SAPIENZA
UNIVERSITÀ DI ROMA

"Sapienza" University of Rome
Faculty of Information Engineering,
Informatics and Statistics
Department of Computer Science

Machine Learning

Lecture notes integrated with the book "Machine Learning", Tom Mitchell

Author
Simone Bianco

February 19, 2025

Contents

Information and Contacts

Personal notes and summaries collected as part of the *Machine Learning* course offered by the degree in Computer Science of the University of Rome "La Sapienza".

Further information and notes can be found at the following link:
https://github.com/Exyss/university-notes. Anyone can feel free to report inaccuracies, improvements or requests through the Issue system provided by GitHub itself or by contacting the author privately:

- Email: **bianco.simone@outlook.it**
- LinkedIn: **Simone Bianco**

The notes are constantly being updated, so please check if the changes have already been made in the most recent version.

Suggested prerequisites:

Sufficient knowledge of calculus, probability and algorithm design

Licence:

1

Introduction on machine learning

1.1 What is machine learning?

While many common tasks can be easily solved by computers through an algorithm, some are hard to formalize as a series of steps to be executed in a deterministic way. As an analogy, consider how language is made of syntax and semantics. Syntax can easily be formalized as a sequence of sub-structures that make up a phrase. If a sentence is slightly malformed, the machine can have a hard time trying to reconstruct the correct syntax, but in most cases this can be achieved. For semantics, instead, we have a whole different problem: some words could have more meanings, giving sentences different interpretations depending on the context of the conversation. This task is clearly harder for a machine. Sometimes, not even humans are capable of solving it!

In the past, these type of tasks where solved through *expert systems*, that being any system programmed by a human expert to solve a specific task. Expert systems can be viewed as a sequence of if-else conditions: if the task requires x then do y, and so on. Not all tasks can be solved through expert systems. In particular, some tasks need different solutions for many cases, making these primitive systems useless due to all cases being impossible to program.

To solve this type of complex and variable tasks, we use **machine learning**, which slowly teaches the machine how to solve the problem in the best way possible. The idea here is to program computers in a way that improves a specific *performance criterion* through *example data* and *past experiences*.

Machine learning uses **data mining** — the act of producing knowledge from known data — to increase the experience of the machine in solving the designed problem. In general, machine learning comes in handy when one of the following conditions holds:

- There is no human expertise on the task
- Human experts are unable to explain their expertise
- The solution needs to adapt to particular cases

The field of machine learning had an exponential growth in recent years due to the growing flood of online data — the so called *big data phenomenon* — and the increase of computational power to process such data through advanced algorithms based on theoretical results. First, we give a formal definition of a *learning problem*.

Definition 1: Learning problem

A **learning problem** is the improvement over a task T with respect to a performance measure P based on experience E.

For example, suppose that we want to program a machine that learns how to play checkers. We define the learning problem as:

- The task T is to play checkers
- The performance measure P is the percentage of games won in a tournament
- The experience E is the opportunity to play against self

But how can we improve such performance measure? What *exactly* should the machine learn? These questions reduce the learning problem to finding a valid mathematical representation of T, P and E. The training process can be described by four phases:

1. The human expert suggests what is an optimal move for each configuration of the board
2. The human expert evaluates each configuration, ranking them by optimality
3. The computer plays against a human an automatically detects with configurations lead to a win, a loss or a draw
4. The computer plays against itself to improve performance

Formally, this whole process can be expressed as a simple mathematical function called **target function**. In particular, we want to choose a target function that represents the learning problem in the best way possible and that can be computed by a machine.

For instance, consider the function $V : \text{Board} \to \mathbb{R}$, defined as follows:

- If b is a final board state and it is a win then $V(b) = 100$
- If b is a final board state and it is a loss then $V(b) = -100$
- If b is a final board state and it is a draw then $V(b) = 0$
- If b is not a final board state then $V(b) = V(b^*)$, where b^* is the best final board state that can be achieved starting from the board b playing the optimal moves

This function perfectly models our learning problem. However, it cannot be computed by any program since we haven't defined what an optimal set of moves is. We need a new definition that encodes this concept of optimal strategy for a checkers game.

For example, we can re-define V as follows:

$$V(b) = w_0 + w_1 \cdot \text{bp}(b) + w_2 \cdot \text{rp}(b) + w_3 \cdot \text{bk}(b) + w_4 \cdot \text{rk}(b) + w_5 \cdot \text{bt}(b) + w_6 \cdot \text{rt}(b)$$

where:

- $\text{bp}(b)$ is the number of black pieces
- $\text{rp}(b)$ is the number of red pieces
- $\text{bk}(b)$ is the number of black kings
- $\text{rk}(b)$ is the number of red kings
- $\text{bt}(b)$ is the number of red pieces threatened by black pieces
- $\text{rt}(b)$ is the number of black pieces threatened by red pieces

With this formulation, we have reduced the concept of leaning checkers to estimating the best possible values of the coefficients $w_1, \ldots, w_6$, which care called **weights**, that maximize the value of $V(b)$ for any board state b. This estimation process is referred to as *learning the function* V.

Definition 2: Learned function

Given a target function $f : X \to Y$ with weights $w_1, \ldots, w_k$, we define the **learned function** $\widehat{f} : X \to Y$ as the current approximation of f computed by a learning algorithm.

By definition, the learned function $\widehat{f}$ will never be equal to the target function f: the target function's weights are always unknown by definition. The idea here is to approximate f by repeatedly applying small changes to the weights $\widehat{w_1}, \ldots, \widehat{w_k}$ of $\widehat{f}$ in order to estimate the weights of f. To learn a function f, we need a *dataset*. A dataset is a set of instances that can be used by a learning algorithm to improve the performance of the learned function.

Definition 3: Dataset

Let $f : X \to Y$ be a target function and let $f_{\text{train}}(x)$ be the training value obtained by x in the training data. Given a set of n training inputs $X_D = \{x_1, \ldots, x_n\}$, the **dataset** of the learning problem is the set of samples defined as:

$$D = \{(x_i, f_{\text{train}}(x_i)) \mid x_i \in X_D\}$$

After training, the learned function $\widehat{f}$ will have learned the values of the inputs in the dataset, returning a value as close as possible to the one in the dataset (in some cases the returned value is exactly the same). However, we are interested in estimating the *other* possible inputs, i.e. those that aren't in the dataset.

In summary, a machine learning problem is the task of learning a target function $f : X \to Y$ through a dataset D for a set X_D of n inputs. To learn a function f means computing an approximating function $\widehat{f}$ that returns values as close as possible to f, especially for values outside of the dataset D, implying that $\forall x \in X - X_D$ it should hold that $f(x) \approx \widehat{f}(x)$. In order for the learned function to be *good*, the set of training inputs X_D must be very very small compared to the set of total inputs, meaning that $|X_D| \lll |X|$.

There are distinct types of machine learning problems based on:

- The type of dataset used:
  1. **Supervised learning**: problems where the model learns patterns from labeled data. Here, the dataset corresponds to $D = \{(x_i, y_i) \mid i \in X_D\}$, where y_i is the sample of the function value for x_i
  2. **Unsupervised learning**: problems where the model learns patterns from unlabeled data. Here, the dataset corresponds to $D = \{x_i \mid i \in X_D\}$
  3. **Reinforcement learning**: problems in which an agent learns to make decisions by interacting with an environment and receiving rewards or penalties based on its actions.
- The type of function to be learned:
  1. **Discrete Classification**: the input set is $X = A_1 \times \ldots \times A_n$, where A_i is a finite set, and the output set is $Y = \{C_1, \ldots, C_k\}$. When $k = 2$, i.e. we have only two classes, we say that the problem is a **Concept Learning** problem
  2. **Discrete Regression**: the input set is $X = A_1 \times \ldots \times A_n$, where A_i is a finite set, and the output set is $Y = \mathbb{R}^m$.
  3. **Continuous Classification**: the input set is $X = \mathbb{R}^n$ and the output set is $Y = \{C_1, \ldots, C_k\}$.
  4. **Continuous Regression**: the input set is $X = \mathbb{R}^n$ and the output set is $Y = \mathbb{R}^m$.

Classification problems are based on the classification of inputs into predetermined categories, while regression problems involve the approximation of functions defined over $\mathbb{R}$. Reinforcement learning, instead, is used for dynamic systems with unknown or partially known evolution model, usually robotic tasks and game playing.

1.2 Hypotheses, consistency and representation

After discussing the basic notation and terminology, we are ready to deepen our understanding on how to learn a problem.

Definition 4: Hypothesis space

Given a target function f, an **hypothesis space** H for f is a set of functions $h \in H$, where h is called hypothesis, that can be learned in order to reach an approximation of f.

The representation of a hypothesis space highly depends on the type of problem. For instance, consider the problem of classifying natural numbers into primes numbers and composite numbers. This corresponds to a discrete classification problem with two classes $\mathbb{P}$ and $\mathbb{N}-\mathbb{P}$, i.e. a concept learning problem where we want to learn the concept of prime number. The target function of the problem is thus described as $f : \mathbb{N} \to \{\mathbb{P}, \mathbb{N}-\mathbb{P}\}$. Here, the simplest hypothesis space is the set $H = \{h_A : \mathbb{N} \to \{A, \mathbb{N}-A\} \mid A \subseteq \mathbb{N}\}$.

Given a performance measure P over a dataset D and a hypothesis space H, the learning task is to find the best approximation $h^* \in H$ of the function f using the dataset D.

$$h^* \in \arg\max_{h \in H} P(h, D)$$

Finding such optimal hypothesis is the core of a learning problem. However, by definition, a hypothesis space may also contain hypotheses that are clearly a wrong approximation of f over D. For instance, given a dataset D for a target function f, for any hypothesis $h \in H$ we can check wether $h(x) = f(x)$ only for instances $x \in X_D$ since the other values are unknown in D. This means that some hypothesis may have some values that are *inconsistent* with the dataset itself, making them useless in the learning process. A hypothesis $h \in H$ is said to be **consistent** with a dataset D of a target function f if and only if $h(x) = f(x)$ for all $x \in X_D$. The subset of hypothesis that are consistent with a dataset is called **version space**.

Definition 5: Version space

The **version space** of a target function f with respect to the hypothesis space H and the dataset D, written as $\mathrm{VS}_{H,D}$, is the subset of H that contains all the hypotheses that are consistent with D.

$$\mathrm{VS}_{H,D} = \{h \in H \mid \forall x \in X_D \;\; h(x) = f(x)\}$$

Consider again the previous example. Suppose that we're working with the following dataset $D = \{(1, \mathbb{N}-\mathbb{P}), (3, \mathbb{P}), (5, \mathbb{P}), (6, \mathbb{N}-\mathbb{P}), (7, \mathbb{P}), (10, \mathbb{N}-\mathbb{P})\}$. Here, the version space would restrict our interest to all those functions $h_A : \mathbb{N} \to \{A, \mathbb{N}-A\}$ where the elements $3, 5, 7$ lie inside A and the elements $1, 6, 10$ lie outside of A.

$$VS_{H,D} = \{h_A : \mathbb{N} \to \{A, \mathbb{N}-A\} \mid \{3,5,7\} \subseteq A \subseteq \mathbb{N} \text{ and } \{1,6,10\} \subseteq \mathbb{N}-A\}$$

By definition, the best approximation for a performance measure P over a dataset D and a hypothesis space H must clearly lie inside the version space $VS_{H,D}$. Hence, we can restrict our interest to the version space itself.

$$h^* \in \arg\max_{h \in VS_{H,D}} P(h, D)$$

The concept of version space is based on the **inductive learning hypothesis**: any hypothesis that is consistent with the target function over a dataset of *adequate size* will also approximate the target function well over other unobserved examples. In other words, if we consider a dataset of adequate size then every hypotheses inside the version space will be a nice approximation of the target function.

The simplest way to compute the version space is through the **List-Then-Eliminate** algorithm, a bruteforce algorithm that enumerates all the hypothesis space and then test the consistency of each hypothesis, discarding the invalid ones. Even though this algorithm is correct, it is also clearly *infeasible* since enumerating all the different hypothesis would require an exponential amount of time. We'll see improved ways to compute the version space given by a dataset.

Algorithm 1: List-Then-Eliminate

Given a hypothesis space H and a dataset D, the algorithm returns $VS_{H,D}$

```
function listThenEliminate(H, D)
    VS_{H,D} := H
    for (x, f(x)) ∈ D do
        H' := {h ∈ H | h(x) ≠ f(x)}
        VS_{H,D} = VS_{H,D} − H'
    end for
    return VS_{H,D}
end function
```

1.2.1 Representation power and generalization power

In order for the inductive learning hypothesis to hold, the size of the dataset is a **critical** factor: if the hypothesis space is too *powerful* and the search is complete, then the system won't be able to classify new instances, meaning that we have no generalization power.

For instance, consider a generic concept learning problem described by the target function $c : X \to \{0, 1\}$. Let D be the chosen dataset. For any hypothesis space H, it's easy to see that H is actually *associated* with a particular set of instances, that being all instances that are classified as positive by such hypothesis. In fact, we have a mapping ϕ between the hypothesis space H and the power set $\mathcal{P}(X)$.

$$\phi_H : H \to \mathcal{P}(X) : h_A \mapsto A = \{x \in X \mid h_A(x) = 1\}$$

In general, this mapping is not surjective, meaning that there is a subset A of $\mathcal{P}(X)$ that is is not covered by a hypothesis inside H. In fact, we prefer cases where such

mapping is not surjective. This is known as the **hypothesis space representation issue**: some hypothesis spaces may be useless even when we restrict our interest to the version space.

For example, suppose that there is a hypothesis spaces H_1, H_2 such that H_1 cannot represent $\mathcal{P}(X)$, i.e. ϕ_{H_1} is not surjective, while H_2 can:

- In $VS_{H_1,D}$ we have that $\exists x \in X - X_D$ for which there are two hypotheses $h, h' \in \text{VS}_{H_1,D}$ such that $h(x) = 0$ and $h'(x) = 1$.
- In $VS_{H_2,D}$ we have that $\forall x \in X - X_D$ there are two hypotheses $h, h' \in \text{VS}_{H_2,D}$ such that $h(x) = 0$ and $h'(x) = 1$.

The small difference in the quantifier has enormous impacts on the usefulness of these two spaces. If we use H_1 then we expect that the approximation found by any algorithm will give the wrong value for some unlabeled inputs $x \in X - X_D$ due to the presence of two functions that can be chosen for x. If we use H_2, instead, we expect that every unlabeled input will have a wrong value.

These observations imply that the *more information* the hypothesis space encapsulates about the values in X_D, the *harder* it becomes to **generalize** and predict values for samples outside X_D. In other words, a more expressive hypothesis space can **overfit** to the data, making it more difficult to make accurate predictions on unsampled data. We'll return on the problem of overfitting the dataset in following sections.

The process of reducing the *representation power* of the hypothesis space in favor of *generalization power* – as in reducing the hypothesis space from H' to H in the previous example – is called **language bias**. Ideally, we want our hypothesis space to be as good as possible. Clearly, the best possible hypothesis space contains only the optimal approximating function $h^* \in H$. In this case, the previous observations regarding the hypothesis space representation issue are solved: every unlabeled data will have only one value inside the function. The process of selecting one particular hypothesis among the set of possible ones – i.e., choosing $h^* \in H$ – is called **search bias**.

In machine learning, the concept of learning bias is crucial for improving a model's ability to generalize. A good learning bias helps guide the learning algorithm towards patterns in the data that are useful for predicting unseen samples, increasing the system's generalization power. This bias allows the model to make accurate predictions on new data that wasn't part of the training set. Without such a bias, a system would simply **memorize** the dataset, failing to predict values for samples outside the training set, rendering it *ineffective* in real-world applications. Systems lacking generalization capabilities would be of little use, as they wouldn't be able to provide meaningful predictions beyond the data they were trained on.

In real-world applications, datasets often contain **noise**, which refers to irrelevant or erroneous information that can distort the true underlying patterns in the data. Noise can come from a variety of sources, including measurement errors, data entry mistakes, incomplete data or random fluctuations in the system being studied. This noise complicates the *learning process*, as machine learning models may *struggle* to distinguish between true *signal* and *noise*.

A noisy data-point in a dataset D for a function f can be formulated as a pair $(x_i, y_i) \in D$, where $y_i \neq f(x_i)$. This means that there may be *no consistent hypothesis* with noisy data, i.e. $\text{VS}_{H,D} = \varnothing$. In these scenarios, **statistical methods** must be employed to implement robust algorithms, in order to reduce the noise in the data.

1.3 Performance evaluation

After discussing which hypotheses we're interested in, we'll now focus on evaluating the performance of a hypothesis. To give an intuition, we'll focus on classification problems. Let $f : X \to Y$ be a classification problem. Let $\mathcal{D}$ be a probability distribution over X, meaning that each element in X has an associated probability of being drawn. The sum of all the probabilities in $\mathcal{D}$ must be 1. Let S be a set of n samples drawn from X according to $\mathcal{D}$ and for which we know the value $f(x)$. By definition, S corresponds to a dataset of n random elements. For any possible hypothesis h returned by a learning algorithm obtained through S, we want to know what is the best estimate of the *accuracy* of h over future instances drawn from the same distribution and what is the *probable error* of this accuracy estimate. First, we have to define two error measures.

Definition 6: True error

Let $f : X \to Y$ be a classification problem. Given a hypothesis h, the **true error** of h with respect to f and a probability distribution $\mathcal{D}$ over X, written as $\text{error}_{\mathcal{D}}(h)$, is defined as:

$$\text{error}_{\mathcal{D}}(h) = \Pr_{x \in \mathcal{D}}[h(x) \neq f(x)]$$

The *true accuracy* of h is defined as $\text{accuracy}_{\mathcal{D}}(h) = 1 - \text{error}_{\mathcal{D}}(h)$.

The true error of a hypothesis is clearly a perfect error measure. However, it's easy to see that computing the true error of h is infeasible since we would have to compute such probability over all the inputs. To fix this, we consider another type of error measure that is less precise but computable.

Definition 7: Sample error

Let $f : X \to Y$ be a classification problem. Given a hypothesis h, the **sample error** of h with respect to f and a data sample S over $\mathcal{D}$, written as $\text{error}_S(h)$, is defined as:

$$\text{error}_{\mathcal{S}}(h) = \frac{1}{n} \sum_{x \in S} \delta(x)$$

where $\delta(x) = 1$ if $h(x) \neq f(x)$ and $\delta(x) = 0$ otherwise. The *sample accuracy* is defined as $\text{accuracy}_S(h) = 1 - \text{error}_S(h)$.

The sample error can be efficiently computed over a small data sample. The goal of a learning system is to be accurate in $h(x)$. However, sample accuracy may often fool us into thinking that our hypothesis is good: if $\text{accuracy}_S(h)$ is very high but but $\text{accuracy}_{\mathcal{D}}(h)$

is poor, our system is not be very useful.

We also notice that, by definition, $\text{error}_S(h)$ is actually a random variable depending on S. Sampling two different sets S and S' from $\mathcal{D}$ may different values for $\text{error}_S(h)$ and $\text{error}_{S'}(h)$. For this reason, we're also interested in the expected value $\mathbb{E}_{S\subseteq\mathcal{D}}[\text{error}_S(h)]$ of the sample error, i.e. the weighted average over all the possible samples S. The **estimation bias** for $\mathcal{D}$ is defined as the difference between the expected sample error and the true error.

$$\text{bias}_{\mathcal{D}}(h) = \mathop{\mathbb{E}}_{S\subseteq\mathcal{D}}[\text{error}_S(h)] - \text{error}_{\mathcal{D}}(h)$$

We notice that the estimation bias is equal to 0 if and only if $\mathbb{E}_{S\subseteq\mathcal{D}}[\text{error}_S(h)] = \text{error}_{\mathcal{D}}$. However, for any learning algorithm it is *impossible* to prove that the bias is exactly 0. This derives from the very concept of learning function. Hence, in order for a hypothesis to be as good as possible, we want an estimation bias that is as close as possible to 0.

If S is the training set used to compute h through some learning algorithm then $\text{error}_S(h)$ will always be have some bias. In order to get an **unbiased estimate**, the training set S used to compute h and the evaluation set S' must be chosen independently. However, even with an unbiased sample, $\text{error}_{S'}(h)$ may still vary from $\text{error}_{\mathcal{D}}(h)$ – the smaller the set S', the greater the expected variance.

But how good is an estimate of $\text{error}_{\mathcal{D}}(h)$ provided by a single $\text{errors}_S(h)$? From the theory of statistical analysis, we can derive the concept of **confidence interval**. If S contains n samples drawn independently of one another over $\mathcal{D}$, S is independent of h and $n \geq 30$ then with approximately $N\%$ of probability we have that:

$$|\text{error}_S(h) - \text{error}_{\mathcal{D}}(h)| \leq z_N \sqrt{\frac{\text{error}_S(h)(1-\text{error}_S(h))}{n}}$$

where the value z_n is a constant given by the *confidence level* $N\%$.

$N\%$	50%	68%	80%	90%	95%	98%	99%
z_n	0.67	1.00	1.28	1.64	1.96	2.33	2.58

Figure 1.1: Relation between each confidence level and its constant

This result shows that a single sample error (when unbiased) is also a good estimation measure for the true error, up to some confidence level. However, in order to get a good approximation we must make some trade offs between *training* and *testing*:

1. Using more samples for training and less for testing gives a better approximating function h, but $\text{error}_S(h)$ may not be a good approximation of $\text{error}_D(h)$.
2. Using less samples for training and more for testing gives a worse approximating function h', but $\text{error}_S(h')$ is guaranteed to be a good approximation of $\text{error}_D(h)$.

Usually, a good trade off for medium sized datasets is a training set of size $\frac{2}{3}|X|$ and a testing set of size $\frac{1}{3}|X|$. However, computing $\text{error}_S(h)$ is not enough: the estimation bias is defined through $\mathbb{E}_{S\subseteq\mathcal{D}}[\text{error}_S(h)]$. Computing the expected sample error is a task as

hard as computing the true error. Nonetheless, thanks to the **central limit theorem**, the expected sample error can be easily approximated by averaging the computed values of $\text{error}_{S_1}(h), \ldots, \text{error}_{S_k}$, where $S_1, \ldots, S_k$ are independent from each other.

$$\lim_{k \to +\infty} \overline{\varepsilon}_k - \mathop{\mathbb{E}}_{S \subseteq \mathcal{D}}[\text{error}_S(h)] = 0 \implies \lim_{k \to +\infty} \overline{\varepsilon}_k = \mathop{\mathbb{E}}_{S \subseteq \mathcal{D}}[\text{error}_S(h)]$$

where $\overline{\varepsilon}_k = \frac{1}{k} \sum_{i \in [k]} \text{error}_{S_i}(h)$. With all the tools that we have discussed, we're now ready to discuss our first unbiased estimator: the **K-Fold Cross Validation** algorithm.

Algorithm 2: K-Fold Cross Validation for Sample Error

Given a dataset D, a learning algorithm L for a function f and a value $k > 0$, the K-Fold Cross Validation returns an estimation $\text{error}_{L,D}$ of the expected sample error of L over D.

function kFoldCrossValidation(D, L, k)
  Partition D into k sets $S_1, \ldots, S_k$ where $|S_i| > 30$
  for $i = 1, \ldots, k$ **do**
    $T_i = D - S_i$ ▷ T_i is the training set
    $h_i = L(T_i)$
    $\delta_i = \text{error}_{S_i}(h_i)$
  end for
  return $\frac{1}{k} \sum_{i \in [k]} \delta_i$
end function

1.3.1 Hypothesis comparison

Consider the case where we have two independent hypotheses h_1 and h_2 for some discrete-valued target function. Hypothesis h_1 has been tested on a sample S_1 and h_2 has been tested on an sample S_2. S_1 and S_2 are independent from each other and they are drawn from the same distribution $\mathcal{D}$. The difference d between the true errors of these two hypotheses is given by

$$d = \text{error}_{\mathcal{D}}(h_1) - \text{error}_{\mathcal{D}}(h_2)$$

The obvious choice for the estimator $\widehat{d}$ of d is given by

$$\widehat{d} = \text{error}_{S_1}(h_1) - \text{error}_{S_2}(h_2)$$

In fact, it's very easy to prove that this is indeed an unbiased estimator.

$$\mathop{\mathbb{E}}_{S \subseteq \mathcal{D}}[\text{error}_S(h_1) - \text{error}_S(h_2)] = \mathop{\mathbb{E}}_{S \subseteq \mathcal{D}}[\text{error}_S(h_1)] - \mathop{\mathbb{E}}_{S \subseteq \mathcal{D}}[\text{error}_S(h)] = d$$

This definition of error comparison allows us to mathematically describe the concept of **overfitting**. In previous sections, we described this phenomenon as the property of a hypothesis of being too much expressive, i.e. it performs pretty good on the training data but it makes poor predictions on unsampled data.

Definition 8: Overfitting

Let $h \in H$ be a hypothesis for a function f obtained through the training data S, where H is the chosen hypothesis space. We say that h **overfits** the set S if there is an another hypothesis $h' \in H$ obtainable through S that has a sample error higher than h but a true error lower than h.

$$\text{error}_S(h) < \text{error}_S(h') \qquad \text{error}_{\mathcal{D}}(h) > \text{error}_{\mathcal{D}}(h')$$

Suppose that we have two learning algorithms L_A and L_B. We want to determine which of them produces the best approximation of f on average. The K-Fold Cross Validation algorithm that we discussed in the previous section can be easily adapted to compare L_A and L_B.

Algorithm 3: K-Fold Cross Validation for Comparison

Given a dataset D, two learning algorithms L_A, L_B for a function f and a value $k > 0$, if the algorithm returns True then we expect that L_A is better than L_B

function kFoldCrossComparison($D, L_A, L_B k$)
  Partition D into k sets $S_1, \ldots, S_k$ where $|S_i| > 30$
  for $i = 1, \ldots, k$ **do**
    $T_i = D - S_i$ ▷ T_i is the training set
    $h_i = L_A(T_i)$
    $h'_i = L_B(T_i)$
    $\delta_i = \text{error}_{S_i}(h_i) - \text{error}_{S_i}(h'_i)$
  end for
  $\varepsilon = \frac{1}{k} \sum_{i \in [k]} \delta_i$
  return $\varepsilon < 0$
end function

1.3.2 Performance metrics

Until now, we have focused on error and accuracy. Intuitively, we expect that if the accuracy is high, i.e. the error is low, then our hypothesis must be a good approximation. However, this is not always the case.

For instance, consider a concept learning problem $f : X \to \{+, -\}$ with a training set where 90% of the samples are negative, meaning that they are labeled with $-$. Consider the hypothesis h that labels every single element $x \in X_D$ as negative. Even if this hypothesis is clearly bad, the accuracy would still be 90%. Of course, we're usually interested in the accuracy of approximations yield by learning models. In some cases, accuracy only is not enough to assess the performance of a classification method.

In a more statistical sense, error and accuracy can be also described through the concept of positives and negatives:

- A **true positive** is a positive element that gets classified as positive

- A **true negative** is a negative element that gets classified as negative
- A **false positive** is a negative element that gets classified as positive
- A **false negative** is a positive element that gets classified as negative

Here, the error rate is described as the ratio between all the errors (false positives and false negatives) and all the samples:

$$\text{error} = \frac{\text{FP} + \text{FN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}}$$

while the accuracy is still defined as the complement of the error.

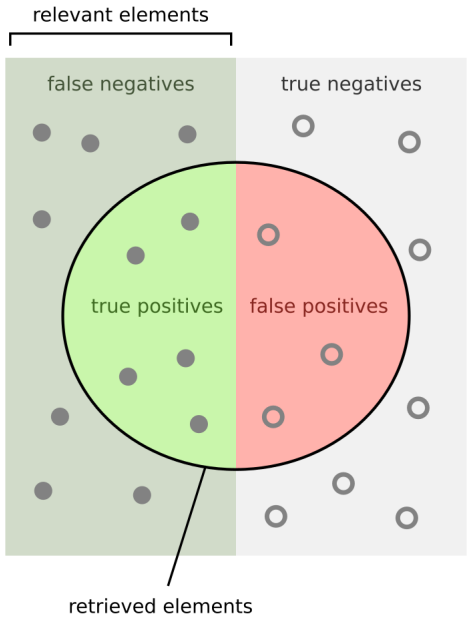

Figure 1.2: The relevant elements are the elements that are really positive, while the retrieved elements are the ones that have been labeled as positive

As discussed in the example above, these two measures may be inappropriate when the datasets are unbalanced. For this reason, we also consider two additional measures. The first measure is the **recall** (also called *true positive rate*), defined as the ratio between the true positives and the relevant elements. The recall measures the ability of the model to avoid false negatives.

$$\text{recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

The second measure is the **precision** (also called *positive predictive value*), defined as the ratio between the true positives and the retrieved elements. The precision measures the ability of the model to avoid false positives.

$$\text{precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

The recall and the precision are usually used to compute the **F-score**, which measures the predictive performance of a model. More formally, the F_1-score is the harmonic mean of the precision and recall. Thus, it symmetrically represents both precision and recall in one single metric.

$$\mathrm{F}_1 = \frac{2 \cdot \mathrm{recall} \cdot \mathrm{precision}}{\mathrm{recall} + \mathrm{precision}}$$

Another very easy to read measure linked to classification errors is the **confusion matrix**. Given a classification problem $f : X \to Y$ and a hypothesis h produced by a model for f, each row and column of the confusion matrix is labeled with the classes in Y. Each cell $c_{i,j}$ of a confusion matrix C contains the number of samples whose true class is C_i and whose class predicted by h is C_j. When the confusion matrix produced by the analysis of the performance of a solution retrieved through an algorithm is "accumulated" towards the central diagonal, the solution is a good model for the problem. Otherwise, the model is considered bad. The ratio between the sum of the main diagonal and the total number of samples corresponds to the accuracy of the model.

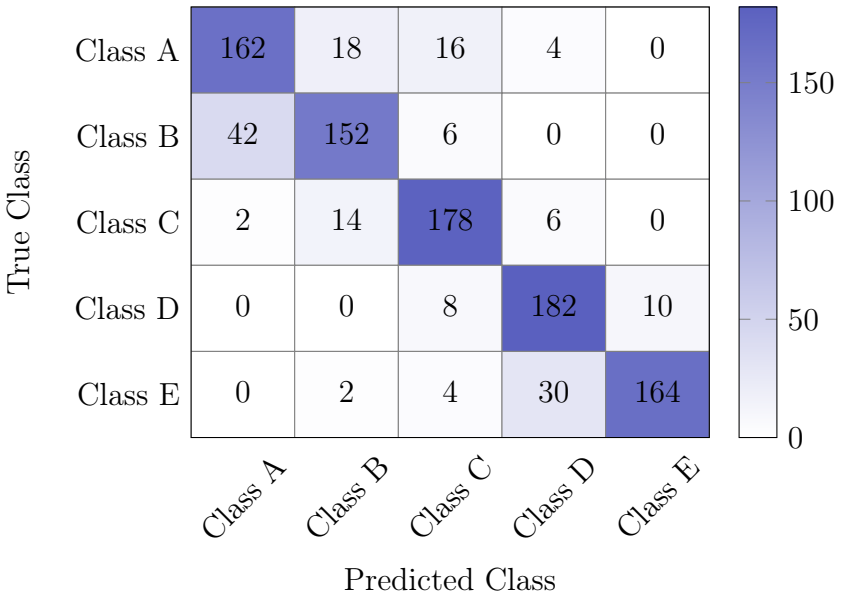

Figure 1.3: Example of a good confusion matrix for a dataset with 200 samples per class. The sum of the values in each row i must be equal to the number of samples in the dataset with true class C_i (sum equal to 200 in this example). The accuracy of this model is 0.843.

Sometimes, the values inside a confusion matrix are represented as percentages: each cell $c_{i,j}$ contains the ratio between the number of samples whose true class is C_i and whose predicted class is C_j over the total number of samples whose true class is C_i. In this case, the diagonal represents the *accuracy* of the model for each class in Y.

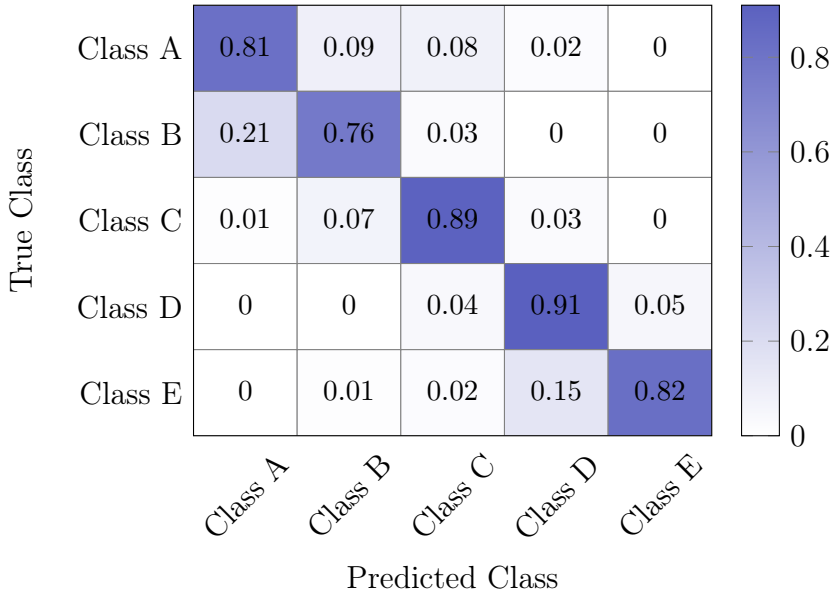

Figure 1.4: Percentage representation of the previous confusion matrix. The sum of the values in each row must be equal to 1.

For regression problems $f : X \to \mathbb{R}^k$, we need a different type of metrics. In particular, given a test set $S = \{(x_i, t_i)\}_{i\in[N]}$, performance can be measured in various ways:

- **Mean Absolute Error (MAE)**

$$\frac{1}{N}\sum_{i=1}^{N}\left|\widehat{f}(x_i) - t_i\right|$$

- **Mean Squared Error (MSE)**

$$\frac{1}{N}\sum_{i=1}^{N}(\widehat{f}(x_i) - t_i)^2$$

- **Root Mean Squared Error (RMSE)**

$$\sqrt{\frac{1}{N}\sum_{i=1}^{N}(\widehat{f}(x_i) - t_i)^2}$$

The K-Fold Cross Validation algorithm can also be adapted to compute these performance measures.

2

Decision trees

2.1 Decision trees learning

After discussing the general idea behind machine learning problems and how to evaluate the solution retrieved by an algorithm, we're now ready to focus on how these algorithms work. In particular, we'll start by discussing decision trees.

Given a discrete input space described by m attributes $X = A_1 \times \ldots \times A_m$, where each A_i is a finite set, and a problem $f : X \to Y$, a **decision tree** is a n-ary tree where:

- Each internal node is labeled by an attribute A_i
- Each branch outgoing from a node labeled with A_i denotes a possible value of an attribute $v \in \text{Values}(A_i)$, where the latter is the set of possible values for A_i
- Each leaf node of the tree is labeled with a class $j \in Y$

Given an input, a decision tree computes by querying the internal nodes (starting from the root) and always proceeds on the edge corresponding to the attribute value assumed by the input. In other words, a decision tree is nothing more than a set of rules that classifies any given input. Decision trees are a very easy computational model and they are capable of computing any discrete function. For example, suppose that we want to solve the concept learning problem $\text{PlayTennis} : X \to \{\text{Yes}, \text{No}\}$ relative to deciding if good conditions are met in order to play tennis. The input set that we'll be working with is the following:

$$X = \{\text{Outlook} \times \text{Temperature} \times \text{Humidity} \times \text{Wind}\}$$

where:

$$\text{Outlook} = \{\text{Sunny}, \text{Overcast}, \text{Rain}\} \qquad \text{Temperature} = \{\text{Hot}, \text{Mild}, \text{Cold}\}$$

$$\text{Humidity} = \{\text{Normal}, \text{High}\} \qquad \text{Wind} = \{\text{Weak}, \text{Strong}\}$$

Day	Outlook	Temperature	Humidity	Wind	PlayTennis
D1	Sunny	Hot	High	Weak	No
D2	Sunny	Hot	High	Strong	No
D3	Overcast	Hot	High	Weak	Yes
D4	Rain	Mild	High	Weak	Yes
D5	Rain	Cool	Normal	Weak	Yes
D6	Rain	Cool	Normal	Strong	No
D7	Overcast	Cool	Normal	Strong	Yes
D8	Sunny	Mild	High	Weak	No
D9	Sunny	Cool	Normal	Weak	Yes
D10	Rain	Mild	Normal	Weak	Yes
D11	Sunny	Mild	Normal	Strong	Yes
D12	Overcast	Mild	High	Strong	Yes
D13	Overcast	Hot	Normal	Weak	Yes
D14	Rain	Mild	High	Strong	No

Figure 2.1: Example dataset for *PlayTennis*

A decision tree for this problem would look like the following:

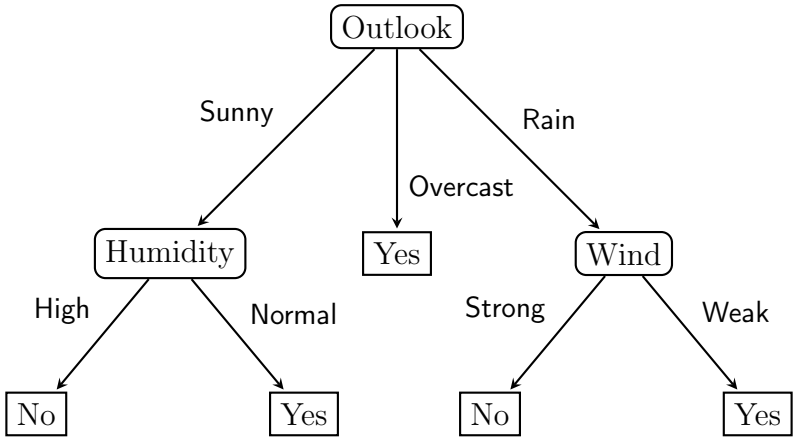

Figure 2.2: Example of decision tree for *PlayTennis*.

In concept learning problems, each decision tree can be described as a disjunction of conjunction of the attributes tested by the tree. In particular, we consider only paths that end up with a positive leaf. For instance, the tree described above can also be represented as following:

$$(\text{Outlook} = \text{Sunny} \wedge \text{Humidity} = \text{Normal})$$

$$\vee(\text{Outlook} = \text{Overcast})\vee$$

$$(\text{Outlook} = \text{Rain} \wedge \text{Wind} = \text{Weak})$$

2.1.1 Entropy and the ID3 algorithm

In this context, the hypothesis space is the *set of all possible decision trees*. We'll now define the algorithm that we'll be using to derive an optimal decision tree from a given dataset: the **Iterative Dichotomiser 3 (ID3)** algorithm.

Algorithm 4: ID3 algorithm

Given a dataset D for a concept learning problem $f : X \to \{+, -\}$, an attribute list L containing the attributes of X, the algorithm returns a decision tree for f.

function ID3(D, L)
  Initialize an empty decision tree T
  Create a root node in T
  if All the examples in D are positive **then**
    Label the root with $+$
  else if All the examples in D are negative **then**
    Label the root with $-$
  else if $L = \varnothing$ **then**
    Label the root with the most common value for A in D
  else
    Let A be the *best decision attribute* in L for D
    Label the root with A
    for $v \in$ Values(A) **do**
      Let D_a be the subset of D whose tuples have A set to v
      if $D_a = \varnothing$ **then**
        Create a leaf node labeled with the most common value for A in D
        Add an edge labeled with $A = v$ from the root to the leaf node
      else
        Compute the subtree $T_A =$ ID3($D_a, L - \{A\}$)
        Add an edge labeled with $A = v$ from the root to T_A
      end if
    end for
  end if
  Return T
end function

In the ID3 algorithm described above, the *best decision attribute* is an attribute whose optimality depends on a pre-defined criteria. Based on the criteria chosen, different attributes may be selected. Trying to define an optimal criteria is crucial. In fact, based on the chosen attribute order, we may get a completely different decision tree.

The commonly used measure for this task is the **information gain**, which measures how well a given attribute *separates* the training examples according to their target classification. The information gain of an attribute is measured as reduction in **entropy**, which measures the impurity of a sample.

Definition 9: Entropy

Let S be a sample set for a classification problem $f : X \to Y$. For each $i \in Y$, we denote with p_i the proportion of elements of S that are classified as C_i. The **entropy** of S is defined as:

$$\text{Entropy}(S) = -\sum_{i \in Y} p_i \log_2 p_i$$

Note: we assume that $0 \log_2 0 = 0$

In the special case of boolean classification problems – since we have only two classes – we denote with $p_\oplus$ and $p_\ominus$ the proportions of positive and negative samples in S ($p_\ominus = 1 - p_\oplus$). In this case, the entropy of S is defined as:

$$\text{Entropy} = -p_\oplus \log_2 p_\oplus - p_\ominus \log_2 p_\ominus$$

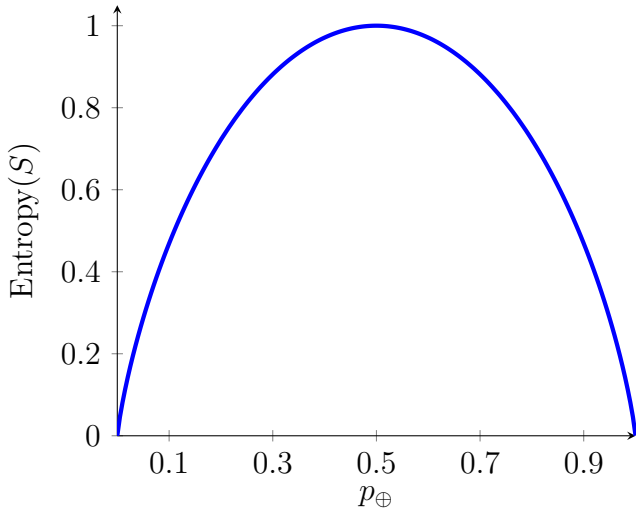

Figure 2.3: The entropy function relative to a boolean classification

Through the graph above, it's easy to see that in boolean classification problems the entropy reaches its maximum value when $p_\oplus = 0.5$, i.e. when the two classes are perfectly balanced, while it reaches its minimum when $p_\oplus = 0$ or $p_\oplus = 1$.

Definition 10: Information gain

The **information gain** for a sample set S is defined as the expected reduction in entropy of S caused by the selection of a value for the attribute A.

$$\text{Gain}(S, A) = \text{Entropy}(S) - \sum_{v \in \text{Values}(A)} \frac{|S_v|}{|S|} \text{Entropy}(S_v)$$

where $S_v = \{s \in S \mid A(s) = v\}$

Suppose that we have a sample set $S = [9+, 5-]$ (where this notation implies that we have 9 positives and 5 negatives) and that we are testing the information gain of the attribute $\text{Wind} = \{\text{Weak}, \text{Strong}\}$. First, we compute the entropy of S:

$$\text{Entropy}(S) = -\frac{9}{14}\log_2\left(\frac{9}{14}\right) - \frac{5}{14}\log_2\left(\frac{5}{14}\right) \approx 0.940$$

After looking at the sample set, we get that $S_{\text{Weak}} = [6+, 2-]$ and $S_{\text{Strong}} = [3+, 3-]$. The entropy of these two subsets corresponds to:

$$\text{Entropy}(S_{\text{Weak}}) = -\frac{6}{8}\log_2\left(\frac{6}{8}\right) - \frac{2}{8}\log_2\left(\frac{2}{8}\right) \approx 0.811$$

$$\text{Entropy}(S_{\text{Strong}}) = -\frac{3}{6}\log_2\left(\frac{3}{6}\right) - \frac{3}{6}\log_2\left(\frac{3}{6}\right) = 1$$

Hence, the information gain on S knowing the attribute Wind corresponds to:

$$\text{Gain}(S, \text{Wind}) = \text{Entropy}(S) - \frac{8}{14}\text{Entropy}(S_{\text{Weak}}) - \frac{6}{14}\text{Entropy}(S_{\text{Strng}}) \approx 0.048$$

Each time the ID3 algorithm has to select the best attribute, it always chooses the one with the highest information gain, corresponding to the one that most reduces the entropy. Eventually, the entropy of the "implicitely partitioned" dataset will reach 1 or 0, meaning that the associated portion of the dataset contains only positive or negative values.

Consider again the dataset S shown in Figure 2.1. This dataset contains 9 positive entries and 5 negative entries. First, the ID3 algorithm computes the information gain of all of the four attributes, finding that Outlook is the attribute that yields the highest gain.

Outlook	Temperature	Humidity	Wind
0.246	0.151	0.048	0.029

Figure 2.4: Gain on S for each attribute

After setting Outlook as the root node, the algorithm now computes the three subtrees, one for each possible value of Outlook. Since the dataset S_{Overcast} contains only positive samples, the leaf node is labeled with Yes. The two other branches, instead, have to recursively compute their subtrees using information gain.

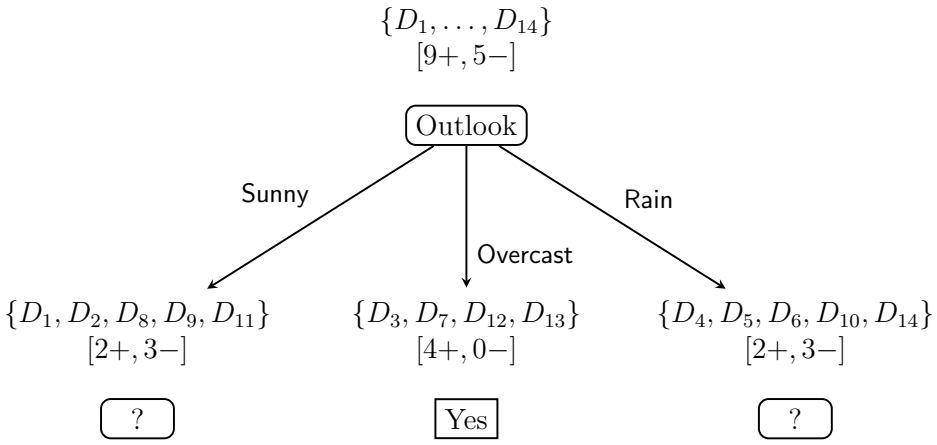

Figure 2.5: The partially learned decision tree after the first level of recursion

After repeating the process for each level of recursion, we get the decision tree shown in Figure 2.2. On each level of recursion, even though such hypothesis space is complete, meaning that every target function lies inside, the ID3 algorithm moves through such space in a greedy manner, returning only a **local minima**, while the target function is a **global minima**. Moreover, each step of the algorithm requires to analyze all the training examples, making this approach **not incremental** – if we want to add more data, the whole tree must be recomputed. However, this also ensures that our statistically-based search choices are robust to noisy data.

2.1.2 Overfitting in decision trees

A common issue in decision tree learning is the **size** of the decision tree yield by the algorithm, that is the number of nodes int the output tree on average. The importance of size comes from the nature of the algorithm itself: since the hypothesis space is complete, if allow the tree to have an uge number of nodes then it will eventually become a perfect approximation of the dataset.

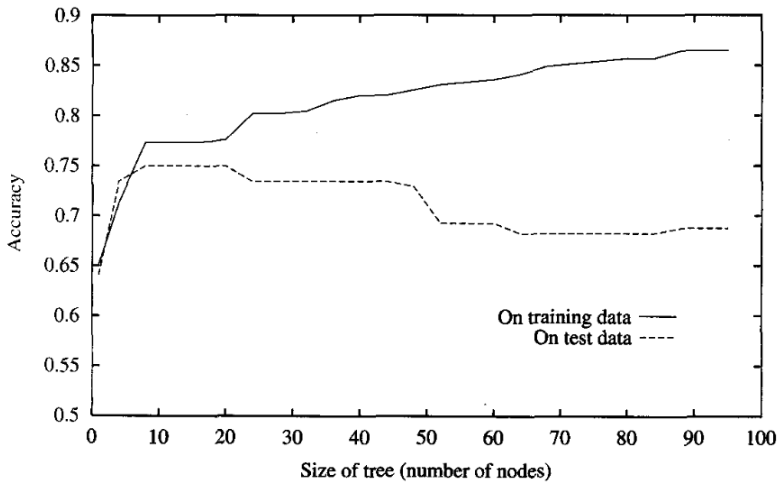

Figure 2.6: Overfitting in decision trees

In the worst case, each entry of the dataset is a leaf of the decision tree. When this happens, the solution found by the algorithm is clearly overfitting the data. In decision tree learning, overfitting is unavoidable. However, it can be reduced through some techniques. The first common technique involves enforcing a maximum growth on the tree by avoiding the recursive process when splitting the data wouldn't give a statistically significant improvement. The second technique grows the full decision tree and then prunes nodes that aren't significant. We'll focus on the second technique.

To determine the optimal tree size, we use a separate set of examples (distinct from the training examples) to evaluate the utility of post-pruning, then apply a statistical test to estimate accuracy of a tree on the entire data distribution.

In **reduced-error pruning**, the data is split into a training and a validation set.The easiest way to achieve each pruning step is to greedily select the subtree that is cut.

1. We copy the decision tree and randomly select a subtree
2. We replace the whole subtree it with a single leaf node labeled with the most common leaf label in the subtree.
3. We test the accuracy on the new subtree through the validation set

After evaluating the accuracy of every possible cut, we choose the one that increases the accuracy the most. Eventually, we'll reach a point where any additional cut will decrease the accuracy, concluding the pruning procedure.

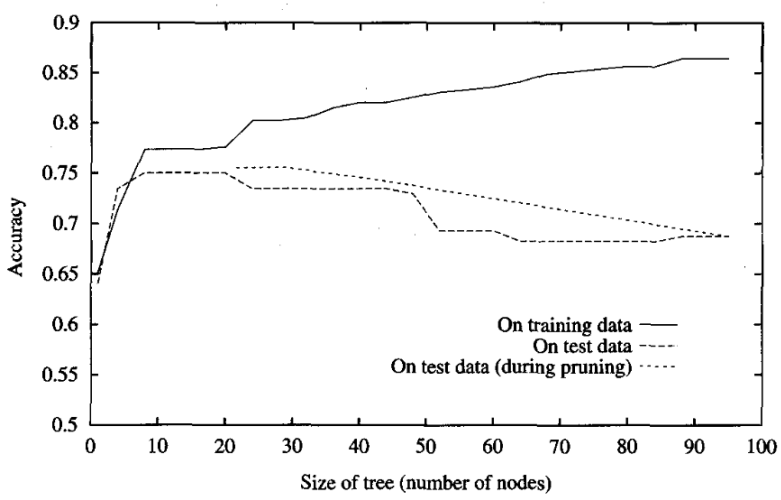

Figure 2.7: Performance after reduced-error pruning.

One more successful method for finding high accuracy hypotheses is the **rule post-pruning** technique. Variants of this technique are used by C4.5, another decision tree learning algorithm.

1. We first infer the decision tree allowing it to overfit the dataset
2. We convert the decision tree into an equivalent set of rules as shown in previous sections

3. We generalize each rule as much as possible independently of others
4. We remove multiple instances of the same generalized rule
5. We sort the final set of rules into a desired sequence
6. We convert the sequence back to a decision tree

Decision trees can also be used for classification of continuous-valued attributes. For instance, given the continuous-valued attribute Temperature, we can create a boolean variable that works with this attribute, such as "Temperature > 72.3". To work with such variables, the best approach is to pre-define the various splits of the set of continuous values.

Moreover, decision trees can also use multi-valued attributes, such as dates. However, the information gain of these variables is usually too high. To relax this issue, one approach is to use the **Gain Ratio** instead of the gain.

$$\text{GainRatio}(S, A) = \frac{\text{Gain}(S, A)}{\text{SplitInformation}(S, A)}$$

where $\text{SplitInformation}(S, A) = -\sum_{i \in Y} \frac{|S_i|}{|S|} \log_2 \left(\frac{|S_i|}{|S|} \right)$ and $S_v = \{s \in S \mid A(s) = v\}$

In some instances, we may want to give more importance to some particular attributes. To do so, we can define the **cost** for each attribute and replace the gain with one of the following values (they have no standard name)

$$\frac{\text{Gain}^2(S, A)}{\text{Cost}(A)} \qquad \frac{2^{\text{Gain}(S,A)} - 1}{(\text{Cost}(A) + 1)^w} \text{ for some } w \in [0, 1]$$

If some examples in the dataset have no value for the attribute selected by ID3, we can still use it through one of the following methods:

1. If node u tests A, assign most common value of A among other examples sorted to node u
2. Assign most common value of A among other examples with same target value
3. Assign a probability p_i to each possible value $v_i \in \text{Value}(A)$, assigning fractions of p_i to each descendant in tree

3

Probabilistic models

3.1 Uncertainty and probability

When the problem that we want to learn is based on actions that may assume continuous values, defining a sequence discrete decision becomes hard. For instance, consider the action A_t representing that Alice leaves for the airport t minutes before her flight. In order to know which t will satisfy the problem, we have to sort out many sub-problems, such as partial observability (road state, other drivers' plans, ...), noisy sensors (traffic reports), uncertainty in action outcomes (flat tire, ...), complexity of modelling and predicting traffic, ...

Hence an approach purely based on logical deduction will either risk falsehood, leads to conclusions that are too weak for decision making due to them requiring way too many conditions to be met ("A_{25} get me there on time if there's no accident on the bridge and it doesn't rain and my tires remain intact ...") or may lead to non-optimal decisions that ensure the outcome ("A_{1440} will surely suffice, but I have to stay overnight in the airport"). When this is the case, the best option is to just accept **uncertainties** and find a way to work with them. In particular, this implies using a probabilistic approach.

Definition 11: Sample space and probability space

A **sample space** Ω is a set of finite or infinite outcomes. The elements $\omega \in \Omega$ are usually called *atomic event* or *outcome of a random process*.

A **probability space** is a function $\Pr : \Omega \to [0, 1]$ defined on a sample space where the sum of all the values equals 1

$$\sum_{\omega \in \Omega} \Pr[\omega] = 1$$

Given a sample space Ω, a probability space associates a probability in the range $[0, 1]$ to each element of Ω. Suppose that we want to model a probability space that represents

the outcomes of a dice roll. The sample space can be defined as:

$$\Omega = \{1, 2, 3, 4, 5, 6\}$$

If we're using a standard dice, the probability space will be defined as:

$$\Pr[\omega] = \left\langle \frac{1}{6}, \frac{1}{6}, \frac{1}{6}, \frac{1}{6}, \frac{1}{6}, \frac{1}{6} \right\rangle$$

If the dice is loaded, instead, some outcomes may be more-likely. For instance, we could have the following probability space:

$$\Pr[\omega] = \left\langle \frac{1}{2}, 0, \frac{1}{6}, 0, \frac{1}{6}, \frac{1}{6} \right\rangle$$

Definition 12: Event

Given a probability space $\Pr : \Omega \to [0, 1]$, an **event** is a subset of Ω. The probability of an event $A \subseteq \Omega$ is given by the sum of the probabilities of all of the outcomes in the event.

$$\Pr[A] = \sum_{\omega \in A} \Pr[\omega]$$

Using the previous loaded dice example, the event "roll a value lower than 4" is described by the event $A = \{1, 2, 3\}$ and its probability is $\Pr[A] = \frac{5}{6}$. Events are a simple way to reason about probability. However, sometimes they aren't as intuitive as they look. For this reason, we usually prefer to work with *random variables*.

Definition 13: Random variable

Given a probability space $\Pr : \Omega \to [0, 1]$, a **random variable** is a function $X : \Omega \to B$, where B is an arbitrary set of values.

A random variable associates one of the values in B to every single element of the sample space. To work with a random variable $X : \Omega \to B$, we often consider the event "$X = x_i$", where $x_i \in B$. This event is equivalent to the set $\{\omega \in \Omega \mid X(\omega) = x_i\}$. Due to this, a random variable can be viewed both as a function and a variable.

$$\Pr[X = x_i] = \Pr[\{\omega \in \Omega \mid X(\omega) = x_i\}] = \sum_{\substack{\omega \in \Omega : \\ X(\omega) = x_i}} \Pr[\omega]$$

When we're working with a boolean random variable, i.e. when the value set B is $\{0, 1\}$, we often denote the event "$X = 1$" with X, while the event "$X = 1$" is denoted with $\neg X$. This allows us to write intersections and unions of events as simple conjunctions and disjunctions of boolean random variables.

$$\Pr[\neg A \wedge B] = \sum_{\substack{\omega \in \Omega : \\ A(\omega) = 0, B(\omega) = 1}} \Pr[\omega]$$

Instead of working with a probability space in order to compute the probabilities of each value assumable by a random variable, we often directly consider a **probability distribution**, a function assigning a probability value to all possible assignments of a random variable. The *joint probability distribution* for a set of random variables gives the probability of every atomic joint event on those random variables.

Pr[PlayTennis ∧ Weather]		Weather			
		Sunny	Rainy	Cloudy	Snowy
PlayTennis	True	0.576	0.02	0.064	0.01
	False	0.144	0.08	0.016	0.09

Figure 3.1: Example of joint probability distribution

We notice that, by definition, the probability of the conjunction of two events is not always equal to the product of the two probabilities of the events.

$$\Pr[A \wedge B] \stackrel{?}{=} \Pr[A] \cdot \Pr[B]$$

When the equality holds, we say that the two events are **independent** from each other. For the probability of the disjunction of two events, instead, we can always use the following formula derived from the *inclusion-exclusion principle*:

$$\Pr[A \vee B] = \Pr[A] + \Pr[B] - \Pr[A \wedge B]$$

Random variables can also influence each others. For instance, if we know that some outcomes of an event A are more likely to happen when some conditions are met, we can restrict our interest to such cases. The probability of an event A given that an event B happened is written $\Pr[A \mid B]$. This is also known as **conditional** (or *posterior*) probability and its defined as:

$$\Pr[A \mid B] = \frac{\Pr[A, B]}{\Pr[B]}$$

where $\Pr[A, B] = \Pr[A \wedge B]$. We notice that the events A and B are independent if and only if $\Pr[A \mid B] = \Pr[A]$. Sometimes, we may be also be given *conditional probability distributions*.

Pr[PlayTennis | Weather]		Weather			
		Sunny	Rainy	Cloudy	Snowy
PlayTennis	True	0.8	0.2	0.8	0.1
	False	0.2	0.8	0.2	0.9

Figure 3.2: Example of conditional probability distribution

When this is the case, conditional probability can be used to compute joint probability. This is also known as the *product rule*.

$$\Pr[A, B] = \Pr[A \mid B] \cdot \Pr[B]$$

When the values of a random variable Y are mutually exclusive, we may compute the total probability of another random variable X through joint probability or conditional probability. This is called **total probability**.

$$\begin{aligned}\Pr[X = x] &= \Pr[(X = x, Y = y_1) \vee \ldots \vee (X = X, Y = y_k)] \\ &= \sum_{i=1}^{k} \Pr[X = x, Y = y_i] \\ &= \sum_{i=1}^{k} \Pr[X = x \mid Y = y_i] \cdot \Pr[Y = y_i]\end{aligned}$$

Conditional probability can clearly be viewed as a **normalization factor** α applied to a joint probability. For instance, given the conditional probability, we have that:

$$\Pr[A \mid B] = \frac{\Pr[A, B]}{\Pr[B]} = \alpha \Pr[A, B]$$

where the normalization factor is $\alpha = \frac{1}{\Pr[B]}$. Viewing such probability as nothing more than a normalized instance of the joint one allows us to reason about maximizing and minimizing values in a simpler way: since each element is afflicted by this constant factor, we can just ignore.

$$\operatorname*{arg\,max}_{x \in \text{Values}(X)} \Pr[X = x \mid Y = y] = \operatorname*{arg\,max}_{x \in \text{Values}(X)} \alpha \Pr[X = x, Y = y] = \operatorname*{arg\,max}_{x \in \text{Values}(X)} \Pr[X = x, Y = y]$$

The most important rule derived from the very definition of conditional probability is **Bayes' rule**. This rule allows us to invert the order of the events: to compute the probability of A given B, we can use the probability of B given A

Proposition 1: Bayes' rule

Given two events A and B, it holds that:

$$\Pr[A \mid B] = \frac{\Pr[B \mid A] \cdot \Pr[A]}{\Pr[B]}$$

Bayes' rule becomes even stronger in the context of maximizing and minimizing values thanks to normalization:

$$\operatorname*{arg\,max}_{x \in \text{Values}(X)} \Pr[X = x \mid Y = y] = \operatorname*{arg\,max}_{x \in \text{Values}(X)} \Pr[Y = y \mid X = x] \cdot \Pr[X = x]$$

Moreover, when the conditions $Y_1, \ldots, Y_k$ are independent from each other, we also get that:

$$\Pr[X \mid Y_1, \ldots, Y_k] = \alpha \Pr[Y_1, \ldots, Y_k \mid X] \cdot \Pr[X] = \alpha \Pr[Y_1 \mid X] \cdot \ldots \cdot \Pr[Y_k \mid X] \cdot \Pr[X]$$

Another interesting consequence of the product rule is the *chain rule*, where we repeatedly apply the product rule on each joint probability:

$$\begin{aligned}\Pr[X_1, \ldots, X_{n-1}, X_n] &= \Pr[X_1, \ldots, X_{n-1}] \cdot \Pr[X_n \mid X_1, \ldots, X_{n-1}] \\ &= \Pr[X_1, \ldots, X_{n-2}] \cdot \Pr[X_{n-1} \mid X_1, \ldots, X_{n-2}] \cdot \Pr[X_n \mid X_1, \ldots, X_{n-1}] \\ &= \prod_{i=1}^{n} \Pr[X_i \mid X_1, \ldots, X_{i-1}]\end{aligned}$$

The chain rule is usually used with **Bayesian networks**, a graphical notation for conditional independence assertions and hence for compact specification of full joint distributions.

Given a set of variables, a Bayesian networks is a directed acyclic graph containing one node for each variable. The directed edges of the graph describe influences between variables: an edge (Y, X) implies that Y influences X. Each influence (Y, X) is associated with a conditional probability $\Pr[X \mid Y]$. In the simplest case, the conditional distribution is represented as a **Conditional Probability Table (CPT)** giving the distribution over X for each combination of its parent values.

For instance, consider the following situation. Suppose, that while we're working, our neighbor John calls to say that our alarm is ringing, but our other neighbor Mary doesn't call. However, we know that the alarm can be set off by minor earthquakes. Can we use probabilities to decide if there is a burglar?

First, we define five variables: Burglar, Earthquake, Alarm, JohnCalls, MaryCalls. Then, we consider the relations between such variables based on what we know:

- A burglar can set the alarm, hence Burglar $\in$ Parent(Alarm)
- An earthquake can set the alarm, hence Earthquake $\in$ Parent(Alarm)
- The alarm can cause John to call, hence Alarm $\in$ Parent(JohnCalls)
- The alarm can cause Mary to call, hence Alarm $\in$ Parent(MaryCalls)

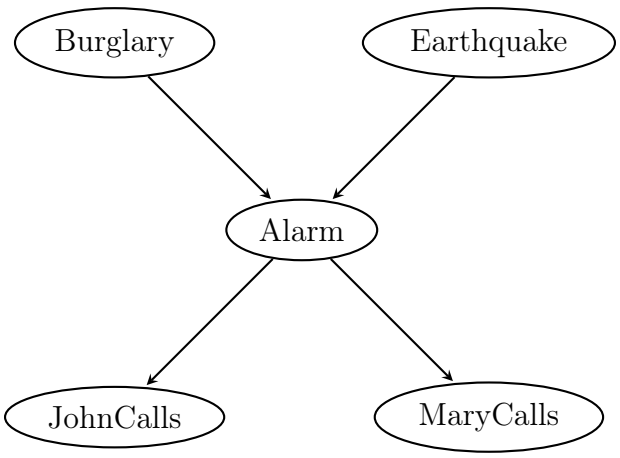

Figure 3.3: The Bayesian network representing the *BurlaryProblem*

For each variable, we construct a CPT representing the conditional probabilities for each of the parent attributes.

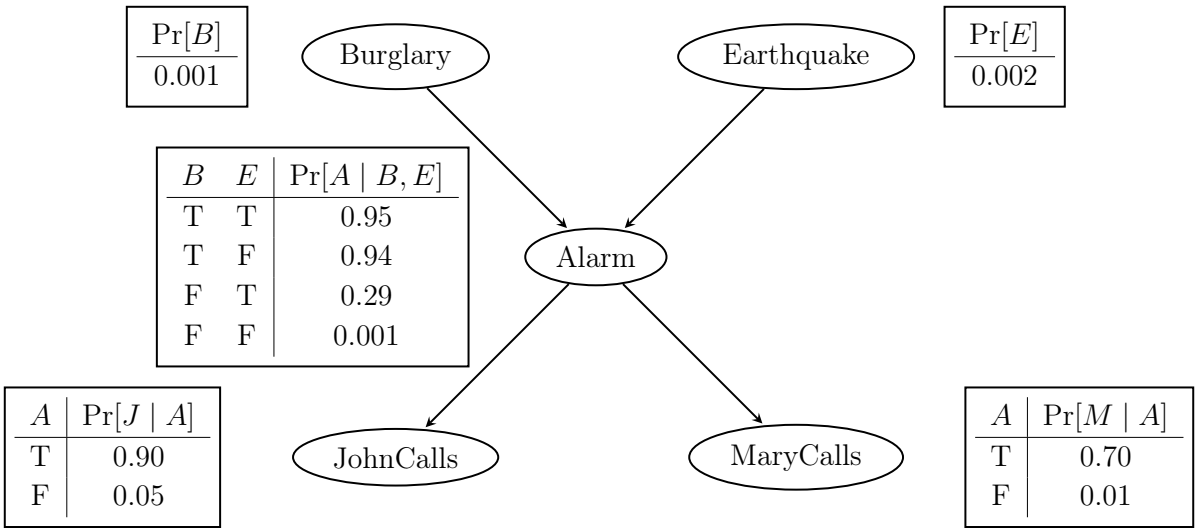

Figure 3.4: The Bayesian network representing the *BurlaryProblem*

Using the chain rule, the probability that both Mary and John call when the alarm rings while there is no burlar or earthquake is given by:

$$\Pr[\neg \mathrm{B}, \neg \mathrm{E}, \mathrm{A}, \mathrm{J}, \mathrm{M}] = \Pr[\neg B] \cdot \Pr[\neg E] \cdot \Pr[A \mid \neg B, \neg E] \cdot \Pr[J \mid A] \cdot \Pr[M \mid A] \approx 0.00063$$

3.2 Bayesian learning

3.2.1 Optimal Bayes classifier

In classification problems, Bayesian learning provides practical learning algorithms based on multiple parameters, combining prior knowledge (in particular prior probabilities) with new observed data, making probabilistic decisions.

Given a dataset and an instance $x \in X$, the latter gets classified with the most probable class for that instance and that dataset. In other words, given a classification problem $f : X \to Y$, a dataset D and a new instance $x \in X$, the best class prediction $C^* \in Y$ for x over D is given by:

$$C^* \in \arg\max_{C \in Y} \Pr[C \mid x, D]$$

Similarly, learning is also achieved through probability. Given dataset D and hypothesis space H, we're interested in the hypothesis $h^* \in H$ that maximizes the probability for the given dataset.

$$h^* \in \arg\max_{h \in H} \Pr[h \mid D]$$

As the name suggests, Bayesian learning is based on Bayes' rule. In general, we're interested in two types of hypotheses: the *maximum a posteriori* hypothesis and the *maximum*

likelihood hypothesis. The former is achieved by applying Bayes' rule, inverting the order of the events:

$$h_{\text{MAP}} \in \arg\max_{h \in H} \Pr[h \mid D] = \arg\max_{h \in H} \Pr[D \mid h] \cdot \Pr[h]$$

The latter, instead, also assumes that the hypotheses are **uniformly distributed**, i.e. that they all have the same probability. In this case, since $\Pr[h_i] = \Pr_[\, h_j]$ for all $h_i, h_j \in H$, we can ignore this probability:

$$h_{\text{ML}} \in \arg\max_{h \in H} \Pr[h \mid D] = \arg\max_{h \in H} \Pr[D \mid h] \cdot \Pr[h] = \arg\max_{h \in H} \Pr[D \mid h]$$

Definition 14: MAP and ML hypotheses

Given a classification problem $f : X \to Y$ and a dataset D, the **maximum a posteriori (MAP)** hypothesis h_{MAP} and the **maximum likelihood (ML)** hypothesis h_{ML} are defined as:

$$h_{\text{MAP}} \in \arg\max_{h \in H} \Pr[D \mid h] \cdot \Pr[h]$$

$$h_{\text{ML}} \in \arg\max_{h \in H} \Pr[D \mid h]$$

By definition, h^*_{MAP} and h^*_{ML} are two types of optimal hypotheses for a *given* dataset D. Hence, for any instance $x \in X$, the classes $h^*_{\text{MAP}}(x)$ and $h^*_{\text{MAP}}(x)$ may not be the most probable classification for x in *general*, meaning that we don't care about the specific dataset. For instance, suppose we have three hypotheses h_1, h_2, h_3 for a dataset D, where:

$$\Pr[h_1 \mid D] = 0.4 \qquad \Pr[h_2 \mid D] = 0.3 \qquad \Pr[h_3 \mid D] = 0.3$$

Given an instance $x \in X$, suppose that:

$$h_1(x) = \oplus \qquad h_2(x) = \ominus \qquad h_3(x) = \ominus$$

Here, for both MAP and ML hypotheses, the most probable hypothesis is clearly h_1. Hence, we would classify x as type $\oplus$. However, the most probable class for x in general is clearly $\ominus$. To fix this issue, we can use the **Optimal Bayes Classifier** method, which is based on total probability. Given an instance $x \in X$ and a class $C \in Y$ with a dataset D, we have that:

$$\Pr[C \mid x, D] = \sum_{h \in H} \Pr[C \mid x, h, D] \cdot \Pr[h \mid x, D] = \sum_{h \in H} \Pr[C \mid x, h] \cdot \Pr[h \mid D]$$

The last equivalence is given by the fact that:

- When h is fixed, the probability that $h(x) = C$ is independent from D, hence $\Pr[C \mid x, h, D] = \Pr[C \mid x, h]$
- h is always independent from x, hence $\Pr[h \mid x, D] = \Pr[h \mid D]$

Definition 15: Optimal Bayes Classifier

Given a classification problem $f : X \to Y$, a dataset D and a new instance $x \in X - X_D$, the Optimal Bayes Classification for x over D is given by:

$$C_{\text{OB}} \in \arg\max_{C \in Y} \sum_{h \in H} \Pr[C \mid x, h] \cdot \Pr[h \mid D]$$

As the name suggests, this method is an **optimal learner**, meaning that no other classification method using the same hypothesis space and same prior knowledge can outperform this method on average. This method maximizes the probability that the new instance x is classified correctly. For instance, consider again the previous example. We have that:

$$\begin{array}{lll} \Pr[h_1 \mid D] = 0.4 & \Pr[h_2 \mid D] = 0.3 & \Pr[h_3 \mid D] = 0.3 \\ \Pr[\oplus \mid x, h_1] = 1 & \Pr[\oplus \mid x, h_2] = 0 & \Pr[\oplus \mid x, h_3] = 0 \\ \Pr[\ominus \mid x, h_1] = 0 & \Pr[\ominus \mid x, h_2] = 1 & \Pr[\ominus \mid x, h_3] = 1 \end{array}$$

We notice that:

$$\sum_{h_i \in H} \Pr[\oplus \mid x, h_i] \cdot \Pr[h_i \mid D] = 0.4 \qquad \sum_{h_i \in H} \Pr[\ominus \mid x, h_i] \cdot \Pr[h_i \mid D] = 0.6$$

Hence, the Optimal Bayes Classifier would correctly label the instance x with $\ominus$. To get the full picture behind the power of this optimal learner, we consider a more complex example.

Suppose that we have four kinds of candy bags:

1. The first type contains only candies with the cherry flavor
2. The second type contains 75% of cherry candies and 25% of lime candies
3. The third type contains 50% of cherry candies and 50% of lime candies
4. The fourth type contains 25% of cherry candies and 75% of lime candies
5. The fifth type contains only candies with the lime flavor

A box containing bags of candies arrives to our shop. We know that:

1. 10% of the bags are of the first flavor
2. 20% of the bags are of the second flavor
3. 40% of the bags are of the third flavor
4. 20% of the bags are of the fourth flavor
5. 10% of the bags are of the fifth flavor

We choose a random bag from the box – without knowing its type – and extract some candies from it. We want to learn which type of bag we picked and what is the probability of extracting a candy of a specific flavor next.

First, we model our distribution prior probabilities for the hypothesis space. Each type of bag corresponds to a hypothesis, meaning that:

$$\Pr[H] = \left\langle \frac{1}{10},\ \frac{1}{5},\ \frac{2}{5},\ \frac{1}{5},\ \frac{1}{10} \right\rangle$$

The distribution for extracting a lime candy for each hypothesis is given by:

$$\Pr[\ell \mid H] = \left\langle 0,\ \frac{1}{4},\ \frac{1}{2},\ \frac{3}{4},\ 1 \right\rangle$$

Since we haven't extracted any candy, our previous knowledge is none – hence, the dataset is empty. The probability of the first candy having lime flavor in this case is given by:

$$\Pr[\ell \mid x_1] = \sum_{h_i \in H} \Pr[\ell \mid x, h_i] \cdot \Pr[h_i] = 0 \cdot \frac{1}{10} + \frac{1}{4} \cdot \frac{1}{5} + \frac{1}{2} \cdot \frac{2}{5} + \frac{3}{4} \cdot \frac{1}{5} + 1 \cdot \frac{1}{10} = \frac{1}{2}$$

Suppose that the first candy is indeed a lime candy. For the following extraction, our dataset will be $D_1 = \{(x_1, \ell)\}$. Through Bayes' rule, we have that:

$$\Pr[H \mid D_1] = \frac{\Pr[D_1 \mid H] \cdot \Pr[H]}{\Pr[D_1]} = 2 \Pr[D_1 \mid H] \cdot \Pr[H] = 2 \Pr[\ell \mid H] \cdot \Pr[H]$$

Hence, we have that:

$$\Pr[H \mid D_1] = 2 \cdot \left\langle 0 \cdot \frac{1}{10},\ \frac{1}{4} \cdot \frac{1}{5},\ \frac{1}{2} \cdot \frac{2}{5},\ \frac{3}{4} \cdot \frac{1}{5},\ 1 \cdot \frac{1}{10} \right\rangle = \left\langle 0,\ \frac{1}{10},\ \frac{2}{5},\ \frac{3}{10},\ \frac{1}{5} \right\rangle$$

The probability of extracting a second lime candy is given by:

$$\Pr[\ell \mid x_2, D_1] = \sum_{h_i \in H} \Pr[\ell \mid x_2, h_i] \cdot \Pr[h_i \mid D_1] = 0 \cdot 0 + \frac{1}{4} \cdot \frac{1}{10} + \frac{1}{2} \cdot \frac{2}{5} + \frac{3}{4} \cdot \frac{3}{10} + 1 \cdot \frac{1}{5} = \frac{13}{20}$$

Suppose that we extract another lime candy. The dataset is now $D_2 = \{(x_2, \ell), (x_1, \ell)\}$.

$$\Pr[H \mid D_2] = \frac{\Pr[D_2 \mid H] \cdot \Pr[H]}{\Pr[D_2]} = \frac{20}{13} \Pr[D_2 \mid H] \cdot \Pr[H]$$

Since the data samples in D_2 are independent from each other, we have that:

$$\Pr[H \mid D_2] = \frac{20}{13} \Pr[\{(x_2, \ell)\} \mid H] \cdot \Pr[\{(x_1, \ell)\} \mid H] \cdot \Pr[H] = \frac{20}{13} \Pr[\ell \mid H] \cdot \Pr[H \mid D_1]$$

Again, we now have that:

$$\Pr[H \mid D_2] = \frac{20}{13} \cdot \left\langle 0 \cdot 0,\ \frac{1}{4} \cdot \frac{1}{10},\ \frac{1}{2} \cdot \frac{2}{5},\ \frac{3}{4} \cdot \frac{3}{10},\ 1 \cdot \frac{1}{5} \right\rangle = \left\langle 0,\ \frac{1}{26},\ \frac{4}{13},\ \frac{9}{26},\ \frac{4}{13} \right\rangle$$

If we keep extracting a lime candy, the posterior probability of the fifth hypothesis will skyrocket. Using the MAP hypothesis, the fifth one will be selected. However, the selected hypothesis may not be the correct one: the fifth hypothesis clearly overfits our dataset.

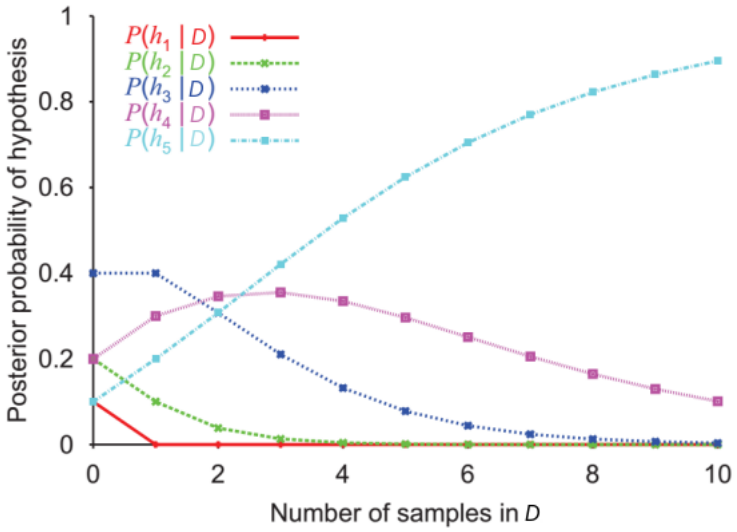

Figure 3.5: Probability of the five hypotheses with the increase in the number of samples

Consider now a new manufacturer producing bags with an arbitrary choice of cherry/lime candies. Let $\theta \in [0,1]$ be the ratio of cherry candies over all the N candies, i.e. $\theta = \frac{\text{num. of cherry candies}}{N}$. For each possible θ, we consider the hypothesis h_θ, where:

$$\Pr[c \mid h_\theta] = \theta \qquad \Pr[\ell \mid h_\theta] = 1 - \theta$$

This implies that we have a continuous hypothesis space. The dataset is given by the number of cherry samples c and lime samples ℓ, where $N = c + \ell$. We want to find the ML hypothesis of this setup.

$$h^*_{\text{ML}} \in \arg\max_{\theta \in [0,1]} \Pr[D \mid h_\theta]$$

We notice that:

$$\Pr[D \mid h_\theta] = \prod_{i=1}^{N} \Pr[d_i \mid h_\theta] = \theta^c \cdot (1-\theta)^\ell$$

where $d_i \in D$. Hence, we get that:

$$h_{\theta_{\text{ML}}} \in \arg\max_{\theta \in [0,1]} \Pr[D \mid h_\theta] = \theta^c \cdot (1-\theta)^\ell$$

To get an even better result, we can use the properties of logarithms:

$$h_{\text{ML}} \in \arg\max_{\theta \in [0,1]} \theta^c \cdot (1-\theta)^\ell = \arg\max_{\theta \in [0,1]} \log(\theta^c \cdot (1-\theta)^\ell) = \arg\max_{\theta \in [0,1]} c \log\theta + \ell \log(1-\theta)$$

Since we're working with the continuous interval $[0, 1]$, to find the value $\theta^* \in [0, 1]$ that maximizes the probability we can use derivatives:

$$\frac{dL(D \mid h_\theta)}{d\theta} = \frac{c}{\theta} - \frac{\ell}{1-\theta} \implies \frac{c}{\theta} - \frac{\ell}{1-\theta} = 0 \implies \theta_{ML} = \frac{c}{c+\ell} = \frac{c}{N}$$

concluding that the value θ_{ML} which gives the best hypothesis h_{ML} is the one representing the ratio of cherry candies in the dataset – as anyone would expect.

In general, given a dataset D where for each $d_i \in D$ we have that $d_i \in \{0, 1\}$, assuming a probability distribution for D over an interval Θ, the maximum likelihood estimation is given by:

$$\theta_{ML} \in \arg\max_{\theta \in \Theta} \log \Pr[d_i \mid h_\theta]$$

In the particular case of Bernoulli distributions, i.e. where the variable X represents the number of extractions of positive type over N total extractions and for any $k \in [0, N]$ it holds that $\Pr[X = k] = \theta^k (1-\theta)^{N-k}$, we always get that:

$$\theta_{\text{ML}} = \frac{|\{d_i \in D \mid d_i = 1\}|}{|D|}$$

3.2.2 Naïve Bayes classifier

The Bayes Optimal Classifier discussed in the previous section provides the best result, but it's not practical when the hypothesis space is large due to the necessity of enumerating all hypotheses. Instead, we usually work with a weaker version of this classifier, the **Naïve Bayes classifier**.

Consider a target function $f : X \to Y$. Each instance $x \in X = A_1 \times \ldots \times A_n$ is described by a sequence of values $a_1, a_2, \ldots, a_n$, hence:

$$\begin{aligned}\arg\max_{C \in Y} \Pr[C \mid x, D] &= \arg\max_{C \in Y} \Pr[C \mid a_1, \ldots, a_n, D] \\ &= \arg\max_{C \in Y} \Pr[a_1, \ldots, a_n \mid C, D] \cdot \Pr[C \mid D]\end{aligned}$$

If we assume that the values of $a_1, \ldots, a_n$ are independent from each other, we get that:

$$\arg\max_{C \in Y} \Pr[a_1, \ldots, a_n \mid C, D] \cdot \Pr[C \mid D] = \arg\max_{C \in Y} \Pr[C \mid D] \prod_{a_i \in x} \Pr[a_1 \mid C, D]$$

We observe that the assumption $\Pr[a_1, \ldots, a_n \mid C, D] \approx \prod_{a_i \in x} \Pr[a_i \mid C, D]$ is often false. However, it still works surprisingly well, enabling us to estimate the probability without enumerating the whole hypothesis space.

Definition 16: Naïve Bayes Classifier

Given a classification problem $f : X \to Y$, a dataset D and a new instance $x \in X - X_D$, the Naïve Bayes Classification for x over D is given by:

$$C_{\text{NB}} \in \arg\max_{C \in Y} \Pr[C \mid D] \prod_{a_i \in x} \Pr[a_i \mid C, D]$$

To estimate $\Pr[C \mid D]$, we consider the ratio of positive instances, i.e. instances being classified as class C, over all the instances in the dataset.

$$\widehat{\Pr}[C \mid D] = \frac{|(a, b) \in D \mid b = C|}{|D|}$$

Similarly, to estimate $\Pr[a_i \mid C, D]$ we consider the ratio of positive instances with the value a_i over all the positive instances in the dataset.

$$\widehat{\Pr}[a_i \mid C, D] = \frac{|(a, b) \in D \mid b = C \text{ and } a_i \text{ is an attr. of } a|}{|(a, b) \in D \mid b = C|}$$

Algorithm 5: Naïve Bayes Classifier

Given a dataset D for a classification problem $f : X \to Y$ and a dataset D, where $X = A_1 \times \ldots \times A_k$, the algorithm learns the dataset using the Naïve Bayes Classifier and classifies new instances $x \in X - X_D$.

function NaïveBayesLearn(X, Y, D)
  for $C \in Y$ **do**
    Estimate $\Pr[C \mid D]$ by computing $\widehat{\Pr}[C \mid D]$
    for $A_k \in X$ **do**
      for $a_i \in A_k$ **do**
        Estimate $\Pr[a_i \mid C, D]$ by computing $\widehat{\Pr}[a_i \mid C, D]$
      end for
    end for
  end for
end function
function ClassifyNewInstance(x)
  $C_{\text{NB}} \in \arg\max_{C \in Y} \widehat{\Pr}[C \mid D] \prod_{a_i \in x} \widehat{\Pr}[a_i \mid C, D]$
end function

We notice that this estimate has a flaw: if none of the training instances in the dataset with target class C have attribute value a_i then $\widehat{\Pr}[a_i \mid C, D] = 0$, meaning that $\widehat{\Pr}[C \mid D] \prod_{a_i \in x} \widehat{\Pr}[a_i \mid C, D] = 0$, meaning that no new instance can get classified as C. To fix this issue, we use a small additional value m acting as "virtual" instances and the value p, the previous estimate of $\Pr[a_i \mid C, D]$.

$$\widehat{\Pr}[a_i \mid C, D] = \frac{mp + |(a, b) \in D \mid b = C \text{ and } a_i \text{ is an attr. of } a|}{m + |(a, b) \in D \mid b = C|}$$

We also notice that, since we're interested in the solution maximizing the probability, these approximations don't have to be accurate. In fact, they must only try to preserve which solution will be chose. For instance, the previous estimations preserve the maximum-likelihood solution.

Surprisingly, the Naïve Bayes Classifier can also be adapted to learn how to classify text, i.e. identify them as spam, e-mail, web reviews, etc. Consider the target function $f : \text{Docs} \to Y$. Any document $d \in \text{Docs}$ is described by a sequence of words $d = w_1 w_2 \dots w_k$. Using the classifier we have that:

$$\Pr[d \mid C, D] = \prod_{w_i \in d} \Pr[w_i \mid C, D]$$

where $\Pr[w_i \mid C, D]$ is the probability of word w_i occurring in a document of class C in D. Let $V = \{w_1, \dots, w_n\}$ be the set of all words appearing in any document $d \in D$. This is referred to as the *vocabulary* of D. After fixing an arbitrary order on V, each document $d \in D$ can be represented by an n-dimensional feature vector over V, where each d_i is an encoding of word w_i in the document, which can be achieved through various methods:

1. **Boolean feature vector**: $d_i = 1$ if w_i occurs in the document, otherwise $d_i = 0$.
2. **Ordinal feature vector**: $d_i = k$ if w_i occurs k times in the document
3. **Real-valued feature vector (tf-idf)**: $d_i = \text{tf}(w_i, \text{doc}) \cdot \text{idf}(w_i, D)$, where $\text{tf}(w_i, \text{doc})$ is the frequency of the term w_i in D and $\text{idf}(w_i, \text{doc})$ is the inverse document frequency of w_i in D.

In the first case, we're working with multivariate Bernoulli variables. Hence,we have that:

$$\Pr[d \mid C, D] = \prod_{i=1}^{n} \Pr[w_i \mid C, D]^{d_i} \cdot (1 - \Pr[w_i \mid C, D])^{1-d_i}$$

For the maximum-likelihood solution, the used estimate of the probability is:

$$\widehat{\Pr}[w_i \mid C, D] = \frac{t_{i,C} + 1}{t_C + 2}$$

where $t_{i,C}$ is the number of documents in D of class C containing w_i and t_C is the number of documents in D of class C. In the second case, instead, we're working with multinomial variables, implying that:

$$\Pr[d \mid C, D] = \frac{n!}{d_1! \cdot \ldots \cdot d_n!} \prod_{i=1}^{n} \Pr[w_i \mid C, D]^{d_i}$$

For the maximum-likelihood solution, the used estimate of the probability is:

$$\widehat{\Pr}[w_i \mid C, D] = \sum_{d \in D} \frac{tf_{i,C} + \alpha}{tf_C + \alpha \, |V|}$$

where $tf_{i,C}$ is the term frequency of w_i in a document of class C, tf_C is the frequency of all terms in a document of class C and α is a smoothing parameter ($\alpha = 1$ for Laplace smoothing)

This type of text classification has some evident issues. In fact, each of the three methods of representation looses context information – for instance, the order in which the words occur in the document is important. To improve this model, we can eliminate *stop words* ("the", "a", etc.), apply *stemming* by replacing words with their basic form ("likes", "liking" $\rightarrow$ "like") or use *n-grams*, where tokens (sequences of words) are used.

3.3 Probabilistic models for classification

3.3.1 Probabilistic generative models

Consider a classification problem $f : \mathbb{R}^d \rightarrow Y$ and a dataset D, we want to estimate $\Pr[C \mid x, D]$ for a new instance $x \in \mathbb{R}^d - \mathbb{R}^d_D$ and a class $C \in Y$. In **probabilistic generative models**, this estimation is achieved through Bayes' rule, like we did in the previous section for Bayes classifiers. The more general approach for these models is to reduce these probabilities to the computation of a variant of the **sigmoid function**, defined as:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

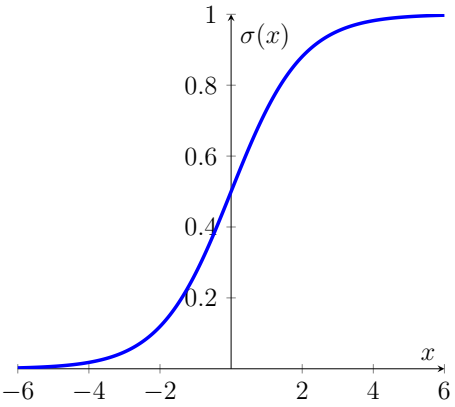

Figure 3.6: The sigmoid function

Assuming that $Y = \{C_1, C_2\}$, we can manipulate Bayes' rule in the following way:

$$\begin{aligned}
\Pr[C_1 \mid x, D] &= \frac{\Pr[x \mid C_1, D] \cdot \Pr[C_1 \mid D]}{\Pr[x \mid D]} \\
&= \frac{\Pr[x \mid C_1, D] \cdot \Pr[C_1 \mid D]}{\Pr[x \mid C_1, D] \cdot \Pr[C_1 \mid D] + \Pr[x \mid C_2, D] \cdot \Pr[C_2 \mid D]} \\
&= \frac{1}{1 + \dfrac{\Pr[x \mid C_2, D] \cdot \Pr[C_2 \mid D]}{\Pr[x \mid C_1, D] \cdot \Pr[C_1 \mid D]}}
\end{aligned}$$

By setting $\alpha = \ln \dfrac{\Pr[x \mid C_1, D] \cdot \Pr[C_1 \mid D]}{\Pr[x \mid C_2, D] \cdot \Pr[C_2 \mid D]}$, we get the sigmoid function $\sigma(\alpha)$.

Consider now the parametric model where $\Pr[x \mid C_i, D]$ has the normal distribution $\mathcal{N}(x; \mu_i, \Sigma)$, where μ_i is the mean and Σ is a covariance matrix. We have that:

$$\alpha = \ln \frac{\Pr[x \mid C_1, D] \cdot \Pr[C_1 \mid D]}{\Pr[x \mid C_2, D] \cdot \Pr[C_2 \mid D]} = \ln \frac{\mathcal{N}(x; \mu_1, \Sigma) \cdot \Pr[C_1 \mid D]}{\mathcal{N}(x; \mu_2, \Sigma) \cdot \Pr[C_2 \mid D]} = w^T x + w_0$$

where:

$$w = \Sigma^{-1}(\mu_1 - \mu_2) \qquad w_0 = -\frac{1}{2}\mu_1^T \Sigma^{-1} \mu_1 + \frac{1}{2}\mu_2^T \Sigma^{-1} \mu_2 + \frac{\Pr[C_1 \mid D]}{\Pr[C_2 \mid D]}$$

where $\Pr[C_1 \mid D] = p$ and $\Pr[C_2 \mid D] = 1 - p$. To summarize, we have that:

$$\Pr[C_1 \mid x, D] = \sigma(w^T x - w_0) \qquad \Pr[C_2 \mid x, D] = 1 - \sigma(w^T x - w_0)$$

$$\Pr[x \mid C_1, D] = \mathcal{N}(x; \mu_1, \Sigma) \qquad \Pr[x \mid C_2, D] = \mathcal{N}(x; \mu_2, \Sigma)$$

$$\Pr[C_1 \mid D] = p \qquad \Pr[C_2 \mid D] = 1 - p$$

This allows us to easily automatize the computation of such probability since we can skip the computation of the logarithm. However, to achieve this we still have to estimate the values of μ_1, μ_2, p and Σ. Given data set $D = \{(x_n, t_n) \mid i \in [N]\}$, where $t_i = 1$ if x_i is in class C_1 and $t_i = 0$ otherwise. Let N_1 be the number of samples in D belonging to C_1 and N_2 be the number of samples in C_2 ($N = N_1 + N_2$). The maximum-likelihood solution here is given by:

$$\Pr[t \mid p, \mu_1, \mu_2, \Sigma] = \prod_{i=1}^{N} (p\, \mathcal{N}(x_i; \mu_1, \Sigma))^{t_i} ((1-p)\, \mathcal{N}(x_i; \mu_2, \Sigma))^{1-t_i}$$

After maximizing the logarithmic likelihood, we always obtain that:

$$\widehat{\mu}_1 = \frac{1}{N_1} \sum_{i=1}^{N} t_i x_i \qquad \widehat{\mu}_2 = \frac{1}{N_2} \sum_{i=1}^{N} (1 - t_i) x_i$$

$$\widehat{p} = \frac{N_1}{N} \qquad \widehat{\Sigma} = \frac{N_1}{N} S_1 + \frac{N_2}{N} S_2$$

where $S_j = \frac{1}{N_j} \sum_{i \in C_j} (x_i - \widehat{\mu}_j)(x_i - \widehat{\mu}_j)^T$.

More generally, when $Y = \{C_1, \ldots, C_k\}$, the dataset is formalized as $D = \{(x_i, t_i) \mid i \in [N]\}$, where t_i is a one-hot indicator vector with $t_{i,j} = 1$ if and only if x_i is in class C_j. In this case, the estimate is given by:

$$\Pr[C_j \mid x] = \frac{\Pr[x \mid C_j, D] \cdot \Pr[C_j, D]}{\sum_{h=1}^{k} \Pr[x \mid C_h, D] \cdot \Pr[C_h \mid D]} = \frac{e^{\alpha_j}}{\sum_{h=1}^{k} e^{\alpha_h}}$$

where $\alpha_h = \ln \Pr[x \mid C_h] \Pr[C_h]$. This model is called **Gaussian Naïve Bayes classification**, where $\forall j \in [k]$ we have:

$$\widehat{\Pr}[C_j \mid D] = \widehat{p}_j = \frac{N_j}{N} \qquad \Pr[x \mid C_j, D] = \mathcal{N}(x; \mu_j, \Sigma)$$

$$\widehat{\mu}_j = \frac{1}{N_j} \sum_{i=1}^{N} t_{i,j} x_i \qquad \widehat{\Sigma} = \textstyle\sum_{j=1}^{k} \frac{N_j}{N} S_j$$

where $S_j = \frac{1}{N_j} \sum_{i \in C_j} (x_i - \widehat{\mu}_j)(x_i - \widehat{\mu}_j)^T$. The use of matrix-like computation can be further extended by also modeling other elements as matrices. For instance, given a problem $f : \mathbb{R}^d \to Y$, a dataset $D = \{(x_i, t_i) \mid i \in [N]\}$ can also be represented as $\langle X, t \rangle$, where X is an $N \times d$ matrix of input values and t is a vector of N output values.

To make notation more compact, we set $\widetilde{w} = \begin{bmatrix} w_0 \\ w \end{bmatrix}$ and $\widetilde{x} = \begin{bmatrix} 1 \\ x \end{bmatrix}$ in order to get:

$$a_h = w^T x + w_0 = \widetilde{w}^T \widetilde{x}$$

The maximum-likelihood solution for a parametric model M_Θ of dataset $D = \langle X, t \rangle$ with $\Pr[t \mid \Theta, X]$ is thus given by:

$$\Theta^* \in \arg\max_{\Theta} \ln \Pr[t \mid \Theta, X]$$

When M_Θ belongs to the exponential family, the probability $\Pr[t \mid \theta, X]$ can be expressed in the form $\Pr[t \mid \widetilde{w}, X]$

$$\Theta^* \in \arg\max_{\widetilde{w}} \ln \Pr[t \mid \widetilde{w}, X]$$

3.3.2 Probabilistic discriminative models

Consider a classification problem $f : \mathbb{R}^d \to Y$ where $X \subseteq \mathbb{R}^d$ and a dataset $D = \langle X, t \rangle = \{(x_i, t_i) \mid i \in [N]\}$, with $t_i \in \{0, 1\}$. Differently from generative models, in **probabilistic discriminative models**, to estimate $\Pr[C \mid x, D]$ for a new instance $x \in X - X_D$ and a class $C \in Y$, we directly estimate the probability:

$$\Pr[C_j \mid \widetilde{x}, D] = \frac{e^{\alpha_j}}{\sum_{h=1}^{k} e^{\alpha_h}}$$

without using Bayes' rule, where $\alpha_h = \widetilde{w}^T \widetilde{x}$. In this case, the maximum-likelihood is still given by:

$$\Theta^* \in \arg\max_{\widetilde{w}} \ln \Pr[t \mid \widetilde{w}, X]$$

The typical example of discriminative model is **logistic regression**. Suppose that we are working with only two classes. The probability $\Pr[t \mid \widetilde{w}, X]$ – recall that $D = \langle X, t \rangle$ – is given by:

$$\Pr[t \mid \widetilde{w}, X] = \prod_{i=1}^{N} p_i^{t_i} (1 - p_i)^{1 - t_i}$$

where $p_i = \Pr[C_1 \mid \widetilde{x_i}] = \sigma(\widetilde{w}^T \widetilde{x_i})$. By applying the negative logarithm, we obtain a new type of error function: the **cross-entropy error function**.

Definition 17: Cross-entropy error function

Given a classification problem $f : \mathbb{R}^d \to Y$ and a dataset $D = \langle X, t \rangle$, the **cross-entropy** (or *negative log likelihood*) error function is defined as:

$$E(\widetilde{w}) = -\ln \Pr[t \mid \widetilde{w}, X] = -\sum_{i=1}^{N} t_i \ln p_i + (1 - t_i) \ln(1 - p_i)$$

Logistic regression is based on solving the following optimization problem:

$$\widetilde{w}^* \in \arg\min_{\widetilde{w}} E(\widetilde{w})$$

There are many efficient ways to solve this problem. The most common one is the **iteratively reweighted least squares (IRLS)** method, based on *Newton-Raphson iterative method*, which uses derivatives to get closer and closer to the optimal solution. In particular, since we're working with higher dimensions, we'll be using the concept of gradient and gradient descent.

1. First, compute the gradient of the error with respect to $\widetilde{w}$:

$$\nabla E(\widetilde{w}) = \sum_{i=1}^{N} (p_i - t_i)\widetilde{x_i}$$

2. Apply the gradient descent step by setting

$$\widetilde{w} := \widetilde{w} - H(\widetilde{w})^{-1} \nabla E(\widetilde{w})$$

   where $H(\widetilde{w})$ is the Hessian matrix of $E(\widetilde{w})$, i.e. $H(\widetilde{w}) = \nabla\nabla E(\widetilde{w})$.

3. Repeat until a certain improvement treshold is reached or after a fixed number of iterations

Observation 1: Gradient

For those who aren't confident with multivariate calculus, the gradient of a function $f : \mathbb{R}^n \to \mathbb{R}^m$ is defined as:

$$\nabla f(x) = \begin{bmatrix} \frac{\partial f}{\partial x_1}(x) \\ \vdots \\ \frac{\partial f}{\partial x_n}(x) \end{bmatrix}$$

In other words, the gradient of f is the vector whose components are the partial derivatives of f with respect to the components of the input

These computation can also be made more compact and immediate – thus the strength of this model:

$$\nabla E(\widetilde{w}) = \widetilde{X}^T(p(\widetilde{w}) - t)$$
$$H(\widetilde{w}) = \widetilde{X}^T R(\widetilde{w})\widetilde{X}$$

where $p(\widetilde{w}) = \begin{bmatrix} p_1 & \cdots & p_N \end{bmatrix}^T$, $\widetilde{X} = \begin{bmatrix} \widetilde{x_1} & \cdots & \widetilde{x_N} \end{bmatrix}$ and $R(\widetilde{w})$ is the diagonal matrix where $r_{i,i} = p_i(1 - p_i)$.

Algorithm 6: Iterative reweighted least squares (IRLS)

Given a classification problem $f : \mathbb{R}^d \to Y$, where $Y = C_1, C_2$, and a dataset $D = \langle X, t \rangle$, the algorithm solves the optimization problem $\widetilde{w}^* \in \arg\min_{\widetilde{w}} E(\widetilde{w})$.

function IRLS(Y, X, t)
  $\widetilde{w} = 0$
  do
    $\widetilde{w} = \widetilde{w} - (\widetilde{X}^T R(\widetilde{w})\widetilde{X})^{-1}\widetilde{X}^T(p(\widetilde{w}) - t)$
  while $E(\widetilde{w})$ decreases
end function

4

Linear models

4.1 Linear models for classification

In linear models, we're interested in studying classification problems $f : \mathbb{R}^d \to Y$ with a dataset whose instances are **linearly separable**. A dataset is said to be linearly separable when there exists a hyperplane that separates the instance space into two regions such that differently classified instances lie on different regions.

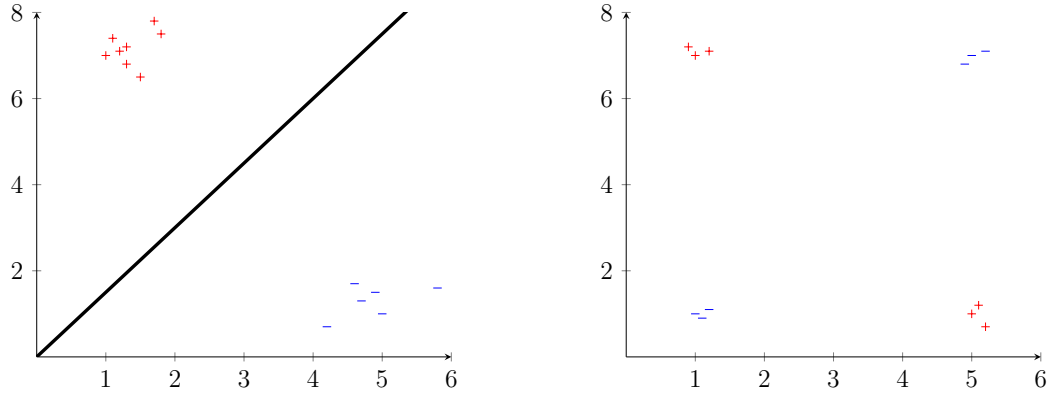

Figure 4.1: A linearly separable dataset (left) and a non-linearly separable one (right)

The functions describing such hyperplanes are called **linear discriminant functions**. Suppose that $Y = \{C_1, \ldots, C_k\}$. We define k linear functions:

$$y_1(x) = w_1^T x + w_{1_0} = \widetilde{w_1}^T \widetilde{x} \qquad \cdots \qquad y_k(x) = w_k^T x + w_{k_0} = \widetilde{w_k}^T \widetilde{x}$$

where $\widetilde{w_h} = \begin{bmatrix} w_{h_0} \\ w_h \end{bmatrix}$ and $\widetilde{x} = \begin{bmatrix} 1 \\ x \end{bmatrix}$. Each instance $x \in \mathbb{R}^d$ gets classified as C_j if the $y_j(x)$ is the one with maximum value.

$$C_j \in \arg\max_{C_h \in Y} y_h(x)$$

In particular, we observe that for each $C_j, C_{j'}$ the hyperplane $(\widetilde{w_j} - \widetilde{w_{j'}})^T \widetilde{x} = 0$ acts as a **decision boundary**, splitting the regions of points classified as either C_j or $C_{j'}$.

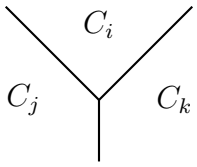

Figure 4.2: Example of how regions get partitioned by a K-class discriminant

Given the following matrix:

$$y(x) = \begin{bmatrix} y_1(x) \\ \vdots \\ y_k(x) \end{bmatrix} = \begin{bmatrix} \widetilde{w_1}^T \\ \vdots \\ \widetilde{w_k}^T \end{bmatrix} \widetilde{x} = \widetilde{W}^T \widetilde{x}$$

where $\widetilde{W} = \begin{bmatrix} \widetilde{w_1} & \cdots & \widetilde{w_k} \end{bmatrix}$, leaning a linear model is equivalent to estimating the matrix $\widetilde{W}$ for which $y(x) = \widetilde{W}^T \widetilde{X}$ defines the K-class discriminant.

In particular, we'll restrict our interest to models working directly on the samples x. All the results that we will discuss also hold if we consider a non-linear transformation of the inputs $\phi(x)$, called **basis functions**. The idea is to use these transformations to convert the feature space into a linearly separable one in order to apply the discussed methods. In particular, decision boundaries will be linear in the feature space $\phi(x)$ and non-linear in the original space x. Clearly, classes that are linearly separable in the feature space $\phi(x)$ may not be as easily separable in the input space x.

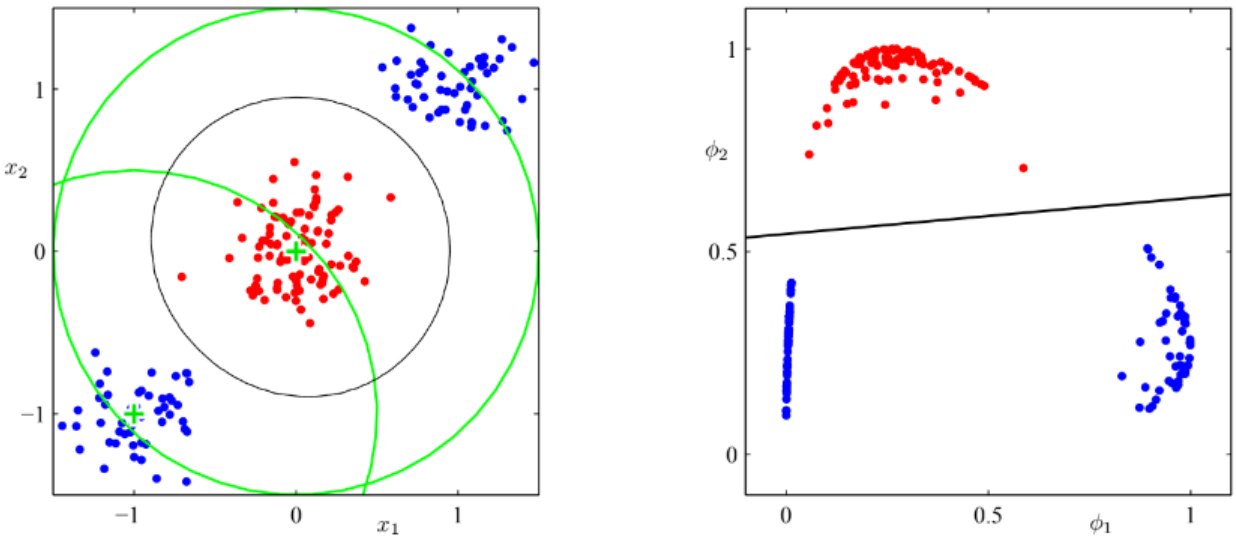

Figure 4.3: Example of a basis function being applied on a non-linearly separable feature space, making the property hold on the basis feature space.

4.1.1 Least squares method

The first method that we'll discuss is the **least squares method**. Consider problem $f : \mathbb{R}^d \to Y$ a dataset $D = \{(x_i, t_i) \mid i \in [N]\}$, where each t is a 1-of-K encoding, meaning that $t_{i,j} = 1$ if $x_i \in C_j$, otherwise $t_{i,j} = 0$. We want to find the linear discriminant $y(x) = \widetilde{W}^T x\widetilde{x}$. Let $\widetilde{X} = \begin{bmatrix}\widetilde{x_1} & \cdots & \widetilde{x_N}\end{bmatrix}^T$ and let $T = \begin{bmatrix}t_1 & \cdots & t_N\end{bmatrix}$. We notice that each row of the product $\widetilde{X}\widetilde{W}$ contains the predicted class for the associated input, while the matrix T contains the real class for the input. When the two matrices are equal, the model is perfect. Hence, we want to minimize the difference between such matrices. To do so, we minimize the **sum-of-squares error function**.

Definition 18: Sum-of-squares error function

Given a linearly separable classification problem $f : \mathbb{R}^d \to Y$ and a dataset $D = \{(x_i, t_i) \mid i \in [N]\}$, the **sum-of-squares error function** is defined as:

$$E(\widetilde{W}) = \frac{1}{2}\operatorname{tr}\left((\widetilde{X}\widetilde{W} - T)^T(\widetilde{X}\widetilde{W} - T)\right)$$

We notice that each entry $a_{i,i}$ on the diagonal of the inner matrix in the above definition is the double of the square of the difference between x_i's predicted class and t_i. Hence, the trace of the is equal to the sum-of-squares. The halving factor is just a convention.

This minimization problem can be actually solved in a closed form. In fact, we have that:

$$\widetilde{W} = (\widetilde{X}^T\widetilde{X})^{-1}\widetilde{X}^T T$$

concluding that:

$$y(X) = \widetilde{W}^T\widetilde{X} = T^T(\widetilde{X}^\dagger)^T X$$

where $\widetilde{X}^\dagger = (\widetilde{X}^T\widetilde{X})^{-1}\widetilde{X}^T$ is the **pseudo-inverse** of $\widetilde{X}$. We observe that this is the first method we have seen where we can actually compute a perfect optimal solution, making this method very easy to use. However, this model still has some problems: the K-class discriminant obtained is not robust to **outliers**, i.e. samples that are very far away from the other samples that are classified with the same class.

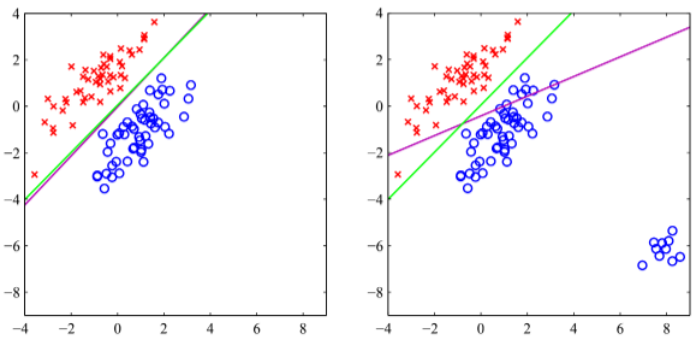

Figure 4.4: Example of how outliers "rotate" the solution

4.1.2 Perceptron

Consider a linear classification problem $f : \mathbb{R}^d \to \{-1, +1\}$. A **perceptron** is a binary classifier that assigns one of two classes to a real-valued input vector. Let $w = \begin{bmatrix} w_0 & w_1 & \cdots & w_d \end{bmatrix}^T$ be a weight vector and let $x \in \mathbb{R}^{d+1}$ where $x_0 = 1$ – from now on we'll drop the $\widetilde{w}, \widetilde{x}$ notation. A perceptron is a function $o(x)$ that can be defined in two ways: **tresholded** or **untresholded**. In the untresholded case, we have that $o(x) = w^T x$, while in the tresholded case we have that $o(x) = \text{sign}(w^T x)$, where: The perceptron $o(x)$ given by w is defined as:

$$\text{sign}(w^T x) = \begin{cases} +1 & \text{if } w_0 + w_1 x_1 + \ldots + w_d x_d > 0 \\ -1 & \text{otherwise} \end{cases}$$

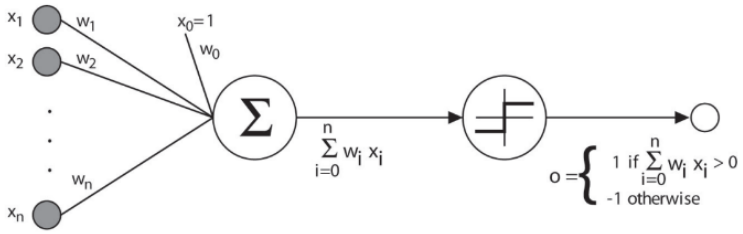

Figure 4.5: A tresholded perceptron

In both cases, to learn w from a dataset $D = \{(x_i, t_i) \mid i \in [N]\}$, we want to minimize the squared error (*loss function*):

$$E(w) = \frac{1}{2} \sum_{i=1}^{N} (t_i - o(x_i))^2$$

Since we're working with real values, the minimization problem can be solved through derivatives and an iterative approach. This is called the **perceptron training rule**.

1. First, compute the derivative of $E(w)$ with respect to each component of w_j of w

$$\frac{\partial E}{\partial w_j} = \sum_{i=1}^{N} (t_i - o(x_i))(-x_{j,i})$$

2. Update each component w_j by setting $w_j := w_j + \Delta w_j$, where:

$$\Delta w_j = -\eta \frac{\partial E}{\partial w_j} = \eta \sum_{i=1}^{N} (t_i - o(x_i)) x_{j,i}$$

   with η being a small constant called *learning rate*

3. Repeat until a certain improvement treshold is reached (or after a fixed number of iterations)

The choice of η is critical: if η is small then each iteration will only slightly move the previous linear discriminant, while if η is large each iteration may significantly move the discriminant.

Moreover, when η is small, the final solution will either be very close to the samples of the first class or very close to the samples of the second class. In other words, the solution cannot lie in the middle of the two sample groups. It can be proven that this procedure will converge if the training data is linearly separable and η is sufficiently small. However, convergence doesn't imply termination, hence the terminating condition.

After learning the optimal w, the predicted class of a new instance $x \in \mathbb{R}^d - \mathbb{R}^d_D$ is – in both treshholded and untresholded perceptrons – given by $\text{sign}(w^T x)$.

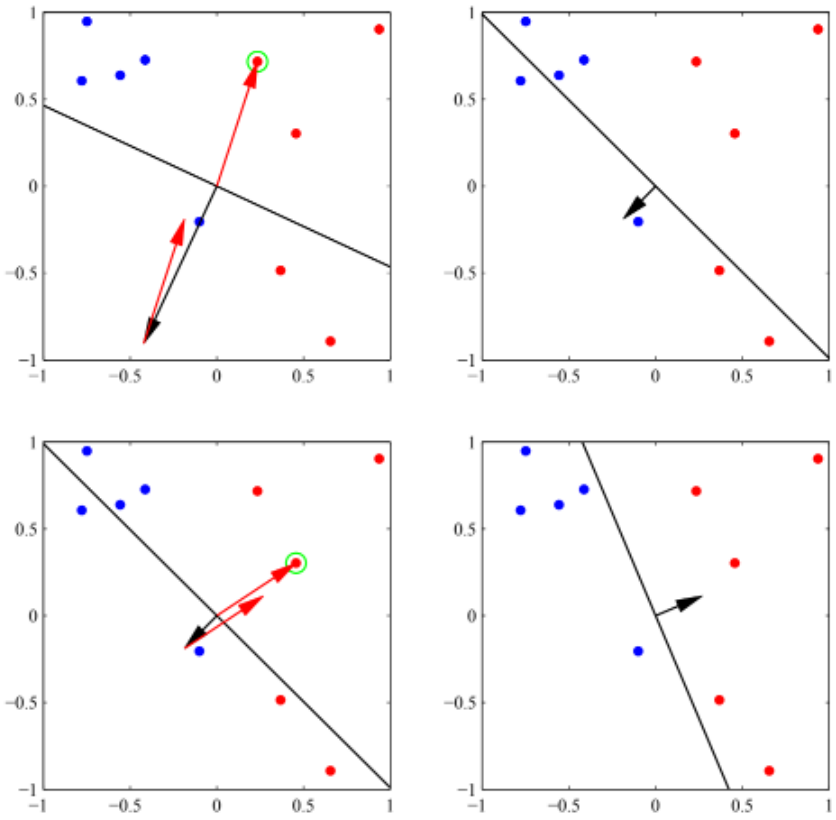

Figure 4.6: Two iterations of the perceptron training rule with a small η value. The vector w is represented by the black arrow, while the vector δw is represented by the red arrow. The green-circled samples circled are misclassified, hence their feature vector is considered for the next iteration.

4.1.3 Fisher's linear discriminant

Consider a linear classification problem $f : \mathbb{R}^d \to \{C_1, C_2\}$. We want to determine a linear discriminant $y = w^T x$ with a vector $w = \begin{bmatrix} w_0 & w_1 & \cdots w_d \end{bmatrix}^T$ for which $x \in C_1$ if $y \geq -w_0$ and $x \in C_2$ otherwise.

Consider a dataset with N_1 points in C_1 and N_2 points in C_2. Let m_1 and m_2 be the average of their distributions:

$$m_1 = \frac{1}{N_1} \sum_{x_i \in C_1} x_1 \qquad\qquad m_2 = \frac{1}{N_2} \sum_{x_i \in C_2} x_i$$

To maximize class separation, we have to adjust the value of w in order to maximize the function $J(w) = w^T(m_2 - m_1)$, subject to $\|w\| = 1$ – implying that $w \propto m_2 - m_1$.

Consider the segment connecting the points represented by m_1 and m_2. Then, the bisector of this segment will always define the maximal linear discriminant w. In fact, if we consider the projections of all the points in the dataset on a line parallel to the segment we get an almost correct separation of the two classes. However, in case of oriented covariance, some samples may get misclassified.

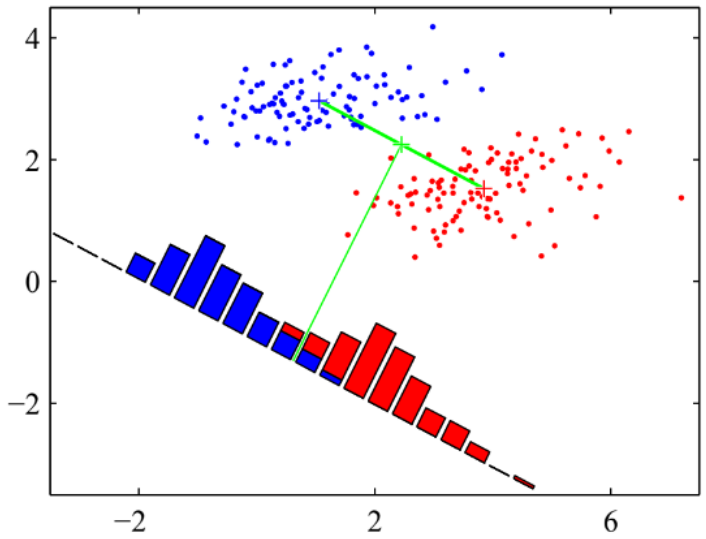

Figure 4.7: The blue and red crosses represent the average points of the two distributions. The green bisector of green segment connecting the two crosses is the almost valid linear discriminant. The colored bars represent the number of samples projected on the dashed line perpendicular to the green segment.

From Figure 4.7, it's easy to see that in order to get a perfect linear discriminant it would suffice to slightly rotate this bisector. To achieve this, **Fisher's discriminant** allows some degree of freedom. In particular, the *Fisher criterion* is defined as the function:

$$J(w) = \frac{w^T S_B w}{w^T S_W w}$$

where S_B, called the *between class scatter*, is defined as:

$$S_B = (m_2 - m_1)(m_2 - m_1)^T$$

and where S_W, called the *within class scatter*, is defined as:

$$S_W = \sum_{x_i \in C_1} (x_i - m_1)(x_i - m_1)^T + \sum_{x_i \in C_2} (x_i - m_2)(x_i - m_2)^T$$

Again, we want to maximize $J(w)$. Using derivatives – hence by solving the equation $\frac{dJ}{dw} = 0$ – we get that the optimal solution w^* is proportional to $S_W^{-1}(m_2 - m_1)$. In particular, an almost optimal solution can be achieved by directly setting $w = S_W^{-1}(m_2 - m_1)$ – the optimal solution is proportional to this one – where $w_0 = w^T m$ with m being the global average of all samples.

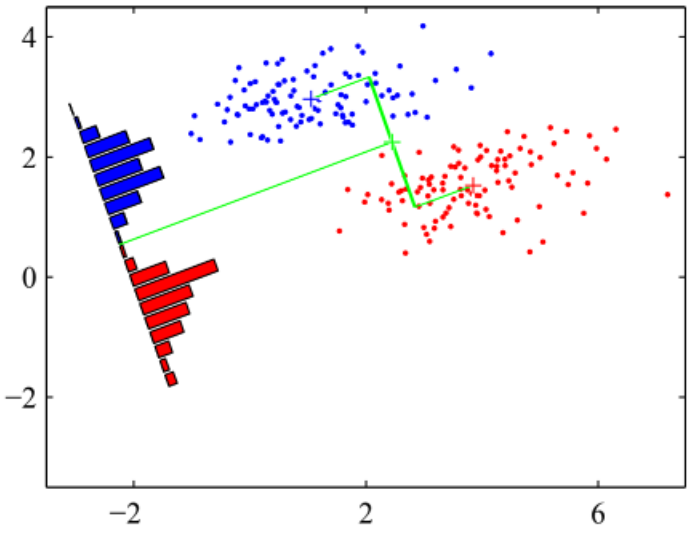

Figure 4.8: The optimal solution depends on the inverse of the within class scatter.

For a multiple classes generalization, we consider the discriminant $y = W^T x$, where we want to maximize:

$$J(W) = \text{tr}\left((WS_W W^T)(WS_W W^T)^{-1}\right)$$

In perceptrons, we saw how the final discriminant is always very close to one of the two datasets. Using Fisher's discriminant, instead, the final discriminant is nothing more than a rotation of a bisector, hence it will always lie exactly in the middle of the two sample groups. But which of the two solutions is better? For samples that are in the dataset, the two solutions are equivalent: all the samples will be correctly classified. For new samples, instead, we expect Fisher's solution to have a lower probability of misclassification.

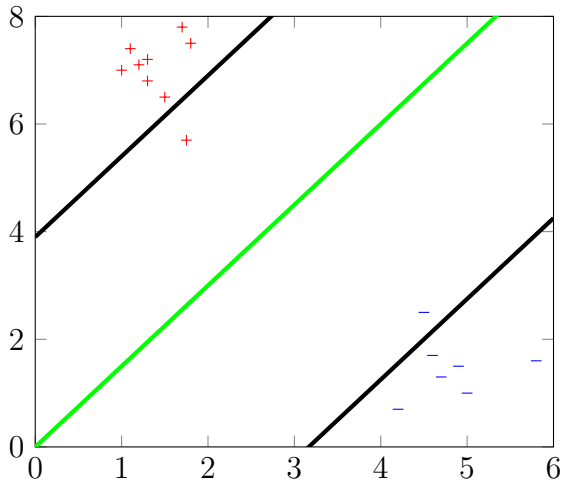

Figure 4.9: The green discriminant may be found through Fisher's criterion. The two black lines may be found through perceptrons. Fisher's discriminant makes no errors, while both perceptron discriminants make an error.

4.1.4 Support Vector Machines

After discussing how Fisher's discriminant is (on average) better than perceptrons due to how the generated discriminant has a good margin of error from both sample groups, hence lower probability of misclassifying samples, we can build a new model based on this idea. **Support Vector Machines (SVM)** aims at maximizing such margin, providing better accuracy.

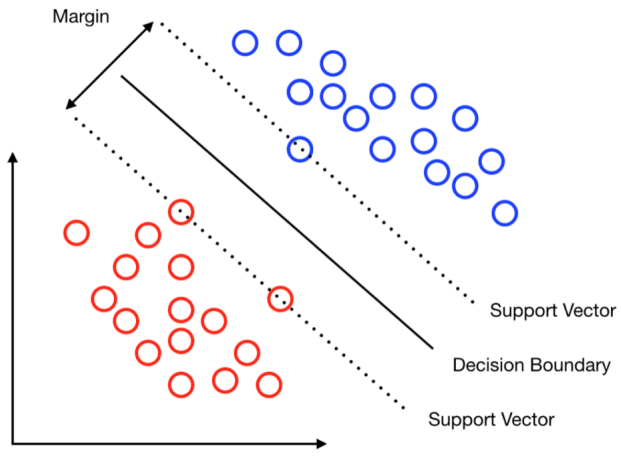

Figure 4.10: General idea behind SVMs

Consider a binary classification problem $f : \mathbb{R}^d \to \{+1, -1\}$ and a linearly separable dataset $D = \{(x_i, t_i) \mid i \in [N]\}$ where $t_i \in \{+1, -1\}$. Then, there is a function $y(x) = w^T x + w_0$ such that forall $i \in [N]$ it holds that $y(x_i) > 0$ if $t_i = +1$ and $y(x_i) < 0$ if

$t_i = -1$, implying that:

$$\forall i \in [N] \;\; t_i y(x_i) > 0$$

Let x_k be the point of D closest to the affine hyperplane $h : w^T x + w_0 = 0$. Then, the margin is the smallest distance from x_k and h, which is equal to $\frac{|y(x_k)|}{\|w\|}$. Using the fact that $|y(x_i)| = t_i y(x_i)$, we can compute the margin by minimizing such distance:

$$\min_{i \in [N]} \frac{|y(x_i)|}{\|w\|} = \frac{1}{\|w\|} \min_{i \in [N]} t_i (w^T x + w_0)$$

Thus, to find the hyperplane which maximizes such margin, we can solve the following optimization problem:

$$w^*, w_0^* \in \arg\max_{w, w_0} \frac{1}{\|w\|} \min_{i \in [N]} t_i (w^T x + w_0)$$

After the maximum margin hyperplane $h^* : {w^*}^T x + w_0^*$ is found, there will be at least two closest points $x_k^{\oplus}$ and $x_k^{\ominus}$, one for each class.

We also notice that, since this doesn't affect the solution, we can also rescale all the points in such a way that $t_i(w^T x + w_0) = 1$ holds for all $i \in [N]$. Hence, we consider the canonical representation where $t_i(w^T x + w_0) \geq 1$ for all $i \in [N]$. In this setup, the optimal solution is given by:

$$w^*, w_0^* \in \arg\max_{w, w_0} \frac{1}{\|w\|} \min_{i \in [N]} t_i (w^T x + w_0) = \arg\max_{w, w_0} \frac{1}{\|w\|} = \arg\min_{w, w_0} \frac{1}{2} \|w\|^2$$

subject to $t_i(w^T x + w_0) \geq 1$ for all $i \in [N]$. Hence, we get a quadratic programming optimization problem, which can be solved through the Lagrangian method. For our interest, we won't dive into how this problem is solved, considering directly its solution:

$$w^* = \sum_{i=1}^{N} a_i^* t_i x_i$$

where each a_i^* is a Lagrangian multiplier, a byproduct of the following Lagrangian optimization problem:

$$L(a) = \sum_{i=1}^{N} a_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} a_i a_j t_i t_j x_i^T x_j$$

subject to:

$$a_i \geq 0 \qquad \forall i \in [N]$$

$$\sum_{i=1}^{N} a_i t_i = 0 \qquad \forall i \in [N]$$

Proposition 2: Karush-Kuhn-Tucker (KKT) condition

Given the above solution to the maximal margin hyperplane problem:

$$w^* = \sum_{i=1}^{N} a_i^* t_i x_i$$

For each $i \in [N]$, it holds that either $a_i^* = 0$ or $t_i y(x_i) = 1$

The KKT conditions ensures that, in the canonical representation, every time $t_i y(x_i) \neq 1$ then $a_i^* = 0$, meaning that these terms do not contribute to the solution. Hence, the **support vectors** are given by each x_i such that $t_i y(x_i) = 1$ (and thus $a_i^* > 0$).

$$SV = \{x_i \mid i \in [N], t_i y(x_i) = 1\}$$

Hence, we get that:

$$y(x) = \sum_{x_i \in SV} a_i^* t_i x^T x_i + w_0^*$$

To compute w_0^*, we pick a support vector $x_k \in SV$. Since x_k satisfies $t_k y(x_k) = 1$, we get that:

$$t_k \left(\sum_{x_i \in SV} a_i^* t_i x_k^T x_i + w_0^* \right) = 1$$

Moreover, since $t_i \in \{-1, +1\}$, we know that $t_i^2 = 1$, concluding that:

$$w_0^* = t_k - \sum_{x_i \in SV} a_i^* t_i x_k^T x_i$$

The optimization problem for determining w^*, w_0^* ($d+1$ dimensions) is thus transformed into an optimization problem for determining a^* ($|D|$ dimensions). This method is very efficient when $d \ll |D|$, in particular when d is large or infinite.

A more stable solution can be obtained by averaging over all support vectors:

$$w_0^* = \frac{1}{|SV|} \sum_{x_k \in SV} \left(t_k - \sum_{x_i \in SV} a_i^* t_i x_k^T x_i \right)$$

In conclusion, the maximum margin hyperplane $h^* : y(x) = 0$ is given by:

$$y(x) = \sum_{x_i \in SV} a_i^* t_i x^T x_i + \frac{1}{|SV|} \sum_{x_k \in SV} \left(t_k - \sum_{x_i \in SV} a_i^* t_i x_k^T x_i \right)$$

The classification of new instance $x \in \mathbb{R}^d - \mathbb{R}_D^d$ is determined by $\text{sign}(y(x))$.

We observe that this method can also be applied when the dataset is *almost* linearly separable, i.e. when only a few points are on the wrong side, with addition of some *slack variables* ξ_i for all $i \in [N]$:

- We set $\xi_i = 0$ if the point x_i is on the correct side of the decision boundary and outside the margin (or on its border)
- We set $0 < \xi_i \leq 1$ if the point x_i is in the correct side of the decision boundary but inside the margin
- We set $\xi_i \geq 1$ if the point x_i is in the wrong side of the decision boundary

These slack variables are used to a *soft margin constraint* version of the optimization problem :

$$w^*, w_0^* \in \arg\min_{w,w_0} \frac{1}{2} \|w\|^2 + C \sum_{i=1}^{N} \xi_i$$

subject to:

$$t_i y(x_i) \geq 1 - \xi_i \qquad \forall i \in [N]$$
$$\xi_i \geq 0 \qquad \forall i \in [N]$$

where C is a constant (usually the inverse of a regularization coefficient).

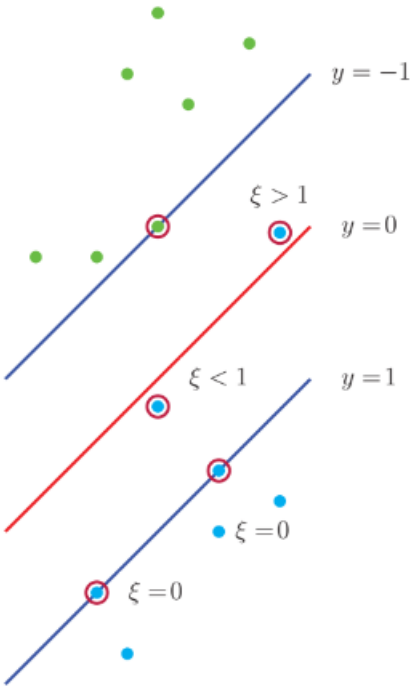

Figure 4.11: Example of slack variable settings.

4.2 Linear models for regression

We recall that regression problems consist in functions where the co-domain is real valued. In particular, in this section we'll consider functions of the form $f : \mathbb{R}^d \to \mathbb{R}$. Here, the dataset $D = \{(x_i, t_i) \mid i \in [N]\}$ associated real value t_i to the input point x_i.

We recall that regression typically involves the approximation of some function. In particular, **linear regression** involves linear models capable of approximating linear functions. As in the previous section, we consider functions of the form $y(x; w) = w^T x$, where $x = \begin{bmatrix} 1 & x_1 & \cdots & x_d \end{bmatrix}$ is the input and $w = \begin{bmatrix} w_0 & w_1 & \cdots & w_d \end{bmatrix}^T$ are the parameters of the model.

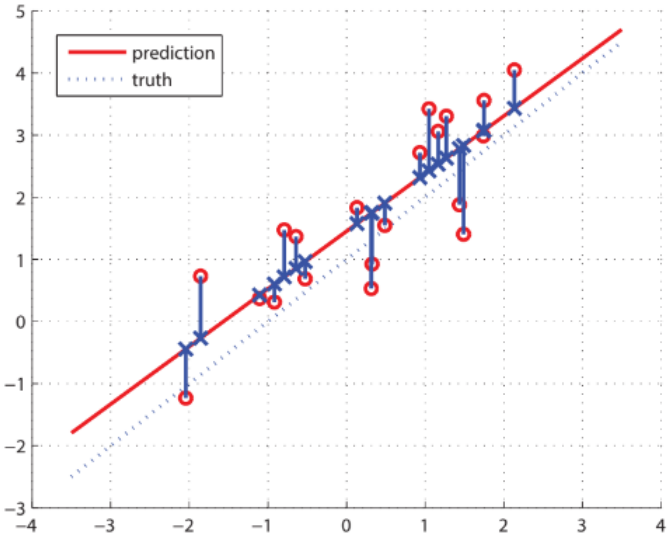

Figure 4.12: Example of approximation through linear regression.

As for linear classification, in linear regression we can also consider linear basis function models, where we apply a non-linear function ϕ on the input variables $y(x; w) = w^T \phi(x)$, where we assume $\phi(x_0) = 1$. By definition, these models are still linear in w.

One of the most commonly used basis function in linear regression is the polynomial function up to some degree. For instance, assuming that $d = 1$, a common basis function $\phi : \mathbb{R} \to \mathbb{R}^M$ is $\phi(x) = \begin{bmatrix} 1 & x & x^2 & \cdots & x^M \end{bmatrix}^T$ in order to get:

$$y = w_0 + w_1 x + w_2 x^2 + \ldots + w_M x^M$$

Here, the choice of the degree M is critical: a small degree will underfit the function, while a large degree will overfit it. Other common choices include the radial function, the sigmoid function and the hyperbolic tangent function.

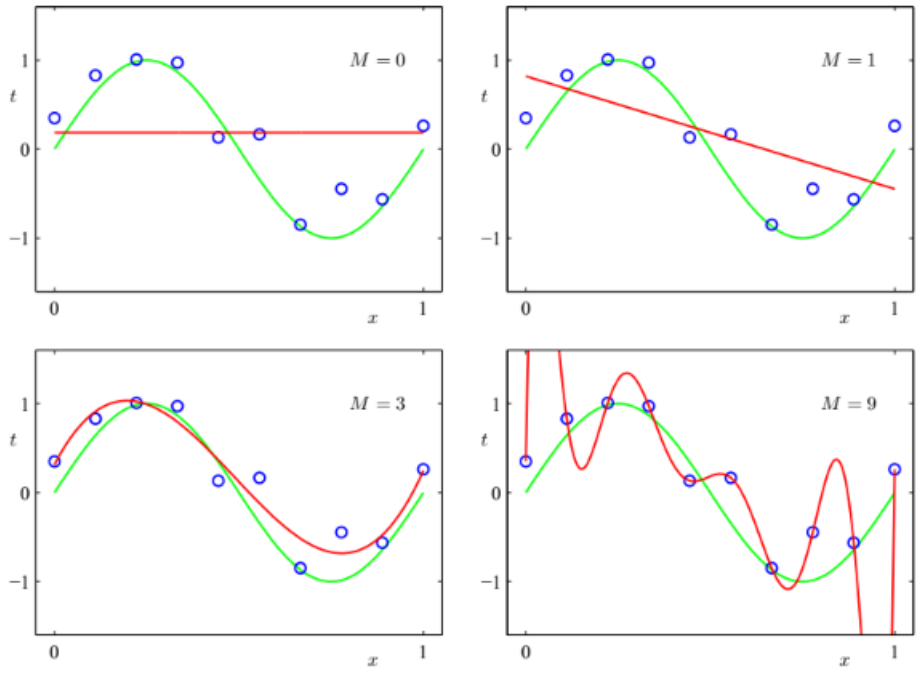

Figure 4.13: The green line is the true function. $M = 0$ and $M = 1$ underfit the function, while $M = 9$ overfits it

Common algorithms for linear regression include:

- Maximum likelihood and least squares
- Sequential Learning

In the **maximum likelihood and least squares** method, a target value t is given by $y(x; w)$, affected by some additive noise ε, i.e. $t = y(x; w) + \varepsilon$. For instance, assuming *Gaussian noise*, we have that the probability of having an error of magnitude ε with precision β is given by:

$$\Pr[\varepsilon \mid \beta] = \mathcal{N}(\varepsilon \mid 0, \beta^{-1})$$

Hence, we get that:

$$\Pr[t \mid x, w, \beta] = \mathcal{N}(t \mid y(x; w), \beta^{-1})$$

Moreover, assuming that the observations $t_1, \ldots, t_N$ are i.i.d., the maximum likelihood function is given by:

$$\Pr[t_1, \ldots, t_N \mid x_1, \ldots x_N, w, \beta] = \prod_{i=1}^{N} \mathcal{N}(t_i \mid w^T \phi(x_i), \beta^{-1})$$

By applying the logarithmic, we get that:

$$\begin{aligned}\ln \Pr[t_1, \dots, t_N \mid x_1, \dots x_N, w, \beta] &= \sum_{i=1}^{N} \ln \mathcal{N}(t_i \mid w^T \phi(x_i), \beta^{-1}) \\ &= -\beta \underbrace{\frac{1}{2} \sum_{i=1}^{N} (t_i - w^T \phi(x_i))^2}_{E_D(w)} - \frac{N}{2} \ln(2p\beta^{-1})\end{aligned}$$

We notice that the sum-of-square error $E_D(w) = \frac{1}{2} \sum_{i=1}^{N} (t_i - w^T \phi(x_i))^2$ lies inside the previous calculation, hence the name of this method.

Thus, we conclude that the maximum likelihood (under zero-mean Gaussian noise assumption) corresponds to the least square error minimization:

$$\arg\max_w \Pr[t_1, \dots, t_N \mid x_1, \dots x_N, w, \beta] = \arg\min_w \frac{1}{2} \sum_{i=1}^{N} (t_i - w^T \phi(x_i))^2$$

We also notice that we can express $E_D(w)$ as:

$$E_D(w) = \frac{1}{2}(t - \Phi w)^T (t - \Phi w)$$

where:

$$t = \begin{bmatrix} t_1 \\ \vdots \\ t_N \end{bmatrix} \qquad \Phi = \begin{bmatrix} \phi(x_1)_0 & \phi(x_1)_1 & \cdots & \phi(x_1)_M \\ \phi(x_2)_0 & \phi(x_1)_1 & \cdots & \phi(x_1)_M \\ \vdots & \vdots & \ddots & \vdots \\ \phi(x_N)_0 & \phi(x_N)_1 & \cdots & \phi(x_N)_M \end{bmatrix}$$

The optimality condition is reached when $\nabla E_D(w) = 0$, which happens if and only if $\Phi^T \Phi w = \Phi^T t$, concluding that:

$$w_{ML} = (\Phi^T \Phi)^{-1} \Phi^T t = \Phi^\dagger t$$

In the **sequential learning** method, we use a *stochastic gradient descent*, where in each iteration we set:

$$\widehat{w} := \widehat{w} - \eta \nabla E_{\{x_i\}}(w) = \widehat{w} + \eta(t_i - \widehat{w}^T \phi(x_i))\phi(x_i)$$

where $i \in [N]$ is randomly chosen, η is a small parameter called *learning rate* and $E_{\{x_i\}}$ is the error on the i-th sample.

To control overfitting, we use a technique called **regularization**, where we consider:

$$\arg\min_w E_D(w) + \lambda E_W(w)$$

with $\lambda > 0$ being the *regularization factor*. For the vector $E_W(w)$, we commonly chose the value:

$$E_W(w) = \frac{1}{2} w^T w$$

While other choices include:

$$E_W(w) = \sum_{j=0}^{M} |w_j|^q$$

for some $q > 0$.

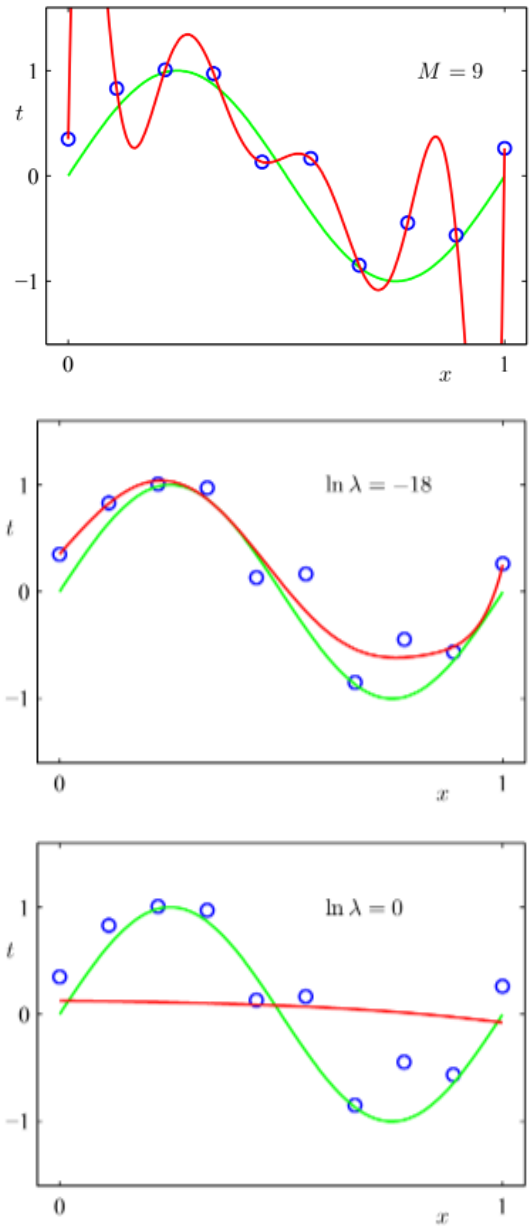

Figure 4.14: Examples of linear regression without regularization (top), with regularization factor $\ln \lambda = -18$ (middle) and with regularization factor $\ln \lambda = 0$ (bottom)

5 Advanced techniques

5.1 Kernel methods

So far, we have considered input objects that are represented by a vector $x \in \mathbb{R}^d$ or by a non-linear transformation $\phi(x)$ that makes the dataset linearly separable. But what about objects of different length or infinite dimensions (e.g. strings, trees, images, ...)? To work with these type of objects, we need to use **kernel functions**, i.e. functions that evaluate the similarity between two objects, mapping them to a set of points on the Euclidean space.

Definition 19: Kernel function

Given two multi-dimensional inputs $x, x' \in \mathcal{X}$ of potentially different dimensions, a **kernel function** is a function $K : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ which evaluates the similarity between x and x'.

Usually, kernel functions are also symmetric, i.e. $k(x, x') = k(x', x)$, and non-negative, i.e. $k(x, x') \geq 0$, even though these properties are not strictly required. There are many types of kernel functions, including:

- *Linear*:
$$k(x, x') = x^T x'$$
- *Polynomial*:
$$k(x, x') = (\beta x^T x' + \gamma)^d$$
where $\beta, \gamma \in \mathbb{R}$ and $d \in \mathbb{N}_{>1}$
- *Radial Basis function (RBF)*:
$$k(x, x') = \exp(-\beta \left|x - x'\right|^2)$$
where $\beta \in \mathbb{R}$

- *Sigmoid*:
$$k(x, x') = \tanh(\beta x^T x' + \gamma)$$
where $\beta, \gamma \in \mathbb{R}$

The input data in the dataset D must usually be normalized in order for the kernel to be a good similarity measure. Several types of normalizations include:

- *Min-max*:
$$\overline{x} = \frac{x - x_{min}}{x_{max} - x_{min}}$$
where x_{max}, x_{min} are respectively the maximum and minimum value in D
- *Standardization*:
$$\overline{x} = \frac{x - \mu}{\sigma}$$
where μ, σ are respectively the mean and standard deviation of D
- *Unit vector*:
$$\overline{x} = \frac{x}{\|x\|}$$

For the rest of this section, we'll be assuming to be working with normalized data.

Proposition 3: Kernel trick (or kernel substitution)

Given a kernel function k, whenever an input vector x is used in an algorithm only in the form of an inner product $x^T x'$ for some x', we replace such inner product with $k(x, x')$

When applicable, the kernel trick is valid for any input pair x, x', even those with infinite dimension. Moreover, in presence of a basis function ϕ being applied on the feature space, the kernel trick also works: we substitute $\phi(x)^T \phi(x')$ with $k(x, x')$. This also implies that we don't need to know the value of $\phi(x), \phi(x')$, making it very powerful.

For instance, consider a linear model $y(x; w) = w^T x$ with dataset $D = \{(x_i, t_i) \mid i \in [N]\}$. Consider the sum-of-squares error function with an additional regularization, defined in the same way as the previous section but without the non-linear transformation applied:
$$E(w) = \frac{1}{2} \sum_{i=1}^{N} (t_i - w^T x_i)^2 + \lambda w^T w$$

This error function can be re-written as:
$$E(w) = \frac{1}{2} \sum_{i=1}^{N} (t - Xw)^T (t - Xw) + \lambda \|w\|^2$$

where $t = \begin{bmatrix} t_1 & \cdots & t_N \end{bmatrix}^T$ is the output vector and $X = \begin{bmatrix} x_1 & \cdots & x_N \end{bmatrix}^T$ is the design matrix. Here, the optimal solution is obtained by $\nabla E(w) = 0$, which happens when:

$$w^* = -\frac{1}{\lambda}\sum_{i=1}^{N}(t_i - w^T x_i)x_i = \sum_{i=1}^{N}\alpha_i x_i$$

where $\alpha_i = -\frac{1}{\lambda}(t_i - w^T x_i)$. We get that:

$$y(x, w^*) = \sum_{i=1}^{N}\alpha_i x_i^T x$$

Moreover, we also notice that the optimal solution can also be computed in a matrix-like form. In particular, we have that:

$$w^* = (X^T X + \lambda I_N)^{-1} X^T t = X^T (XX^T + \lambda I_N)^{-1} t = X^T \alpha$$

where $\alpha = (XX^T + \lambda I_N)^{-1} t$. As before, we get that:

$$y(x, w^*) = \sum_{i=1}^{N}\alpha_i x_i^T x$$

The matrix $K = XX^T$ is usually called **Gram's matrix**.

$$K = \begin{bmatrix} x_1^T x_1 & \cdots & x_1^T x_N \\ \vdots & \ddots & \vdots \\ x_N^T x_1 & \cdots & x_N^T x_N \end{bmatrix}$$

By applying any kernel trick with a kernel k, we get that:

$$y(x, w^*) = \sum_{i=1}^{N}\alpha_i k(x_i, x)$$

while Gram's matrix becomes:

$$K = \begin{bmatrix} k(x_1, x_1) & \cdots & k(x_1, x_N) \\ \vdots & \ddots & \vdots \\ k(x_1, x_n) & \cdots & k(x_N, x_N) \end{bmatrix}$$

The kernel trick comes in handy mostly in Support Vector Machines (SVM). We saw how in this model the optimal solution has the form:

$$w^* = \sum_{i=1}^{N}\alpha_i x_i$$

The linear discriminant can thus be expressed as:

$$y(x;\alpha) = \text{sign}\left(w_0 + \sum_{i=1}^{N} \alpha_i x_i^T x\right)$$

Applying the kernel trick we get that:

$$y(x;\alpha) = \text{sign}\left(w_0 + \sum_{i=1}^{N} \alpha_i k(x_i, x)\right)$$

with:

$$w_0^* = \frac{1}{|SV|} \sum_{x_k \in SV} \left(t_k - \sum_{x_i \in SV} a_i^* t_i k(x_k, x_i)\right)$$

The kernel trick can also be applied to the regularization of a loss function. Consider the following loss function:

$$E(w) = C \sum_{i=1}^{N} E(y_i, t_i) + \frac{1}{2} \|w\|^2$$

where $y_i = w^T x_i$ and C is the inverse of λ (recall that $\alpha = (K + \lambda I_N)^{-1} t$) and E_ε is an ϵ-insensitive error function.

$$E_\varepsilon(y_i, t_i) = \begin{cases} 0 & \text{if } |y_i - t - i| < \varepsilon \\ |y_i - t_i| & \text{otherwise} \end{cases}$$

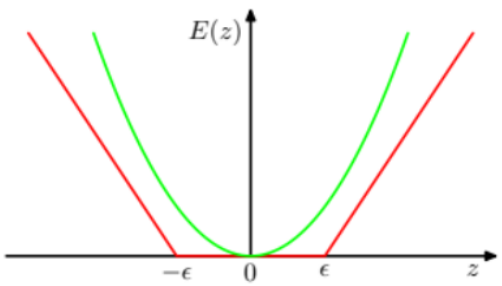

Figure 5.1: The red line represents E_ε where $z = |y_i - t_i|$ as parameter

Since this error function is not differentiable, the problem becomes hard to solve. By introducing the slack variables $\xi_i^+, \xi_i^- \geq 0$ for all $i \in [N]$ and adding the constraints:

$$y(x_i; w) - \varepsilon - \xi_i^- \leq t_i \leq y(x_i; w) + \varepsilon + \xi_i^+$$

For the points inside the ε-tube around the solution, i.e. those where the following inequality holds:

$$y(x_i; w) - \varepsilon \leq t_i \leq y(x_i; w) + \varepsilon$$

hence we have that $\xi_i^+, \xi_i^- = 0$. When $\xi_i^+ > 0$ (or $\xi_i^- > 0$), instead, we have that $t_i < y(x_i; w) + \varepsilon$ (or $t_i < y(x_i; w) - \varepsilon$). The loss function can thus be rewritten as the optimization problem:

$$E(w) = C \sum_{i=1}^{N} (\xi_i^+ - \xi_i^-) + \frac{1}{2} \|w\|^2$$

subject to:

$$\begin{aligned} t_i &\leq y(x_i; w) + \varepsilon + \xi_i^+ \qquad i \in [N] \\ t_i &\geq y(x_i; w) - \varepsilon - \xi_i^- \qquad i \in [N] \\ \xi_i^+, \xi_i^- &\geq 0 \qquad i \in [N] \end{aligned}$$

Again, this is a quadratic program that can be solved through the following Lagrangian method, obtaining that:

$$y(x) = \sum_{i=1}^{N} (\widehat{a_i} - \widehat{a_i}') \cdot k(x, x_i) + \widehat{w_0}$$

$$\widehat{w_0} = t_i - \varepsilon - \sum_{i=1}^{N} (\widehat{a_i} - \widehat{a_i}') \cdot k(x, x_i)$$

where for each $i \in [N]$ we have that $0 < \widehat{a_i} < C$. From the KKT condition, support vectors contribute to predictions:

- If $\widehat{a_i} > 0$ then $\varepsilon + \xi_i + y(x_i; w) - t_i = 0$, hence the data points lie on or above the upper boundary of the ε-tube
- If $\widehat{a_i}' > 0$ then $\varepsilon + \xi_i - y(x_i; w) + t_i = 0$, hence the data points sill lie on or above the upper boundary of the ε-tube

For all the other data points inside the ε-tube, we have that $\widehat{a_i}, \widehat{a_i}' = 0$, thus they don't contribute to the prediction.

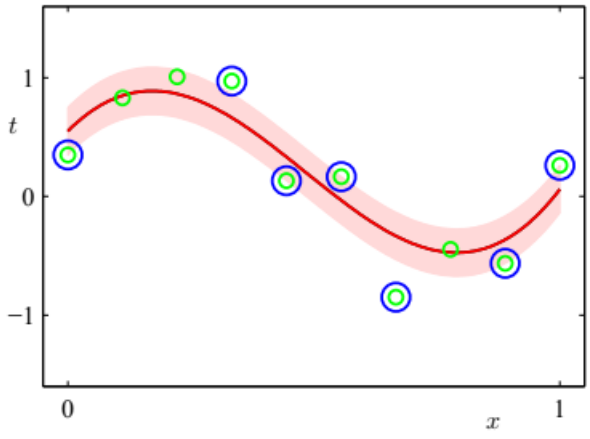

Figure 5.2: Example of support vectors and the ε insensitive tube.

5.2 K-Nearest Neighbors

In the previous sections, we have only focused on models that have a fixed number of parameters. In *non-parametric models*, instead, the number of parameters grows with the amount of data. **Instance based learning** is one of the simplest non-parametric models. In these model, classification is dictated only by the instances, without the necessity of finding an optimal solution. One way to achieve instance based learning is through the **K-Nearest Neighbors (K-NN)** algorithm. Every time a new instance $x \in X - X_D$ has to be classified, the algorithm computes the set of the K-nearest neighbors $N_K(x, D)$ of x in X_D, assigning to x the most common class among the majority of the neighbors.

Algorithm 7: K-Nearest Neighbors (K-NN)

Given a classification problem $f : X \to Y$, a dataset $D = \{(x_i, t_i) \mid i \in [N]\}$ and a new instance $x \in X - X_D$, the algorithm classifies a new instance $x \in X - X_D$.

function K-NN(K, x, D)
  Compute $N_K(x, D)$
  Assign to x the most common class in $N_K(x, D)$
end function

Clearly, the likelihood of a class C being the chosen class for x is given by:

$$\Pr[C \mid x, D, K] = \frac{1}{K} \sum_{x_i \in N_K(x,D)} \mathbb{1}(t_i = C)$$

where $\mathbb{1}(e) = 1$ if the event e is true, otherwise $\mathbb{1}(e) = 0$. We also notice that, when $K = 1$, this algorithm actually partitions the feature space into a Voronoi tesselation.

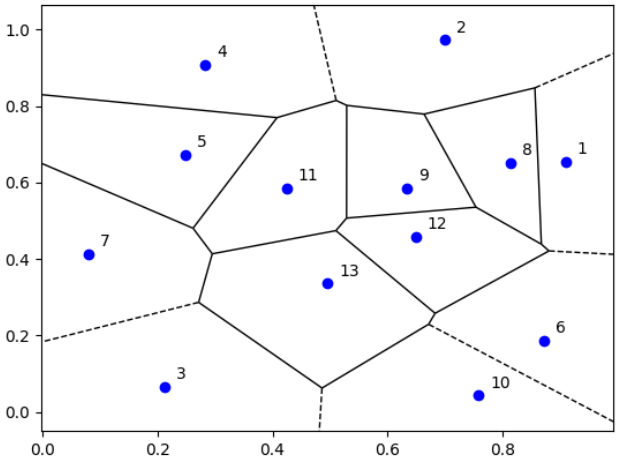

Figure 5.3: Example of Voronoi tessellation produced by K-NN when $K = 1$.

Although simple and effective, the K-NN algorithm requires the whole dataset to be

memorized and available at any time. Moreover, the computation of $N_K(x, D)$ strictly depends on the type of distance function used. Moreover, we also notice that the choice for the value K is *critical*: a small K will produce rough and dull regions, while a bigger value K produces smoother regions, reducing overfitting. However, choosing a value K that is too big will make the computation expensive.

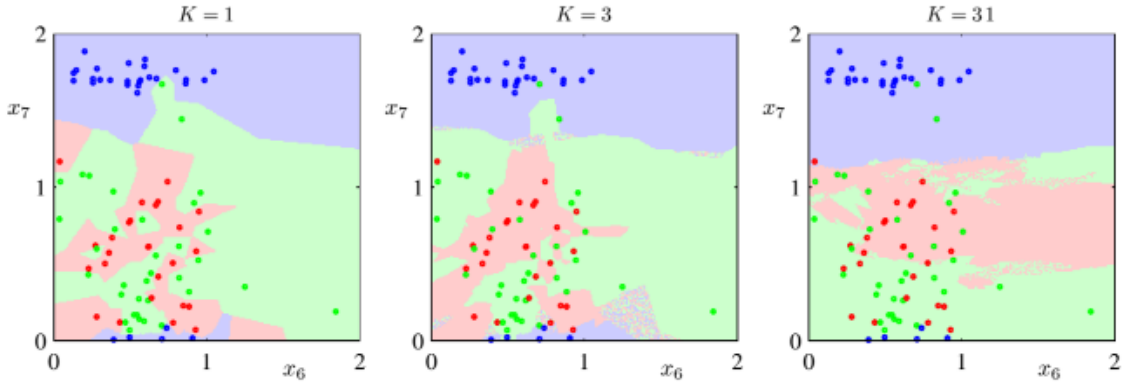

Figure 5.4: Difference in smoothness of the regions based on the value K.

A commonly used distance function is $\text{dist}(x, x') = \|x - x'\|^2$. In fact, this distance function can be easily kernelized with a kernel function k:

$$\|x - x'\|^2 = x^T x + x'^T x' - 2x^T x' = k(x, x) + k(x', x') - 2k(x, x')$$

The K-NN algorithm can also be used to compute a **locally weighted regression**, a regression model formulated on the K-nearest neighbors of a query sample. The union of mode locally weighted regressions can be used to estimate a rough regression model for the feature space.

Algorithm 8: Locally weighted regression

Given a classification problem $f : X \to \mathbb{R}$, a dataset $D = \{(x_i, t_i) \mid i \in [N]\}$ and an query sample $x_q \in X - X_D$, the algorithm fits a local regression model around x_q

function LocallyWeightedRegression(K, x_q, D)
  Compute $N_K(x_q, D)$
  Fit a regression model $y(x; w)$ on $N_K(x_q, D)$
  Return $y(x_q; w)$
end function

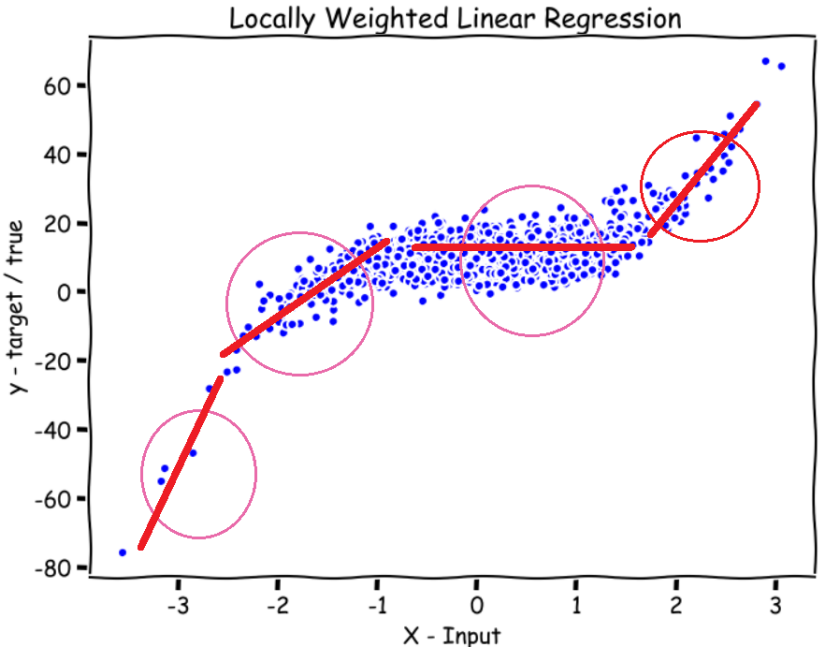

Figure 5.5: Example of multiple locally weighted regressions.

5.3 Multiple learners approach

Until now, we have focused on trying to find a powerful complex model capable solving our problems. In the **multiple learners approach**, instead, we focus on training many different models to then combine their results. These models can be trained in parallel (via *voting* or *bagging*) or in sequence (via *boosting*).

Given a dataset $D = \{(x_i, t_i) \mid i \in [N]\}$, the **voting** method first trains a set of models $y_1(x), \ldots, y_m$ on the same dataset to then make predictions using:

- *Regression*:

$$y_{\text{voting}}(x) = \sum_{j=1}^{M} w_j y_j(x)$$

  where $w_j \geq 0$ and $\sum_{j=1}^{M} w_j = 1$ – this acts as the prior probability of each model.

- *Classification* (weighted majority):

$$y_{\text{voting}}(x) = \arg\max_{C} \sum_{j=1}^{M} w_j \mathbb{1}(y_j(x) = C)$$

where $w_j \geq 0$ and $\sum_{j=1}^{M} w_j = 1$ – this acts as the prior probability of each model.

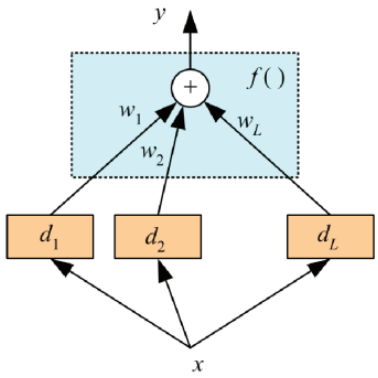

Figure 5.6: Example of voting method where the function f combines the results of the models.

The voting method can also be implemented through more advanced techniques that manipulate how the results are combined together:

- *Gating* (or *mixture of experts*): an additional non-linear gating function changes the weights $w_1, \ldots, w_M$ based on the input value.

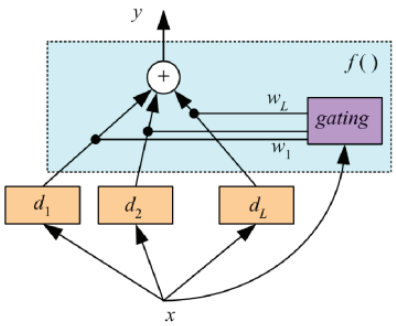

Figure 5.7: Example of voting method with gating applied.

- *Stacking*: the combinating function is also learned.

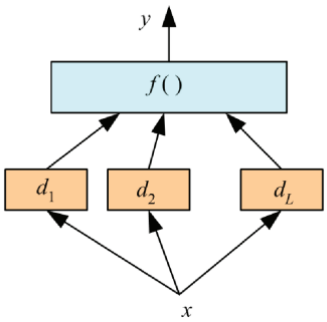

Figure 5.8: Example of voting method with stacking applied.

- *Cascading*: the models learn in a cascading way, feeding the output to the next model when a confidence treshold is not reached.

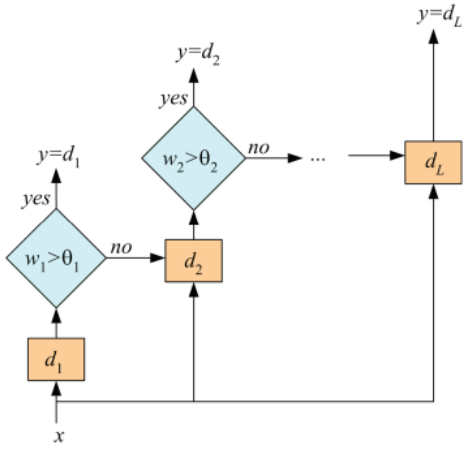

Figure 5.9: Example of voting method with cascading applied.

In the **bagging** method, instead, the models are trained on different bootstrap datasets $D_1, \ldots, D_M$ that are extrapolated from the original dataset D, i.e. $D_i \subset D$. The samples of this datasets are chosen with random sampling with replacement, meaning that they may not cover the whole original dataset and may not be pairwise disjoint. In general, this approach gives better results compared to training each model with the same dataset. After training each model $y_j(x)$ with the dataset D_j, the final predictions are made with a voting scheme:

$$y_{\text{bagging}}(x) = \frac{1}{M} \sum_{j=1}^{M} y_j(x)$$

A concept similar to cascading voting is the **boosting** method. Here, the weak learners are trained sequentially, where each classifier is trained on weighted data. These weights depend on the performance of the previous classifiers: points that are misclassified by previous classifiers are given greater weight. Finally, the predictions are based on the weighted majority of votes.

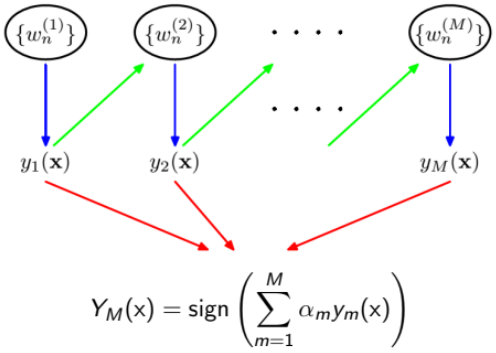

Figure 5.10: Example of boosting method.

The main algorithm used for boosting is **AdaBoost**. In this approach, the individual learners can be weak, but as long as the performance of each one is slightly better than random guessing, the final model can be proven to converge to a strong learner.

Algorithm 9: AdaBoost

Given a classification problem $f : X \to \mathbb{R}$ and a dataset $D = \{(x_i, t_i) \mid i \in [N]\}$ where $t_i \in \{-1, +1\}$, the algorithm produces a classifier using the boosting method.

function AdaBoost(X, D)

Initialize $w_1^{(1)}, \ldots, w_N^{(1)}$ to $\frac{1}{N}$

for $j \in [M]$ **do**

Train a weak learner $y_j(x)$ by minimizing the weighted error function

$$J_j = \sum_{i=1}^{N} w_i^{(j)} \mathbb{1}(y_j(x_i) \neq t_i)$$

Evaluate

$$\varepsilon_j = \frac{J_j}{\sum_{i=1}^{N} w_i^{(j)}}$$

Evaluate

$$\alpha_j = \ln \frac{1 - \varepsilon_j}{\varepsilon_j}$$

Set

$$w_i^{(j+1)} = w_i^{(j)} \exp(\alpha_j \mathbb{1}(y_j(x_i) \neq t_i))$$

end for

Return the classifier $Y_M(x) = \text{sign}\left(\sum_{j=1}^{M} \alpha_j y_j(x)\right)$

end function

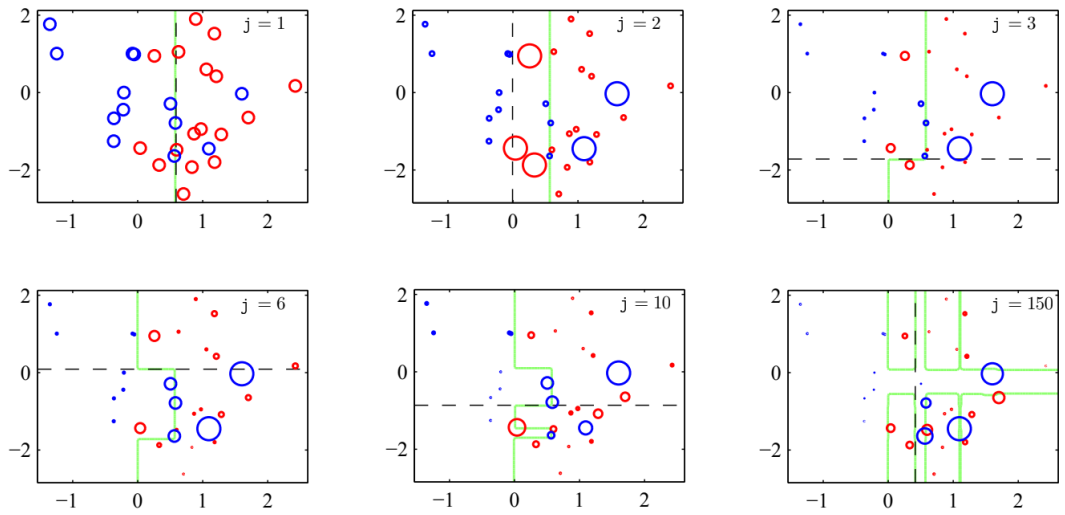

Figure 5.11: Example of iterations in the AdaBoost algorithm. The green line represents the current final classifier.

The AdaBoost algorithm can be explained as the sequential minimization over $a_1, \ldots, a_M$, $y_1, \ldots, y_M$ of the following exponential error function:

$$E = \sum_{i=1}^{N} \exp(-t_i f_M(x_i))$$

where:

$$f_M(x) = \frac{1}{2} \sum_{j=1}^{M} a_j y_j(x)$$

Assume that $y_1(x), \ldots, y_{M-1}(x), \alpha_1, \ldots, \alpha_{M-1}$ are fixed. We notice that:

$$\begin{aligned} E &= \sum_{i=1}^{N} \exp(-t_i f_M(x_i)) \\ &= \sum_{i=1}^{N} \exp(-t_i f_{M-1}(x_i) - \frac{1}{2} t_i \alpha_M y_M(x_i)) \\ &= \sum_{i=1}^{N} w_i^{(M)} \exp(-\frac{1}{2} t_i \alpha_M y_M(x_i)) \end{aligned}$$

where $w_i^{(M)} = \exp(-t_i f_{M-1}(x_i))$ being constant while minimizing with respect to $y_M(x)$ and α_M. From the sequential minimization given by AdaBoost, we know that:

$$w_i^{(M)} = w_i^{(M-1)} \exp(\alpha_{N-1} \mathbb{1}(y_{M-1}(x_i) \neq t_i)) \qquad \alpha_M = \ln \frac{1 - \varepsilon_M}{\varepsilon_M}$$

Since the predictions are made with:

$$\text{sign}(f_M(x)) = \text{sign}\left(\frac{1}{2} \sum_{j=1}^{M} a_j y_j(x)\right)$$

we get that:

$$Y_M(x) = \text{sign}\left(\sum_{j=1}^{M} \alpha_j y_j(x)\right)$$

proving that AdaBoost minimizes such error function. The AdaBoost algorithm is very fast, simple and easy to program. Moreover, no prior knowledge about the base learners is required and no parameters have to be tuned (except for the number M of learners), yielding theoretical guarantees given sufficient data and base learners with moderate accuracy. However, the performance strictly depends on data (noise sensitive) and which base learners are used (it can fail with insufficient data or when base learners are too weak).

6

Artificial Neural Networks

6.1 Feedforward Neural Networks

Neural network learning methods provide a robust approach to approximating real-valued, discrete-valued, and vector-valued target functions. For certain types of problems, such as learning to interpret complex real-world sensor data, artificial neural networks are among the most effective learning methods currently known.

Given a function $f : X \to Y$, where Y is either a discrete set of classes or a real-valued interval, and a dataset $D = \{(x_i, t_i) \mid i \in [N]\}$, our general framework will be to define an approximating function $y = \widehat{f}(x; \theta)$ and learn the parameters θ in order to get $\widehat{f}(x_i) \approx t_i$.

Artificial Neural Networks (ANN) draw inspiration from brain structures, where the neurons fire sequentially, forming a chain, in order to produce an output. In our context, each hidden layer's output can be seen as an array of unit (neuron) activations based on the connections with the previous units.

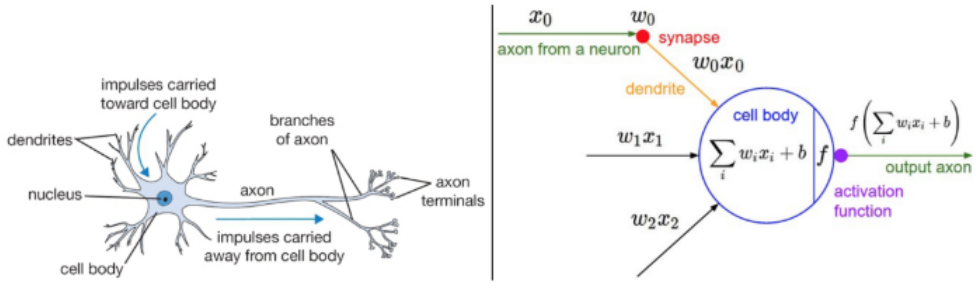

Figure 6.1: Comparison between a brain neuron and an artificial neuron.

ANNs are often called **Feedforward Neural Networks (FNN)** or **Multilayer Perceptrons (MLP)**. In particular, the term *feedforward* refers to how the information flows from input to output without any loop, while the term *network* refers to how the function

f is a composition of elementary functions in an acyclic graph. These internal functions are called **activation functions**.

In general, we have that:

$$f(x;\theta) = f^{(n)}(\dots f^{(2)}(f^{(1)}(x;\theta^{(1)}),\theta^{(2)})\dots,\theta^{(n)})$$

where $f^{(i)}$ is the i-th layer of the network and $\theta^{(i)}$ are the corresponding parameters. By definition, FNNs are **chain structures**. The length of the chain is called *depth* of the network and the final layer is called *output layer*. **Deep learning** is the subset of artificial intelligence focused on studying networks with a large depth.

But why use FNNs instead of the previous models? This choice is largely due to some typical restrictions of the previous models: linear models cannot model interaction between input variables, while kernel methods require the choice of a suitable kernels. In particular, both generic kernels (RVF, polynomial, ...) and hand-crafted kernels (application specific) transform the linear problem into a convex one. FNNs, instead, are based on the complex combination of many parametric functions, making the problem non-convex. The true strength behind FNNs is state through the following very powerful theorem.

Theorem 1: Universal Approximation theorem

A FFN with a linear output layer and at least one hidden layer with any "squashing" activation function (e.g. sigmoid, ...) can approximate any Borel measurable function with any desired amount of error, provided that enough hidden units are used.

To give an easy example, suppose that we want to learn the function XOR : $\{0,1\}^2 \to \{0,1\}$. Consider the complete dataset:

$$D = \{(\begin{bmatrix}0 & 0\end{bmatrix}^T, 0), (\begin{bmatrix}0 & 1\end{bmatrix}^T, 1), (\begin{bmatrix}1 & 0\end{bmatrix}^T, 1), (\begin{bmatrix}1 & 1\end{bmatrix}^T, 0)\}$$

We notice that, even thought this dataset is non-linearly separable, by applying a basis function we can easily make it linearly separable. Using linear regression with mean squared error (MSE):

$$J(\theta) = \frac{1}{N}\sum_{i=1}^{N}(t_i - y(x_i))^2$$

with $y(x) = w^T x + w_0$, the optimal solution would be given by $w = 0$ and $w_0 = \frac{1}{2}$. Hence, for any possible input the answer would always be $\frac{1}{2}$. This is clearly a wrong model for the problem.

With FNNs, instead, we can define a two layer network to solve the problem.

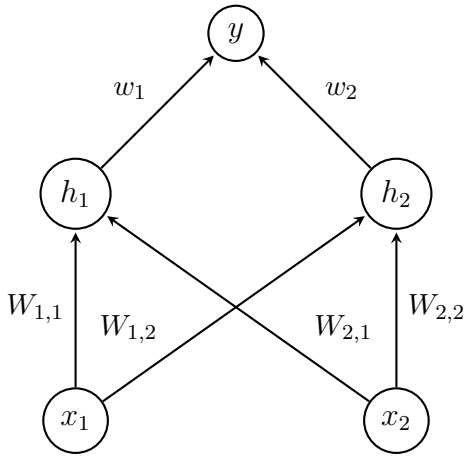

Figure 6.2: The two layer network solving the XOR function on 2 bits.

The universal approximation theorem tells us that the output layer y must be a linear function. Hence, we set $y = w^T h + b$.

The hidden layer h, instead, must use a "squashing" activation function. Hence, we set $h = \text{ReLU}(W^T x + c)$, where ReLU is the *Rectified Linear Unit* function.

$$\text{ReLU}(z) = \max(0, z) = \begin{cases} z & \text{if } z \geq 0 \\ 0 & \text{if } z < 0 \end{cases}$$

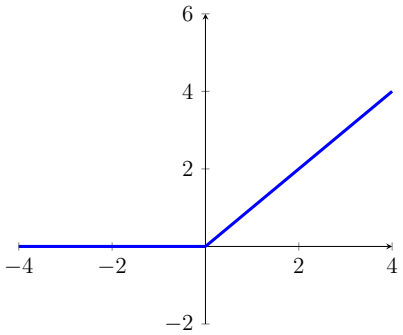

Figure 6.3: The Rectified Linear Unit function

The full model is thus equal to:

$$y(x) = f(x; \theta) = w^T \max(0, W^T, x + c) + b$$

with $\theta = \langle W, c, w, b \rangle$.

Once the model has been defined, we can now learn the parameters θ by minimizing the MSE loss function, obtaining that:

$$W = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \quad c = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \quad w = \begin{bmatrix} 1 & -2 \end{bmatrix} \quad b = 0$$

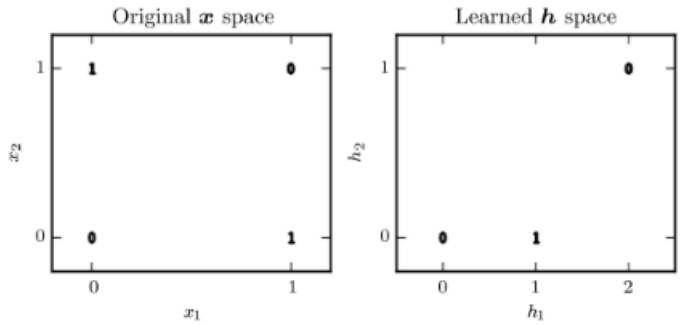

Figure 6.4: The two feature spaces of the two layers after the learning process.

The crucial point in the performance of a FNN is its architecture:

- *Depth choice*: how many layers should the network have?
- *Width choice*: how many units should each layer of the network have?
- *Activation function choice*: which kind of units should each layer have?
- *Loss function choice*: which kind of cost function should the network learn on?

We notice that the universal approximation theorem doesn't say how many units are necessary, but only that a sufficient amount of them is needed. In general, the width of the network tends to be exponential with respect to the size of the input. Moreover, we also notice that, in theory, a short and very wide network can approximate any function. In practice, deep and narrow networks are easier to train and provide better results in generalization.

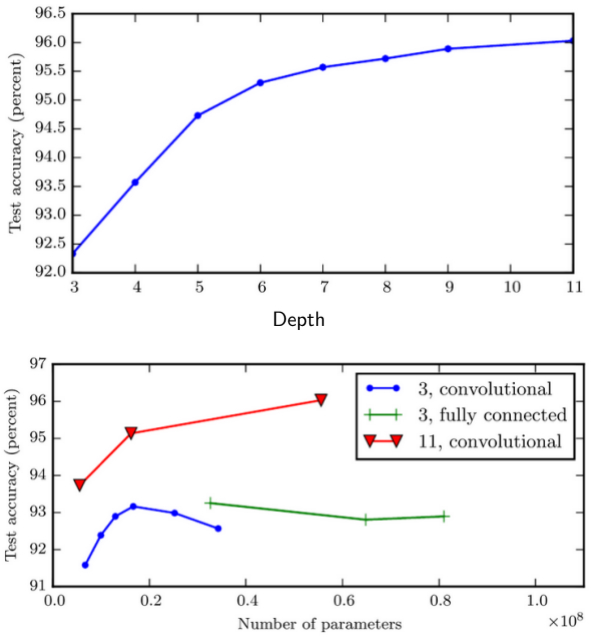

Figure 6.5: Relation between depth, number of parameters and accuracy.

For the choice of activation functions and loss functions, we have to make some remarks regarding the use of gradient-based learning. In particular, we observe that unit saturation can hinder learning. A neuron is said to be saturated when extremely large weights cause the neuron to produce gradient values that are very close to the range boundary. When units saturate, the gradient becomes very small, making lower and lower improvements. To avoid this, we have to choose suitable cost functions and unit non-linearity functions.

By definition, the FNN model implicitly defines a conditional distribution $\Pr[t \mid x, \theta]$. The loss function that we'll be using is the cross-entropy maximum likelihood principle:

$$J(\theta) = \mathop{\mathbb{E}}_{x,t\sim\mathcal{D}}[-\ln \Pr[t \mid x, \theta]]$$

where $x, t \sim \mathcal{D}$ denotes the extraction of two random variables x, t from $\mathcal{D}$. Assuming that we have additive Gaussian noise, we know that:

$$\Pr[t \mid x, \theta] = \mathcal{N}(t \mid f(x; \theta), \beta^{-1} I)$$

and thus that:

$$J(\theta) = \mathop{\mathbb{E}}_{x,t\sim\mathcal{D}}\left[\frac{1}{2}\mathcal{N}(t \mid f(x; \theta), \beta^{-1} I)\right]$$

In other words, the maximum likelihood estimation with additive Gaussian noise corresponds to the mean squared error minimization.

The choice the **output unit activating function** is strictly related to the choice of the cost function. Let h be the output of the last hidden layer, meaning that $y = f^{(n)}(h; \theta)$. Since the universal approximation theorem requires that this activating function is linear, we have a restricted set of choices.

- For *regression*, the identity is chosen as activating function – meaning that $y = W^T h + b$ – while the chosen loss functions is the mean squared error (equivalent to maximum likelihood under additive Gaussian noise assumption). In this setup, the output units never saturate due to them being linear.
- For *binary classification*, the sigmoid is chosen as activating function – meaning that $y = \sigma(W^T h + b)$ – while the chosen loss functions is the binary cross entropy, where the likelihood is modeled by a Bernoulli distribution:

  $$J(\theta) = \mathop{\mathbb{E}}_{x,t\sim\mathcal{D}}[-\ln \Pr[t \mid x]]$$

  In this setup, the output units saturate only when they give the correct answer.
- For *multi-class classification*, the softmax is chosen as activating function

  $$y_i = \text{softmax}(\alpha^{(i)}) = \frac{\exp(\alpha^{(i)})}{\sum_{j\in Y} \exp(\alpha_j)}$$

where $\alpha^{(i)} = w_i^T h + b_i$, while the chosen loss functions is the categorical cross entropy, where the likelihood is modeled by a multinomial distribution:

$$J_i(\theta) = \mathop{\mathbb{E}}_{x,t\sim\mathcal{D}}[-\ln \text{softmax}(\alpha^{(i)})]$$

In this setup, the output units saturate only when there are minimal errors.

For the **hidden unit activation functions**, instead, we have no theoretical principles, only intuition. In fact, predicting which activation function will work best is usually impossible. Common choices include the rectified linear unit function $\text{ReLU}(\alpha)$, the sigmoid function $\sigma(\alpha)$ and the hyperbolic tangent function $\tanh(\alpha)$. In particular, we observe that the last two are closely related since:

$$\tanh(\alpha) = 2\sigma(2\alpha) - 1$$

Since none of these functions has a logarithmic term inside them, the units saturate easily. Moreover, gradient based learning is very slow. The hypoerbolic tangent is usually preferred since it gives larger gradients with respect to the sigmoid function.

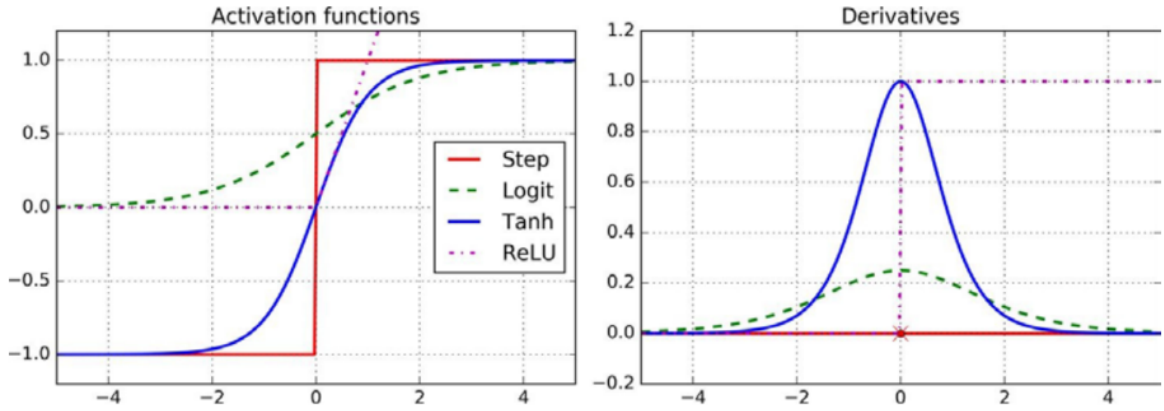

Figure 6.6: Comparison of the functions and their derivatives. The logit function is the inverse function of the sigmoid.

6.2 Backpropagation and learning in ANNs

Inside a FNN, information flows forward through the network when computing the output for an input. To train the network, we need to compute the gradients with respect to the network parameters. The usual choice for this task is the **backpropagation** algorithm, used to propagate gradient computations from the loss function through the whole network. This algorithm makes extensive use of the derivative chain rule – recall that the network is a sequence of composite functions. We observe that backpropagation isn't a training algorithm: it's just a method to compute the gradient. Moreover, the method isn't specific to FFNs.

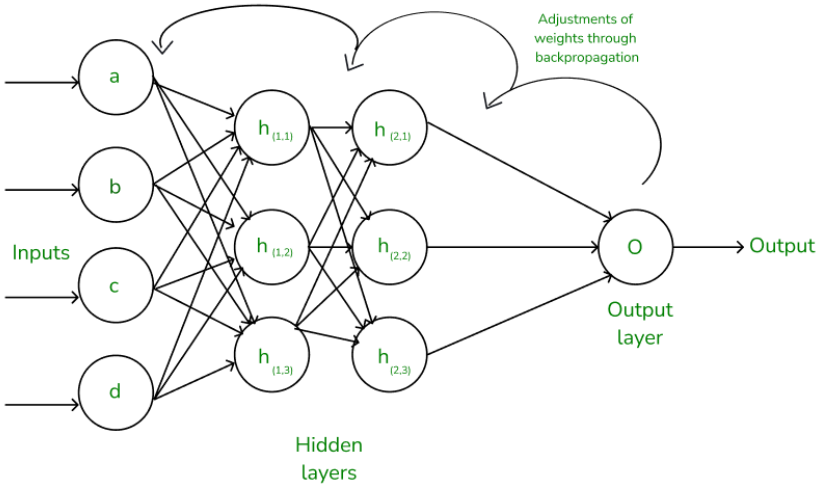

Figure 6.7: Visual representation of backpropagation.

The backpropagation algorithm is divided in two steps: a `ForwardStep` and a `BackwardStep`. The former computes the output value of the network and the loss value for that input, while the latter computes the gradient propagating it from the loss value to each layer of the network. The `ForwardStep` is nothing more than an iterative algorithm that computes the composition of the functions.

Algorithm 10: Forward step of backpropagation

Given the network depth ℓ, the weight matrices $W^{(1)}, \dots, W^{(\ell)}$, the bias parameters $b^{(1)}, \dots, b^{(\ell)}$, the input value x and the target value t, the following algorithm computes the output value of the network.

function ForwardStep$(\ell, W^{(1)}, \dots, W^{(\ell)}, b^{(1)}, \dots, b^{(\ell)}, x, t)$
  $h^{(0)} = x$
  for $k \in [\ell]$ **do**
    $\alpha^{(k)} = b^{(k)} + W^{(k)} h^{(k-1)}$
    $h^{(k)} = f^{(k)}\left(\alpha^{(k)}\right)$
  end for
  $y = h^{(\ell)}$
  $J = L(t, y)$ ▷ L is the loss function
  Return y, J
end function

The `BackwardStep` is highly based on the vectorial chain rule. Given $y = g(x)$ and $z = f(y) = f(g(x))$, through the chain rule we have that:

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$$

Similarly, for the vector functions $g : \mathbb{R}^m \to \mathbb{R}^n$, $f : \mathbb{R}^n \to \mathbb{R}$ we have that:

$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j} \frac{\partial y_j}{\partial x_j}$$

Equivalently, in vector notation we have that:

$$\nabla_x z = \left(\frac{\partial y}{\partial x}\right)^T \nabla_y z$$

Algorithm 11: Backward step of backpropagation

Given the network depth ℓ, the weight matrices $W^{(1)}, \ldots, W^{(\ell)}$, the bias parameters $b^{(1)}, \ldots, b^{(\ell)}$, the loss value J obtained through the `ForwardStep`, the following algorithm computes the gradient propagating it from loss function to all the layers of the network.

function BACKWARDSTEP($\ell, W^{(1)}, \ldots, W^{(\ell)}, b^{(1)}, \ldots, b^{(\ell)}, J$)
  $g = \nabla_y J$
  for $k = \ell, \ell - 1, \ldots, 1$ **do**
    $g = \nabla_{\alpha^{(k)}} J$, eq. to $g = g \odot f'^{(k)}(\alpha^{(k)})$ ▷ $\odot$ denotes the element-wise product
    $\nabla_{b^{(k)}} J = g$
    $\nabla_{W^{(k)}} J = g \cdot (h^{(k-1)})^T$
    $g = \nabla_{h^{(k-1)}} J$, eq, to $g = (W^{(k)})^T \cdot g$
  end for
  Return g
end function

We notice that the first three instructions inside the for loop of `BackwardStep` actually propagate the gradients to the pre-nonlinearity activations, while the fourth instruction propagates the gradients to the next lower-level hidden layer.

Suppose that we have a network with a 2D input $x = \begin{bmatrix} x_1 & x_2 \end{bmatrix}$ and two hidden layers $h^{(1)} = \begin{bmatrix} h_1^{(1)} & h_2^{(1)} \end{bmatrix}$ and $h^{(2)} = \begin{bmatrix} h^{(2)} \end{bmatrix}$. The last hidden layer is the output layer, hence $y = h^{(2)}$. Let $f^{(1)} = \text{ReLU}$ and $f^{(2)} = \text{id}$.

In the forward step, in order to compute $h^{(1)}, h^{(2)}$ we have to fist compute $\alpha^{(1)} = \begin{bmatrix} \alpha_1^{(1)} & \alpha_2^{(1)} \end{bmatrix}$ and $\alpha^{(2)} = \begin{bmatrix} \alpha^2 \end{bmatrix}$. Since $h^{(0)} = x$, we have that:

$$\alpha^{(1)} = b^{(1)} + W^{(1)} x = \begin{bmatrix} b_1^{(1)} \\ b_2^{(1)} \end{bmatrix} + \begin{bmatrix} w_{1,1}^{(1)} x_1 + w_{1,2}^{(1)} x_2 \\ w_{2,1}^{(1)} x_1 + w_{2,2}^{(1)} x_2 \end{bmatrix}$$

and $h^{(1)} = f^{(1)}(\alpha^{(1)}) = \text{ReLU}(\alpha^{(1)})$. On the second layer, we have that:

$$\alpha^{(2)} = b^{(2)} + W^{(2)} h^{(1)} = \begin{bmatrix} b^{(1)} \end{bmatrix} + \begin{bmatrix} w_{1,1}^{(1)} h_1^{(1)} + w_{1,2}^{(1)} h_2^{(1)} \end{bmatrix}$$

and $h^{(2)} = f^{(2)}(\alpha^{(2)}) = \alpha^{(2)}$. The loss functions MSE is thus given by:

$$L(t, y) = \frac{1}{2}(t - y)^2 = \frac{1}{2}(t - \alpha^{(2)})^2$$

and the parameters of the model are $\theta = \left\langle b^{(1)}, b^{(2)}, W^{(1)}, W^{(2)} \right\rangle$.

$$b^{(1)} = \begin{bmatrix} b_1^{(1)} \\ b_2^{(1)} \end{bmatrix} \quad b^{(2)} = \begin{bmatrix} b^{(2)} \end{bmatrix} \quad W^{(1)} = \begin{bmatrix} w_{1,1}^{(1)} & w_{1,2}^{(1)} \\ w_{2,1}^{(1)} & w_{2,2}^{(1)} \end{bmatrix} \quad W^{(2)} = \begin{bmatrix} w_{1,1}^{(2)} & w_{1,2}^{(2)} \end{bmatrix}$$

This concludes the forward step. To show the correctness of the backward step, we compute the gradients ourself and then show that the algorithm does the same computations in a more efficient way. First, we compute the gradient of the loss function over y.

$$\nabla_y J(\theta) = \frac{\partial J(\theta)}{\partial y} = \frac{\partial}{\partial y}\frac{1}{2}(t - y)^2 = y - t$$

Using the chain rule, we compute the gradient of the loss function over $b^{(2)}$ and $W^{(2)}$.

$$\nabla_{b^{(2)}} J(\theta) = \frac{\partial J(\theta)}{\partial y}\frac{\partial y}{\partial b^{(2)}} = \frac{\partial J(\theta)}{\partial y} \cdot 1 = y - t$$

$$\nabla_{W^{(2)}} J(\theta) = \begin{bmatrix} \frac{\partial J(\theta)}{\partial y}\frac{\partial y}{\partial w_{1,1}^{(2)}} & \frac{\partial J(\theta)}{\partial y}\frac{\partial y}{\partial w_{1,2}^{(2)}} \end{bmatrix} = \begin{bmatrix} \frac{\partial J(\theta)}{\partial y} \cdot h_1^{(1)} & \frac{\partial J(\theta)}{\partial y} \cdot h_2^{(1)} \end{bmatrix} = (y - t)(h^{(1)})^T$$

Similarly, we compute the gradient of the loss function over $h^{(1)}$ and $\alpha^{(1)}$.

$$\nabla_{h^{(1)}} J(\theta) = \begin{bmatrix} \frac{\partial J(\theta)}{\partial y}\frac{\partial y}{\partial h_1^{(1)}} \\ \frac{\partial J(\theta)}{\partial y}\frac{\partial y}{\partial h_2^{(1)}} \end{bmatrix} = \begin{bmatrix} \frac{\partial J(\theta)}{\partial y} w_{1,1}^{(2)} \\ \frac{\partial J(\theta)}{\partial y} w_{1,2}^{(2)} \end{bmatrix} = (y - t)(W^{(2)})^T$$

$$\nabla_{\alpha^{(1)}} J(\theta) = \begin{bmatrix} \frac{\partial J(\theta)}{\partial h_1^{(1)}}\frac{\partial h_1^{(1)}}{\partial \alpha_1^{(1)}} \\ \frac{\partial J(\theta)}{\partial h_2^{(1)}}\frac{\partial h_2^{(1)}}{\partial \alpha_2^{(1)}} \end{bmatrix} = \begin{bmatrix} \frac{\partial J(\theta)}{\partial h_1^{(1)}} \cdot \text{step}(\alpha_1^{(1)}) \\ \frac{\partial J(\theta)}{\partial h_2^{(1)}} \cdot \text{step}(\alpha_2^{1}) \end{bmatrix} = ((y - t)(W^{(2)})^T)\text{step}(\alpha^{(1)})$$

where step is the step function, the derivative of the ReLU function:

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

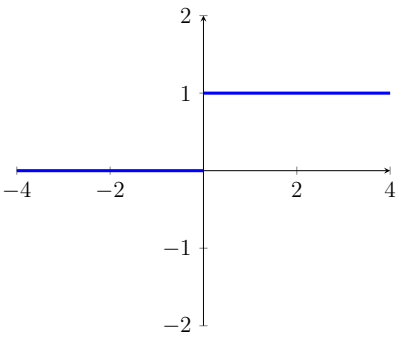

Figure 6.8: The step function

Finally, we compute the gradient of the loss function over $b^{(1)}$ and $W^{(1)}$.

$$\nabla_{b^{(1)}} J(\theta) = \begin{bmatrix} \frac{\partial J(\theta)}{\partial \alpha_1^(1)} \frac{\partial \alpha_1^{(1)}}{\partial b_2^{(1)}} \\ \frac{\partial J(\theta)}{\partial \alpha_2^(1)} \frac{\partial \alpha_2^{(1)}}{\partial b_2^{(1)}} \end{bmatrix} = \begin{bmatrix} \frac{\partial J(\theta)}{\partial \alpha_1^(1)} \cdot 1 \\ \frac{\partial J(\theta)}{\partial \alpha_2^(1)} \cdot 1 \end{bmatrix} = (y-t)(W^{(2)})^T \odot \text{step}(\alpha^{(1)})$$

$$\nabla_{W^1} J(\theta) = \begin{bmatrix} \frac{\partial J(\theta)}{\partial \alpha_1^(1)} \frac{\partial \alpha_1^{(1)}}{\partial w_{1,2}^{(1)}} & \frac{\partial J(\theta)}{\partial \alpha_1^(1)} \frac{\partial \alpha_2^{(1)}}{\partial w_{2,2}^{(1)}} \\ \frac{\partial J(\theta)}{\partial \alpha_1^(1)} \frac{\partial \alpha_1^{(1)}}{\partial w_{1,2}^{(1)}} & \frac{\partial J(\theta)}{\partial \alpha_2^(1)} \frac{\partial \alpha_2^{(1)}}{\partial w_{2,2}^{(1)}} \end{bmatrix} = \begin{bmatrix} \frac{\partial J(\theta)}{\partial \alpha_1^(1)} \cdot x_1 & \frac{\partial J(\theta)}{\partial \alpha_1^(1)} \cdot x_2 \\ \frac{\partial J(\theta)}{\partial \alpha_1^(1)} \cdot x_1 & \frac{\partial J(\theta)}{\partial \alpha_2^(1)} \cdot x_2 \end{bmatrix} = ((y-t)(W^{(2)})^T \odot \text{step}(\alpha^{(1)}))x$$

By comparing the steps of the algorithm and our calculations, it's easy to see that the backforward algorithm is indeed correct. The Backpropagation algorithm is used inside many training algorithms. In particular, we focus on:

- Stochastic Gradient Descent (SDG)
- SGD with momentum
- Algorithms with adaptive learning rates

Algorithm 12: Stochastic Gradient Descent

Given a dataset D, a learning rate $\eta \geq 0$ and the initial parameters $\theta^{(1)}$, the algorithm computes the gradient descent using the backpropagation and random sampling.

function StochasticGradientDescent($\eta, \theta^{(1)}$)
  $k = 1$
  while the stopping criterion is not met **do**
    Randomly sample a minibatch $\{x^{(1)}, \ldots, x^{(m)}\} \subset D$
    Compute $g = \frac{1}{m} \nabla_\theta \sum_{i=1}^{m} L(f(x^{(i)}, \theta^{(k)}), t^{(i)})$ ▷ Computed through backprop.
    $\theta^{(k+1)} = \theta^{(k)} - \eta g$
    $k = k + 1$
  end while
  Return $\theta^{(k)}$
end function

The stopping condition usually used in stochastic gradient descent is a treshold parameter for improvement or a fixed number of iterations. In some advanced models, the hyperparameter η changes according to some rule through iterations, where for the first τ iteration ($k \leq tau$) we have:

$$\eta^{(k+1)} = \left(1 - \frac{k}{\tau}\right) \eta^{(k)} + \frac{k}{\tau} \eta^{(\tau)}$$

while for all other iterations ($k > \tau$) we have $\eta^{(k)} = \eta^{(\tau)}$. This partially removes the overhead of having to find a good static hyperparameter η – the algorithm may diverge if η is too large, while it may be too slow if η is too small.

In SDG with momentum, an additional value called **momentum** is used to (potentially) accelerate learning. The usefulness of momentum comes in play due to how stochastic gradient can largely vary through the iterations.

Algorithm 13: SDG with momentum

Given a dataset D, a learning rate $\eta \geq 0$, a momentum $\mu \in [0, 1)$ and the initial parameters $\theta^{(1)}$, the algorithm computes the gradient descent using the backpropagation and random sampling.

function SDG_MOMENTUM($\eta, \theta^{(1)}$)
  $k = 1$
  $v^{(1)} = 0$ ▷ v acts as "velocity"
  while the stopping criterion is not met **do**
    Randomly sample a minibatch $\{x^{(1)}, \ldots, x^{(m)}\} \subset D$
    Compute $g = \frac{1}{m} \nabla_\theta \sum_{i=1}^{m} L(f(x^{(i)}, \theta^{(k)}), t^{(i)})$ ▷ Computed through backprop.
    Compute $v^{(k+1)} = \mu v^{(k)} - \eta g$
    $\theta^{(k+1)} = \theta^{(k)} - v^{(k+1)}$
    $k = k + 1$
  end while
  Return $\theta^{(k)}$
end function

In some cases, the momentum is applied before computing the gradient. This is variant is called **Nesterov momentum**. Sometimes, this variant improves the convergence rate.

$$\overline{\theta} = \theta^{(k)} + \eta v^{(k)}$$

$$g = \frac{1}{m} \nabla_{\overline{\theta}} L(f(x^{(i)}, \overline{\theta}), t^{(i)})$$

Other algorithms can have an adaptive learning rate: based on the analysis of the gradient of the loss function it is possible to determine, at any step of the algorithm, whether the learning rate should be increased or decreased. Some examples include:

- AdaGrad
- RMSProp
- Adam

As with other ML approaches, regularization is an important feature to reduce overfitting (generalization error). For FNN, we have several options and they can even be applied together:

- *Parameter norm penalties*: we add a regularization term E_{reg} to the cost function:

$$E_{\text{reg}}(\theta) = \sum_j |\theta_j|^q$$

$$\overline{J}(\theta) = J(\theta) + \lambda E_{\text{reg}}(\theta)$$

- *Dataset augmentation*: we generate additional data and include them in the dataset, usually applying data transformations (e.g. image rotation, scaling, varying illumination conditions, ...) or adding noise.
- *Early stopping*: we stop iterations early to avoid overfitting to the training dataset, usually using cross-validation do determine the best iteration to stop on.
- *Parameter sharing*: we impose that the subsets of model parameters are equal. This also improves memory storage.
- *Dropout*: we randomly ignore network units with some probability p at each iteration.

6.3 Convolutional Neural Networks

6.3.1 Convolution

Up to now, we treated inputs as general feature vectors. In some cases, inputs have a special structure that can be exploited to get improved results. In general, **Convolutional Neural Networks (CNN)** work with any type of structure based on *signals*, numerical representations of physical quantities. Deep learning can be directly applied on signals by using suitable operators. In particular, signals have to be represented as a N-dimensional *tensor*.

- An audio file is just a variable length 1D vector, i.e. a 1D tensor
- An image file is just a multi-channel 2D matrix, i.e. a 3D tensor
- A video file is just a sequence of multi-channel 2D matrices sampled through time, i.e. a 4D tensor

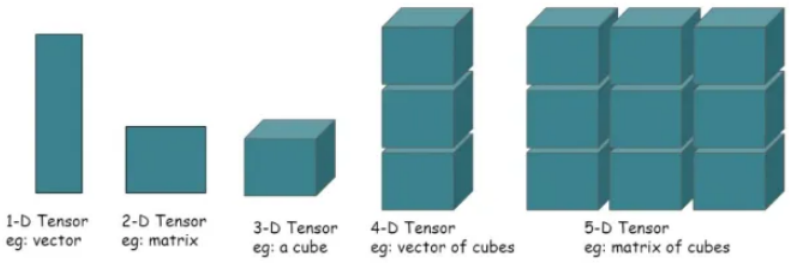

Figure 6.9: Tensors with different dimensions.

CNNs are based on **convolution**, a mathematical operation on two functions f, g that produces a new function $f * g$. For continuous functions, convolution is defined as the integral of the product of the two functions after one is reflected about the y-axis and shifted. The integral is evaluated for all values of shift, producing the convolution function. The function that gets shifted is called **kernel** of the convolution. For discrete functions, the same definition holds, but the integral becomes an infinite sum.

Definition 20: Convolution

Given two continuous real functions f, g, the convolution $f * g$ is defined as:

$$(f * g)(x) = \int_{-\infty}^{+\infty} f(a) \cdot g(x - a)\, da$$

For discrete functions, convolutions is defined as:

$$(f * g)(x) = \sum_{a=-\infty}^{+\infty} f(a) \cdot g(x - a)$$

Another mathematical operation strictly related to convolution is **cross-correlation**. This operation is similar to convolution, where none of the two functions gets flipped. For this reason, cross-correlation is often known as *sliding dot product*. By manually flipping one of the two functions, cross-comparison can be used to compute convolution.

Definition 21: Cross-correlation

Given two continuous real functions f, g, the cross-correlation $f \star g$ is defined as:

$$(f \star g)(x) = \int_{-\infty}^{+\infty} f(x + a) \cdot g(a)\, da$$

For discrete functions, convolutions is defined as:

$$(f \star g)(x) = \sum_{a=-\infty}^{+\infty} f(x + a) \cdot g(a)$$

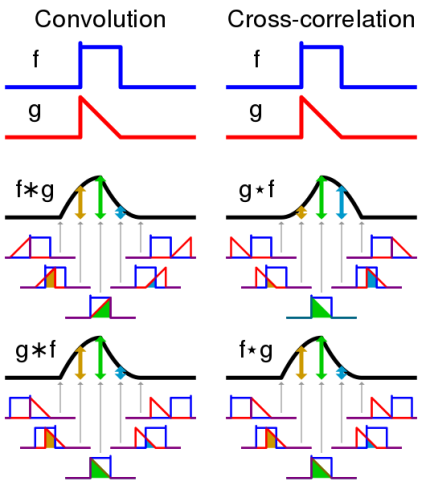

Figure 6.10: Comparison between correlation and cross-comparison. Convolution is commutative, while cross-comparison is not.

We observe that cross-correlation is the function that is actually implemented in machine learning libraries, due to it being more efficient to compute through dynamic programming. Since tensors can be viewed as linear applications, convolution and cross-correlation also apply on them. For 2 dimensions, the *limited* discrete 2D convolution of I and K is defined as:

$$(I * K)(i, j) = \sum_{m \in S_1} \sum_{n \in S_2} I(m, n) K(i - m, j - n)$$

where S_1, S_2 are two finite sets – hence the *limited* in the name of the operation. In other words, during the computation, the kernel slides in both directions.

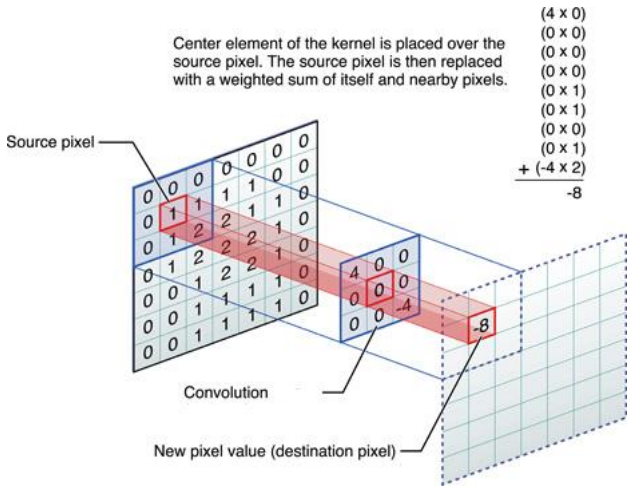

Figure 6.11: Example of 2D convolution of a 2D image and a 2D kernel.

Similarly, 3D convolution is defined as:

$$(I * K)(i, j, k) = \sum_{m \in S_1} \sum_{n \in S_2} \sum_{h \in S_3} I(m, n, h) K(i - m, j - n, k - h)$$

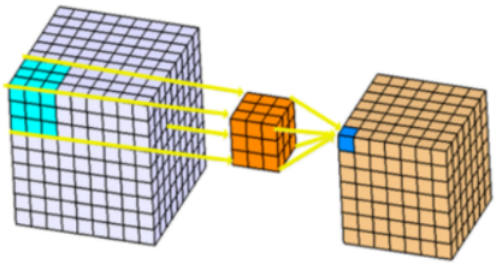

Figure 6.12: Example of 3D convolution of a 3D image and a 3D kernel.

We observe that the name 1D/2D/3D convolution does not refer to the dimensions of the input and the kernel. Instead, they refer to the number of directions where the kernel slides. For instance, 2D convolutions can be also applied on a 3D input and kernel:

$$(I * K)(i, j) = \sum_{m \in S_1} \sum_{n \in S_2} \sum_{h \in S_3} I(m, n, h) K(i - m, j - n, h)$$

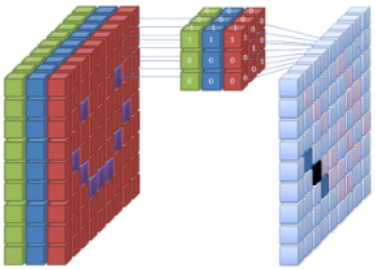

Figure 6.13: Example of computation in 2D convolution of a 3D image and a 3D kernel.

Convolution can be applied in many ways. To make things simpler, we introduce some terminology to easily describe these processes.

- *Input size*: $(w_{\text{in}} \times h_{\text{in}} \times d_{\text{in}})$ are the dimensions of the input
- *Kernel size*: $(w_{\text{k}} \times h_{\text{k}} \times d_{\text{k}})$ are the dimensions of the kernel
- *Feature map* (or *Depth slice*): output of the convolution between an input and a kernel
- *Depth*: (d) is the number of kernels used
- *Padding*: (p) is the number of fillers for outer rows or columns (typically set to 0)
- *Stride*: (s) is the number of pixels the kernel moves each time it slides (when set to 1, no cell of the image is skipped)
- *Receptive field*: region in the input space that a particular feature is affected by

When the **multiple kernels** are used, convolution produces multiple feature maps. For instance, if we have an input of size 32×32 and we apply 2D convolution on 1 kernel with dimensions $5 \times 5 \times 1$ kernel, we get a feature map of size 28×28. Instead, if we apply 6 kernels with dimensions $5 \times 5 \times 3$, each kernel is individually applied on a copy of the input, producing 6 feature maps of size 28×28. These 6 feature maps are then represented by a $28 \times 28 \times 6$ tensor. In general, if we have an input of size

$$w_{\text{in}} \times h_{\text{in}} \times d_{\text{in}}$$

and d_{out} kernels of size

$$w_{\text{k}} \times h_{\text{k}} \times d_{\text{k}}$$

where $d_{\text{k}} = \text{in}$, the output will always be a 3D tensor of size

$$w_{\text{out}} \times h_{\text{out}} \times d_{\text{out}}$$

where w_{out} and h_{out} are computed according to *stride* and *padding* (we'll see how in the next sections). For 3D convolutions, the constraint becomes $d_{\text{in}} > d_{\text{k}}$ ad the output is a 4D tensor of size $w_{\text{out}} \times h_{\text{out}} \times z_{\text{out}} \times d_{\text{out}}$, where z_{out} is also computed according to *stride* and *padding*.

6.3.2 Convolution in neural networks

As the name suggests, CNNs are just FNNs that use convolution in one or more layers. The learning phase finds which kernels are optimal for each layer **convolutional layer**. Each of these layers is formed by three sub-layers, called *stages*:

1. **Convolution stage**: application of 2D convolution between input and kernel
2. **Detector stage**: application of non-linear activation function on the output of the previous stage
3. **Pooling**: reduction in size of the output of the previous stage

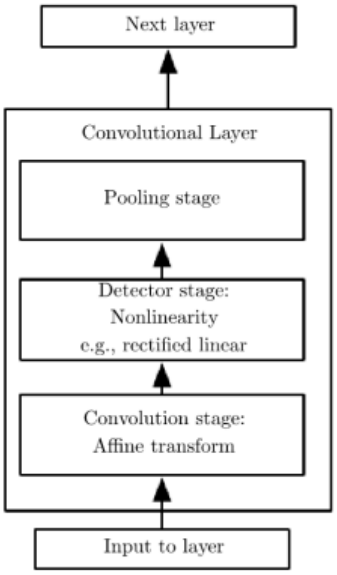

Figure 6.14: A convolutional layer.

The first stage of a convolutional layer applies 2D convolution. We observe that in this stage, 2D convolution is actually applied through cross-correlation:

$$(I * K)(i, j) = \sum_{m \in S_1} \sum_{n \in S_2} I(i + m, j + n) K(m, n)$$

This process is very efficient in terms of memory requirements and generalization accuracy, in particular **sparse connectivity** and **parameter sharing**. In the former (also called *sparse interactions*), the outputs depend only on a few inputs. In other words, instead of having a completely connected network, we have a sparse network. To achieve this, we use a kernel that is much smaller than input. This property is particularly useful signals such as images: we expect that each pixel has more influence on other pixels that are close ot it, while far away pixels will be not be influenced by it. In this case, we would prefer a sparse connectivity patter that connects pixels that are close to each other to the same units.

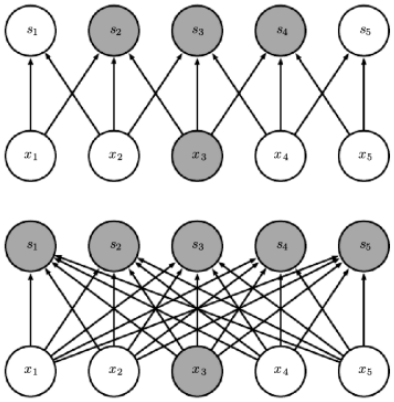

Figure 6.15: Example of a sparse connectivity CNN (top) and a fully connected CNN (bottom). In the former, the input x_3 influences a restricted set of units of the next layer.

In parameter sharing, instead, we learn only one set of parameters (for the kernel) that is shared to all the units. In other words, some weights are shared by all neurons in a particular feature map. This helps to reduce the number of parameters in the whole system, making it computationally cheap.

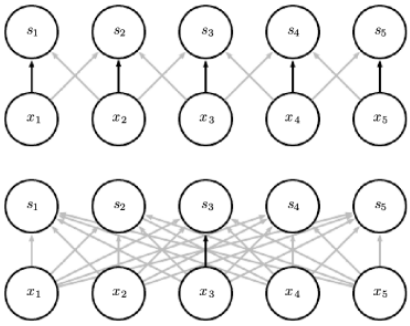

Figure 6.16: Example of a parameter sharing in CNN (top) and non-parameter sharing (bottom). The black arrows represent the parameters shared by all inputs.

Another important operation for this stage is **padding**. This operation controls what happens when the kernel is applied to values that are on the "border" of the input. For instance, suppose that we have an 1D input of size 32 and a 1D kernel of size 5. By applying 1D convolution, we would first apply the kernel to the first 5 values of the input and then slide by one pixel (assuming stride equal to 1). By repeating this operation for the whole input, we would obtain a 1D feature map of size 28 (the number of possible slides). In general, if the input has size w_{in} and the kernel has size w_{k}, we get a feature map of size $w_{\text{in}} - (w_{\text{k}} - 1)$.

With padding, instead, we extend the "borders" of the input in order to get more applications of the kernel. In particular, when the padding value p is equal to $\left\lfloor \frac{w_{\text{k}}}{2} \right\rfloor$, the feature map will have the same size of the input. This is called *same padding*. When the padding value p is equal to 0, instead, we say that we have *valid padding* – due to how only valid applications are done.

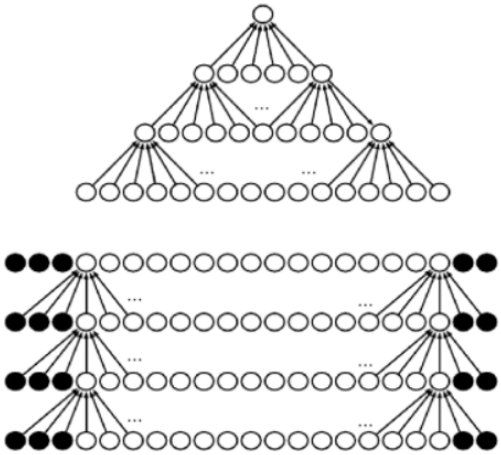

Figure 6.17: Example of CNN without padding (top) and CNN with padding (bottom).

The second stage of a convolutional layer, i.e. the **detector** stage, is nothing more than the traditional application of the activation function of a typical FNN. A more interesting stage is the **pooling stage**. This stage is in charge of *reducing* the size of the output before feeding it to the next layer.

Formally, pooling implements invariance to local translations. When *max pooling* is applied, the stage returns the maximum value of a rectangular region. When *average pooling* is applied, instead, the stage returns the average value of a rectangular region. When applied with a *stride* value larger than 1, the pooling stage reduces the size of the output layer.

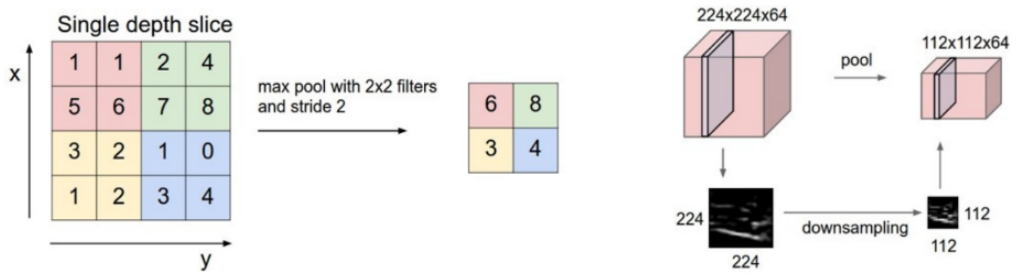

Figure 6.18: Example of 2×2 max pooling being applied with stride 2. This stage introduces downsampling as a byproduct.

Given an input of size $w_{\text{in}} \times h_{\text{in}} \times d_{\text{in}}$, d_{out} kernels of size $w_{\text{k}} \times h_{\text{k}} \times d_{\text{k}}$, a stride value s and a padding p, the dimensions $w_{\text{out}}, h_{\text{out}}$ of the d_{out} output feature maps are given by:

$$w_{\text{out}} = \frac{w_{\text{in}} - w_{\text{k}} + 2p}{s} + 1$$

$$h_{\text{out}} = \frac{h_{\text{in}} - h_{\text{k}} + 2p}{s} + 1$$

$$d_{\text{out}} = \text{Number of kernels}$$

Since the only trainable element in the layer is the kernel, the total number of trainable parameters of the convolutional layer is:

$$|\theta| = \underbrace{w_{\text{k}} \cdot h_{\text{k}} \cdot d_{\text{in}} \cdot d_{\text{out}}}_{\text{kernel weights}} + \underbrace{d_{\text{out}}}_{\text{bias}}$$

By applying these three stages, each convolutional layer inside a CNN applies reduces the size of a single feature map while also increasing the number of such feature maps. In other words, we sequentially trade size for depth until we reach a **flattened layer**, i.e. a very deep 1D tensor. Once this layer has been reached the network proceeds with a sequence of traditional fully connected layers that classify the extracted flattened layer.

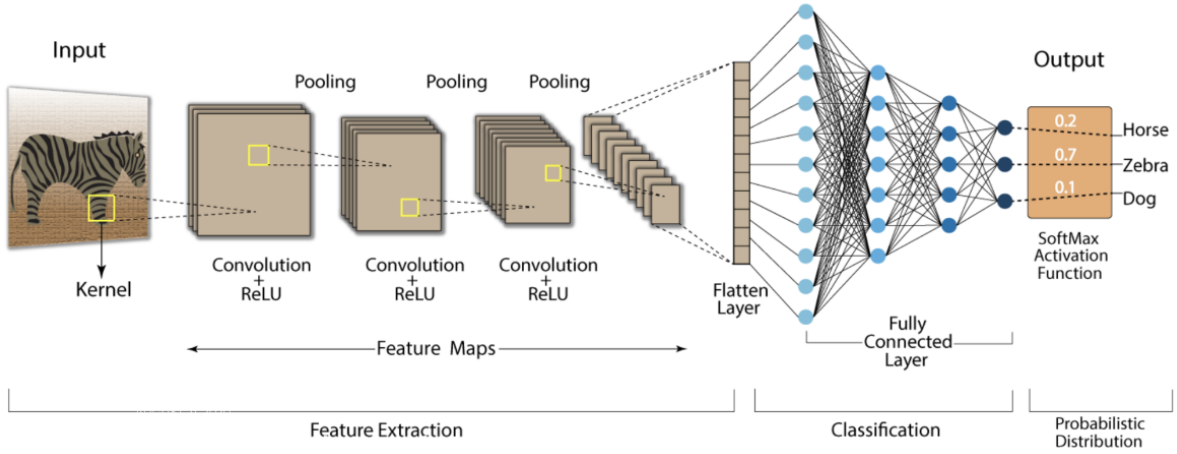

Figure 6.19: Typical structure of a CNN with multiple convolutional layers.

The recent success of CNNs is yielded by continuous theoretical advancements in neural networks. However, they also have some downsides: they require massive parallel computing capacities to speed up computations (GPUs/TPUs), very large training sets (ImageNet/MS COCO) and developing frameworks (Keras, Theano, Tensorflow, PyTorch, ...).

6.3.3 Transfer learning

Convolutional neural networks are, nowadays, among the most used types of neural network. Over the years, researchers have produced huge datasets (such as ImageNet) that can satisfy the necessity of these models. However, these dataset are clearly specific to some tasks. For new tasks, however, we would require new specific datasets to be created. This bruteforce approach is clearly infeasible: it's impossible to sample a huge dataset for every possible problem. To fix this issue, we use the concept of **transfer learning**.

Let D be a *domain* defined by datapoints $x_i \in X$ distributed according to $\mathcal{D}(x)$. Let T be a *learning task* defined by labels $y \in Y$, a target function $f : X \to Y$ be a target function and a distribution $P_{\mathcal{D}}(y \mid x)$.

Given a target domain D_T and a target learning task T_T, we want to improve the learning of $f_T : X_T \to Y_T$ using the knowledge in a source domain D_S and a source learning task T_S. In other words, we want to use the training for a source function $f_S : X_S \to Y_S$ to improve the training of a target function $f_T : X_T \to Y_T$.

In order for this process to work, there must be some intersection between the labels Y_S, Y_T and the inputs X_S, X_T. If the intersection is too small, learning capabilities cannot be transferred.

The first solution for transfer learning is **fine tuning**. In this case, for the target problem we use the same network architecture of the pre-trained model up to some layer, while also "copying" their parameters. The copied layers cannot are fixed (*frozen weights*) and they cannot be trained on the new model. To adapt the model for the new task, the last layer of the architecture is usually slightly changed.

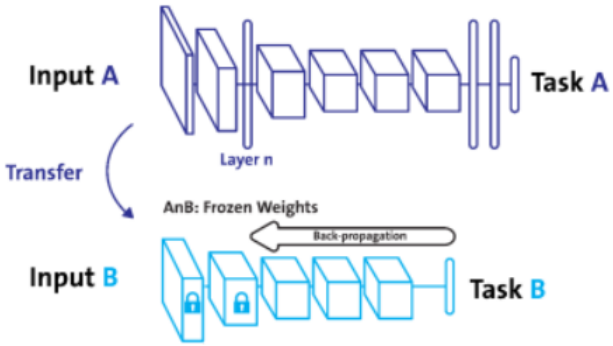

Figure 6.20: Example of fine tuning.

The second solution for transfer learning is **CNNs as feature extractors**. In this case, we extract the features of a specific layer of the source CNN (usually flattened or dense layers) and then collect the extracted features x' of the training/validation split and associate corresponding labels t in a new dataset $D' = \{(x'_i, t_i) \mid i \in [N]\}$. After this process, we train a new classifier C' (another ANN, an SVM, ...) using the dataset D' and classify the extracted features of the test set using C'. This process has a huge advantages (there is no need to train a new CNN) but also huge disadvantages (features cannot be modified, hence source and target should be as compatible as possible).

7 Unsupervised learning

7.1 K-means

In previous chapter, we have strictly focused on supervised learning, i.e. learning methods where the dataset contains a target value for each datapoint. In **unsupervised learning**, instead, the dataset has no target values. In other words, the dataset is always of the form $D = \{x_i \mid i \in [N]\}$.

A simple unsupervised learning model is the **Gaussian Mixture Model (GMM)**. Here, the dataset is generated by a mixed probability distribution is formed by the weighted average of K different gaussian distributions.

$$\Pr[x] = \sum_{k=1}^{K} p_k \mathcal{N}(x; \mu_k, \Sigma_k)$$

where p_k, μ_k and Σ_k are respectively the prior probability, the mean and the covariance matrix. Each instance x_i of the dataset is generated by:

1. Randomly sample one of the Gaussians according to prior probabilities
2. Generate an instance at random according to the chosen Gaussian distribution

In this model, the learning phase is reduced to generating such K Gaussians from a dataset in order to produce new samples. To estimate these parameters, we use the **K-means** algorithm, which computes K means of data generated from K Gaussian distributions.

Algorithm 14: K-means

Given a dataset $D = \{x_i \mid i \in [N]\}$ and a value K, the algorithm computes K means for K Gaussian distributions.

function K-MEANS(D, K)
  Initialize K random centroids for K clusters
  do
    Assign each $x_i \in D$ to the cluster with the closest centroid
    Recompute the centroids of the clusters
  while at least one sample switches cluster
  Return the K centroids of the clusters.
end function

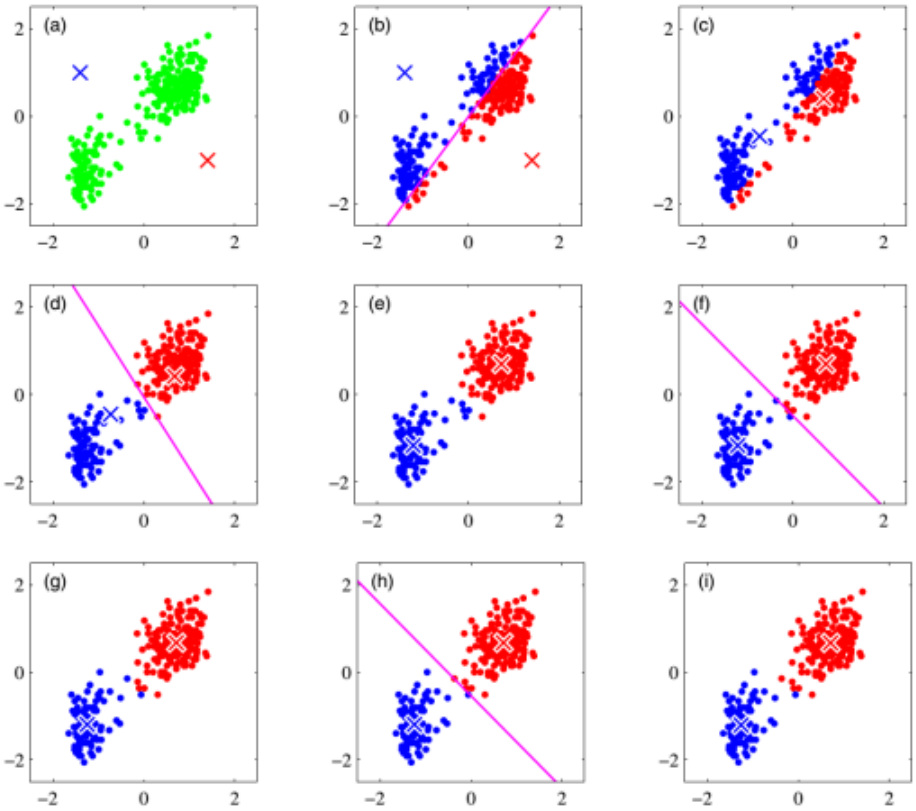

Figure 7.1: Execution of the K-means algorithm with $K = 2$

The convergence of the K-means algorithm will always occur if the following conditions are satisfied:

1. For each switch, the sum of distances from each training sample to that training sample's group centroid is decreased.
2. There are only finitely many partitions of the training examples into K clusters.

We also observe that the K-means algorithm, although simple, has many downsides:

- The choice of the hyperparameter K is critical and must be determined beforehand using intuition
- The algorithm is sensitive to initial condition (local optimum) when few data is available
- The algorithm is not robust to outliers since data that is very far from may pull the centroids away.
- The result is always a circular cluster shape due to it being based on distance

7.2 Estimation Maximization

A more advanced algorithm called **Estimation Maximization (EM)** is capable of not only computing the means of the K Gaussians, but also their covariance matrices and their prior probabilities. Before proceeding, we have to slightly modify our GMM model. In particular, we introduce a new vector $z = \begin{bmatrix} z_1 & \dots & z_K \end{bmatrix}^T$ using a 1-out-of-K encoding where $z_k \in \{0, 1\}$, where $\Pr[z_k = 1] = p_k$. Thus, we have that:

$$\Pr[z] = \prod_{k=1}^{K} p_k^{z_k}$$

For a given value z, we have that $\Pr[x \mid z_k = 1] = \mathcal{N}(x; \mu_k, \Sigma_k)$. Thus, we get that:

$$\Pr[x \mid z] = \prod_{k=1}^{K} \mathrm{N}(x; \mu_k, \Sigma_k)^{z_k}$$

Since z is a 1-out-of-K encoding and $\Pr[z_k = 1] = p_k$, using total probability we have

$$\Pr[x] = \sum_{z} \Pr[z] \Pr[x \mid z] = \sum_{k=1}^{K} p_k \mathcal{N}(x; \mu_k, \Sigma_k)$$

In other words, the GMM distribution $\Pr[x]$ can be seen as the marginalization of a distribution $\Pr[x \mid z]$ over z. Given the dataset $D = \{x_i \mid i \in [N]\}$, each datapoint x_i is associated to a corresponding unknown z_i, where $z_{i,k} = 1$ if and only if x_i is sampled from the Gaussian k. These vectors z_i are called **latent variables**. Unlike observable variables, which can be directly measured or observed, latent variables represent underlying factors or constructs that are not directly observable but are inferred from other measurable variables.

For each $z_{i,k}$, we define $\gamma(z_{i,k})$ as:

$$\gamma(z_{i,k}) = \Pr[z_{i,k} = 1 \mid x_i] = \frac{\Pr[z_{i,k}] = 1 \cdot \Pr[x_i \mid z_{i,k} = 1]}{\Pr[x_i]} = \frac{p_k \mathcal{N}(x_i; \mu_k, \Sigma_k)}{\sum_{j=1}^{K} p_j \mathcal{N}(x_i; \mu_j, \Sigma_j)}$$

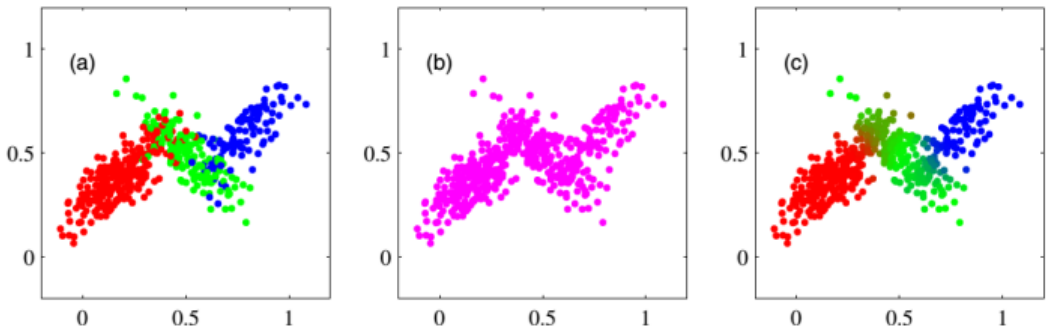

Figure 7.2: (a) $\Pr[x, z]$ with 3 latent variables z (red, gree, blue) – (b) $\Pr[x]$ marginalized distribution – (c) $\gamma(z_{i,k})$ posterior distribution

Consider the maximum likelihood of the GMM:

$$\arg\max_{p,\mu,\Sigma} \ln \Pr[X \mid p, \mu, \Sigma]$$

The solution of this optimization problem is given by:

$$p_k = \frac{N_k}{N}$$

$$\mu_k = \frac{1}{N_k} \sum_{i=1}^{N} \gamma(z_{i,k}) x_i$$

$$\Sigma_k = \frac{1}{N_k} \sum_{i=1}^{N} \gamma(z_{i,k})(x_i - \mu_k)(x_n - \mu_k)^T$$

where $N_k = \sum_{i=1}^{N} \gamma(z_{i,k})$. To efficiently compute these solutions, we use the EM algorithm. This algorithm is divided into two steps: the *Estimation step* and the *Maximization step*. The former computes $\gamma(z_{i,k})$ given current values of p_k, μ_k, Σ_k, while the latter computes p_k, μ_k, Σ_k given the current values of $\gamma(z_{i,k})$. By repeatedly applying these operations, the algorithm converges to an estimation of the optimal solution.

Algorithm 15: Estimation Maximization (EM)

Given a dataset $D = \{x_i \mid i \in [N]\}$ and a value K, the algorithm computes the means, covariance matrices and prior probabilities for K Gaussian distributions.

function EM(D, K)
  Initialize $p_k^{(0)}, \mu_k^{(0)}, \Sigma_k^{(0)}$ for each $k \in [K]$
  Set $t = 0$
  while the terminating condition is not met **do**
    Compute the E-Step for each $k \in [K]$ and for each $i \in [N]$:

$$\gamma(z_{i,k})^{(t+1)} = \frac{p_k^{(t)} \mathcal{N}(x_i; \mu_k^{(t)}, \Sigma_k^{(t)})}{\sum_{j=1}^{K} p_j^{(t)} \mathcal{N}(x_i; \mu_j^{(t)}, \Sigma_j^{(t)})}$$

    Set $N_k^{(t+1)} = \sum_{i=1}^{N} \gamma(z_{i,k})^{(t+1)}$
    Compute the M-Step for each $k \in [K]$ for each $i \in [N]$:

$$p_k^{(t+1)} = \frac{N_k^{(t+1)}}{N} \qquad \mu_k^{(t+1)} = \frac{1}{N_k^{(t+1)}} \sum_{i=1}^{N} \gamma(z_{i,k})^{(t+1)} x_i$$

$$\Sigma_k^{(t+1)} = \frac{1}{N_k^{(t+1)}} \sum_{i=1}^{N} \gamma(z_{i,k})^{(t+1)} (x_i - \mu_k^{(t+1)})(x_i - \mu_k^{(t+1)})^T$$

    Set $t = t + 1$
  end while
  Return $p_k^{(t)}, \mu_k^{(t)}, \Sigma_k^{(t)}$ for each $k \in [K]$
end function

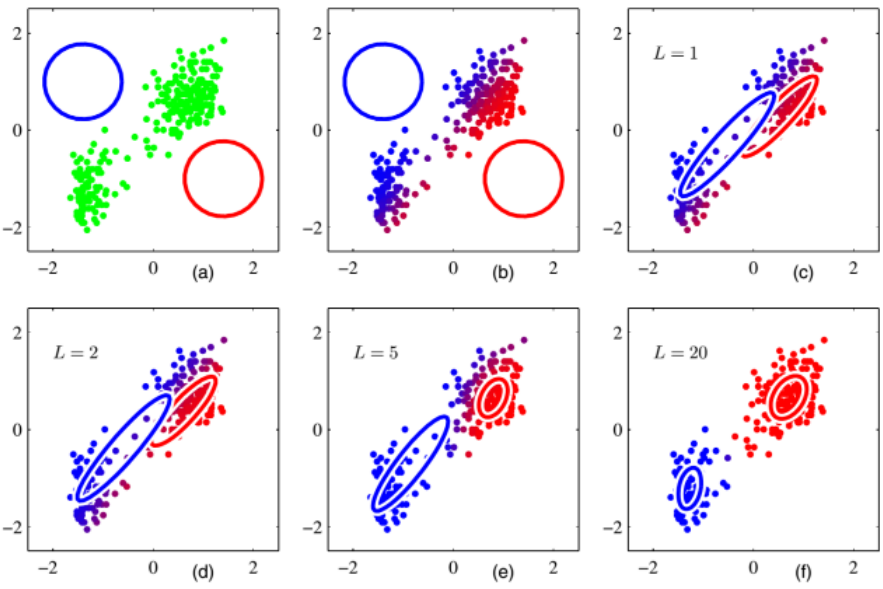

Figure 7.3: Execution of the EM algorithm with $K = 2$.

The EM algorithm converges to the local maximum likelihood, providing estimates of the latent variables $z_{i,k}$. In some sense, it can be viewed as the extended version of the K-means algorithm, where probabilistic assignments to a cluster $z_{i,k}$.

We also observe that the EM can also be generalized to other distributions. In the general form, we have some observable data $X = \{x_1, \ldots, x_N\}$, some unobserved latent values $Z = \{x_1, \ldots, z_N\}$ and a parametrized probability distribution $\Pr[Y \mid \theta]$ where $Y = \{y_1, \ldots, y_N\}$ is the full data $y_i = \langle x_i, z_i \rangle$ and θ are the parameters. The problem asks to determine the optimal parameters θ^* that locally maximize $\mathbb{E}[\ln \Pr[Y \mid \theta]]$. This general version has many uses, in particular in unsupervised clustering, Bayesian networks and Hidden Markov models.

In its general form, the EM algorithm defines a likelihood function $Q(\theta' \mid \theta)$ on variables $Y = X \cup Z$, using observed X and current parameters θ to estimate Z:

1. In the Estimation step, we compute $Q(\theta' \mid \theta)$ using the current hypothesis θ and observed data X to estimate the probability distribution over Y

$$Q(\theta' \mid \theta) = \mathbb{E}[\ln \Pr(T \mid \theta') \mid \theta, X]$$

2. In the Maximization step, we replace the hypothesis θ with the hypothesis θ' that maximizes Q

$$\theta := \arg\max_{\theta'} Q(\theta' \mid \theta)$$

8

Dimensionality reduction

8.1 Continuous latent variables

Consider the following 64×57 image inside the USPS dataset, representing the digit 6. This image can be mathematically described as an element of the set $\mathbb{R}^{64\times57\times1}$ – the last dimension is the number of channels.

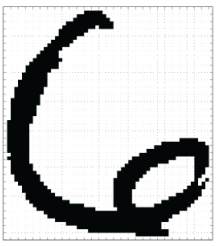

Figure 8.1: Example of a digit image inside the USPS dataset.

In the data space $\mathbb{R}^{64\times57\times1}$ of the USPS dataset (or a subset of it), we may have other images with random black pixels. In the context of digit recognitions, images similar to the one below represent no real information. This means that large part of the data space is actually useless.

Figure 8.2: Example of another image in the same data space.

Consider now the dataset formed of all the rotations without translation of the image in Figure 8.1. Each point in this data set is clearly also a point in $\mathbb{R}^{64\times57\times1}$. However, across this dataset there is only **one degree of freedom** of variability for the samples, corresponding to the rotation value. This means that the samples of the dataset actually live on a manifold inside $\mathbb{R}^{64\times57\times1}$ whose **intrinsic dimensionality** is one.

In this context, the rotation value is a latent variable since we can observe only the image vectors and we are not told which values of the rotation variable were used to create them. However, we know that the samples in this dataset are somehow related to each other. To get a good representation of this dataset inside the data space, we're interested in finding the best linear transformation from our original data space to a new coordinate system such that the directions, called **principal components**, capture the largest *variation* in the data and can be easily identified by projecting the samples on the new coordinate system.

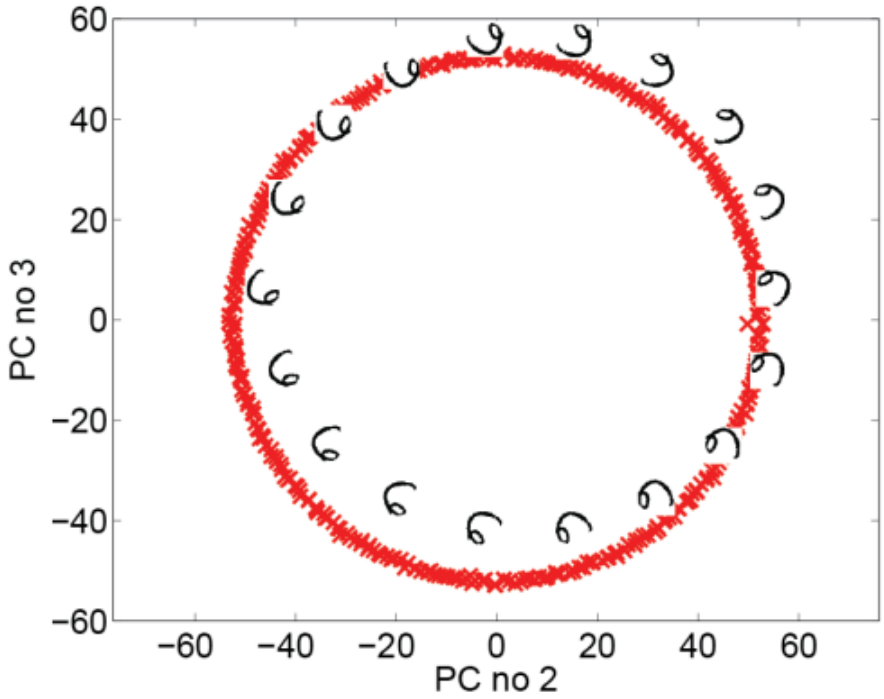

Figure 8.3: Representation of the dataset projected onto the optimal principal subspace.

From the figure above, we notice that the rotation dataset assumes a circle-like structure when projected onto the principal subspace. This is actually pretty common for data that has "structure", i.e. strong intrinsic relations.

8.2 Principal Components Analysis

Principal Component Analysis (PCA) focuses on finding the principal components inside a data that define the optimal *principal subspace*. PCA is used not only for dimensionality reduction, but also for (lossy) data compression, data visualization and feature extraction.

Given a dataset $D = \{x_i \mid i \in [N]\} \subseteq \mathbb{R}^d$, we want to maximize the data variance after projecting it to some vector u_1. In particular, u_1 is a unit vector, meaning that $\|u\| = u_1^T u_1 = 1$. For each point $x_i \in D$, the projection on u_1 is given by $x_i^T u_1$.

Consider now the mean value of all the data points $\overline{x} = \frac{1}{N}\sum_{i=1}^{N} x_i$. We observe that the mean of the projected points is equal to the projection of the mean, i.e. $\overline{x}^T u_1$, while the variance of the projected points is given by:

$$\frac{1}{N}\sum_{i=1}^{N}(u_1^T x_i - u_1^T \overline{x})^2 = u_1 S u_1$$

where S is the covariance matrix of the dataset:

$$S = \frac{1}{N}\sum_{i=1}^{N}(x_i - \overline{x_i})(x_i - \overline{x_i})^T = \frac{1}{N}X^T X$$

with X being the data-centered matrix of the dataset:

$$X = \begin{bmatrix} (x_1 - \overline{x})^T \\ \vdots \\ (x_N - \overline{x})^T \end{bmatrix}$$

To maximize the projected variance, we have to solve the following optimization problem:

$$\arg\max_{u_1} u_1^T S u_1$$

subject to $u_1^T u_1 = 1$. Using a Lagrange multiplier λ_1, this problem is equivalent to the unconstrained maximization of:

$$\arg\max_{u_1} u_1^T S u_1 + \lambda_1(1 - u_1^T u_1)$$

By setting to zero the derivative with respect to u_1 in order to solve the problem, we obtain that:

$$Su_1 - \lambda_1 u_1 = 0 \implies Su_1 = \lambda_1 u_1$$

In other words, u_1 must be an eigenvector of S and λ_1 is its eigenvalue. Moreover, using the fact that $u_1^T u_1 = 1$, we also get that:

$$u_1^T S u_1 = u_1^T \lambda_1 u_1 \implies u_1^T S u_1 = \lambda_1$$

which is exactly the variance of the samples after the projection. In conclusion, the variance is maximal when u_1 is the eigenvector of S corresponding to the largest eigenvalue λ_1.

After finding the first principal component u_1, if more dimensions are wanted then the whole process can be repeated by picking the next eigenvalues with the largest value in order to obtain the other principal components. In particular, we recall that eigenvectors associated with different eigenvalues are always orthogonal to each other, meaning that the basis $u_1, \ldots, u_m$ produced by applying the process on the m largest eigenvalues is orthonormal.

Algorithm 16: Principal Component Analysis (PCA)

Given a dataset $D = \{x_i \mid i \in [N]\} \subseteq \mathbb{R}^d$ and a value m, where $m < d$, the algorithm return and orthonormal basis for the principal subspace of dimension m.

function PCA(D, m)
  Compute the mean $\overline{x}$ of D
  Compute the data-centered matrix X of D using $\overline{x}$
  Compute the covariance matrix matrix S of D using X
  Find the eigenvalues of S
  Find the eigenvectors $u_1, \ldots, u_m$ of the m largest eigenvalues of S
  Return $u_1, \ldots, u_m$
end function

Given the principal components $u_1, \ldots, u_m$, the linear transformation that maps each original sample $x_i \in D$ to its representation in the principal subspace is given by:

$$\phi : D \to \text{span}(u_1, \ldots, u_m) : x_i \mapsto \begin{bmatrix} x_i^T u_1 \\ \vdots \\ x_i^T u_m \end{bmatrix}$$

We observe that the projection over the new coordinate system looses some information. Hence, if the dimensions chosen for the principal subspace is too small, the dataset will become unusable. To reduce the error, we consider another formulation of the problem. Given the original d dimensional dataset D, let $u_1, \ldots, u_d$ be a complete orthonormal base, i.e. such that:

$$u_j^T u_j = \delta_{i,j} = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{otherwise} \end{cases}$$

In other words, each vector of the basis has exactly one component set to 1, while all the others are set to 0. No vectors share the same 1-component. Given $x_i \in D$, let $\alpha_1, \ldots, \alpha_d$ be the coefficient of the linear combination of x_i over $u_1, \ldots, u_d$. We notice that:

$$x_i = \sum_{k=1}^{d} \alpha_k u_k = \sum_{k=1}^{d} (x_i^T u_k) u_k$$

Our goal is to approximate x_i using a lower dimensional representation over a restricted subset of m vectors taken from $u_1, \ldots, u_d$.

For each sample x_i, let $\widetilde{x_i}$ be defined as:

$$\widetilde{x_i} = \sum_{k=1}^{m} z_{i,k} u_k + \sum_{h=m+1}^{d} b_h u_h$$

where the terms $z_{i,k}$ depend on x_i and the terms b_h are shared among all samples. We evaluate the approximation error using MSE:

$$J = \frac{1}{N} \sum_{i=1}^{N} \|x_i - \widetilde{x_i}\|^2$$

Minimizing with respect to $z_{i,k}$ and b_h, we get that $z_{i,k} = x_i^T u_k$ for $1 \leq k \leq m$ and $b_h = \widetilde{x_i}^T u_k$. for $m+1 \leq h \leq d$. In other words, for each x_i the best estimation of $\widetilde{x_i}$ is given by the first m components of x_i and the last $d-m$ components shared among all samples. Hence, we get that:

$$x_i - \widetilde{x_i} = \sum_{h=m+1}^{d} (x_i^T u_h) u_h + \widetilde{x_i}^T u_h = \sum_{h=m+1}^{d} ((x_i - \widetilde{x_i}^T) u_h) u_h$$

meaning that the overall approximation error becomes:

$$J = \frac{1}{N} \sum_{i=1}^{N} \sum_{h=m+1}^{d} \|x_i - \widetilde{x_i}\|^2 (x_i^T u_h - \widetilde{x_i}^T u_h)^2 = \sum_{h=m+1}^{d} u_h^T S u_h$$

By minimizing the error using a Lagrange multiplier λ_1, this problem is equivalent to the unconstrained maximization of:

$$\underset{u_{m+1},\ldots,u_d}{\arg\min} \sum_{h=m+1}^{d} u_h^T S u_h + \lambda_h (1 - u_h^T u_h)$$

Setting the derivative over u_h to zero for each index, we have that:

$$S u_h - \lambda_h u_h = 0 \implies S u_h = \lambda_h u_h$$

meaning that each u_h is the eigenvector of S associated with the eigenvalue λ_h. Simplifying, we get that:

$$\underset{u_{m+1},\ldots,u_d}{\arg\min} \sum_{h=m+1}^{d} u_h^T S u_h + \lambda_h (1 - u_h^T u_h) = \underset{u_{m+1},\ldots,u_d}{\arg\min} \sum_{h=m+1}^{d} \lambda_h$$

Hence, the principal components $u_{m+1}, \ldots, u_d$ that minimize the error are the eigenvectors associated with the smallest eigenvalues $\lambda_{m+1}, \ldots, \lambda_d$ of S and the approximation error is given by their sum.

To recap, we have that:

- The m largest eigenvalues of the covariance matrix S of the dataset give the m principal components that maximize the variance of the projections.
- The m smallest eigenvalues of the covariance matrix S of the dataset give the m principal components that minimize the error of the projections.

Instead of computing the full eigenvalue decomposition of S – a very slow process considering that d may be a huge value –, there are more efficient ways to compute PCA, such as efficient eigenvalue decomposition restricted only m eigenvectors or singular value decompositions of the centered data matrix X.

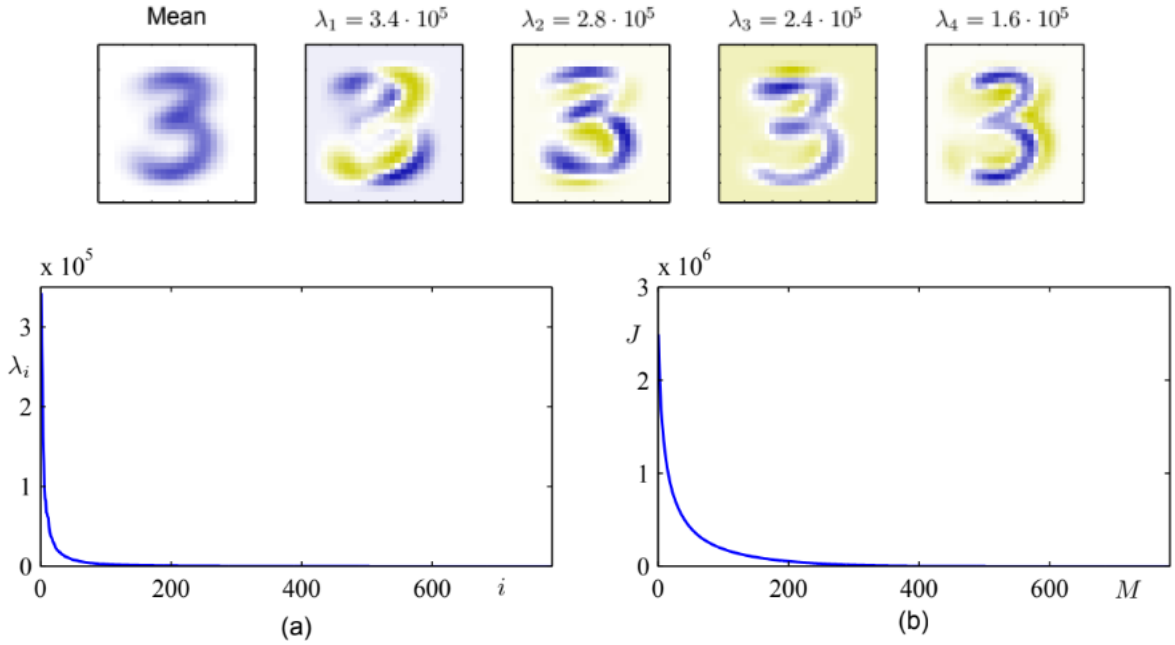

Figure 8.4: (a) Eigenvalue spectrum and (b) sum of discarded eigenvalues (error)

If the number of points in the dataset is smaller than the dimensionality of the data space (such as a small set of high resolution images), meaning that $N < d$, at least $d - N + 1$ eigenvalues of S are equal to zero, making the process of finding eigenvalues for S very inefficient. To make the process faster, we observe that given $S = \frac{1}{N} X^T X$ the eigenvector equation becomes:

$$\frac{1}{N} X^T X u_i = \lambda_i u_i$$

By left-multiplying by X, we obtain:

$$\frac{1}{N} X X^T X u_i = \lambda_i X u_i \implies \frac{1}{N} X X^T v_i = \lambda_i v_i$$

where $v_i = X u_i$, concluding that XX^T has the same $N - 1$ eigenvalues of $X^T X$ (the others are 0). When N is smaller than d, the matrix eigenvalues of XX^T can actually be computed more efficiently. After finding the eigenvectors v_i of XX^T of our interest, we notice that by left multiplying by X^T we obtain that:

$$\frac{1}{N} X^T X X^T v_i = \lambda_i X^T v_i \implies \frac{1}{N} X z_i = \lambda_i z_i$$

where $z_i = X^T v_i$, hence z_i is an eigenvector of S with eigenvalue λ_i. After finding such vectors, we can normalize them by computing:

$$u_i = \frac{1}{\sqrt{N\lambda_i}} X^T v_i$$

To summarize, when $N < d$ we can find the eigenvectors with the following procedures:

1. Compute the data-centered matrix X
2. Compute (efficiently) the $N-1$ eigenvalues λ_i of XX^T
3. Select the eigenvalues of interest $\lambda_1, \ldots, \lambda_k$ and compute their eigenvectors $v_i, \ldots, v_i$ over XX^T
4. Compute $u_i = \frac{1}{\sqrt{N\lambda_i}} X^T v_i$ to get the eigenvectors for $\lambda_1, \ldots, \lambda_k$ over S

Another version of PCA can be implemented through probabilities. In this version, we assume that for the vector x of all the N inputs it holds that $x = Wz + \mu$ for some W, z, μ, where z are the lower dimensional latent variables. We also assume Gaussian distribution over z and linear-Gaussian relationship between the latent variables and the data:

$$\Pr[z] = \mathcal{N}(z; 0, 1)$$

$$\Pr[x \mid z] = \mathcal{N}(x; Wz + \mu, \sigma^2 I)$$

The model parameters in this case become W, μ, σ. The marginal distribution is given by:

$$\Pr[x] = \int_{-\infty}^{+\infty} \Pr[x \mid z] \cdot \Pr[z]\, dz = \mathcal{N}(x; \mu, C)$$

where $C = WW^T + \sigma^2 I$, while the posterior distribution is:

$$\Pr[z \mid x] = \mathcal{N}(z; M^{-1}W^T(x - \mu), \sigma^2 M)$$

with $M = W^T W + \sigma^2 I$. The maximum likelihood for the data X is thus computed through:

$$\arg\max_{W,\mu,\sigma} \ln \Pr[X \mid W, \mu, \sigma^2] = \arg\max_{W,\mu,\sigma} \sum_{i=1}^{N} \ln \Pr[x_i \mid W, \mu, \sigma^2]$$

Setting the derivatives to zero, we have the following closed solution for μ_{ML}:

$$\mu_{\mathrm{ML}} = \overline{x} = \frac{1}{N} \sum_{i=1}^{N} x_i$$

while W and σ^2 depend on the eigenvalues and eigenvectors of S (a non-trivial result that we'll be omitting). We observe that this maximum likelihood can also be computed through an algorithm similar to the EM algorithm.

8.3 Autoencoders

An **autoencoder** is a type of neural network architecture designed to efficiently compress (encode) input data down to its essential features, then reconstruct (decode) the original input from this compressed representation. In other words, an autoencoder can be viewed as the combination of two neural networks, where the former network transforms the input into a low dimensional representation and the latter reconstructs the original input based on reconstruction loss. The usefulness of autoencoders comes from the training process: if the decoder-side of the full network is capable of reconstructing the original data in a good way, then the representation produced by the encoder-side is optimal.

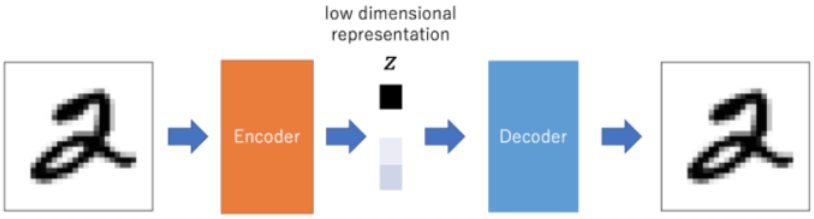

Figure 8.5: Schematic of an autoencoder.

Autoencoders are formed by reduced size hidden layers (bottleneck) which learn to reconstruct their input by minimizing a loss function. Given a dataset $D = \{x_i \mid i \in [N]\}$, autoencoders are trained with the same sample x_i used both for input and output. Autoencoders learn ho to encode and decode the samples in a dataset into a low-dimensional space. For this reason, they can also be seen as a method for non-linear principal component analysis.

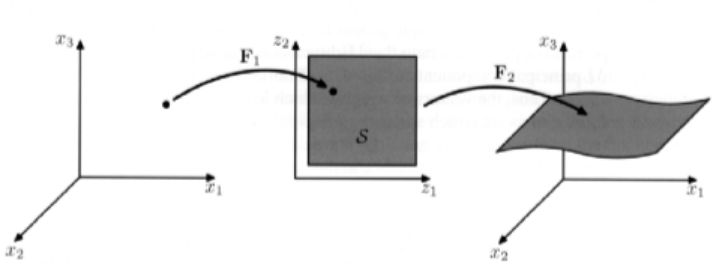

Figure 8.6: 3D reconstruction from the 2D hidden layer generates a 2D manifold in 3D.

For example, consider an autoencoder for a dataset representing the values $\{0, \ldots, 7\}$, where each value is represented with an 1-out-of-8 encoding, i.e. 4 is represented as $\begin{bmatrix}0 & 0 & 0 & 0 & 1 & 0 & 0 & 0\end{bmatrix}$. A possible autoencoder layout for this dataset is the network formed an input layer with 8 nodes, a hidden layer with 3 nodes and an output layer with 8 nodes. After training, the first layer will have learned how to optimally encode the values $\{0, \ldots, 7\}$ with a 3 dimensional vector.

Autoencoders are also very useful for anomaly detection, which can be viewed as a binary classification problem $f : X \to \{n, a\}$, where n is the "normal" class and a is the "abnormal" class. Using a dataset that contains only normal samples, i.e. $D = \{(x_i, n) \mid$ ß $\in [N]\}$, the autoencoder is capable of predicting if a test $x \in X - X_D$ is normal or abnormal.

1. Train an autoencoder with a dataset $\{x_i \mid i \in [N]\}$ assuming that each sample in the dataset is not abnormal
2. Compute the final train loss
3. Determine a treshold δ, such as $\delta = \text{mean}(loss) + \text{std}(loss)$
4. Given $x' \in X - X_D$, reconstuct x with the autoencoder and compute loss$'$
5. If loss$' < \delta$ return n, otherwise return a

Autoencoders can be based on different learning methods. The most common ones are **Variational Autoencoders (VAE)**, which focus on learning the latent space structure, and **Generative Adversarial Networks (GAN)**, which focus on learning a distribution and don't use a latent space.

In VAEs, the goal is to modify data in specific directions, identifying the most meaningful ones for the latent spacae (e.g. "distort" faces by adding glasses or changin expressions, produce digits from differet hand-writing styles, distort 3D meshes, ...). In other words, we want to feed vectors to the VAE and get a realistic samples of $\Pr[X]$, where X is the sample set. We expect that similar vector values should produce similar generated instances. To achieve this, the encoder produces a distribution instead of an actual encoding, while the decoder operates on sample from this distribution

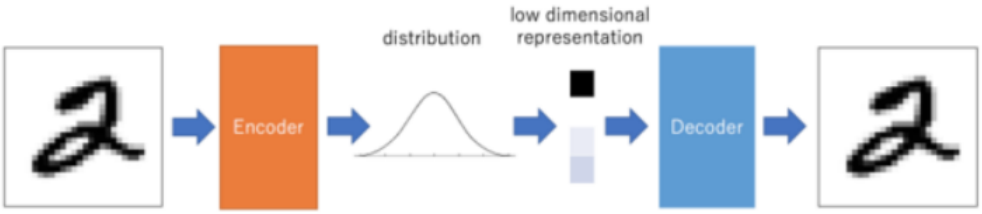

Figure 8.7: Schematic of Variational Autoencoder (VAE)

To learn the distribution, the encoder works on a parametric distribution, typically Gaussian, yielding a mean μ and a covariance matrix Σ. In the training process, we use a loss term based on Kullback-Leibler divergence.

$$\text{loss} = \text{MSE} + \text{dKL}(\mathcal{N}(z; \mu, \Sigma), \mathcal{N}(z; 0, I))$$

We also notice that the sampling operation is non differentiable, making backpropagation hard to use. To fix this issue, we add the obtained sample to the backpropagation pro-

cess (re-parametrization), meaning that the encoder's gradients will also depend on the extracted sample.

In GANs, instead, the goal is to produce samples from an input data distribution $\mathcal{X}$. To achieve this, the idea is to invert a neural network (or a CNN if needed). The encoder processes the input and produces a vector, called *code*. The decoder receives a random code and produces an image and reconstructs the image. If the encoder is a CNN, the decoder uses "deconvolutional layers", realized through transposed convolution.

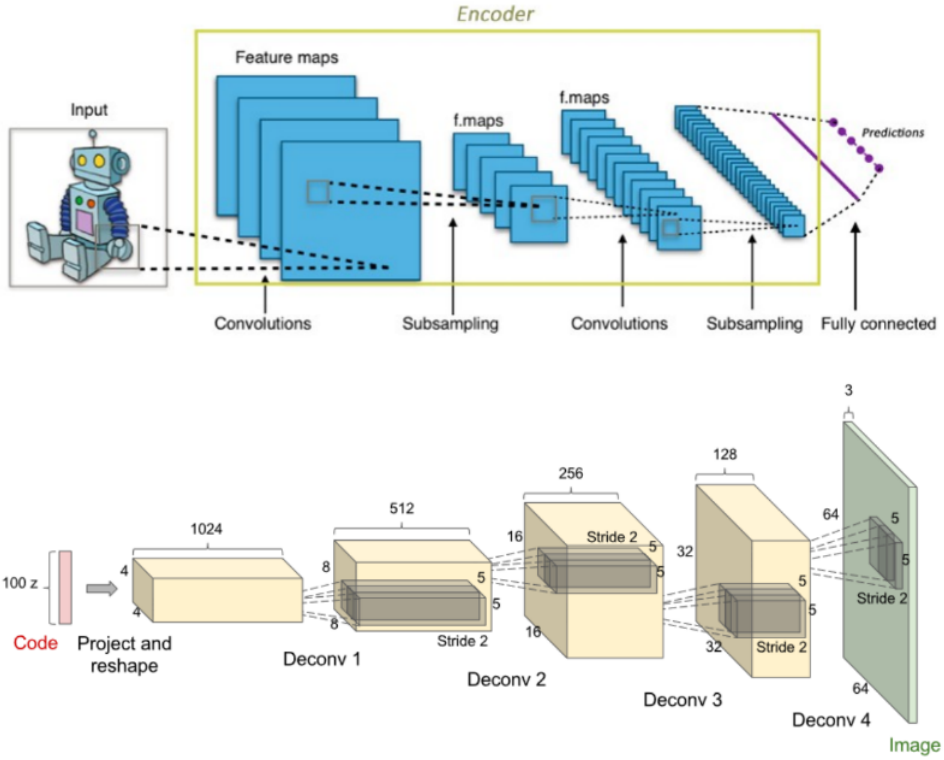

Figure 8.8: Schematic of a Generative Adversarial Networks (GAN)

To train the decoder, we use *adversarial training*. This training model is based on two or more components that afflict each other: any improvement for one component negatively affects the other components. Eventually, the components will reach a good compromise that maximizes performance. In GAN, adversarial training is realized through a combination of two networks, a generator network (the decoder) and a discriminator network (a critic for the decoder's output).

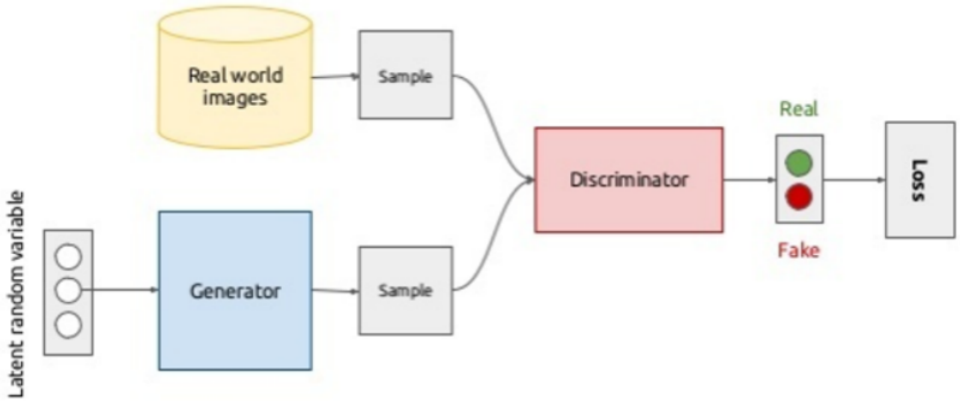

Figure 8.9: Schematic of adversarial training in GANs

The generator produces samples from the distribution $\Pr[X]$, while the discriminator identifies if a sample actually comes from $\Pr[X]$ or not (the distribution is unknown to the discriminator). To make the two networks compete with each other, the generator tries to foold the discriminator in believing that the sample is correct, while the discriminator tries to separate real samples from the fake ones.

1. Train the discriminator with a batch of data $\{(x_i, \text{real})\} \cup \{x'_j, \text{fake}\}$, where $x_i \in X_D$ and x'_j are images generated from the generator with random values for the latent variables
2. Train the generator by using the entire model with discriminator layers fixed, i.e. non-trainable, with a batch of data $\{(r_k, \text{real})\}$, where r_k are random values of the latent variable

We observe that the GAN model is actually very powerful. In fact, they can also be used to **attack** other models. For instance, we can train a GAN model to make it generate images that are slight alterations of a correct image and make the other model believe that it resembles something completely different.

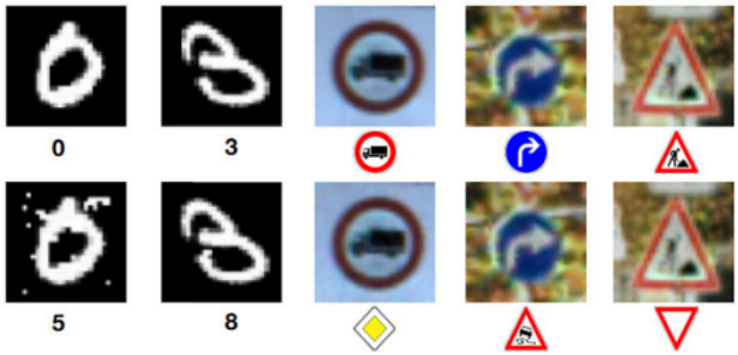

Figure 8.10: A GAN is capable of generating adversarial images (bottom) similar to correct one (top) that make another ML model believe they resemble something completely different.

9 Reinforcement learning

9.1 Dynamic systems

A **dynamic system** is a system that evolves over time. At any given time k, a dynamic system has a *state* x_k representing the current situation of the system. The *state transition model* (or *evolition rule*) of the dynamical system is a function f_k that describes what future states follow from the current state x_k and the current state noise w_k, i.e. $f_k(x_k, w_k) = x_{k+1}$. The function is often deterministic but in some cases it is also randomized. The system is also provided with an *observation model*, a function h_k that takes in input the current state x_k and the current observation noise v_k, returning the current observation z_k.

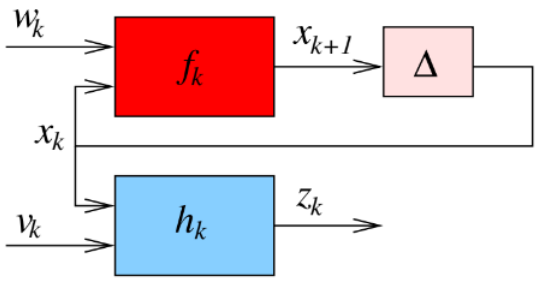

Figure 9.1: A dynamic system.

Dynamic systems are typically used to predict the future state of a system. In learning, we can use dynamic systems as a prediction tool. In particular, given the past observations $z_0, \ldots, z_k$, we want to learn a model (f, h). In a more formal way, we'll be considering dynamic systems described by:

- A set of states X, which can be either an explicite discrete finite representation, a continuous representation or a probabilistic representation

- A set of actions A, which can be either an explicite discrete finite representation, a continuous representation
- A transition function δ, which can be deterministic, non-deterministic or probabilistic
- A set of observations Z, which can be either an explicite discrete finite representation, a continuous representation or a probabilistic representation

Each state of a dynamic system encodes all the past knowledge needed to predict the future, all the knowledge gathered through operation and all the knowledge needed to pursue the goal. When the state is fully observable, meaning that the agent is capable of taking an instantanous snapshot of the current state, the decision making problem for an agent is to decide which action must be executed in a given state. In other words, the agent has to compute the function $\pi : X \to A$.

The dynamic system model fully expresses the concept of **reinforcement learning**. Differently from supervised and unsupervised learning, in RL we are given a behaviour function $\pi : X \to A$ and a more complex dataset:

$$D = \{(x_1, a_1, r_1, \ldots, x_i, a_i, r_i)^{(j)} \mid j \in [N]\}$$

where each r_k represents the **reward value** obtained by executing action a_k in state x_k. The idea behind reinforcement learning is to make the model learn the optimal strategy that maximizes the total reward value obtain through a *Carrot and Stick Approach*.

9.2 Markov Decision Process

A **Markov Decision Process (MDP)** is any process, i.e. sequence of actions, used for decision making that has the so-called *Markov property*:

- Once the current state is known, the evolution of the dynamic system does not depend on the history of states, actions and observations.
- The current state contains all the information needed to predict the future.
- Future states are conditionally independent of past states and past observations given the current state.
- The knowledge about the current state makes past, present and future observations statistically independent.

In this simple model, the states are assumed to be fully observable, meaning that we don't need to consider observations obtained through noise.

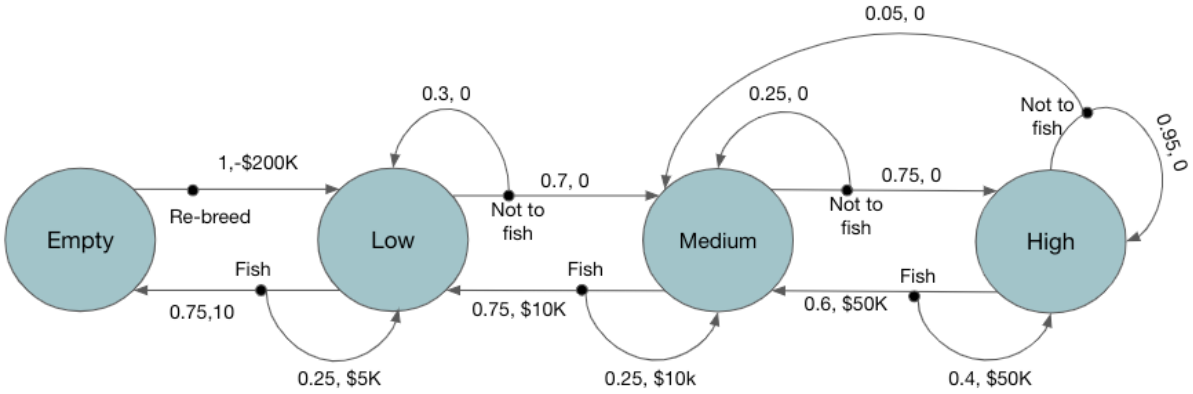

Figure 9.2: Graphical representation of an MDP for salmon fishing.

Formally, MDPs are described as a quadruple (X, A, δ, r). Depending on the context, the definition of such quadruple changes:

- In an MDP with *deterministic transitions*, X is the set of states, A is the set of actions, $\delta : X \times A \to X$ is a transition function, $r : X \times A \to \mathbb{R}$ is a reward function.
- In an MDP with *non-deterministic transitions*, X is the set of states, A is the set of actions, $\delta : X \times A \to \mathcal{P}(X)$ is a transition function, $r : X \times A \times X \to \mathbb{R}$ is a reward function, where the additional state represents the destination state. When action a is executed, the state x transitions to one of the states $x' \in \delta(x, a)$ with equal probability.
- In an MDP with *stochastic transitions*, X is the set of states, A is the set of actions, $\delta : X \times A \to \mathcal{P}(X)$ is a transition function, $r : X \times A \times X \to \mathbb{R}$ is a reward function, where the additional state represents the destination state. When action a is executed, the state x transitions to one of the states $x' \in \delta(x, a)$ according to a probability distribution over transitions $\Pr[x' \mid x, a]$.

In all three cases, the Markov property is enforced by $x_{t+1} = \delta(x_t, a_t)$ and $r_t = r(x_t, a_t)$. In presence of non-deterministic or stochastic actions, the state resulting from the execution of an action is not known before the execution of the action, but it can be fully observed after its execution.

To give a better intuition behind how these type may vary based on the context, suppose that we want to learn a controller for the grid in Figure 9.3 that reaches the right-most side of the environment from any initial state.

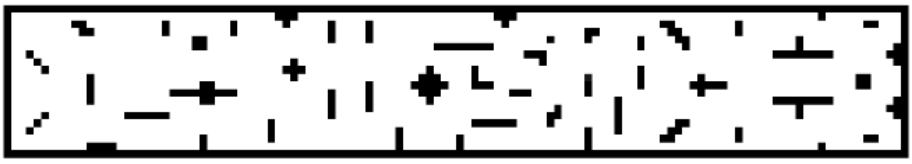

Figure 9.3: Example of a $N \times M$ grid environment with some obstacles.

A deterministic MDP for this problem may be defined as:

- $X = \{(i, j) \mid i \in N, j \in M\}$
- $A = \{\text{Left}, \text{Right}, \text{Up}, \text{Down}\}$
- δ : cardinal movements with deterministic effects, i.e. $\delta((i, j), \text{Left}) = (i - 1, j)$. if destination state is a black square, the agent remains in the current state.
- r : 1000 for reaching the right-most column, -10 for hitting any obstacle, 0 otherwise.

A stochastic MDP for this problem, instead, may be defined as:

- $X = \{(i, j) \mid i \in N, j \in M\}$
- $A = \{\text{Left}, \text{Right}\}$
- δ : cardinal movements with probabilistic effects (0.1 probability of moving diagonally). If destination state is a black square, the agent remains in the current state.
- r : 1000 for reaching the right-most column, -10 for hitting any obstacle, +1 for any Right action and -1, any Left action, 0 otherwise.

9.2.1 Optimal policies

Given an MDP, we want to find the function $\pi : X \to A$ representing the optimal action to be executed for each state. This function is called **optimal policy**. The optimality criteria is defined with respect to the expected value of the *cumulative discounted reward* $V^{\pi}(x)$. In the deterministic case, we define:

$$V^{\pi}(x) = \sum_{i=1}^{+\infty} \gamma^{i-1} \overline{r_i}$$

where $\overline{r_i} = r(x_i, a_i, x_{i+1}), a_i = \pi(x_i)$ and $\gamma \in [0, 1]$ is the discount factor for future rewards, while in the non-deterministic or stochastic case we define:

$$V^{\pi}(x) = \mathbb{E}\left[\sum_{i=1}^{+\infty} \gamma_{i-1} \overline{r_i}\right]$$

The optimal policy is the one that maximizes $V^{\pi}(x)$ over all $x \in X$:

$$\pi^* \in \arg\max_{\pi} V^{\pi}(x) \; \forall x \in X$$

In other words, we have that π^* is an optimal policy if and only if for any other policy π it holds that $V^{\pi^*}(x) \geq V^{\pi}(x)$ for all $x \in X$.

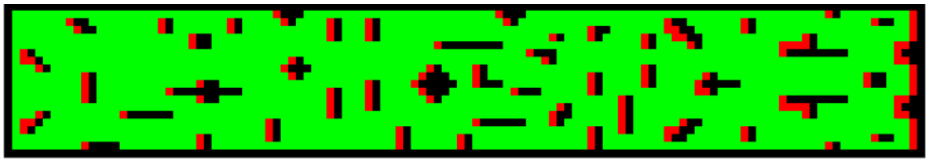

Figure 9.4: Example of optimal policy for the stochastic MDP of the previous example. The green squares represent a Right action, while the red squares represent a Left action.

Figuring out the optimal policy depends entirely on which parts of the MDP are known or not. For instance, if the whole quadruple (X, A, δ, r) is known, the optimal policy can be found through reasoning or planning. If some parts of the tuple are unknown (typically the reward function), a leaning phase is needed.

To give an example, suppose that we are playing on a bandit with n arms. This machine can be viewed as a one-state MDP $(\{x_0\}, A, \delta, r)$ where $\delta(x_0, a_i) = x_0$ for all $a_i \in A$ and $r : A \to \mathbb{R}$.

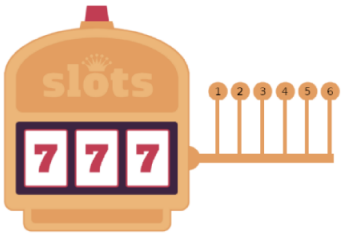

Figure 9.5: A bandit with 6 arms.

Since the reward depends only on the arm being pulled, the optimal policy would clearly be the one which always pulls the optimal arm, i.e. the one with maximal reward. To find this policy, we must use different methods based on the situation. When r is deterministic, meaning that each arm will always return the same reward, the solution is given by:

$$\pi^*(x_0) \in \arg\max_{a_i \in A} r(a_i)$$

If r is also known then $\pi^*(x_0) = a_i^*$, where $r(a_i^*) \geq r(a_j)$ for all $a_j \in A$. If r is unknown, instead, then we can find the optimal policy through $|A|$ iterations: we individually execute each a_i and collect the reward, returning the action that gave the highest reward.

When r is stochastic (or non-deterministic), meaning that the reward collected by pulling the same arm may change, the solution is given by:

$$\pi^*(x_0) \in \arg\max_{a_i \in A} \mathbb{E}[r(a_i)]$$

If r is also known, the solution is given by $\pi^*(x_0) = a_i^*$, where a_i^* is the action that maximizes the reward value on average based on the distribution of r for a_i. For instance,

if $r(a_i) = \mathcal{N}(\mu_i, \sigma_i)$ then:

$$a_i^* \in \arg\max_{a_i \in A} \mathbb{E}[r(a_i)] = \arg\max_{a_i \in A} \mu_i$$

When r is non-deterministic and unknown, instead, things become more difficult. The typical approach is to use an *online algorithm*, i.e. an algorithm where the input is fed piece-by-piece in a serial fashion instead of having the entire input available from the start.

1. Inizialize a data structure Θ
2. For $t = 1, \ldots, T$ (until termination condition is reached)
    3. *Choose* an action $a_{(t)} \in A$
    4. *Execute* $a_{(t)}$ and *collect* the reward $r_{(t)}$
    5. Update Θ with the collected result
6. Return π^* according to how Θ works

The actions written in italics in the above algorithm are executed in an online fashion. Generally, a large number of interactions is needed, meaning that $T \gg |A|$. The way in which the final strategy is chosen strictly depends on the type of data structure used, which also depends on the type of distribution we expect the function r to have. For instance, if $r(a_i) = \mathcal{N}(\mu_i, \sigma_i)$ then the algorithm can be implemented as follows:

1. Inizialize $\Theta_{(0)}[i] = 0$ and $c[i] = 0$ for all $i \in |A|$.
2. For $t = 1, \ldots, T$ (until termination condition is reached)
    3. *Choose* an index $\widehat{i}$ for action $a_{(t)} = a_{\widehat{i}} \in A$
    4. *Execute* $a_{(t)}$ and *collect* the reward $r_{(t)}$
    5. Increment $c[\widehat{i}]$
    6. Update $\Theta_{(t)}[\widehat{i}] = \frac{1}{c[\widehat{i}]}(r_{(t)} + (c[\widehat{i}] - 1) \cdot \Theta_{(t-1)}[\widehat{i}])$
6. Return $\pi^*(x_0) = a_i^*$ with $a_i^* \in \arg\max_{a_i \in A} \Theta_{(T)}[i]$

Given an agent accomplishing a task according to an MDP $<X, A, \delta, r>$ for which the functions δ and r are unknown, we want to determine the optimal policy π^* that the agent should use. We recall that, differently from supervised learning, in reinforcement learning the dataset has the form:

$$D = \{(x_1, a_1, r_1, \ldots, x_i, a_i, r_i)^{(j)} \mid j \in [N]\}$$

Hence, we cannot use samples of the form $(x_{(j)}, \pi(x_j))$ in order to learn π^*. In the general case, to learn a policy we use **experimentation strategies**. The idea is to explore possible strategies in order to find the actions that maximize the value of a quality function $\widehat{Q}(x, a)$ and then repeat the process on strategies using those actions. We split this process

in two phases. In the *exploration phase*, we randomly select an action a (usually with a low value of $\widehat{Q}(x,a)$). In the *exploitation phase*, we select the action a that maximizes $\widehat{Q}(x,a)$.

The simples experimentaton strategy is the **ε-greedy strategy**: given a value $\varepsilon \in [0,1]$, we select either a random action with probability ε or the best action with probability $1-\varepsilon$. The ε value usually decreases over time (exploration followed by exploitation).

A more advanced experimentation strategy is the **soft-max strategy**. Actions with higher $\widetilde{Q}$ values are assigned higher probabilities and all actions have at least some probability, i.e. no action has probability zero. For every state $x \in X$ and every action $a_i \in A$, we have that:

$$\Pr[a_i \mid x] = \frac{k^{\widehat{Q}(x,a_i)}}{\sum_{a_j \in A} k^{\widehat{Q}(x,a_j)}}$$

where $k > 0$ determines how strongly the selection favors actions with high $\widehat{Q}$ values. The k value usually increases over time (exploration followed by exploitation).

When δ and r are unknown, the agent cannot predict the effect of its actions. However, it can still execute them and then observe the outcome. The learning task is thus performed by repeatedly chosing an action, executing it in order to observe the resulting new state and collect its reward.

There are two main approaches to learning with MDP:

- **Value iteration** (or *Q-Learning*): estimate V^* and then compute π^*
- **Policy iteration**: estimate directly π^*

9.2.2 Value iteration (Q-Learning)

In **value iteration**, the agent could learn the value function $V^{\pi^*}(x)$ (written as $V^*(x)$ for short) from which it could determine the optimal policy through the following formula:

$$\pi^*(x) \in \arg\max_{a \in A} r(x,a) + \gamma V^*(\delta(x,a))$$

where $\gamma \in [0,1]$ is the discount factor for future rewards. However, since δ and r are unknown, this policy cannot be directly computed. In the deterministic case, we define the quality function $Q^\pi(x,a)$ for a policy π as the value we expect to receive when executing a in state x and then acting according to π. In other words, we have that:

$$Q^\pi(x,a) = r(x,a) + \gamma V^\pi(\delta(x,a))$$

By definition, if the agent learns Q, we can determine the optimal policy without knowing δ and r since:

$$\pi^*(x) \in \arg\max_{a \in A} r(x,a) + \gamma V^*(\delta(x,a)) = \arg\max_{a \in A} Q^*(x,a)$$

We observe that:

$$V^*(x) = \max_{a \in A} r(x,a) + \gamma V^*(\delta(x,a)) = \max_{a \in A} Q^*(x,a)$$

Hence, we get that $Q^*(x,a)$ can be defined in terms of itself:

$$Q^*(x,a) = r(x,a) + \gamma V^*(\delta(x,a)) = r(x,a) + \gamma \max_{a' \in A} Q^*(\delta(x,a),a')$$

Let $\widehat{Q}$ be the current approximation of Q. The training rule in this case would be:

$$\widehat{Q}_{(t)}(x,a) = \overline{r} + \gamma \max_{a' \in A} \widehat{Q}_{(t-1)}(x',a')$$

where $\overline{r} = r(x,a,x')$ is the immediate reward and x' is the state resulting from applying action a to state x.

Algorithm 17: Deterministic Q-Learning

Given a deterministic MDP (X, A, δ, r), the following online algorithm returns an optimal policy π^*

function Q_LEARN_DET(X, A, δ, r)
  For each x, a initialize a table entry $\widehat{Q}_{(0)}(x,a) = 0$
  Observe the current state x
  for $t = 1, \ldots, T$ **do** ▷ Until termination condition
    Choose a action a ▷ According to ε-greedy or soft-max strategy
    Execute the action a
    Observe the new state x'
    Collect the immediate reward $\overline{r}$
    Set $\widehat{Q}_{(t)}(x,a) = \overline{r} + \gamma \max_{a' \in A} \widehat{Q}_{(t-1)}(x',a')$
    Set $x = x'$
  end for
  Return π^* defined as $\pi^*(x) \in \arg\max_{a \in A} \widehat{Q}_{(T)}(x,a)$
end function

We observe that for each $t \in [T]$ it holds that $\widehat{Q}_{(t)}(x,a)$ always underestimates $Q^*(x,a)$. In fact, it actually holds that $0 \leq \widehat{Q}_{(t)}(x,a) \leq \widehat{Q}_{(t+1)}(x,a) \leq Q^*(x,a)$. Convergence is guaranteed if all state-actions pares are visited infinitely often.

To give an example of how the algorithm works, consider a 2×3 grid of rectangles. Starting from the bottom left square S_0, we want to reach the top right square G.

S_3	S_4	G
S_0	S_1	S_2

Figure 9.6: The 2×3 grid

At each time, we can move by one square in any of the four cardinal directions L, R, U, D (only if by moving we're still inside the grid). Each movement has an associated reward value: if we move from a different square to the goal square we get 100 points, otherwise we get 0 points.

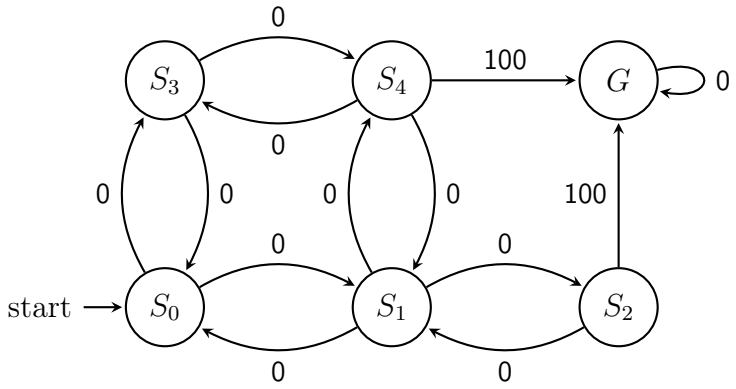

Figure 9.7: Deterministic MDP describing the grid goal problem.

Here, the value function V^* of the optimal policy assumes the following values:

$\mathbf{x}$	S_0	S_1	S_2	S_3	S_4	G
$\mathbf{V}^*(\mathbf{x})$	81	90	100	90	100	0

After the training process, the $\widehat{Q}$ converges to values very close to the optimal ones:

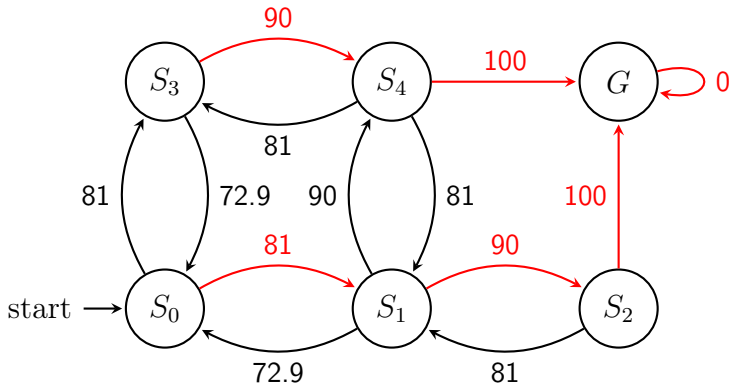

Figure 9.8: Values learned through the deterministic Q learning process. The red arrows represent the optimal policy returned by the algorithm.

The evaluation of RL agents is usually performed through cumulative reward gained over time. Due to the various exploration phases that the algorithm may take, the cumulative reward plot may be very noisy. A better approach could involve the repeated execution (until termination condition) of k steps of learning followed by an evaluation of the current learned policy.

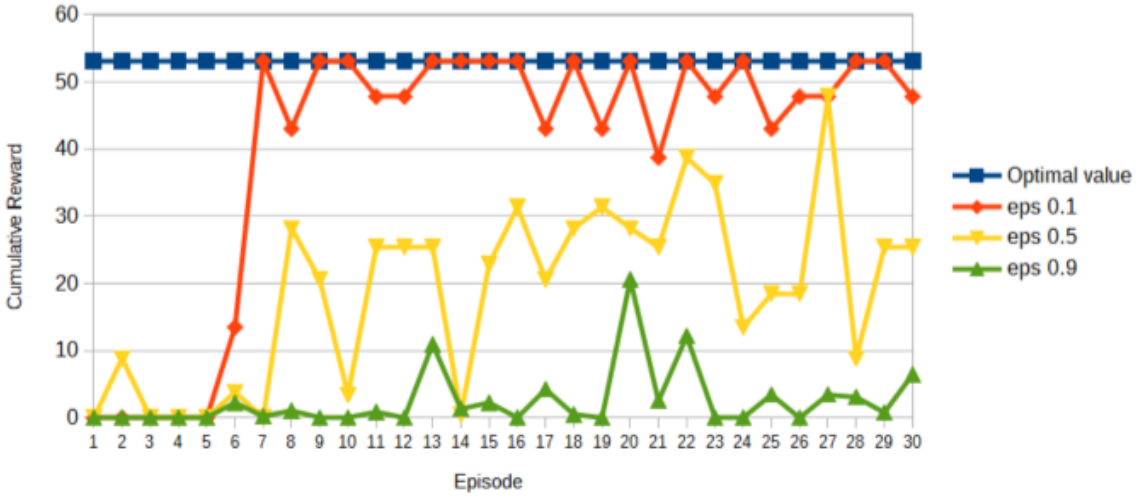

Figure 9.9: Cumulative reward plot of an RL agent learning phase.

In the stochastic (or non-deterministic) case, the deterministic Q-Learning algorithm is not guaranteed to converge due to $\widehat{Q}_{(t)}(x,a) \leq \widehat{Q}_{(t+1)}(x,a)$ not always being true. Hence, we have to use a new training rule. First, we recall that the value function $V^{\pi}(x)$ is defined in terms of expected value. Hence, we define Q^{π} as:

$$Q^{\pi}(x,a) = \mathbb{E}[r(x,a) + \gamma V^{\pi}(\delta(x,a))]$$

Similarly to the deterministic case, we get that:

$$\begin{aligned} Q^{*}(\pi)(x,a) &= \mathbb{E}[r(x,a) + \gamma V^{*}(\delta(x,a))] \\ &= \mathbb{E}[r(x,a)] + \gamma\, \mathbb{E}[V^{*}(\delta(x,a))] \\ &= \mathbb{E}[r(x,a)] + \gamma \sum_{x' \in X} \Pr[x' \mid x,a] \cdot V^{*}(x') \\ &= \mathbb{E}[r(x,a)] + \gamma \sum_{x' \in X} \Pr[x' \mid x,a] \cdot \max_{a' \in A} Q^{*}(x',a') \end{aligned}$$

While the training rule is adapted as:

$$\begin{aligned} \widehat{Q}_{(t)}(x,a) &= \widehat{Q}_{(t-1)}(x,a) + \alpha_{(t-1)}(\overline{r} + \gamma \max_{a' \in A}(\widehat{Q}_{(t-1)}(x',a')) - \widehat{Q}_{(t-1)}(x,a)) \\ &= (1 - \alpha_{(t-1)})\widehat{Q}_{(t-1)}(x,a) + \alpha(\overline{r} + \gamma \max_{a' \in A} \widehat{Q}_{(t-1)}(x',a')) \end{aligned}$$

where $\alpha_{(t-1)} = \frac{1}{1+v_{(t-1)}(x,a)}$, with $v_{(t-1)}(x,a)$ being the total number of times the state-action pair (x,a) has been visited up to the t-th iteration. Through this rule, the non-deterministic Q-Learning algorithm is guaranteed to converge when every pair state-action is visited infinitely often.

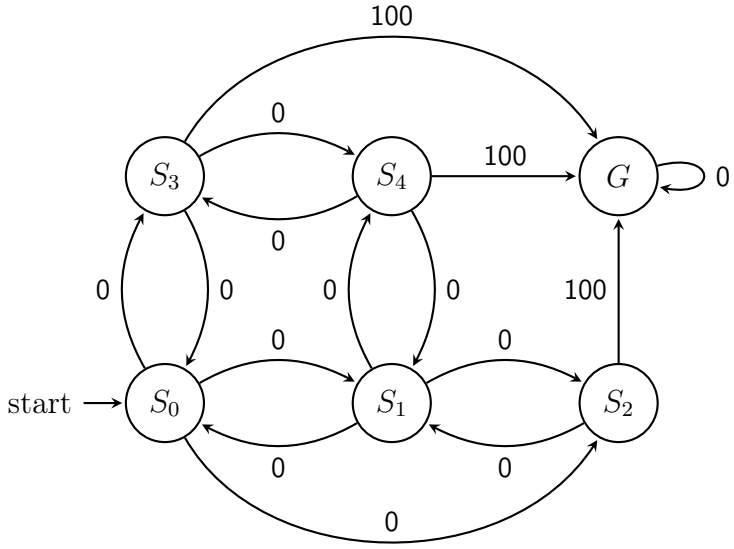

Figure 9.10: Non-deterministic MDP describing the grid goal problem where we're also allowed to move right by two squares.

For non-deterministic learning there are also more advanced Q-Learning algorithm: the TD algorithm and the SARSA algorithm. **Temporal difference (TD)** Q-learning is a variant of Q-leaning where the discrepancy between successive Q estimates reduces over time. Given a value n, the n step time difference of the function Q is given by:

$$\widehat{Q}^{*^{(n)}}(x_t, a_t) = \overline{r}_t + \gamma \overline{r}_{t+1} + \gamma^{n-1} \overline{r}_{t+n-1} + \gamma^n \max_{a' \in A} \widehat{Q}(x_{t+n}, a')$$

Given a value $\lambda \in [0, 1]$, we define:

$$Q^\lambda(x_t, a_t) = (1 - \lambda) \sum_{i=1}^{+\infty} \lambda^{i-1} Q^{*^{(i)}}(x_t, a_t)$$

which is equivalent to:

$$Q^\lambda(x_t, a_t) = r_t + \gamma((1 - \lambda) \max_{a' \in A} \widehat{Q}(x_t, a_t) + \lambda Q^\lambda(x_{t+1}, a_{t+1}))$$

We observe that:

- When $\lambda = 0$, we have that $Q^\lambda(x_t, a_t) = \widehat{Q}^{*^{(1)}}(x_t, a_t)$
- When $\lambda = 1$, only observed r_{t+1} are considered
- When $0 < \lambda < 1$, the algorithm increases the emphasis on discrepancies based on more distant look-aheads

The $TD(\lambda)$ algorithm uses the above training rule, which sometimes converges faster than standard Q-Learning. Moreover, it is guaranteed to converge $\lambda \in [0, 1]$ for learning V^*.

In **SARSA** Q-learning, instead, we use a tuple (s, a, r, s', a') (equivalent to (x, a, r, x', a') in our notation) in order to define:

$$\widehat{Q}_{(t)}(x, a) = \widehat{Q}_{(t-1)}(x, a) + \alpha(\overline{r} + \gamma \widehat{Q}_{(t-1)}(x', a') - \widehat{Q}_{(t-1)}(x, a))$$

where x', a' are chosen accordingly to a policy based on current estimate of Q

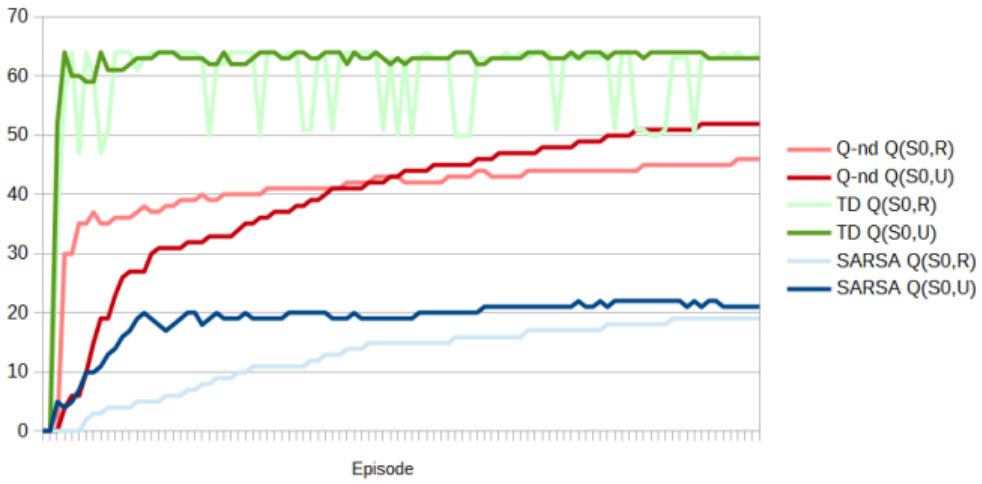

Figure 9.11: Comparison of the non-deterministic Q-Learning variants.

We observe that, in all variants, an explicit representation of $\widehat{Q}$ through a table may not be feasible for large models. In this case, we define an approximating function $Q_\theta(x, a) = \theta_0 + \theta_1 F_1(x, a) + \ldots + \theta_n F_n(x, a)$ and use a linear regression to learn it, usually through a neural network.

9.2.3 Policy iteration

In **policy iteration** we directly estimate the optimal policy π instead of estimanting the functions V and Q. Each policy can be parametricly represented as:

$$\pi_\theta(x) = \max_{a \in A} \widehat{Q}_\theta(x, a)$$

Let $\rho(\theta)$, called *policy value*, be the expected value of executing π_θ. The policy gradient is given by $\nabla_\theta \rho(\theta)$. In its generic formThe policy gradient algorithm for a parametric representation of the policy $\pi_\theta(x)$ corresponds to:

1. Choose θ
2. Repeat until terminating condition: estimate $\nabla_\theta \rho(\theta)$ through experiments and set $\theta := \theta + \eta \nabla_\theta \rho(\theta)$

In robot learning, we use a more refined approach:

1. Let π be an initial policy
2. Repeat until terminating condition: compute t random perturbations of π $R_1, \ldots, R_k$ and set π as the best combination of $R_1, \ldots, R_k$

The perturbations are generated from π by setting $R_i = \{\theta_1 + \delta_1, \ldots, \theta_N + \delta_N\}$, with δ_i being randomly chosed from $\{-\varepsilon_j, 0, +\varepsilon_j\}$ for some small fixed value ε_j relative to θ_j. The last step is usually problem-specific. In the eastiest case, we set

$$\pi \in \arg\max_{j \in [k]} F(R_j)$$

for some evaluation function F. For more complex cases, we combine $R_1, \ldots, R_t$ by setting:

$$\pi := \pi + \frac{A}{|A|}\eta$$

with:

$$A_j = \begin{cases} 0 & \text{if } \mathrm{Avg}_{0,j} > \mathrm{Avg}_{-\varepsilon,j} \text{ and } \mathrm{Avg}_{0,j} > \mathrm{Avg}_{+\varepsilon,j} \\ \mathrm{Avg}_{+\varepsilon,j} - \mathrm{Avg}_{-\varepsilon,j} & \text{otherwise} \end{cases}$$

and where:

- $\mathrm{Avg}_{-\varepsilon,j}$ is the average score of all R_i with a negative perturbation
- $\mathrm{Avg}_{0,j}$ is the average score of all R_i with a zero perturbations
- $\mathrm{Avg}_{+\varepsilon,j}$ is the average score of all R_i with a positive perturbation

9.3 Hidden Markov Model

In some cases, even the actions of a Markov Decision Process may be unknown. When this happens, the stochastic MDP model becomes equivalent to a **Markov Chain**, a dynamic system evolving according to the Markov property where future evolutions depend only on the current state x_t instead of a pair (x_t, a_t). In other words, Markov chains can be seen as a stochastic MDP where there is only one action that can be executed.

A **Hidden Markov Model (HMM)** is a triple (X, Z, π_0) where X is a finite set of discrete states that are non-observable in a direct way, Z is a set of observations that can be discrete or continuous and π_0 is an initial distribution. Each state x_t assumes a hidden value (hence the non-observability) and it can stochastically transition to the state x_{t+1}. Moreover, each state x_t can stochastically produce an observation z_t. Since the states cannot be observed directly, the goal is to learn the current state by observing the current observation.

The transition model and observation model are governed by $\Pr[x_t \mid x_{t-1}]$ and $\Pr[z_t \mid x_t]$. For each state, a *state transition matrix* A determines the transition probabilities for the values assumed by x_t and x_{t+1}:

$$a_{i,j} = \Pr[x_t = j \mid x_{t-1} = i]$$

Similarly, the observation model is described by a function b_k for each value k where:

$$b_k(z_t) = \Pr[z_t \mid x_t = k]$$

The initial probability distribution always corresponds to $\pi_0 = \Pr[x_0]$.

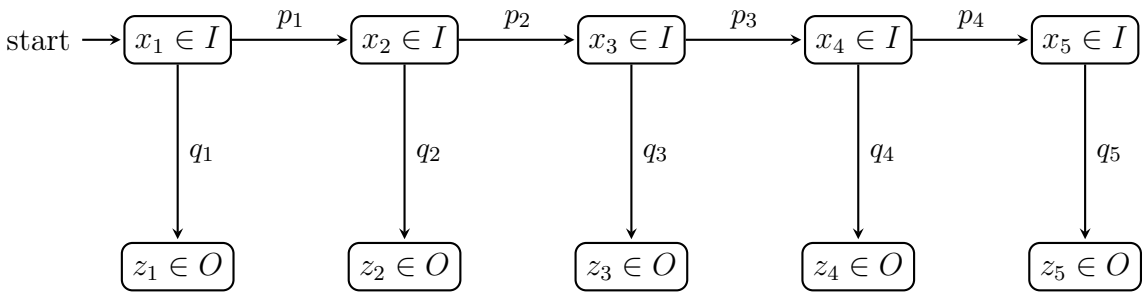

Figure 9.12: Example of a HMM. The set I is discrete, while the set O can be either discrete or continuous. For each i we have that $p_i = \Pr[x_{i+1} = a \mid x_i = b]$ and $q_i = \Pr[z_t \mid x_i = a]$

HMMs are typically used for status recognition. For instance, a sensor may recognize the currently unknown status of the stoplight (discrete set of states) only through photos of a traffic light (continuous set of multi-dimensional observations).

To infer on HMMs, we work with one of two setups:

- **Filtering**: we assume that

$$\Pr[x_T = k \mid z_{1:T}] = \frac{\alpha_T^k}{\sum_j \alpha_T^j}$$

- **Smoothing**: we assume that

$$\Pr[x_T = k \mid z_{1:T}] = \frac{\alpha_T^k \beta_T^k}{\sum_j \alpha_T^j \beta_T^j}$$

where x_T is the last state, $\alpha_t^k = \Pr[x_t = k, z_{1:t}]$ and $\beta_t^k = \Pr[z_{t:T} \mid x_t = k]$. The application of the chain rule on HMMs allows us to factorize the entire model through a single probability that expresses it:

$$\Pr[x_{0:T}, z_{1:T}] = \Pr[x_0] \cdot \Pr[z_0 \mid x_0] \cdot \Pr[x_1 \mid x_0] \cdot \ldots \cdot \Pr[z_{T-1} \mid x_{T-1}] \cdot \Pr[x_T \mid x_{T-1}]$$

This allows us to stochastically compute the values assumed by the states starting from $\Pr[x_T = k \mid z_{1:T}]$ and by computing α_t^k, β_t^k for each $t \in [T]$. To achieve this, we split the process in two steps In the *forward step*, we compute α_t^k using the following procedure:

1. For each state k let:
$$\alpha_0^k = \pi_0 b_k(z_0)$$
2. For each $t = 1, \ldots, T$ and each state k let:
$$\alpha_t^k = b_k(z_t) \sum_j \alpha_{t-1}^i a_{i,k}$$

Note: recall that $a_{i,k} = \Pr[x_t = k \mid x_{t-1} = i]$ and $b_k(z_t) = \Pr[z_t \mid x_t = k]$.

In the *backward step*, we compute β_t^k in a similar way:

1. For each state k let:
$$\beta_T^k = 1$$
2. For each $t = T-1, \ldots, 1$ and each state k let:
$$\beta_t^k = \sum_j \beta_{t+1}^i a_{k,j} b_j(z_{t+1})$$

During the learning phase, given the output sequences we want to determine the maximum lihelihood estimate of the parameters of the HMM (transmission and emission probabilities). If the states can be observed at training time, the transition and observation models can be estimated with statistical analysis by setting:

$$a_{i,j} = \Pr[x_t = j \mid x_{t-1} = i] = \frac{|\{i \to j \text{ transitions}\}|}{|\{i \to * \text{ transitions}\}|}$$

$$b_k(z) = \Pr[z_t \mid x_t = k] = \frac{|\{\text{observe } z \text{ in state } k\}|}{|\{\text{observe } * \text{ in state } k\}|}$$

If the states cannot be observed even in the training phase, we compute a local maximum likelihood using a variant of the EM algorithm.

9.4 Partially Observable MDP

A **Partially Observable MDP (POMDP)** is a generalization of the MDP model where the agent's decision process assumes that the system dynamics are determined by an MDP, but the agent cannot directly observe the underlying state. In other words, a POMDP combines the decision making of MDP and the non-observability of HMM.

A POMDP is a quintuple (X, A, Z, δ, r, o) where:

- X is a set of states
- A is a set of actions
- Z is a set of observations
- $P[x_0]$ is a probability distribution of the initial state
- $\delta(x, a, x') = \Pr[x' \mid x, a]$ is a probability distribution over transitions
- $r(x, a)$ is a reward function
- $o(x', a, z') = \Pr[z' \mid x', a]$ is a probability distribution over observations

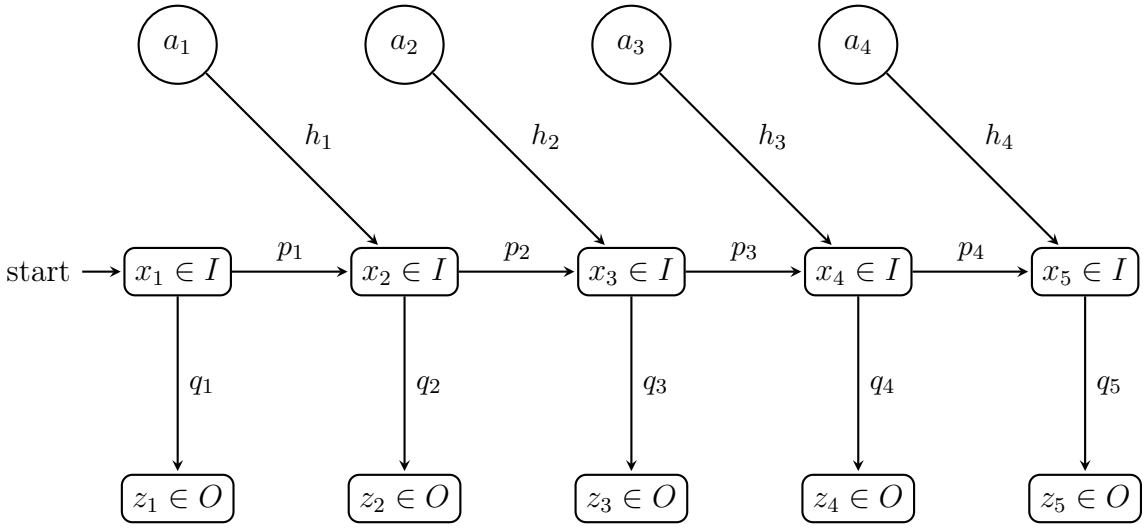

Figure 9.13: Example of a POMDP.

As per standard MDPs, we want to find an optimal policy that the agent can use. A trivial solution is to define the policy using a history of observations-action pairs. To achieve this, we would need a long history in order to correctly estimate the probabilities. The state-of-the-art solution is to use **belief states**, a set of possible states the process may be in. Let $b(x)$ be a probability distribution over the states. A POMDP can be described as an MDP (B, A, τ, ρ) where:

- B is an infinite set of belief states
- A is a set of actions
- $\tau(b, a, b')$ is a probability distribution over transitions
- $\rho(b, a, b')$ is a reward function

In this context, we want to get the optimal policy $\pi^* : B \to A$. Given a current belief state b, an action a and an observation z' obtained after the execution of a, let $\mathrm{SE}(b, a, z')$ be a state-estimating function whose output is the next belief state $b'(x')$. Through some manipulation, we obtain that:

$$\begin{aligned} b'(x') &= \mathrm{SE}(b, a, z') \\ &= \Pr[x' \mid b, a, z'] \\ &= \frac{\Pr[z' \mid x', b, a] \cdot \Pr[x' \mid b, a]}{\Pr[z' \mid b, a]} \\ &= \frac{\Pr[z' \mid x', b, a] \cdot \sum_{x \in X} \Pr[x' \mid b, a, x] \cdot \Pr \Pr[x \mid b, a]}{\Pr[z' \mid b, a]} \\ &= \frac{o(x', a, z) \cdot \sum_{x \in X} \delta(x, a, x') \cdot b(x)}{\Pr[z' \mid b, a]} \end{aligned}$$

The transition function τ is defined as:

$$\tau(b, a, b') = \Pr[b' \mid b, a] = \sum_{z \in Z} \Pr[b' \mid b, a, z] \cdot \Pr[z \mid b, a]$$

where $\Pr[b' \mid b, a, z] = 1$ if $b' = \mathrm{SE}(a, b, z)$, 0 otherwise. Similarly, the reward function ρ is defined as:

$$\rho(b, a) = \sum_{x \in X} b(x) r(x, a)$$

To train the model, we also define a value function V in terms of belief states:

$$V(b) = \max_{a \in A} \rho(b, a) + \gamma \sum_{b' \in B} \tau(b, a, b') V(b')$$

By substituting the definitions of τ and ρ, we get that:

$$V(b) = \max_{a \in A} \sum_{x \in X} b(x) r(x, a) + \gamma sum_{z \in Z} \Pr[z \mid b, a] \cdot V(b_z^a)$$

where b_z^a is the belief state obtained from action a and observation z. To estimate the value function, variants of Q-leaning can be used on a discretized version of the belief MDP obtained by discretizing the distributions $b(x)$. A similar method can be devised for any MDP solving technique.

Finally, we observe that, by construction, the optimal belief policy $\pi^* : B \to A$ that we obtain is observation-based: since the belief state is determined by observations, in the final policy the next action will also depend on the observed values. In other words, the optimal solutions is actually a **policy tree** where, after an action is executed, the computation branches based on the observed values.

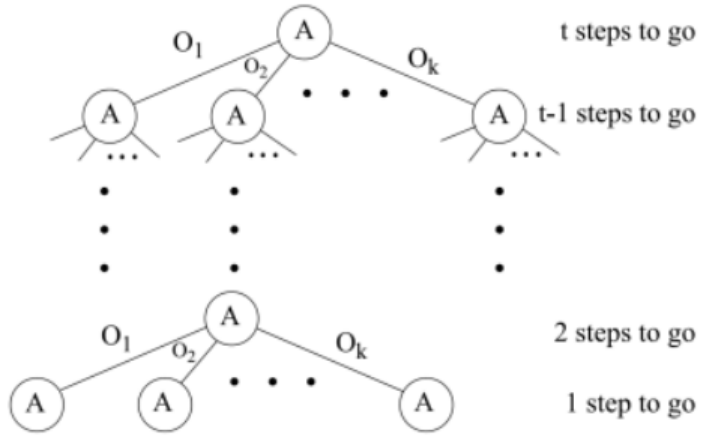

Figure 9.14: Structure of a policy tree.