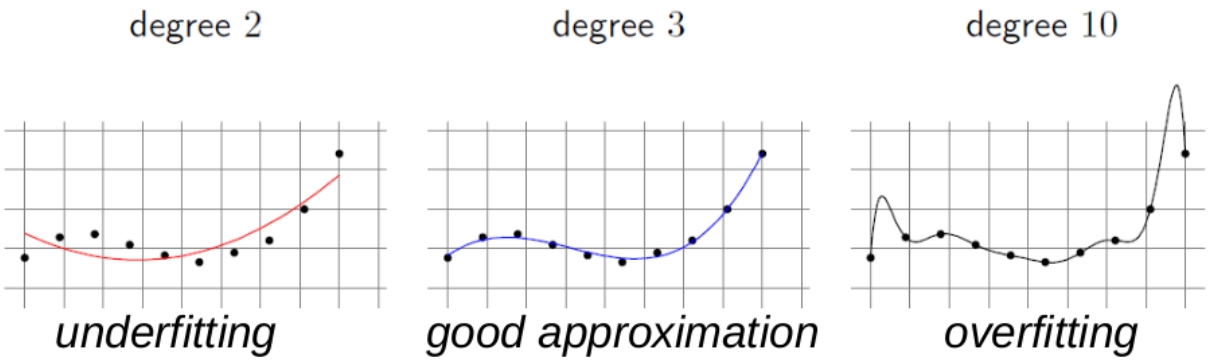


Figure 4.2: Comparison between underfitting and overtitting

We need to estimate the generalization error $L_{\mathcal{D}}(h)$ for a function h (the one selected with ERM). To do this, we use a *test set*, a new set of samples different (disjoint) from the training set used to pick h. Sometimes we use also a *validation set* for selecting the hyper-parameters of the algorithm or to evaluate errors while iterative training procedures are running.

To summarize the procedure of training a ML algorithm parametrized by some Hyper-Parameters (HP):

1. Select Hyper-Parameters values
2. Train the algorithm with the choosen parameters on the training set
3. Evaluate performances on the validation set
4. Go back to 1. and select new HP values
5. Select the final HP leading to the smallest validation error
6. Compute error estimation on the test set

Chapter 5

Linear Predictors

In this chapter we will study the family of linear predictors, one of the most useful families of hypothesis classes. Many learning algorithms widely used in practice, rely on linear predictors, because of their intuitiveness and ease of interpretation, as well as for their efficiency.
We will introduce several hypothesis classes:

- Halfspaces (classification)
- Linear Regression (regression)
- Logistic Regression (classification modeled as a regression problem)

and the algorithms to find the predictors:

- Linear Programming (for halfspaces)
- Perceptron (for halfspaces)
- Least Squares (for regression)

First, we define the class of affine functions:

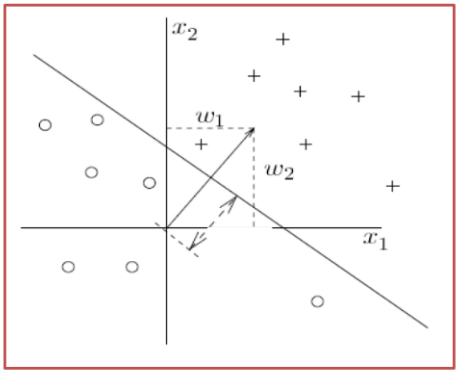
$$L_d = \left\{ h_{\mathbf{w},b} : \mathbf{w} \in \mathbb{R}^d, b \in \mathbb{R} \right\}$$

where

$$h_{\mathbf{w},b}(\mathbf{x}) = \langle \mathbf{w}, \mathbf{x} \rangle + b = \left(\sum_{i=1}^{d} w_i x_i \right) + b$$

Each member of L_d is a function $\mathbf{x} \to \langle \mathbf{w}, \mathbf{x} \rangle + b$ parameterized by a vector $\mathbf{w} \in \mathbb{R}^d$ (of *weights*) and a scalar (called *bias*) $b \in \mathbb{R}$.
The different hypothesis classes of linear predictors are composition of functions $\phi : \mathbb{R} \to \mathcal{Y}$ on L_d. For example, in binary classification, we can choose ϕ to be the sign function, and for regression problems, where $\mathcal{Y} = \mathbb{R}$, ϕ is simply the identity function.
We can give a (2D) geometric interpretation of this functions. As can be seen in figure 5.1:

- The bias is proportional to the offset of the line from the origin
- The weights determine the slope of the line
- The weight vector is perpendicular to the line



$$x_2 = -\frac{w_1}{w_2}x_1 - \frac{b}{w_2}$$

Figure 5.1: Geometrical interpretation of the class of affine functions

It may be more convenient to incorporate the bias b into the vector $\mathbf{w}$ as an extra coordinate with a value of 1 in all $x \in \mathcal{X}$; namely let $\mathbf{w}' = (b, w_1, \ldots, w_d) \in \mathbb{R}^{d+1}$ and let $\mathbf{x}' = (1, x_1, \ldots, x_d) \in \mathbb{R}^{d+1}$. Therefore:

$$h_{\mathbf{w},b}(\mathbf{x}) = \langle \mathbf{w}, \mathbf{x} \rangle + b = \langle \mathbf{w}', \mathbf{x}' \rangle$$

It follows that each affine function in $\mathbb{R}^d$ can be rewritten as an *homogeneous* linear function in $\mathbb{R}^{d+1}$ applied over the transformation that appends the constant 1 to each input vector. Therefore, whenever it simplifies the presentation, we will omit the bias term and refer to L_d as the class of homogeneous linear functions of the form $h_{\mathbf{w}}(\mathbf{x}) = \langle \mathbf{w}, \mathbf{x} \rangle$.

5.1 Halfspaces

The first hypothesis class we consider is the class of halfspaces, designed for binary classification problems, namely $\mathcal{X} = \mathbb{R}^d$ and $\mathcal{Y} = \{-1, 1\}$. The class of halfspaces is defined as follow:

$$HS_d = sign \circ L_d \{\mathbf{x} \to sign\left(h_{\mathbf{w},b}(\mathbf{x})\right) : h_{\mathbf{w},b} \in L_d\}$$

In other words, each halfspace hypothesis in HS_d is parameterized by $\mathbf{w} \in \mathbb{R}^d$ and $b \in \mathbb{R}$ and upon receiving a vector $\mathbf{x}$ the hypothesis returns the label $sign\left(\langle \mathbf{w}, \mathbf{x} \rangle + b\right)$.
To illustrate this hypothesis class geometrically, it is instructive to consider the case $d = 2$. Each

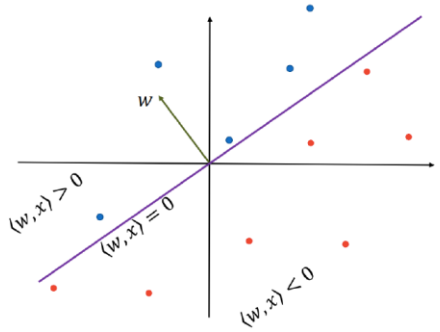


Figure 5.2: Two-dimensional representation of halfspaces

hypothesis forms a hyperplane that is perpendicular to the vector $\mathbf{w}$ and intersects the vertical

axis at the point $(0, -b/w_2)$. The instances that are "above" the hyperplane, that share an acute angle with $\mathbf{w}$, are labeled positively. Instances that are "below" the hyperplane, that share an obtuse angle with $\mathbf{w}$, are labeled negatively. An example can be seen in figure 5.2.
However, as we can see from figure 5.3, to find an ERM halfspace we need to be in a realizable case: in this context, the realizable case is often referred to the *separable* case, since it is possible to separate with an hyperplane all the positive examples from all the negative ones, while it is computationally hard to implement the ERM rule in the non separable case (agnostic case).

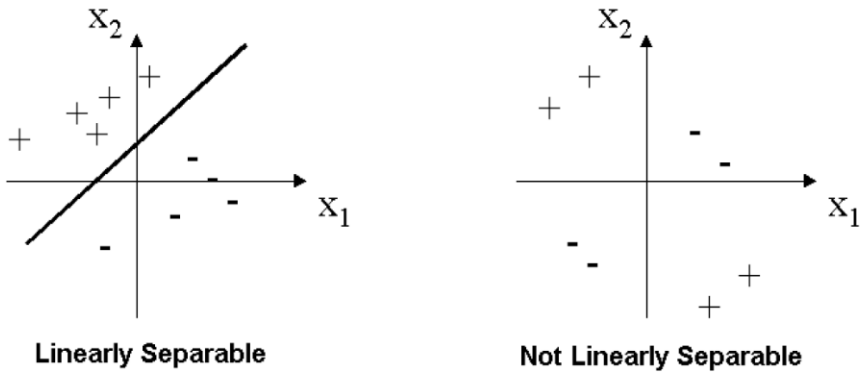


Figure 5.3: Realizability condition of ERM algorithm for halfspaces

5.1.1 Linear Programming for the Class of Halfspaces

Linear programs (LP) are problems that can be expressed as maximizing a linear function subject to linear inequalities. The main target, consist into finding the quantity $\max_{\mathbf{w}\in\mathbb{R}^d}\langle \mathbf{u}, \mathbf{w}\rangle$ subject to $A\mathbf{w} \geq \mathbf{v}$ where $\mathbf{w} \in \mathbb{R}^d$ is the vector of variables we wish to determine, A is a $m \times d$ matrix and $\mathbf{v} \in \mathbb{R}^m$ and $\mathbf{u} \in \mathbb{R}^d$ are vectors. Linear programs can be solved efficiently namely, in a polinomial time.
Let's assume, for simplicity, the homogeneous case, and let $S = \{(\mathbf{x_i}, y_i)\}_{i=1}^m$ be a training set of size m. Sinc we assume the realizable case, an ERM predictor should have zero errors on the training set ($L_S(h_S)$). So we are looking for some vector $\mathbf{w} \in \mathbb{R}^d$ for which $sign\left(\langle \mathbf{w}, \mathbf{x_i}\rangle\right) = y_i$; or, equivalently, $y_i\langle \mathbf{w}, \mathbf{x_i}\rangle > 0 \ \forall i = 1, \ldots, m$. Let $\mathbf{w}^*$ be a vector that satisfies this condition (it must exist since we assumed realizability). Define $\gamma = \min_i \left(y_i\langle \mathbf{w}^*, \mathbf{x_i}\rangle\right)$ and let $\bar{\mathbf{w}} = \mathbf{w}/\gamma$. Therefore, for all i we have

$$y_i\langle \bar{\mathbf{w}}, \mathbf{x_i}\rangle = \frac{1}{\gamma} y_i\langle \mathbf{w}^*, \mathbf{x_i}\rangle \geq 1$$

We have thus shown that there exist a vector that satisfy $y_i\langle \mathbf{w}, \mathbf{x_i}\rangle \geq 1$ and clearly, such a vector is an ERM predictor.
To find a vector that satisfy the previous equation we have to use the following procedure: set A to be the $m \times d$ matrix whose rows are the instances multiplied by y_i. Namely $A_{ij} = y_i x_{i,j}$ where $x_{i,j}$ represents the j-th component of the vector $\mathbf{x_i}$. Let $\mathbf{v}$ be the vector $(1, \ldots, 1) \in \mathbb{R}^m$. Then, we can re-write the previous equation as $A\mathbf{w} \geq \mathbf{v}$.
This procedure requires also a maximization object, because the $\mathbf{w}$ that satisfy the constraints are equal candidates as output hypothesis. For this reason, we set a "dummy" objective $\mathbf{u} = (0, \ldots, 0) \in \mathbb{R}^d$.

5.1.2 Perceptron for Halfspaces

A different implementation of the ERM rule is the Perceptron algorithm of Rosenblatt (1938). The main target of this algorithm is to find the vector $\mathbf{w}$ that represents the separating hyperplane, in homogeneous coordinates.
The Perceptron is an iterative algorithm that construct a sequence of vectors $\mathbf{w}^{(1)}, \mathbf{w}^{(2)}, \ldots,$ starting from a vector $\mathbf{w}^{(1)}$ set to be the all-zero vector. At iteration t the Percetron finds an example i that is mislabeled by $\mathbf{w}^{(t)}$, namely, an example for which $sign(\langle \mathbf{w^{(t)}}, \mathbf{x_i} \rangle) \neq y_i$. Then the Perceptron updates $\mathbf{w}^{(t)}$ by adding to it the instance $\mathbf{x_i}$ scaled by the label y_i:
$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} + y_i \mathbf{x_i}$. Recall that our goal is to have $y_i \langle \mathbf{w}, \mathbf{x_i} \rangle > 0$ for all i and note that

$$y_i \langle \mathbf{w^{(t+1)}}, \mathbf{x_i} \rangle = y_i \langle \mathbf{w^{(t)}} + \mathbf{y_i x_i}, \mathbf{x_i} \rangle = y_i \langle \mathbf{w^{(t)}}, \mathbf{x_i} \rangle + \|x_i\|^2$$

So every update of the vector $\mathbf{w}$ guides the solution to be "more correct" on the i-th example, the couple $(\mathbf{x_i}, y_i)$.

Perceptron Algorithm

Input: a training set $(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_m, y_m)$
Initialization: $w^{(1)} = (0, ..., 0)$

for t = 1,2,..,m:
  if $\exists\ i$ such that $y_i \langle \mathbf{w^{(t)}}, \mathbf{x_i} \rangle \leq 0$ then
  $\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} + y_i \mathbf{x_i}$
else:
  Output: w

The following theorem guarantees that in the realizable case, the algorithm stops with all the sample points classified.

Theorem 5.1. *Assume that* $(\mathbf{x_1}, y_1), \ldots, (\mathbf{x_m}, y_m)$ *is separable; let* $B = \min \{\|\mathbf{w}\| : \forall i \in [m],\ y_i \langle \mathbf{w}, \mathbf{x_i} \rangle \geq 1\}$ *and let* $R = \max_i \|\mathbf{x_i}\|$. *Then, the Perceptron algorithm stops after at most* $(RB)^2$ *iterations, and when it stops it holds that* $\forall i \in [m]$, $y_i \langle \mathbf{w^{(t)}}, \mathbf{x_i} \rangle > 0$ *(so when there are no more wrongly classified samples).*

Proof. By the definition of the stopping condition, if the Perceptron stops it must have separated all the examples. We will show that if the Perceptron runs for T iterations, then we must have $T \leq (RB)^2$.
Let's define $\mathbf{w}^*$ as the vector that achieve the *min* definition of B i.e. among all vectors that satisfy the constraint $y_i \langle \mathbf{w}^*, \mathbf{x_i} \rangle \geq 1$ for all i, $\mathbf{w}^*$ is the one with minimal norm. The idea of the proof is to show that after T iterations, the cosine of the angle between $\mathbf{w}^*$ and $\mathbf{w}^{(T+1)}$ is at least $\sqrt{T}/RB$, namely

$$\frac{\langle \mathbf{w}^*, \mathbf{w^{(T+1)}} \rangle}{\|\mathbf{w}^*\| \|\mathbf{w}^{(T+1)}\|} \leq \frac{\sqrt{T}}{RB}$$

By the Cauchy-Schwartz inequality, the left-hand side of the relation above is at most 1. Therefore, ths would imply that

$$1 \geq \frac{\sqrt{T}}{RB} \quad \Longrightarrow \quad T \leq (RB)^2$$

which will conclude our proof.
We will proceed in two parts:

1. We will show that, for the numerator, $\langle \mathbf{w}^*, \mathbf{w}^{(\mathbf{T+1})} \rangle \geq T$.
   A first iteration $\mathbf{w}^{(1)} = (0, \ldots, 0)$ and therefore $\langle \mathbf{w}^*, \mathbf{w}^{(\mathbf{1})} \rangle = 0$, while on iteration t, if we update using example $(\mathbf{x}_i, y_i)$ we have that

   $$\langle \mathbf{w}^*, \mathbf{w}^{(\mathbf{T+1})} \rangle - \langle \mathbf{w}^*, \mathbf{w}^{(\mathbf{t})} \rangle = \langle \mathbf{w}^*, \mathbf{w}^{(\mathbf{t+1})} - \mathbf{w}^{(\mathbf{t})} \rangle = \langle \mathbf{w}^*, \mathbf{y_i x_i} \rangle = y_i \langle \mathbf{w}^*, \mathbf{x_i} \rangle \geq 1$$

   Therefore, after performing T iterations, we get

   $$\langle \mathbf{w}^*, \mathbf{w}^{(\mathbf{T+1})} \rangle = \sum_{t=1}^{T} \left(\langle \mathbf{w}^*, \mathbf{w}^{(\mathbf{T+1})} - \langle \mathbf{w}^*, \mathbf{w}^{(\mathbf{t})} \rangle \rangle \right) \geq T$$

2. Next, we will show that $\|\mathbf{w}^*\| \left\| \mathbf{w}^{(T+1)} \right\| \leq \sqrt{T} RB$.
   We upper bound $\left\| \mathbf{w}^{(T+1)} \right\|$ observing that, for each iteration t we have that

   $$\left\| \mathbf{w}^{(T+1)} \right\|^2 = \left\| \mathbf{w}^{(t)} + y_i \mathbf{x_i} \right\|^2 = \left\| \mathbf{w}^{(t)} \right\|^2 + 2 y_i \langle \mathbf{w}^{(\mathbf{t})}, \mathbf{x_i} \rangle + y_i^2 \|\mathbf{x_i}\|^2 \leq \left\| \mathbf{w}^{(t)} \right\|^2 + R^2$$

   when the last inequality is due to the fact that example i is necessarily such that $y_i \langle \mathbf{w}^{(\mathbf{t})}, \mathbf{x_i} \rangle \leq 0$, and the norm of $\mathbf{x_i}$ is at most R. Now, since $\left\| \mathbf{w}^{(1)} \right\|^2 = 0$, if we use the previous relation recursively for T iterations, we obtain that

   $$\left\| w^{(T+1)} \right\|^2 \leq T R^2 \quad \Longrightarrow \quad \sqrt{T} R$$

Combining the relations we have just found, and using the fact that $\|\mathbf{w}^*\| = B$, we obtain that

$$\frac{\langle \mathbf{w}^{(\mathbf{T+1})}, \mathbf{w}^* \rangle}{\|\mathbf{w}^*\| \|\mathbf{w}^{(T+1)}\|} \geq \frac{T}{B\sqrt{T}R} = \frac{\sqrt{T}}{BR}$$

With this we can conclude our proof. □

Notes on Perceptron:

- Perceptron is simple to implement, and is guaranteed to converge (at least for the realizable cases).
- The convergence rate depends on the parameter B, which in some situations might be exponentially large in d:
  - If the input vectors are not normalized and arranged in some unfavorable ways, the running time can be very long.
  - In such cases, it would be better to implement the ERM problem with Linear Programming.
- The solution is not unique: you can have many different results, which one is picked depends on starting values.
- If data are not separable the Perceptron should be run for some time and we should keep the best solution found up to that point.

Car Age (years)	Price (€)
4	6300
4	5800
5	5700
5	4500
7	4500
7	4200
8	4100
9	3100
10	2100
11	2500
12	2200

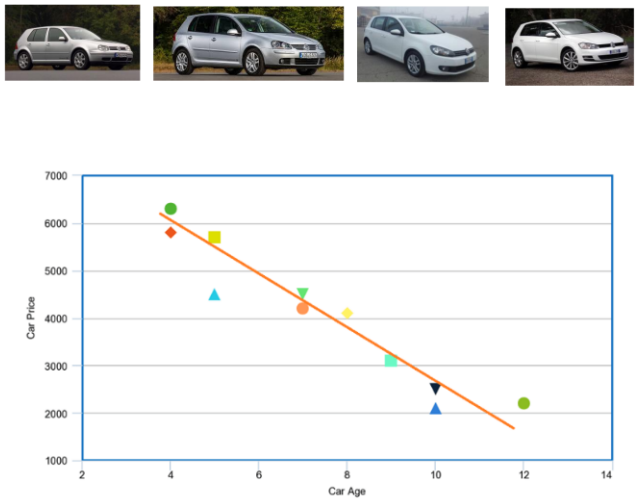


Figure 5.4: Example of linear regression method

5.2 Linear Regression

Linear Regression is a common statistical tool used to estimate the relation between some explanatory variables and some real valued outcome. Cast as a learning problem, the domain set $\mathcal{X}$ is a subset of $\mathbb{R}^d$ and the label set $\mathcal{Y}$ is the set of real numbers. We would like to learn a function $h : \mathbb{R}^d \to \mathbb{R}$ that best approximates the relationship between our variables (input and output).
The hypothesis class of linear regression predictors is simply the set of linear functions:

$$\mathcal{H}_{reg} = L_d = \left\{\mathbf{x} \to \langle \mathbf{w}, \mathbf{x} \rangle + b : \mathbf{w} \in \mathbb{R}^d, b \in \mathbb{R}\right\}$$

Next, we need to define a loss function for regression. While in classification problems the definition of the loss is straightforward, as a predictor can guess or not the correct result, in the regression case there are solutions that are "more correct" than others. We therefore need to define how much we shall "penalize" for the discrepancy between the prediction $h(\mathbf{x})$ and the correct label y. A common way is to use the **squared-loss function**:

$$\ell(h, (\mathbf{x}, y)) = (h(\mathbf{x}) - y)^2$$

For this loss function, the corresponding empirical risk is called the **Mean-Squared Error**:

$$L_S(h) = \frac{1}{m} \sum_{i=1}^{m} (h(\mathbf{x}_i) - y_i)^2 \tag{5.1}$$

Of course, there are a variety of other loss functions that one can use, for example the *absolute value loss function*, defined as $\ell(h, (\mathbf{x}, y)) = |h(\mathbf{x}) - y|$.
An example of linear regression can be seen in figure 5.4.

5.3 Least Squares

The Least Squares is the algorithm that solves the ERM problem for the hypothesis class of linear regression predictors with respect to the squared loss. The ERM problem with respect to

this class, given a training set S and using the homogeneous version of L_d is to find the parameters vector that minimize the mean-squared error (5.1) between the estimated and training values, namely

$$\underset{\mathbf{w}}{argmin} L_S(h_{\mathbf{w}}) = \underset{\mathbf{w}}{argmin} \frac{1}{m} \sum_{i=1}^{m} \left(\langle \mathbf{w}, \mathbf{x_i} \rangle - y_i \right)^2$$

To solve the problem we calculate the gradient of the objective function and we impose it equal to zero:

$$\frac{2}{m} \sum_{i=1}^{m} \left(\langle \mathbf{w}, \mathbf{x_i} \rangle - y_i \right) \mathbf{x_i} = 0$$

We can rewrite the problem as $A\mathbf{w} = \mathbf{b}$ where

$$A = \left(\sum_{i=1}^{m} \mathbf{x_i} \mathbf{x_i^T} \right) \quad \text{and} \quad \mathbf{b} = \sum_{i=1}^{m} y_i \mathbf{x_i}$$

If A is invertible then the solution of the ERM problem is

$$\mathbf{w} = A^{-1}\mathbf{b}$$

Inversion of A is the most expensive operation, from a computational point of view. However, the case in which A is not invertible requires a special handling (not part of the course).

5.4 Polynomial Regression

Some learning tasks call for nonlinear predictors, such as polynomial predictors. Take, for instance, a one dimensional polynomial function of degree n that is

$$p(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$$

where $(a_0, \ldots, a_n)$ is a vector of coefficients of size $n+1$ that we have to estimate.
One way to learn this class is by reduction to a linear regression problem, which we already know how to solve. To translate a polynomial regression problem to a linear regression one we define the mapping:

$$\psi : \mathbb{R} \to \mathbb{R}^{n+1} \qquad \psi(x) = (1, x, x^2, \ldots, x^n)$$

So, we obtain that:

$$p(\psi(x)) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n = \langle \mathbf{a}, \psi(\mathbf{x}) \rangle$$

and we can find the optimal vector coefficients using, for example, the Least Squares method. What we have just done is moving to an higher dimensional space, transforming the nonlinear problem we were trying to solve in an easier linear one.
We also need to notice that however the variables are not independent and the optimization can become unstable for large n.

5.5 Logistic Regression

In Logistic Regression we learn a family of functions h from $\mathbb{R}^d$ to the interval $[0,1]$. It is mostly used for classification problems to reframe them as regression problem: we can in fact interpret $h(\mathbf{x})$ as the *probability* that 1 is the label of $\mathbf{x}$.
The hypothesis class associated with logistic regression is the composition of a **sigmoid function** $\phi_{sig} : \mathbb{R} \to [0,1]$ over the class of linear functions L_d. In particular, the most used is the *logistic function*, represented in picture 5.5 and defined as

$$\phi_{sig}(z) = \frac{1}{1+e^{-z}}$$

Properties:

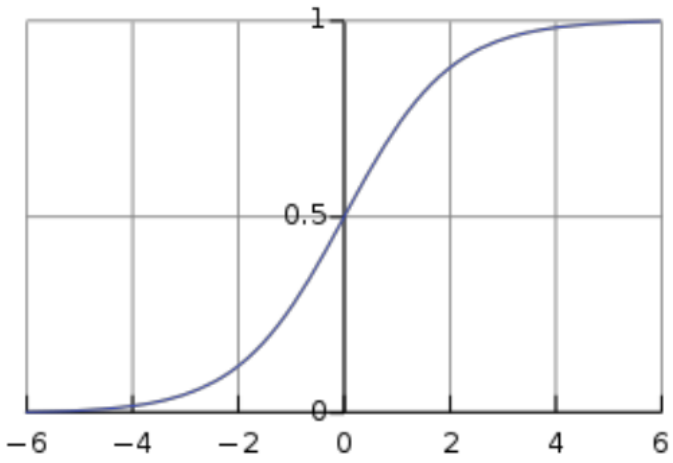


Figure 5.5: Logistic Function

- It is bigger than 1/2 for positive values and smaller for negative ones.
- Tends to 1 for large positive values and to 0 for negative ones.
- Can be viewed as a scaled and shifted *sign* function.

The hypothesis class, in the homogeneous case, is

$$H_{sig} = \phi_{sig} \circ L_d = \left\{ \mathbf{x} \to \phi_{sig}\left(\langle \mathbf{w}, \mathbf{x} \rangle\right) : \mathbf{w} \in \mathbb{R}^d \right\}$$

Note that when the scalar product $\langle \mathbf{w}, \mathbf{x} \rangle$ is very large then $\phi_{sig}(\langle \mathbf{w}, \mathbf{x} \rangle)$ is close to 1, while when that quantity is very small then $\phi_{sig}(\langle \mathbf{w}, \mathbf{x} \rangle)$ is close to 0.
Recall that the predictor of the halfspace corresponding to a vector $\mathbf{w}$ is the quantity $sign(\langle \mathbf{w}, \mathbf{x} \rangle)$. Therefore the predictions of the halfspace hypothesis and the logistic hypothesis are very similar whenever $|\langle \mathbf{w}, \mathbf{x} \rangle|$ is large. However, when $|\langle \mathbf{w}, \mathbf{x} \rangle|$ is close to 0 we have that $\phi_{sig}(\langle \mathbf{w}, \mathbf{x} \rangle) \simeq 1/2$. Intuitively, the logistic hypothesis is not sure about the value of the label, so it guessed that is $sign(\langle \mathbf{w}, \mathbf{x} \rangle)$ with probability slight to 50%. In contrast, the halfspace hypothesis always output a deterministic prediction of either 1 or -1, even if $|\langle \mathbf{w}, \mathbf{x} \rangle|$ is very close to 0.

Next, we specify a loss function:

$$\ell\left(h_{\mathbf{w}}, (\mathbf{x}, y)\right) = \log\left(1 + e^{-y\langle \mathbf{w}, \mathbf{x}\rangle}\right)$$

Therefore, given a training set $S = (\mathbf{x_1}, y_1), \ldots, (\mathbf{x_m}, y_m))$, the ERM problem associated with logistic regression is

$$\underset{\mathbf{w}\in\mathbb{R}^d}{argmin} \frac{1}{m} \sum_{i=1}^{m} \log\left(1 + e^{-y_i\langle \mathbf{w}, \mathbf{x_i}\rangle}\right)$$

5.6 Maximum Likelihood Estimation

The Maximum Likelihood Estimator (MLE) is a statistical approach to a learning problem, used to find the parameters that maximize the joint probability o a given dataset, assuming a specific parametric probability function.
MLE essentially uses a *generative* approach, in which it is assumed that the underlying distribution over the data has a specific parametric form.
Given a training set $S = ((\mathbf{x_1}, y_1), \ldots, (\mathbf{x_m}, y_m))$, we assume that each $(\mathbf{x_i}, y_i)$ is i.i.d. from some probability distribution $\mathcal{P}_\theta$ that is characterized by some parameters that we want to estimate.
Next, we consider the *likelihood* of data (given the parameters), i.e. the joint probability of the dataset:

$$L(S|\theta) = \prod_{i=1}^{m} \mathcal{P}_\theta(\mathbf{x_i})$$

Then, to maximize this function, we consider its logarithm: the function log is monotonic, so it will preserve the maximum but it will be also simpler to differentiate.

$$\log(L(S|\theta)) = \log\left(\prod_{i=1}^{m} \mathcal{P}_\theta(\mathbf{x_i})\right) = \sum_{i=1}^{m} \log\left(\mathcal{P}_\theta(\mathbf{x_i})\right)$$

At this point, we want to compute the best choice for the parameter θ, that is the one that maximize the likelihood function:

$$\hat{\theta} = \underset{\theta}{argmax}\, L(S|\theta)$$

5.6.1 MLE and Logistic Regression

We can show that MLE solution is equivalent to ERM solution for logistic regression.
As we know, for logistic regression:

1. Assume the training set $S = ((\mathbf{x_1}, y_1), \ldots, (\mathbf{x_m}, y_m))$
2. $P\left[y_i = 1\right] = h_{\mathbf{w}}(\mathbf{x_i}) = \frac{1}{1+e^{-\langle \mathbf{w}, \mathbf{x_i}\rangle}} = \frac{1}{1+e^{-y_i\langle \mathbf{w}, \mathbf{x_i}\rangle}}$
3. $P\left[y_i = -1\right] = 1 - h_{\mathbf{w}}(\mathbf{x_i}) = \frac{1}{1+e^{\langle \mathbf{w}, \mathbf{x_i}\rangle}} = \frac{1}{1+e^{-y_i\langle \mathbf{w}, \mathbf{x_i}\rangle}}$
4. Likelihood of the training set (joint probability):

$$P\left[S|\mathbf{w}\right] = \prod_{i=1}^{m} \left(\frac{1}{1 + e^{-\langle \mathbf{w}, \mathbf{x_i}\rangle}}\right)$$

$$\log(P\left[S|\mathbf{w}\right]) = -\sum_{i=1}^{m} \log\left(1 + e^{-y_i \langle \mathbf{w}, \mathbf{x_i} \rangle}\right)$$

While the Maximum Likelihood Estimator is:

$$\underset{\mathbf{w}}{argmax}\ L(S; \mathbf{w}) = \underset{\mathbf{w}}{argmax}\ \log(P[S|\mathbf{w}]) = \underset{\mathbf{w}}{argmin}\ \sum_{i=1}^{m} \log\left(1 + e^{-y_i \langle \mathbf{w}, \mathbf{x_i} \rangle}\right)$$

Chapter 6

VC Dimension

We have seen, in the previous chapters, that a finite class ($|\mathcal{H}| < \infty$) is agnostic PAC learnable. What about infinite classes? We would like to understand when infinite classes are learnable and also the number of samples that we need to work with.
To show that the size of the hypothesis class is not the right characterization of its sample complexity, we first present a simple example of an infinite-size hypothesis class that is learnable.

Example 6.1. *Let $\mathcal{H}$ be the set of threshold functions over the real line, namely $\mathcal{H} = \{h_a : a \in \mathbb{R}\}$ where $h_a : \mathbb{R} \to \{0, 1\}$ is a function such that $\begin{cases} 1 & \text{if } x < a \\ 0 & \text{otherwise} \end{cases}$.*
Clearly, $\mathcal{H}$ is of infinite size, but we can show that $\mathcal{H}$ is PAC learnable, using the ERM rule, with sample complexity $m_{\mathcal{H}}(\varepsilon, \delta) \leq [\log(2/\delta)/\varepsilon]$.

Proof. Let a^* be a threshold such that the hypothesis $h^*(x)$ achieves $L_{\mathcal{D}}(h^*) = 0$. Let $\mathcal{D}_x$ be the marginal distribution over the domain $\mathcal{X}$ and let $a_0 < a^* < a_1$ be such that

$$\mathop{\mathbb{P}}_{x \sim \mathcal{D}_x} [x \in (a_0, a^*)] = \mathop{\mathbb{P}}_{x \sim \mathcal{D}_x} [x \in (a^*, a_1)] = \varepsilon$$

Given a training set S, let $b_0 = \max \{x : (x, 1) \in S\}$ and $b_1 = \min \{x : (x, 0) \in S\}$. Let b_S be a threshold corresponding to ERM hypothesis h_S, which implies that $b_S \in (b_0, b_1)$. Therefore, the sufficient condition for $L_{\mathcal{D}}(h_S) \leq \varepsilon$ is that both $b_0 \geq a_0$ and $b_1 \leq a_1$. In other words):

$$\mathop{\mathbb{P}}_{S \sim \mathcal{D}^m} [L_{\mathcal{D}}(h_S) > \varepsilon] \leq \mathop{\mathbb{P}}_{S \sim \mathcal{D}^m} [b_0 < a_0 \vee b_1 > a_1] \leq \mathop{\mathbb{P}}_{S \sim \mathcal{D}^m} [b_0 < a_0] + \leq \mathop{\mathbb{P}}_{S \sim \mathcal{D}^m} [b_1 > a_1]$$

where, in the second disequation, we used the union bound. The event $b_0 < a_0$ happens only if all examples in S are not in the interval (a_0, a^*), whose probability mass is defined to be ε, namely:

$$\mathop{\mathbb{P}}_{S \sim \mathcal{D}^m} [b_0 < a_0] =\leq \mathop{\mathbb{P}}_{S \sim \mathcal{D}^m} [\forall (x, y) \in S,\ x \notin (a_0, a^*)] = (1 - \varepsilon)^m \leq e^{-\varepsilon m}$$

Since we assume $m \geq \log(2/\delta)/\varepsilon$ it follows that the equation is at most $\delta/2$. In the same way it is easy to see that

$$\mathop{\mathbb{P}}_{S \sim \mathcal{D}^m} [b_1 > a_1] \leq \frac{\delta}{2}$$

Combining it with the previous relation we conclude our proof. □

We see, therefore, that while finiteness of $\mathcal{H}$ is a sufficient condition for learnability, it is not a necessary condition.
Before giving the next definitions, let us recall the No Free Lunch theorem and its proof: we have shown that without restricting the hypothesis class, for any learning algorithm, an adversary can construct a distribution for which the learning algorithm will perform poorly, while there is another learning algorithm that will succeed on the same distribution. To proove that, the idea was to select a distribution concentrated on a set C (so a restriction) on which the algorithm fails. Since we are considering distributions that are concentrated on elements of C, we should study $\mathcal{H}$ behaves on C.

Definition 6.1. *(**Restriction of $\mathcal{H}$ to C**)*
Let $\mathcal{H}$ be a class of functions from $\mathcal{X}$ to $\{0,1\}$ and let $C = \{c_1, \dots, c_m\} \subset \mathcal{X}$. The restriction $\mathcal{H}_C$ of $\mathcal{H}$ to C is:

$$\mathcal{H}_C = \{[h(c_1), \dots, h(c_m)] : h \in \mathcal{H}\}$$

where we represent each function from C to $\{0,1\}$ as a vector in $\{0,1\}^{|C|}$.

If the restriction of $\mathcal{H}$ in C is the set of all functions from C to $\{0,1\}$, then we say that $\mathcal{H}$ *shatters* the set C. Formally

Definition 6.2. *(**Shattering**)*
A hypothesis class $\mathcal{H}$ shatters a finite set $C \subset \mathcal{X}$ if the restriction of $\mathcal{H}$ to C is the set of all functions from C to $\{0,1\}$, that is $|\mathcal{H}_C| = 2^{|C|}$.

Getting back to the construction of an adversarial distribution as in the proof of the No-Free-Lunch theorem, we see that whenever some set C is shattered by $\mathcal{H}$, the adversary is not restricted by $\mathcal{H}$, as they can construct a distribution over C based on any target function from C to $\{0,1\}$, while still maintaining the realizability assumption. This immediately yields:

Corollary 6.0.1. *Let $\mathcal{H}$ be a hypothesis class of functions from $\mathcal{X}$ to $\{0,1\}$. Let m be a training set size. Assume that there exist a set $C \subset \mathcal{X}$ of size $2m$ that is shattered by $\mathcal{H}$. Then for any learning algorithm A there exist a distribution $\mathcal{D}$ over $\mathcal{X} \times \{0,1\}$ and a predictor $h \in \mathcal{H}$ such that $L_d(h) = 0$ but with probability at least 1/7 over the choice of S we have that $L_{\mathcal{D}}(A(S)) \geq 1/8$.*

This corollary tells us that if $\mathcal{H}$ shatters some set C of size $2m$ then we cannot learn $\mathcal{H}$ using m examples.

Definition 6.3. *(**VC-dimension**)*
The VC-dimension of an 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 that $\mathcal{H}$ has infinite VC-dimension.

Notice that:

- If $\mathcal{H}$ is a finite class, for any set C we have $|\mathcal{H}_C| \leq |H|$ and thus C cannot be shattered if $|\mathcal{H}| \leq 2^{|C|}$. This implies that $VCdim(\mathcal{H}) \leq \log_2(|\mathcal{H}|)$.
- If $\mathcal{H}$ has an infinite VC-dimension it is not PAC learnable: as a consequence of the corollary 6.0.1, there will always exist a subset of size $2m$ that is shattered for any m.

To show that $VCdim(\mathcal{H}) = d$ we need to show that:

1. $VCdim(\mathcal{H}) \geq d$: there exists a set C of size d that is shattered by $\mathcal{H}$.

2. $VCdim(\mathcal{H}) \leq (d+1)$: every set C of size $d+1$ is not shattered by $\mathcal{H}$.

Example 6.2. *(Threshold functions)*
Let $\mathcal{H}$ be the class of threshold functions over $\mathbb{R}$. Take a set $C = \{c_1\}$. Now, if we take $a = c_1 + 1$, then we have $h_a(c_1) = 1$, and if we take $a = c_1 - 1$, then we have $h_a(c_1) = 0$. Therefore, $\mathcal{H}_C$ is the set of all the functions from C to $\{0,1\}$, and $\mathcal{H}$ shatters C. Now let's take a set $C = \{c_1, c_2\}$, where $c_1 \leq c_2$. No $h \in \mathcal{H}$ can account for the labeling $(0,1)$, because any threshold that assign the label 0 to c_1 must assign the label 0 to c_2 as well. Therefore not all functions from C to $\{0,1\}$ are included in $\mathcal{H}_C$; hence C is not shattered by $\mathcal{H}$.
So, we just proved that an arbitrary set $C = \{c_1\}$ is shattered by $\mathcal{H}$ (where $\mathcal{H}$ is the class of threshold functions over $\mathbb{R}$); therefore $VCdim(\mathcal{H}) \geq 1$. At the same time, an arbitrary set $C = \{c_1, c_2\}$ is not shattered by $\mathcal{H}$. So we can conclude that $VCdim(\mathcal{H}) = 1$.

Example 6.3. *(Intervals)*
Let $\mathcal{H}$ be the class of intervals over $\mathbb{R}$, namely $\mathcal{H} = \{h_{a,b} : a, b \in \mathbb{R}, a < b\}$, where $h_{a,b} : \mathbb{R} \to \{0,1\}$ is a functions that is 1 for $x \in (a,b)$ and 0 otherwise. Let's take the set $C = \{1,2\}$. Then, $\mathcal{H}$ shatters C and therefore $VCdim(\mathcal{H}) \geq 2$. Now let's make an arbitrary set $C = \{c_1, c_2, c_3\}$ and assume without loss of generality that $c_1 \leq c_2 \leq c_3$. Then, the labeling (1,0,1) cannot be obtained by an interval, and therefore $\mathcal{H}$ does not shatter C. We therefore conclude that $VCdim(\mathcal{H}) = 2$.

Example 6.4. *(Axis aligned rectangles)*
Let $\mathcal{H}$ be the class of axis aligned rectangles, formally: $\mathcal{H} = \left\{h_{(a_1,a_2,b_1,b_2)} : a_1 \leq a_2 \text{ and } b_1 \leq b_2\right\}$, where

$$h_{(a_1,a_2,b_1,b_2)} = \begin{cases} 1 & \text{if } a_1 \leq x_1 \leq a_2 \text{ and } b_1 \leq x_2 \leq b_2 \\ 0 & \text{otherwise} \end{cases}$$

As can be seen in figure 6.1, it is easy to find a set of 4 points that are shattered, so $VCdim(\mathcal{H}) \geq 4$. Now let's consider any set $C \in \mathbb{R}^2$ of 5 points from which we are gonna take the

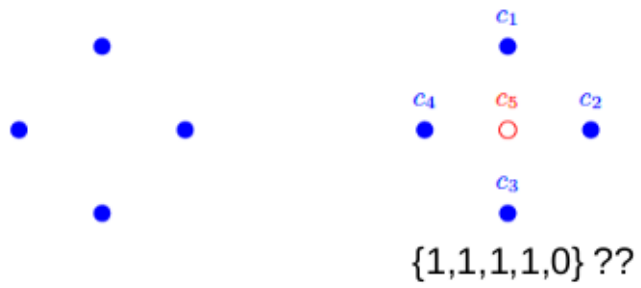


Figure 6.1: Left: 4 points are shattered by axis aligned rectangles. Right: any axis aligned rectangle cannot label c_5 by 0 and the rest of the points by 1

leftmost, the rightmost, the highest and the lowest one. Without loss of generality, denote $C = \{c_1, \ldots, c_5\}$ and let c_5 be the point that was not selected. It is impossible to obtain the label (1,1,1,1,0) by any axis aligned rectangle, because such rectangle must contain $c_1, \ldots, c_4$, but at the same time must contain c_5 too, because its coordinates are within the intervals defined by the selected points. So this C is not shattered by $\mathcal{H}$, and therefore $VCdim(\mathcal{H}) = 4$.

Example 6.5. *(**VC dimension of halfspaces hypothesis class**)*
We want to prove that the VC dimension of the class of homgeneous halfspaces in $\mathbb{R}^d$ is d.

- *$VCdim(\mathcal{H}) \geq d$: I have to find at least one set of size d that is shattered by $\mathcal{H}$. For this reason I consider the set of vectors $\mathbf{e}_1, \ldots, \mathbf{e}_d$, that are the vectors of the canonical base in $\mathbb{R}^d$: this set is shattered by the class $\mathcal{H}$. In fact, for labeling $y_1, \ldots, y_d$, and setting $\mathbf{w} = (y_1, \ldots, y_d)$, then $sprodwe_i = y_i$ for all i.*

- *$VCdim(\mathcal{H}) \leq d$: I have to show that every possible set of size $d+1$ cannot be shattered. Let's consider the generic $d+1$-set $\mathbf{x}_1, \ldots, \mathbf{x}_{d+1}$; these vectors, as we known from geometry, are linearly dependent, i.e. there exists a set of parameters $\alpha_1, \ldots, \alpha_{d+1}$ (not all zeros) such that $\sum_{i=1}^{d+1} \alpha_i \mathbf{x_i} = 0$. So, if I define the two sets $I = \{i : \alpha_i > 0\}$ and $J = \{j : \alpha_j < 0\}$, at least one of these is not empty. Let us first assume that both of them are nonempty, then:*

$$\sum_{i \in I} \alpha_i \mathbf{x}_i = \sum_{j \in J} |\alpha_j| \mathbf{x}_j$$

Now suppose that $\mathbf{x}_1, \ldots, \mathbf{x}_{d+1}$ are shattered by the class of homogeneous halfspaces. Then, there must exist a vector $\mathbf{w}$ such that $\langle \mathbf{w}, \mathbf{x_i} \rangle > 0$ for all $i \in I$ while $\langle \mathbf{w}, \mathbf{x_j} \rangle < 0$ for all $j \in J$. It follows that

$$0 < \sum_{i \in I} \alpha_i \langle \mathbf{x_i}, \mathbf{w} \rangle = \langle \sum_{\mathbf{i \in I}} \alpha_\mathbf{i} \mathbf{x_i}, \mathbf{w} \rangle = \langle \sum_{\mathbf{j \in J}} |\alpha_\mathbf{j}| \mathbf{x_j}, \mathbf{w} \rangle = \sum_{j \in J} |\alpha_j| \langle \mathbf{x_j}, \mathbf{w} \rangle < 0$$

which leads to a contradiction. Finally, if J (respectively, I) is empty, then the right-most (respectively, left-most) inequality should be replaced by an equality, which still leads to contradiction.

Also, the VC dimension of the class of nonhomogeneous halfspaces in $\mathbb{R}^d$ is $d+1$.

Care, that the VC dimension does not always correspond to the number of parameters!

Example 6.6. *We can look for example at the class $\mathcal{H} = \{h_\theta : \theta \in \mathbb{R}\}$, $h_\theta : \mathcal{X} \to \{0, 1\}$ $h_\theta = [0.5 \sin(\theta x)]$.*
It can be proved that $\mathcal{H}$ has infinite VC dimension.

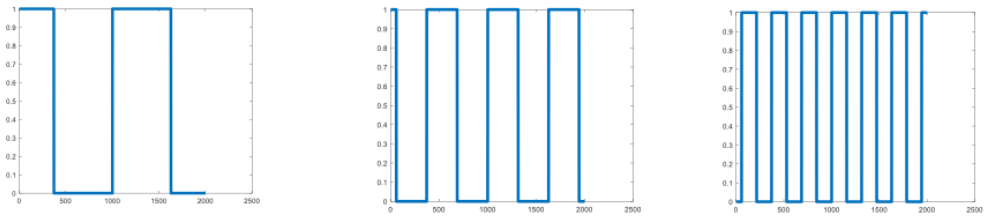


Figure 6.2: Example of a hypothesis class with infinity VC-dimension

6.1 Fundamental Theorem of Statistical Learning

We have already shown that a class of infinite VC-dimension is not learnable; however, the opposite statement is also true, as guarantee by the following theorem.

Theorem 6.1. ***(The Fundamental Theorem of Statistical Learning)***
Let $\mathcal{H}$ be an hypothesis class of function from a domain $\mathcal{X}$ to $\{0,1\}$ and let the loss function be the 0-1 loss. Then, the following are equivalent:

1. *$\mathcal{H}$ has the uniform convergence property.*
2. *Any ERM rule is a successful agnostic PAC learner for $\mathcal{H}$.*
3. *$\mathcal{H}$ is agnostic PAC learnable.*
4. *$\mathcal{H}$ is PAC learnable.*
5. *Any ERM rule is a successful PAC learner for $\mathcal{H}$.*
6. *$\mathcal{H}$ has a finite VC-dimension.*

Not only does the VC-dimension characterize PAC learnability; it even determines the sample complexity, as can be seen in the next theorem.

Theorem 6.2. ***(The Fundamental Theorem of Statistical Learning - Quantitative version)***
Let $\mathcal{H}$ be an hypothesis class of function from a domain $\mathcal{X}$ to $\{0,1\}$ and let the loss function be the 0-1 loss. Assume that $VCdim(\mathcal{H}) = d < \infty$. Then, there are absolute constants C_1, C_2 such that:

1. *$\mathcal{H}$ has the uniform convergence property with sample complexity*

$$C_1 \frac{d + \log(1/\delta)}{\varepsilon^2} \leq m_{\mathcal{H}}^{UC}(\varepsilon, \delta) \leq C_2 \frac{d + \log(1/\delta)}{\varepsilon^2}$$

2. *$\mathcal{H}$ is agnostic PAC learnable with sample complexity*

$$C_1 \frac{d + \log(1/\delta)}{\varepsilon^2} \leq m_{\mathcal{H}}(\varepsilon, \delta) \leq C_2 \frac{d + \log(1/\delta)}{\varepsilon^2}$$

3. *$\mathcal{H}$ is PAC learnable with sample complexity*

$$C_1 \frac{d + \log(1/\delta)}{\varepsilon} \leq m_{\mathcal{H}}(\varepsilon, \delta) \leq C_2 \frac{d \log(1/\varepsilon) + \log(1/\delta)}{\varepsilon}$$

Chapter 7

Model selection and validation

When approaching some practical problem, we usually can think of several algorithms that may yield a good solution, each of which might have several parameters. How can we choose the best algorithm for the particular problem at hand? And how do we set the algorithm's parameters? This task is often called **model selection**.
The most used approaches are the **Structral Risk Minimization** (SRM), that is not part of the course, and using a **validation set**.

7.1 Validation Set

The basic idea behind this approach is to partition the training data into two sets, one to be used to train all the candidate models, and the other to decide which of them yields the best results. In model selection tasks, we try to find the right balance between approximation and estimation errors. More generally, if our learning algorithm fails to find a predictor with a small risk, it is important to understand whether we suffer from overfitting or underfitting.
We would often like to get a better estimation of the true risk of the output predictor of a learning algorithm. So far we have derived bounds on the estimation error of a hypothesis class, which tell us that for all hypotheses in the class, the true risk is not very far from the empirical risk. However, these bounds might be loose and pessimistic, as they hold for all hypotheses and all possible data distributions. A more accurate estimation of the true risk can be obtained by using some of the training data as a validation set, over which one can evalutate the success of the algorithm's output predictor. The only price to pay with this method is just part of the training data.
So, let's assume that we have picked a predictor h (for example by ERM rule); let V be a set of m_ν samples used as validation set, and let L_V be the loss (in $[0,1]$) computed on this set; then, using the Hoeffding Inequality (3.1), we obtain the following theorem:

Theorem 7.1. *For every $\delta \in (0,1)$, with probability of at least $1-\delta$ over the choice of the validation set V we have:*

$$|L_V(h) - L_{\mathcal{D}}(h)| \leq \sqrt{\frac{\log\left(\frac{2}{\delta}\right)}{2m_\nu}}$$

It can be useful to notice that the bound guaranteed from the previous theorem does not depend on the algorithm or on the training set used to construct the predictor, and also it is tighter than

the "usual" bounds. The reason is that this bound is independent from the way the predictor was generated.
Let's compare, for example, the error on a predictor h obtained by applying ERM predictor with respect to a class with VC-dimension d, over a training set of m samples, that is given by the Fundamental Theorem of Statistical Learning (6.1), and the bound imposed by 7.1:

$$L_{\mathcal{D}}(h) \leq L_S(h) + \sqrt{C\frac{d + \log(1/\delta)}{m}}$$

From fundamental theorem

$$L_{\mathcal{D}}(h) \leq L_V(h) + \sqrt{\frac{\log(2/\delta)}{2m_\nu}}$$

With validation set

Therefore, taking m_ν at least at the same order of m, we obtain an estimate that is more accurate by a factor that depends on the VC-dimension.
However, since the algorithm haven't seen the validation set yet, we are protected from the risk of overfitting.

7.2 Validation for Model Selection

Validation can be naturally used for model selection.

- We first have to train different algorithms (or the same algorithm but with different parameters) on the training set, obtaining a set of ERM predictors $\mathcal{H} = \{h, h_2, \ldots, h_r\}$.
- Then, we have to choose the predictor that minimizes the error on the validation set. In other words, we apply the ERM predictor over the validation set.
- This process is very similar to learning a finite hypothesis class. The only difference is that $\mathcal{H}$ is not fixed ahead of time but rather depends on the training set. However, since the validation set is independent from the training set we get that it is also independent from $\mathcal{H}$ and therefore the same techniques we used to derive bounds for finite hypothesis class holds here as well.

Combining theorem 7.1 with union bound, in particular, we obtain the following theorem:

Theorem 7.2. *Let $\mathcal{H} = \{h_1, \ldots, h_r\}$ be an arbitrary set of predictors and assume that the loss function is in $[0, 1]$. Assume that a validation set V of size m_ν is sampled indipendent from $\mathcal{H}$. Then, with probability of at least $1 - \delta$ over the choice of V we have*

$$\forall h \in \mathcal{H},\ |L_{\mathcal{D}}(h) - L_V(h)| \leq \sqrt{\frac{\log(2|H|/delta)}{2m_\nu}}$$

This theorem essentially tells us that the error on the validation set approximates the true error as long as $\mathcal{H}$ is not too large (too many methods can bring to overfitting).

Example 7.1. *To illustrate how validation is useful we exploit against the following example in figure 7.1: we have a training set, made by the black points, that has been fitted with ERM polynomials of degree 2,3 and 10. But this time we also despict an additional validation set (white points): as can be seen, the polynomial of degree 10 has minimal training error, yet the polynomial of degree 3 has the minimal validation error, and hence it will be chosen as best model. The figure on the right is called* ***Model-Selection Curve****, and shows the training error*

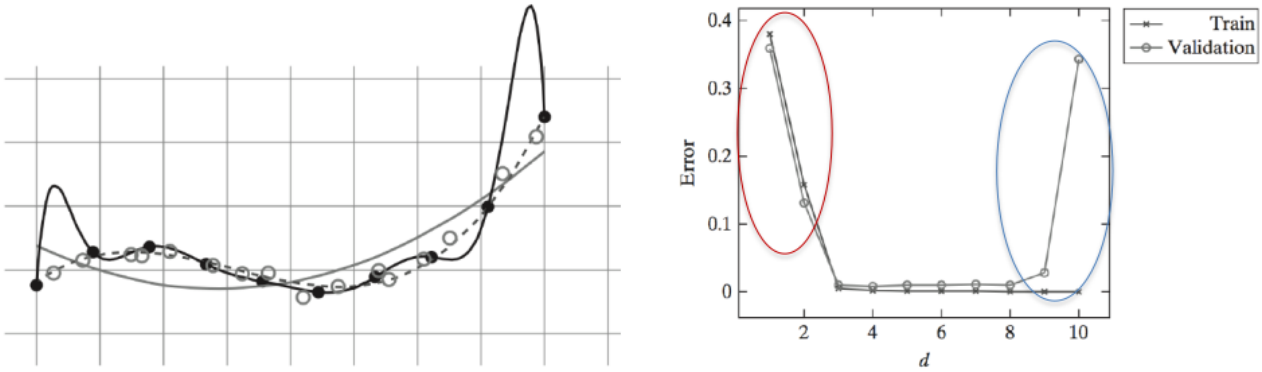


Figure 7.1: Example of Validation efficiency

and the validation error as a function of the complexity of the model considered (in this case the polynomial fitting). As can be shown, the training error is monotonically decreasing as we increase the polynomial degree (which is the complexity of the model, in this case), while, on the other hand, the validation error first decreases but then starts to increase, which indicates that we are starting to suffer from overfitting.
However, it is always useful to compute both training and validation error and compare it. If they are close, this means that the algorithm works on training data in the same way he does on the real world.

7.2.1 Grid Search for Multiple Parameters

What happen when we have to estimate one (or more) parameters with value in $\mathbb{R}$?
A possible way to solve this problem is starting with a rough grid of values and and plot the corresponding model-selection curve. On the basis of the curve, we should zoom into the correct (where "correct" is intended respect to our problem) region and start again the procedure with a finer grid. However, this research trough the grid does not always find the global optimal solution for our parameters, but it can give a reasonable approximation.

7.2.2 Train-Validation-Test Split

In most practical applications, we split the available examples in 3 sets. The first, and bigger one, is used for training the algorithm, while the second one is used as a validation set to decide which of our hypothesis is the best to describe our model. After that, we test the performances of the output predictor on the third set, and the number obtained is used as an estimator of the true error of the learned predictor.

Important: the test set is not involved in the choice of the best predictor and, if for some reason we decide to change our best hypothesis, this test set cannot be used again to estimate the true error.

7.3 K-Fold Cross Validation

The validation procedure described so far assumes that data is plentiful and that we have the ability to sample a fresh validation set. But in reality data are often scarce, and we would not "waste" data on validation. The k-fold cross validation technique is designed to give an accurate estimate of the true error without wasting too much data.
In this procedure, the original training set is partitioned into k subsets (*folds*) of size m/k. For each fold, the algorithm is trained on the set built from the union of the other folds, and then the error of its output is estimated using the starting fold. The final estimate of the true error is computed as the average of the values found for each fold. A schematic representation of the technique is figure 7.2.

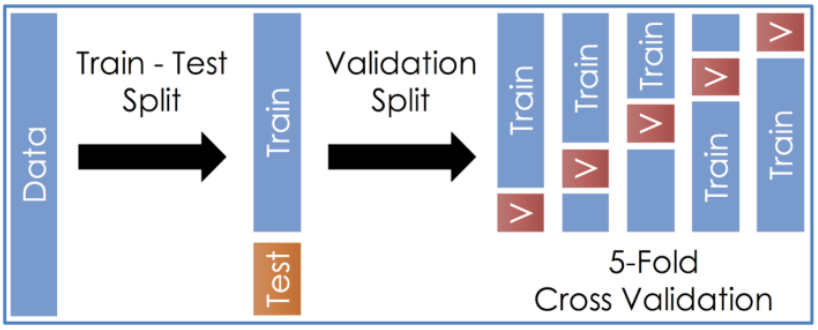


Figure 7.2: Scheme of a 5-Fold Cross Validation

K-Fold Cross Validation is often used for model selection: in this case after selecting the model, the final hypothesis is obtained from training on the entire training set. A pseudo-code of the technique used for model selection is given in box 7.3: the procedure receives an input training set S, a set of possible parameter values Θ, the number of folds k and a learning algorithm, and outputs the best parameter as well as the hypothesis trained by this parameter on the entire training set.

k-Fold Cross Validation for Model Selection

Input:
training set $(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_m, y_m)$
set of parameter values Θ
learning algorithm A
number of folds k

Partition: S in $S_1, S_2, \ldots, S_k$

foreach $\theta \in \Theta$
for $i = 1, \ldots, k$:
$h_{i,\theta} = A(S/S_i; \theta)$ \∗ removing the i-th fold ∗\
$error(\theta) = \frac{1}{k} \sum_{i=1}^{k} L_{S_i}(h_{i,\theta})$

Output:
$\theta^* = argmin_\theta \left[error(\theta)\right]$
$h_{\theta^*} = A(S; \theta^*)$

Error Decomposition

Let's recall all the error we have define until now:

- $L_{\mathcal{D}}(h^*)$ is the *approximation error*, i.e. the true error over the best hypothesis in $\mathcal{H}$.
- $L_{\mathcal{D}}(h_S) - L_{\mathcal{D}}(h^*)$ is the *estimation error*, i.e. the difference between the true error of the best hypothesis in $\mathcal{H}$ and the true error of the ERM solution.
- $L_S(h_S)$ is the *training error* i.e. the empirical error of the ERM solution on the training set S.
- $L_V(h_S)$ is the *validation error* i.e. the error over the validation set V of the ERM solution.

Thank to this we can decompose the true error of the ERM solution as follows (and that we have already described in section 4.1):

$$L_{\mathcal{D}}(h_S) = L_{\mathcal{D}}(h^*) L_{\mathcal{D}}(h_S) - L_{\mathcal{D}}(h^*)$$

As well with the training and the validation error:

$$L_{\mathcal{D}}(h_S) = (L_{\mathcal{D}}(h_S) - L_V(h_S)) + (L_V(h_S) - L_S(h_S)) + L_S(h_S)$$

However, it is more difficult to estimate the approximation error of the class, and so we give such an error decomposition, that let us to focus on training and validation error. The first term of this decomposition, $L_{\mathcal{D}}(h_S) - L_V(h_S)$, can easily bounded using theorem 7.1; when the second one, $L_V(h_S) - L_S(h_S)$, tends to be large we can say that our algorithm suffers from "overfitting"; when the last term, the empirical risk $L_S(h_S)$, is large, we can say that this is a case of "underfitting".

7.4 When Learning Fails

Consider the following scenario: you were given a learning task and have approached it with a choice of an hypothesis class, a learning algorithm, and parameters. You used a validation set to tune the parameters and tested the learned predictor on a test set. The test result, unfortunately, turn out to be unsatisfactory. What went wrong, then, and what should you do next?
There are many elements that can be changed to gain better results, for example:

- Get a larger sample
- Change the hypothesis class (enlarging it, reducing it or completely changing the parameters or the class itself)
- Change the feature representation of the data
- Change the optimization algorithm used to apply the learning rule

In order to find the best remedy, it is essential first to understand the cause of the bad performance. Remember that the approximation error does not depend on the sample size or on the algorithm being used, but only on the distribution $\mathcal{D}$ and on the hypothesis class $\mathcal{H}$. Therefore, if the approximation error is large, it will not help us to enlarge the training set size, and it also does not make sense to reduce the hypothesis class. What can be beneficial in this case is to enlarge the hypothesis class or completely change it (if we have some alternative prior knowledge in the form of a different hypothesis class). The estimation error of the class, instead, does depend on the sample size. Therefore, if we have large estimation error we can make an effort to obtain more training examples. We can also consider reducing the hypothesis class. However, it doesn't make sense to enlarge the hypothesis class in that case.
One possible way to distinguish between over and underfitting and estimate di approximation error $L_{\mathcal{D}}(h^*)$ is by plotting the so called *learning curves*. To produce a learning curve we train the algorithm on prefixes of data of increasing sizes (for example we can first train it on 10% of data, then on 20% and so on). For each of these data subgroup we calculate the training error and the validation one (on a predefine validation set), and then we plot it together as in figure 7.3. Such curves help us to distinguish the two aforementioned scenarios. In general, as long as

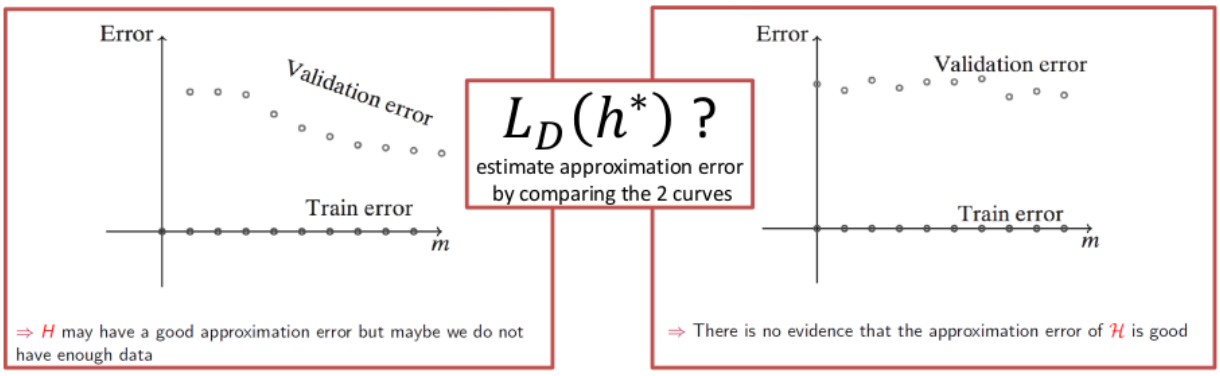


Figure 7.3: Example of learning curves. Left: scenario in which the number of examples is always smaller than the VC-dimension of the problem. Right: scenario in which the approximation error is zero and the number of examples is larger than the VC-dimension.

the approximation error is greater than zero we expect the training error to grow with the sample size, as a larger amount of data points makes it harder to provide an explanation for all of them. On the other hand, the validation error tends to decrease with the increase in the sample size. If the VC-dimension is finite, when the sample size goes to infinity, the validation and train errors converge to the approximation error.
To summarize the discussion, the following steps should be applied:

1. If learning involves parameter tuning, plot the model-selection curve to make sure that they have been tuned appropriately.

2. If the training error is excessively large, consider enlarging the hypothesis class, completely change it, or change the feature representation of the data.

3. If the training error is small, plot learning curves and try to deduce from them if the problem is the estimation error or the approximation one.

4. If the approximation error seems to be small enough, try to obtain more data. If it is not possible, consider reducing the complexity of the hypothesis class.

5. If the approximation error seems to be large as well, try to change the hypothesis class or the feature representation of the data completely.

Chapter 8

Regularization and Stability

In this chapter we are gonna introduce a new learning paradigm, called **Regularized Loss Minimization**, or RLM, for short. In RLM we minimize the sum of the empirical risk and a regularization function, that quantifies the complexity of hypotheses. Another view or regularization is as a **stabilizer** of the learning algorithm. An algorithm is considered stable if a slight change of the input does not change the output much.

8.1 Regularized Loss Minimization

RLM is a learning rule in which we jointly minimize the empirical risk and a regularization function. Formally, given an hypothesis h, characterized by a vector of parameters $\mathbf{w} = (w_1, \dots, w_d)^T \in \mathbb{R}^d$ (for example the coefficients of a linear regression or the weights of a neural network), a *regularization function* is a map $R : \mathbb{R}^d \to \mathbb{R}$, function of $\mathbf{w}$, for which the RLM rule outputs an hypothesis in

$$\underset{\mathbf{w}}{argmin} \left(L_S(\mathbf{w}) + R(\mathbf{w}) \right) \tag{8.1}$$

where $L_S(\mathbf{w})$ is the standard loss and $R(\mathbf{w})$ is a regularization term that measures, in some way, the "complexity" of the found solution, it balances between a low empirical risk and aiming at less complex hypotheses.
There are many possible regularization functions that one can use, reflecting some prior knowledge about the problem. One of the most simple regularization functions (that make use of the ℓ_2 norm), known as **Tikhonov Regularization**, is

$$R(\mathbf{w}) = \lambda \|\mathbf{w}\|^2 = \lambda \sum_{i=1}^{d} w_i^2$$

where $\lambda > 0$ is a scalar.
Using this formula, the learning rule becomes

$$A(S) = \underset{\mathbf{w}}{argmin} \left(L_S(\mathbf{w}) + \lambda \|\mathbf{w}\|^2 \right)$$

One interpretation sees the norm $\|\mathbf{w}\|^2$ as a measurement of the complexity of the hypothesis class defined by the vector $\mathbf{w}$, and the parameter λ as a coefficient used to control the trade-off between a low empirical risk and an high complexity (risk of overfitting).

8.1.1 Ridge Regression

Applying the ERM rule with Tikhonov regularization to linear regression with the square loss, we obtain the following learning rule:

$$\underset{\mathbf{w}\in\mathbb{R}^d}{argmin}\left(\lambda\|\mathbf{w}\|^2+\frac{1}{m}\sum_{i=1}^{m}\frac{1}{2}(\langle\mathbf{w},\mathbf{x_i}\rangle-y_i)^2\right)$$

Performing linear regression using this equation, i.e. find the vector $\mathbf{w}$ that minimizes the squared loss, is called **Ridge Regression**.
To solve it we have to compare the gradient of the object to zero, obtaining the set of linear equations:

$$(2\lambda m\mathbb{1}+A)\,\mathbf{w}=\mathbf{b}$$

where A and $\mathbf{b}$ are defined as

$$A=\left(\sum_{i=1}^{m}\mathbf{x_i}x_i^T\right)\qquad\text{and}\qquad\mathbf{b}=\sum_{i=1}^{m}y_i\mathbf{x_i}$$

Since A is a positive semidefinite matrix, the matrix $2\lambda m\mathbb{1}+A$ has all its eigenvalues bounded by $2\lambda m$. Hence, this matrix is invertible, and the solution to Ridge regression becomes:

$$\mathbf{w}=(2\lambda m\mathbb{1}+A)^{-1}\,\mathbf{b}$$

8.2 Stable rules do not overfit

Intuitively, a learning algorithm is stable if a small change of the input to it does not change the output much. But what "a small change on the input" means?
Let's start replacing only one sample! Let A be an algorithm, $S=(z_1,\ldots,z_m)$ be a training set of m samples, and let $A(S)$ denote the output of A. Now, let's add to S an additional example z', and define $S^{(i)}=(z_1,\ldots,z_{i-1},z',z_{i+1},\ldots,z_m)$. We want to measure the consequences of this small addition on the output of A, and we can do it by comparing the loss of the hypothesis $A(S)$ to the loss of $S(A^{(i)})$ on z_i. Intuitively, a good learning algorithm will have $\ell(A(S),z_i)-\ell(A(S^{(i)}),z_i)\geq 0$, since in the first term the algorithm does not observe the example z_i. If this difference is found quite large, we suspect that the learning algorithm might overfit. This is because the learning algorithm drastically changes its prediction on z_i if it observes it in the training set. This is formalized in the following theorem:

Theorem 8.1. *Let $\mathcal{D}$ be a distribution. Let $S=(z_1,\ldots,z_m)$ be an i.i.d. sequence of examples and let z' be another i.i.d. example. Let $U(m)$ be the uniform distribution over $[m]$. Then, for any learning algorithm*

$$\mathop{\mathbb{E}}_{S\sim\mathcal{D}^m}\left[L_{\mathcal{D}}(A(S))-L_S(A(S))\right]=\mathop{\mathbb{E}}_{(S,z')\sim\mathcal{D}^{m+1},i\sim U(m)}\left[\ell(A(S^{(i)},z_i))-\ell(A(S,z_i))\right]$$

Proof. Since S and z' are both drawn i.i.d. from $\mathcal{D}$, we have that for every i:

$$\mathop{\mathbb{E}}_{S}\left[L_{\mathcal{D}}(A(S))\right]=\mathop{\mathbb{E}}_{S,z'}\left[\ell(A(S),z')\right]=\mathop{\mathbb{E}}_{S,z'}\left[\ell(A(S^{(i)}),z_i)\right]$$

On the other hand, we can write

$$\mathop{\mathbb{E}}_{S}\left[L_S(A(S))\right] = \mathop{\mathbb{E}}_{S,i}\left[\ell(A(S), z_i)\right]$$

Combining the two equations we conclude our proof. □

When the right-side of equation in the previous theorem is small, we say that A is *stable* algorithm. Formally:

Definition 8.1. *(**On-Average-Replace-One-Stable**) Let $\varepsilon : \mathbb{N} \to \mathbb{R}$ be a monotonically decreasing function. We say that a learning algorithm A is OAROS with rate $\varepsilon(m)$ if for every distribution $\mathcal{D}$*

$$\mathop{\mathbb{E}}_{(S,z')\sim\mathcal{D}^{m+1},i\sim U(m)}\left[\ell(A(S^{(i)}, z_i)) - \ell(A(S, z_i))\right] \leq \varepsilon(m)$$

The theorem written above tells us that a learning algorithm that does not overfit is not necessary a good learning algorithm (for example, an algorithm that always outputs the same hypothesis). One should find an algorithm that both fit the training set and at the same time be stable. We will see that this will be exactly the role of the parameter λ of the RLM rule.

8.3 Tikhonov Regularization as a Stabilizer

In the previous section we saw that stable rules don't overfit. In this section we will show that applying the RLM rule with Tikhonov regularization, $\lambda\|\mathbf{w}\|^2$, leads to a stable algorithm. The main property of the Tikhonov regularization that we rely is on is that it makes the objective of RLM *strongly convex*, as defined in the following:

Definition 8.2. *(**Strongly Convex Functions**) A function f is λ-strongly convex if for all $\mathbf{w}$,$\mathbf{u}$, and $\alpha \in (0,1)$ we have*

$$f(\alpha\mathbf{w} + (1-\alpha)\mathbf{u}) \leq \alpha f(\mathbf{w}) + (1-\alpha)f(\mathbf{u}) - \frac{\lambda}{2}\alpha(1-\alpha)\|w-u\|^2$$

Clearly, every convex function is 0-strongly convex. An example of this property can be found in figure 8.1. The following lemma resume the important properties of strong convexity.

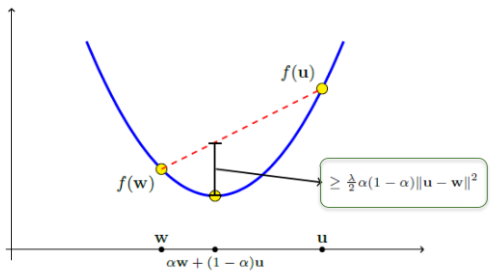


Figure 8.1: Strongly convex function

Lemma 8.1. *1. The function $f(\mathbf{w}) = \lambda\|\mathbf{w}\|^2$ is 2λ strongly convex.*

2. If f is λ-strongly convex and g is convex, then $f + g$ is λ-strongly convex.

3. If f is λ-strongly convex and $\mathbf{u}$ is a minimizer of f, then, for any $\mathbf{w}$

$$f(\mathbf{w}) - f(\mathbf{u}) \geq \frac{\lambda}{2}\|w - u\|^2$$

8.3.1 Lipschitz Loss

Recall the definition of Lipschitzness.

Definition 8.3. ***(Lipschitzness)***
Let $C \subset \mathbb{R}^d$. A function $f : \mathbb{R}^d \to \mathbb{R}^k$ is ρ-Lipschitz over C if $\forall\ \mathbf{w_1}, \mathbf{w_2} \in C$ we have that

$$\|f(\mathbf{w_1} - f(\mathbf{w_2}))\| \leq \|\mathbf{w_1} - \mathbf{w_2}\|$$

Intuitively, we are asking to a function to not change too much.
If a function is derivable and its derivative is bounded by ρ at any points, then this function is ρ-Lipschitz.
Given such definition, the following theorem holds:

Theorem 8.2. *Assume that the loss function is convex and ρ-Lipschitz continuous. Then, the RLM rule with regularizer $\lambda\|\mathbf{w}\|^2$ is OAROS (8.1) with rate $\frac{2\rho^2}{\lambda m}$. This means that, for the RLM rule:*

$$\mathop{\mathbb{E}}_{S\sim\mathcal{D}^m}\left[L_{\mathcal{D}}(A(S)) - L_S(A(S))\right] \leq \frac{2\rho^2}{\lambda m}$$

8.4 Fitting-Stability Tradeoff

We can rewrite the expected risk of a learning algorithm as

$$\mathop{\mathbb{E}}_{S}\left[L_{\mathcal{D}}(A(S))\right] = \mathop{\mathbb{E}}_{S}\left[L_S(A(S))\right] + \mathop{\mathbb{E}}_{S}\left[L_{\mathcal{D}}(A(S)) - L_S(A(S))\right]$$

The first term reflects how well $A(S)$ fits the training set while the second term reflects the difference between the true and empirical risks of $A(S)$, that, as we already shown, is equivalent to the stability of A. Since our goal is to minimize the risk of the algorithm, we need the sum of both terms to be small.
We can show that the stability term decreases as the regularization parameter, λ, increases. On the other hand, the empirical risk increases with λ. We therefore face a tradeoff between fitting and overfitting, that is quite similar to the bias-complexity tradeoff we discussed in a previous chapter.
With the following theorem (no proof) we derive bounds on the empirical risk term for the RLM rule.

Theorem 8.3. *Assume that the loss function is convex and ρ-Lipschitz. Then, the RLM rule with the regularization function $\lambda\|\mathbf{w}\|^2$ satisfies*

$$\forall \mathbf{w}^*, \quad \mathop{\mathbb{E}}_{S}\left[L_{\mathcal{D}}(A(S))\right] \leq L_{\mathcal{D}}(\mathbf{w}^*) + \lambda\|\mathbf{w}^*\|^2 + \frac{2\rho^2}{\lambda m}$$

This bound is often called an *oracle inequality*: if we think of $\mathbf{w}^*$ as an hypothesis with low risk, the bound tells us how many examples are needed so that $A(S)$ will be almost as good as $\mathbf{w}^*$, had we known the norm of $\mathbf{w}^*$. In practice, however, we usually do not know the norm of $\mathbf{w}^*$, so it is more convenient to tune λ on the basis of a validation set.

Chapter 9

Stochastic Gradient Descent

Till now, we have seen that the goal of our learning algorithm is to find a predictor that minimize the risk function $L_{\mathcal{D}}(h) = \mathbb{E}_{z\sim\mathcal{D}}[\ell(h,z)]$. But we can't directly find the mathematical minimum of such function, since it depends on the unknown distribution $\mathcal{D}$.
For this reason, in this chapter we will discuss a new learning method, called *Stochastic Gradient Descent* (SGD), that try to minimize directly the risk function $L_{\mathcal{D}}(\mathbf{w})$ using a gradient descent procedure, an iterative optimization procedure in which at each step we improve the solution by taking a step along the negative of the gradient of the function to be minimized at the current point.
Of course, in our case, since we do not know $\mathcal{D}$ we can't even know the gradient of $L_{\mathcal{D}}$: SGD circumvents this problem by allowing the optimization procedure to take a step along a random direction, as long as the expected value of the direction is the negative of the gradient. We will see that this trick is relatively simple.
The advantage of SGD, in the context of convex learning problems, over the regularized risk minimization learning rule is that SGD is an efficient algorithm that can be implemented in a few lines of code, yet still enjoys the same sample complex- ity as the regularized risk minimization rule.

9.1 Gradient Descent

Before we describe the stochastic gradient descent method, we would like to describe the standard gradient descent approach for minimizing a differentiable convex function $f(\mathbf{w})$.
The gradient of a differential function $f : \mathbb{R}^d \to \mathbb{R}$ is simply the vector of the derivatives:

$$\boldsymbol{\nabla} f = \left(\frac{\partial f(\mathbf{w})}{\partial w_1}, \ldots, \frac{\partial f(\mathbf{w})}{\partial w_d}\right)$$

As we know from math theory, the gradient is the vector that points in the direction in which the function f increases itself, in a region close to $\mathbf{w}$.
We start from an initial value, for example $\mathbf{w}^{(1)} = \mathbf{0}$, then, at each iteration, we take a step in the direction of the negative of the gradient at the current point. Update step:

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} + \eta \boldsymbol{\nabla} f(\mathbf{w}^{(t)})$$

where $\eta > 0$ is a parameter we will discuss of later. Intuitively, since the gradient points in the direction of the greatest rate of increase of f around $\mathbf{w}^{(t)}$, the algorithm makes a small step in

the opposite direction, thus decreasing the value of the function. Eventually, after T iterations, the algorithm outputs the averaged vector, $\bar{\mathbf{w}} = \frac{1}{T}\sum_{t=1}^{T}\mathbf{w}^{(t)}$. The output could also be the last vector, $\mathbf{w}^{(T)}$ (favourite solution of the professor), or the best performing vector, $\underset{t\in[T]}{argmin} f(\mathbf{w}^{(t)})$.
Another way to motivate Gradient Descent technique and the update rule above is by relying on Taylor first order approximation, that is (calculated in $\mathbf{w}$) $f(\mathbf{u}) = f(\mathbf{w}) + \langle \mathbf{u}-\mathbf{w}, \boldsymbol{\nabla}\mathbf{f}(\mathbf{w})\rangle$. If f is convex, this approximation lower bounds f:

$$f(\mathbf{u}) \leq f(\mathbf{w}) + \langle \mathbf{u}-\mathbf{w}, \boldsymbol{\nabla}\mathbf{f}(\mathbf{w})\rangle$$

9.1.1 Gradient Descent for Convex-Lipschitz Functions

To analyze the convergence rate of the GD algorithm, we limit ourselves to the case of convex-Lipschitz functions, that is the easiest one. We can write the following lemma:

Lemma 9.1. *Let f be a convex, ρ-Lipschitz function, and let $\mathbf{w}^* \in argmin_{\mathbf{w}:\|\mathbf{w}\|\leq B} f(\mathbf{w})$. If we run the GD algorithm on f for T steps with $\eta = \sqrt{\frac{B^2}{\rho^2 T}}$, then the output vector $\bar{\mathbf{w}}$ satisfies*

$$f(\bar{\mathbf{w}}) - f(\mathbf{w}^*) \leq \frac{B\rho}{\sqrt{T}}$$

Furthermore, for every $\varepsilon > 0$, to achieve $f(\bar{\mathbf{w}}) - f(\mathbf{w}^*) \leq \varepsilon$, it suffices to run the algorithm for a number of iterations

$$T \geq \frac{B^2\rho^2}{\varepsilon^2}$$

9.2 Stochastic Gradient Descent

In *stochastic gradient descent* (SGD) we do not require the update direction to be based exactly on the gradient. Instead, we allow the direction to be a random vector and only require that its expected value at each iteration will equal the gradient direction. Computing the gradient at every step, in fact, can be computationally demanding, so we should avoid it.
The explicit implementation of this algorithm is written in box 9.2, while a graphical comparison between the two techniques described until now is represented in figure 9.1.

Stochastic Gradient Descent (SGD) for minimizing $f(\mathbf{w})$

Input:
Scalar $\eta > 0$
Integer $T > 0$

Initialize: $\mathbf{w}^{(1)} = \mathbf{0}$

for $t = 1, 2, \ldots, T$:
choose $\mathbf{v_t}$ from a random distribution such that $\mathbb{E}\left[\mathbf{v_t}|\mathbf{w}^{()}\right] \in \partial(\mathbf{w}^{()})$
update: $\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta\mathbf{v_t}$

Output: $\bar{\mathbf{w}} = \frac{1}{T}\sum_{t=1}^{T}\mathbf{w}^{(t)}$ or $\bar{\mathbf{w}} = \mathbf{w}^{(T)}$ or $\bar{\mathbf{w}} = \frac{1}{T-T_0}\sum_{t=T_0}^{T}\mathbf{w}^{(t)}$

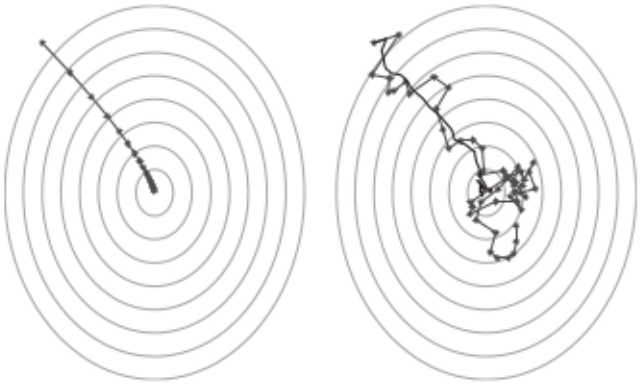


Figure 9.1: An illustration of the gradient descent algorithm (left) and the stochastic gradient descent algorithm (right). The function to be minimized is $25(x+6)^2+(y-8)^2$. For the stochastic case, the solid line depicts the averaged value of $\mathbf{w}$.

9.2.1 SGD vs GD

Why should we use Stochastic Gradient Descent method respect to the other one, to solve machine learning problems?
We have seen the method of empirical risk minimization, where we minimize the empirical risk, $L_S(\mathbf{w})$, as an estimate to minimizing $L_{\mathcal{D}}(\mathbf{w})$. *SGD* allows us to take a different approach and minimize $L_{\mathcal{D}}(\mathbf{w})$ directly.
Since we do not know the distribution $\mathcal{D}$, we cannot simply calculate $\boldsymbol{\nabla} L_{\mathcal{D}}(\mathbf{w}^{(t)})$ and minimize it with the *GD* method. And then this technique is very expensive from a computational point of view, since it needs to process all the training data at each iteration, because the gradient has to be computed for every couple $(\mathbf{x}_i, y_i \in S)$. And this becomes a problem for large dataset, so it isn't the best method for real world ML problems.
On the other hand *SGD* is much faster, since at each step only one sample (taken randomly) is used for the computation and, in theory we could get the final solution at the first steps too! So, as many other stochastic methods, its efficiency comes up for complex problems and large datasets. Anyway also this algorithm has some "problems". As we can see in picture 9.1 the "trajectory" drown by the method is "noisier" than the other, and there is a possibility of jumping out from local minima. For this reason we often look for some approaches to stabilize the solution, for example taking the average of a set of final iterations (or all of these), to take into account fluctuations phenomena.

9.2.2 SGD for Risk Minimization

There are many ML tasks, in which the Stochastic Gradient Descent is used, for example:

- Risk Minimization (ERM)
- Regularized Loss Minimization (RLM)
- Support Vector Machine (SVM)
- Neural Networks (NN)

In this section we will discuss about the use of SGD for risk minimization.
Let us first consider the case of differentiable functions, which contains also $L_{\mathcal{D}}$. The construction of the random vector $\mathbf{v_t}$ will be as follows: first we sample $z \sim \mathcal{D}$, then we define $\mathbf{v_t}$ to be de gradient of the function $\ell(\mathbf{w}, z)$ with respect to $\mathbf{w}$, at the point $\mathbf{w}^{(t)}$. Then, by linearity of the gradient, we have

$$\mathbb{E}\left[\mathbf{v_t}|\mathbf{w}^{(t)}\right] = \mathop{\mathbb{E}}_{z\sim\mathcal{D}}\left[\boldsymbol{\nabla}\ell(\mathbf{w}^{(t)}, z)\right] = \boldsymbol{\nabla}\mathop{\mathbb{E}}_{z\sim\mathcal{D}}\left[\ell(\mathbf{w}^{(t)}, z)\right] = \boldsymbol{\nabla}L_{\mathcal{D}}(\mathbf{w}^{(t)})$$

The gradient of the loss function $\ell(\mathbf{w}, z)$ at $\mathbf{w}^{(t)}$ is therefore an unbiased estimate of the gradient of the risk function $L_{\mathcal{D}}(\mathbf{w}^{(t)})$ and is easily constructed by sampling a single fresh example $z \sim \mathcal{D}$ at each iteration t.
To summarize, the stochastic gradient descent framework for minimizing the risk we write the box 9.2.2.

Stochastic Gradient Descent (SGD) for minimizing $L_{\mathcal{D}}(\mathbf{w})$

Input:
Scalar $\eta > 0$
Integer $T > 0$

Initialize: $\mathbf{w}^{(1)} = \mathbf{0}$

for $t = 1, 2, \ldots, T$:
sample $z \sim \mathcal{D}$
pick $\mathbf{v_t} = \boldsymbol{\nabla}\ell(\mathbf{w}^{(t)}, z)$
update: $\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta\mathbf{v_t}$

Output: $\bar{\mathbf{w}} = \frac{1}{T}\sum_{t=1}^{T}\mathbf{w}^{(t)}$ or $\bar{\mathbf{w}} = \mathbf{w}^{(T)}$

We shall now use our analysis of SGD to obtain a sample complexity analysis for learning convex-Lipschitz-bounded problems. In fact, we can prove that:

Theorem 9.1. *Consider a convex-Lipschitz-bounded learning problem with parameters ρ, B. Then, for every $\varepsilon > 0$, if we run the SGD method for minimizing $L_{\mathcal{D}}(\mathbf{w})$ with a number of iterations (i.e., number of examples)*

$$T \geq \frac{B^2\rho^2}{\varepsilon^2}$$

and with $\eta = \sqrt{\frac{B^2}{\rho^2 T}}$, the output of SGD satisfies:

$$\mathbb{E}\left[L_{\mathcal{D}}(\bar{\mathbf{w}})\right] \leq \min_{\mathbf{w}\in\mathcal{H}} L_{\mathcal{D}}(\mathbf{w}) + \varepsilon$$

9.2.3 SGD for λ-strongly convex functions and RLM

There is also a variant of *SGD* that enjoys a faster convergence rate for problems in which the objective function is strongly convex, where the definition of this strongly convexity has been written in 8.2. In particular, we can show (but we won't) that it is a good strategy to use an adaptive step size of value $\eta_t = \frac{1}{\lambda t}$.

But there is a particular case of strongly convex function that can be very useful for us, that is the Regularized Loss Minimization (RLM): we can, in fact, rewrite the associated optimization problem 8.1 (dividing, for convenience, λ by 2) as

$$\min_{\mathbf{w}} \left(\frac{\lambda}{2} \|\mathbf{w}\|^2 + L_S(\mathbf{w}) \right)$$

Then, defining $f(\mathbf{w}) = \frac{\lambda}{2}\|\mathbf{w}\|^2 + L_S(\mathbf{w})$, we notice that f is a $2\frac{\lambda}{2} = \lambda$-strongly function, and for this reason the problem can be solved in a efficient way with learning rate $\eta_t = \frac{1}{\lambda t}$. To analyze the resulting algorithm, we first rewrite the update rule:

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} + \frac{1}{\lambda t}\left(\lambda \mathbf{w}^{(t)} + \mathbf{v_t}\right) = \left(1 - \frac{1}{t}\right)\mathbf{w}^{(t)} - \frac{1}{\lambda t}\mathbf{v_t} = \frac{t-1}{t}\mathbf{w}^{(t)} - \frac{1}{\lambda t}\mathbf{v_t} =$$
$$= \frac{t-1}{t}\left(\frac{t-2}{t-1}\mathbf{w}^{(t-1)} - \frac{1}{\lambda(t-1)}\mathbf{v_{t-1}}\right) - \frac{1}{\lambda t}\mathbf{v_t} = -\frac{1}{\lambda t}\sum_{i=1}^{t}\mathbf{v_i}$$

Moreover, if loss is a ρ-Lipschitz function, there is a theorem (that we will not discuss) that guarantee that after T iterations:

$$\mathbb{E}\left[f(\bar{\mathbf{w}})\right] - f(\mathbf{w}^*) \leq \frac{4\rho^2}{\lambda t}(1 + \log(T)$$

SGD: Issues

The selection of the learning rate η is a critical point:

- if η is too small, the optimization will be stable, but the convergence can be very slow.
- if η is too large, the convergence is faster, but the optimization can become very unstable.
- the various parameters have different behaviors and the learning rate could be too fast for some and too slow for others.

One simple solution to this problem can be the use of *adaptive learning rates*, that are variables that change during the learning process to better approximate the correct solution. Anyway, there aren't general rules according which the parameter should change: we could pre-define a schedule, or we could tell the program to reduce η every time the loss becomes too small. However these approaches requires rules and thresholds to be defined in advance and thus are difficult to adapt to different problems. Sometimes is also better to not update all parameters to the same extent, but perform a larger update for some, and smaller for others.

Momentum

The stochastic gradient descent method has troubles (i.e. it oscillates) in areas where he surface curves much more steeply in one dimension than in another (which are common around local maxima or minima). In this case the correction can be applied modifying directly the update rule, adding a new *momentum* term, that helps accelerating SGD in the relevant direction avoiding oscillations. An example of corrected updating rule is:

$$\mathbf{v_t} = \gamma \mathbf{v_{t-1}} + \eta \boldsymbol{\nabla}_\theta \mathbf{J}(\theta)$$

where γ is the momentum parameter, and it is usually set at 0.9.
Using momentum is like pushing a ball down a hill. The ball accumulates momentum as it rolls downhill, becoming faster and faster on the way. The same thing happens to our parameter updates: the momentum term increases for dimensions whose gradients point in the same directions and reduces updates for dimensions whose gradients change directions. As a result, we gain faster convergence and reduced oscillation.

Advanced SGD Schemes

- **Adagrad**: adapts the learning rate for each parameter independently.
  - It performs smaller updates (i.e. low learning rates) for parameters associated with frequently occurring features.
  - It performs larger updates (i.e. high learning rates) for parameters associated with infrequent features.
- **Adadelta** and **RMSprop**: improved versions of Adagrad
- **Adam**: Adaptive Moment Estimation
  - It also computes adaptive learning rates for each parameter.
  - It combines ideas from Adagrad and momentum.
  - Whereas momentum can be seen as a ball running down a slope, Adam behaves like a heavy ball with friction, which thus prefers flat minima in the error surface.

Chapter 10

Support Vector Machines

In this chapter we will discuss a very useful machine learning tool: the support vector machine paradigm (SVM) for learning linear predictors in high dimensional feature spaces. The high dimensionality of the feature space raises both sample complexity and computational complexity challenges.
Let $S = (\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_m, y_m)$ be a training set of examples, where each $\mathbf{x}_i \in \mathbb{R}^d$ and $y_i \in \{-1, 1\}$. As we know from a previous chapter, we can say that this training set is linearly separable, if there exists an halfspace $(\mathbf{w}, b)$ such that $y_i = sign(\langle \mathbf{w}, \mathbf{x_i} \rangle + b) \geq 0$ for all i (or, equivalently $y_i(\langle \mathbf{w}, \mathbf{x_i} \rangle + b) > 0$). All halfspaces $(\mathbf{w}, b)$ that satisfy this condition are ERM hypotheses (their 0-1 error is zero, which is the minimum possible error); so for any separable training sample, there are many ERM halfspaces. Which one of them should the learner pick? Let's consider, for example, the training set described in the picture 10.1.

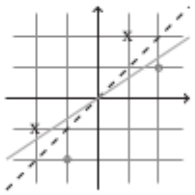


Figure 10.1: Multiple solution of an halfspace problem

While both the dashed and solid hyperplanes separate the four examples, our intuition would probably lead us to prefer the dashed hyperplane over the solid one.
One way to formalize this intuition is using the concept of **margin**.
The margin of a hyperplane with respect to a training set is defined to be the minimal distance between a point in the training set and the hyperplane. If an hyperplane has a large margin, then it will still separate the training set even if we slightly perturb each instance. Recall also that, given a separating hyperplane, defined by $L = \{\mathbf{v} : \langle \mathbf{v}, \mathbf{w} \rangle + b = 0\}$, and given a sample $\mathbf{x}$, the distance of $\mathbf{x}$ to L is

$$d(\mathbf{x}, L) = \min \{\|\mathbf{x} - \mathbf{v}\| : \mathbf{v} \in L\}$$

Moreover, the following theorem holds:

Theorem 10.1. *The distance between a point* $\mathbf{x}$ *and the hyperplane defined by* $(\mathbf{w}, b)$*, where* $\|\mathbf{w}\| = 1$ *is* $|\langle \mathbf{w}, \mathbf{x} \rangle + b|$.

On the basis of this theorem, the closest point in the training set to the separating hyperplane (i.e. the margin) is $\min_i |\langle \mathbf{w}, \mathbf{x} \rangle + b|$.

10.1 Hard-SVM

$Hard - SVM$ is the learning rule in which we return an ERM hyperplane that separates the training set with the largest possible margin (obviously, it works only for linearly separable data). Given above the formula of the distance between a point $\mathbf{x}$ and an hyperplane with $\|\mathbf{w}\| = 1$, the Hard-SVM rule is

$$\underset{(\mathbf{w},b):\|\mathbf{w}\|=1}{argmax} \min_{i\in[m]} |\langle \mathbf{w}, \mathbf{x_i} \rangle + b| \quad \text{such that} \quad \forall\, i\ y_i \left(\langle \mathbf{w}, \mathbf{x_i} \rangle + b\right) > 0$$

Whenever there is a solution to the preceding problem (i.e., we are in the separable case), we can write an equivalent problem as follows:

$$\underset{(\mathbf{w},b):\|\mathbf{w}\|=1}{argmax} \min_{i\in[m]} y_i \left(\langle \mathbf{w}, \mathbf{x_i} \rangle + b\right) > 0 \tag{10.1}$$

Next, we give another equivalent formulation of the $Hard - SVM$ rule as a quadratic optimization problem (in which the objective is a convex quadratic function and the constraints are linear inequalities).

Hard-SVM

Input:
$S = (\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_m, y_m)$
Solve:
$(\mathbf{w}_0, b_0) = \underset{(\mathbf{w},b)}{argmin} \|\mathbf{w}\|^2 \quad \text{such that} \quad \forall i,\ y_i \left(\langle \mathbf{w}, \mathbf{x_i} \rangle + b\right) \geq 1$
Output:
$\hat{\mathbf{w}} = \frac{\mathbf{w_0}}{\|\mathbf{w_0}\|}$, $\hat{b} = \frac{b_0}{\|\mathbf{w_0}\|}$

The lemma that follows shows that the output of Hard-SVM is indeed the separating hyperplane with the largest margin. Intuitively, Hard-SVM looks for $\mathbf{w}$ of minimal norm among all the vectors that separate the data and for which $|\langle \mathbf{w}, \mathbf{x_i} \rangle + b| \geq 1$ for all i. In other words, we enforce the margin to be 1, but now the units in which we measure the margin scale with the norm of $\mathbf{w}$. Therefore, finding the largest margin halfspace boils down to finding $\mathbf{w}$ whose norm is minimal. Formally:

Lemma 10.1. *The output of Hard-SVM (10.1) is a solution for 10.1*

Proof. Let $(\mathbf{w}^*, b^*)$ be a solution of equation 10.1 and define the margine achieved by them as $\gamma^* = \min_{i\in[m]} y_i \left(\langle \mathbf{w}^*, \mathbf{x_i} \rangle + b^*\right)$. Therefore, for all i we have

$$y_i \left(\langle \mathbf{w}^*, \mathbf{x_i} \rangle + b^*\right) \geq \gamma^* \qquad \text{or, equivalently} \qquad y_i \left(\langle \frac{\mathbf{w}^*}{\gamma^*}, \mathbf{x_i} \rangle + \frac{b^*}{\gamma^*}\right) \geq 1$$

Hence, the pair $(\frac{\mathbf{w}^*}{\gamma^*}, \frac{b^*}{\gamma^*})$ satisfies the conditions of the quadratic optimization problem (10.1). Therefore $\|\mathbf{w_0}\| \leq \frac{\mathbf{w}^*}{\gamma^*} = \frac{1}{\gamma^*}$. It follows that for all i:

$$y_i \left(\langle \hat{\mathbf{w}}, \mathbf{x_i} \rangle + \hat{b}\right) = \frac{1}{\|\mathbf{w_0}\|} y_i \left(\langle \mathbf{w_0}, \mathbf{x_i} \rangle + b_0\right) \geq \frac{1}{\|\mathbf{w_0}\|} \geq \gamma^*$$

Since $\|\hat{\mathbf{w}}\| = 1$ we obtain that $(\hat{\mathbf{w}}, \hat{b})$ is an optimal solution for 10.1. □

10.1.1 The Homogeneous Case

It is often more convenient to consider homogenous halfspaces, namely, halfspaces that pass through the origin and are thus defined by $sign(\langle \mathbf{w}, \mathbf{x} \rangle)$, where the bias term b is set to be zero. Hard-SVM for homogenous halfspaces amounts to solving

$$\mathbf{w_0} = \underset{\mathbf{w}}{argmin} \|\mathbf{w}\|^2 \qquad \text{such that for all i} \qquad y_i \langle \mathbf{w}, \mathbf{x_i} \rangle \geq 1 \tag{10.2}$$

As we discussed in a previous chapter, we can reduce the problem of learning nonhomogenous halfspaces to the problem of learning homogenous halfspaces by adding one more feature to each instance of x_i, thus increasing the dimension to $d+1$. Notice that, however, the optimization problem given in box 10.1 does not regularize the bias term b, while if we learn a homogenous halfspace in $\mathbb{R}^{d+1}$ using that implementation then we regularize the bias term (i.e., the $d+1$ component of the weight vector) as well. However, regularizing b usually does not make a significant difference to the sample complexity.

10.2 Support Vectors

The name "Support Vector Machine" stands for the fact that the solution of the Hard-SVM, $\mathbf{w}_0$, is supported (i.e. is the linear span of) the examples that are exactly at the distance of $1/\|\mathbf{w}_0\|$ from the separating hyperplane. So these *support vectors* are the one at a minimum distance from $\mathbf{w}_0$ and in particular they are the only training vector that matter for defining the solution of the algorithm.

Theorem 10.2. *Let $\mathbf{w}_0$ be defined as before as $\mathbf{w}_0 = \min_{\mathbf{w}} \|\mathbf{w}\|^2$ such that for all i $y_i \langle \mathbf{w}, \mathbf{x_i} \rangle \geq 1$ and let $I = \{i : |\langle \mathbf{w_0}, \mathbf{x_i} \rangle|\} = 1$ (i.e. the set that contains the indices of support vectors). Then, there exist coefficients $\alpha_1, \ldots, \alpha_m$ such that*

$$\mathbf{w}_0 = \sum_{i \in I} \alpha_i \mathbf{x_i}$$

The examples $\{\mathbf{x_i} : i \in I\}$ are the Support Vectors.

Notice that solving Hard-SVM is equivalent to find the coefficients α_i for the support vectors, since for all the others $\alpha_i = 0$.

10.3 Duality

Historically, many of the properties of SVM have been obtained by considering the *dual* of equation 10.2. Our presentation of SVM does not rely on duality so, for completeness, we present the same formula re-written as a maximization problem:

$$\max_{\alpha \in \mathbb{R}^m : \alpha \geq 0} \sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_{i=1}^{m} \sum_{j=1}^{m} \alpha_i \alpha_j y_i y_j \langle \mathbf{x_i}, \mathbf{x_j} \rangle$$

Note that the dual problem only involves inner products between instances and does not require direct access to specific elements within an instance. This property is important when implementing SVM with kernels, as we will discuss later.

10.4 Soft SVM

The Hard-SVM formulation assumes that the training set is linearly separable, which is a rather strong assumption. **Soft-SVM** can be viewed as a relaxation of the Hard-SVM rule that can be applied even if the training set is not linearly separable, and also that take into account the violation of the condition of separability into the objective function.
The optimization rule described for Hard-SVM impose to **every** sample of the training set to satisfy the relation $y_i\left(\langle \mathbf{w}, \mathbf{x_i}\rangle + b\right) \geq 1$ which is a quite hard constrain. A natural relaxation is to allow the constraint to be violated for some of the examples in the training set. This can be modeled by introducing non negative slack variables, $\xi_1, \ldots, \xi_m$, and replacing the request above with

$$y_i\left(\langle \mathbf{w}, \mathbf{x_i}\rangle + b\right) \geq 1 - \xi_i \tag{10.3}$$

so using these variable ξ_i to quantify how much the constrain is violated.
Soft-SVM jointly minimizes the norm of $\mathbf{w}$ (corresponding to the margin) and the average of ξ_i (corresponding to the constraint violation from misclassified points). The trade off between the two terms is controlled by a parameter λ. This leads to the Soft-SVM optimization problem, in the box 10.4.

Soft-SVM

Input:
$S = (\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_m, y_m)$
Parameter:
$\lambda > 0$
Solve:
$\min_{\mathbf{w},b,\xi}\left(\lambda\|\mathbf{w}\|^2 + \frac{1}{m}\sum_{i=1}^{m}\xi_i\right)$ such that $\forall i \quad y_i\left(\langle \mathbf{w}, \mathbf{x_i}\rangle + b\right) \geq 1 - \xi_i$ and $\xi_i \geq 0$
Output:
$\mathbf{w}, b$

We can rewrite condition 10.3 as a regularized loss minimization problem, defining a new loss function called **Hinge Loss**:

$$\ell^{hinge}\left((\mathbf{w}, b), (\mathbf{x}, y)\right) = \max\left\{0, 1 - y\left(\langle \mathbf{w}, \mathbf{x}\rangle + b\right)\right\}$$

An equivalent form of 10.3, in function of the averaged hinge loss on a training set S is

$$\min_{\mathbf{w},b}\left(\lambda\|\mathbf{w}\|^2 + \underbrace{\frac{1}{m}\sum_{i=1}^{m}\ell^{hinge}\left((\mathbf{w}, b), (\mathbf{x}_i, y_i)\right)}_{L_S^{hinge}(\mathbf{w},b)}\right)$$

It is often more convenient to consider Soft-SVM for learning a homogenous halfspace, where the bias term b is set to be zero, which yields the following optimization problem:

$$\min_{\mathbf{w}}\left(\lambda\|\mathbf{w}\|^2 + L_S^{hinge}(\mathbf{w})\right)$$

where

$$L_S^{hinge}(\mathbf{w}) = \frac{1}{m}\sum_{i=1}^{m}\max\left\{0, 1 - y\langle \mathbf{w}, \mathbf{x_i}\rangle\right\}$$

To solve problems like this we can, as usually, use one of the standard solvers for optimization problems, or use the stochastic gradient descent, as we do in the following subsection.

10.4.1 SGD for Soft-SVM

In this subsection we describe a very simple algorithm for solving the optimization problem of Soft-SVM (relying on the SGD framework), namely:

$$\min_{\mathbf{w},b}\left(\frac{\lambda}{2}\|\mathbf{w}\|^2+\frac{1}{m}\sum_{i=1}^{m}\max\left\{0,1-y\left(\langle\mathbf{w},\mathbf{x_i}\rangle+b\right)\right\}\right) \tag{10.4}$$

where the factor $\frac{1}{2}$ in the regularization term is added to simplify some computations.
Now, we apply the SGD rules for RLM problem (9.2.2) to the loss function

$$f^{hinge}(\mathbf{w})=\max\left\{0,1-y\left(\langle\mathbf{w},\mathbf{x}\rangle+b\right)\right\}$$

for which the (sub)gradient at $\mathbf{w}$ is

$$\mathbf{v^{hinge}}=\begin{cases}0 & if\ 1-y\langle\mathbf{w},\mathbf{x}\rangle\leq 0\\ -y\mathbf{x} & if\ 1-y\langle\mathbf{w},\mathbf{x}\rangle\leq 0\end{cases}$$

So, the update rule becomes:

$$\mathbf{w}^{(t+1)}=\mathbf{w}^{(t)}-\eta\mathbf{v}^{(t)}\qquad\text{or}\qquad\mathbf{w}^{(t+1)}=-\frac{1}{\lambda t}\sum_{j=1}^{t}\mathbf{v}^{(t)}$$

where the first relation comes from the "standard" version of SGD, while the second one from the variant of SGD for λ-strongly convex functions.
To summarize the algorithm needed to implement SGD to solve Soft-SVM optimizatio problem, we first define

$$\theta^{(t)}=-\sum_{j=1}^{t-1}\mathbf{v_j}$$

and we obtain the procedure written in box 10.4.1.

SGD for solving Soft-SVM

Goal: Solve equation 10.4
Initialize: $\theta^{(1)}=\mathbf{0}$
Parameter: T
Algorithm:
for $t=1,\ldots,T$
$\mathbf{w}^{(t)}=\frac{1}{\lambda t}\theta^{(t)}$
choose i randomly from $\{1,\ldots,m\}$
if $y_i\langle\mathbf{w^{(t)}},\mathbf{x_i}\rangle<1$ then $\theta^{(t+1)}=\theta^{(t)}+y_i\mathbf{x_i}$
else: $\theta^{(t+1)}=\theta^{(t)}$

Output:
$\bar{\mathbf{w}}=\frac{1}{T}\sum_{t=1}^{T}\mathbf{w}^{(t)}$ or $\bar{\mathbf{w}}=\mathbf{w}^{(T)}$

As we can notice, the Hinge loss allow us to correct the update rule by a quantity $-y_i\mathbf{x_i}$ only when the sample is misclassified, leaving it unchanged when the sample is correctly classified.

Chapter 11

Kernel Methods

In the previous chapter we described the *SVM* paradigm for learning halfspaces in high dimensional feature spaces. This enables us to enrich the expressive power of halfspaces by first mapping the data into a high dimensional feature space, and then learning a linear predictor in that space. While this approach greatly extends the expressiveness of halfspace predictors, it raises both sample complexity and computational complexity challenges. In the previous chapter we tackled the sample complexity issue using the concept of margin. In this chapter we tackle the computational complexity challenge using the method of *kernels*.

11.1 Embeddings into feature spaces

Example 11.1. *As we saw, SVM is a real powerful algorithm, but still limited to linear models, that cannot always be used (directly!). Recall, for example, the calculation of the VC-dimension of threshold functions in* $\mathbb{R}$*. As we can see from figure 11.1, our data is not linearly separable with a traditional linear predictor, so how we can solve our problem?*
An idea can be the application of a non linear transformation to each point in the training set, followed by the learning process with a predictor in the transformed space.

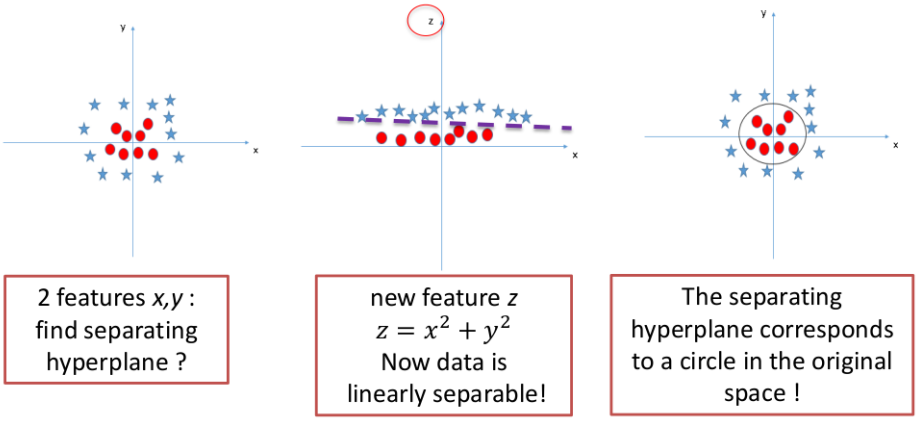


Figure 11.1: Solution to the halfspace problem in 1d

The generalized rule of the ideas used in the previous example (defining a *non linear* map from the input space to a new one, typically larger) can be written as follows:

- Given a domain set $\mathcal{X}$ and a learning task, choose a mapping $\psi : \mathcal{X} \to \mathcal{F}$, for some **feature space** $\mathcal{F}$, that is nothing but the range of ψ and usually coincide with $\mathbb{R}^n$ for some n, or also with an arbitrary Hilbert space (even of infinite size).
- Given a sequence of labeled samples $S = (\mathbf{x_1}, y_1), \ldots, (\mathbf{x_m}, y_m)$, map them to $\hat{S} = (\psi(\mathbf{x_1}), y_1), \ldots, (\psi(\mathbf{x_m}), y_m)$.
- Train a linear predictor h over $\hat{S}$.
- Predict the label of a test point $\mathbf{x}$, to be $h(\psi(\mathbf{x}))$.

The success of this learning paradigm depends on choosing a good ψ for a given learning task: that is, a ψ that will make the image of the data distribution (close to being) linearly separable in the feature space, thus making the resulting algorithm a good learner for a given task. Picking such an embedding requires prior knowledge about that task.

11.2 The Kernel Trick

We have just seen that embedding the input space into some higher dimensional feature space make halfspaces more expressive. However, the computational complexity can become really huge, due to the fact that we are computing linear separators over very high dimensional data. The common solution to this problem is *kernel based learning*. The term "kernels" is used in this context to describe inner products in the feature space. Given an embedding ψ of some domain space $\mathcal{X}$ into some Hilbert space, we define the kernel function

$$K(\mathbf{x}, \mathbf{x}') = \langle \psi(\mathbf{x}), \psi(\mathbf{x}') \rangle$$

One can think of K as specifying similarity between instances and of the embedding ψ as mapping the domain set $\mathcal{X}$ into a space where these similarities are realized as inner products. It turns out that many learning algorithms for halfspaces can be carried out just on the basis of the values of the kernel function over pairs of domain points. The main advantage of such algorithms is that they implement linear separators in high dimensional feature spaces without having to specify points in that space or expressing the embedding ψ explicitly.

Example 11.2. *To provide an example of the advantages of using this kernel method, let's consider a second degree polynomial*

$$\psi(\mathbf{x}) = (1, x_1, x_2, \ldots, x_d, x_1x_1, x_2x_2, \ldots, x_dx_d)^T \qquad \textit{with } \mathbf{x} \in \mathbb{R}^d$$

The dimension of $\psi(\mathbf{x}) = 1 + d + d^2$.

$$\langle \psi(\mathbf{x}), \psi(\mathbf{x}') \rangle = 1 + \sum_{i=1}^{d} x_i x_i' + \sum_{i=1}^{d} \sum_{j=1}^{d} x_i x_j x_i' x_j' \qquad \Theta(d^2)$$

Also note that

$$\sum_{i=1}^{d} \sum_{j=1}^{d} x_i x_j x_i' x_j' = \left(\sum_{i=1}^{d} x_i x_i' \right) \left(\sum_{j=1}^{d} x_j x_j' \right) = \left(\langle \mathbf{x}, \mathbf{x}' \rangle \right)^2$$

Therefore

$$K_\psi(\mathbf{x}, \mathbf{x}') = \langle \psi(\mathbf{x}), \psi(\mathbf{x}') \rangle = 1 + \langle \mathbf{x}, \mathbf{x}' \rangle + \left(\langle \mathbf{x}, \mathbf{x}' \rangle \right)^2 \qquad \Theta(d)$$

As we can see, computing $\psi(\mathbf{x})$ requires a time of the order of $\Theta(d^2)$, while computing $K_\psi(\mathbf{x}, \mathbf{x}')$ with the last formula just a $\Theta(d)$. So, the kernel trick guarantee us a computational advantage preventing us to waste resources on the calculation of $\psi(\mathbf{x})$.

11.2.1 Application to SVM

In the previous chapter we saw that regularizing the norm of $\|\mathbf{w}\|$ yields a small sample complexity even if the dimensionality of the feature space is high. Interestingly, as we will show later, regularizing the norm of $\mathbf{w}$ is also helpful in overcoming the computational problem. To do so, first note that all versions of the SVM optimization problem we have derived are instances of the following general problem:

$$\min_{\mathbf{w}} \left(f\left(\langle \mathbf{w}, \psi(\mathbf{x_1}) \rangle, \ldots, \langle \mathbf{w}, \psi(\mathbf{x_m}) \rangle \right) + R(\|\mathbf{w}\|) \right) \tag{11.1}$$

where $f : \mathbb{R}^m \to \mathbb{R}$ is an arbitrary function and $R : \mathbb{R}_+ \to \mathbb{R}$ is a monotonically non decreasing function. The following theorem shows that there exists an optimal solution for these kind of problems, that lies in the span of $\{\psi(\mathbf{x_1}, \ldots, \psi(\mathbf{x_m}))\}$.

Theorem 11.1. ***(Representer Theorem)***
Assume that ψ is a mapping from $\mathcal{X}$ to an Hilbert space. Then, there exists a vector $\alpha \in \mathbb{R}^m$ such that $\mathbf{w} = \sum_{i=1}^m \alpha_i \psi(\mathbf{x_i})$ is an optimal solution for the equation 11.1.

Proof. Let $\mathbf{w}^*$ be an optimal solution of equation 11.1. Since $\mathbf{w}^*$ is an element of an Hilbert space, we can rewrite $\mathbf{w}^*$ as

$$\mathbf{w}^* = \sum_{i=1}^m \alpha_i \psi(\mathbf{x}_i) + \mathbf{u}$$

where $\langle \mathbf{u}, \psi(\mathbf{x_i}) \rangle = 0$ for all i. Set $\mathbf{w} = \mathbf{w}^* - \mathbf{u}$. Clearly, $\|\mathbf{w}^*\|^2 = \|\mathbf{w}\|^2 + \|\mathbf{u}\|^2$, thus $\|\mathbf{w}\| \leq \|\mathbf{w}^*\|$. Since R is non decreasing, we obtain that $R(\|\mathbf{w}\|) \leq R(\|\mathbf{w}^*\|)$. Additionally, for all i we have that

$$y_i \langle \mathbf{w}, \psi(\mathbf{x_i}) \rangle = y_i \langle \mathbf{w}^* - \mathbf{u}, \psi(\mathbf{x_i}) \rangle = y_i \langle \mathbf{w}^*, \psi(\mathbf{x_i}) \rangle$$

hence

$$f\left(y_1 \langle \mathbf{w}, \psi(\mathbf{x_1}) \rangle, \ldots, y_m \langle \mathbf{w}, \psi(\mathbf{x_m}) \rangle \right) = f\left(y_1 \langle \mathbf{w}^*, \psi(\mathbf{x_1}) \rangle, \ldots, y_m \langle \mathbf{w}^*, \psi(\mathbf{x_m}) \rangle \right)$$

We have shown that the objective of equation 11.1 at $\mathbf{w}$ cannot be larger than the objective at $\mathbf{w}^*$ and therefore $\mathbf{w}$ is also an optimal solution. Since $\mathbf{w} = \sum_{i=1}^m \alpha_i \psi(\mathbf{x_i})$ we conclude our proof. □

The main consequence of this theorem is that we can optimize the problem for the coefficients α_i, getting another problem that this time depends only on $K(\mathbf{x}, \mathbf{x}') = \langle \psi(\mathbf{x}), \psi(\mathbf{x}') \rangle$, without any explicitly computation of $\psi(\mathbf{x})$ or $\psi(\mathbf{x}')$.

Proof.) On the basis of the representer theorem we can optimize Equation 11.1 with respect to the coefficients α instead of the coefficients $\mathbf{w}$ as follows.
Writing $\mathbf{w} = \sum_{j=1}^m \alpha_j \psi(\mathbf{x}_j)$, we have that, for all i:

$$\langle \mathbf{w}, \psi(\mathbf{x_i}) \rangle = \langle \sum_{\mathbf{j}} \alpha_{\mathbf{j}} \psi(\mathbf{x_j}), \psi(\mathbf{x_i}) \rangle = \sum_{j=1}^m \alpha_j \langle \psi(\mathbf{x_j}), \psi(\mathbf{x_i}) \rangle = \sum_j \alpha_j K(\mathbf{x}_j, \mathbf{x}_i)$$

And similarly:

$$\|\mathbf{w}\|^2 = \langle \sum_{\mathbf{j}} \alpha_{\mathbf{j}} \psi(\mathbf{x_j}), \sum_{\mathbf{i}} \alpha_{\mathbf{i}} \psi(\mathbf{x_i}) \rangle = \sum_{ij} \alpha_i \alpha_j \langle \psi(\mathbf{x_j}), \psi(\mathbf{x_i}) \rangle = \sum_{ij} \alpha_i \alpha_j K(\mathbf{x}_i, \mathbf{x}_j)$$

Let $K(\mathbf{x}, \mathbf{x}') = \langle \psi(\mathbf{x}), \psi(\mathbf{x}') \rangle$ be a function that implements the kernel function with respect to the embedding ψ. Instead of solving 11.1, we can solve the equivalent problem

$$\min_{\alpha \in \mathbb{R}^m} f\left(\sum_{j=1}^m \alpha_j K(\mathbf{x}_j, \mathbf{x}_1), \ldots, \sum_{j=1}^m K(\mathbf{x}_j, \mathbf{x}_m) \right) + R\left(\sqrt{\sum_{i,j=1}^m \alpha_i \alpha_j K(\mathbf{x}_j, \mathbf{x}_i)} \right)$$

To solve this new optimization problem we do not need any direct access to elements in the feature space. The only thing we should know is how to calculate inner products in the feature space, or equivalently, to calculate the kernel function. In fact, we need to know the value of the $m \times m$ matrix G, that is $G_{i,j} = K(\mathbf{x}_i, \mathbf{x}_j)$, which is often called the *Gram matrix*.
In particular, specifying the preceding to the Soft-SVM given in equation 10.4, we can rewrite the problem as

$$\min_{\alpha \in \mathbb{R}^m} \left(\lambda \alpha^T G \alpha + \frac{1}{m} \sum_{i=1}^m \max\{0, 1 - y_i (G\alpha)_i\} \right) \tag{11.2}$$

Once we learn the coefficients α we can calculate the prediction on a new instance by

$$\langle \mathbf{w}, \psi(\mathbf{x}) \rangle = \sum_{j=1}^m \alpha_j \langle \psi(\mathbf{x_j}), \psi(\mathbf{x}) \rangle = \sum_{j=1}^m \alpha_j K(\mathbf{x}_j, \mathbf{x})$$

The advantage of working with kernels rather than directly optimizing $\mathbf{w}$ in the feature space is that in some situations the dimension of the feature space is extremely large while implementing the kernel function is very simple. □

11.2.2 Polynomial Kernels

The k degree polynomial kernel is defined to be

$$K(\mathbf{x}, \mathbf{x}') = (1 + \langle \mathbf{x}, \mathbf{x}' \rangle)^k$$

We could show (but we will not), that this is indeed a kernel function, of the type $K(\psi(\mathbf{x}, \mathbf{x}'))$; and since the mapping $\psi : \mathbb{R}^n \to \mathbb{R}^{(n+1)^k}$ contains all the monomials up to degree k, an halfspace over the range of ψ corresponds to a normal predictor of degree k over the original halfspace. Hence, learning a halfspace with a k degree polynomial kernel enables us to learn polynomial predictors of degree k over the original space. Note that here the complexity of implementing it is $\mathcal{O}(n)$ while the dimension of the feature space is on the order of $\mathcal{O}\left(n^k\right)$.

11.2.3 Gaussian Kernel

Given a scalar $\sigma > 0$, the Gaussian Kernel, that is still a function of the type $K(\mathbf{x}, \mathbf{x}')$, is defined to be

$$K(\mathbf{x}, \mathbf{x}') = e^{-\frac{\|x - x'\|^2}{2\sigma}}$$

Here the feature space is of infinite dimension, but evaluating the kernel is still very simple. Intuitively, the Gaussian kernel sets the inner product in the feature space between $\mathbf{x}$ and $\mathbf{x}'$ 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, and σ is a parameter that controls the scale determining what we mean by "close". As we can see in figure 11.2, this standard deviation corresponds to the trade-off between precisely fit the training set (with risk of overfitting) or finding a less accurate but more general solution.

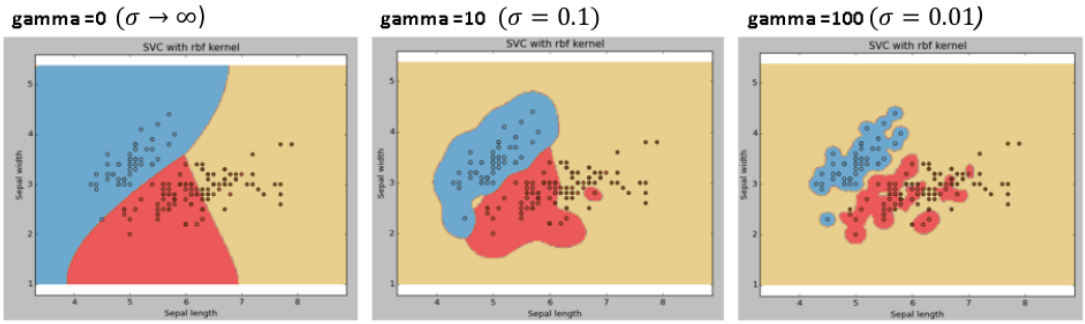


Figure 11.2: Graphical representation of the Gaussian Kernel method with different values of the trade-off parameter (gamma parameter of sklearn correspond to the inverse of σ).

Potentially, we can learn any polynomial predictor over the original space by using a Gaussian Kernel. Finally, recalling that the VC-dimension of the class of all polynomial predictors is infinite, we could think of some contradictions, but there isn't any, because the sample complexity required to learn with a Gaussian kernel depends on the margin in the feature space, which will be large if we are lucky, but can in general be arbitrarily small.
The Gaussian kernel is also called the *RBF* kernel, for **Radial Basis Functions**.
In figure 11.3 we can also see the advantages of using the RBF/Gaussian kernel respect to the linear one.

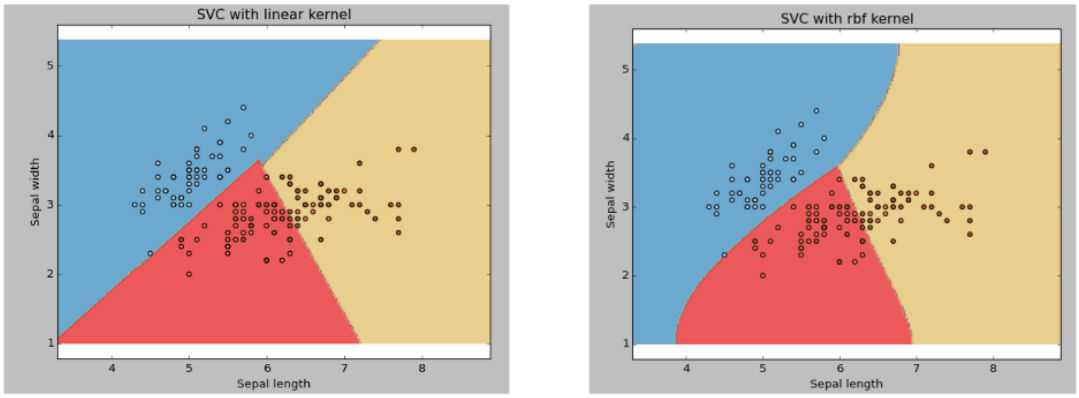


Figure 11.3: Linear vs RBF Kernel

Practical SVM with Kernels

With figure 11.2 and 11.3 we have already seen two example of practical application of the kernel method. We want also to focus on the trade-off between solutions with a large margin and some errors respect to solutions with a lower margin but less errors. The parameter that control such trade-off is λ, as we can easily see from the optimization problem

$$\min_{\mathbf{w}} \left(\|\mathbf{w}\|^2 + L_S^{hinge}(\mathbf{w}) \right)$$

The importance of this parameter, that has to be chosen properly, can be observed in figure 11.4.

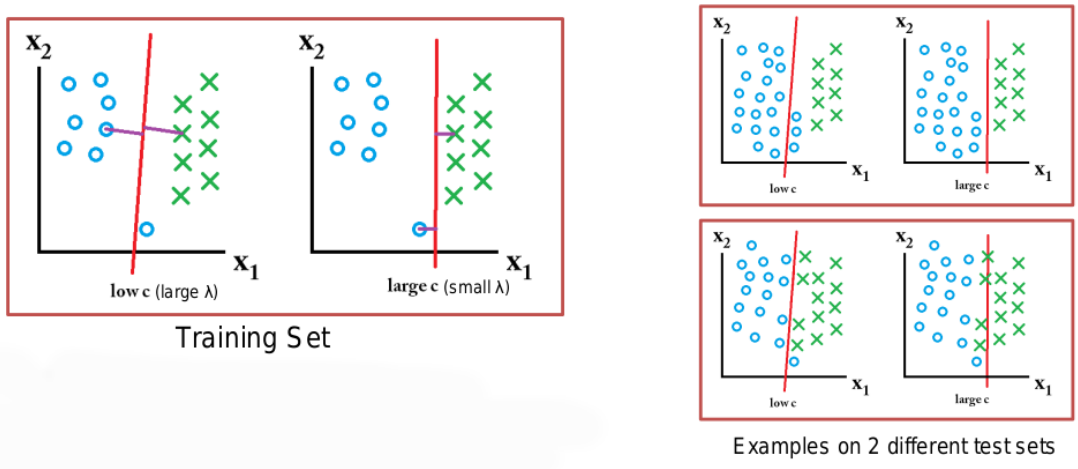


Figure 11.4: Trade-off parameter in SVM with Kernels (the parameter c in sklearn, libsvm and other ML tools has the same role but weights the loss term, i.e., works in the opposite direction)

So, we need to improve our choice of parameters λ and σ to obtain the best possible results.

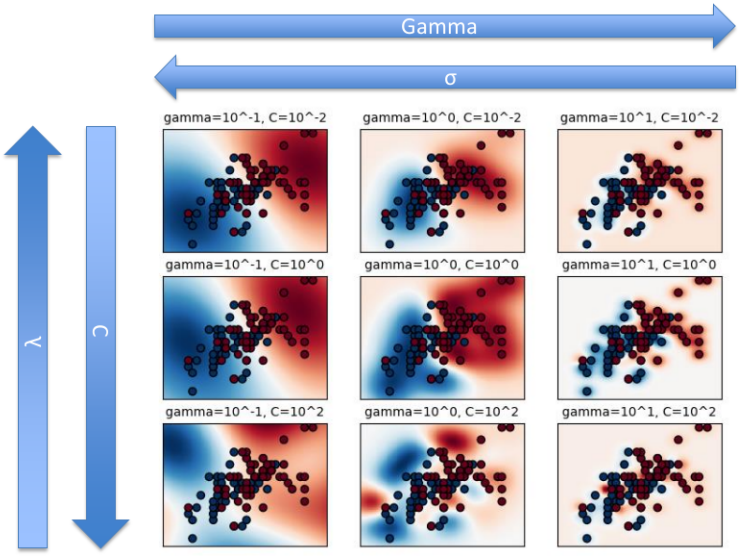


Figure 11.5: Grid Search

11.2.4 Implementing Soft-SVM with Kernels

While we could have designed an algorithm for solving equation 11.2, there is an even simpler approach that directly tackles the Soft-SVM optimization problem inn the feature space

$$\min_{\mathbf{w}} \left(\frac{\lambda}{2}\|\mathbf{w}\|^2 + \frac{1}{m}\sum_{i=1}^{m} \max\left\{0, 1 - y\langle \mathbf{w}, \psi(\mathbf{x_i})\rangle\right\} \right) \tag{11.3}$$

while only using kernel evaluations. So our main goal is to rewrite the algorithm described in box 10.4.1 to solve the previous equation. Formally, let $K(\mathbf{x}, \mathbf{x}') = \langle \psi(\mathbf{x}), \psi(\mathbf{x}')\rangle$ be the kernel function, and let's redefine the two vectors $\boldsymbol{\theta}(t)$ and $\mathbf{w}(t)$ that we used in the SGD procedure as $\boldsymbol{\beta}(t)$ and $\boldsymbol{\alpha}(t)$ such that:

$$\boldsymbol{\theta}(t) = \sum_{j=1}^{m} \beta_j^{(t)} \psi(\mathbf{x}_j) \qquad \mathbf{w}(t) = \sum_{j=1}^{m} \alpha_j^{(t)} \psi(\mathbf{x}_j)$$

The vectors $\boldsymbol{\beta}$ and $\boldsymbol{\alpha}$ are updated according to the following procedure (11.2.4).

SGD for solving Soft-SVM with Kernels

Goal: Solve equation 11.3
Initialize: $\boldsymbol{\beta}^{(1)} = \mathbf{0}$
Parameter: T
Algorithm:
for $t = 1, \ldots, T$
$\boldsymbol{\alpha}^{(t)} = \frac{1}{\lambda t}\boldsymbol{\beta}^{(t)}$
choose i randomly from $\{1, \ldots, m\}$
for all $j \neq i$ set $\beta_j^{(t+1)} = \beta_j^{(t)}$
if $y_i \sum_{j=1}^{m} \alpha_j^{(t)} K(\mathbf{x}_j, \mathbf{x}_i) < 1$ then $\boldsymbol{\beta}^{(t+1)} = \beta_i^{(t)} + y_i$
else: $\beta_i^{(t+1)} = \beta_i^{(t)}$

Output:
$\bar{\mathbf{w}} = \sum_{j=1}^{m} \bar{\alpha}_j \psi(\mathbf{x}_j)$ where $\bar{\boldsymbol{\alpha}} = \frac{1}{T}\sum_{t=1}^{T} \boldsymbol{\alpha}^{(t)}$

Chapter 12

Neural Networks

An artificial neural network is a model of computation inspired by the structure of neural networks (NN) in the brain. In simplified models of the brain, it consists of a large number of basic computing devices (neurons) that are connected to each other in a complex communication network, through which the brain is able to carry out highly complex computations. Artificial neural networks are formal computation constructs that are modeled after this computation paradigm. A neural network can be described as a directed graph whose nodes correspond to neurons and edges correspond to links between them. Each neuron receives as input a weighted sum of the outputs of the neurons connected to its incoming edges. The first proposal of a neural network was in 1940-50 but only in the 80-90s these ideas were applied to the practical, but with results lower than SVM and other techniques: we had to wait until 2010 to have performing algorithms.

12.1 FeedForward Neural Network

The idea behind neural networks is that many neurons can be joined together by communication links to carry out complex computations. It is common to describe the structure of a neural network as a graph whose nodes are the neurons and each (directed) edge in the graph links the output of some neuron to the input of another neuron. We will restrict our attention to feedforward network structures in which the underlying graph does not contain cycles. The network is typically organized into *layers*: each neuron takes in input only the output of neurons of the previous layer. A feedforward neural network is described by a directed acyclic graph, $G = (V, E)$, and a weight function over the edges, $w : E \to \mathbb{R}$ (where V are the neurons, $|V|$ the size of the network and E indicates the connections between neurons, by directed edges).

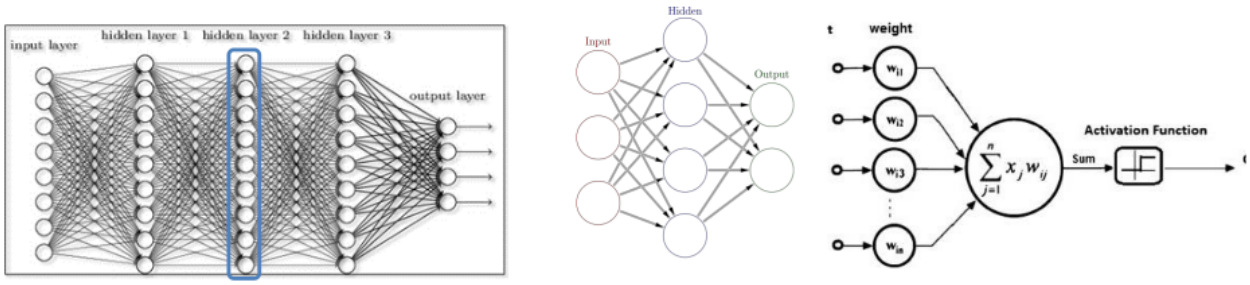


Figure 12.1: Graph of a Feedforward network

So, each neuron:

- Takes in input the sum of the outputs of **all** the connected neurons from previous layer weighted by the edge weights.
- Each single neuron is modeled as a simple scalar **activation function** $\sigma : \mathbb{R} \to \mathbb{R}$.

To simplify the description of the calculation performed by the network, we have assumed that the network is organized in layers. That is, the set of nodes can be decomposed into a union of (nonempty) disjoint subsets, $V = \bigcup_{t=0}^{T} V_t$ such that each edge in E connects some node in V_{t-1} to some node in V_t , for some $t \in [T]$.
Notation:

- V_t, $t = 0, \ldots, T$: t-th layer.
- $d^t + 1$: number of nodes of layer t.
  - " $+1$": constant neuron (avoid bias, as in homogeneous coordinates).
- V_0: input layer.
- V_T: output layer.
- Layers $V_1, \ldots, V_{T-1}$ are often called hidden layers.
- T: depth of the network ($T = 2$ in "classic" NN, $T >> 2$ in deep networks).
- $v_{t,i}$: i-th neuron in the t-th layer.
- $\mathbf{v}^{(t)} = (1, v_{t,1}, \ldots, v_{t,d^t})^T$: all neurons of layer t.
- $w_{rj}^{(t+1)} = w\left(v_{t,r}, v_{t+1,j}\right)$: weight of arc from neuron r of layer t to neuron j of layer $t+1$.
- $\mathbf{w}_j^{(t)} = \left(w_{0j}^{(t)}, \ldots, w_{d^t j}^{(t)}\right)^T$: all weights of arcs in input to neuron j of layer t.
- $w^{(t)}$: matrix of weights of all arcs incoming to layer t.

$$w^{(t)} = \begin{bmatrix} w_{01}^{(t)} & w_{02}^{(t)} & \ldots & w_{0d^{(t)}}^{(t)} \\ w_{11}^{(t)} & w_{12}^{(t)} & \ldots & w_{1d^{(t)}}^{(t)} \\ \vdots & \vdots & \ldots & \vdots \\ w_{d^{(t-1)}1}^{(t)} & w_{d^{(t-1)}2}^{(t)} & \ldots & w_{d^{(t-1)}d^{(t)}}^{(t)} \end{bmatrix}$$

We denote $o_{t,i}$ the output of $v_{t,i}$ when the network is fed with the input vector $\mathbf{x}$. Therefore, for $i \in [d^t]$ we have $o_{0,i}(\mathbf{x}) = x_i$, and for $i = n+1$ we have o $o_{0,i}(\mathbf{x}) = 1$ (because the last layer, V_0 always output '1').
So, how does a network work? Let's assume we have calculated the outputs of the neurons at layer t, we want to calculate them at layer $t+1$. We proceed as follow: the output of a neuron is a non-linear (activation) function applied to the linear combination of the inputs coming from the previous layer. When the network is fed with input $\mathbf{x}$:

$$a_{t+1,j} = \langle \mathbf{w_j^{(t+1)}}, \mathbf{v^{(t)}} \rangle = \text{output of neuron before the activation function}$$

$$o_{t+1,j}(\mathbf{x}) = \sigma\left(\sum_{r:(v_{t,r},v_{t+1,j}\in E)} w\left(v_{t,r}, v_{t+1,r}\right) o_{t,r}(\mathbf{x})\right) = \sigma\left(a_{t+1,j}(\mathbf{x})\right)$$

In vector notation: $$o_{t+1,j} = \sigma\left(\left\langle \mathbf{w_j^{(t+1)}}, \mathbf{v^{(t)}} \right\rangle\right) = \sigma\left(a_{t+1,j}\right)$$

Forward Propagation

So, practically, if we want to take an input sample and compute the output of the network we have to pass trough the outputs of each layer, using the previous one as a new input ($v^{(0)} \to v^{(1)} \to v^{(2)} \to \cdots \to v^{(T)}$). We can summarize this with the series of instructions in box 12.1.

Forward Propagation of Neural Networks

Input: $x = (x_1, \ldots, x_d)^T$; NN with 1 output node

Algorithm:
$v^{(0)} \leftarrow (1, x_1, \ldots, x_d)^T$;
for $t \leftarrow 1$ to T do:
  $a^{(t)} \leftarrow \left(w^{(t)}\right)^T v^{(t-1)}$;
  $v^{(t)} \leftarrow \left(1, \sigma(a^{(t)})^T\right)^T$;
  $y \leftarrow v^{(T)}$;
return y;

Output: prediction y of NN

12.1.1 Activation Functions

A feedforward neural network is described by a directed acyclic graph, $G = (V, E)$, and a weight function over the edges, $w : E \to \mathbb{R}$. Each single neuron is modeled as a simple scalar function, $\sigma : \mathbb{R} \to \mathbb{R}$. We will focus on some possibilities for σ, like the sign function, the threshold function or the sigmoid. We call σ the *activation function* of the neurons.

Sign and Threshold Functions

$$\sigma(a) = sign(a) \qquad \mathbb{R}^n \to [-1, 1]$$

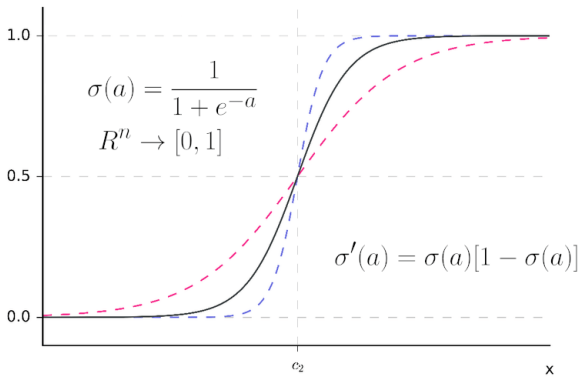
$$f(x) = \begin{cases} 0 & \text{if } 0 > x \\ 1 & \text{if } x \geq 0 \end{cases}$$

- **Advantages:**
  - Simple and fast
  - Nice interpretation as the firing rate of a neuron
    - -1 = not firing

    - 1 = firing

- **Disadvantages:**
  - Output is not smooth/continuous
  - Saturate and kill gradients, thus NN will barely learn

Sigmoid Function



- **Advantages:**
  - Smooth Output
  - Nice interpretation as the firing rate of a neuron
    - 0 = not firing at all
    - 1 = fully firing

- **Disadvantages:**
  - Sigmoid neurons saturate and kill gradients, thus NN will barely learn: when the neuron's activation are 0 or 1 (saturate)
    - gradient at these regions is almost zero
    - almost no signal will flow to its weights
    - if if initial weights are too large then most neurons would saturate

Tanh and ReLU Function

Finally we have these two particular functions, that can be observed in figure 12.2: For what regard *tanh* function (on the left,) it is just a scaled and shifted version of the sigmoid function (in fact $tanh(x) = 2sigm(2x) - 1$), and for this reason its use has similar advantages and disadvantages than the sigmoid's one (like the saturation problem). The main difference is that *tanh* function's output is zero-centered.
ReLU function, instead, it is more interesting:

- Trains much faster:

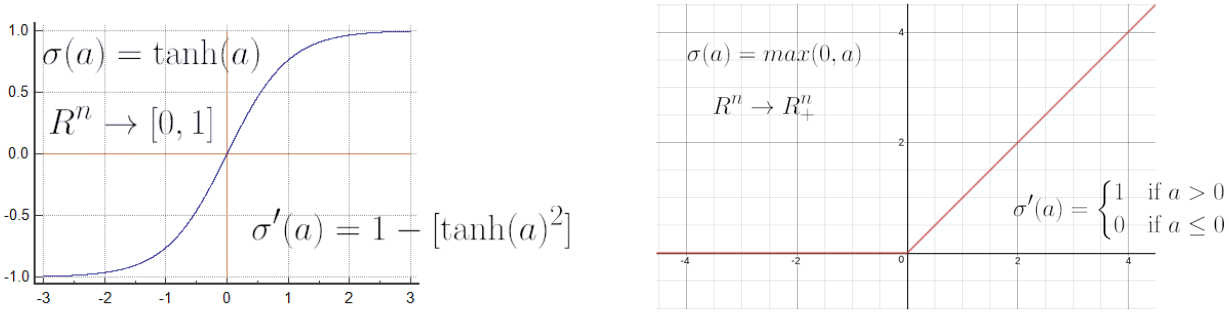


Figure 12.2: Tanh (on the left) and ReLU (on the right) functions

  - o accelerates the convergence of SGD
  - o due to linear, non-saturating form
- Less expensive operations:
  - o compared to sigmoid/tanh (exponentials etc.)
  - o implemented by simply thresholding a matrix at zero
- More expressive
- Prevents the gradient vanishing problem
- Most deep networks use ReLU nowadays

12.1.2 Learning Neural Networks

Once we have specified a neural network by (V, E, σ, w), we obtain a function $h_{V,E,\sigma,w} : \mathbb{R}^{|V_0-1|} \to \mathbb{R}^{|V_T|}$. Any set of such functions can serve as an hypothesis class for learning. Usually, we define a hypothesis class of neural network predictors by fixing the graph (V, E) as well as the activation function σ, and letting the hypothesis class be all functions of the form $h_{V,E,\sigma,w}$ for some $w : E \to \mathbb{R}$. The triplet (V, E, σ) is often called the **architecture** of the network. We denote the hypothesis class by

$$\mathcal{H}_{V,E,\sigma} = \{h_{V,E,\sigma,w} : w \text{ is mapping from } E \text{ to } \mathbb{R}\}$$

Where w contains the parameters that are going to be learned, so the scope of the training of a NN is finding the optimal set of weights.

12.2 The Expressive Power of Neural Networks

In this section we study the expressive power of neural networks, namely, what type of functions can be implemented using a neural network. More concretely, we will fix some architecture, V, E, σ, and will study what functions hypotheses in $mclH_{V,E,\sigma}$ can implement, as a function of the size of V.

We start the discussion with studying which type of Boolean functions (i.e., functions from $\{\pm 1\}^d$ to $\{\pm 1\}$) can be implemented by $\mathcal{H}_{V,E,sign}$. Then observe that for every computer in which real numbers are stored using b bits, whenever we calculate a function $f : \mathbb{R}^n \to \mathbb{R}$ on such a computer we in fact calculate a function $g : \{\pm 1\}^n b \to \{\pm 1\}^b$. Therefore, studying which Boolean functions can be implemented by $\mathcal{H}_{V,E,sign}$ can tell us which functions can be implemented on a computer that stores real numbers using b bits.

Proposition 12.1. *For every d, there exists a graph (V, E) of depth 2 such that $\mathcal{H}_{V,E,sign}$ contains all functions from $\{-1, 1\}^d$ to $\{-1, 1\}$.*

Proof. We construct a graph with $|V_0| = n + 1$, $|V_1| = 2^n + 1$. Let E be all possible edges between adjacent layers. Now let $f : \{\pm 1\}^n \to \{\pm 1\}$ be some boolean function. We need to show that we can adjust the weights so that the network will implement f. Let $\mathbf{u_1}, \ldots, uu_k$ be all vectors in $\{\pm 1\}^n$ on which f outputs 1. Observe that for every i and every $\mathbf{x} \in \{\pm 1\}^n$, if $\mathbf{x} \neq \mathbf{u}_i$ then $\langle \mathbf{x}, \mathbf{u_i} \rangle \leq n - 2$ and if $\mathbf{x} = \mathbf{u}_i$ then $\langle \mathbf{x}, \mathbf{u_i} \rangle = n$. It follows that the function $g_i(\mathbf{x}) = sign\left(\langle \mathbf{x}, \mathbf{u_i} \rangle - n + 1\right)$ is equal to 1 if and only if $\mathbf{x} = \mathbf{u}_i$. This means that we can adapt the weights between V_0 and V_1 so that for every $i \in [k]$, the neuron $v_{1,i}$ implements the function $g_i(\mathbf{x})$. Next, we observe that $f(\mathbf{x})$ is the disjunction of the functions $g_i(\mathbf{x})$, and therefore can be written as

$$f(\mathbf{x}) = sign\left(\sum_{i=1}^{k} g_i(\mathbf{x}) + k - 1\right)$$

which concludes our proof. □

This proposition means that, without restricting the size of the network, every Boolean function can be implemented using a neural network of depth 2. However, this is a weak property, as the size of the resulting network might be exponentially large, because of the number of the necessary hidden layers (previous proof).

Proposition 12.2. *For every d, let $s(d)$ be the minimal integer such that there exists a graph (V, E) with $|V| = s(d)$ such that $\mathcal{H}_{V,E,sign}$ contains all the functions from $\{-1, 1\}^d$ to $\{-1, 1\}$. Then $s(d)$ is ab exponential function of d.*

We obtain analogs results using the sigmoid as the activation function.
Next, we can formulate the same proposition also for real functions:

Proposition 12.3. *For every fixed $\varepsilon > 0$ and every Lipschitz function $f[-1, 1]^d \to [-1, 1]$ is is possible to construct a neural network such that for every input $\mathbf{x} \in [-1, 1]^d$ the output of the NN is in $[f(\mathbf{x} - \varepsilon), f(\mathbf{x}) + \varepsilon]$.*

So NN are universal approximators! However, as in the case of Boolean functions, the size of the network here again cannot be polynomial in d.

Proposition 12.4. *Fix some $\varepsilon \in (0, 1)$. For every d, let $s(d)$ be the minimal integer such that there exists a graph (V, E) with $|V| = s(d)$ such that $\mathcal{H}_{V,E,\sigma}$, with $\sigma = sigmoid$, can approximate, with precision ε, every 1-Lipschitz function $f[-1, 1]^d \to [-1, 1]$. Then s(d) is exponential in d.*

Example 12.1. *We next provide several geometric illustrations of functions $f : \mathbb{R}^2 \to \{\pm 1\}$ and show how to express them using a neural network with the sign activation function.*
Let us start with a depth 2 network, namely, a network with a single hidden layer. Each neuron in the hidden layer implements an halfspace predictor. Then, the single neuron at the output

layer applies an halfspace on top of the binary outputs of the neurons in the hidden layer. So the network contains all hypothesis which are an intersection of $k-1$ halfspaces, where k is the number of neurons in the hidden layer; namely, they can express all convex polytopes with $k-1$ faces, as can be seen in picture 12.3 on the left, as an intersection of 5 halfspaces. Adding, instead, one more layer, we can implement a function that indicates whether $\mathbf{x}$ is in some convex polytope, since the neurons in the output layer implement the disjunction of its inputs, computing an union of polytopes (figure 12.3, on the right).

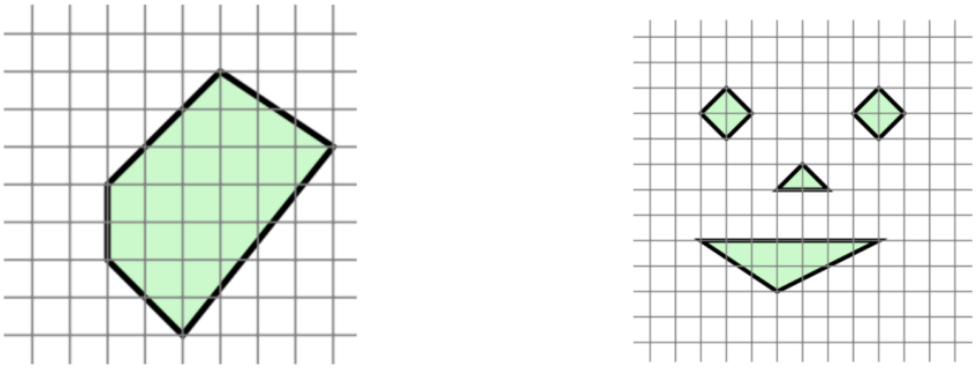


Figure 12.3: Geometric example of the implementation of 2-layer (left) and 3-layer (right) Neural Network in which each neuron corresponds to an halfspace

12.3 Sample Complexity and Runtime of a Neural Network

Now, we will discuss about the sample complexity of learning the class $\mathcal{H}_{V,E,\sigma}$. Recall that the fundamental theorem of learning tells us that the sample complexity of learning an hypothesis class of binary classifiers depends on its VC-dimension. Therefore, we focus on calculating the VC-dimension of hypothesis classes of the form $\mathcal{H}_{V,E,\sigma}$, where the output layer of the graph contains a single neuron.

Proposition 12.5. *The VC-dimension of $\mathcal{H}_{V,E,sign}$ is $\mathcal{O}(|E|\log(|E|))$.*

Proposition 12.6. *The VC-dimension of $\mathcal{H}_{V,E,sign}$ is $\mathcal{O}\big(|V|^2|E|^2\big)$.*

So, large Neural Networks require a lot of data!
But, assuming we have enough data, can we find the best hypothesis? The problem is that applying the ERM rule to NN is computationally difficult, even for relatively small NN.

Theorem 12.1. *Let $k \geq 3$. For every d let (V,E) be a layered graph with d input nodes at the (only) hidden layer, where one of them is the constant neuron, and a single output node. Then, it is **NP-hard** to implement the ERM rule with respect to $\mathcal{H}_{V,E,sign}$.*

Also approximation of ERM rule (for example predictors with low empirical error) are computationally infeasible. One could think of changing the architecture of the network to circumvent the hardness result, for example applying the ERM rule on other, larger networks, or also changing the activation function (with the sigmoid for example). But all this approaches are gonna fail.

12.4 SGD and BackPropagation

In the previous section we have shown that applying the ERM rule on a NN is computationally very hard. So the successive idea is:

- Forward propagate the training data and compute the loss
- Consider the loss as a function of the weights and compute the gradient of the loss w.r.t the weights
- Update the weights with SGD

Since E is a finite set, we can think of the weight function as a vector $\mathbf{w} \in \mathbb{R}^{|E|}$. Suppose the network has d input neurons and k output neurons, and denote by $h_{\mathbf{w}} : \mathbb{R}^d \to \mathbb{R}^k$ the function calculated by the network is the weight function is defined by $\mathbf{w}$. Recall that the SGD algorithm acts for minimizing the risk function

$$L_{\mathcal{D}}(\mathbf{w}) = \mathop{\mathbb{E}}_{(\mathbf{x},\mathbf{y})\sim\mathcal{D}} [\ell(h_{\mathbf{w}}(\mathbf{x}), \mathbf{y})]$$

so we can use the same code written in 9.2.2 with a few modifications, which are relevant to the neural network application because of the non-convexity of the objective function.
First, while in the general implementation we initialized $\mathbf{w}$ to the zero vector, in this case it has to be set to a randomly chosen vector with all values close to zero. This is because an initialization with the zero vector will lead all hidden neurons to have the same weights (if the network is a full layered network). In addition, the hope is that if we repeat the SGD procedure several times, where each time we initialize the process with a new random vector, one of the runs will lead to a good local minimum. Second, while a fixed step size, η, is guaranteed to be good enough for convex problems, here we utilize a variable step size η_t because of the non-convexity of the loss function. Third, we output the best performing vector on a validation set; in addition, it is sometimes helpful to add a regularization on the weights, with parameter λ (so we try to minimize $L_{\mathcal{D}(\mathbf{w})} + \frac{\lambda}{2}\|\mathbf{w}\|^2$). Finally, the gradient does not have a closed form solution. Instead, it is implemented using the backpropagation algorithm, which will be described next.

SGD for Neural Networks

Parameters:
Number of iterations τ
Step size sequence $\eta_1, \eta_2, \dots \eta_\tau$
Regularization parameter $\lambda > 0$

Input:
Network: layered graph $G = (V, E)$
Differentiable activation function $\sigma : \mathbb{R} \to \mathbb{R}$

Algorithm:
Chose $\mathbf{w}^{[1]} \in \mathbb{R}^{|E|}$ at random
(from a distribution s.t. $\mathbf{w}^{[1]}$ is close enough to $\mathbf{0}$)
for $s = 1, 2, \dots, \tau$
sample $(\mathbf{x}, \mathbf{y}) \sim \mathcal{D}$
calculate gradient $\mathbf{v}_s = backpropagation(\mathbf{x}, \mathbf{y}, \mathbf{w}, (E, V), \sigma)$
update $\mathbf{w}^{[s+1]} = \mathbf{w}^{[s]} - \eta_s \left(\mathbf{v}_s + \lambda \mathbf{w}^{[s]}\right)$

Output:
$\bar{\mathbf{w}}$ is the best performing $\mathbf{w}^{[s]}$ on a validation set.

Now we would like to focus on how the algorithm compute the gradient of the vector $\mathbf{w}$, and so on the function that we called **backpropagation**. The standard update rule, provided by SGD algorithm would be:

$$w_{ij}^{(t)[s+1]} = w_{ij}^{(t)[s]} - \eta_s \frac{\partial L_S}{\partial w_{ij}^{(t)[s']}}$$

where t indicates the layer, s is the iteration index of the algorithm, η_s is the learning rate and last partial derivative is nothing but the gradient of the loss function with respect to each single weight, that is

$$\frac{\partial L_S}{\partial w} = \frac{\partial}{\partial w}\left(\frac{1}{m}\sum_{i=1}^{m} \ell(h, (\mathbf{x}_i, y_i))\right) = \frac{1}{m}\sum_{i=1}^{m} \frac{\partial L(h, (\mathbf{x}_i, y_i))}{\partial w}$$

and that has to be determined. But, as we know, we can compute the loss function only on the output, i.e. only after the last layer (but recall that each neuron contains also the non-linear activation function).
To describe the backpropagation we start defining the change in error with respect to the weighted average before the non-linear transformation as

$$\delta^{(t)} = \frac{\partial L}{\partial \mathbf{a}^{(\mathbf{t})}} = \begin{bmatrix} \dfrac{\partial L}{\partial a_{t,1}} \\ \vdots \\ \dfrac{\partial L}{\partial a_{t,d^t}} \end{bmatrix}$$

where $a_{t,j}$ is the output of the j-th neuron before the application of the activation function. So we have to decompose the gradient with the chain rule, remembering that each $w_{ij}^{(t)}$ impacts only

on $a_{t,j}$ and that σ' depends on the selected activation function:

$$\frac{\partial L}{\partial w_{ij}^{(t)}} = \frac{\partial L}{\partial a_{t,j}} \frac{\partial a_{t,j}}{\partial w_{ij}^{(t)}} = \delta_j^{(t)} \left(\sum_{k=0}^{d^{(t+1)}} w_{kj}^{(t)} v_{t-1,k} \right) = \delta_j^{(t)} v_{t-1,i}$$

$$\delta_j^{(t)} = \frac{\partial L}{\partial v_{t,j}} \frac{\partial v_{t,j}}{\partial a_{t,j}} = \frac{\partial L}{\partial v_{t,j}} \sigma'(a_{t,j})$$

But how dows effectively the loss change w.r.t. $v_{t,j}$?

- Changes in layer t affects only neurons in layer $t+1$ (and then each following layer up tp the loss at the end).
- Each neuron can affect all the neurons in the next layer.
- We need to sum contributions to all the neurons in layer $t+1$.
- We need also $\delta^{(t+1)}$ to compute the $\delta^{(t)}$ (i.e. the solution of the next layer).

The last point of the list is the reason of the name of the method.

$$\frac{\partial L}{\partial v_{t,j}} = \sum_{k=1}^{d^{(t+1)}} \frac{\partial L}{\partial a_{t+1,k}} \frac{\partial a_{t+1,k}}{\partial v_{t,j}} = \sum_{k=1}^{d^{(t+1)}} w_{jk}^{(t+1)} \delta_k^{(t+1)}$$

$$\implies \qquad \delta_j^{(t)} = \sigma'(a_{t,j}) \sum_{k=1}^{d^{(t+1)}} w_{jk}^{(t+1)} \delta_k^{(t+1)}$$

So each layer needs the solution of the following one: at this point the strategy consist into starting from the last layer, since $\delta^{(L)}$ can be computed from the loss on the output, and than backpropagating the gradients through all the layers up to the first.

$$v^{(0)} \leftarrow v^{(1)} \leftarrow v^{(2)} \leftarrow \cdots \leftarrow v^{(T)}$$

We can summarize the backpropagation in the box 12.4.

BackPropagation

Input:
Data point $(\mathbf{x}_i, y_i)$
Neural Network (with weights $w_{ij}^{(t)}$, for $1 \leq t \leq T$)

Output:
$\delta^{(t)}$ for $t = 1, \ldots, T$
Compute $a^{(t)}$ and $v^{(t)}$ for $t = 1, \ldots, T$;
$\delta^{(L)} \leftarrow \frac{\partial L}{\partial a^{(L)}}$;
for $t = T-1$ downto 1 do
    $\delta_j^{(l)} \leftarrow \sigma'(a_{t,j}) \cdot \sum_{k=1}^{d^{(l+1)}} w_{jk}^{(t+1)} \delta_k^{(t+1)}$ for all $j = 1, \ldots, d^{(t)}$
return $\delta^{(1)}, \ldots, \delta^{(T)}$;

Now we can finally write a recap of the algorithm in the following box (12.4).

BackPropagation Algorithm with SGD for training Neural Networks

Input:
Training data $(x_1, y_1), \ldots, (x_m, y_m)$

Algorithm:
Initialize $w_{ij}^{(t)} \ \forall \ i, j, t$
for $s \leftarrow 0, 1, 2, \ldots$ do: // Until convergence
pick (x_k, y_k) at random from training data; // SGD
compute $v_{t,j} \quad \forall \ j, t$ // Forward Propagation
compute $\delta_j^{(t)} \quad \forall \ j, t;$ // Backward Propagation
$w_{ij}^{(t)[s+1]} = w_{ij}^{(t)[s]} - \eta v_{t-1,i} \delta_j^{(t)} \quad \forall \ i, j, t$ // Update weights

Output:
NN weights $w_{ij}^{(t)}$

NN training's tips and tricks:

- **Preprocessing**:
  - Typically all inputs are normalized and centered.
  - Both local or global normalization strategies can be used.
- **Initialization of the weights**:
  - All to 0 does not work.
  - Random values around 0 (regime where model is roughly linear).
  - Uniform or normal distribution can be used.
  - Multiple initialization and training, then select the best result (with the smallest training error).
  - In deep NN *Glorot* initialization is often used: normal distribution with variance inversely proportional to the sum of the number of incoming and outcoming connections of the neurons.
- **When to stop?**
  - Small training error.
  - Small marginal improvement in error.
  - Upper bound on number of iterations.
- **Loss function usually has multiple local minima**:
  - With highly dimensional spaces the risk is smaller than in low dimensional ones.
  - Run stochastic gradient descent for different (random) initial weights.

- **Regularization**:
  - Minimize weighted sum of the loss with the sum of all the weights.
  - Avoid too large weights and make optimization more stable.
  - $L1$ or $L2$ regularization can be used:

$$L1 = L_S(h) + \frac{\lambda}{m} \sum_{ijt} \left| w_{ij}^{(t)} \right| \qquad L2 = L_S(h) + \frac{\lambda}{m} \sum_{ijt} \left(w_{ij}^{(t)} \right)^2$$

Chapter 13

Convolutional Neural Networks

There are many issues in the *fully connected* feedforward NN moedl we have seen up to now:

- Each neuron of a generic layer $t-1$ is connected with *each* neuron of layer t: this means that there is a very huge number of edges/weights to deal with.
- The domain structure is not taken into account: the model, in fact, doesn't consider that a neuron can be "closer", in the sense of "more correlated", to some, end less to others. Some domains have a structure, for example the pixels in an image (a single pixel is more related to the closer than to the far apart ones) or the letters in a text (a letter in a text is more related to letters of the same word than to the ones 10 pages ahead).
  Interesting features are often local, shift-invariant and deformation-invariant (example in figure 13.1):
  - Some patterns are much smaller than the whole image.
  - The same patterns can appear in different places.
  - Similar detectors in different regions share similar parameter

Figure 13.1: Example of patterns in an image

In deep learning, a *convolutional neural network* is a class of deep neural networks, most commonly applied to analyzing visual imagery. They are also known as shift invariant or space invariant artificial neural networks (SIANN), based on their shared-weights architecture and translation invariance characteristics. They have applications in image and video recognition, recommender systems, image classification, medical image analysis, and natural language processing.
It is a Deep Learning algorithm which can take in an input image, assign importance (learnable weights and biases) to various aspects/objects in the image and be able to differentiate one from

the other. The pre-processing required in a ConvNet is much lower as compared to other classification algorithms. While in primitive methods filters are hand-engineered, with enough training, ConvNets have the ability to learn these filters/characteristics. The architecture of a ConvNet is analogous to that of the connectivity pattern of Neurons in the Human Brain and was inspired by the organization of the Visual Cortex. Individual neurons respond to stimuli only in a restricted region of the visual field known as the Receptive Field. A collection of such fields overlap to cover the entire visual area.
The main features of this new model of convolutional Neural Network are:

- Local connectivity: receptive field for each neuron.
- Shared ("tied") weights: spatially invariant response.
- Multiple feature maps.
- Subsampling (pooling).

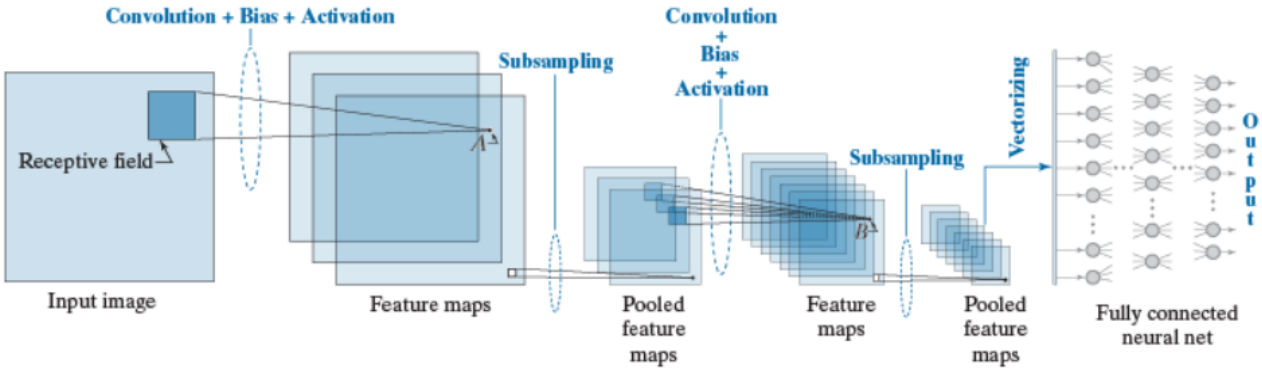


Figure 13.2: So, what about training some "small" detectors and let each detector "move around"?

Local connectivity and shared ("tied") weights
As you can see in figure 13.3, on the left, each orange unit is *only* connected to **neighboring** blue units. Also, all the orange units share the same weights **w**, such that each one computes the same function, but with a different input window.
Multiple feature map Also, if you look at figure 13.3 (this time on the right) you can see that all orange units compute the same function, but with different input windows, differently from the one computed by red units.

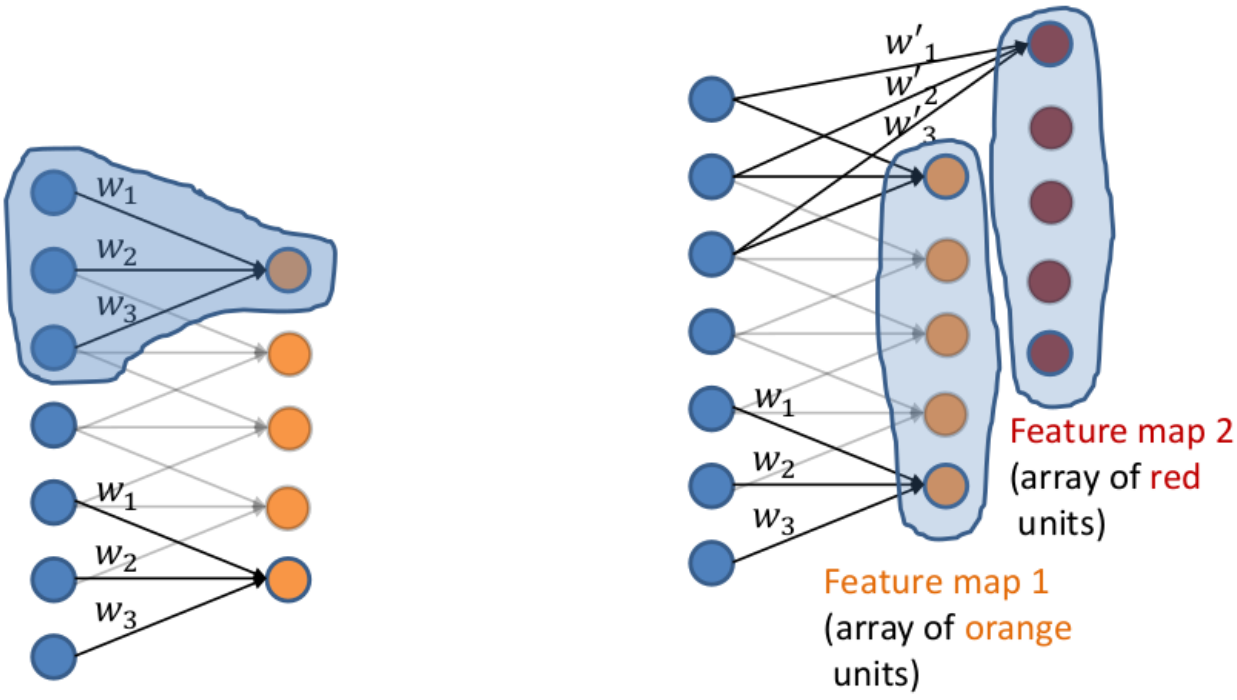


Figure 13.3: Convolutional NN features

13.1 Application of a CNN to an image (just an deepening)

In figure 13.4, we have an RGB image which has been separated by its three color lanes (red, green and blue). You can imagine how computationally intensive things would get once the images reach dimensions, say 8K. The role of the CNN is to reduce the images into a form which is easier to process, without losing features which are critical for getting a good prediction. This is important when we are to design an architecture which is not only good at learning features but also is scalable to massive datasets.

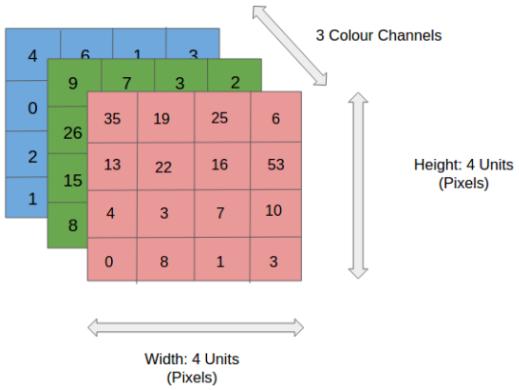


Figure 13.4: Convolutional NN features

An image is nothing but a matrix of pixel values, right? So why not just flatten the image (e.g.

3x3 image matrix into a 9x1 vector) and feed it to a Multi-Level Perceptron for classification purposes? Uh.. not really. In cases of extremely basic binary images, the method might show an average precision score while performing prediction of classes but would have little to no accuracy when it comes to complex images having pixel dependencies throughout. A Convolutional NN is able to successfully capture the spatial and temporal dependencies in an image through the application of relevant filters. The architecture performs a better fitting to the image dataset due to the reduction in the number of parameters involved and reusability of weights. In other words, the network can be trained to understand the sophistication of the image better.
Back to our figure, we can start considering a generic (simpler) example: we have an image of dimension 5 (Height) x 5(Breadth) x 1(Number of channels), to which we apply a *kernel/filter*, we simply choose a 3 x 3 x 1 matrix, to "extract" the convolutional layer:

$$\begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix}$$

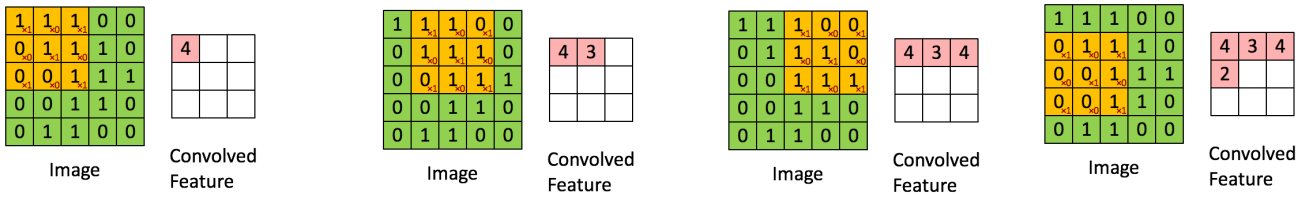


Figure 13.5: Calculation of the convolutional layer

The filter moves to the right with a certain *Stride Value* (in this case 1) till it parses the complete width. Moving on, it hops down to the beginning (left) of the image with the same stride value and repeats the process until the entire image is traversed.
In the case of images with multiple channels (e.g. RGB), the Kernel has the same depth as that of the input image. Matrix multiplication is performed between over all the "layers" of the image, and all the results are summed with the bias to give us a squashed one-depth channel Convoluted Feature Output, as we can see in figure 13.6.
The objective of the Convolution Operation is to extract the high-level features such as edges, from the input image. Convolutional NN need not be limited to only one Convolutional Layer. Conventionally, the first one is responsible for capturing the low-level features such as edges, color, gradient orientation, etc. With added layers, the architecture adapts to the high-level features as well, giving us a network which has the wholesome understanding of images in the dataset, similar to how we would.

Pooling Layer

Similar to the Convolutional Layer, the Pooling layer is responsible for reducing the spatial size of the convolved feature. This is to decrease the computational power required to process the data through dimensionality reduction. Furthermore, it is useful for extracting dominant features which are rotational and positional invariant, thus maintaining the process of effectively training of the model. There are two types of Pooling: Max Pooling and Average Pooling. Max Pooling returns the maximum value from the portion of the image covered by the Kernel. On the other

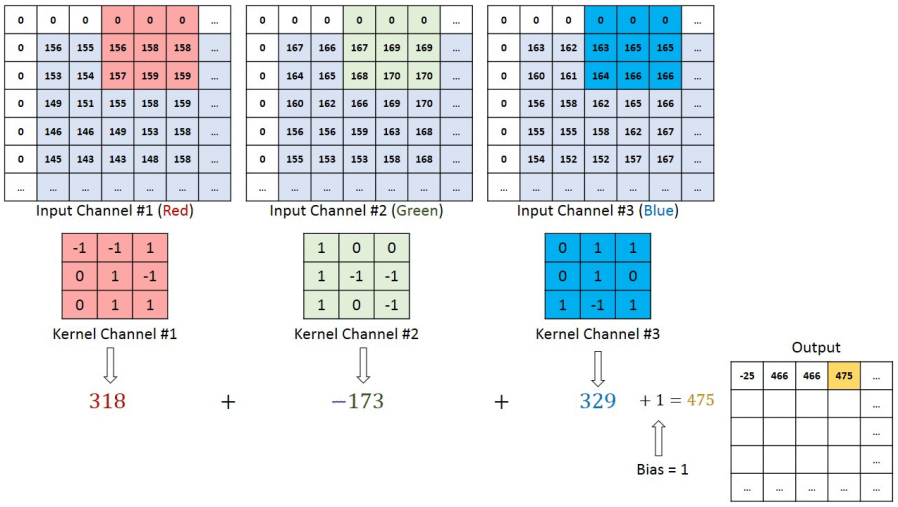


Figure 13.6: Calculation of the convolutional layer for a multiple channel image

hand, Average Pooling returns the average of all the values from the portion of the image covered by the Kernel. Max Pooling also performs as a Noise Suppressant. It discards the noisy activations altogether and also performs de-noising along with dimensionality reduction. On the other hand, Average Pooling simply performs dimensionality reduction as a noise suppressing mechanism. Hence, we can say that Max Pooling performs a lot better than Average Pooling. The Convolutional Layer and the Pooling Layer, together form the i-th layer of a Convolutional Neural Network. Depending on the complexities in the images, the number of such layers may be increased for capturing low-levels details even further, but at the cost of more computational power.

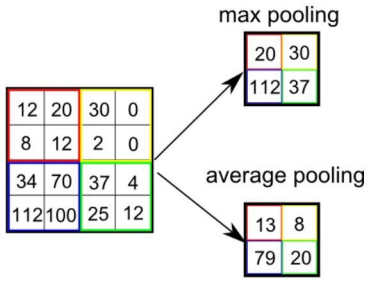


Figure 13.7: Types of pooling

Classification — Fully Connected Layer

Adding a Fully-Connected layer is a (usually) cheap way of learning non-linear combinations of the high-level features as represented by the output of the convolutional layer. The Fully-Connected layer is learning a possibly non-linear function in that space. Now that we have converted our input image into a suitable form for our Multi-Level Perceptron, we shall flatten the image into a column vector. The flattened output is fed to a feed-forward neural network and backpropagation applied to every iteration of training. Over a series of epochs, the model is able to distinguish between dominating and certain low-level features in images and classify them.

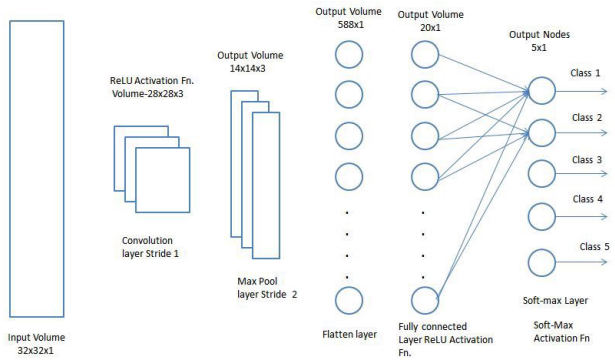


Figure 13.8: Classification with a fully connected layer

Chapter 14

Deep Learning: Advanced Approaches

In this chapter we will make a small introduction on deep learning, focusing on 3 principal topics/features:

1. *Modeling temporal information*: Recurrent Neural Networks (RNN) and Long-Short Term Memory (LSTM)
2. *Generative models*: Generative Adversarial Networks (GAN)
3. *Advanced CNN schemes*: Residual networks, skip connections, auto-encoders

14.1 Exploit Temporal Information

Not all problems can be fitted into a representation with fixed-length inputs and outputs! We just need to think, for example, to a problem that act on a series of bits and requires an output "YES" if the number of 1s is even, or "NO" if it is not (e.g. "1000010101" is a YES (4 ones) while "100011" is NO (3 ones)). So it is hard or impossible to choose a fixed context window from problems like this, because there can always be a new sample longer than anything seen until now.
Also problem such as *speed recognition* or *time-series prediction* require a system able to store and use context information.
In this case, the so called **Recurrent Neural Networks** (RNN) come into our help. Such algorithms, schematized in figure 14.1, take the previous outputs (or hidden state) as new inputs, and with him all the "historical" information brought by it. RNNs are useful as their

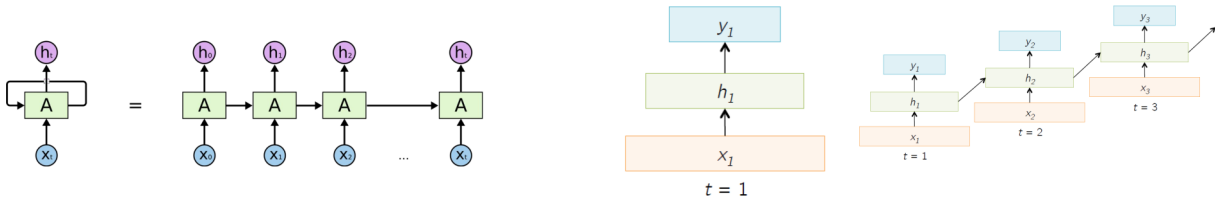


Figure 14.1: Recurrent Neural Networks

intermediate values (state) can store information about past inputs for a time that is not fixed a priori.