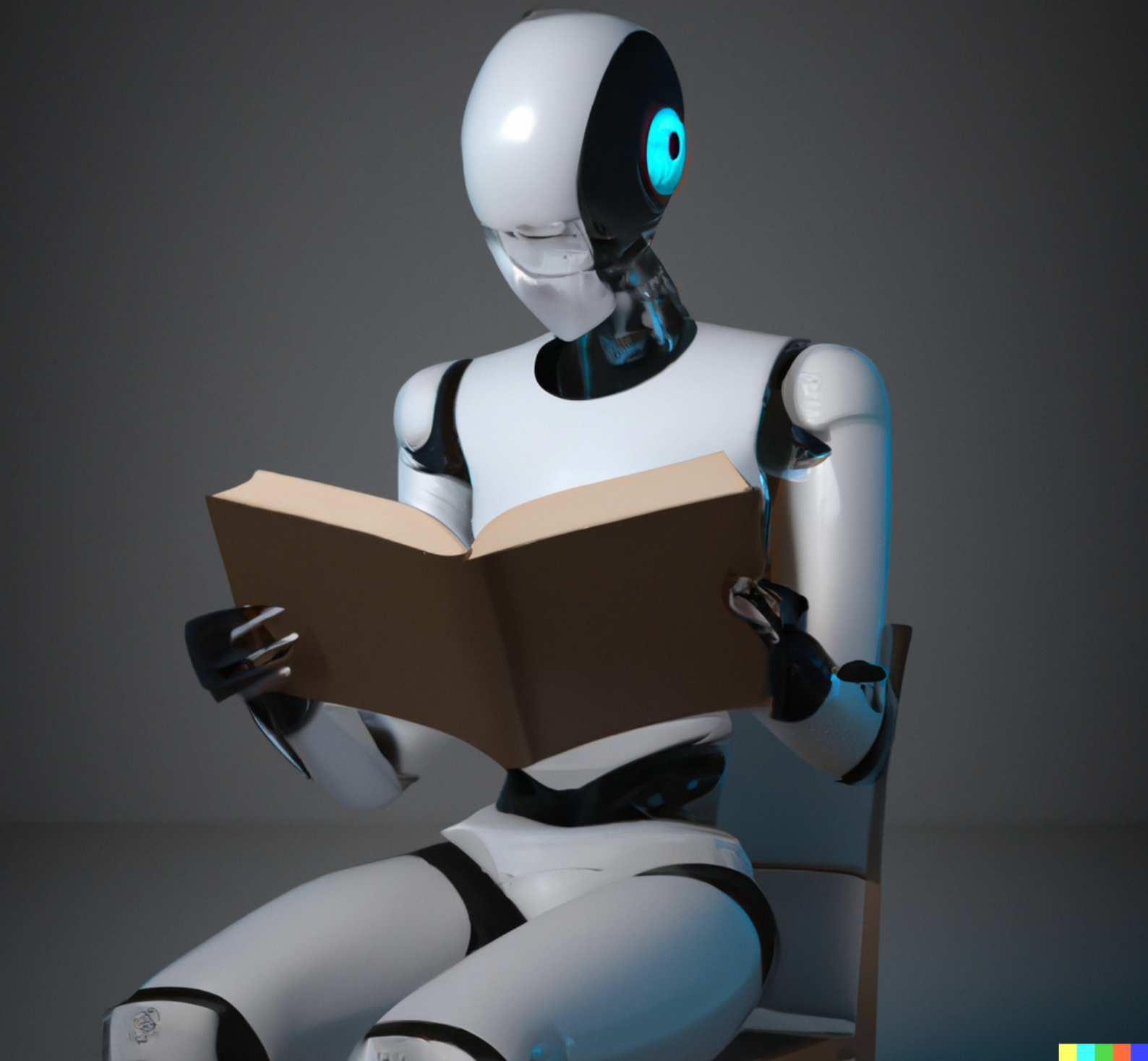
Notes on
MACHINE LEARNING
Claudio Cusano



Cover image automatically generated by Dall-E 2 (`https://openai.com/product/dall-e-2`) upon the query "A humanoid robot reading a book on machine learning, digital art".

Notes on Machine Learning

Claudio Cusano

Spring, 2023

Contents

Lecture 1

Introduction

One of the main goals of computer science in general, and artificial intelligence in particular, is that of building machines able to solve challenging problems by autonomously taking decisions under unforeseen conditions. Learning from experience is a key requirement to achieve this ambitious goal. Indeed, since the foundation of artificial intelligence, machine learning was recognized as an important sub-field. In 1950 Alan Turing, the father of computer science, argued that simulating the mind of a "child" machine with the ability of learning could be simpler than building a "grown-up" machine already endowed with intelligence.

The term *Machine Learning* was introduced in 1959 by Arthur Samuel, one of the pioneers of artificial intelligence. He wrote one of the first successful programs with learning capabilities: a program that plays the game of checkers and that adjusts its style by analyzing past games. He gave a definition of machine learning as "the field of study that gives computers the ability to learn without being explicitly programmed".

In the early years of machine learning the most influential idea was that of *connectionism*. Connectionists argued that natural intelligence was an emerging property, the result of the interaction among a multitude of simple elements (the neurons in the brain). The attempt to reproduce the neural system led first to the development of the *perceptron* (a model of a single neuron) and then of *artificial neural networks*. However, these earlier models were only able to learn to solve simple problems and, both in theory and in practice, they were found to be too limited.

In the Eighties the logical/deductive approach became very popular and attracted most of the interest of researchers in artificial intelligence. Applications like expert systems were in fact far more successful than the solutions coming from connectionism, and the study of artificial neural networks was discontinued. In that period machine learning and artificial intelligence took different directions and the two research communities separated.

Machine learning was re-founded, this time starting from statistics and probability theory. Led by Vladimir Vapnik, researchers redefined the new field that was renamed as *statistical learning theory*. Differently from the past, the objectives were limited to the solution of specific problems of moderate complexity. Most applications were related to pattern recognition problems, such as optical character recognition, for which deductive reasoning was still struggling to get satisfactory answers. During the Nineties many successful models were developed. Notably Corinna Cortez and Vapnik itself discovered Support Vector Machines, a model which became the *de-facto* standard for many machine learning applications for two or three decades.

Over time, machine learning became more and more successful thanks to the development of the Internet and to the growing availability of data. With more data and more computational resources, it became possible to design more complex models. These factors brought in the last decade to the so called "deep learning revolution". Artificial neural networks were rediscovered and it became clear that their earlier failure was mostly due to the small scale of the first experiments. Today neural networks composed of millions of artificial neurons are routinely used to solve problems that just ten years ago were confined in the realm of science fiction. Applications like automatic language translation, image understanding, speech recognition, self-driving cars, etc. are now possible thanks to the pioneering work made by researchers who, despite the opinion of the majority, continued to pursue the goal of creating machines able to learn.

1.1 Promises and pitfalls

Machine learning nowadays is used in a variety of domains for many different applications including: medical diagnosis, robot control, document classification, image recognition, biometric identification, financial forecasting, natural language processing, computer vision, speech recognition, marketing analysis, digital photography, market segmentation... the list is too long to be reported.

Why machine learning is so widespread? The main reason is that there are many problems for which is very difficult do identify a good solution that could be programmed. Computer programs are just a way of encoding *algorithms*, and algorithms are made of precisely and unambiguously defined computational steps. However, most of the actions we perform as human beings, as well as most of the decisions we take, are not the result of the conscious execution of an algorithm. Which algorithms do we use to recognize a face in a picture, to understand a sentence in our native language, or to walk down the street? The processes driving all these actions are very difficult to

describe in the terms required by algorithms. Nevertheless we are perfectly able to execute them with minimal effort.

Machine learning represents a tempting approach to solve all these problems for which we are not able to explicitly program a satisfactory solution. In some cases machine learning is attractive because it can be used when we are not able to define the problem, or even when we did not understood the problem! As an extreme example, have a look at the MuseNet project[1], in which researchers trained a neural network to compose music.

An important advantage of machine learning over conventional programming is that often the same models can be used in different domains. Sometimes it is enough to retrain them on different data to make them learn to solve new problems. For instance, the artificial Go player AlphaZero[2] after minimal modifications has been been able to learn to play also the games of Chess and Shoji.

Usually you DON'T use machine learning to solve problems for which a viable algorithmic solution is known. In fact, one of the drawbacks of machine learning is that it works in a non-deterministic way. Even the most accurate models sometimes make mistakes. This makes problematic their use in mission critical domains. Moreover, machine learning systems work as "black boxes", since it is very difficult to decipher the "reasoning" they follow to come to their decisions. In particular, errors can be very surprising and may occur unexpectedly even in dealing with simple cases.

Another weakness of systems based on machine learning is that the quality of their decisions depends on the quality of the data used to train them. If the data contain contradictory or simply wrong information, the systems may learn to make mistakes. For instance, a system whose output is text in natural language, when trained on bad data may compose sentences with grammatical errors or objectionable language. Since the Internet and the social media include a large variety of discrimination, prejudice, rudeness, hatred, ignorance... it is too easy for systems trained on data from these sources to learn negative expressions.

When dealing with sensible matters, data should be carefully scrutinized to prevent the system from learning bad human habits. Early face recognition systems were able to recognize only Caucasian people, because the training data, often collected in western universities, included few people of other ethnicities. Similarly, a system for hiring people could learn that some group of candidates (say, women) tends to have bad careers and it may reinforce this prejudice with its recommendations.

[1] `https://openai.com/blog/musenet/`

[2] `https://deepmind.com/blog/article/alphazero-shedding-new-light-grand-games-chess-shogi-and-go`

As a final remark, be careful in assigning meaning to the answers obtained from a machine learning model. They are useful to make predictions that help us in solving problems. However, they cannot be lightly used to build theories about the world. Machine learning is mostly based on *statistical models*, not on *causal models*. It is well known in statistics that "correlation does not imply causation", which means that the correlation between two variables do not allow to deduce any cause-effect relationship. Similarly, the fact that a machine learning model may be able to make accurate predictions about a given system does not mean that we can use it to understand its fundamental laws. We may use a machine learning model to identify possible frauds, but we cannot use it to automatically convict the suspects!

1.2 Key concepts

Let's start from the following loose definition:

> *Machine Learning is the field of study that gives computers the ability to learn without being explicitly programmed.*

This definition can accommodate for a multitude of different approaches to the problem of making computers able to learn from experience. In fact, a large variety of theories and techniques have been formulated in the attempt to reach that goal. Without ambition of being exhaustive, here we will try to describe the elements shared by most of them. Exceptions, however, are quite common.

A machine learning systems is made of a few components. The main role is played by a *model* (sometimes called *hypothesis*). The model describes how to derive the answer y of the system from the data x. This input/output relationship in the simplest cases is represented by a parametric function:

$$h_\theta(x) \to y. \tag{1.1}$$

The process of learning (also called *training*) consists in finding good values for the parameters θ by looking at some set of data, called the *training set*. This process is defined by a *learning algorithm* (Figure 1.1). It is quite common the case in which learning is achieved by minimizing the divergence of the observed behavior of the model from the desired one. This concept is quantified by a *loss function*. Sometimes the learning algorithm is just a general purpose numerical optimization algorithm, such as the Newton's algorithm, which is used to find the parameters of the model minimizing the loss function. In most cases learning is applied just once. When finished the model with the parameters found can be deployed in the application for which it was designed.

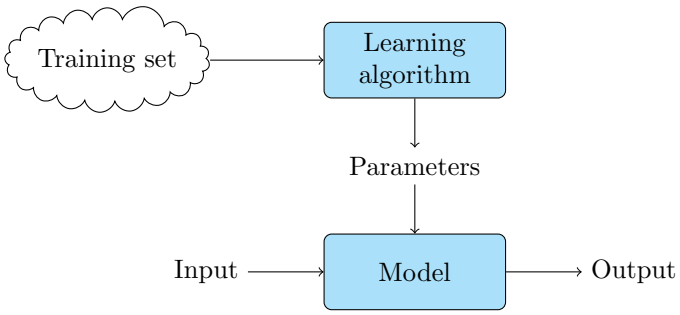


Figure 1.1: Schematic view of a machine learning system.

Probably the most important concept about machine learning is that of *generalization*: the model is expected to provide correct answers for data not seen during training. Without generalization, machine learning would be just a complicated way of memorizing data.

1.3 Learning scenarios

The most critical issue for machine learning algorithms is the source of the information to learn from. We can distinguish three main scenarios. In *supervised learning* the model learns from *labeled* data, in *unsupervised learning* it uses *unlabeled* data and in *reinforcement learning* it learns from the interaction with an environment. These scenarios will be discussed in the following sections.

Supervised learning

In *supervised learning* the training data is annotated with the right answer. Training a model in the supervised learning scenario means, in practice, making it able to predict the right answer for new data.

Supervised learning is the scenario in which machine learning achieved the majority of its successes. Probably this is because it is easy to express problems in terms of the relationship between input and output. The loss function is just a measure of the difference between the output of the model and the expected output, and many supervised learning algorithms consist just in minimizing the average difference over the training data.

Supervised learning can be further divided on the basis of the nature of the information to be predicted. When the predictions are real numbers we have a *regression* problem. Regression analysis is a classic topic in mathematics and statistics and predates machine learning by more than a century. The theory is well understood and there are many models and algorithms. In

some cases, notably in *least squares regression*, there is a closed-form solution. These are some examples of regression problems:

- given some weather data, predict the temperature in a given place and moment in time;
- given financial data, predict the future price of oil;
- predict the score that a user will assign to a movie;
- given a picture portraying a face, estimate the age of the subject.

When the predictions are chosen from a finite set of options we have a *classification* problem. The options, which are called *classes*, are mutually exclusive and every input case belong exactly to one of them. A model for this kind of problems is called *classifier* and often work by assigning a score to each class and by taking the class with the highest score. Classification tends to be more difficult to model than regression. One complication is that it is not obvious to design a loss function since the predictions of the classifier are either correct or not, with no half measures. The followings are examples of classification problems:

- spam detection: classify an e-mail message as 'spam' or 'not spam';
- medical diagnosis: given the patient data determine if he or she is healthy or not.
- image recognition: assign a label, chosen from a predefined set, representing the content of a picture;
- gender classification: tell if the subject of a picture is male or female.

For some problems the predictions are not just single numbers or a class labels, but are data organized in more complex structures such as sequences, trees, maps or graphs. In these cases we have a problem of *structured prediction*. Structured prediction can be very hard and only in the last decade it was possible to approach it outside toy problems. Are examples of problems of structured prediction:

- language translation: given a sentence in a language, translate it in another language;
- pose estimation: given the image of a person, find a skeleton made of the arms, legs, torso, head… connected by the joints;
- speech to text: transcribe the recording of a sentence pronounced by a human speaker;

- tagging: assign a set of labels to an item like a photograph, a movie or a product. This would be a classification problem if classes were mutually exclusive, but it is not when multiple labels can be assigned to the same item at the same time.

Unsupervised learning

In *unsupervised learning* the training data is not annotated. The goal is to identify common structures that are useful to understand a given phenomenon. The lack of supervision makes problems of unsupervised learning hard to define and to solve. Loss functions must evaluate the output of the models without having the "right answer" as reference. These are some tasks that are classified in the unsupervised learning category:

- *clustering* is a common activity in data exploration. It is also called *unsupervised classification* and consists in finding a rule that divides data into categories that are homogeneous according to some criteria. Cluster analysis can be used to segment customers of a service on the basis of their personal information.

- *Anomaly detection* is another example of unsupervised classification in which the machine learning model has to separate "normal" data from anomalies. This is also called *one-class classification* since all the training examples are usually assumed to belong to the class of normal data. Anomaly detection can be used to identify potential frauds in financial transactions.

- *Density estimation* is a task borrowed from statistics that is closely related to anomaly detection. In density estimation the goal is to evaluate the likelihood of the observations. For instance, density estimation can be used to predict the probability of occurrence of events.

- In *dimensionality reduction* the goal is to find a transformation that reduces the complexity in the input data, possibly by recognizing interesting attributes and by discarding useless information. This is often used as a preliminary step before supervised learning.

An important advantage of the unsupervised scenario is that it is relatively easy to collect large training sets. In fact, annotating the data often represents the main cost of building training sets for supervised learning. Unlabeled data is usually cheap and its quantity may compensate for the lack of information provided by supervision. Some researchers argue that unsupervised learning could be the key for the next big jump ahead in machine learning.

Reinforcement learning

In *reinforcement learning* the model is used by an agent operating in an environment (real or simulated). The environment provides a feedback about the actions performed by the agent and this feedback is used to adjust the model. Reinforcement learning algorithms may exploit annotated data, like in supervised learning, or just the feedback of the environment.

Thanks to reinforcement learning an agent can become able to plan sequences of actions and to devise complex adaptive strategies. These qualities are especially useful in automation, planning, autonomous vehicles and other similar domains. They have been very successful in playing games such as Go, Chess and Poker, where reinforcement learning agents reached superhuman performance.

1.4 Data sets

Data is the most important factor in the development of machine learning systems. Difficulties that seem insurmountable are often overcome by increasing the quantity or quality of training data. Many important advancements have been driven by the collection of public *data sets* that became standard benchmarks and that are important resources for the comparison of competing models and techniques.

Data sets are not just a set of data, they also contain annotations (useful for supervised learning) and experimental protocols that guarantees the repeatability of the experiments.

A good data set is often more important than a good model. Data sets containing inaccurate or wrong information can mislead learning algorithms. Data sets that are not representative of the problem can cause the learned model to depend on unrealistic assumptions that make it useless when deployed in a real scenario.

Many machine learning models do not deal directly with raw data. In these cases the input to the model is a group of *features* that numerically encode useful properties of data. For instance, an image recognition instead of considering the pixels of the images could work by analyzing carefully chosen statistics that numerically encode the color distribution, the occurrences of specific patterns etc. Sometimes good features make it possible to design models that are both simple and accurate.

1.5 Programming

Building machine learning systems does not require any special programming skill. They are made of algorithms that can be implemented in any

programming language, including C, Java, Lisp etc. Since the algorithms include many numerical operations many people find convenient to use environments, such as Matlab and R, that have been especially designed for that purpose. Bear in mind, however, that beside research and education, machine learning algorithms most of the times are not implemented from scratch, but are taken from one of the many libraries developed by the community.

The Python programming language, thanks to its flexibility, ease of use and excellent numerical libraries, became the most popular choice among machine learning practitioners. Scikit-learn, Tensorflow and Pytorch are examples of state-of-the-art libraries developed by the most advanced companies in the field and are routinely used by developers and researchers. And the best part is that the whole ecosystem is open source and free to use!

The pvml library

The source code that will be shown here is not intended to serve as a reference for professional use. In fact, it is provided only for educational purpose and has been written with the intent of being easy to understand and to experiment with. For any serious application you are encouraged to adopt one of the libraries mentioned earlier. In particular, the code is not as efficient as it could be and its execution may take a multiple of the time required by professional solutions.

The code has been packaged in the pvml library to make it easier to use. However, it may be even easier to take the code directly from the text and to adapt it to your needs. In fact, implementing your own version of the algorithms may be a very good exercise and you are strongly encouraged to try it at least a few times.

The pvml library is available on the Python Package Index[3] and on the github platform[4].

1.6 Summary

This lecture presented an overview of the field of machine learning. In particular it touched its main motivations and application scenarios at a very introductory level. Theories, best practices, applications and technical details will be discussed in the next lectures. In particular:

- we traced back the history of machine learning since its beginning;

[3] https://pypi.org/project/pvml
[4] https://github.com/claudio-unipv/pvml

- understood strengths and weaknesses of machine learning systems compared to conventional algorithms;
- we got acquainted with some terminology of the field and some key concepts;
- understood the differences among the main learning scenarios (supervised, unsupervised and reinforcement learning), including their most common incarnations;
- we briefly discussed the technologies that can be used to develop machine learning systems.

Lecture 2

Classification

Automatic classification is one of the most common problems in machine learning. Many human activities include some form of classification. You can see an example of classification every time you visit a bookstore where volumes are sorted by topic. When a new book arrives the clerk has to decide where to put it. There is a fixed set of sections where the book may belong to (novels, travel, arts, science...) and the clerk must choose one by looking at some *features* of the book. These features may be the words in the title, the image on the cover, the type of the publisher... or even information from main text, if there is enough time to read it.

In short, the problem of classification consists in finding a rule that assigns items of some kind (e.g. books) to one *class* (or category) chosen among a finite set of predefined ones. Many machine learning methods have been developed to address this problem. Here are some domains where they have been applied:

- in medical diagnosis patients are classified as healthy or sick on the basis of their medical records;
- in image recognition the subject of pictures is categorized by looking at the values of the pixels;
- in spam filters e-mail messages are classified as being spam or not;
- in fraud detection transactions are classified as trustworthy or fraudulent.

From a mathematical point of view, a *classifier* is a function $h : \mathcal{X} \to \mathcal{Y}$ mapping the elements of $\mathcal{X}$ to the classes in $\mathcal{Y}$. The set $\mathcal{X}$ includes the mathematical descriptions of the items to classify. Most of the times, these descriptions will be vectors of real numbers but, in general, they may be any kind of representations, including lists, trees, sets, etc.

In defining the classes in a classification problem only few constraints need to be satisfied:

- the classes must be *mutually exclusive*, that is, no element belongs to multiple classes;
- the classes must be *exhaustive*, that is, there are no elements that do not belong to any of the classes;
- there must be a finite number of classes.

The set $\mathcal{Y}$ can be any finite set of labels $\{\omega_0, \omega_1, \ldots, \omega_{k-1}\}$ representing the classes. However, to simplify the notation we will just use a set of integer numbers $\{0, 1, \ldots, k-1\}$, but it must be clear that the actual values of these labels are irrelevant so that, for instance, class 3 is no closer to class 4 than it is to class 5.

A classifier can be found through *supervised learning*, provided that a suitable *training set* is available. The training set is formed by examples in $\mathcal{X}$ coupled to class labels in $\mathcal{Y}$. Given the training set

$$\{(x_0, y_0), (x_1, y_1), \ldots, (x_{m-1}, y_{m-1})\}\,,\ x_i \in \mathcal{X}, y_i \in \mathcal{Y}, \tag{2.1}$$

a supervised learning method aims to find a classifier h that reproduces the relationship between the examples and their labels. Typically, the search for a good classifier will be restricted to a parametric family of functions $\{h_\theta : \theta \in \Theta\}$ so that the problem of learning is reduced to that of choosing a suitable value for the parameter θ.

2.1 The first example

Let's start with an example where we take the part of a group of students who want to know how many hours of study are needed to pass an exam. To do so they collect information about what happened in the past. They asked 90 previous students how much they prepared for the exam and whether they passed or not. These information form a training set $\{(x_0, y_0), \ldots, (x_{89}, y_{89})\}$. Each x_i is a real number representing how many hours the i-th student studied, and the corresponding y_i is 1 if the student passed the exam and 0 otherwise. So in this case $\mathcal{X} = \mathbb{R}$ and $\mathcal{Y} = \{0, 1\}$. Since there are exactly two classes, this is a *binary classification* problem.

Figure 2.1 shows the data forming the training set. It is clear that studying more hours gives more chances to pass the exam, but the "best" number is not obvious. All the students that studied less than 50 hours failed. Those that studied more than 65 passed the exam. So the right threshold must be between these two values.

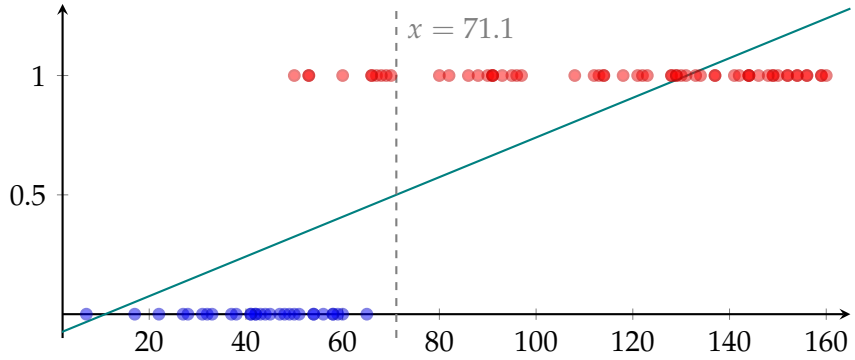


Figure 2.1: The training set for the "exam" problem. The straight line has been obtained by least squares linear regression, and the dashed line represents the decision threshold of the corresponding classifier.

Our students want to find a classifier $h : \mathbb{R} \to \{0, 1\}$ that could tell them, for a given number x of preparation hours, if it is more likely to pass the exam (1) or not (0). They decide to try with *linear least squares regression*, a widely used model for finding relationships in data. It consists in finding a straight line, defined by its slope w and its intercept b, that minimizes the average squared distance between the values on the line and the corresponding labels:

$$\min_{w,b} \frac{1}{m} \sum_{i=0}^{m-1} (wx_i + b - y_i)^2 . \tag{2.2}$$

One of the reasons why linear least squares is so popular is that it is easy to find the optimal solution $w^\star, b^\star$. In fact, it can be computed with the following formulae:

$$w^\star = \frac{\sum_{i=0}^{m-1}(x_i y_i - \mu_x \mu_y)}{\sum_{i=0}^{m-1}(x_i - \mu_x)^2}, \quad b^\star = \mu_y - w^\star \mu_x, \tag{2.3}$$

where μ_x is the average x_i and μ_y is the average y_i.

From a straight line we can derive the classifier $h_{\boldsymbol{\theta}}$ defined in terms of the two parameters $\boldsymbol{\theta} = (w, b)$. Given a value x, the classifier chooses the class label closest to the line:

$$h_{\boldsymbol{\theta}}(x) = \begin{cases} 1 & \text{if } \; wx + b \geq \frac{1}{2}, \\ 0 & \text{otherwise.} \end{cases} \tag{2.4}$$

Note that when $wx + b = \frac{1}{2}$ both labels would be equally reasonable. In that case we break the tie arbitrarily by choosing class 1. Figure 2.1 shows the results of applying linear least squares regression to the exam data. The classifier predicts 1 (exam passed) if the amount of study exceeds the 77.1 hours.

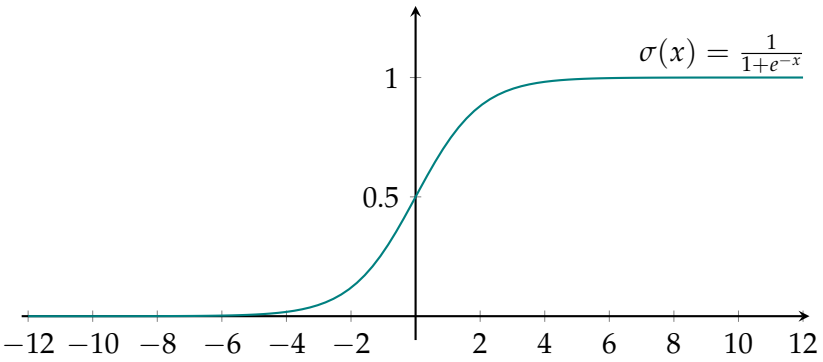


Figure 2.2: The sigmoid function.

However, this threshold is certainly too high! For instance, the classifier predicts a failure for a student who prepare himself for 70 hours, but everyone who studied that much passed the exam in the past.

Of the 90 samples in the training set the classifier assigns the wrong label to 10 cases (the *error rate* is about 11.1%). The reason why linear regression doesn't work well is that the model does not focus on the important region on the plot, but tries instead to fit all the data at the same time. The straight line can assume values that are very far away from the class labels 0 and 1, and those values influence the predictions more than they should. Adding more samples with $y = 1$ and large values of x (or with $y = 0$ and small x) would move the threshold to the right (or to the left) even though these new samples should not make any significant difference.

To obtain better predictions we should change the kind of function on which the classifier is based. Ideally we need a function that

- assumes values between 0 and 1;
- smoothly changes its values (i.e. is continuous and differentiable);

as we will see later on, smoothness helps most machine learning algorithms. A function that fulfills these conditions is the *sigmoid*:

$$\sigma(x) = \frac{1}{1+e^{-x}}. \tag{2.5}$$

The sigmoid is depicted in Figure 2.2: it assumes values close to 0 for very small values of its argument and approaches 1 for large values. The function is strictly increasing and slowly changes from its minimum to its maximum.

A classification model based on the sigmoid function is the *logistic regression*.

2.2 Logistic regression

The linear regression model has not been designed for classification and if we try to use it for that purpose, we might obtain bad results. Logistic regression, instead, has been especially defined to address binary classification problems. It is a simple model that combines a linear function with the sigmoid. First, the linear function is applied to compute a score z, called *logit*:

$$z = wx + b, \tag{2.6}$$

then the sigmoid transforms the logit into a value $\hat{p}$ in the range $(0, 1)$:

$$\hat{p} = \sigma(z) = \frac{1}{1 + e^{-z}}. \tag{2.7}$$

The logistic regression model assumes that, for a given x, the corresponding $\hat{p}$ is an estimate of the conditional probability that the correct class is 1:

$$\hat{p} \approx P(y = 1|x). \tag{2.8}$$

Given a training set $\{(x_0, y_0), \ldots, (x_{m-1}, y_{m-1})\}$ how can we find the best values $w^\star$ and $b^\star$ that optimally fit the data? A common approach, taken from statistics, is to apply the *maximum likelihood criterion*. Informally, the likelihood $\mathcal{L}$ represents how likely it is to observe the training set, given a particular choice of the parameters. If we assume that the training samples are independently drawn from some underlying unknown probability distribution, we can express the likelihood as the product of the probabilities (or of the densities, for continuous data) of observing the individual samples:

$$\mathcal{L} = \prod_{i=0}^{m-1} p(x_i, y_i). \tag{2.9}$$

The maximum likelihood criterion simply consists in choosing the parameters that maximize the likelihood, that is, in choosing the parameters that maximize the chance to observe the training set that we actually observed.

In practice, we prefer to optimize sums of things instead of products. In fact, instead of maximizing the likelihood is more common to minimize the *negative log likelihood*, that is simply minus the natural logarithm of the likelihood:

$$-\log \mathcal{L} = -\sum_{i=0}^{m-1} \log p(x_i, y_i), \tag{2.10}$$

where the logarithm transforms products in sums; being the logarithm a monotonic function, the values of the parameters maximizing the likelihood will be exactly the same that minimize the negative log likelihood. The change

in sign makes it easier to interpret the single terms as costs or errors, as it is common in machine learning (note that the logarithm of a probability is negative or zero). Another common manipulation is to use the Bayes' rule to introduce conditional probabilities:

$$\begin{aligned} -\log \mathcal{L} &= -\sum_{i=0}^{m-1} \log p(x_i, y_i) \\ &= -\sum_{i=0}^{m-1} \log(P(y_i|x_i)p(x_i)) \\ &= -\sum_{i=0}^{m-1} \log P(y_i|x_i) - \sum_{i=0}^{m-1} \log p(x_i), \end{aligned} \tag{2.11}$$

and this is useful since the second summation can be ignored if it does not depend on the parameters of the model.

Back to logistic regression, let $z_i = wx_i + b$ and $\hat{p}_i = \sigma(z_i)$. By the definition of the model we can express conditional probabilities as follows:

$$P(y_i|x_i) = \begin{cases} \hat{p}_i & \text{if } y_i = 1, \\ 1 - \hat{p}_i & \text{if } y_i = 0, \end{cases} \tag{2.12}$$

since the probability of a sample being of class 0 is one minus the probability of being of class 1. We can then take the negative logarithm, and write it as a single expression (remember that y_i can be only 0 or 1):

$$-\log P(y_i|x_i) = -y_i \log \hat{p}_i - (1 - y_i) \log(1 - \hat{p}_i). \tag{2.13}$$

This expression is called *cross entropy* and is a measure of the divergence between the label y_i and the predicted probability $\hat{p}_i$. Plugging it into the negative log likelihood, we obtain the following optimization problem:

$$\min_{w,b} \sum_{i=0}^{m-1} -y_i \log \hat{p}_i - (1 - y_i) \log(1 - \hat{p}_i), \tag{2.14}$$

where we considered only the first of the two sums in the negative log likelihood since the term $\log p(x_i)$ does not depend on w and b.

The solution of the optimization problem in the case of the exam data is represented in Figure 2.3 Note how the sigmoid function allow to model quite reasonably the probability of passing the exam. Starting from about 80 hours of study we have $\hat{p} \approx 1$ which means that, according to the model, who studies that much is guaranteed to pass the exam. Similarly, for values of about 30 hours or less $\hat{p} \approx 0$ (no chance to pass the exam). For intermediate values, the probability of passing the exam smoothly increases with the number of hours of study.

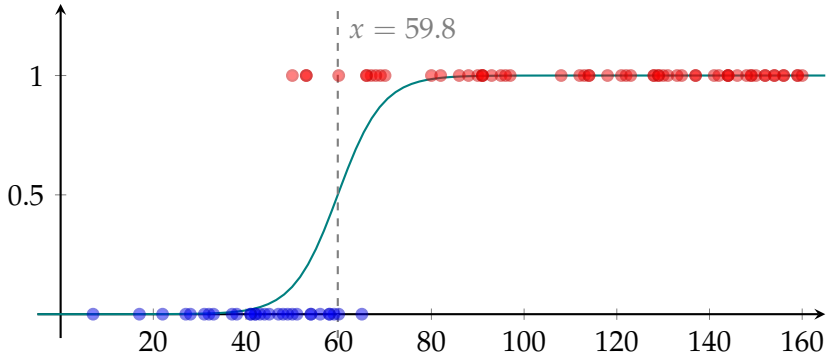


Figure 2.3: Logistic regression applied to the "exam" problem. The dashed line represents the threshold value corresponding to a 50% chance of passing the exam.

If we need a classifier to make sharp predictions we can define one that take the most probable of the two classes:

$$h_{\boldsymbol{\theta}}(x) = \begin{cases} 1 & \text{if } \hat{p} \geq \frac{1}{2}, \\ 0 & \text{otherwise,} \end{cases} \tag{2.15}$$

where $\boldsymbol{\theta} = (w, b)$ is the vector of parameters.

Logistic regression is a significant improvement over linear least squares regression. To begin with, it has a lower error rate: only 5 of the 90 training cases (about 5.5%) are labeled with the wrong class. Moreover, we can interpret it in a probabilistic way to take complex decisions. For instance, our students can use it to estimate the number of hours required to have a chance of 90% of passing the exam (just look for the x that solves the equation $\sigma(wx + b) = 0.9$). And these advantages are even more evident in more complex binary classification problems. One limitation of logistic regression is that the optimal values of its parameters cannot be found by applying a simple formula. However, as we will see, there are simple algorithms that we can use to find good approximations.

2.3 Multidimensional models

Most of the times a single feature is not enough to make accurate predictions. In fact, non trivial classification problems usually involve a large number of features. The introduction of multiple features only slightly changes the formulation of classification problems. Instead of having real-valued elements, the training set will include n-dimensional vectors:

$$\{(\mathbf{x}_0, y_0), (\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_{m-1}, y_{m-1})\}, \ \mathbf{x}_i \in \mathbb{R}^n, y_i \in \mathcal{Y}, \tag{2.16}$$

where the dimension n represents the number of features.

For logistic regression (and for many other classification models), to pass from one to multiple features it is enough to include one different weight w_i for each of the n components of the training vectors. The expression for the logit becomes:

$$z = w_0 x_0 + w_1 x_1 + \cdots + w_{n-1} x_{n-1} + b = \mathbf{w}^\intercal \mathbf{x} + b, \tag{2.17}$$

where $\mathbf{w}^\intercal \mathbf{x}$ denotes the scalar product between the vector of weights $\mathbf{w}$ and the vector of features $\mathbf{x}$. The estimated probability $\hat{p}$ is computed as before as the sigmoid of the logit $\sigma(z)$.

Similarly to the one dimensional case, the maximum likelihood criterion allows to find the value of the parameters $\boldsymbol{\theta} = (\mathbf{w}, b)$ as the solution of the following optimization problem:

$$\min_{\mathbf{w},b} \sum_{i=0}^{m-1} -y_i \log \hat{p}_i - (1 - y_i) \log(1 - \hat{p}_i). \tag{2.18}$$

The set of points where the classifier changes the predicted class ($\hat{p} = 0.5$) is called *decision boundary*. For logistic regression the decision boundary is a $n - 1$ dimensional linear subspace of $\mathbb{R}^n$. When there are $n = 2$ features the decision boundary is a straight line, when $n = 3$ it is a plane, when $n > 3$ it is a *hyperplane*. The classifiers of this kind are called *linear classifiers*.

The equation defining the decision boundary is

$$\mathbf{w}^\intercal \mathbf{x} + b = 0, \tag{2.19}$$

which has as solutions the points where the predicted probability is exactly 50%. From a geometrical point of view, the orientation of the weight vector $\mathbf{w}$ determines the orientation of the decision boundary. In fact, the hyperplane is orthogonal to $\mathbf{w}$. The sense of the vector $\mathbf{w}$ tells on which side of the hyperplane the classifier assigns the class 1. The bias b, instead, is related to the position of of the hyperplane. More precisely, the distance of the hyperplane from the origin is given by $|b|/\|\mathbf{w}\|$. Figure 2.4 shows these relationships for the bidimensional case.

From the point of view of the classification problem, the weights represent how much each individual feature contributes to the prediction of the classifier. The influence of a feature x_i depends on the magnitude $|w_i|$ of the corresponding weight (the larger the magnitude, the more important the feature). The sign of w_i tells if the probability for class $y = 1$ is increasing or decreasing with respect to x_i. A feature that is irrelevant for the classification is characterized by a weight close to 0. The bias b tells how much the classifier is biased towards class 1 (positive bias) or class 0 (negative bias).

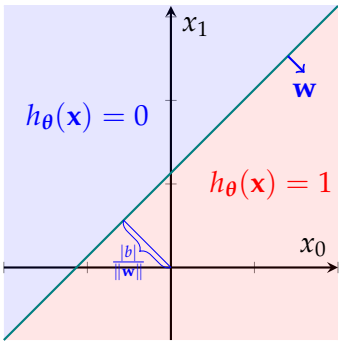


Figure 2.4: Decision boundary of a linear classifier. The boundary is a hyperplane (in two dimensions a straight line) dividing the space in two parts to which are assigned the two classes. The hyperplane is defined by the solution of the equation $\mathbf{w}^\intercal \mathbf{x} + b = 0$.

Note that scaling b and $\mathbf{w}$ by the same non-null factor does not change the decision boundary. However, it changes the probabilities estimated by the logistic model. The steepness of the sigmoidal ramp in fact depends on the magnitude $\|\mathbf{w}\|$. For small values of $\|\mathbf{w}\|$ the probability estimates slowly change from 0 to 1. For $\|\mathbf{w}\| \to \infty$ the estimates tend to change sharply from 0 to 1 around the decision boundary.

2.4 Predicting exams from two features

Coming back to the "exam" example, our students decide to try to improve the predictions of their classifier by including another feature. In addition to the number of hours of study (x_0) they also gather information about the number of hours of lectures to which each student attended (x_1). Then they computed the optimal weights $\mathbf{w} = (w_0, w_1)$ and bias b by minimizing the cross entropy. The decision boundary of the resulting classifier is depicted in Figure 2.5.

The new classifier is significantly more accurate than its one-dimensional version. In fact, only in one case the prediction does not match the correct class (1.11%). The weights of the classifier are: $w_0 = 0.6878$ and $w_1 = 0.2576$. Since they are both positive, we can infer that increments in both x_0 and x_1 contribute to improve the chances of passing the exam. The model also tells us that one hour of study is worth more than one hour of lecture. The bias is negative ($b = -50.0528$) which implies that the classifier is biased toward the class 0, that is, is quite unlikely to pass the exam with zero hours of both study and lectures.

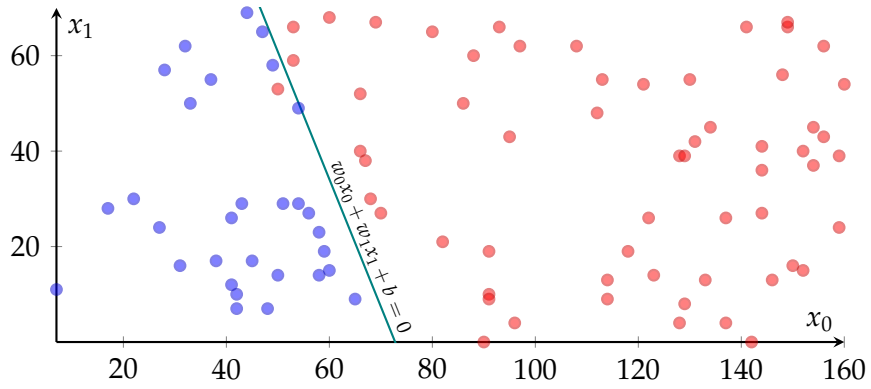


Figure 2.5: Logistic regression applied to the "exam" problem. The line corresponds to the *decision boundary* found by logistic regression, which is defined by the equation $w_0x_0 + w_1x_1 + b = 0$.

2.5 Optimization

Logistic regression usually deals with classification problem better than linear least squares. One disadvantage of logistic regression is that it has not a closed-form solution. Instead, like many other machine learning methods, an approximate solution is found with numerical optimization algorithms.

An optimization algorithm that is widely used for this purpose is *gradient descent*. It is an iterative algorithm that, starting from an initial solution, progressively improves it until it is not possible to make further progress. Gradient descent is inspired by the following analogy: suppose that you are lost on a mountain and that you want to reach the bottom of the valley. You proceed one step at a time, in the direction of the steepest descent until you reach a place where it is not possible to go further downwards (*steepest descent* is another name of this algorithm). Given a differentiable function, the direction of steepest descent is given by the opposite of its gradient. Each iteration of gradient descent improves its current solution by moving a bit in that direction.

For many machine learning methods the training procedure consists in finding the parameters $\boldsymbol{\theta}$ that minimizes a *loss function* $L(\boldsymbol{\theta})$ (also called *cost function*). Given an initial solution $\boldsymbol{\theta}$, gradient descent computes an updated solution $\boldsymbol{\theta}'$ as follows:

$$\boldsymbol{\theta}' \leftarrow \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}), \tag{2.20}$$

where $\nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta})$ is the gradient of L with respect to $\boldsymbol{\theta}$. The coefficient η is called *learning rate* and determines the length of the steps. The procedure starts with some initial vector $\boldsymbol{\theta}$ and continues to improve it until it converges. Convergence can be detected by some stopping criterion. A widely used one consists

in terminating the procedure when the difference $\|\boldsymbol{\theta}' - \boldsymbol{\theta}\|$ becomes smaller than a set threshold (this happens when the gradient is close to zero, which means that the current solution is close to a local minimum).

The gradient descent method can be implemented with a few lines of Python code:

```
def gradient_descent(loss_grad, theta, lr=0.001, steps=100):
    for _ in range(steps):
        grad = loss_grad(theta)
        theta -= lr * grad
    return theta
```

The function accepts the callable paramer: `loss_grad` representing $\nabla_{\boldsymbol{\theta}} L(\cdot)$, and an array `theta` representing the initial $\boldsymbol{\theta}$. It returns the solution found after `steps` iterations.

The method is guaranteed to converges to a *local minimum*, provided that the learning rate η is small enough. However, small values of η slow down the convergence of the method, so choosing a good value may be problematic. For convex functions the local minimum would be also the global one. Luckily this is the case of the loss function of logistic regression.

The loss function for logistic regression is the cross entropy that we previously defined as a function of its parameters $\boldsymbol{\theta} = (\mathbf{w}, b)$:

$$L(\mathbf{w}, b) = \frac{1}{m} \sum_{i=0}^{m-1} -y_i \log \hat{p}_i - (1 - y_i) \log(1 - \hat{p}_i). \tag{2.21}$$

Note that in practice it is common to optimize the average cross entropy instead of the sum, to contain its value in case of large training sets. The $1/m$ factor does not change the solution of the optimization problem.

To minimize the average cross entropy with the gradient descent algorithm we need to compute its gradient with respect to the parameters $\mathbf{w}$ and b. Let's define the cross entropy for a single training example as $L_i = -y_i \log \hat{p}_i - (1 - y_i) \log(1 - \hat{p}_i)$. The derivative of L_i with respect to b can be decomposed in simple terms by applying the chain rule:

$$\frac{\partial L_i}{\partial b} = \frac{\partial L_i}{\partial \hat{p}_i} \cdot \frac{\partial \hat{p}_i}{\partial z_i} \cdot \frac{\partial z_i}{\partial b}. \tag{2.22}$$

The first is:

$$\frac{\partial L_i}{\partial \hat{p}_i} = \frac{\partial}{\partial \hat{p}_i}(-y_i \log \hat{p}_i - (1 - y_i) \log(1 - \hat{p}_i)) = -\frac{y_i}{\hat{p}_i} + \frac{1 - y_i}{1 - \hat{p}_i}, \tag{2.23}$$

the second is:

$$\frac{\partial \hat{p}_i}{\partial z_i} = \frac{\partial}{\partial z_i} \frac{1}{1 + e^{-z_i}} = \frac{e^{-z_i}}{(1 + e^{-z_i})^2} = \frac{1}{1 + e^{-z_i}} \cdot \left(1 - \frac{1}{1 + e^{-z_i}}\right) = \hat{p}_i(1 - \hat{p}_i), \tag{2.24}$$

and the third is:

$$\frac{\partial z_i}{\partial b} = \frac{\partial}{\partial b}(\mathbf{w}^\intercal \mathbf{x_i} + b) = 1. \tag{2.25}$$

Finally, the product of the three derivatives results in:

$$\begin{aligned} \frac{\partial L_i}{\partial b} &= \left(-\frac{y_i}{\hat{p}_i} + \frac{1-y_i}{1-\hat{p}_i}\right) \cdot \hat{p}_i(1-\hat{p}_i) \cdot 1 = -y_i(1-\hat{p}_i) + (1-y_i)\hat{p}_i \\ &= -y_i + y_i\hat{p}_i + \hat{p}_i - y_i\hat{p}_i = \hat{p}_i - y_i, \end{aligned} \tag{2.26}$$

To get the derivatives with respect to the weights we can proceed in the same way:

$$\frac{\partial L_i}{\partial w_j} = \frac{\partial L_i}{\partial \hat{p}_i} \cdot \frac{\partial \hat{p}_i}{\partial z_i} \cdot \frac{\partial z_i}{\partial w_j}, \tag{2.27}$$

where the last derivative is:

$$\frac{\partial z_i}{\partial w_j} = \frac{\partial}{\partial w_j}(w_0 x_{i0} + w_1 x_{i1} + \cdots + w_{n-1} x_{i,n-1} + b) = x_{ij}, \tag{2.28}$$

that, combined with the other two, results in:

$$\frac{\partial L_i}{\partial w_j} = x_{ij}(\hat{p}_i - y_i). \tag{2.29}$$

Finally, the derivatives of the loss function for the entire training set is just the average of the derivatives just found for the single training samples:

$$\frac{\partial}{\partial b} L(\mathbf{w}, b) = \frac{1}{m} \sum_{i=0}^{m-1} \hat{p}_i - y_i, \tag{2.30}$$

$$\frac{\partial}{\partial w_j} L(\mathbf{w}, b) = \frac{1}{m} \sum_{i=0}^{m-1} x_{ij}(\hat{p}_i - y_i), \quad j = 0, \ldots, n-1, \tag{2.31}$$

and the latter can also be written more compactly in vector notation:

$$\nabla_{\mathbf{w}} L(\mathbf{w}, b) = \frac{1}{m} \sum_{i=0}^{m-1} \mathbf{x}_i(\hat{p}_i - y_i), \tag{2.32}$$

or even in a matrix form:

$$\nabla_{\mathbf{w}} L(\mathbf{w}, b) = \frac{1}{m} X^\intercal \cdot (\hat{\mathbf{p}} - \mathbf{y}), \tag{2.33}$$

where $\mathbf{y}$ and $\hat{\mathbf{p}}$ are the vectors composed of the m training labels and the corresponding estimates, and where X is a $m \times n$ matrix having as rows the feature vectors:

$$X = \begin{bmatrix} x_{0,0} & x_{0,1} & \cdots & x_{0,n-1} \\ x_{1,0} & x_{1,1} & \cdots & x_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ x_{m-1,0} & x_{m-1,1} & \cdots & x_{m-1,n-1} \end{bmatrix}. \tag{2.34}$$

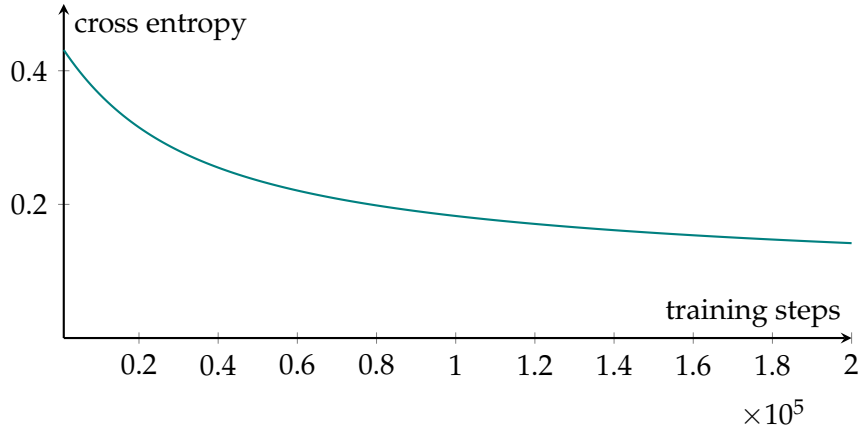


Figure 2.6: Average cross entropy for the "exam" problem, during the execution of the gradient descent algorithm. The whole training procedure included 200 000 iterations, with a learning rate of 10^{-3}.

The matrix form allows to write the corresponding Python code in a very compact way:

```
def logreg_inference(X, w, b):
    logits = X @ w + b
    probabilities = 1 / (1 + np.exp(-logits))
    return probabilities
```

Finally, the training procedure can be implemented by combining gradient descent with the functions above:

```
def logreg_train(X, Y, lr=1e-3, steps=1000):
    m, n = X.shape
    w = np.zeros(n)
    b = 0
    for step in range(steps):
        P = logreg_inference(X, w, b)
        grad_w = ((P - Y) @ X) / m
        grad_b = (P - Y).mean()
        w -= lr * grad_w
        b -= lr * grad_b
    return w, b
```

The result of calling `logreg_train` on the "exam" data was already shown in Figure 2.5. Figure 2.6 shows how gradient descent progressively reduces the cross entropy. However, it does so very slowly! After 200 000 steps the method still did not convergence (even though, in this case, more iterations would not improve the result). Other solutions found earlier during the training are depicted in Figure 2.7. We may try to accelerate the procedure by increasing the learning rate, that was set to 10^{-3}. However, larger learning

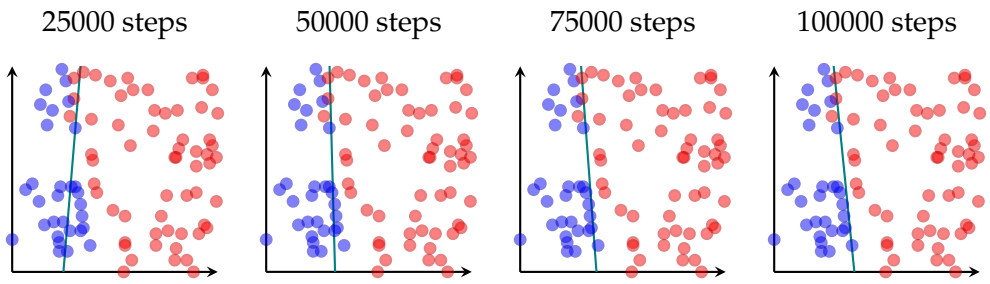


Figure 2.7: Decision boundary after 25 000, 50 000, 75 000 and 100 000 training steps.

rates makes the first steps too large to be really useful. A smaller learning rate, instead, would just increase the number of iterations required to get to a reasonable solution.

Second-order methods

Gradient descent is very simple and, for convex functions, its convergence to the minimum is guaranteed (provided that the learning rate is small enough). A faster convergence, however, can be achieved by more advanced optimization methods. A very quick alternative, for instance, is the Newton method. It uses second-order derivatives to make large jumps in the parameters space. The updated parameters $\boldsymbol{\theta}'$ are computed as follows:

$$\boldsymbol{\theta}' \leftarrow \boldsymbol{\theta} - (H_L(\boldsymbol{\theta}))^{-1} \nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}), \tag{2.35}$$

where $H_L(\boldsymbol{\theta})$ is the Hessian matrix of L computed in $\boldsymbol{\theta}$. Note that there is no learning rate. This algorithm may converge in a very small number of steps. Its main drawback is that it requires the computation (and the inversion) of the Hessian Matrix, which can be very large. Its size is the square of the number of parameters which, in some cases, can be very large (modern machine learning models may include millions of parameters).

A more viable option is represented by *quasi-Newton* methods, that approximate the Hessian, or its inverse, insead of computing it. A very popular choice is the *Limited-memory BFGS* method. Each of its steps requires time and memory that are linear in the number of parameters. Anyway, when the number of parameters is too large, gradient descent and its variants is the only feasible option.

2.6 Multinomial logistic regression

The most evident shortcoming of logistic regression is that it is restricted *binary classification*. In many classification problems the samples may belong to more than just two classes. Without loss of generality we may denote them with the numerical labels $\mathcal{Y} = \{0, 1, \ldots, k-1\}$. The *multinomial logistic regression* model is a generalization of logistic regression that works with more than two classes.

Given a sample $\mathbf{x} \in \mathbb{R}^n$, predicts a vector of k values representing the estimates of the probabilities that $\mathbf{x}$ belongs to each of the k classes.

Instead of a weight vector the model uses an independent set of weights for each class. The weights form a $k \times n$ matrix W, with W_{ij} being the weight that is applied to x_j for the computation of the score for the i-th class. There is also a set of biases, one for each class, organised in a a k-dimensional vector $\mathbf{b}$. Different logits *logit* $z_0, z_1, \ldots, z_{k-1}$ are computed as follows:

$$z_i = \sum_{j=0}^{n-1} W_{ij} x_j + b_i, \tag{2.36}$$

which can also be written in vector form:

$$\mathbf{z} = W\mathbf{x} + \mathbf{b}, \tag{2.37}$$

Logits are then transformed into a vector $\hat{\mathbf{p}}$ of probability estimates by applying the *softmax* operator $\mathbb{R}^k \to \mathbb{R}^k$:

$$\hat{\mathbf{p}} = \text{softmax}(\mathbf{z}), \quad \hat{p}_c = \frac{e^{z_c}}{\sum_{j=0}^{k-1} e^{z_j}}, \quad c = 0, 1, \ldots, k-1, \tag{2.38}$$

The name of the operator comes from the fact that it tends to smoothly assign high values (close to one) to the component corresponding the the largest logit. It can be considered as an extension of the sigmoid (in the case of two logits z_0 and z_1 the softmax would give $\hat{p}_0 = \sigma(z_0 - z_1)$ and $\hat{p}_1 = \sigma(z_1 - z_0)$).

Note that all the elements of the vector $\hat{\mathbf{p}}$ are non negative and that they sum up to one. These properties make it suitable to represent a probability distribution over the classes. In fact we may use it to model the conditional posterior probability that the true class is c, given that we observed the features $\mathbf{x}$:

$$P(y = c|\mathbf{x}) \approx \hat{p}_c. \tag{2.39}$$

The final decision $\hat{y}$ of the model is just the class that maximizes such a probability:

$$\hat{y} = \underset{c \in \{0,\ldots,k-1\}}{\arg\max}\ \hat{p}_c = \underset{c \in \{0,\ldots,k-1\}}{\arg\max}\ z_c, \tag{2.40}$$

note that according to the definition of the sofmax operator, the largest probability corresponds to the largest logit.

Training procedure

Training the multinomial logistic regression model consists in finding W and $\mathbf{b}$ that give high probability estimates for the true classes in a training set, and low estimates for the others. Given a training set of m samples

$$\{(\mathbf{x}_0, y_0), (\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_{m-1}, y_{m-1})\}\,,\ \mathbf{x}_i \in \mathbb{R}^n, y_i \in \{0, 1, \ldots, k-1\}, \quad (2.41)$$

we observe that for the i-th sample:

$$p(\mathbf{x}_i, y_i) = p(y_i|\mathbf{x}_i)p(\mathbf{x}_i) \approx \hat{p}_{i,y_i} p(\mathbf{x}_i), \quad (2.42)$$

where $\hat{\mathbf{p}}_i = \text{softmax}(\mathbf{z}_i)$ is the vector of probability estimates for the i-th training sample, and therefore $\hat{p}_{i,y_i}$ is the estimated probability that the model assigns to the actual class label y_i. Similalry to the logistic regression case we proceed to minimize the negative log likelihood:

$$-\log \mathcal{L} = -\sum_{i=0}^{m-1} \log p(\mathbf{x}_i, y_i) = -\sum_{i=0}^{m-1} \log \hat{p}_{i,y_i} - \sum_{i=0}^{m-1} \log p(\mathbf{x}_i). \quad (2.43)$$

The second term is independent on the class label and can be therefore ignored. The first term can be simplified if we express the class labels with *one-hot vectors*. An one-hot vector is a vector in which all components are zero except a single one which has value 1 (they are called like this because the are "hot" in just one place). The one-hot vector for class $c = 0$ is $(1, 0, \ldots, 0)$, the one for class $c = 1$ is $(0, 1, 0, \ldots, 0)$, and so on. In the following we will denote as $\bar{\mathbf{y}}_i$ the one-hot vector for the i-th training sample:

$$\bar{y}_{ic} = \begin{cases} 1 & \text{if } y_i = c, \\ 0 & \text{otherwise.} \end{cases} \quad (2.44)$$

One-hot vectors can be used to define the *cross entropy* L_i for the i-th training sample:

$$L_i = -\log \hat{p}_{i,y_i} = -\sum_{c=0}^{k-1} \bar{y}_{ic} \log \hat{p}_{ic}, \quad (2.45)$$

The cross entropy can be considered as a measure of the divergence between the estimated distribution of class probabilities, with the target distribution, where the target distribution is the one-hot vector. The cross entropy would be zero when these two distributions are identical and is large if they are very different.

Cross entropy can be used to express the optimization problem for training the multinomial logistic regression model as follows (it is common to minimize the average cross entropy instead of the sum):

$$\min_{W,\mathbf{b}} J(W, \mathbf{b}) = \frac{1}{m} \sum_{i=0}^{m-1} L_i. \quad (2.46)$$

Optimization

Problem (2.46) is convex, and its solution can be approximated by gradient descent or with similar methods. To do so we just have to compute the gradient of the objective function with respect to W and $\mathbf{b}$. First let's define L_i (the cross entropy for the i-th sample) as a function of $\mathbf{z}_i = W\mathbf{x}_i + \mathbf{b}$:

$$\begin{aligned} L_i &= -\sum_{c=0}^{k-1} \bar{y}_{ic} \log \hat{p}_{ic} = -\sum_{c=0}^{k-1} \bar{y}_{ic} \log \frac{e^{z_{ic}}}{\sum_{j=0}^{k-1} e^{z_{ij}}} = -\sum_{c=0}^{k-1} \bar{y}_{ic} \left(\log e^{z_{ic}} - \log(\sum_{j=0}^{k-1} e^{z_{ij}}) \right) \\ &= \left(-\sum_{c=0}^{k-1} \bar{y}_{ic} z_{ic} \right) + \left(\sum_{c=0}^{k-1} \bar{y}_{ic} \log \sum_{j=0}^{k-1} e^{z_{ij}} \right) = -\left(\sum_{c=0}^{k-1} \bar{y}_{ic} z_{ic} \right) + \log \sum_{j=0}^{k-1} e^{z_{ij}}, \end{aligned} \tag{2.47}$$

where the first step depends on the definition of the softmax operator, and the last on the fact that $\sum_c \bar{y}_{ic} = 1$. From this we can compute the derivative of the cross entropy:

$$\frac{\partial}{\partial z_{id}} L_i = -\bar{y}_{id} + \frac{e^{z_{id}}}{\sum_{j=0}^{k-1} e^{z_{ij}}} = -\bar{y}_{id} + \hat{p}_{id}. \tag{2.48}$$

The derivatives of the objective function with respect to the parameters can be obtained by applying the chain rule (remember that $z_{id} = \sum_{j=0}^{n-1} W_{dj} x_{ij} + b_d$):

$$\frac{\partial}{\partial W_{dj}} J(W, \mathbf{b}) = \frac{1}{m} \sum_{i=0}^{m} \frac{\partial L_i}{\partial z_{id}} \frac{\partial z_{id}}{\partial W_{dj}} = \frac{1}{m} \sum_{i=0}^{m} (\hat{p}_{id} - \bar{y}_{id}) x_{ij}, \tag{2.49}$$

which can be rewritten in vector form as:

$$\nabla_W J(W, \mathbf{b}) = \frac{1}{m} \sum_{i=0}^{m-1} (\hat{\mathbf{p}}_i - \bar{\mathbf{y}}_i) \mathbf{x}_i^\intercal. \tag{2.50}$$

We can obtain a similar expression for the parameter $\mathbf{b}$:

$$\nabla_{\mathbf{b}} J(W, \mathbf{b}) = \frac{1}{m} \sum_{i=0}^{m-1} (\hat{\mathbf{p}}_i - \bar{\mathbf{y}}_i). \tag{2.51}$$

A possible implementation in the Python language is reported here below. The functions process a matrix of feature vectors (one vector per row).

```
def multinomial_logreg_inference(X, W, b):
    logits = X @ W + b.T
    # softmax
    probs = np.exp(logits) / np.exp(logits).sum(1, keepdims=True)
    return probs
```

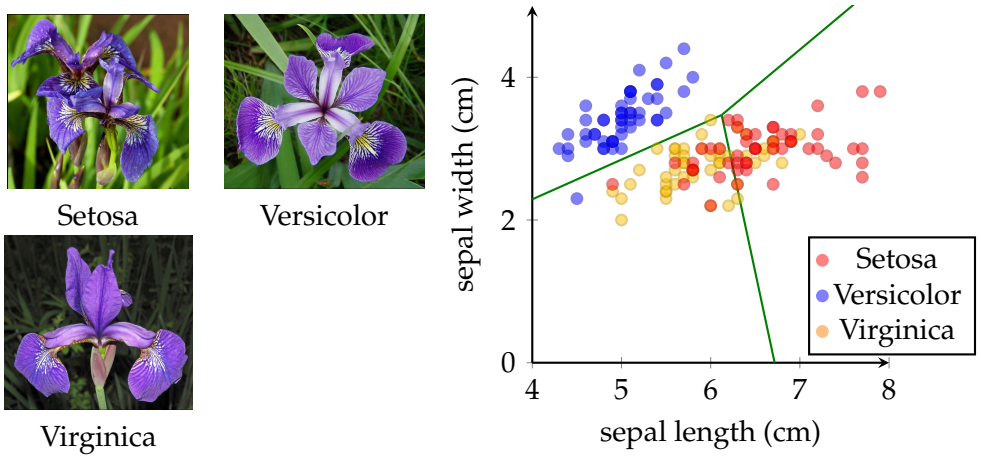


Figure 2.8: Example of a multi-class problem: classification into three different kinds of iris. The decision boundary has been found by multinomial logistic regression.

```
def multinomial_logreg_train(X, Y, lr=1e-3, steps=1000):
    m, n = X.shape
    k = Y.max() + 1  # number of classes
    W = np.zeros((n, k))
    b = np.zeros(k)
    # Build the one hot vectors H
    H = np.zeros((m, k))
    H[np.arange(m), Y] = 1
    for step in range(steps):
        P = multinomial_logreg_inference(X, W, b)
        grad_W = (X.T @ (P - H)) / m
        grad_b = (P - H).mean(0)
        W -= lr * grad_W
        b -= lr * grad_b
    return W, b
```

Example: iris classification

A classic example for multi-class classification is given by the Iris dataset. The data set contains 3 classes of 50 instances each, where each class refers to a type of iris plant (*setosa*, *versicolor* and *virginica*). Samples consist in four features: the length of the petal, the widht of the petal, the length of the sepal (another part of the flower), and the width of the sepal. Figure 2.8 shows the result of applying multinomial logistic regression (only the two sepal features are used). One of the classes can be easily separated from the others. However the distributions of iris versicolor and virginica overlap.

Note how multinomial logistic regression is able to deal reasonably with both cases. Note also that the decision boundary is still linear.

2.7 Summary

We introduced the concept of classification and we shown a first example of classification problem together with a possible solution. More in detail in this lecture we covered the following topics:

- we defined classification as the problem of assigning items to classes or categories;
- we introduced the idea of tackling classification problems by supervised learning, that is, by learning from the examples in a training set;
- we introduced logistic regression as a model suitable for binary classification; we shown why it works better than linear least squares;
- we stated the maximum likelihood as a criterion for the selection of the parameters;
- we showed how, for the logistic model, minimizing the negative log likelihood leads to the minimization of the average cross entropy;
- we used the gradient descent algorithm to implement a learning algorithm;
- we introduced the multinomial logistic regression model as a multiclass extension of the logistic regression model.

Linear least squares (optional)

To define the linear regression model we can take the same steps that we used to introduce logistic regression. This time, however, we are dealing with a *regression* problem, where we are interested in real-valued predictions.

Given a training set $(\mathbf{x}_0, y_0), \ldots, (\mathbf{x}_{m-1}, y_{m-1})$, with $\mathbf{x}_i \in \mathbb{R}^n$ and $y_i \in \mathbb{R}$. The model assumes that the target value y_i follows a Gaussian distribution centered around the linear prediction $\mathbf{w}^\intercal \mathbf{x}_i + b$:

$$p(y_i|\mathbf{x}_i) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{\left(y_i - (\mathbf{w}^\intercal \mathbf{x}_i + b)\right)^2}{2\sigma^2}}, \tag{2.52}$$

where σ^2 is the variance, that is assumed to be uniform for all the samples. In practice, the model assumes that the output values are the result of a linear deterministic process corrupted by some Gaussian noise.

The best parameters can be found minimizing the negative log likelihood:

$$
\begin{aligned}
-\log \mathcal{L} &= -\log \prod_{i=0}^{m-1} p(y_i|\mathbf{x}_i)p(\mathbf{x}_i) = \sum_{i=0}^{m-1} -\log p(y_i|\mathbf{x}_i) + \sum_{i=0}^{m-1} -\log p(\mathbf{x}_i) \\
&= m \log \sigma\sqrt{2\pi} + \frac{1}{2\sigma^2} \sum_{i=0}^{m-1} \left(y_i - (\mathbf{w}^\intercal \mathbf{x}_i + b)\right)^2 + \sum_{i=0}^{m-1} -\log p(\mathbf{x}_i).
\end{aligned}
\tag{2.53}
$$

Only one term depends on the parameters $\mathbf{w}$ and b, the others can be ignored. Applying the maximum likelihood criterion corresponds to the solution of the following optimization problem:

$$
\min_{\mathbf{w},b} L(\mathbf{w}, b) = \frac{1}{m} \sum_{i=0}^{m-1} \left(y_i - (\mathbf{w}^\intercal \mathbf{x}_i + b)\right)^2, \tag{2.54}
$$

where the factor $1/m$ is introduced, without changing the solution, to make the loss function L easier to interpret. In this form it is called *mean squared error*, or simply *MSE*.

The MSE is clearly convex with respect to the parameters, therefore the solution can be found by determining the values that makes null the gradient. It is easier to do so by using a matrix notation. Let's define the *characteristic matrix* $A \in \mathbb{R}^{m\times(n+1)}$:

$$
A = \begin{bmatrix}
x_{0,0} & x_{0,1} & \dots & x_{0,n-1} & 1 \\
x_{1,0} & x_{1,1} & \dots & x_{1,n-1} & 1 \\
\vdots & \vdots & \ddots & \vdots & \vdots \\
x_{m-1,0} & x_{m-1,1} & \dots & x_{m-1,n-1} & 1
\end{bmatrix}, \tag{2.55}
$$

which includes one feature vector per row, terminated by a trailing 1. We will use a single vector of $\boldsymbol{\theta}$ obtained by appending b at the end of the vector $\mathbf{w}$. The minimization of the MSE can be then written as:

$$
\min_{\boldsymbol{\theta}} L(\boldsymbol{\theta}) = \frac{1}{m} \left\|\mathbf{y} - A\boldsymbol{\theta}\right\|^2, \tag{2.56}
$$

where $\mathbf{y}$ is the vector of the target values and where the norm $\|\cdot\|^2$ is used as a convenient way to denote the sum of the squares.

It is easy to derive the gradient of the loss, obtaining the following expression:

$$
\nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}) = -\frac{2}{m} A^\intercal (\mathbf{y} - A\boldsymbol{\theta}), \tag{2.57}
$$

which becomes null when $\boldsymbol{\theta}$ is the solution of the following system of $n+1$ linear equations:

$$
(A^\intercal A)\boldsymbol{\theta} = A^\intercal \mathbf{y}. \tag{2.58}
$$

Note that this solution is just a generalization of that reported in Equation 2.3 for the one-dimensional case.

Unlike logistic regression, linear regression has a closed form solution. However, for very large feature vectors (e.g. $n > 10\,000$) the solution may be too expensive to compute. In that cases it may be preferable to fallback to an approach based on an iterative optimization algorithm, such as gradient descent.

Lecture 3

Generalization and regularization

Good students don't just memorize the content of their textbooks. Instead, they understand general principles and learn to apply them in different situations. Similarly, a good classifier cannot be recognized only from how well it reproduces the data in the training set.

The strategy we considered so far for supervised learning consists in minimizing the differences between the actual labels of the training examples and those predicted by a classifier. This approach may sound perfectly logical, but it does not necessarily agree with the goal of machine learning. Learning cannot be reduced to the problem of reproducing the labels seen during training: for that it is enough to have good memory! The real goal of machine learning is to make correct predictions for new data, that was not used for training. The ability to make good predictions for previously unobserved data is called *generalization*.

3.1 Training and test sets

To assess how much a classifier generalizes we have to look outside the training set. The simplest approach consists in using a separate set of examples. Such a set is called *test set*. Just like the training set, the test set is made of samples of which we know the corresponding labels:

$$\{(\tilde{x}_0, \tilde{y}_0), (\tilde{x}_1, \tilde{y}_1), \ldots, (\tilde{x}_{s-1}, \tilde{y}_{s-1})\}\,,\ \tilde{x}_i \in \mathcal{X}, \tilde{y}_i \in \mathcal{Y}. \tag{3.1}$$

Differently from the training set, **the test set is not used during training**. When the training process is complete, the test set allows to obtain a fair evaluation of how good is the resulting model in making predictions.

We can assess the generalization capability of a classifier by comparing its training and test performance. The comparison can be based on the same

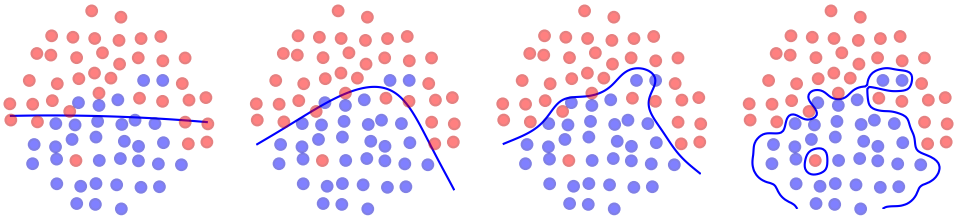


Figure 3.1: A training set and the decision boundaries of four different binary classifiers. The complexity of the classifiers increaeses from left to right.

loss function used for training, or by simply computing the *accuracy* on the two sets. The accuracy is defined as the fraction of samples that are correctly classified (some people prefer to use the *error rate* which is just the fraction of incorrectly classified samples).

Very often the performance on the training set are better (lower loss or higher accuracy) than those computed on the test set. This is understandable, since it is certainly easier to make correct predictions on the data seen during training. When the difference is very large, we say that there is an *overfitting* of the training set. A model that overfits is like a student that memorizes the textbooks without really learning anything.

This concept is illustrated in Figure 3.1 which shows one training set for a binary classification problem and four different classifiers. The rightmost classifier is an example of overfitting. Its decision boundary is clearly too complex as it learned to accomodate all the training examples regardless how much they look noisy or exceptional. Let's imagine a test set for the same problem: many of its element close to the boundary would probably be on the wrong side.

The leftmost classifier in Figure 3.1 is not good either. It's accuracy on the training set is quite bad (there are 12 errors), so it is a meager consolation that it would probably generalize well on a test set. This condition is called *underfitting*, and is a consequence of a classifier that is "too simple" to model the classes.

To summarize, the goal of a learning algorithm is to find a model that (i) performs well on the training set; and (ii) generalizes to new data. Failing the first objective leads to underfitting, failing the second causes overfitting. Overfitting and underfitting are two opposite conditions representing failures of the learning process and both must be carefully avoided by choosing the "right" level of complexity of the classifier. For instance, the two central classifiers in Figure 3.1 seem good. Some centuries ago, William of Ockham proposed this general principle (today known as *Occam's razor*): "Entities should not be multiplied without necessity", which is sometimes

paraphrased as "the simplest solution is most likely the right one".

However, to avoid overfitting and underfitting we can't rely just on our intuition, we need suitable methods and techniques. In order to introduce them we have first to reconsider the classification problem from a probabilistic point of view.

3.2 Empirical risk minimization

Generalization is a property arising from statistics. Training and test sets are composed of samples of un underlying process that we assume to be, up to some degree, predictable. Predictability comes from the assumption that the laws governing that process does not arbitrarily change and that they exhibit some regularity that is reflected in the data.

We can model the process underlying a supervised learning problem with a probability distribution $\mathcal{D}$ over the pairs in $\mathcal{X} \times \mathcal{Y}$. We can think to that distribution as a source that we can use to generate a stream of pairs:

$$(x_0, y_0), (x_1, y_1), \ldots, (x_i, y_i), \ldots \tag{3.2}$$

Each pair (x_i, y_i) is supposed to be drafted according to the probability distribution $\mathcal{D}$. A very common assumption is that all the elements of the training set are mutually independent. This means that the occurrence of a given training pair does not influence the probability of sampling the other pairs. Following the terminology used in probability theory, we say that training samples are *independent and identically distributed*, a property that is usually abbreviated as i.i.d.

Ideally, we would like to find the classifier that makes the smallest amount of errors. The chance for a classifier h to make a mistake is called *risk* and is defined as the probability that for a pair (x, y) drafted from $\mathcal{D}$ the output of the classifier and the class label disagree:

$$R(h) = P(h(x) \neq y), \quad (x, y) \sim \mathcal{D}. \tag{3.3}$$

In the following we will use the terms "risk" and "error" as synonimes.

The classifier $h^\star$ minimizing the risk can be defined in a relatively simple way:

$$h^\star = \arg\min_h R(h), \tag{3.4}$$

$$h^\star(x) = \arg\max_{y \in \mathcal{Y}} P(y|x), \tag{3.5}$$

that is, the optimal classifier is the one that, given x, chooses the class y that maximizes the conditional probability $P(y|x)$.

Thanks to the Bayes' rule we can express this conditional probability in terms of the probability density function $p_{\mathcal{D}}(x, y)$ of the distribution $\mathcal{D}$:

$$P(y|x) = \frac{p_{\mathcal{D}}(x,y)}{p(x)} = \frac{p_{\mathcal{D}}(x,y)}{\sum_{y' \in \mathcal{Y}} p_{\mathcal{D}}(x,y')}, \tag{3.6}$$

where the denominator comes from the fact that each sample belongs to exactly one class so that we can apply the total probability rule: $p(x) = \sum_{y' \in \mathcal{Y}} p_{\mathcal{D}}(x, y')$. The risk $R(h^\star)$ is the lowest possible achievable, and it is called *Bayes error*. For some problems the Bayes error is zero. This happens when, for any x, there is only one possible correct choice for y. However, there are many problems for which the optimal error is not zero. As an extreme case, just consider the problem of guessing the outcome of a coin toss: for this problem the Bayes error is 0.5.

So, why don't we just use the optimal classifier $h^\star$? Unfortunately, in general, we don't know the distribution $\mathcal{D}$! Therefore we cannot use it to compute $h^\star(x)$. Supervised learning methods search for a good approximation of $h^\star$ by leveraging the information contained in a training set of pairs sampled from $\mathcal{D}$.

The strategy of choosing the classifier by minimizing the amount of errors in the training set is called *empirical risk minimization*. The empirical risk R_{emp} of classifier h is defined as the fraction of training cases for which the output does not match the class label:

$$R_{\text{emp}}(h) = \frac{1}{m} \sum_{i=0}^{m-1} E_i, \quad E_i = \begin{cases} 1 & \text{if } h(x_i) \neq y_i, \\ 0 & \text{if } h(x_i) = y_i. \end{cases} \tag{3.7}$$

The term "empirical" simply means that we count the errors that we observe instead of deriving their amount from a theory or by logic.

By minimizing the empirical risk (or some other loss function) we obtain the classifier $h^\dagger$:

$$h^\dagger = \arg\min_h R_{\text{emp}}(h). \tag{3.8}$$

Beware, however, that the minimization of the empirical risk does not necessariliy imply the minimization of the "real" risk R. The minimization of R_{emp} , in fact, results in a *biased* estimate of the real risk, because it tend to adapt the classifier to random patterns in the training data that are not necessarily present in other data sampled from $\mathcal{D}$.

Despite the fact that we cannot compute the real risk, we can approximate it on an independent test set of pairs sampled from $\mathcal{D}$. The disagreement between $h^\star$ and $h^\dagger$ (in practice the overfitting) can be limited if $h^\dagger$ is restricted to belong to a set fo classifiers of a limited complexity. Statistical learning theory provides many theoretical results about this issue that are beyond the scope of our current discussion.

The performance on the test set are a more reliable estimate of the goodness of a classifier than those measured on the training set. The measures made on the test set are accurate provided that the following conditions are met:

- the test set is large enough to allow statistically significant measurements;
- the elements in the training and in the test set are sampled from the same underlying distribution;
- only the training set is used to train the classifier, and the test set is used only to assess its performance.

Model complexity

The easier for a model to adapt to any training set, the more prone to overfitting it is. Therefore, to improve generalization we need to control the *complexity* (also called *capacity*) of the model. A rough measure of the level of complexity is given by the number of parameters of the model (a more precise measure is given at the end of the lecture). These usually depends on the dimensionality of the feature space. For instance, the logistic regression model for a n-dimensional feature space includes $n + 1$ parameters (n weights and one bias). Even linear classifiers can overfit if there is a large number of features. In fact, when there are many features it becomes likely to find spurious correlations with the class label, especially when the training set is small. Suppose to take ten students, five who passed an exam and five who did not. Consider to use as features: the height of the students, their weight, their age, the size of their feet... and another few dozens. We may discover that one of these, just by chance, allows to identify whose who passed the exam.

Increasing the size of the training set will eventually cause a reduction in the generalization error for any classification model, provided that its complexity is finite. In this case, in fact, at some point the training algorithm will need to find an efficient way to correlate the feature vectors with the training labels, avoiding overfitting. The lower the complexity of the model, the sooner this happens.

For models of infinite complexity, however, no guarantees exist. These models typically have a number of parameters that grows linearly with the number of training samples and they can overfit arbitrarily large training sets.

Nearest neighbor

A very straightforward classification strategy is implemented by the *nearest neighbor* (NN) classifier. A NN classifier memorizes all training pairs and, when asked to predict the class of a new sample $\mathbf{x}$, it returns the class label of the closest training sample (ties are arbitrarily broken). The decision function h is therefore the following:

$$h(\mathbf{x}) = y_{\hat{\imath}(\mathbf{x})}, \quad \hat{\imath}(\mathbf{x}) = \arg\min_j \|\mathbf{x} - \mathbf{x}_j\|, \tag{3.9}$$

where the Euclidean distance can be replaced by any other distance measure. Nearest neighbor is very simple (it does not require a true training algorithm) and in some cases it also works reasonably well. A possible implementation in Python is given here below. To speed-up the computation it uses the equality $\|\mathbf{x} - \mathbf{x}'\|^2 = \|\mathbf{x}\|^2 + \|\mathbf{x}'\|^2 - 2\mathbf{x}^\mathsf{T}\mathbf{x}'$.

```
def nearest_neighbor_inference(X, Xtrain, Ytrain):
    X_squared = (X ** 2).sum(1)
    Xt_squared = (Xtrain ** 2).sum(1)
    prod = 2 * X @ Xtrain.T
    d_squared = X_squared[:, None] + Xt_squared[None, :] - prod
    index = np.argmin(d_squared, 1)
    labels = Ytrain[index]
    return labels
```

Nearest neighbor is a notable example of a classifier with infinite capacity. It makes no training errors (unless there are identical samples with different class labels) and it adapts without any effort to training sets of arbitrary size. However, this does not imply a low test error. In practice, nearest neighbor always works great on the training set, but little can be said about how well it generalizes (sometimes it does!). Figure 3.2 shows the nearest neighbor classifier applied to a binary classification problem.

Some additional considerations on the nearest neightbor classifier:

- it is a simple model, which is easy to implement and requires no explicit training. These properties make it a good candidate as a first classification strategy to explore a new problem.
- It is computationally demanding, in particular when there are many training samples to memorize and to compare against.
- It handles multiple classes without problems.
- Its accuracy depends on the distance measure used, which must represent well the similarity between pairs of feature vectors.
- Having inifinite capacity (the parameters are all the training features and labels), it often generalizes poorly.

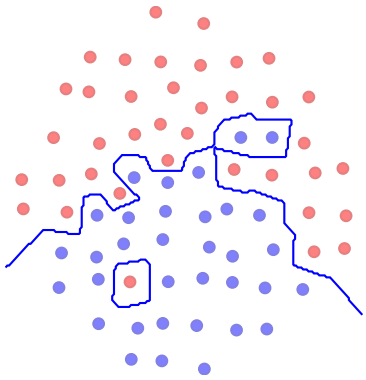


Figure 3.2: Decision boundary of the nearest neighbor classifier applied to a binary classification problem.

3.3 Regularization

Avoiding useless features is a good start to build an effective classifier by reducing the number of its parameters. In most cases, however, the number of potentially useful features can be very large and we cannot reduce it without the risk of excluding also the good ones. Another way to limit the complexity of a classifier is to constrain the values of its parameters. This is usually achieved by introducing an additional term in the objective function that is optimized by the learning algorithm. This technique is called *regularization*. Supervised learning with regularization usually consists in minimizing an objective function $J_\lambda(\boldsymbol{\theta})$ having the following structure:

$$\min_{\boldsymbol{\theta}} J_\lambda(\boldsymbol{\theta}) = \frac{1}{m} \sum_{i=0}^{m-1} L(h_{\boldsymbol{\theta}}(x_i), y_i) + \lambda R(\boldsymbol{\theta}). \tag{3.10}$$

The first term of J is the average of a loss function over the training set. By minimizing it, we obtain a classifier with a small training error. The second term $R(\boldsymbol{\theta})$ is used to favor certain values for the parameters $\boldsymbol{\theta}$, implicitly constraining it to a subset of possible optimal values. The coefficient $\lambda \geq 0$ is used to balance the contribution of the two terms: for $\lambda = 0$ we fall back to unregularized learning, with a small training error and a high risk of overfitting. For $\lambda \to +\infty$ the regularizion term becomes dominant, imposing a strong constraint to the parameters of the classifier that improves its generalization, but also increases the training error (underfitting).

Regularized logistic regression

Let's see how regularization can be used to improve the generalization of a linear classifier. The main idea is that generalization can be achieved by

favoring the distribution of the weights of the classifier over a large number of features. To do so, we may push the learning algorithm to select a "small" vector of weights, for instance one with a small sum of squares $w_0^2 + w_1^2 + \cdots + w_{n-1}^2$. In fact, the vector $(\frac{1}{2}, \frac{1}{2})$ that considers two features has a smaller sum of squares than the vector $(1, 0)$ which ignores one of them. This approach is called L_2 *regularization* or *weight decay*, and corresponds to the introduction of the following regularization term:

$$R(\mathbf{w}) = \sum_{k=0}^{n-1} w_k^2 = \|\mathbf{w}\|^2, \tag{3.11}$$

where $\|\cdot\|$ denotes the Euclidean norm (also called L_2 norm).

Regularized logistic regression is a popular classification technique that consists in minimizing the cross entropy (the loss function of logistic regression) combined with L_2 regularization:

$$\min_{\mathbf{w},b} J_\lambda(\mathbf{w}, b) = \frac{1}{m} \sum_{i=0}^{m-1} (-y_i \log(\hat{p}_i) - (1 - y_i) \log(1 - \hat{p}_i)) + \lambda \|\mathbf{w}\|^2. \tag{3.12}$$

For logistic regression L_2 regularization has a simple interpretation: the length of the weight vector $\|\mathbf{w}\|$ determines the "steepness" of the probability estimate $\hat{p} = \sigma(\mathbf{w}^\intercal \mathbf{x} + b)$. For $\|\mathbf{w}\| = 0$ this estimate is flat, and equal to $\frac{1}{2}$. For $\|\mathbf{w}\| \to +\infty$ the estimate tends to a step function suddendly jumping from 0 to 1. The consequence of this is that for moderately small values of the weights the learning algorithm penalizes decision boundaries close to the training samples. Note that the bias b does not contribute to regularization.

The solution to problem (3.12) can be found by using general purpose optimization algorithms suchs as gradient descent. To do so, we need to compute the gradient of the objective function with respect to the parameters. With the introduction of L_2 regularization we have to adjust the derivatives in Equation (2.32) that we already derived for logistic regression. Considering that $\nabla_{\mathbf{w}}(\|\mathbf{w}\|^2) = 2\mathbf{w}$, we obtain:

$$\nabla_{\mathbf{w}} J_\lambda(\mathbf{w}, b) = \frac{1}{m} \sum_{i=0}^{m-1} \mathbf{x}_i (p_i - y_i) + 2\lambda \mathbf{w}. \tag{3.13}$$

Note that the derivative with respect to b does not change, since b does not occur in the regularization term. When λ is large enough, the regularization term speeds up the convergence of the algorithm, since it makes the objective function more convex. A possible implementation is the following:

```
1 def logreg_train(X, Y, lambda_, lr=1e-3, steps=1000):
2     m, n = X.shape
3     w = np.zeros(n)
```

```
    b = 0
    for step in range(steps):
        P = logreg_inference(X, w, b)
        grad_w = ((P - Y) @ X) / m + 2 * lambda_ * w
        grad_b = (P - Y).mean()
        w -= lr * grad_w
        b -= lr * grad_b
    return w, b
```

where the parameter `lambda_` represents the regularization coefficient (the trailing underscore is required because `lambda` is a Python keyword). The only differences with respect to the unregularized version are the parts highlighted in yellow.

Multinomial logistic regression can be regularized in the same way. The squared *Frobenius norm* is used as regularization term:

$$R(W) = \sum_{i=0}^{n-1} \sum_{j=0}^{k-1} W_{ij}^2 = \|W\|_F^2. \tag{3.14}$$

Again, the source code is very similar to the unregularized version:

```
def multinomial_logreg_train(X, Y, lambda_, lr=1e-3, steps=1000):
    m, n = X.shape
    k = Y.max() + 1  # number of classes
    W = np.zeros((n, k))
    b = np.zeros(k)
    # Build the one hot vectors H
    H = np.zeros((m, k))
    H[np.arange(m), Y] = 1
    for step in range(steps):
        P = multinomial_logreg_inference(X, W, b)
        grad_W = (X.T @ (P - H)) / m  + 2 * lambda_ * W
        grad_b = (P - H).mean(0)
        W -= lr * grad_W
        b -= lr * grad_b
    return W, b
```

3.4 Example: fruit classification

Industrial automation poses many scenarios were we can find classification problems. Imagine a factory with a conveyer belt moving two types of fruits, oranges and lemons, that must be divided in order to be processed. A binary classifier could be used to distinguish among the two species. Each fruit would be analyzed by sensors measuring some features. For this examples we will consider the size of the fruit and the brightness of its skin, which could be both obtained with some kind of optical sensor. Oranges are

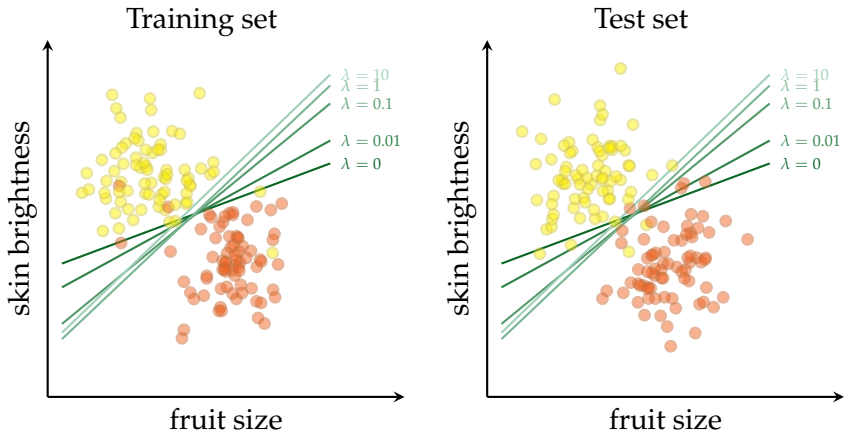


Figure 3.3: Citrus classification. Multiple linear classifiers are trained on the training set (left plot) by changing the regularization coefficient λ. The same classifier are shown on top of the test set (right plot).

expected to be larger than lemons, while lemons will often have a brighter skin.

First of all we need to collect data. By running the system for a while we collect 300 samples of fruits: 150 lemons and 150 oranges. We manually label each sample with the correct class: $y = 0$ for lemons and $y = 1$ for oranges. We randomly split the data into a training set and a test set, both containing 150 samples.

Then we will use the training set to build a classifier minimizing the objective function of the regularized logistic regression model. The effect of regularization would be easier to observe with more complex classifiers, with a larger number of features or with a highly non-linear decision boundary, but it may play an important role in this example as well.

What value of the coefficient λ should we use? The optimal value depends on the specific problem and on the training set and cannot be guessed in advance. Therefore, we try with different values. For each value we build a different classifier and we evaluate it on the test set. Figure 3.3 (on the left) shows the training set and a group of decision boundaries corresponding to the classifiers that we obtained with different values of λ. For $\lambda = 0$ we have the usual unregularized logistic regression. In that case the learning algorithm tries to fit as many samples as possible on the right side of the decision boundary. Unfortunately, in doing so it tends to focus too much on the samples close to the boundary causing some overfitting. The effect of introducing some regularization is the rotation of the decision boundary towards a more "natural" separation of the two classes. We can appreciate the effect on the test set on the right: the lowest test error is achieved for $\lambda = 0.1$. For larger

values of λ the classifier tends to ignore what happens in region between the two classes causing a loss of precision in the positioning of the decision boundary (underfitting).

3.5 Probabilistic interpretation

The intuition behind L_2 regularization is that by encouraging the use of multiple features it promotes robust classifiers. This effect has also a theoretical justification.

So far we used the maximum likelihood criterion to chose the parameters that best fit the data. That criterion consists in selecting the parameters that maximizes the probability of the observed data, but does not consider the parameters themselves as part of the random process: they are just fixed, but unknown. Every value for the parameter is considered as equally acceptable, provided that it agrees with the training set. However, by looking only at the training set this approach risks to overfit the data.

An alternative approach consists in considering the parameters as part of the underlying random process: imagine that the parameters $\boldsymbol{\theta}$ are randomly drawn from some distribution, and then each element in the training set is independently drawn according to them. Within this scenario, we may like to select the parameters that are the most probable given the data we observe. This criterion is called *maximum a posteriori* and consists in maximizing the following conditional distribution:

$$\max_{\boldsymbol{\theta}} p(\boldsymbol{\theta}|(x_0, y_0), \ldots, (x_{m-1}, y_{m-1})), \tag{3.15}$$

where the distribution is conditioned on the pairs in the training set. To simplify the problem we can use the Bayes' rule as follows:

$$p(\boldsymbol{\theta}|(x_0, y_0), \ldots, (x_{m-1}, y_{m-1})) = \frac{p((x_0, y_0), \ldots, (x_{m-1}, y_{m-1})|\boldsymbol{\theta})p(\boldsymbol{\theta})}{p((x_0, y_0), \ldots, (x_{m-1}, y_{m-1}))}. \tag{3.16}$$

The term $p(\boldsymbol{\theta})$ is the *prior probability distribution*, or simply the *prior*, and represents some kind of information we have about the possible values of the parameters before taking the data into account. The term in the denominator does not depend on $\boldsymbol{\theta}$ and therefore can be ignored. Moreover, we can make use of the hypothesis that the elements in the training set are drawn i.i.d. to decompose the first term in the numerator:

$$\max_{\boldsymbol{\theta}} \left(\prod_{i=0}^{m-1} p(x_i, y_i|\boldsymbol{\theta}) \right) p(\boldsymbol{\theta}). \tag{3.17}$$

Like we did for the maximum likelihood criterion, we can turn it to a minimization problem by taking the negative logarithm:

$$\min_{\boldsymbol{\theta}} \left(\sum_{i=0}^{m-1} -\log p(x_i, y_i|\boldsymbol{\theta}) \right) - \log p(\boldsymbol{\theta}). \tag{3.18}$$

The first term is just the negative log likelihood $-\log \mathcal{L}$. The other is the logarithm of the prior. A common choice is to assume that the parameters are normally distributed as follows:

$$p(\boldsymbol{\theta}) = \frac{1}{\sqrt{(2\pi)^d \sigma^{2d}}} e^{-\frac{1}{2\sigma^2}\|\boldsymbol{\theta}\|^2}, \tag{3.19}$$

where σ^2 is the variance of each one of the d parameters $\theta_0, \theta_1, \ldots, \theta_{d-1}$ that form $\boldsymbol{\theta}$. Note that the distribution is centered around zero. By applying the properties of logarithms we can compute the negative logarithm of the distribution:

$$-\log p(\boldsymbol{\theta}) = \frac{d}{2}\log(2\pi\sigma^2) + \frac{1}{2\sigma^2}\|\boldsymbol{\theta}\|^2, \tag{3.20}$$

where the first term is independent on $\boldsymbol{\theta}$, so it can be omitted in the optimization problem. The second term can be plugged in Equation (3.18) that becomes:

$$\min_{\boldsymbol{\theta}} \left(\sum_{i=0}^{m-1} -\log p(x_i, y_i|\boldsymbol{\theta}) \right) + \frac{1}{2\sigma^2}\|\boldsymbol{\theta}\|^2, \tag{3.21}$$

and this is exactly the maximum likelihood criterion with the addition of L_2 regularization, where the regularization coefficient depends on the variance of the prior $\lambda = \frac{1}{2\sigma^2}$.

Summing up, L_2 regularization corresponds to the application of the maximum a posteriori criterion with a Gaussian prior. This means that we are assuming, without looking at the data, that "small" parameters are more likely than "large" parameters. Note that the error term increases with the number of training samples, therefore the regularization term is more relevant when the training set is small.

For linear classifiers it is common to regularize only the vector of weights, for which there is an intuitive explanation, and not the bias. This makes very little difference in practice.

3.6 L_1 regularization

Different hypotheses on the prior distribution of the parameters lead to different kinds of regularization. A common alternative to the Gaussian prior,

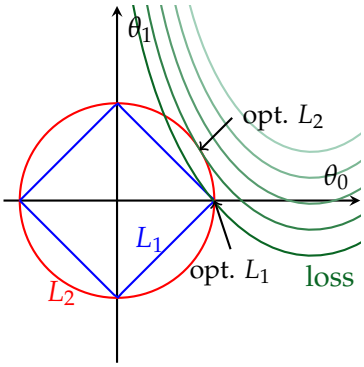


Figure 3.4: Example of level curves corresponding to the loss function and the regularization terms. The optimal solutions corresponds to points where the curves are mutually tangent. For L_1 regularization the vertices are more likely to be optimal than the other points.

is the Laplacian prior:

$$p(\boldsymbol{\theta}) = \left(\frac{\sqrt{2}}{\sigma}\right)^d e^{-\frac{\|\boldsymbol{\theta}\|_1}{\sqrt{2}\sigma}}, \tag{3.22}$$

which, when included in the maximum a posteriori criterion, results in the L_1 regularization. In this case the regularization term is:

$$R(\boldsymbol{\theta}) = |\theta_0| + |\theta_1| + \cdots + |\theta_{d-1}| = \|\boldsymbol{\theta}\|_1, \tag{3.23}$$

where $\|\cdot\|_1$ denotes the L_1 norm.

L_1 regularization has the advantage of favoring *sparse* solutions, that is, solutions where many parameters are zero. This can simplify the interpretation of the the resulting model. To see why this happens we have to consider that for a regularized model the optimal solution is a trade-off between the loss function on the training set and the regularization term. In the point corresponding to the optimal solution the level curves of the two terms must be tangent. By looking at Figure 3.4 we see that the level curve of the regularization term is perfectly symmetric so that no point on it has more chances to be the optimum. For L_1 regularization, instead, the vertices of the level curve are more likely to be the points where the curves are mutually tangent (for the same reason why if you throw a ball to a surface it can touch with any point, while if you throw a cube it is more likely to hit with one of the vertices). These vertices lie on the axis where one of the parameters is zero. This effect is even more evident when there are more than two parameters.

From the implementation point of view the differences betwen L_2 and L_1 regularization are minimal:

```python
def logreg_l1_train(X, Y, lambda_, lr=1e-3, steps=1000):
    m, n = X.shape
    w = np.zeros(n)
    b = 0
    for step in range(steps):
        P = logreg_inference(X, w, b)
        grad_w = ((P - Y) @ X) / m + lambda_ * np.sign(w)
        grad_b = (P - Y).mean()
        w -= lr * grad_w
        b -= lr * grad_b
    return w, b
```

where `lambda_ * np.sign(w)` is the gradient of the regularization term. Actually the regularization term is not differentiable everywhere, but it is *subdifferentiable* which is enough to make gradient descent work well. For our purposes we can proceed by assuming the following:

$$\frac{\partial}{\partial \theta_i} \|\boldsymbol{\theta}\|_1 = \begin{cases} +1 & \text{if } \theta_i > 0, \\ 0 & \text{if } \theta_i = 0, \\ -1 & \text{if } \theta_i < 0, \end{cases} \tag{3.24}$$

which is the behavior of the `np.sign` function.

3.7 Some guidelines

When dealing with a machine learning problem theoretical insights can be very helpful. In addition to these there are also some very good advice that have been distilled over time by practitioners. First of all it is obvious that the availability of data is the most important factor. Many issues in machine learning can be solved just by gathering more data. Overfitting, for instance, is more likely to occur when there is a small number of training samples.

The data must be divided into a training and a test set. Usually is a bad idea to have more test than training data. Someone suggests to split the data in half. However, there is no point in having a very large test set. The test set should be large enough to obtain accurate evaluations, but not larger. The most important rule is that the test sample must be randomly drawn from the available data. Often a big mistake is to pick the test data with some special criterion. The consequence of this mistake is that training and test data end up having a different distribution so that the model trained for one distribution cannot be assessed on data coming from another distribution.

After the collection of the data set, features need to be selected and computed. There are many techniques for feature selection. For now it is enough to realize that increasing the number of features not necessarily produces a

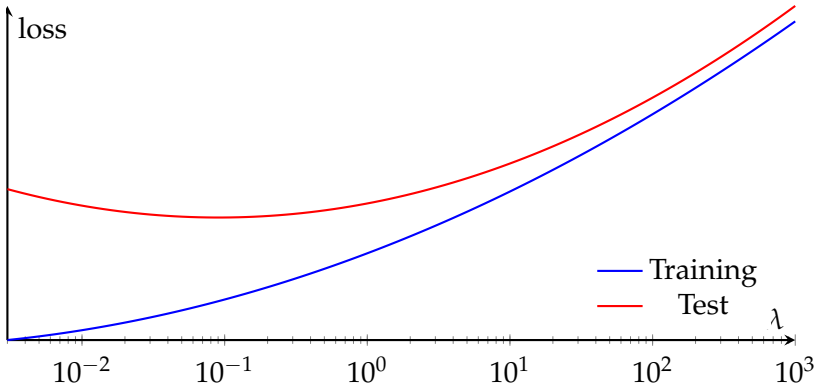


Figure 3.5: Typical behavior of training and test Loss varying the regularazion coefficient λ.

better classifier. In particular if the data is scarce we have to limit the number of features.

Overfitting can be detected by comparing the performance of a trained model on the training set and on the test set. If there is a large gap we have to decrease the complexity of the classifier by reducing the number of features, or by introducing a regularization term. L_2 regularization is the most common choice, unless we have a strong preference for sparse solutions. In that case we should prefer L_1 regularization.

If we use regularization we have to find a good value for the regularization coefficient λ. It is a common practice to try values having different orders of magnitude. For instance, many researchers uses the following schema:

$$\lambda \in \{\dots, 0.001, 0.003, 0.01, 0.03, 0.1, 0.3, 1, 3, 10, 30, 100, 300, \dots\}, \qquad (3.25)$$

where the range of values is selected so that the smallest have the same performance of the unregularized model ($\lambda = 0$), and the largest is big enough to make the loss term irrelevant. A typical behavior is that depicted in Figure 3.5.

It may happens that both training and test performance are too low, even if the classifier generalizes well. In this case the classifier is underfitting the data. To address conditions we should increase the complexity of the classification model by selecting more features or by adopting a more powerful model, such as those that we will see later on.

A useful tool to detect overfitting and underfitting are the "learning curves", obtained by plotting training and test loss (or the accuracies, or other performance measures) during the training iterations. Figure 3.6 shows an examples characterized by an evident overfitting. Note how there is a large gap between training and test performance.

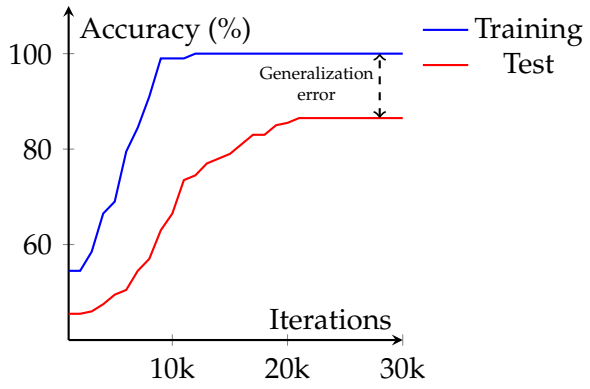


Figure 3.6: Example of learning curves.

Early stopping

While the training accuracy tends to increase monotonically during training, test accuracy may start to decrease after a while. This happens because it may take time for a model to overfit the training set. An alternative to explicit regularization is to use the *early stopping* technique, which consists in stopping the training algorithm before it converges, trying to freeze the model when it still generalizes well. Early stopping is very easy to implement (it is enough to reduce the number of training steps), but it may be tricky to identify the righ moment to stop. Looking at the test accuracy is not a valid solution because the test data should not be used to drive the training process.

Bayes error

When the test accuracy matches the training accuracy there is no overfitting. However, it might be the case that the model is underfitting and that better performance can be achieved by increasing the complexity of the model. How could we know that?

The ideal target is given by the Bayes error, which by definition is the lowest achievable. However, in most cases it is unknown. Sometimes we know that the Bayes error is zero (for instance, for problems where there is an algorithm able to compute the right label for each sample). In problems where the target label is assigned by humans, we can use the performance of a human classifier as an estimate of the Bayes error. This is the case of many complex problems, such as image recognition, document classification etc. However, even in these complex domains models capable of achieving *super-human* performance are becoming a commonplace.

3.8 Summary

This lecture focused on the concept of generalization which is the ability of a machine learning model to make correct predictions for new data.

- We identified the need for an independent test set to evaluate the trained models;
- we discussed the phenomenon of overfitting, a conditions in which a classifier fits too strictly the particular samples in the training set failing, at the same time, to model the test data. This usually happens when the model is too complex with respect to the size of the training set;
- we also defined the opposite condition called underfitting, in which the model is too simple and cannot fit well both training and test data.
- we briefly made a distinction between models with finite and infinite complexity;
- we presented the nearest neighbor classifier as an example of classification model of infinite complexity;
- we introduced the concept of regularization as a technique to improve generalization;
- we discussed two kind of regularization: L_2 and L_1 regularization and we justified them by introducing the maximum a posteriori criterion;
- we defined the regularized logistic regression model and we illustrated how it can be implemented in practice.

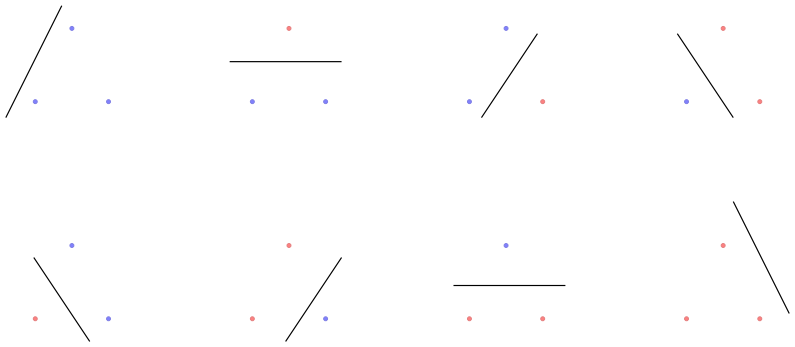
The VC dimension (optional)

The complexity of a model can be roughly measured by the number of its parameters. However, this measure is not very accurate and does not always work (for instance, because we can artificially increase the number of parameters without affecting its real complexity).

Better measures of complexity exist, and they allow to derive theoretical results of great relevance. One of the most important complexity measure is the VC dimension, called as such because it was defined by Vladimir Vapnik and Alexey Chervonenkis.

The VC dimension measures the degree of complexity of a set H of binary classifiers. Typically H corresponds to a parametric model, but it is not required.

The dimension is based on the fact that a rich set of classifier is able to represent many different ways of labeling a set of samples. Given a set of m samples $x_0, x_1, \ldots, x_{m-1}$ there are 2^m ways in which we can label them with the classes 0 and 1. A set of classifier H that is able to reproduce all the 2^m different labelings is said to *shatter* that set of samples. The figure here below shows, for instance, that the set of linear classifiers is able to shatter a set of 3 samples in the bidimensional plane.



The VC dimension of a set H is the size m of the largest set of samples that it can shatter. If H can shatter arbitrarily large sets of samples, then its VC dimension is ∞.

For instance, it is easy to show that the VC dimension of the set of linear classifiers in a n-dimensional space is $n+1$. The VC dimension of the nearest neighbor classifier is, instead, ∞.

The VC dimension is related to the generalization capability of a model. A model with infinite VC dimension can indefinitely continue to record the training labels, no matter the size of the training set. A model with finite VC dimension, instead, at some point needs to start to find an efficient way to correlate the feature vectors with the training labels because it has a finite capacity to adapt to the data.

The VC dimension can be used to bound the distance between the real and the empirical risk obtained through Empirical Risk Minimization. If VC_{dim} is the VC dimension of set H, and if we choose the classifier with the lowest empirical risk R_{emp} on the training data, for all $0 < \delta \leq 1$, with probability at least $1 - \delta$, the following holds true:

$$R \leq R_{\text{emp}} + \sqrt{\frac{VC_{\text{dim}}(\log(2m/VC_{\text{dim}}) + 1) - \log(\delta/4)}{m}},$$

where R is the real unknown risk, and m is the size of the training set. In practice, the generalization error of the model (the difference between the risk and the empirical risk) increases with the ratio VC_{dim}/m. This means that: (i) the larger the VC_{dim} dimension, the more likely is the model to overfit

the training data; (ii) the risk of overfitting can be mitigated by increasing the size of the training set; (iii) models with infinite VC dimension may continue to overfit independently on the size of the training set.

Lecture 4

Linear classifiers and Support Vector Machines

Despite their simplicity, linear classifiers can be very powerful. In fact, they tend to be faster, easier to train and less prone to overfitting than complex non-linear models. Logistic regression is a typical example of a linear classification model, but many others exist. Linear classification models assume that the classes can be separated (at least approximately) by simple planar decision boundaries.

The decision boundary has equation $\mathbf{w}^\intercal \mathbf{x} + b = 0$, where $\mathbf{w}$ and b are the parameters of the model. Given a sample $\mathbf{x}$ the decision $\hat{y}$ taken by the linear classifier depends on the sign of the score (or logit) $z = \mathbf{w}^\intercal \mathbf{x} + b$:

$$\hat{y} = \begin{cases} 1 & \text{if } \mathbf{w}^\intercal \mathbf{x} + b \geq 0, \\ 0 & \text{if } \mathbf{w}^\intercal \mathbf{x} + b < 0, \end{cases} \tag{4.1}$$

where the points on the boundary ($z = 0$) could be actually classified as either of class 0 or 1.

4.1 Linearly separable classes

The easiest condition for a linear classification model is when the classes can be exactly separated by a hyperplane. The training set

$$\{(\mathbf{x}_0, y_0), (\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_{m-1}, y_{m-1})\}\,,\ \mathbf{x}_i \in \mathbb{R}^n, y_i \in \{0, 1\}, \tag{4.2}$$

is *linearly separable* if there exist $\mathbf{w}^\star \in \mathbb{R}^n$ and $b^\star \in \mathbb{R}$ such that

$$\mathbf{w}^{\star\intercal} \mathbf{x}_i + b^\star > 0 \text{ if } y_i = 1, \tag{4.3}$$

$$\mathbf{w}^{\star\intercal} \mathbf{x}_i + b^\star < 0 \text{ if } y_i = 0,\ \forall i \in \{0, \ldots, m-1\}. \tag{4.4}$$

Note that if there is one separating hyperplane, then there are infinitely many.

When classes are linearly separable, there is a simple algorithm that can be used to find a separating hyperplane. The algorithm is called *perceptron* and it works as follows:

1. initialize $\mathbf{w} \leftarrow \mathbf{0}$ and $b \leftarrow 0$;
2. for each training pair $\mathbf{x}_i, y_i$:
   a) if $\mathbf{w}^\intercal \mathbf{x}_i + b > 0$ then set $\hat{y}_i \leftarrow 1$, else set $\hat{y}_i \leftarrow 0$;
   b) update the weights as $\mathbf{w} \leftarrow \mathbf{w} + (y_i - \hat{y}_i)\mathbf{x}_i$,
   c) and update the bias as $b \leftarrow b + (y_i - \hat{y}_i)$;
3. repeat step 2 until all training samples are correctly classified.

Similarly to gradient descent for logistic regression, the perceptron algorithm predicts the labels $\hat{y}_i$ of the training samples, and when it finds that the prediction differs from the true label y_i it updates the parameters $\mathbf{w}$ and b. Note that the parameters do not change when samples are correctly classified ($y_i - \hat{y}_i = 0$).

The algorithm can be implemented in just a few lines of Python:

```
def perceptron_train(X, Y, steps=10000):
    w = np.zeros(X.shape[1])
    b = 0
    for step in range(steps):
        errors = 0
        for i in range(X.shape[0]):
            d = (1 if X[i, :] @ w + b > 0 else 0)
            w += (Y[i] - d) * X[i, :].T
            b += (Y[i] - d)
            errors += np.abs(Y[i] - d)
        if errors == 0:
            break
    return w, b
```

```
def perceptron_inference(X, w, b):
    logits = X @ w + b
    labels = (logits > 0).astype(int)
    return labels
```

If the training set is linearly separable the perceptron algorithm terminates in a finite number of steps (see the box below).

Convergence of the perceptron (optional)

The proof of convergence of the perceptron algorithm is quite simple. First of all, let's simplify the notation with a simple trick. We augment the samples by appending a single 1 as last component: $\bar{\mathbf{x}}_i = \begin{pmatrix}\mathbf{x}_i & 1\end{pmatrix}$. Similarly, we combine $\mathbf{w}$ and b into a single vector of parameters: $\boldsymbol{\theta} = \begin{pmatrix}\mathbf{w} & b\end{pmatrix}$ so that we have:

$$\mathbf{w}^\mathsf{T}\mathbf{x} + b = \boldsymbol{\theta}^\mathsf{T}\bar{\mathbf{x}}. \tag{4.5}$$

By the separability hypothesis we know that there exist $\boldsymbol{\theta}^\star = \begin{pmatrix}\mathbf{w}^\star & b^\star\end{pmatrix}$ satisfying the conditions (4.3) and (4.4). In fact we can rewrite them as follows:

$$\boldsymbol{\theta}^{\star\mathsf{T}}\bar{\mathbf{x}}_i \geq \delta \quad \text{if } y_i = 1, \tag{4.6}$$

$$\boldsymbol{\theta}^{\star\mathsf{T}}\bar{\mathbf{x}}_i \leq -\delta \text{ if } y_i = 0, \quad \forall i \in \{0, \ldots, m-1\}, \tag{4.7}$$

where, beside the change in notation, we replaced the inequality > 0 with $\geq \delta$ and < 0 with $\leq -\delta$. This can be done because we have a finite number of training samples, and it is easy to show that $\delta = \min_i |\boldsymbol{\theta}^{\star\mathsf{T}}\bar{\mathbf{x}}_i|$ satisfies all the inequalities.

The perceptron algorithm starts with $\boldsymbol{\theta}_0 = \mathbf{0}$ and changes it every time it encounters a prediction error. In fact, with a minor abuse of the notation, we can ignore correct predictions and consider only the sequence of misclassified samples:

$$\bar{\mathbf{x}}_1, \bar{\mathbf{x}}_2, \ldots, \bar{\mathbf{x}}_E, \ldots \tag{4.8}$$

After each error the parameters are updated, forming the sequence:

$$\boldsymbol{\theta}_0, \boldsymbol{\theta}_1, \ldots, \boldsymbol{\theta}_E, \ldots \tag{4.9}$$

where $\boldsymbol{\theta}_E$ is the vector of parameters obtained after E errors. Following the algorithm we have

$$\boldsymbol{\theta}_E = \boldsymbol{\theta}_{E-1} + (y_E - \hat{y}_E)\bar{\mathbf{x}}_E, \tag{4.10}$$

where $\hat{y}_E$ is computed with $\boldsymbol{\theta}_{E-1}$. From this we can derive the following inequality:

$$\begin{aligned} \|\boldsymbol{\theta}_E\|^2 &= \|\boldsymbol{\theta}_{E-1} + (y_E - \hat{y}_E)\bar{\mathbf{x}}_E\|^2 \\ &= \|\boldsymbol{\theta}_{E-1}\|^2 + 2(y_E - \hat{y}_E)\boldsymbol{\theta}_{E-1}^\mathsf{T}\bar{\mathbf{x}}_E + \|\bar{\mathbf{x}}_E\|^2 \\ &\leq \|\boldsymbol{\theta}_{E-1}\|^2 + \|\bar{\mathbf{x}}_E\|^2, \end{aligned} \tag{4.11}$$

where the last inequality depends on the fact that, by hypothesis, $\bar{\mathbf{x}}_E$ is misclassified by $\boldsymbol{\theta}_{E-1}$. In fact there are only two possibilities:

- $y_E = 1$ and $\boldsymbol{\theta}_{E-1}^\mathsf{T}\bar{\mathbf{x}}_E < 0$. Therefore $\hat{y}_E = 0$;
- or $y_E = 0$ and $\boldsymbol{\theta}_{E-1}^\mathsf{T}\bar{\mathbf{x}}_E > 0$. Therefore $\hat{y}_E = 1$;

in both cases the term $2(y_E - \hat{y}_E)\boldsymbol{\theta}_{E-1}^{\mathsf{T}}\bar{\mathbf{x}}_E$ is negative. The inequality can be iterated backward in the sequence of parameters, obtaining the following:

$$\|\boldsymbol{\theta}_E\|^2 \leq \|\bar{\mathbf{x}}_1\|^2 + \|\bar{\mathbf{x}}_2\|^2 + \cdots + \|\bar{\mathbf{x}}_E\|^2 \leq ER^2, \tag{4.12}$$

where R is the maximum over all the norms of the samples $R = \max_E \|\bar{\mathbf{x}}_E\|$.

Another inequality involving $\boldsymbol{\theta}_E$ can be derived by taking into account its relationship with $\boldsymbol{\theta}^\star$:

$$\begin{aligned}\|\boldsymbol{\theta}_E\|^2\|\boldsymbol{\theta}^\star\|^2 \geq (\boldsymbol{\theta}^{\star\mathsf{T}}\boldsymbol{\theta}_E)^2 &= \left(\boldsymbol{\theta}^{\star\mathsf{T}}\sum_{i=1}^{E}(y_i - \hat{y}_i)\bar{\mathbf{x}}_i\right)^2 = \left(\sum_{i=1}^{E}(y_i - \hat{y}_i)\boldsymbol{\theta}^{\star\mathsf{T}}\bar{\mathbf{x}}_i\right)^2 \\ &\geq \left(\sum_{i=1}^{E}\delta\right)^2 = E^2\delta^2,\end{aligned} \tag{4.13}$$

where the first is the Cauchy-Schwarz inequality, and the last follows from the hypothesis that

- either $y_i = 1$ and $\hat{y}_i = 0$ (because we are considering errors). Therefore $\boldsymbol{\theta}^{\star\mathsf{T}}\bar{\mathbf{x}}_i \geq \delta$ because of the hypothesis on $\boldsymbol{\theta}^\star$;
- or $y_i = 0$ and $\hat{y}_i = 1$. Therefore $\boldsymbol{\theta}^{\star\mathsf{T}}\bar{\mathbf{x}}_i \leq -\delta$;

and in both cases $(y_i - \hat{y}_i)\boldsymbol{\theta}^{\star\mathsf{T}}\bar{\mathbf{x}}_i \geq \delta$.

Combining the inequalities (4.12) and (4.13) we obtain:

$$E \leq \frac{R^2\|\boldsymbol{\theta}^\star\|^2}{\delta^2}, \tag{4.14}$$

which represents an upper bound on the number of the training errors made by the perceptron algorithm before a separating hyperplane is found.

4.2 Maximum-margin classifiers

When the training set is linearly separable the perceptron algorithm is guaranteed to find a separating hyperplane. However, under these conditions, the gap between the classes can accomodate for infinitely many other separating hyperplanes. Figure 4.1 shows an example of a linearly separable training set with a group of separating hyperplanes: which of those should we prefer? Intuitively we may want to avoid those hyperplanes that pass too close to the training samples. In fact we expect that in the proximity of each sample are likely to occur other samples of the same class. A good strategy

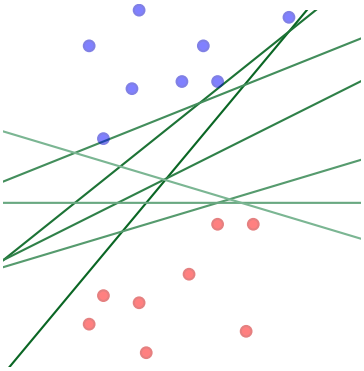


Figure 4.1: A set of points that is linearly separable. There are infinitely many separating hyperplanes. Some of them are shown as examples.

consists in choosing the hyperplane that is as far as possible from all the training samples. This hyperplane would be the one maximizing the separation between the two classes.

Consider two parallel separating hyperplanes. The region between them is called *margin*. The *maximum-margin hyperplane* is the hyperplane that lies halfway between the pair of parallel hyperplanes of maximal margin. Figure 4.1 illustrates this concept.

The margin of a separating hyperplane can be computed through basic notions of geometry. Suppose to have a training set that is linearly separable and supose that $\mathbf{w}$ and b defines a separating hyperplane:

$$\mathbf{w}^\intercal \mathbf{x}_i + b > 0 \text{ if } y_i = 1, \tag{4.15}$$

$$\mathbf{w}^\intercal \mathbf{x}_i + b < 0 \text{ if } y_i = 0, \quad \forall i \in \{0, \ldots, m-1\}. \tag{4.16}$$

Without loss of generality we may assume that $\min_i |\mathbf{w}^\intercal \mathbf{x}_i + b| = 1$. If that is not the case we can scale both $\mathbf{w}$ and b by a suitable positive factor to make it happen without affecting the hyperplane. We can then rewrite the separability constraints as follows:

$$\mathbf{w}^\intercal \mathbf{x}_i + b \geq +1 \text{ if } y_i = 1, \tag{4.17}$$

$$\mathbf{w}^\intercal \mathbf{x}_i + b \leq -1 \text{ if } y_i = 0, \quad \forall i \in \{0, \ldots, m-1\}. \tag{4.18}$$

where the equality holds true for at least one training sample, which intuitively would be the closest to the hyperplane. Let $\mathbf{x}_q$ be the closest sample, and let's suppose that its label is $y_q = 1$. Therefore we have:

$$\mathbf{w}^\intercal \mathbf{x}_q + b = 1. \tag{4.19}$$

Let d be the distance between $\mathbf{x}_q$ and its projection $\mathbf{x}'_q$ onto the separating hyperplane. We have then:

$$\mathbf{x}'_q = \mathbf{x}_q - d\frac{\mathbf{w}}{\|\mathbf{w}\|}. \tag{4.20}$$

Moreover, since $\mathbf{x}'_q$ lies on the hyperplane we also have:

$$\mathbf{w}^\intercal \mathbf{x}'_q + b = 0. \tag{4.21}$$

Finally, by substituting Equation (4.20) in (4.21) we obtain:

$$\mathbf{w}^\intercal \mathbf{x}_q - d\frac{\mathbf{w}^\intercal \mathbf{w}}{\|\mathbf{w}\|} + b = 0, \tag{4.22}$$

which, considering that $\mathbf{w}^\intercal \mathbf{w} = \|\mathbf{w}\|^2$, gives us the a formula for d

$$d = \frac{\mathbf{w}^\intercal \mathbf{x}_q + b}{\|\mathbf{w}\|} = \frac{1}{\|\mathbf{w}\|}. \tag{4.23}$$

Finally, the margin is just twice as much $2d = \frac{2}{\|\mathbf{w}\|}$. It is straightforward to show that the same result is obtained if $\mathbf{x}_q$ has label $y_q = 0$. Therefore, the maximum-margin separating hyperplane is represented by the parameters $\mathbf{w}$ and b that are found by maximizing the margin $\frac{2}{\|\mathbf{w}\|}$ under the constraints (4.17) and (4.18).

4.3 Support Vector Machines

Support Vector Machines (SVM) are a popular family of classifiers based on the maximization of the margin. In practice, instead of maximizing the margin, it is easier to solve the following optimization problem:

$$\begin{aligned} &\min_{\mathbf{w},b} \frac{\|\mathbf{w}\|^2}{2} \\ &\text{subject to } \begin{cases} \mathbf{w}^\intercal \mathbf{x}_i + b \geq +1 \text{ if } y_i = 1, \\ \mathbf{w}^\intercal \mathbf{x}_i + b \leq -1 \text{ if } y_i = 0, \quad \forall i \in \{0, \ldots, m-1\}. \end{cases} \end{aligned} \tag{4.24}$$

In fact, this is equivalent to the maximization of $\frac{2}{\|\mathbf{w}\|}$. This is a quadratic optimization problem subject to a set of linear constraints. For this kind of problems there are very efficient algorithms that are guaranteed to find the exact solution.

However, this approach works only under the hypothesis that the training set is linearly separable, which is a very restrictive condition. In fact, even a small and simple set can result impossible to separate with a hyperplane. A classic example is represented by the "exclusive or" (XOR) problem

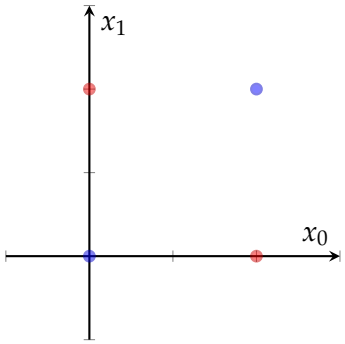


Figure 4.2: A training set for the "exclusive or" problem. The set is not linearly separable since it is not possible to divide the samples of the two classes with a straight line.

depicted in Figure 4.2. The XOR problem is a binary classification problem in which the classes are assigned to replicate the output of a XOR logic gate. Each sample has two components assuming the values 0 and 1: if they are different the sample has label 1, otherwise it has label 0. It is easy to see that no linear classifier can separate the two samples $(0,1), (1,0)$ of class 1 from the two samples $(0,0), (1,1)$ of class 0.

Soft margin

To deal with training sets that are not linearly separable we have to be tolerant to violations of the constraints (4.17) and (4.18). To do so, instead of rejecting the solutions not satisfying all the constraints we penalize them by introducing an extra cost in the objective function.

More in detail, consider the score $z_i = \mathbf{w}^\intercal \mathbf{x}_i + b$:

- if $y_i = 1$ and $z_i \geq 1$ the i-th sample is on the right side of the separating hyperplane outside the margin region and therefore no penalization should be introduced;
- for the same reason, no penalization is needed for the case in which $y_i = 0$ and $z_i \leq -1$;
- if, instead, $y_i = 1$ and $z_i < 1$ then the sample is linearly penalized by the amount $1 - z_i$;
- and when $y_i = 0$ and $z_i > -1$ the penalization is $1 + z_i$.

This penalization scheme corresponds to the use of the *hinge loss* function, defined as

$$h(y, z) = \max\{1 - (2y - 1)z, 0\}. \tag{4.25}$$

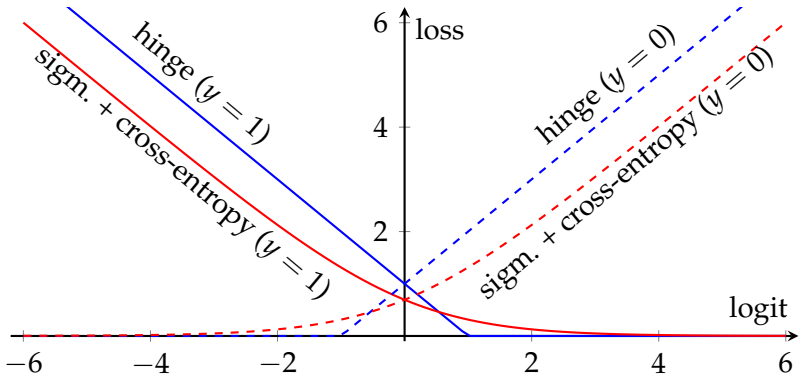


Figure 4.3: Hinge loss and sigmoid + cross entropy loss, given the logits $z = \mathbf{w}^\intercal \mathbf{x} + b$ for the two cases $y = 1$ and $y = 0$.

The hinge loss represents how bad it is to assign the score $z = \mathbf{w}^\intercal \mathbf{x} + b$ (which we can call logit) to a sample of class y. This loss linearly penalizes scores that are smaller than 1 when y is 1, and that are greater than -1 when y is 0. The $\max\{\cdot, 0\}$ part ensures that the loss is always non-negative.

The replacement of hard constraints with the hinge loss leads to the so called *soft-margin* SVM which corresponds to the solution of the following optimization problem:

$$\min_{\mathbf{w},b} J_\lambda(\mathbf{w}, b) = \left(\frac{1}{m} \sum_{i=0}^{m-1} h(y_i, \mathbf{w}^\intercal \mathbf{x}_i + b) \right) + \lambda \frac{\|\mathbf{w}\|^2}{2}, \tag{4.26}$$

The term $\left(\frac{1}{m} \sum_{i=0}^{m-1} h(y_i, \mathbf{w}^\intercal \mathbf{x}_i + b) \right)$ pushes the resulting classifier to label correctly as many samples as possible. The term $\frac{\|\mathbf{w}\|^2}{2}$ favors a large margin. The parameter λ determines the trade-off between the two terms.

The soft margin formulation allows for training samples to lie inside the margin, or even in the wrong side of the separating hyperplane. These samples are characterized by a loss that is strictly positive. The loss is exactly zero when the corresponding sample is correctly classified and is located outside the (soft) margin region.

This model is similar to regularized logistic regression. In fact both corresponds to the minimization of the combination of an average loss function with a L_2 regularization term, weighted by a regularization coefficient λ. The only difference is that in SVM the hinge loss replaces the cross entropy loss.

Figure 4.3 compares the hinge loss with the composition of sigmoid and cross entropy used in logistic regression.

Support vectors

It is easy to show that the objective function $J_\lambda(\mathbf{w}, b)$ in problem (4.26) is convex. This guarantees that there is a single global minimum. In particular, the optimal weights $\mathbf{w}$ can be written as a linear combination of the training samples:

$$\mathbf{w} = \sum_{i=0}^{m-1} \alpha_i \mathbf{x}_i, \tag{4.27}$$

and the geometry of the problem makes it so most of the coefficients α_i are zero. In fact, the separating hyperplane depends only on the points that determine the margin. More precisely, we can divide the training samples on the basis of their logits $z_i = \mathbf{w}^\intercal \mathbf{x}_i + b$:

- those for which $y_i = 1$ and $z_i = 1$, as well as those for which $y_i = 0$ and $z_i = -1$, are samples that lie exactly on the boundary of the margin region;

- those for which $y_i = 1$ and $z_i < 1$, or $y_i = 0$ and $z_i > -1$, are samples that are inside the margin region or that are on the wrong side of the hyperplane;

- those for which $y_i = 1$ and $z_i > 1$, or $y_i = 0$ and $z_i < -1$, are samples that are on the correct side of the hyperplane and are outside the margin region.

Most samples usually fall in the third category, and the corresponding coefficients α_i are guaranteed to be zero. Samples in the first and in the second category are called *Support Vectors* and are the only ones directly determining the separating hyperplane. In practice, moving the support vectors changes the optimal separating hyperplane, while perturbing the other samples leaves the solution unchanged. Figure 4.4 shows an example of SVM with highlighted the support vectors.

Optimization

Training a SVM, with soft margin or not, consists in solving a quadratic optimization problem. There are optimization algorithms which deal with this kind of problems by quickly finding the exact solution. However, we can also proceed by applying the gradient descent algorithm. To do so, we need an expression for the gradient of the objective function of (4.26) with respect to

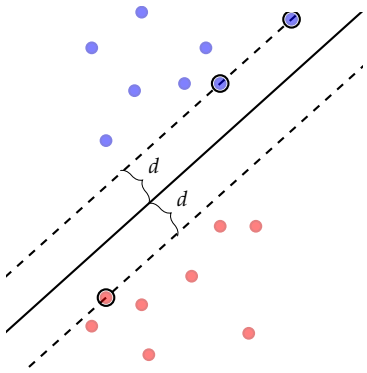


Figure 4.4: SVM Classifier obtained by maximizing the margin $2d = \frac{2}{\|\mathbf{w}\|}$. The solid line represents the decision boundary of equation $\mathbf{w}^\intercal \mathbf{x} + b = 0$ while the dashed lines represent the hyperplanes of equation $\mathbf{w}^\intercal \mathbf{x} + b = \pm 1$ which delimit the margin. There are three support vectors, they are the three circled samples that lie exactly on the dashed lines.

the parameters $\mathbf{w}$ and b:

$$\nabla_{\mathbf{w}} J_\lambda(\mathbf{w}, b) = \left(\frac{1}{m} \sum_{i=0}^{m-1} h'(y_i, \mathbf{w}^\intercal \mathbf{x}_i + b) \mathbf{x}_i \right) + \lambda \mathbf{w}, \tag{4.28}$$

$$\frac{\partial}{\partial b} J_\lambda(\mathbf{w}, b) = \frac{1}{m} \sum_{i=0}^{m-1} h'(y_i, \mathbf{w}^\intercal \mathbf{x}_i + b), \tag{4.29}$$

where the derivative h' of the hinge loss with respect to the logit z is given by:

$$h'(y, z) = \begin{cases} -1 & \text{if } y = 1 \text{ and } z < 1, \\ 1 & \text{if } y = 0 \text{ and } z > -1, \\ 0 & \text{otherwise,} \end{cases} \tag{4.30}$$

which can be easily inferred by looking at its plot. More precisely, the hinge loss is not differentiable, but it is *subdifferentiable*, which luckily is enough to make gradient descent work. Its implementation for training linear SVMs is the following:

```
def svm_train(X, Y, lambda_, lr=1e-3, steps=1000):
    m, n = X.shape
    w = np.zeros(n)
    b = 0
    for step in range(steps):
        z = X @ w + b
        hinge_diff = -Y * (z < 1) + (1 - Y) * (z > -1)
        grad_w = (hinge_diff @ X) / m + lambda_ * w
        grad_b = hinge_diff.mean()
```

```
        w -= lr * grad_w
        b -= lr * grad_b
    return w, b
```

```
def svm_inference(X, w, b):
    z = X @ w + b
    labels = (z > 0).astype(int)
    return labels
```

4.4 Multi-class SVMs

Binary classification models such as SVMs can be adapted to work with more than two classes. One way to achieve this is to break the multi-class problem into multiple binary classification problems. The two most common strategies to do so are called *one versus rest* and the *one versus one*.

One versus rest

The one vs. rest (or one vs. all) strategy consists in training one binary classifier for each class. The sample of that class are considered positive and those of all the other classes are considered negative. This strategy requires that the underlying binary classification model is able to output a confidence score instead of just a class label. The combined classifier predicts as output the class for which the highest confidence score has been obtained.

The following code implements the one vs. rest strategy for linear SVMs. It is straightforward to adapt it to other models.

```
def one_vs_rest_train(X, Y, lambda_, lr=1e-3, steps=1000):
    k = Y.max() + 1
    W = np.zeros((X.shape[1], k))
    b = np.zeros(k)
    for c in range(k):
        Ybin = (Y == c)
        wbin, bbin = svm_train(X, Ybin, lambda_, lr, steps)
        W[:, c] = wbin
        b[c] = bbin
    return W, b

def one_vs_rest_inference(X, W, b):
    scores = X @ W + b.T
    labels = scores.argmax(1)
    return labels
```

One versus one

In the one vs. one strategy a binary classifier is trained for each pair of classes. For instance, if there are four classes $(0, 1, 2, 3)$ there will be six classifiers (0 vs. 1, 0 vs. 2, 0 vs. 3, 1 vs. 2, 1 vs. 3 and 2 vs. 3). Note that each binary classification problem includes only a fraction of the original training set (only the samples belonging to one of the two classes).

At inference time, the combined classifier submits the input samples to the binary classifiers. Each binary classifier votes for one of the two classes it has been trained on, and the class that collects the highest numbers of votes is taken as prediction (ties are broken arbitrarily).

The following code implements the one vs. one strategy.

```
def one_vs_one_train(X, Y, lambda_, lr=1e-3, steps=1000):
    k = Y.max() + 1
    m, n = X.shape
    W = np.zeros((n, k * (k - 1) // 2))
    b = np.zeros(k * (k - 1) // 2)
    j = 0
    # For each pair of classes...
    for pos in range(k):
        for neg in range(pos + 1, k):
            # Build a training subset
            subset = (np.logical_or(Y == pos, Y ==
                neg)).nonzero()[0]
            Xbin = X[subset, :]
            Ybin = (Y[subset] == pos)
            # Train the classifier
            Wbin, bbin = svm_train(Xbin, Ybin, lambda_, lr,
                steps)
            W[:, j] = Wbin
            b[j] = bbin
            j += 1
    return W, b

def one_vs_one_inference(X, W, b):
    # 1) recover the number of classes from s = 1 + 2 + ... + k
    m = X.shape[0]
    s = b.size
    k = int(1 + np.sqrt(1 + 8 * s)) // 2
    votes = np.zeros((m, k))
    scores = X @ W + b.T
    bin_labels = (scores > 0)
    # For each pair of classes...
    j = 0
    for pos in range(k):
        for neg in range(pos + 1, k):
            votes[:, pos] += bin_labels[:, j]
            votes[:, neg] += (1 - bin_labels[:, j])
            j += 1
```

```
36      labels = np.argmax(votes, 1)
37      return labels
```

Note that the one vs. one strategy requires the training of $\frac{1}{2}k(k-1)$ classifiers, which can be a very large number even for a moderate number of classes. However, these classfiers are trained for problems that are easier to solve than those in the one vs. rest strategy. For instance, it is possible that each pair of classes is linearly separable, even when it is not possible to separate each class from all the others simultaneously. Concerning training times, depending on the classification problem it is possible that is faster to solve $\frac{1}{2}k(k-1)$ small classification problems, than k large ones.

Briefly, comparing the two combination strategies we can observe that:

- one vs. rest requires that the binary classifiers provide a confidence measure for their prediction, and that is possible to compare these measures to determine the most likely class. One vs. one only needs classifiers that provide a binary prediction.
- One vs. rest trains one classfier for each class, while the number of classifier in one vs. one is quadratic in the number of classes. It is therefore not realistic to use one vs. one for a large number of classes.
- The binary classification problems in one vs. rest are more complicated to solve than those in the one vs. one case. Sometimes this fact makes it so one vs. one produces more accurate predictions.
- The training of each binary classifier in one vs. rest requires more time with respect to that needed to train one classifer in the one vs. one strategy. In the first case there are more training samples in each subproblem and more training iterations are probably needed.

Figure 4.5 compares the two strategies on the iris classification problem. The accuracy on the training set is slightly higher for the one vs. one strategy (82%, compared to the 79.33% obtained by one vs. rest). Probably this depends on the fact that it is easier to separate two classes rather than separating one class from the other two. Note how for one vs. rest the decision boundary passes through the intersections between the separating hyperplanes (this is not a coincidence! Why?) For one vs. one, instead, the decision boundary and the hyperplanes are partially overlaped (again, this is not a coincidence!)

SVM vs. regularized logistic regression

Support Vector Machines and regularized logistic regression are two models for binary classification differing only in the loss function used to quantify

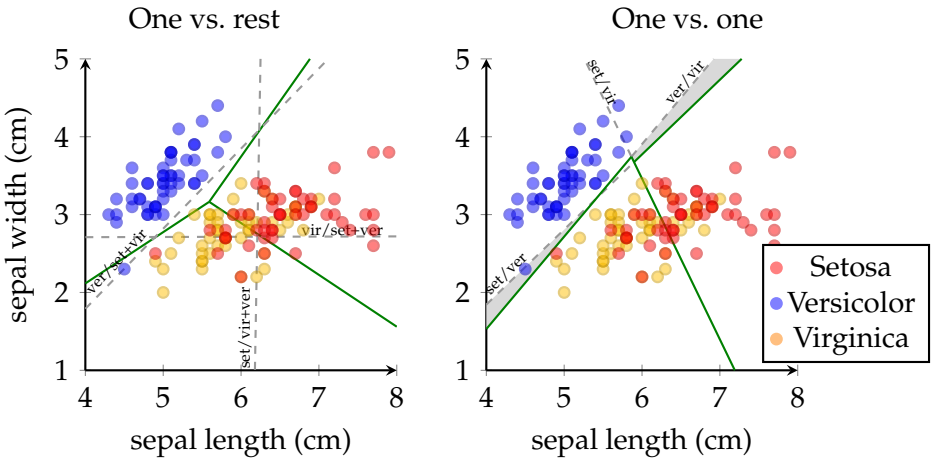


Figure 4.5: Multi-class SVM applied to iris classification by using two different combining strategies: one vs. rest (left) and one vs. one (right). The decision boundary separating the three classes is drawn with a solid line, while the boundaries of the underlying binary classifiers are dashed. For one vs. one there are two regions (filled in gray) where each class receive exactly one vote.

the goodness of prediction on training samples. So, which one should we prefer? There is not a definitive answer. Both models can be very effective and the choice on which one to use depends on several factors. Here are some of them:

- Support Vector Machines can be easier to train thanks to the simple structure of the optimization problem. This is especially true if a specialized quadratic optimization algorithm is used.
- Logistic regression has clear probabilistic interpretation and can be used to estimate the probability that a given sample belong to one of the classes. SVM just predicts the most likely class.
- SVM is based on maximizing the margin, which usually favors generalization.
- SVM depends only on a subset of the training set (the support vectors) and are therefore robust to perturbations of the training samples. This fact can be used to further accelerate the training procedure.
- Logistic regression can be more easily extended to the multi-class case.
- SVM can be extended to non-linear classifiers, as we will see in the next lecture.

4.5 Summary

In this lecture we discussed two models: the perceptron and the Support Vector Machine. Like the logistic regression, they fall in the category of binary linear classifiers. More precisely:

- we described the learning algorithm for the perceptron;
- we introduced the concept of margin for linear classifiers and we used it to define "hard-margin" Support Vector Machines;
- we extended SVMs to the "soft-margin" formulation to address problems that are not linearly separable;
- we adapted the gradient descent algorithm to make it usable to train Support Vector Machines.
- We observed that multiple binary classifiers can be combined together to form a multi-class classifier.
- We defined the one vs. rest strategy in which one classifier is trained for each class.
- We defined the one vs. one strategy in which one classifier is trained for each pair of classes.

Lecture 5

Non-linear Support Vector Machines

Both logistic regression and soft-margin SVM can deal with classification problems in which the classes are not linearly separable. However, there are cases in which a linear classifier cannot reasonably work. Figure 5.1 shows some examples. In these cases it is not possible to separate the training samples of the two classes with a straight line, not even approximately.

Non-linear classifiers are characterized by decision boundaries with complex shapes that can accomodate for training sets distributed in intricate ways. There are many machine learning models that result in non-linear classification. Support Vector Machines, for instance, can be easily extended to the non-linear case.

5.1 The kernel trick

A conceptually simple way to deal with classes that cannot be linearly separated is to use a *feature map* to project the features in a space where they are easier to separate. If the map is not linear, then a linear decision boundary

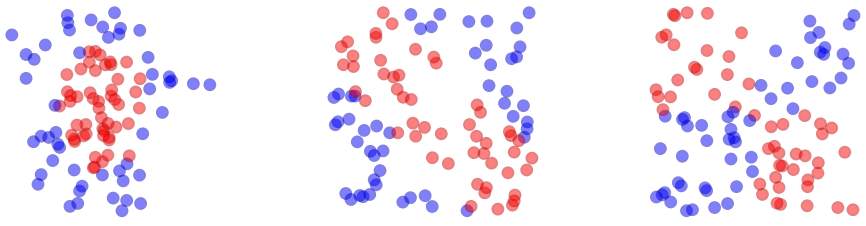


Figure 5.1: Three examples of classification problems that cannot be dealt with satisfactory by using linear classifiers.

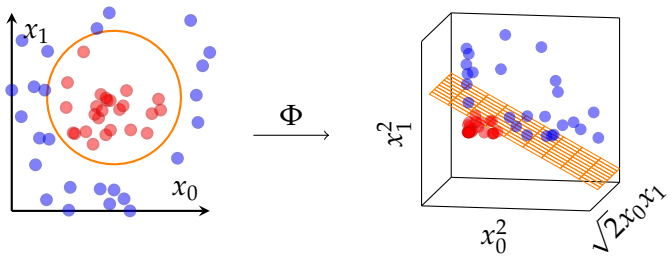


Figure 5.2: A training set that is not linerly separable and that becomes separable after the application of the projection mapping Φ. A linear decision boundary in the destination space $\mathbb{R}^3$ can be backprojected into a non-linear decision boundary in the original $\mathbb{R}^2$ space.

in the destination space would be non-linear in the original space. Consider, for instance, the map $\Phi : \mathbb{R}^2 \to \mathbb{R}^3$ defined as follows:

$$\Phi \begin{pmatrix} x_0 \\ x_1 \end{pmatrix} = \begin{pmatrix} x_0^2 \\ \sqrt{2} \cdot x_0 x_1 \\ x_1^2 \end{pmatrix}, \tag{5.1}$$

then a plane in the destination $\mathbb{R}^3$ space of equation:

$$\mathbf{w}^\intercal \Phi(\mathbf{x}) + b = w_0 x_0^2 + \sqrt{2} w_1 x_0 x_1 + w_2 x_1^2 + b = 0 \tag{5.2}$$

would correspond to an elliptical (or hyperbolical) curve in the original $\mathbb{R}^2$ space. This situation is illustrated in Figure 5.2.

By choosing the right mapping Φ it is possible to make linearly separable training sets that were not in the original space. An expert of a specific classification problem may explicitly design a mapping that emphasizes the distinctive features for the classes. However, this can be a very challenging task, in particular when the original space is already composed of a large number of features.

Alternatively, we may choose a mapping into a very high-dimensional space, a space that is big enough to make linear separation very likely. Unfortunately, a large space poses some practical issues in terms of computational resources and of generalization (a large space implies models with many parameters).

The *kernel trick* is a technique that allows to have the best of both worlds: a large space where samples can be easily separated, and a limited use of computational resources. The kernel trick consists in using a *kernel function* to manipulate vectors mapped into a high-dimensional space by processing the corresponding vectors in the original low-dimensional space.

More precisely, a *kernel* is a binary function $k : \mathcal{L} \times \mathcal{L} \to \mathbb{R}$ for which it exists a mapping $\Phi : \mathcal{L} \to \mathcal{H}$ from a low-dimensional space $\mathcal{L}$ into a high-dimensional vector space $\mathcal{H}$, such that:

$$k(\mathbf{x}, \mathbf{x}') = \Phi(\mathbf{x})^\intercal \Phi(\mathbf{x}'), \tag{5.3}$$

for every pair $\mathbf{x}, \mathbf{x}' \in \mathcal{L}$. In short, a kernel function is a generalization of an inner product. Note that not all binary functions are kernels. Kernels need to follow a given set of mathematical rules (not given here).

A widely used kernel function is the *polynomial kernel*, defined as:

$$k(\mathbf{x}, \mathbf{x}') = \left(\mathbf{x}^\intercal \mathbf{x}' + 1\right)^d, \tag{5.4}$$

where d is an integer parameter representing the degree of the polynomial kernel. Applying the polynomial kernel is equivalent to take the inner product in the space of all the monomials, up to degree d, formed by the combinations of input components. For istance, the mapping Φ corresponding to the polynomial kernel of degree $d = 2$ is:

$$\Phi(\mathbf{x}) = \Phi\begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{pmatrix} = \begin{pmatrix} x_0^2 \\ x_1^2 \\ \vdots \\ x_{n-1}^2 \\ \sqrt{2}x_0x_1 \\ \sqrt{2}x_0x_2 \\ \vdots \\ \sqrt{2}x_{n-2}x_{n-1} \\ \sqrt{2}x_0 \\ \sqrt{2}x_1 \\ \vdots \\ \sqrt{2}x_{n-1} \\ 1 \end{pmatrix}, \tag{5.5}$$

and it is easy to show that $\Phi(\mathbf{x})^\intercal \Phi(\mathbf{x}') = (\mathbf{x}^\intercal \mathbf{x}' + 1)^2$. In this case Φ maps n-dimensional points into a space with $\frac{(n+1)(n+2)}{2}$ dimensions, therefore the explicit computation of Φ is intractable even for moderate values of n.

Another popular one is the *Gaussian RBF kernel* which is defined as

$$k(\mathbf{x}, \mathbf{x}') = e^{-\gamma \|\mathbf{x} - \mathbf{x}'\|^2}. \tag{5.6}$$

In this case the function Φ implicitly maps vectors onto the surface of an infinite-dimensional hypersphere. Therefore, the mapping cannot even be

computed explicitly. The real-valued parameter γ determines how fast the values of the kernel function drops to zero as the distances between the two arguments increases.

In short, the kernel trick can be used to make a machine learning model or algorithm non-linear, as follows:

1. the model/algorithm is reformulated so that the original samples appears only in inner products;

2. the inner products are replaced by a kernel function.

5.2 Kernel SVM

To define a non-linear SVM we start by introducing a mapping Φ into the optimization problem for linear SVM:

$$\min_{\mathbf{w},b} J_\lambda(\mathbf{w}, b) = \left(\frac{1}{m} \sum_{i=0}^{m-1} h(y_i, \mathbf{w}^\intercal \Phi(\mathbf{x}_i) + b) \right) + \lambda \frac{\|\mathbf{w}\|^2}{2}, \tag{5.7}$$

The optimal $\mathbf{w}$ and b can then be used to classify a new sample $\mathbf{x}$ by predicting label 1 if $\mathbf{w}^\intercal \Phi(\mathbf{x}) + b > 0$ and label 0 otherwise.

It can be shown that the optimal $\mathbf{w}$ is guaranteed to be a linear combination of the projected training samples $\Phi(\mathbf{x}_i)$:

$$\mathbf{w} = \sum_{i=0}^{m-1} \alpha_i \Phi(\mathbf{x}_i). \tag{5.8}$$

In fact, instead of looking directly for the best $\mathbf{w}$ we can search for the best coefficients $\boldsymbol{\alpha} = (\alpha_0, \ldots, \alpha_{m-1})$. By substituting (5.8) into (5.7) we obtain:

$$\begin{aligned} \min_{\boldsymbol{\alpha},b} J_\lambda(\boldsymbol{\alpha}, b) = \left(\frac{1}{m} \sum_{i=0}^{m-1} h(y_i, b + \sum_{j=0}^{m-1} \alpha_j \Phi(\mathbf{x}_j)^\intercal \Phi(\mathbf{x}_i)) \right) + \\ \lambda \frac{1}{2} \sum_{i=0}^{m-1} \sum_{j=0}^{m-1} \alpha_i \alpha_j \Phi(\mathbf{x}_i)^\intercal \Phi(\mathbf{x}_j), \end{aligned} \tag{5.9}$$

so that we can apply the kernel trick by replacing all the inner products $\Phi(\cdot)^\intercal \Phi(\cdot)$ with the corresponding kernel function:

$$\begin{aligned} \min_{\boldsymbol{\alpha},b} J_\lambda(\boldsymbol{\alpha}, b) = \left(\frac{1}{m} \sum_{i=0}^{m-1} h(y_i, b + \sum_{j=0}^{m-1} \alpha_j k(\mathbf{x}_j, \mathbf{x}_i)) \right) + \\ \lambda \frac{1}{2} \sum_{i=0}^{m-1} \sum_{j=0}^{m-1} \alpha_i \alpha_j k(\mathbf{x}_i, \mathbf{x}_j). \end{aligned} \tag{5.10}$$

Once we found the optimal coefficients $\boldsymbol{\alpha}$, in general we cannot use them to compute $\mathbf{w}$. In fact it is possible that $\mathbf{w}$ belongs to a very high-dimensional space (even inifinite-dimensional). Luckly we don't need it to classify new samples, but we can proceed again by applying the kernel trick. Given $\mathbf{x}$ we will predict label 1 if $b + \sum_{i=0}^{m-1} \alpha_i k(\mathbf{x}_i, \mathbf{x}) > 0$ and label 0 otherwise.

5.3 Training a kernel SVM

The optimization problem (5.10) is quadratic in the unknown variables $\boldsymbol{\alpha}$ and b. The objective function is convex, therefore there is a single optimal solution and it can be found with specific quadratic optimization algorithms. However, even if not as efficient as a specialized algorithm, gradient descent can be used as well. It is enough to plug into the general algorithm the (sub)gradient of the objective function with respect to the coefficients α_l and the bias b:

$$\begin{aligned}
\frac{\partial}{\partial \alpha_l} J_\lambda(\boldsymbol{\alpha}, b) &= \left(\frac{1}{m} \sum_{i=0}^{m-1} h'(y_i, b + \sum_{j=0}^{m-1} \alpha_j k(\mathbf{x}_j, \mathbf{x}_i)) k(\mathbf{x}_i, \mathbf{x}_l) \right) + \lambda \sum_{i=0}^{m-1} \alpha_i k(\mathbf{x}_i, \mathbf{x}_l), \\
\frac{\partial}{\partial b} J_\lambda(\boldsymbol{\alpha}, b) &= \left(\frac{1}{m} \sum_{i=0}^{m-1} h'(y_i, b + \sum_{j=0}^{m-1} \alpha_j k(\mathbf{x}_j, \mathbf{x}_i)) \right),
\end{aligned} \tag{5.11}$$

where the derivative h' of the hinge loss is the same already encountered in the linear case:

$$h'(y, s) = \begin{cases} -1 & \text{if } y = 1 \text{ and } s < 1, \\ 1 & \text{if } y = 0 \text{ and } s > -1, \\ 0 & \text{otherwise.} \end{cases} \tag{5.12}$$

One important difference between the linear and the non-linear formulations of SVM is the number of parameters. In the linear formulation we need to learn a vector $\mathbf{w}$ of n components, one for each component of the feature vectors. For kernel SVM we need to learn a vector $\boldsymbol{\alpha}$ of m coefficients, one for each training sample. Often the size of the training set m is larger than the number of features n. However, this is not always the case. Since the final solution depends only on a limited number of support vectors, we expect that the optimal $\boldsymbol{\alpha}$ is sparse (i.e. that many components are null). This fact can be exploited to improve the speed of the learning algorithm. Note that during training, if there is enough memory, it is possible to precompute a matrix with the values of the kernel function applied to the training samples $K_{ij} = k(\mathbf{x}_i, \mathbf{x}_j)$ (but beware that this matrix would have m^2 elements!).

An (inefficient) implementation of kernel svm is the following:

```
def ksvm_train(X, Y, kfun, kparam, lambda_, lr=1e-3, steps=1000):
    K = kernel(X, X, kfun, kparam)
    m, n = X.shape
    alpha = np.zeros(m)
    b = 0
    for step in range(steps):
        ka = K @ alpha
        z = ka + b
        hinge_diff = -Y * (z < 1) + (1 - Y) * (z > -1)
        grad_alpha = (hinge_diff @ K) / m + lambda_ * ka
        grad_b = hinge_diff.mean()
        alpha -= lr * grad_alpha
        b -= lr * grad_b
    return alpha, b
```

```
def ksvm_inference(X, Xtrain, alpha, b, kfun, kparam):
    K = kernel(X, Xtrain, kfun, kparam)
    z = K @ alpha + b
    labels = (z > 0).astype(int)
    return labels
```

```
def kernel(X1, X2, kfun, kparam):
    if kfun == "polynomial":
        return (X1 @ X2.T + 1) ** kparam
    elif kfun == "rbf":
        qx1 = (X1 ** 2).sum(1, keepdims=True)
        qx2 = (X2 ** 2).sum(1, keepdims=True)
        cross = 2 * X1 @ X2.T
        return np.exp(-kparam * (qx1 - cross + qx2.T))
    else:
        raise ValueError("Unknown kernel ('%s')" % kfun)
```

5.4 Kernels and overfitting

Venturing in the realm of non-linear classifiers exposes to the risk of overfitting. Regularization helps only to a certain degree and may not be enough if we choose to work with a classification model that is too flexible. For kernel SVM this means that we have to pay attention to the choice of the kernel and of its parameter(s). In fact the model may result of infinite capacity, because the number of parameters is proportional to the size of the training set.

For the polynomial kernel overfitting it is likely to happen if the degree of the kernel is too high. For the Gaussian RBF kernel, generalization depends on the parameter γ. In fact, plugging the kernel into the decision function we

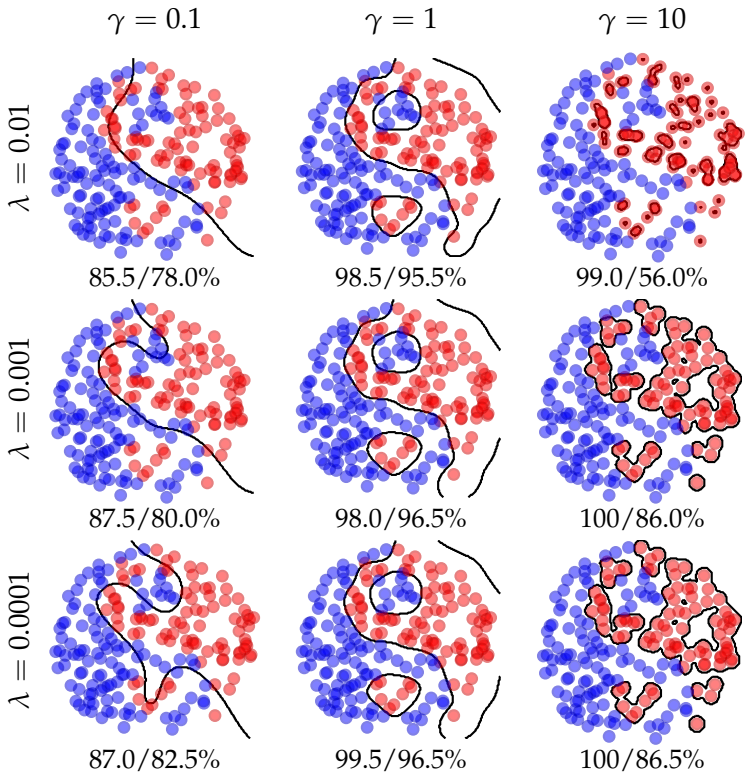


Figure 5.3: Decision boundary of a SVM with Gaussian RBF kernel, varying the regularization coefficient λ and by the kernel parameter γ. Below each plot are reported the corresponding training and test accuracy (computed on other data from the same distribution).

obtain the following:

$$\hat{y} = \begin{cases} 1 & \text{if } b + \sum_{i=0}^{m-1} \alpha_i e^{-\gamma \|\mathbf{x} - \mathbf{x}_i\|^2} > 0, \\ 0 & \text{otherwise,} \end{cases} \tag{5.13}$$

which means that the class predicted by the model depends on a sum of Gaussians centered on the training samples. The parameter γ determines the extent of the influence of single training samples. A small γ would cause the overlapping of many Gaussians. A large γ limits the influence of each training sample to a small neighborhood possibly causing overfitting since the prediction for a new sample would depend on a very small number of elements in the training set. This behavior can be observed in Figure 5.3 which shows for a toy problem how a SVM with Gaussian kernel in influenced by the regularization coefficient λ and by the kernel parameter γ. Note how there is underfitting for γ to small and overfitting for γ too large. The regu-

larization coefficient can only partially correct these issues.

5.5 Summary

In this lecture we looked for the first time to a class of non-linear classifier. These classifier are an extension of the linear SVM.

- We observed how the kernel trick can be used to turn some linear algorithms into non-linear;
- the kernel trick turns linear SVM into the non-linear SVM model;
- we discussed how non-linear SVM can be trained by gradient descent and we shown a prototypical implementation;
- we noticed how kernels are related to generalization and overfitting.

Lecture 6

Model selection

In machine learning, the choice of the model, and of its parameters is called *model selection*. Most models and algorithms include many parameters that need to be properly set to obtain good performance. For instance, non-linear SVMs include the regularization coefficient λ, the kernel function and, often, a parameter for the kernel. To these we may add all settings of the learning algorithm, such as the learning rate and the number of training steps. All these parameters are called *hyperparameters* to distinguish them from those that are learned during training.

It is not usually possible to determine in advance suitable values for the hyperparameters. More realistically, multiple values are tried and the best combination is selected. To do so, a measure of goodness is needed. The training accuracy would not work, since it does not generalize to new data. The accuracy on the test set cannot be used because then it would stop being an unbiased estimate of the quality of the model. The test accuracy, in fact, should be kept independent from any design choice. To see why using the test set for model selection is a bad idea, just imagine to use the outcome of a sequence of rolls to select among a group of dice the "most performing". Even though one of them may have obtained high numbers, there is no reason to expect the same performance for new rolls.

In short, it is a bad idea to use the same data to take decisions and to evaluate their consequences. A possible solution consists in using a separate set of data for model selection. This set is called *validation set*.

In most cases it is easy to form a validation set. It is enough to divide the available data in three subsets: the training set, the test set and the validation set. The division should be random, to ensure that the distributions of the samples in each set is the same. The right proportions for the three sets depends on the availability of data. Too many samples in the test and validation sets makes the training set unnecessarily small, reducing the effectiveness of training. On the other hand, a small validation set makes model selection un-

reliable. A small test set, instead, cannot guarantee an accurate final estimate of the performance of the trained model.

For data sets of limited size (thousands of samples) a popular choice is to divide them using a 60/20/20 proportions. For instance, a data set of 2000 samples could be divided into 1200 training samples, 400 validation samples and 400 test samples.

For large datasets (millions of samples) the training set should include most of the data and the other two sets should be just large enough to allow accurate estimates. For instance, a data set of 1 000 000 samples could be divided into 950 000 training samples, 25 000 validation samples and 25 000 test samples. More validation or test sample would just reduce the effectiveness of training.

The main steps of a machine learning procedure including model selection are:

1. choose some new values for the hyperparameters;
2. train a model by using the chosen hyperparameters;
3. evaluate the result on the validation set, keeping memory of the best hyperparameters;
4. repeat from step 1 until satisfied;
5. set the hyperparameters to the best values found;
6. evaluate the final choice on the test set.

The combinations of values to try for the hyperparameters can be determined with different strategies.

6.1 Grid search

One popular approach to model selection is represented by the *grid search* technique. It consists in

1. identifying a set of possible values for each hyperparameter;
2. train the model with each possible combination;
3. evaluate the trained models on the validation set;
4. take the combination with the highest validation accuracy.

Grid search is simple and effective, but it can be very time consuming. Usually you can afford to use it to select one or two hyperparameters (sometimes three). If the best combination of parameters lie on the boundary of the grid, then the grid must be enlarged to make sure that the range of parameters includes the optimum.

For some hyperparameter identifying a reasonable set of possible values can be tricky. Some exploratory tests may help in identifying the range and the number of values to try. In some cases the hyperparameter may assume values with different orders of magnitude, and the set of possible values should follow a geometric progression. For instance, it is a common practice for the regularization coefficient λ to try values like $\ldots$, 0.01, 0.03, 0.1, 0.3, 1, 3, 10, 30, $\ldots$

The results of the grid search should give some feeling about the sensitivity of the model and it should suggest which hyperparameters need to be carefully tuned. If processing time allows it, it can be a good idea to repeat the search with a finer grid around the combination found in the first round.

Example: page blocks classification

This example will show how grid search may be used in practice to select the parameters of a non-linear SVM with RBF kernel. The problem concerns the analysis of the content of scanned documents. More precisely, the pages of the documents are divided in blocks of homogeneous content, and our task is to learn to classify the blocks as text or graphics. We have a dataset of 2000 instances, each one representing a text block (label 0) or a graphic block (label 1). Each block is described by a vector of ten numerical features:

- the height;
- the width;
- the area (height $\times$ width);
- the aspect ratio (width / height);
- the number of black pixels;
- the number of black pixels after a smoothing algorithm;
- the percentage of black pixels;
- the percentage of black pixels after a smoothing algorithm;
- the number of white-black transitions;
- white-black transitions normalized with respect to the area.

More details about the problem and the dataset are available at the address `https://archive.ics.uci.edu/ml/datasets/Page+Blocks+Classification`.

First of all, the dataset is randomly divided into a training set of 1200, a validation set of 400 and a test set of 400 instances. As classification model we decide to use a non-linear SVM with a Gaussian RBF kernel. Therefore we need to set the following parameters:

- the learning rate;
- the number of gradient descent steps;
- the regularization coefficient λ;
- the kernel parameter γ.

We cannot reasonably use grid search to set all of them. However, the learning rate and the number of steps can be set in a conservative way (many steps with a low learning rate) at the cost of some extra computational effort.

After a few trials we set the learning rate to 0.01 and the training steps to 200 000. We also empirically constrained the other parameters to λ in the range $\lambda \in \{10^{-7}, 3 \times 10^{-7}, 10^{-6}, \ldots, 10^{-3}, 3 \times 10^{-3}\}$ and $\gamma \in \{10^{-6}, 3 \times 10^{-6}, 10^{-6}, \ldots, 10^{-2}, 3 \times 10^{-2}\}$.

Then we run a grid search to find suitable values for the combination of λ and γ. Figure 6.1 shows the accuracy obtained on the training set for each pair of values. As expected, for large values of γ the model adapts easily to the training set, whose instances are correctly classifier more than 99% of the times. Large values of λ, instead, make the model underfit the data. If we would use the training accuracy to select the parameters we would end-up with a large value for γ and a small value for λ, and this combination would correspond to an accuracy on the test set below 80%.

The accuracies on the validation set are reported in Figure 6.2. The best performance are obtained for intermediate values of γ and small values of λ. In case of similar accuracies it is a best practice to favor the combination of hyperparameters resulting in the stronger regularization. Taking this into account we would select the combination $\lambda = \gamma = 10^{-5}$. With this parameter we would obtain an accuracy on the test set of 92%. By choosing the hyperparameters directly on the test set we could obtain slightly better results (92.5%), but that would overestimate the real accuracy of the method and would be caused by random fluctuations in the data. (see Figure 6.3).

Random search

An altervative to grid search is the *random search*. It simply consists in trying a sequence of multiple random combinations of the hyperparameters. This has several advantages over grid search:

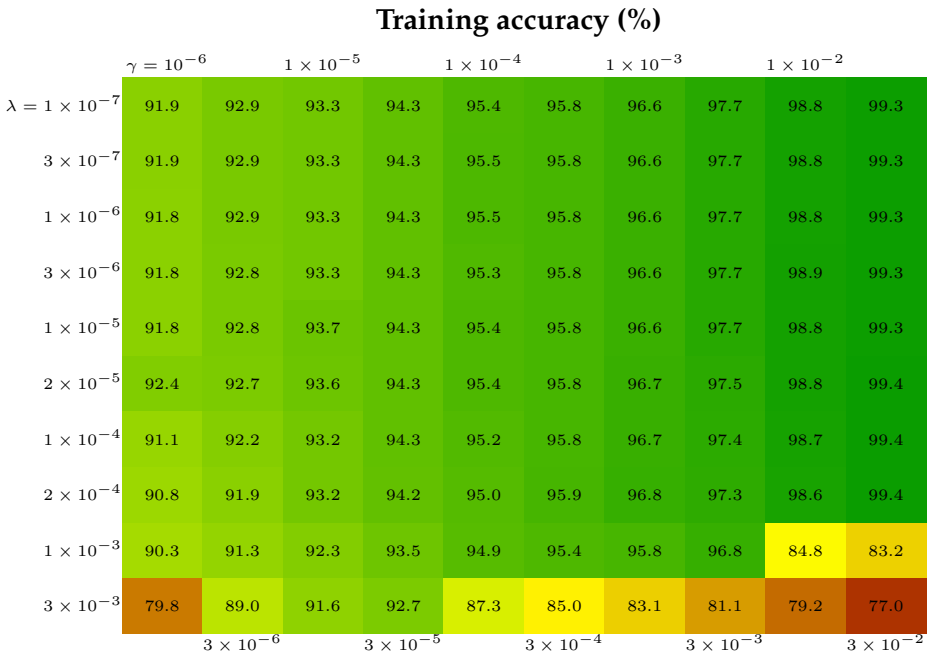


Figure 6.1: Grid search for the problem of page blocks classification. Accuracy on the training set.

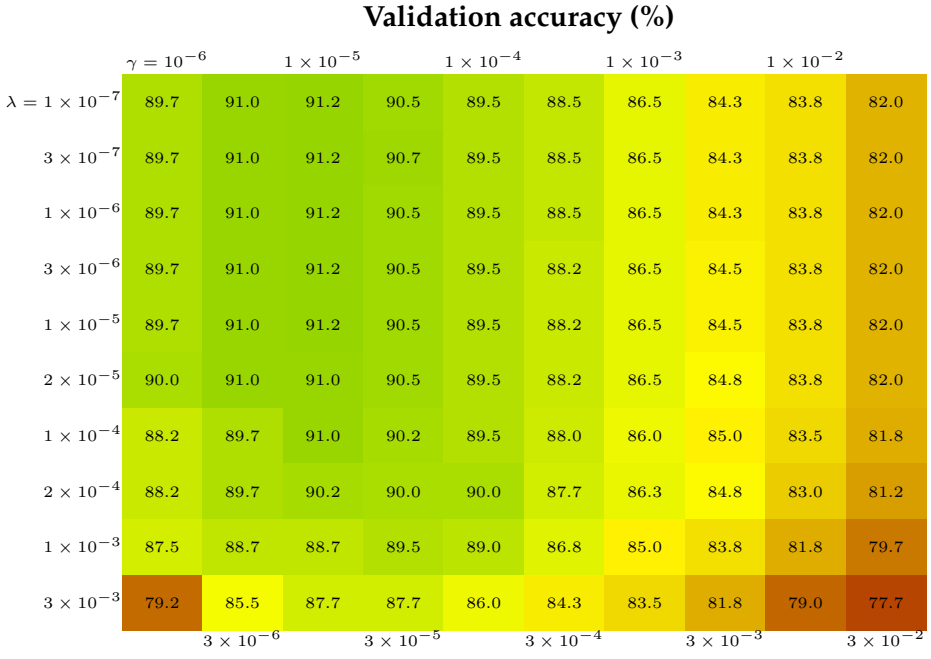


Figure 6.2: Grid search for the problem of page blocks classification. Accuracy on the validation set.

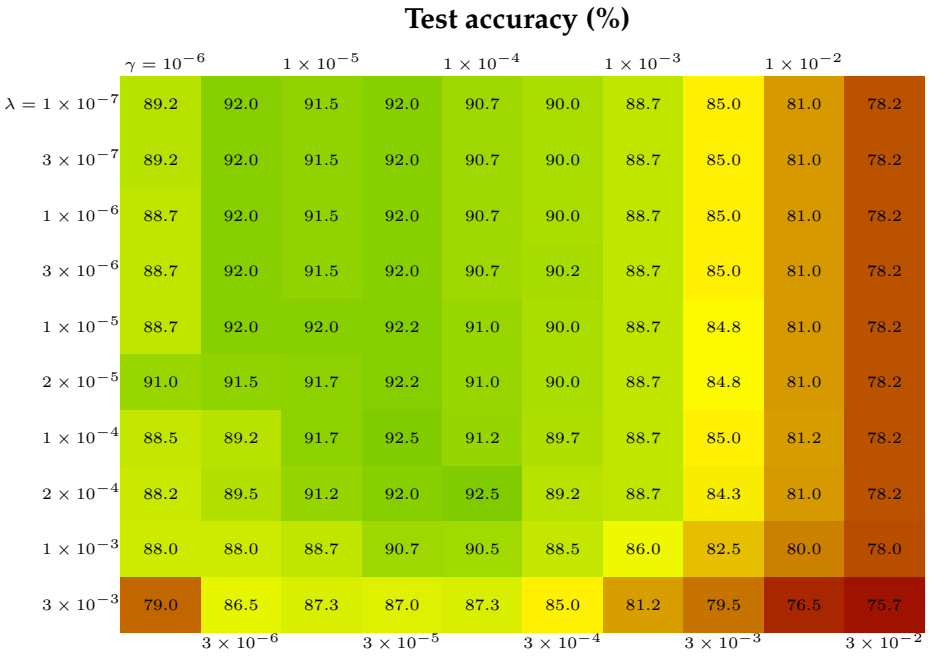


Figure 6.3: Grid search for the problem of page blocks classification. Accuracy on the test set.

- it makes optimal use of the available time, since the exact number of possible combinations is tried for given budget of time;
- it makes it possible to optimize any number of hyperparameters, while grid search is limited to two or three;
- in case there is one hyperparameter with low impact on the final performance, random search does not waste time by trying multiple times the same combination of all the others.

Of course, random search has the downside of being random and therefore less predictable than grid search.

6.2 *k*-fold cross-validation

When data is scarce, the formation of a validation set may subtract valuable samples that could be used to train a better model. The *k-fold cross-validation* technique allows to perform model selection without reserving precious samples for validation.

In *k*-fold cross-validation the training set is randomly split in k subsets (called *folds*) approximately of the same size. Then, k models are trained,

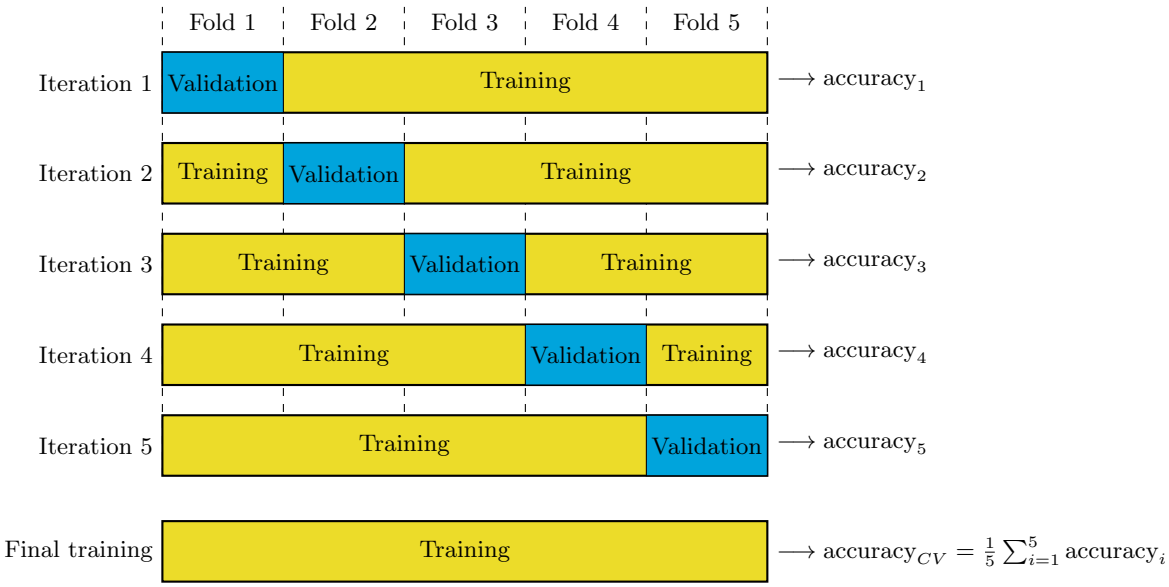


Figure 6.4: Graphical representation of five-fold cross-validation. The data set is randomly divided in five folds. Then five models are trained on four of the five folds and then evaluated on the remaining one. A final model can be trained on the whole data set, and its accuracy is estimated by the average of the accuracies obtained during the five cross-validation iterations.

each one using the data in $k-1$ folds. The fold that is left out is used to estimate the accuracy of the model. Each of the k estimates is unbiased, since it is obtained on data that is not used to train the corresponding model. Their average can be used an estimate of the accuracy of the model that would result from training the model on the whole training set. A schematic view of the whole cross-validation procedure is depicted in Figure 6.4

k-fold cross-validation can be implemented with just a few lines of Python. The code here below is a possible implementation (`train` and `inference` must be replaced by suitable functions).

```
def cross_validation(k, X, Y):
  m = X.shape[0]
  # Randomly assign a fold index to each sample.
  folds = np.arange(m) % k
  np.random.shuffle(folds)
  correct_predictions = 0
  # For each fold...
  for fold in range(k):
      Xtrain = X[folds = fold, :]Ytrain = Y[folds = fold]
      # Train a model
      params = train(Xtrain, Ytrain)
      # Evaluate the model on the left-out fold
```

```
13          Xval = X[folds == fold, :]
14          Yval = Y[folds == fold]
15          predictions = inference(Xval, params)
16          correct_predictions += (predictions == Yval).sum()
17      return correct_predictions / m
```

Using k-fold cross-validation for model selection can be very time consuming, since each evaluation requires k separate training iterations. Some learning algorithms allows to save some time by reusing some of the work done during the iterations. For instance, gradient descent can be initialized to start the optimization from the final state obtained from the previous iteration so that fewer training steps are actually needed.

Number of folds

How many folds should we used for k-fold cross validation? If k is too small the folds are relatively large. Therefore each of the k models is trained on a small subset of the available data and their accuracy will be significantly lower than that of the final model. In short, too few folds lead to a overly pessimistic estimate.

With a large number of folds the cross-validation estimate of the accuracy will be very precise. The main issue about using many folds is the training time. Common values for k are 5 and 10.

Some training algorithms allow to speed-up the training of the models by exploiting some of the work done in training previous models on other folds. For instance, in gradient descent parameters can be initialized with the optimal values previously found instead of starting from scratch, reducing at the same time the number of training steps.

Some algorithms are so fast in training on overlapping sets of data that it is possible to consider each training sample as a separate fold ($k = m$). This technique is called *leave one out*. An example of classification method for which the leave one out approach is very convenient is presented in the next section.

6.3 K-Nearest Neighbors

K-Nearest Neighbors (KNN) is an extension of the nearest neighbor classifier (see Lecture 3 about generalization). Like NN, the training procedure consists in the memorization of all the training samples together with their class labels. Unlike NN, given a new sample $\mathbf{x}$ we look at the k nearest training samples, not just the closest one. The most frequent label among the k neighbors is then chosen as the output of the classifier (ties are decided arbitrarily). By relying on more than just one sample, KNN tends to generalize

better than NN at the cost of some additional computation. However, both NN and KNN can be very expensive for large training sets!

The inference procedure can be implemented quite easily (here we make use of two helper functions, one to computed the matrix of all distances between test and training samples, and one to compute all the histograms of votes).

```
def knn_inference(X, Xtrain, Ytrain, k):
    classes = Ytrain.max() + 1
    D = _dist_matrix(X, Xtrain)
    neighs = np.argpartition(D, k, 1)[:, :k]
    counts = _bincount_rows(Ytrain[neighs], classes)
    labels = np.argmax(counts, 1)
    return labels
```

```
def _bincount_rows(X, values):
    """Compute one histogram for each row."""
    # np.bincount works only on 1D arrays.
    # This extends it to 2D arrays.
    m = X.shape[0]
    idx = X.astype(int) + values * np.arange(m)[:, np.newaxis]
    c = np.bincount(idx.ravel(), minlength=values * m)
    return c.reshape(-1, values)
```

```
def _dist_matrix(X1, X2):
    """Compute the matrix of all squared distances."""
    Q1 = (X1 ** 2).sum(1, keepdims=True)
    Q2 = (X2 ** 2).sum(1, keepdims=True)
    return Q1 - 2 * X1 @ X2.T + Q2.T
```

The performance of KNN depends on the parameter k. With $k = 1$ the method is just the simple nearest neighbor with its dubious generalization. For large values of k the method tends to underfit the data since the label it assigns depend also on training samples that are far away from the one it has to classify (see Figure 6.5 for a toy example). A suitable value for k can be selected with the help of a validation set, or with a folded cross validation.

Actually, the leave one out strategy is computationally cheap for KNN. This is due to the absence of a real training procedure, and to the fact that multiple values for k can be tried very quickly. A simple implementation computes all the distances between pairs of training samples. Then it sorts all the neighbors of all the samples, "leaving out" each sample from its own neighborhood. Finally, for each value of k it computes the most voted class for each sample by its k nearest neighbors, and compares it with the training label. The value of k corresponding to the highest accuracy is then selected.

```
def knn_select_k(X, Y, maxk=101):
    """Leave-one-out selection of the number of neighbors."""
```

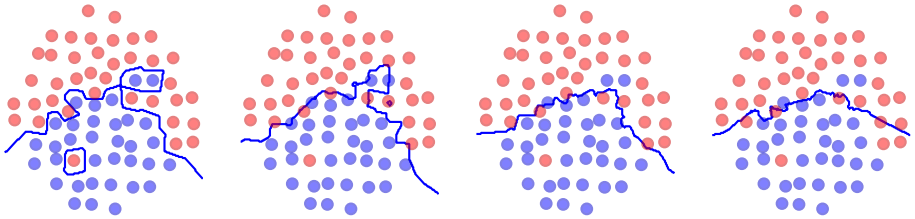


Figure 6.5: Decision boundary of four k-nearest neighbor classifiers applied to a binary classification problem. For $k = 1$ there is clearly some overfitting, while for $k = 21$ the classifier underfits the data.

```
    D = _dist_matrix(X, X)
    classes = Y.max() + 1
    np.fill_diagonal(D, np.inf)
    neighs = np.argsort(D, 1)
    best_k = 1
    best_acc = -1
    for k in range(1, maxk + 1):
        counts = _bincount_rows(Y[neighs[:, :k]], classes)
        labels = np.argmax(counts, 1)
        accuracy = (labels == Y).mean()
        if accuracy > best_acc:
            best_acc = accuracy
            best_k = k
    return best_k
```

For the classification problem in Figure 6.5 this algorithm would choose the value $k = 9$.

6.4 Performance measures

Accuracy is the most obvious performance measure for classification models. It is defined as the fraction of cases for which the model gives the right answer. However, accuracy is not always the best criterion to use to evaluate the goodness of a model.

One issue with accuracy is that it weights each error in the same way. For some applications different kind of errors may have wildly different consequences. For instance, in medical diagnosis a *false positive* (e.g. wrongly detecting the presence of a desease) has a different impact than a *false negative* (e.g. wrongly deciding for the absence of a desease).

A more precise analysis of the errors can be obtained from by building the *confusion matrix* which counts the occurrences of four different cases. The two classes are labeled as "positive" and "negative", where the meaning of the

labels depends on the particular application. The four cases take into account the actual class of the samples (i.e. their true class) and the class predicted by the model:

- a *true positive* (TP) is a case for which both the actual and the predicted classes are positive;
- a *true negative* (TN) is a case for which both the actual and the predicted classes are negative;
- a *false positive* (FP) is a case in which the model predicts the positive class while the actual class is negative;
- a *false negative* (FN) is a case in which the model predicts the negative class while the actual class is positive.

The confusion matrix is usually reported in a tabular form:

		Predicted class	
		Negative	Positive
Actual class	Negative	TN	FP
	Positive	FN	TP

Instead of the counts, sometimes it is preferred to report the *rates* obtained by normalizing each row of the matrix:

$$\begin{aligned} \text{(True Positive Rate) TPR} &= \frac{\text{TP}}{\text{TP}+\text{FN}}, \\ \text{(False Negative Rate) FNR} &= \frac{\text{FN}}{\text{FN}+\text{TP}}, \\ \text{(True Negative Rate) TNR} &= \frac{\text{TN}}{\text{TN}+\text{FP}}, \\ \text{(False Positive Rate) FPR} &= \frac{\text{FP}}{\text{FP}+\text{TN}}, \end{aligned} \tag{6.1}$$

Suppose, for instance, that we are building a system to diagnose a viral infection. Also suppose that we have a test set with 24 patients with the virus (positive cases) and 50 healthy patients (negative cases). If a classification model predicts the presence of the virus in 21 of the positive cases and in 9 of the negative cases, the confusion matrix would be the following:

		Predicted class	
		Negative	Positive
Actual class	Negative	41	9
	Positive	3	21

Or, using rates (in percentages):

		Predicted class	
		Negative	Positive
Actual class	Negative	82.0%	18.0%
	Positive	12.5%	87.5%

Multiclass problems

The confusion matrix can be also used when there are more than two classes. In that case the rows will refer to the actual class labels, while columns will represent the predictions made by the model. Inside the matrix we can report the number of occurrences, or the values normalized by dividing the sum of each row. The following, for instance, is the confusion matrix obtained by multinomial logistic regression on the Iris dataset described in previous lectures (see Figure 2.8):

		Predicted class		
		Versicolor	Virginica	Setosa
Actual class	Versicolor	98%	2%	0%
	Virginica	0%	74%	26%
	Setosa	0%	24%	76%

Correct predictions contributes to the elements on the diagonal of the matrix. Off-diagonal elements represent errors and their analysis reveal the behavior of the model, for instance by highlighting pairs of classes that are easily confused.

Unbalanced problems

Many performance measures can be expressed in terms of the four entries of the confusion matrix. For instance, in binary classification accuracy is defined as:

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{FP} + \text{TN} + \text{FN}}, \tag{6.2}$$

and its complement, the *error rate* as:

$$\text{Error Rate} = 1 - \text{Accuracy} = \frac{\text{FP} + \text{FN}}{\text{TP} + \text{FP} + \text{TN} + \text{FN}}. \tag{6.3}$$

An important case is that in which the two classes have a very different frequency. When this happens, accuracy is not a good measure of performance. A pair of measures that are widely used when classes are unbalanced are *Precision* and *Recall*. Precision is the fraction of cases predicted as positive that are actually positive:

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}. \tag{6.4}$$

Recall is defined as the fraction of positive cases that have been correctly predicted as such:

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}. \tag{6.5}$$

For instance, suppose we want to build a system for the detection of counterfeit banknotes. We have a test set of 10 000 genuine cases (negative class) and 200 counterfeit cases (positive class). If our system detects as counterfeit 300 negative cases and 100 positive cases we have the following measures:

$$\begin{aligned} \text{Accuracy} &= \frac{100 + 9700}{100 + 100 + 9700 + 300} = 98.4\%, \\ \text{Precision} &= \frac{100}{100 + 300} = 25\%, \\ \text{Recall} &= \frac{100}{100 + 100} = 50\%. \end{aligned} \tag{6.6}$$

Note how accuracy seems a very optimistic measure for the system which misses half of the counterfeit banknotes. Precision and recall are more informative: they say that 75% of detections are false alarms (Precision = 25%) and that only half of the counterfeit banknotes are detected (Recall = 50%).

There is a trade-off between precision and recall and many models can be tuned to favor one or the other. For many models it is enough to change the bias b obtained during training. High values of b would make the model more likely to output the positive class, reducing the number of false negatives and increasing the number of false positives. This would make the model more "recall oriented". At the same time, however, the number of false positives would also increase, degrading the precision of the model. Low values of b, for the opposite reasons, would make the model "precision oriented". It is common to consider multiple values of b and to plot the resulting Precision and Recall. One example is shown in Figure 6.6.

The choice of the best combination of precision and recall depends on the consequences of the two kind of errors for the specific application. However, if we really want to summarize the performance into a single generic score we can use the F_1 measure, obtained as the harmonic mean of Precision and Recall:

$$\text{F}_1 = \frac{2}{\frac{1}{\text{Precision}} + \frac{1}{\text{Recall}}} = 2\frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}. \tag{6.7}$$

Alternatively, the *Area Under the Curve* (AUC) in the precision-recall plot is often used as a performance measure independent on a particular choice b.

Precision, Recall and related measures are widely used in domains in which the two classes represents asymmetric concepts. In particular, they are used in:

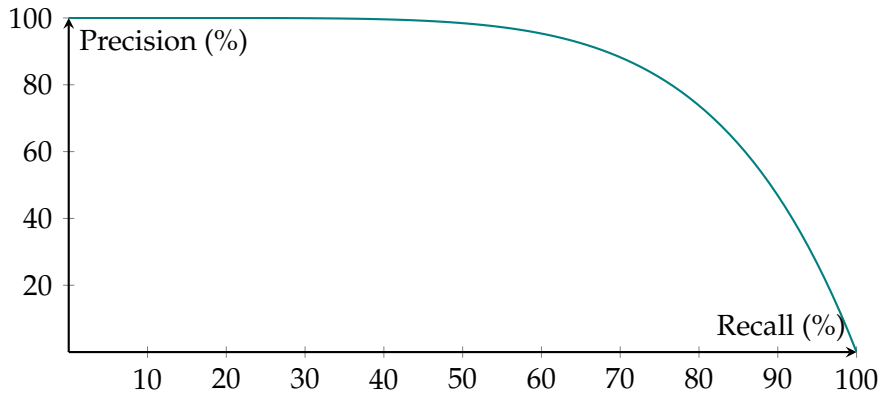


Figure 6.6: Precision-recall curve obtained by varying the bias of a classification model.

- information retrieval, where documents (web pages, for instance) are classified as relevant or not-relevant (for a specific query of the user);
- object detection in images (for instance face detection), where a specific kind of object of interest represents the positive class and everything else is part of the negative class;
- trigger word detection, in which a device processes input sounds trying to identify when the user is pronouncing a specific word or group of words (typically used to activate speech based interfaces).

6.5 Summary

In this lecture we addressed the issue of model selection, which consists in the choice of the model, and of its hyperparameters. In particular, we discussed:

- the need of a *validation set* to evaluate the choices keeping the test set for unbiased estimations of the performance of the selected model;
- a possible workflow for model selection including how to divide data in training, test and validation sets;
- the grid search technique for the selection of the hyperparameters;
- the use of k-fold cross-validation for those cases where there is not enough data to form a validation set;
- the k-nearest neighbors classifier, and the selection of its parameter k through the leave-one-out method;

- the use of the confusion matrix and of other performance measures to assess the performance of binary classifiers.

Lecture 7

Generative methods

Logistic regression, SVM and many other classification models work by modeling the decision boundary among the classes. These learning methods (i.e. those who focus on the decision boundary) are called *discriminative methods*, since they learn how to discriminate among the classes.

An alternative approach is that of the so called *generative methods*. Generative methods focus on modeling each of the classes. As an example, consider a system that need to learn to distinguish among animals. A discriminative method may learn that, for instance, animals weighting more than two tons are elephants, and other similar rules for the other species. A generative method, instead, will learn to model each species independently from the others. It may learn, for instance, that elephants weight between two and six tons, with an average of four. A new animal will be then classified by evaluating how well it would fit in each class.

In the face of outliers, generative methods tend to be more controllable than discriminative methods. For instance, if in the example above we take a dinosaur (and dinosaurs were not represented in the training set), the discriminative method would happily classify it as an elephant, and with a high confidence, since it would fall far away from the decision boundary. The generative method would see that the dinosaur does not fit well in any of the classes. It may still classify it as an elephant (the largest of the known animals), but it would not be hard to identify it as a dubious case.

Many discriminative methods work by directly modeling the posterior probabilities $P(y|\mathbf{x})$. Generative methods, instead, first model the class conditional density functions $p(\mathbf{x}|y)$, then they derive the posteriors by applying the Bayes rule:

$$P(y|\mathbf{x}) = \frac{p(\mathbf{x}|y)P(y)}{p(\mathbf{x})}, \tag{7.1}$$

where the term $P(y)$ is called *prior probability* and represents the probability that a sample is of class y, before taking into account its features. The term

at the denominator $p(\mathbf{x})$ is just the density function of the distribution of the samples in the feature space.

Given the features $\mathbf{x}$ the method would assign the class $\hat{y}$ maximizing the estimated posterior probability. Note that the term $p(\mathbf{x})$ in (7.1) is the same for all the classes, therefore it can be omitted when taking the maximum:

$$\hat{y} = \arg\max_{y \in \mathcal{Y}} P(y|\mathbf{x}) = \arg\max_{y \in \mathcal{Y}} p(\mathbf{x}|y)P(y), \tag{7.2}$$

Prior probabilities are usually not too hard to estimate. Most of the times is enough to take the proportions of the training samples of each class (sometimes it is even acceptable to just consider all the classes as equally probable). Therefore, the most important step in generative methods is the estimation of the conditional density function $p(\mathbf{x}|y)$ of each class.

The term *generative* comes from the fact that you can think to this kind of methods as modeling the process that generates the data. First you randomly pick a class according to the prior distribution $P(y)$, then you randomly pick a sample of that class from the density $p(\mathbf{x}|y)$ (and for some methods this second step can be further decomposed in multiple random choices).

7.1 Gaussian discriminant analysis

The most straightforward way to design a generative method is to assume that $p(\mathbf{x}|y)$ is some known parametric distribution. The most common choice is the *Gaussian distribution* (also called the *normal distribution*), and the corresponding classification model is called *Gaussian Discriminant Analysis* (GDA). In the multivariate case (i.e. more than one dimension) the Gaussian distribution depends on the parameters $\boldsymbol{\mu} \in \mathbb{R}^n$ (the mean vector) and $\Sigma \in \mathbb{R}^{n \times n}$ (the covariance matrix). The Gaussian distribution is defined as (see Section 7.1 for more information about it):

$$\mathcal{N}(\mathbf{x}; \boldsymbol{\mu}, \Sigma) = \frac{1}{(2\pi)^{n/2} |\Sigma|^{1/2}} \exp\left(-\frac{1}{2}(\mathbf{x} - \boldsymbol{\mu})^\intercal \Sigma^{-1} (\mathbf{x} - \boldsymbol{\mu})\right). \tag{7.3}$$

In GDA we assume that the samples of each class are normally distributed. For each class we will have a different mean vector $\boldsymbol{\mu}_y$, $y \in \{0, 1, \dots k-1\}$. For the covariance matrix we may follow two different strategies:

- in the *heteroscedastic* model (i.e. with different dispersion) each class has a different covariance matrix Σ_y;
- in the *homoscedastic* model (i.e. uniform dispersion) all the classes share the same covariance matrix Σ.

Homoscedastic models assume that the classes have distributions with the same shape, just centered around different mean vectors.

The training of the GDA model consists in estimating the parameters of the distributions. Given the training set:

$$\{(\mathbf{x}_0, y_0), (\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_{m-1}, y_{m-1})\}, \ \mathbf{x}_i \in \mathbb{R}^n, y_i \in \{0, 1, \ldots, k-1\}, \quad (7.4)$$

the mean vectors can be estimated with the sample means:

$$\boldsymbol{\mu}_y \simeq \frac{1}{m_y} \sum_{y_i = y} \mathbf{x}_i, \quad (7.5)$$

where m_y is the number of training samples of class y (with an abuse of notation we use $\boldsymbol{\mu}_y$ to denote both the parameter of the distribution and its estimate). Similarly, the class covariances are estimated on the training set:

$$\Sigma_y \simeq \frac{1}{m_y} \sum_{y_i = y} (\mathbf{x}_i - \boldsymbol{\mu}_y)(\mathbf{x}_i - \boldsymbol{\mu}_y)^\intercal. \quad (7.6)$$

For the homoscedastic model the covariance Σ is the average of the class covariances weighted by prior probabilities:

$$\Sigma \simeq \sum_{y=0}^{k-1} P(y)\Sigma_y. \quad (7.7)$$

Note that this is different than just taking the covariance of all samples.

After training GDA models assign to a new sample $\mathbf{x}$ the class $\hat{y}$ with the highest posterior probability $P(y|\mathbf{x})$ or, equivalently, the expression $p(\mathbf{x}|y)P(y)$, as defined in Equation (7.2). Instead of maximizing it we can minimize the negative logarithm, so that the expression can be simplified:

$$\begin{aligned} \hat{y} &= \underset{y \in \{0,\ldots,k-1\}}{\arg\min} \ -\log(p(\mathbf{x}|y)P(y)) \\ &= \underset{y \in \{0,\ldots,k-1\}}{\arg\min} \ -\log p(\mathbf{x}|y) - \log P(y). \end{aligned} \quad (7.8)$$

At this point we can plug in the Gaussian in place of the conditional density:

$$\begin{aligned} -\log p(\mathbf{x}|y) &= -\log\left(\frac{1}{(2\pi)^{n/2}\left|\Sigma_y\right|^{1/2}} \exp\left(-\frac{1}{2}(\mathbf{x} - \boldsymbol{\mu}_y)^\intercal \Sigma_y^{-1}(\mathbf{x} - \boldsymbol{\mu}_y)\right)\right) \\ &= \frac{n}{2}\log(2\pi) + \frac{1}{2}\log\left|\Sigma_y\right| + \frac{1}{2}(\mathbf{x} - \boldsymbol{\mu}_y)^\intercal \Sigma_y^{-1}(\mathbf{x} - \boldsymbol{\mu}_y). \end{aligned} \quad (7.9)$$

Equation (7.9) is a quadratic form having ellipses as level curves. Summing up, the decision function for heteroscedastic GDA is (we can omit the first term since it is independent on the class):

$$\hat{y} = \underset{y \in \{0,\dots,k-1\}}{\arg\min} \frac{1}{2} \log |\Sigma_y| + \frac{1}{2}(\mathbf{x} - \boldsymbol{\mu}_y)^\intercal \Sigma_y^{-1} (\mathbf{x} - \boldsymbol{\mu}_y) - \log P(y). \tag{7.10}$$

A possible implementation of the method in Python is the following:

```
def hgda_train(X, Y):
    k = Y.max() + 1
    m, n = X.shape
    means = np.empty((k, n))
    invcovs = np.empty((k, n, n))
    priors = np.bincount(Y) / m
    for c in range(k):
        means[c, :] = X[Y == c, :].mean(0)
        cov = np.cov(X[Y == c, :].T)
        invcovs[c, :, :] = np.linalg.inv(cov)
    return means, invcovs, priors
```

```
def hgda_inference(X, means, invcovs, priors):
    m, n = X.shape
    k = means.shape[0]
    scores = np.empty((m, k))
    for c in range(k):
        det = np.linalg.det(invcovs[c, :, :])
        diff = X - means[c, :]
        q = 0.5 * ((diff @ invcovs[c, :, :]) * diff).sum(1)
        scores[:, c] = q - 0.5 * np.log(det) - np.log(priors[c])
    labels = np.argmin(scores, 1)
    return labels
```

In the homoscedastic case we can introduce some simplifications. By replacing in (7.9) Σ_y with Σ we obtain:

$$\begin{aligned} -\log p(\mathbf{x}|y) &= \frac{n}{2}\log(2\pi) + \frac{1}{2}\log|\Sigma| + \frac{1}{2}(\mathbf{x} - \boldsymbol{\mu}_y)^\intercal \Sigma^{-1}(\mathbf{x} - \boldsymbol{\mu}_y) \\ &= \frac{n}{2}\log(2\pi) + \frac{1}{2}\log|\Sigma| + \frac{1}{2}\mathbf{x}^\intercal\Sigma^{-1}\mathbf{x} - \boldsymbol{\mu}_y^\intercal\Sigma^{-1}\mathbf{x} + \frac{1}{2}\boldsymbol{\mu}_y^\intercal\Sigma^{-1}\boldsymbol{\mu}_y. \end{aligned} \tag{7.11}$$

Only the terms including $\boldsymbol{\mu}_y$ need to appear in the decision function for homoscedastic GDA:

$$\hat{y} = \underset{y \in \{0,\dots,k-1\}}{\arg\min} -\boldsymbol{\mu}_y^\intercal\Sigma^{-1}\mathbf{x} + \frac{1}{2}\boldsymbol{\mu}_y^\intercal\Sigma^{-1}\boldsymbol{\mu}_y - \log P(y). \tag{7.12}$$

Note that this is a linear classifier with coefficients $\mathbf{w}_y = -\Sigma^{-1}\boldsymbol{\mu}_y$ and biases $b_y = \frac{1}{2}\boldsymbol{\mu}_y^\intercal\Sigma^{-1}\boldsymbol{\mu}_y - \log P(y)$, and with the score for class y given by the expression $\mathbf{w}_y^\intercal\mathbf{x} + b_y$. This way the number of parameters of the model is reduced

to be proportional to the dimension n instead of being proportional to the its square n^2 as in the heteroscedastic case. The implementation in Python of homoscedastic GDA is:

```
def ogda_train(X, Y):
    k = Y.max() + 1
    m, n = X.shape
    means = np.empty((k, n))
    cov = np.zeros((n, n))
    priors = np.bincount(Y) / m
    for c in range(k):
        means[c, :] = X[Y == c, :].mean(0)
        cov += priors[c] * np.cov(X[Y == c, :].T)
    icov = np.linalg.inv(cov)
    W = -(icov @ means.T)
    q = 0.5 * ((means @ icov) * means).sum(1)
    b = q - np.log(priors)
    return W, b
```

```
def ogda_inference(X, W, b):
    scores = X @ W + b.T
    labels = np.argmin(scores, 1)
    return labels
```

Minimum distance classifier

The homoscedastic case can be further simplified by introducing additional assumptions. In particular, we can assume that the components in the feature vectors have uniform variance σ^2 and that they are mutually independent (zero covariance). This means that the covariance matrix is the diagonal matrix $\sigma^2 I$. Under this assumption the expression for the negative logarithm of the conditional density function in (7.9) becomes:

$$-\log p(\mathbf{x}|y) = \frac{n}{2}\log(2\pi) + \frac{n}{2}\log\sigma^2 + \frac{\|\mathbf{x}-\boldsymbol{\mu}_y\|^2}{2\sigma^2}. \tag{7.13}$$

Most of these terms are independent on the class y. If we further assume that classes are equiprobable the decision function is simply the following:

$$\hat{y} = \underset{y\in\{0,\dots,k-1\}}{\arg\min} \|\mathbf{x}-\boldsymbol{\mu}_y\|^2, \tag{7.14}$$

which corresponds, given a new sample $\mathbf{x}$, in taking the class having the closest mean vector. This classifier is called *minimum distance classifier* and is one of the simplest models for supervised classification. Despite its simplicity it may allow to obtain good performance, provided that the feature space is well designed. Minimum distance classifier can be implemented very quickly in Python:

```python
def mindist_train(X, Y):
    k = Y.max() + 1
    n = X.shape[1]
    means = np.empty((k, n))
    for c in range(k):
        means[c, :] = X[Y == c, :].mean(0)
    return means
```

```python
def mindist_inference(X, means):
    k = means.shape[0]
    m = X.shape[0]
    squared_dists = np.empty((m, k))
    for c in range(k):
        squared_dists[:, c] = ((X - means[c, :]) ** 2).sum(1)
    labels = np.argmin(sqdists, 1)
    return labels
```

Comparison of GDA models

Summing up, GDA models presents some advantages if compared with discriminative methods. To begin with, the training process is straightforward and does not require any optimization algorithm. Its bottleneck is often the inversion of the $n \times n$ covariance matrices which requires $O(n^3)$ time with standard methods. The effectiveness of GDA, and of other generative methods, depends on how realistic are the assumptions on the distributions of classes. If the samples are distributed in a completely different way GDA probably is not going to work well. If the data follow all the assumptions then GDA is expected to outperform all the other methods. Concerning the three variants considered, we can observe that:

- heteroscedastic GDA results in a non-linear classifier that takes into account the possibility that the samples of the classes may be distributed in very different ways; the number of parameters of the model is quadratic in the number of features;
- homoscedastic GDA leads to a linear classifier; it is very efficient (the number of parameters can be reduced to be linear in the number of features), but in terms of classification accuracy is often outperformed by linear discriminative methods that do not rely on particular assumptions on the data;
- the minimum distance classifier is very fast and its training consists just in the computation of the class mean vectors; however, its accuracy is acceptable only in a limited number of cases; sometimes it may serve as a baseline to compare against more sophisticated methods.

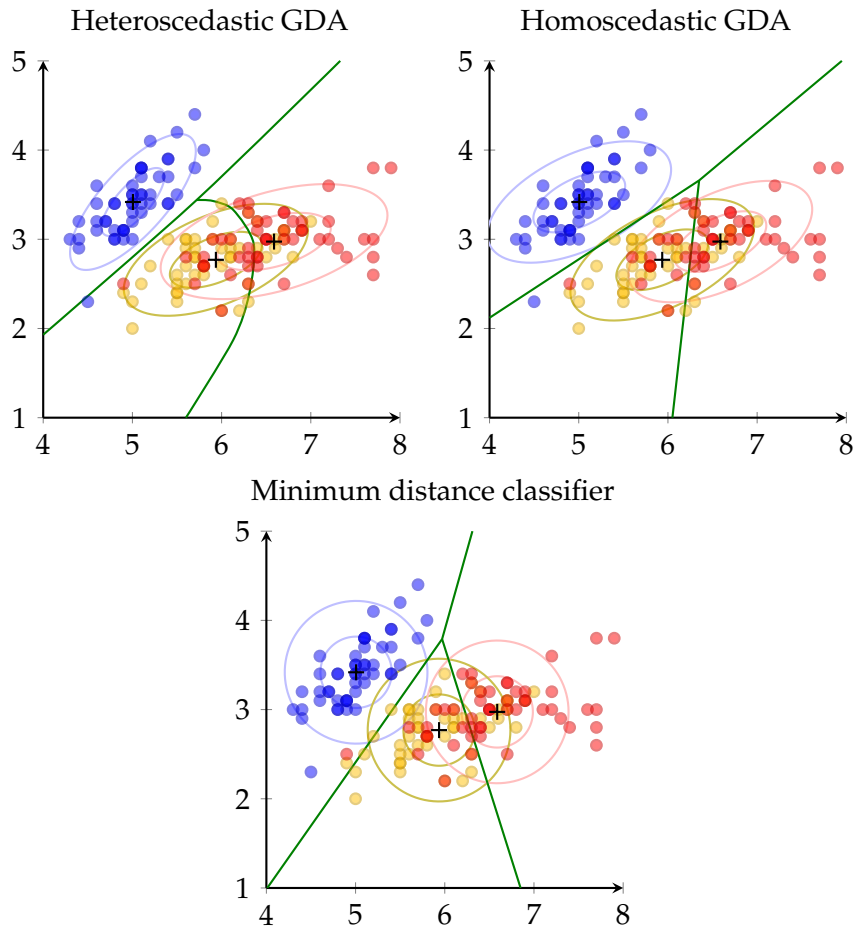


Figure 7.1: Gaussian Discriminant Analysis applied to the iris classification problem. The green lines represent the decision boundary, which is non-linear for heteroscedastic GDA and linear for the other two cases. The ellipses are level curves of the class conditional densities $p(\mathbf{x}|y)$ estimated under the assumptions of the three models: ellipses of three different shapes for heteroscedastic GDA, identical ellipses for homoscedastic GDA, and circles for the minimum distance classifier.

Figure 7.1 shows the three variants of GDA applied to the iris classification problem.

The multivariate Gaussian distribution (optional)

The *Gaussian distribution*, also called *normal distribution*, is probably the most widely known probability distribution. According to the *central limit theorem* the sum of a large number of random independent factors tends toward

this distribution, and this explains its appearance in many natural phenomena. The density function of the Gaussian distribution has the famous "bell" shape (see Figure 7.2), and is defined as:

$$\mathcal{N}(x; \mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right), \tag{7.15}$$

where μ is the mean of the distribution and σ^2 is its variance.

The distribution can be extended to the *multivariate* case (i.e. in multiple dimensions). The parameters are then the mean vector $\boldsymbol{\mu} \in \mathbb{R}^n$ and the *covariance matrix* $\Sigma \in \mathbb{R}^{n \times n}$. The density function is

$$\mathcal{N}(\mathbf{x}; \boldsymbol{\mu}, \Sigma) = \frac{1}{(2\pi)^{n/2} |\Sigma|^{1/2}} \exp\left(-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^\intercal \Sigma^{-1} (\mathbf{x}-\boldsymbol{\mu})\right), \tag{7.16}$$

where $|\Sigma|$ is the determinant of the covariance matrix. The i-th component of $\boldsymbol{\mu}$ is just the expected value of the component x_i of the random variable.

$$\mu_i = E[x_i]. \tag{7.17}$$

The covariance matrix determines how the samples tend to spread around the mean vector. The diagonal element Σ_{ii} is the variance of x_i. The off-diagonal element Σ_{ij} is the *covariance* between x_i and x_j.

$$\begin{aligned} \Sigma_{ii} &= Var[x_i] = E[(x_i - \mu_i)^2], \\ \Sigma_{ij} &= Cov[x_i, x_j] = E[(x_i - \mu_i)(x_j - \mu_j)]. \end{aligned} \tag{7.18}$$

Since the covariance is independent on the order of the variables, the covariance matrix is symmetric. It can be shown that it is also positive semidefinite, that is, it has a complete set of non-negative eigenvalues.

The shape of the density function of the multivariate Gaussian is a multidimensional bell (see Figure 7.3 for some examples). The bell is centered on the mean vector $\boldsymbol{\mu}$ and the level curves are n-dimensional ellipses. When Σ is a diagonal matrix the ellipses have their axes aligned with the axes of the plot with lengths proportional to the elements on the diagonal (i.e. the variances of the components). When Σ is not a diagonal matrix the ellipses are aligned with the eigenvectors with lengths proportional to the eigenvalues (which are the variances of the projections along the eigenvectors).

Given a set of samples $\{\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_{m-1}\}$ the parameters of the multivariate Gaussian distribution can be estimated by maximum likelihood, which consists in the following computations:

$$\begin{aligned} \boldsymbol{\mu} &= \frac{1}{m} \sum_{i=0}^{m-1} \mathbf{x}_i, \\ \Sigma &= \frac{1}{m} \sum_{i=0}^{m-1} (\mathbf{x}_i - \boldsymbol{\mu})(\mathbf{x}_i - \boldsymbol{\mu})^\intercal. \end{aligned} \tag{7.19}$$

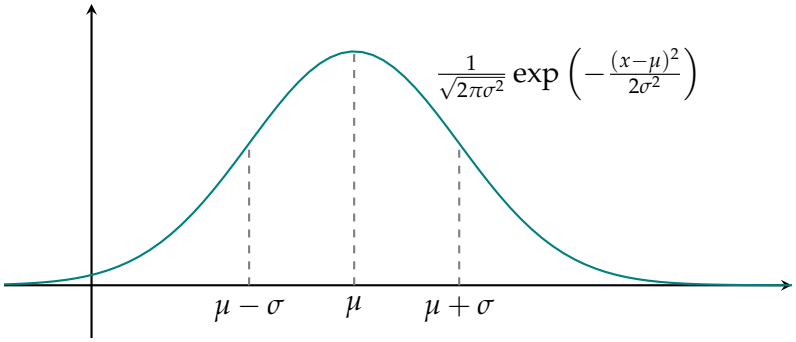


Figure 7.2: The density function of the Gaussian distribution.

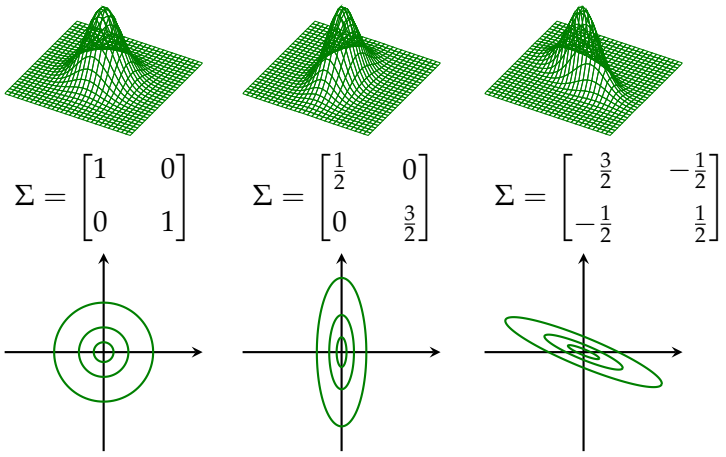


Figure 7.3: The density function of the multivariate Gaussian distribution for three different covariance matrices. The last row shows the level curves of the density functions in the first row.

In python this can be achieved with the functions `mean` and `cov` of the numpy package.

7.2 Naïve Bayes classifier

So far all the models we considered assumed that features have continuous values. There are many classification problems in which features are categorical (i.e. they can assume a finite number of values). Are examples of categorical features the gender of persons (male or female), the color of traffic lights (green, yellow or red), the day of the week (Monday, Tuesday, ..., Sunday) etc.

A feature vectors made only of categorical features can assume only a

finite number of combination of values. In this case, we can build a table representing the density $p(\mathbf{x}|y)$. For instance, if we have a pair of binary features $\mathbf{x} \in \{0,1\}^2$ such a table would have four entries per each class, with the values of $p(\mathbf{x} = (0,0)|y)$, $p(\mathbf{x} = (0,1)|y)$, $p(\mathbf{x} = (1,0)|y)$ and $p(\mathbf{x} = (1,1)|y)$.

However, with n binary features the number of combinations is 2^n, which becomes prohibitively large even for moderate values of n. To make the modeling of $p(\mathbf{x}|y)$ practical we can introduce the assumption that pairs of features are *conditionally independent* given the class. This means that once the class is known, the knowledge about one feature has no effect on our beliefs on the other features. This assumption is called the *Naïve Bayes assumption*. It is a very strong assumption, that greatly simplifies the classification problem. Note that it is different than assuming that the features are independent when the class is not given.

Under the naïve Bayes assumption the density $p(\mathbf{x}|y)$ can be decomposed as a product o simple terms:

$$p(\mathbf{x}|y) = \prod_{j=0}^{n-1} p(x_j|y), \tag{7.20}$$

and each $p(x_j|y)$ can be modeled with a suitable distribution function. Even though it would be possible to model each feature in a completely different way, it is common to consider a single family of parametric distributions. The training algorithm will estimate the parameters for each feature.

Categorical Naïve Bayes

Suppose that each x_j can assume only q_j different values ($x_j \in \{0,1,\ldots,q_j - 1\}$). Then $p(x_j|y)$ would follow a *categorical probability distribution* with parameters $\pi_0^{(j,y)}, \pi_1^{(j,y)}, \ldots, \pi_{q_j}^{(j,y)}$. The parameter $\pi_r^{(j,y)}$ is simply the probability that for samples of class y the j-feature has value r.

$$p(x_j = r|y) = \pi_r^{(j,y)}. \tag{7.21}$$

Given a training set

$$\{(\mathbf{x}_0, y_0), (\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_{m-1}, y_{m-1})\}, x_{ij} \in \{0,1,\ldots,q_j\}, y_i \in \{0,1,\ldots,k-1\}, \tag{7.22}$$

we can just count the occurrences of the feature values for each combination of class and features and use it to estimate the parameters of the probability distributions. Let m_y be the number of training samples of class y ($\sum_y m_y = m$), and let c_{yjr} the number of training samples of class y for which the value

of the j-th feature is r, then we can use the following estimate:

$$\pi_r^{(j,y)} \simeq \frac{c_{yjr}}{m_y}. \tag{7.23}$$

Given a new sample $\mathbf{x}$ the classifier would assign it the class $\hat{y}$ obtained by maximizing the logarithm of the posterior probability:

$$\hat{y} = \underset{y \in \{0,\dots k-1\}}{\arg\max} \log P(y) + \sum_{j=0}^{n-1} \log \pi_{x_j}^{(j,y)}, \tag{7.24}$$

where the logarithm has been introduced, as usual, to transform the products into sums. The prior probabilities $P(y)$ can be given or can be estimated from the training set as $P(y) = \frac{m_y}{m}$.

One drawback of this formulation is that if one of the counters c_{yjr} is zero, then the estimate of $p(\mathbf{x}|y)$ would be zeros whenever $x_j = r$, even if all the other features clearly indicate that the sample is likely to belong to class y. In practice, we would have $\log \pi_{x_j}^{(j,y)} = -\infty$! This pathological case can occur if there is a feature value that is uncommon and the training set is not large enough. A fix to this problem consists in applying the *Laplacian smoothing*, which replaces the estimate in (7.23) with the following:

$$\pi_r^{(j,y)} \simeq \frac{c_{yjr} + 1}{m_y + q_j}. \tag{7.25}$$

In practice Laplacian smoothing introduces one virtual sample for each possible value of the feature, ensuring that every probability is greater than zero. Note that the smoothing preserves the sum of the probabilities $\sum_{r=0}^{q_j-1} \pi_r^{(j,y)} = 1$.

A possible implementation of the categorical naïve Bayes classifier is the following:

```
def categorical_naive_bayes_train(X, Y):
    m, n = X.shape
    q = X.max() + 1
    k = Y.max() + 1
    probs = np.empty((k, n, q))
    class_counts = np.bincount(Y)
    for c in range(k):
        for j in range(n):
            counts = np.bincount(X[Y == c, j], minlength=q)
            probs[c, j, :] = (counts + 1) / (class_counts[c] + q)
    priors = class_counts / m
    return probs, priors
```

Ticket class	$y=0$	$y=1$
First class	71	112
Second class	76	65
Third class	294	92

(a)

Gender	$y=0$	$y=1$
Male	372	80
Female	69	189

(b)

Ticket class	$y=0$	$y=1$
First class	0.16	0.42
Second class	0.17	0.24
Third class	0.67	0.34

(c)

Gender	$y=0$	$y=1$
Male	0.84	0.30
Female	0.16	0.70

(d)

Table 7.1: Number of occurrences (a, b) and probabilities (c, d) for the values of the two features in the two classes of the Titanic problem.

```
def categorical_naive_bayes_inference(X, probs, priors):
    q = probs.shape[2]
    m, n = X.shape
    k = priors.shape[0]
    scores = np.empty((m, k))
    for c in range(k):
        scores[:, c] = np.log(priors[c])
        for j in range(n):
            scores[:, c] += np.log(probs[c, j, X[:, j]])
    labels = np.argmax(scores, 1)
    return labels
```

Example: survivors of the Titanic disaster

Not all Titanic passengers sank with the ship. Some survived the disaster. Two important factors in determining who survived were their gender and the class of their ticket. These are both categorical variables, so we can build a categorical Naïve Bayes classifier that discriminates among survivors ($y=1$) and casualties ($y=0$). From a training set containing the data of 711 passengers we can build the tables with the number of occurrences for the values of the features (see Table 7.1). From these we can compute the probabilities $\pi_r^{(j,y)}$. From the test set we can also estimate the priors obtaining:

$$\begin{aligned} P(0) &= \frac{441}{710} \simeq 0.621, \\ P(1) &= \frac{269}{710} \simeq 0.379. \end{aligned} \tag{7.26}$$

Finally, given a new sample $\mathbf{x}$ we use the model to make a prediction. For instance, for a woman in third class the scores would be:

$$\begin{aligned}(y=0) \rightarrow & \log P(0) + \log \pi_{\text{FEMALE}}^{\text{GENDER},0} + \log \pi_{\text{THIRD}}^{\text{CLASS},0} = \\ & \log 0.621 + \log 0.16 + \log 0.67 \simeq -2.709, \\ (y=1) \rightarrow & \log P(1) + \log \pi_{\text{FEMALE}}^{\text{GENDER},1} + \log \pi_{\text{THIRD}}^{\text{CLASS},1} = \\ & \log 0.379 + \log 0.70 + \log 0.31 \simeq -2.498.\end{aligned} \tag{7.27}$$

and the model would predict survival $\hat{y} = 1$. On a test set of 177 cases this very simple model obtains an accuracy of 76%, which is not great, but not even terrible.

Multinomial Naïve Bayes classifier

The naïve Bayes classifier is especially suitable for text classification. A document (we use the term document to denote any kind of text) can be represented by the words it contains, and we can assume that the words are taken from a vocabulary of n terms. With the categorical model we can represent each word as a feature, and its value would be number of occurrences of that word in the document. This would corresponds to the following generative process: (i) pick the class y; (ii) for each word j in the vocabulary, pick the number of times it should be put in the text according to $p(x_j|y)$ as defined in 7.21. Note that in the categorical model features must have a finite number of possible values while, at least in principle, there is no limit on the number of times a word can appear in a document. More importantly, it is unrealistic to imagine that, for a given word, each number of occurrences is modeled separately.

A more realistic generative process would be the following: (i) pick the class y; (ii) for each position in the document, pick a word according to a new categorical distribution defined on the vocabulary (let $\pi_{y,j}$ the probability to pick the j-th word); (iii) build the feature vector in such a way that x_j is the number of times in which the j-th word has been picked. Here the "naïve" assumption is that each word is sampled independently on the others, given the class. For this process the whole feature vector $\mathbf{x}$ would be a sample of a *multinomial* distribution with parameters $\pi_{y,0}, \pi_{y,1}, \ldots, \pi_{y,n-1}$:

$$\begin{aligned}p(\mathbf{x}|y) &= \frac{(x_0 + x_1 + \cdots + x_{n-1})!}{(x_0!) \times (x_1!) \times \cdots \times (x_{n-1}!)} (\pi_{y,0}^{x_0} \times \pi_{y,1}^{x_1} \times \cdots \times \pi_{y,n-1}^{x_{n-1}}) \\ &= \frac{\left(\sum_{j=0}^{n-1} x_j\right)!}{\prod_{j=0}^{n-1}(x_j!)} \prod_{j=0}^{n-1} \pi_{y,j}^{x_j}.\end{aligned} \tag{7.28}$$

If you are not familiar with this distribution, just consider this example: suppose that our document is just "`home sweet home`". If the word "`home`" is

the number 42 in our vocabulary and the word "`sweet`" is the number 77, then the probability to pick exactly that document out of all the documents of length three would be

$$p(\mathbf{x} = (0, \ldots, 0, 2, 0 \ldots, 0, 1, 0, \ldots, 0)|y) = \frac{3!}{2! \times 1!} \left(\pi^2_{y,42} \times \pi^1_{y,77} \right), \quad (7.29)$$

where the product in the right would be the probability to pick exactly the sequence "`home sweet home`". However, there are other two documents that would result in the same feature vector: "`home home sweet`" and "`sweet home home`". In fact, the fraction on the left accounts for this: the numerator is the number of permutations of three words (6), while the denominator corrects the number of combinations by taking into account the repetitions of the word "`home`".

This particular way of representing text documents is called *Bag of Words* (BoW) since it disregards the order in which the words occur, recording just how many times each term in the vocabulary is in the document.

The Multinomial Naïve Bayes classifier chooses the class which maximizes the product $p(\mathbf{x}|y)P(y)$. To simplify the expression we can, again, maximize its logarithm instead:

$$\begin{aligned} \log\left(p(\mathbf{x}|y)P(y)\right) &= \log\left(\frac{\left(\sum_{j=0}^{n-1} x_j\right)!}{\prod_{j=0}^{n-1}(x_j!)} \prod_{j=0}^{n-1} \pi_{y,j}^{x_j} \right) + \log P(y) \\ &= \log\left(\frac{\left(\sum_{j=0}^{n-1} x_j\right)!}{\prod_{j=0}^{n-1}(x_j!)} \right) + \left(\sum_{j=0}^{n-1} x_j \log \pi_{y,j} \right) + \log P(y), \end{aligned} \quad (7.30)$$

and the first term can be ignored by the classifier since it does not depend on the class. Summing up, the prediction of the model is:

$$\hat{y} = \underset{y \in \{0,\ldots,k-1\}}{\arg\max} \left(\sum_{j=0}^{n-1} x_j \log \pi_{y,j} \right) + \log P(y). \quad (7.31)$$

Note that the whole expression to maximize is linear in the feature values. In fact, we can rewrite the expression for the predicted class in vector form as follows:

$$\hat{y} = \underset{y \in \{0,\ldots,k-1\}}{\arg\max} \left(W\mathbf{x} + \mathbf{b} \right)_y, \quad (7.32)$$

with $W_{ij} = \log \pi_{i,j}$ and with $b_i = \log P(i)$.

The training process for this classifier consists in the estimation of the class/term probabilities $\pi_{y,j}$. Each of them can be estimated by counting the

frequency c_{yj} of the term j in the documents of class y:

$$\pi_{y,j} \simeq \frac{1 + c_{yj}}{n + \sum_{h=0}^{n-1} c_{yh}}, \tag{7.33}$$

where the Laplacian smoothing has been included to prevent issues related to uncommon terms.

Despite its simplicity, the multinomial naïve Bayes classifier applied to bag-of-words features can be very effective, in particular for the classification of documents in classes for which it is possible to reliably estimate the frequency of terms. Its Python implementation is:

```
def multinomial_naive_bayes_train(X, Y):
    m, n = X.shape
    k = Y.max() + 1
    probs = np.empty((k, n))
    for c in range(k):
        counts = X[Y == c, :].sum(0)
        tot = counts.sum()
        probs[c, :] = (counts + 1) / (tot + n)
    priors = np.bincount(Y) / m
    W = np.log(probs).T
    b = np.log(priors)
    return W, b
```

```
def multinomial_naive_bayes_inference(X, W, b):
    scores = X @ W + b.T
    labels = np.argmax(scores, 1)
    return labels
```

Gaussian Naïve Bayes

Even though naïve Bayes is especially popular for discrete features, it can be used for continuous features as well. The "naïve" hypothesis is still that pairs of features are statistically independent given the class label. For Gaussian distributed data this corresponds to the assumption that the covariance matrix Σ_y is diagonal (i.e. that $Cov(x_i, x_j) = 0, \forall i \neq j$). Therefore, the distribution of class samples will have as parameters the mean vector $\boldsymbol{\mu}_y$ and the vector of the variances of the features $\boldsymbol{\sigma}_y^2$:

$$\Sigma_y = \begin{bmatrix} \sigma_{y0}^2 & 0 & \cdots & 0 \\ 0 & \sigma_{y1}^2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \sigma_{y(n-1)}^2 \end{bmatrix}, \tag{7.34}$$

$$
\begin{aligned}
p(\mathbf{x}|y) &= \frac{1}{(2\pi)^{n/2}\left|\Sigma_y\right|^{1/2}} \exp\left(-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^{\intercal}\Sigma_y^{-1}(\mathbf{x}-\boldsymbol{\mu})\right) \\
&= \frac{1}{(2\pi)^{n/2}\prod_{j=0}^{n-1}\sigma_{yj}} \exp\left(-\sum_{j=0}^{n-1}\frac{(x_j-\mu_{yj})^2}{2\sigma_{yj}^2}\right) \\
&= \prod_{j=0}^{n-1}\left(\frac{1}{\sqrt{2\pi}\sigma_{yj}} \exp\left(-\frac{(x_j-\mu_{yj})^2}{2\sigma_{yj}^2}\right)\right),
\end{aligned}
\tag{7.35}
$$

and the last expression shows that the resulting density is the product of n univariate Gaussian densities, one for each individual feature. Taking the negative logarithm brings us to:

$$
\log p(\mathbf{x}|y) = -\frac{n}{2}\log(2\pi) - \frac{1}{2}\sum_{j=0}^{n-1}\log\sigma_{yj}^2 - \sum_{j=0}^{n-1}\frac{(x_j-\mu_{yj})^2}{2\sigma_{yj}^2}, \tag{7.36}
$$

and therefore the final decision $h(\cdot)$ function is:

$$
h(\mathbf{x}) = \underset{y\in\{0,\ldots,k-1\}}{\arg\max} -\frac{1}{2}\sum_{j=0}^{n-1}\log\sigma_{yj}^2 - \sum_{j=0}^{n-1}\frac{(x_j-\mu_{yj})^2}{2\sigma_{yj}^2} + \log P(y), \tag{7.37}
$$

obtained by including the prior probabilities and by omitting the terms independent on the class label. The Python implementation is:

```
def gaussian_naive_bayes_train(X, Y):
    k = Y.max() + 1
    m, n = X.shape
    means = np.empty((k, n))
    vars = np.empty((k, n))
    priors = np.bincount(Y) / m
    for c in range(k):
        means[c, :] = X[Y == c, :].mean(0)
        vars[c, :] = X[Y == c, :].var(0)
    return means, vars, priors
```

```
def gaussian_naive_bayes_inference(X, means, vars, priors):
    m = X.shape[0]
    k = means.shape[0]
    scores = np.empty((m, k))
    for c in range(k):
        diffs = ((X - means[c, :]) ** 2) / (2 * vars[c, :])
        scores[:, c] = -diffs.sum(1)
    scores -=  0.5 * np.log(vars).sum(1)
    scores += np.log(priors)
    labels = np.argmax(scores, 1)
    return labels
```

7.3 Summary

This lecture introduced generative methods as an alternative to discriminative methods. Instead of focusing on the decision boundary generative methods model the classes, in terms of the density function of the features of their samples. Depending on the assumptions made on the shape of the densities, different algorithms are obtained. In particular the following have been discussed:

- Gaussian Discriminant Analysis (GDA), which assumes that features are normally distributed;
- heteroscedastic GDA makes no further assumptions and results in a non-linear classifier with a quadratic number of parameters;
- homoscedastic GDA assumes that the classes share the covariance matrix, and this additional assumption makes the classifier linear;
- the minimum distance classifier is very simple, as a consequence of the strong assumptions it makes (independent features, with the same variance);
- the Naïve Bayes (NB) is a family of methods relying of the "naïve" assumption that features are conditionally independent given the class label;
- in categorical NB the features can assume a finite number of values;
- multinomial NB is especially suitable to deal with histogram representations such as the Bag-of-Word representation of documents;
- the NB assumption can be applied to a Gaussian model, resulting ina simplified version of the heteroscedastic Gaussian analysis.

The accuracy of generative methods depends on how well the underlying assumptions are met in practice, and most of the times discriminative methods perform better. However, their simplicity makes them suitable for those situations where a slow training procedure is not affordable. Generative methods also work well as baseline methods, to check that more sophisticated alternatives are worth the extra complication.

Lecture 8

Feature selection and normalization

Features play a very important role in a machine learning system. Good features allow simple models to obtain good results. Ideally for a classification problem we would like to have features that have similar values for samples of the same class, and in that case we say that the features are *reliable*. We would also like to have features that assume different values for samples of different classes, and in that case we say that the features are *discriminative*.

The computation of features is called *feature extraction*. Sometimes it consists in just gathering the available information from the data. Sometimes it is a very complex process involving advanced transformation of the input data. For instance, in image recognition features are statistics representing important properties of images such as the color distribution, the occurrence of specific patterns etc.

Many techniques have been developed to make features more suitable for the application of machine learning models:

- *feature normalization* methods make features more uniform, simplifying the job of the learning algorithms;
- *feature selection* methods discard features that are not relevant for the specific problem, with a positive impact on both the computational cost and the risk of overfitting the data;
- *dimensionality reduction* methods transform the features reducing, at the same time, their number.

The following sections introduces some of these methods.

8.1 Feature normalization

A common issue is that different features may assume values in very different ranges. For instance, in a medical application one feature may represent the height of patients in meters ($0.5 \leq x_0 \leq 2.5$) and another feature may represent their weight in kilograms ($2 \leq x_1 \leq 200$). This is problematic for many reasons:

- distances between pairs of feature vectors depends mostly on the features with a wide range of values; this would negatively affect models based on distances such as the minimum distance classifier, k Nearest Neighbors and RBF Support Vector Machines;
- linear methods have to learn weights that compensate for the range of the features (large weights for features with small variability and vice-versa) making it difficult to interpret the relevance of the features;
- as a result of the previous point, regularization terms would depend more on features with a small range of values, leaving the others "unregularized";
- many optimization algorithms require more steps to converge.

Luckily, there are very simple techniques that allow to scale all the features to make it have a similar range of values. These techniques usually consist in computing some statistics over the training data and to use them to linearly scale the feature values.

It is very important to keep the statistics computed on the training data and to reuse them also to normalize test and validation data. In particular it is a serious error to normalize test data by using statistics computed on the test set, because the model will be trained under the implicit assumption that all the data is normalization in the same way.

One of the simplest normalization technique consists in scaling all the data in such a way that all features have values in the same range, typically between 0 and 1 (using a different range is trivial). This techniques is called *min-max scaling* and it is based on the computation of the minimum m_j and the maximum M_j values for each feature ($j = 0, 1, \ldots, n-1$):

$$\begin{aligned} m_j &= \min_{i \in \{0,\ldots,m-1\}} x_{ij}, \\ M_j &= \max_{i \in \{0,\ldots,m-1\}} x_{ij}, \end{aligned} \tag{8.1}$$

then each feature of an input feature vector $\mathbf{x}$ is normalized by applying the following linear scaling:

$$\bar{x}_j = \frac{x_j - m_j}{M_j - m_j}. \tag{8.2}$$

This ensures that all the training features assume values between 0 and 1. Note that for data outside the training set normalized values may still fall outside the $[0, 1]$ range (this happens when input features are outside the $[m_j, M_j]$ range). Implementing min-max scaling in Python is straightforward. The following function simulataneously normalizes a training set and a test set.

```
def minmax_normalization(Xtrain, Xtest):
    xmin = Xtrain.min(0)
    xmax = Xtrain.max(0)
    Xtrain = (Xtrain - xmin) / (xmax - xmin)
    Xtest = (Xtest - xmin) / (xmax - xmin)
    return Xtrain, Xtest
```

Sometimes the transformation performed by min-max scaling degenerates due to the presence of a few unrepresentative outliers in the training data. A single very large (or very small) value causes the compression of all the others.

Mean-var scaling is another normalization technique that does not present the same drawback. In mean-var scaling each feature is linearly scaled to have zero mean and unit variance. Training data is used to compute the mean and the standard deviation of each feature:

$$\begin{aligned} \mu_j &= \frac{1}{m}\sum_{i=0}^{m-1} x_{ij}, \\ \sigma_j &= \sqrt{\frac{1}{m}\sum_{i=0}^{m-1}(x_{ij}-\mu_j)^2}, \end{aligned} \tag{8.3}$$

then the components of a given feature vector $\mathbf{x}$ are normalized accordingly

$$\bar{x}_j = \frac{x_j - \mu_j}{\sigma_j}. \tag{8.4}$$

The Python implementation is straightforward:

```
def meanvar_normalization(Xtrain, Xtest):
    mu = Xtrain.mean(0)
    sigma = Xtrain.std(0)
    Xtrain = (Xtrain - mu) / sigma
    Xtest = (Xtest - mu) / sigma
    return Xtrain, Xtest
```

In some applications sparseness of features is a very important property. A feature is sparse when its value is most of the times exactly zero. The bag-of-words representation used in text processing is a typical example of sparsity. Sparse features, in fact, can be stored very efficiently (plus other

advantages). Both min-max scaling and mean-var scaling do not preserve sparsity.

A normalization scheme that is suitable for sparse data is *max abs scaling*. It consists in dividing each feature by the largest absolute value found in the training set:

$$\bar{x}_j = \frac{x_j}{V_j}, \tag{8.5}$$

where V_j is defined as:

$$V_j = \max_{i \in \{0,\dots,m-1\}} |x_{ij}|. \tag{8.6}$$

The Python implementation is the following:

```
def maxabs_normalization(Xtrain, Xtest):
    amax = np.abs(Xtrain).max(0)
    Xtrain = Xtrain / amax
    Xtest = Xtest / amax
    return Xtrain, Xtest
```

Figure 8.1 compares different normalization techniques on a example data set.

Whitening

Another issue is that of highly-correlated features. They risk to be redundant and they can slow down many optimization algorithms. An ideal distribution of data is that where features are uncorrelated and have uniform variance. All this happens when the covariance matrix is the identity I.

The *whitening transform* applies a linear transformation to the data to reach the ideal condition. Given a set of samples $\mathbf{x}_0, \mathbf{x}_1, \dots \mathbf{x}_{m-1}$ it starts by computing the mean vector $\boldsymbol{\mu}$ and the covariance matrix Σ:

$$\begin{aligned} \boldsymbol{\mu} &= \frac{1}{m} \sum_{i=0}^{m-1} \mathbf{x}_i, \\ \Sigma &= \frac{1}{m} \sum_{i=0}^{m-1} (\mathbf{x}_i - \boldsymbol{\mu})(\mathbf{x}_i - \boldsymbol{\mu})^\intercal, \end{aligned} \tag{8.7}$$

then Σ is diagonalized as follows:

$$\Sigma = VDV^\intercal, \tag{8.8}$$

where V is an orthonormal matrix of eigenvectors ($V^\intercal V = I$) and D is the diagonal matrix of the corresponding eigenvalues ($\Sigma V = VD$). Note that Σ is guaranteed to be symmetric and positive semidefinite so that it always has a complete set of real eigenvectors and eigenvalues.

The V, D and $\boldsymbol{\mu}$ are then used to normalize the feature vectors as follows:

$$\bar{\mathbf{x}} = D^{-\frac{1}{2}} V(\mathbf{x} - \boldsymbol{\mu}). \tag{8.9}$$

After the transformation we have that: (i) transformed features have zero mean, (ii) they have unit variance and (iii) they are uncorrelated:

$$\begin{aligned} &\frac{1}{m}\sum_{i=0}^{m-1} \bar{\mathbf{x}}_i = 0, \\ &\frac{1}{m}\sum_{i=0}^{m-1} (\bar{\mathbf{x}}_i - \boldsymbol{\mu})(\bar{\mathbf{x}}_i - \boldsymbol{\mu})^\intercal = I. \end{aligned} \tag{8.10}$$

Whitening can be considered as a more complete meanvar normalization. However, the cost of the diagonalization of Σ is often unjustified, in particular when there are many features. However, the Python implementation is quite simple:

```
def whitening(Xtrain, Xtest):
    mu = Xtrain.mean(0)
    sigma = np.cov(Xtrain.T)
    evals, evecs = np.linalg.eigh(sigma)
    w = evecs / np.sqrt(evals)
    Xtrain = (Xtrain - mu) @ w
    Xtest = (Xtest - mu) @ w
    return Xtrain, Xtest
```

Instance normalization

When all the components in a feature vector represent the same kind of information, a possible alternative to a component by component normalization is that of normalizing the whole vector at once. To do so it could be enough to divide all the components by the norm of the feature vector. L_2 normalization uses the Euclidean norm:

$$\bar{x}_j = \frac{x_j}{\|\mathbf{x}\|_2}, \tag{8.11}$$

and L_1 normalization uses the $\|\cdot\|_1$ norm:

$$\bar{x}_j = \frac{x_j}{\|\mathbf{x}\|_1}. \tag{8.12}$$

The advantage of *instance normalization* (that is the name of this technique) is that we do not need to keep around the statistics for the normalization, since each feature vector is normalized independently. Instance normalization is used, for instance, to normalize bag-of-words representations of text documents to normalize them with respect to the length of the documents.

Instance normalization can be obtained with just a few lines of Python code:

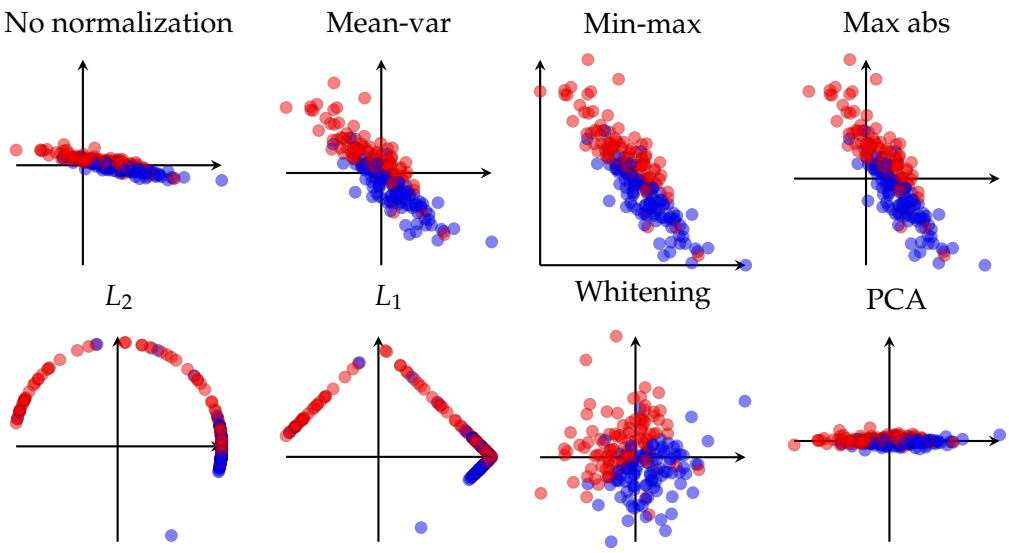


Figure 8.1: Different normalization techniques applied to a data set for binary classification. Classes do not influence normalization and are shown to make more evident how discriminability is affected.

```
def l2_normalization(X):
    q = np.sqrt((X ** 2).sum(1, keepdims=True))
    q = np.maximum(q, 1e-15)  # 1e-15 avoids division by zero
    X = X / q
    return X
```

```
def l1_normalization(X):
    q = np.abs(X).sum(1, keepdims=True)
    q = np.maximum(q, 1e-15)  # 1e-15 avoids division by zero
    X = X / q
    return X
```

8.2 Feature selection

Reducing the number of features can improve the performance of machine learning models. Too many features, in fact, result in models that are too complex favoring overfitting. Features that carry no information about the target labels can be safely dropped, with a gain in the final accuracy and also in the computational resources required to run the learning algorithms.

Feature selection methods try to identify and remove useless features from the data. A very simple technique consists in removing those features that have very little variability. Features with a variance below a set threshold

are discarded. Another technique consists in building a set of models that use a single feature at a time, and then in selecting the features that allow to achieve some minimal performance level (for instance, 55% accuracy for a binary classification problem). This technique removes features that taken alone are not predictive.

A more complex technique is *Recursive feature elimination* (RFE) which consists in building a set of models each one using all but one of the features. The model obtaining the best accuracy on a validation set is retained and the process is repeated until no further improvement is possible. A possible implementation, in which you have to replace the `train` and `inference` functions, is the following:

```
def recursive_feature_elimination(Xtrain, Ytrain, Xval, Yval):
    n = Xtrain.shape[1]
    # Start by using all the features
    best_features = np.ones(n, dtype=np.bool)
    params = train(Xtrain, Ytrain)
    labels = inference(Xval, params)
    best_accuracy = (labels == Yval).mean()
    while True:
        improved = False
        features = best_features.copy()
        for j in features.nonzero()[0]:
            # Evaluate the removal of feature j
            features[j] = False
            params = train(Xtrain[:, features], Ytrain)
            labels = inference(Xval[:, features], params)
            accuracy = (labels == Yval).mean()
            if accuracy > best_accuracy:
                best_accuracy = accuracy
                best_features = features.copy()
                improved = True
            features[j] = True
        # Stop when no improvement is obtained
        if not improved:
            return best_features
```

8.3 Dimensionality reduction

Sometimes removing individual features is not feasible. Even when there are many features, it is possible that none of them is useless when taken alone. Sometimes features are redundant in a non-obvious way. In these cases features can be reduced in number only by replacing them with new features that disentangle the input ones while keeping (most of) the original information.

Dimensionality reduction methods try to transform the feature vectors into low-dimensional representations without losing valuable information. These methods are used to make machine learning models and algorithms more robust to overfitting and less computationally demanding. In some cases they are also used to make features easier to understand. For instance, they can be used to draw data on a bidimensional plot.

Among dimensionality reduction methods, *Principal Component Analysis* is certainly the most widely used.

Principal Component Analysis

Suppose to have a set of n-dimensional feature vectors

$$\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_{m-1}, \tag{8.13}$$

Principal Component Analysis (PCA) looks for d-dimensional linear projections retaining most of the variability of the original set. As new features are selected the *principal components* of the input set of data. The first principal component is the unit vector $\mathbf{v}_0$ maximizing the variance of the projections along it:

$$\mathbf{v}_0 = \underset{\|\mathbf{v}\|_2=1}{\arg\max} \frac{1}{m} \sum_{i=0}^{m-1} (\mathbf{v}^\intercal(\mathbf{x}_i - \boldsymbol{\mu}))^2, \tag{8.14}$$

where $\boldsymbol{\mu} = \sum_{i=0}^{m-1} \mathbf{x}_i$, is the mean vector. By expanding this objective we find out that:

$$\begin{aligned} \frac{1}{m} \sum_{i=0}^{m-1} (\mathbf{v}^\intercal(\mathbf{x}_i - \boldsymbol{\mu}))^2 &= \frac{1}{m} \sum_{i=0}^{m-1} (\mathbf{v}^\intercal(\mathbf{x}_i - \boldsymbol{\mu}))((\mathbf{x}_i - \boldsymbol{\mu})^\intercal \mathbf{v}) \\ &= \mathbf{v}^\intercal \left(\frac{1}{m} \sum_{i=0}^{m-1} (\mathbf{x}_i - \boldsymbol{\mu})(\mathbf{x}_i - \boldsymbol{\mu})^\intercal \right) \mathbf{v} \\ &= \mathbf{v}^\intercal \Sigma \mathbf{v}, \end{aligned} \tag{8.15}$$

where Σ is the covariance matrix estimated on the data set. It can be shown that the maximum of the quadratic form is the largest eigenvalue λ_0 of Σ, and that the direction maximizing it is the corresponding eigenvector $\mathbf{v}_0$.

Similarly, the second principal component is the direction $\mathbf{v}_1$ along which the projections retain the maximal variance not already retained by $\mathbf{v}_0$. It can be shown that $\mathbf{v}_1$ is the eigenvector of Σ corresponding to the second largest eigenvalue λ_1.

The third, fourth... principal components are the remaining eigenvectors of Σ, taken by decreasing eigenvalues $\lambda_2 \geq \lambda_3 \geq \ldots \lambda_{n-1}$. Note that since Σ is symmetric and positive semidefinite it necessarily has a complete basis of orthonormal eigenvectors ($\mathbf{v}_i^\intercal \mathbf{v}_j = 0, \forall i \neq j$).

When used for dimensionality reduction PCA consists in computing the first d principal components and in projecting the data along them. Give a feature vector $\mathbf{x}$ its projection $\bar{\mathbf{x}}$ is defined as:

$$\bar{\mathbf{x}} = \begin{pmatrix} \mathbf{v}_0^\intercal(\mathbf{x}-\boldsymbol{\mu}) \\ \mathbf{v}_1^\intercal(\mathbf{x}-\boldsymbol{\mu}) \\ \vdots \\ \mathbf{v}_{d-1}^\intercal(\mathbf{x}-\boldsymbol{\mu}) \end{pmatrix} = V_d^\intercal(\mathbf{x}-\boldsymbol{\mu}), \tag{8.16}$$

where V_d is the matrix having as columns the first d principal components.

How to determine the number of components to keep? One common strategy is based on the fraction of original variance that is retained by the components. This can be computed from the eigenvalues of Σ as follows:

$$R_d = \frac{\sum_{j=0}^{d-1}\lambda_j}{\sum_{j=0}^{n-1}\lambda_j}, \tag{8.17}$$

and the criterion is to take the minimum number of components d for which R_d is above a set threshold (usually 95% or 99%). A Python implementation of this strategy is the following:

```
def pca(Xtrain, Xtest, mincomponents=1, retvar=0.95):
    # Compute the moments
    mu = Xtrain.mean(0)
    sigma = np.cov(Xtrain.T)
    # Compute and sort the eigenvalues
    evals, evecs = np.linalg.eigh(sigma)
    order = np.argsort(-evals)
    evals = evals[order]
    # Determine the components to retain
    r = np.cumsum(evals) / evals.sum()
    k = 1 + (r >= retvar).nonzero()[0][0]
    k = max(k, mincomponents)
    w = evecs[:, order[:k]]
    # Transform the data
    Xtrain = (Xtrain - mu) @ w
    Xtest = (Xtest - mu) @ w
    return Xtrain, Xtest
```

Figure 8.2 shows an example of application of PCA to a bidimensional data set. The last plot shows how is is possible to partially invert the transformation. The matrix V is orthonormal, which means that $V^{-1} = V^\intercal$. The reconstruction $\bar{\bar{\mathbf{x}}}$ from $\bar{\mathbf{x}}$ is then:

$$\bar{\bar{\mathbf{x}}} = V_d\bar{\mathbf{x}} + \boldsymbol{\mu}, \tag{8.18}$$

Of course, if we retain only a limited number d of components the reconstruction will not be exact. However, it can be shown that, among all the linear

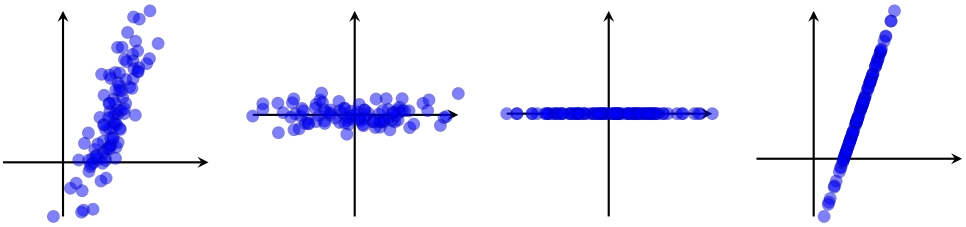


Figure 8.2: Principal Component Analysis applied to a bidimensional data set. From left to right: input data; transformed data; transformed data with one retained component; data reconstructed from the retained component.

transformations $\mathbb{R}^n \rightarrow \mathbb{R}^d$, the one represented by PCA is that minimizing the average reconstruction error on the training set:

$$\frac{1}{m} \sum_{i=0}^{m-1} \|\mathbf{x}_i - \bar{\bar{\mathbf{x}}}_i\|^2. \tag{8.19}$$

Other dimensionality reduction methods

Beside PCA, there are many other dimensionality reduction techniques, based on alternative strategies:

- *Kernel PCA*, which uses the kernel trick to perform non-linear dimensionality reduction;
- *Independent Component Analysis*, which searches for linear features that are statistically independent (instead of being just uncorrelated, as in PCA);
- *Non-negative Matrix Factorization*, which decomposes the data in non-negative features;
- *Multidimensional Scaling*, which preserves the distance between transformed points.

8.4 Summary

In this lectures we briefly addressed some issues related to features. In particular we:

- observed how feature extraction can be of great importance within a machine learning system;

- discussed simple normalization schemes such as minmax scaling and meanvar scaling;
- presented instance normalization as an alternative normalization technique;
- described whitening as an even more complete form of normalization.
- We also briefly discussed some basic feature selection technique.
- We presented PCA as an effective way of performing dimensionality reduction.

Lecture 9

Artificial Neural Networks

The long-term goal of machine learning is to make machines match living beings in their ability to solve problems through learning, and possibly even to go beyond that. Evolution has endowed the most advanced animals with a nervous system capable of very sophisticated learning processes. It is not surprising that researchers took inspiration from these to design machine learning algorithms. The attempt to reproduce the behavior of parts of the animal nervous system brought to the definition of *Artificial Neural Networks* (ANN). The biological inspiration has been progressively disregarded as consequence of new statistical insights and mathematical findings. However, it is worth to introduce ANNs from their original foundations.

Artificial neural networks are modeled as collections of *neurons* communicating through a network of connections. Each neuron performs a simple computation, and the combination of the processing of multiple neurons produces complex behaviors. A single neuron is modeled as having an *activation state* depending on neighboring neurons that send their own activation state through the connection (whose biological counterpart is called *synapse*). The sum of the activations of the neighbors, weighted by the strength of the connections, measures the excitation of the neuron, and if that measure exceeds a threshold, then the neuron is activated.

$$\text{activation} = \begin{cases} 1 & \text{if } \sum_{j=0}^{n-1} w_j x_j \geq \tau, \\ 0 & \text{otherwise,} \end{cases} \tag{9.1}$$

where $x_0, \ldots, x_{n-1}$ are the activations of the neighbors, $w_0, \ldots, w_{n-1}$ are the weights of the corresponding connections and τ is the activation threshold.

This model was proposed in 1958 with the name *perceptron* to solve image recognition problems. A learning algorithm was designed to tune the parameters $w_0, w_1, \ldots, w_{n-1}$ and τ in a supervised way. However, the algorithm (that we already encountered while discussing linear classifiers) works

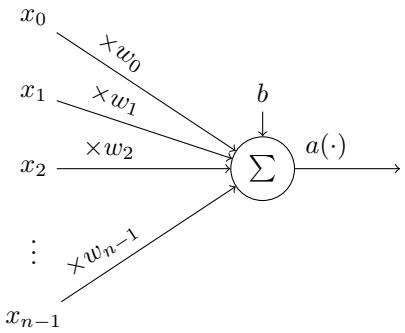


Figure 9.1: Model of artificial neuron.

only for linearly separable classes. The discontinuity of the activation function prevents the application of learning algorithms that use gradient-based optimization algorithms, such as gradient descent.

A neuronal model with a better mathematical behavior can be obtained with a simple adjustment: the binary activation is replaced by a real-valued activation level that is the output of a suitable activation function $a(\cdot)$. Moreover, the threshold τ is replaced by a bias b (this change is not really significant, it is just introduced to make the formulation uniform with respect to the other statistical models). The resulting model is depicted in Figure 9.1, and its mathematical formulation is the following:

$$\text{activation} = a(z) = a(b + \sum_{j=0}^{n-1} x_j w_j), \tag{9.2}$$

where z is the total input to the neuron.

Concerning the activation function, different choices are possible. The most common are:

$$\begin{aligned} a(z) &= \frac{1}{1+e^{-z}}, && \text{(sigmoid)} \\ a(z) &= \tan^{-1}(z), && \text{(arctangent)} \\ a(z) &= \frac{e^z - e^{-z}}{e^z + e^{-z}}, && \text{(hyperbolic tangent)} \\ a(z) &= \max(z, 0). && \text{(ReLU)} \end{aligned} \tag{9.3}$$

Of these the sigmoid is an obvious choice, since it just replaces the activation step of the original non-continuous model with a smooth transition between zero and one. The arctangent and the hyperbolic tangent behave very similarly to the sigmoid, just with a different range of output values ($[-\pi/2, +\pi/2]$ and $[-1, +1]$, respectively). The Rectified Linear Unit activation (ReLU) became very popular with the diffusion of large neural networks

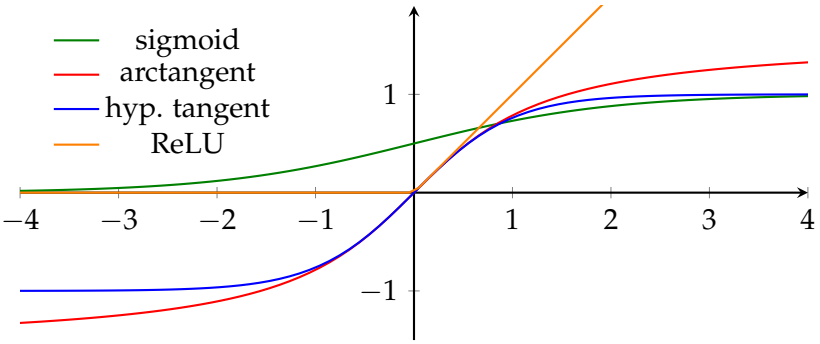


Figure 9.2: Some activation functions for the neuronal model.

since it tends to speed-up the training process. Figure 9.2 shows the plots of these functions.

A single neuron can be used for supervised learning: the inputs will correspond to the features of the samples. The choice of the activation function would reflect the type of the problem. The sigmoid is typically used for binary classification, while the others may be used for regression problems. The training process consists in selecting the weights and the bias that minimize a suitable loss function. For instance cross entropy can be used for binary classification, while L_2 or L_1 is often used for regression. The cross entropy loss with a sigmoid loss function make the neuronal model identical to logistic regression.

In order to design more complicated problems we need to combine multiple neurons in complex ways and to train them simultaneously.

9.1 Multilayer Perceptron

The structure of a neural network is that of a graph where the vertices are the neurons, and the edges are the connections. An important class of neural networks is that of *feedforward* networks, which correspond to directed acyclic graphs, that is, graphs with one-way connections (directed) and without loops (acyclic). In this kind of networks you can always lay out the neurons to make the information flow from one side of the network to the other.

We can divide the neurons of feedforward networks in three groups: input, output and hidden neurons. Input neurons have no incoming connections and their activation is taken directly from the input to the network. Output neurons have no outgoing connections: their activations form the output computed by the network. Hidden neuron have both incoming and outgoing connections: they take the information from other neurons, per-

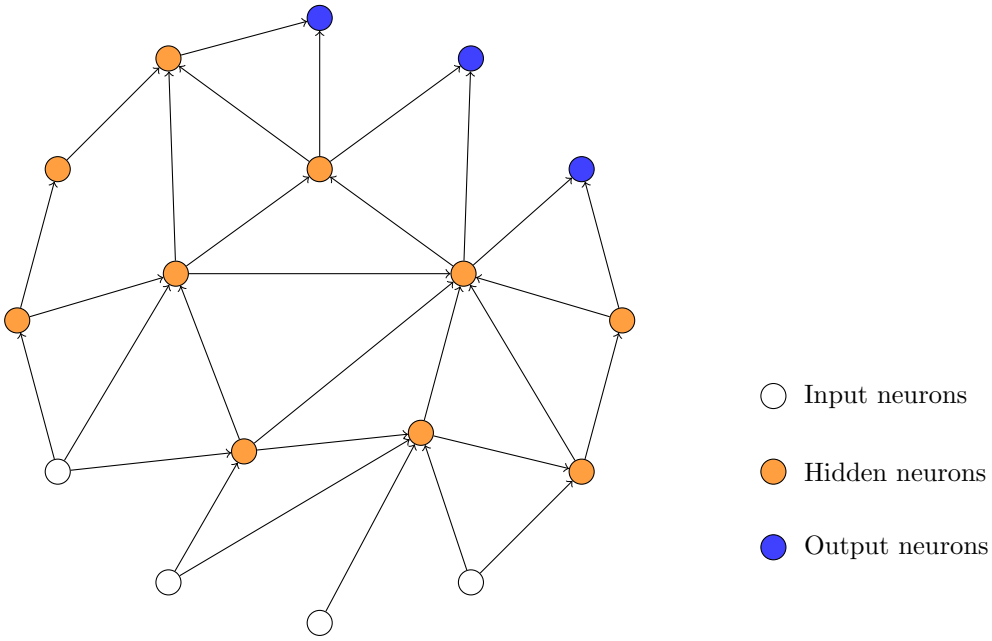


Figure 9.3: Example of a feedforward neural network.

form some processing on it, and pass the result to other neurons. Figure 9.3 shows an example of this kind of network.

Multi-layer perceptrons

A kind of feedforward network that is particularly easy to design and to implement is the *Multi-Layer Perceptron* (MLP). In a multi-layer perceptron neurons are divided into layers. Layers are organized in a sequence, and a neuron in a layer is connected to all the neurons in the following layer. No other connections are allowed. The first layer is composed uniquely of input neurons, while the last layer contains all the output neurons. The intermediate layers are made of hidden neurons. The number of neurons in the input and output layers correspond to the dimension of the problem. For instance, for a classification problem there would be an input neuron for each component of the feature vector, and an output neuron for each class (the class predicted by the network would be that corresponding to the output neuron with the highest activation). The number and the size of the hidden layers determine the complexity of the network. A graphical representation of a MLP is shown in Figure 9.4.

From a mathematical point of view, the MLP is just a repetition of that of a single neuron. Let's consider a MLP of $D+1$ layers: layer 0 is the input layer, layers 1 trough $D-1$ are the hidden layers, and layer D is the output layer

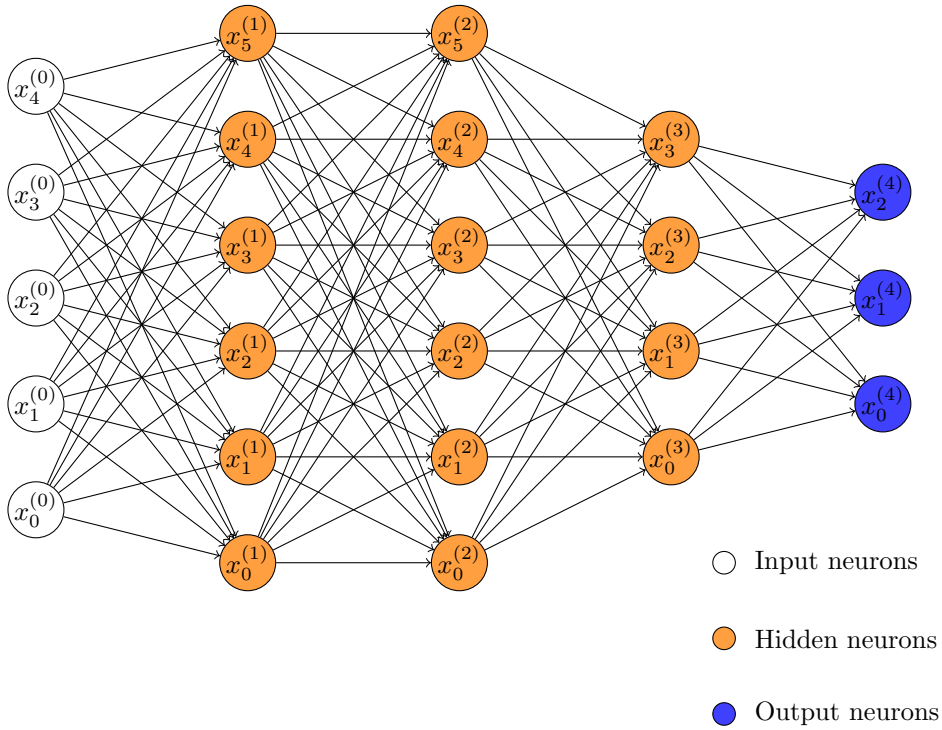


Figure 9.4: Example of a multi-layer perceptron with three hidden layers.

(D is the *depth* of the network). Each layer l contains a given number n_l of neurons. The activations of input neurons are taken from the input features. The activation $x_j^{(l)}$ of the j-th neuron of layer $l > 0$ is computed as:

$$x_j^{(l)} = a(z_j^{(l)}), \tag{9.4}$$

$$z_j^{(l)} = b_j^{(l)} + \sum_{r=0}^{n_l - 1} W_{jr}^{(l)} x_r^{(l-1)}, \tag{9.5}$$

where $z_j^{(l)}$ is the total input to the neuron, $b_j^{(l)}$ is the bias, and $W_{jr}^{(l)}$ is the weight of the connection with the r-th neuron of layer $l-1$. The activation function $a : \mathbb{R} \to \mathbb{R}$ here is assumed to be the same for all neurons, even though the introduction of neuron-dependent activations would not pose any real difficulty. These equations can also be written in a more compact vector form by gathering the activations of layer l into the vector $\mathbf{x}^{(l)}$:

$$\begin{aligned} \mathbf{x}^{(l)} &= \mathbf{a}(\mathbf{z}^{(l)}), \\ \mathbf{z}^{(l)} &= W^{(l)} \mathbf{x}^{(l-1)} + \mathbf{b}^{(l)}, \end{aligned} \tag{9.6}$$

where $\mathbf{a}(\cdot)$ denotes the elementwise application of the activation function. These equations define how to compute the output of the network, given the input. To do so is enough to:

- take the input data as the activations $\mathbf{x}^{(0)}$ of the neurons in the input layer;
- from $\mathbf{x}^{(0)}$ compute the activations $\mathbf{x}^{(1)}$ of the first hidden layer;
- ...
- from $\mathbf{x}^{(l-1)}$ compute the activations $\mathbf{x}^{(l)}$ of the next layer;
- ...
- when all the activations have been computed, take the activations $\mathbf{x}^{(D)}$ of the output layer and use it as the prediction $\hat{\mathbf{y}}$ computed by the neural network.

Training the network to make it produce the "right" outputs is less straightforward.

The backpropagation algorithm

Training a multi-layer perceptron consists in finding the weights $W^{(1)}, \ldots, W^{(D)}$ and the biases $\mathbf{b}^{(1)}, \ldots, \mathbf{b}^{(D)}$ that minimize a suitable loss function $L(\cdot)$ for the problem at hand. For example, in classification problems it is common to follow the same recipe used in multinomial logistic regression which consists in applying the softmax operator to convert the output activations to probability estimates

$$\hat{\mathbf{p}} = \text{softmax}(\mathbf{x}^{(D)}), \tag{9.7}$$

and in computing the cross entropy between these estimates and the target class label:

$$L = \sum_{j=0}^{n_D-1} -\bar{y}_j \log(\hat{p}_j), \tag{9.8}$$

where $\bar{y}_0, \ldots, \bar{y}_{n_D-1}$ is the one-hot vector representation of the class label $y \in \{0, \ldots, n_D - 1\}$. Note that in this case there is no activation ($a(z) = z$) for output neurons before the softmax.

The minimization of the average L can be obtained with gradient-based optimization such as gradient descent. To do so we need to compute the derivatives of the loss with respect to the parameters.

The *backpropagation* algorithm provides a convenient way to compute the derivatives. It is particularly easy to define for MLP, but it can work also

for more complex network architectures. The backpropagation algorithm focuses on the partial derivatives of the loss with respect to the activations of the neurons:

$$\delta_j^{(l)} = \frac{\partial L}{\partial x_j^{(l)}}, \tag{9.9}$$

For the neurons of the output layer these are just the derivatives of the loss function. As already shown for multinomial logistic regression, for the loss in Equation (9.8) they are:

$$\delta_j^{(D)} = \frac{\partial L}{\partial x_j^{(D)}} = \hat{p}_j - \bar{y}_j. \tag{9.10}$$

In vector form this is just $\boldsymbol{\delta}^{(D)} = \hat{\mathbf{p}} - \bar{\mathbf{y}}$.

Backpropagation continues by tracing back the operations executed in the forward pass. From each $\delta_j^{(l)}$, on the basis of Equation (9.4), we can obtain the derivatives with respect to the total inputs to the neurons $z_j^{(l)}$:

$$\frac{\partial L}{\partial z_j^{(l)}} = \frac{\partial L}{\partial x_j^{(l)}} \cdot \frac{\partial x_j^{(l)}}{\partial z_j^{(l)}} = \delta_j^{(l)} \cdot a'(z_j^{(l)}), \tag{9.11}$$

where $a'(\cdot)$ is the derivative of the activation function. The derivatives with respect to the total inputs are used to compute the values of $\delta_r^{(l-1)}$ of the previous layers. To do so the chain rule for the derivative of composite functions is applied to Equation (9.5):

$$\delta_r^{(l-1)} = \frac{\partial L}{\partial x_r^{(l-1)}} = \sum_{j=0}^{n_l - 1} \frac{\partial L}{\partial z_j^{(l)}} \cdot \frac{\partial z_j^{(l)}}{\partial x_r^{(l-1)}} = \sum_{j=0}^{n_l - 1} \left(\delta_j^{(l)} \cdot a'(z_j^{(l)}) \right) W_{jr}^{(l)}. \tag{9.12}$$

The name "backpropagation" comes from the fact that, according to Equation (9.12), the derivatives are computed in the backward direction starting from the output layer and moving towards the input (which is the opposite of the "forward" direction used to compute the output of the network). The expression can be written in vector notation as follows:

$$\boldsymbol{\delta}^{(l-1)} = (W^{(l)})^\intercal (\boldsymbol{\delta}^{(l)} \odot \mathbf{a}'(\mathbf{z}^{(l)})), \tag{9.13}$$

where $\odot$ denotes the elementwise product and $\mathbf{a}'(\cdot)$ the elementwise application of the derivative of the activation function.

The vector $\boldsymbol{\delta}^{(l)}$ allows the computation of the partial derivatives of the loss with respect to the parameters $W_{jr}^{(l)}$ and $b_j^{(l)}$:

$$\begin{aligned} \frac{\partial L}{\partial W_{jr}^{(l)}} &= \frac{\partial L}{\partial z_j^{(l)}} \cdot \frac{\partial z_j^{(l)}}{\partial W_{jr}^{(l)}} = \left(\delta_j^{(l)} \cdot a'(z_j^{(l)})\right) x_r^{(l-1)}, \\ \frac{\partial L}{\partial b_j^{(l)}} &= \frac{\partial L}{\partial z_j^{(l)}} \cdot \frac{\partial z_j^{(l)}}{\partial b_j^{(l)}} = \delta_j^{(l)} \cdot a'(z_j^{(l)}), \end{aligned} \tag{9.14}$$

also expressible in vector notation:

$$\begin{aligned} \nabla_{W^{(l)}} L &= (\boldsymbol{\delta}^{(l)} \odot \mathbf{a}'(\mathbf{z}^{(l)}))(\mathbf{x}^{(l-1)})^\intercal, \\ \nabla_{\mathbf{b}^{(l)}} L &= \boldsymbol{\delta}^{(l)} \odot \mathbf{a}'(\mathbf{z}^{(l)}). \end{aligned} \tag{9.15}$$

It is also common to include some regularization. For neural networks L_2 regularization is the most common and, in this context, is called *weight decay*. To introduce that it is enough to add in the objective function a term $(1/2)\lambda \|W^{(l)}\|_F^2$ for each of the weight matrices. The term $\lambda W^{(l)}$ is then added to the gradient.

Putting together the forward and backward pass, the backpropagation algorithm is therefore:

1. Forward pass:
   a) set the activation of the input neurons $\mathbf{x}^{(0)}$;
   b) for $l = 1, 2, \ldots, D$ do:
      i. compute the input
      $$\mathbf{z}^{(l)} = W^{(l)}\mathbf{x}^{(l-1)} + \mathbf{b}^{(l)};$$
      ii. compute the activations (use the identity for the last layer)
      $$\mathbf{x}^{(l)} = \mathbf{a}(\mathbf{z}^{(l)});$$
   c) compute the posterior probabilities
   $$\hat{\mathbf{p}} = \text{softmax}(\mathbf{x}^{(D)}).$$
2. Backward pass:
   a) compute the output derivatives $\boldsymbol{\delta}^{(D)} = \hat{\mathbf{p}} - \bar{\mathbf{y}}$;
   b) for $l = D-1, D-2, \ldots, 1$ do:

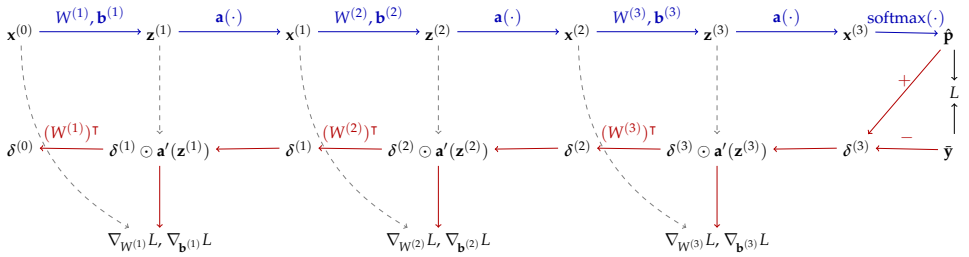


Figure 9.5: Schematic view of backpropagation for a multi-layer perceptron with two hidden layers. This is a classification network ending with a softmax and with the cross entropy loss.

i. compute

$$\boldsymbol{\delta}^{(l)} = (W^{(l+1)})^\intercal (\boldsymbol{\delta}^{(l+1)} \odot \mathbf{a}'(\mathbf{z}^{(l+1)}));$$

ii. compute the derivatives (with L_2 regularization)

$$\nabla_{W^{(l)}} L = (\boldsymbol{\delta}^{(l)} \odot \mathbf{a}'(\mathbf{z}^{(l)}))(\mathbf{x}^{(l-1)})^\intercal + \lambda W^{(l)},$$

iii. and

$$\nabla_{\mathbf{b}^{(l)}} L = \boldsymbol{\delta}^{(l)} \odot \mathbf{a}'(\mathbf{z}^{(l)}).$$

We can use this algorithm to compute the gradients for the parameters corresponding to a single training sample. Given a training set of m samples, the gradients of the average loss is just the average of the single gradients. Finally, the parameters can be updated by gradient descent, with learning rate η:

$$\begin{aligned} W'^{(l)} &\leftarrow W^{(l)} - \eta \nabla_{W^{(l)}} L, \qquad \forall l \in \{1, 2, \dots, D\}. \\ \mathbf{b}'^{(l)} &\leftarrow \mathbf{b}^{(l)} - \eta \nabla_{\mathbf{b}^{(l)}} L, \qquad \forall l \in \{1, 2, \dots, D\}. \end{aligned} \tag{9.16}$$

A visual summary of backpropagation is depicted in Figure 9.5.

Note that the many non-linearities introduced by the activation functions make the objective function non-convex with respect to the parameters. Therefore, gradient descent is not guaranteed to converge to the optimal solution.

The following is a Python class implementing a MLP for classification with one hidden layer. A more complete implementation, supporting any number of hidden layers and a better training procedure, can be found in the PVML library.

```
class MLP:
    def __init__(self, input, hidden, output):
        self.weights1 = np.random.randn(input, hidden) * 0.01
```

```
        self.bias1 = np.zeros(hidden)
        self.weights2 = np.random.randn(hidden, output) * 0.01
        self.bias2 = np.zeros(output)

    def forward(self, X):
        logits1 = X @ self.weights1 + self.bias1
        acts1 = self.activation1(logits1)
        logits2 = acts1 @ self.weights2 + self.bias2
        acts2 = self.activation2(logits2)
        return (logits1, acts1, logits2, acts2)

    def activation1(self, X):
        return np.maximum(X, 0)  # ReLU

    def activation2(self, X):
        e = np.exp(X)
        return e / e.sum(1, keepdims=True)  # Softmax

    def backward(self, Y, forward_values):
        logits1, acts1, logits2, acts2 = forward_values
        grad_acts2 = self.grad_activation2(Y, acts2)
        grad_logits2 = grad_acts2 @ self.weights2.T
        grad_acts1 = self.grad_activation1(logits1, grad_logits2)
        grad_logits1 = grad_acts1 @ self.weights1.T
        return (grad_logits1, grad_acts1, grad_logits2,
            grad_acts2)

    def grad_activation1(self, logits1, grad_logits2):
        return (logits1 > 0) * grad_logits2

    def grad_activation2(self, target, output):
        grad = output.copy()
        grad[np.arange(target.size), target] -= 1
        grad /= target.size
        return grad

    def train(self, X, Y, lr=0.001, steps=1000):
        for step in range(steps):
            acts = self.forward(X)
            grads = self.backward(Y, acts)
            grad_w1 = X.T @ grads[1]
            grad_b1 = grads[1].sum(0)
            grad_w2 = acts[1].T @ grads[3]
            grad_b2 = grads[3].sum(0)
            self.weights1 -= lr * grad_w1
            self.bias1 -= lr * grad_b1
            self.weights2 -= lr * grad_w2
            self.bias2 -= lr * grad_b2
```

Chain rule (optional)

The chain rule is a formula for the computation of the gradient of a composite function. It is an extension of the rule for the computation of the derivative of composite scalar functions (the one that says that if $f(x) = h(g(x))$ then $f'(x) = h'(g(x))g'(x)$).

Let $f : \mathbb{R}^m \to \mathbb{R}$ be a vector function defined as the composition $f(\mathbf{x}) = h(g(\mathbf{x}))$ of two differentiable functions $g : \mathbb{R}^m \to \mathbb{R}^n$, $h : \mathbb{R}^n \to \mathbb{R}$. The partial derivative of f with respect to x_j is given by the following expression:

$$\frac{\partial f}{\partial x_j} = \sum_{r=0}^{n-1} \frac{\partial h(g(\mathbf{x}))}{\partial (g(\mathbf{x}))_r} \cdot \frac{\partial (g(\mathbf{x}))_r}{\partial x_j}, \tag{9.17}$$

which can be written in vector notation:

$$\nabla f(\mathbf{x}) = \mathcal{J} g(\mathbf{x}) \nabla h(g(\mathbf{x})), \tag{9.18}$$

where $\mathcal{J} g(\mathbf{x})$ is the $m \times n$ Jacobian matrix of g at $\mathbf{x}$:

$$(\mathcal{J} g(\mathbf{x}))_{jr} = \frac{\partial (g(\mathbf{x}))_r}{\partial x_j}. \tag{9.19}$$

Depth and width of a MLP

Designing a MLP consists in particular in choosing the number (depth) and size (width) of the hidden layers. Without any hidden layers the MLP just boils down to a linear model. For instance, with a final softmax instead of the activation function, and with a cross-entropy loss it is identical to multinomial logistic regression.

It is known that a feedforward network with a single hidden layer can approximate any continuous function, provided that there are enough hidden neurons. This theorem has been proven by George Cybenko in 1989 and is called the *universal approximation theorem*. We can state the theorem as follows: let $\mathcal{N}(n_0, n_1, n_2)$ be the set of all the functions realized by a multi-layer perceptron with n_0 input neurons, a hidden layer of n_1 neurons and n_2 output neurons:

$$\mathcal{N}(n_0, n_1, n_2) = \left\{ g(\mathbf{x}) := W^{(1)} \mathbf{a}(W^{(0)} \mathbf{x} + \mathbf{b}^{(0)}) + \mathbf{b}^{(1)} \right\}. \tag{9.20}$$

For all functions $f : A \to \mathbb{R}^n$ defined on a compact set $A \subseteq \mathbb{R}^m$, and for all $\epsilon > 0$, there exist $k > 0$ and $g \in \mathcal{N}(m, k, n)$ such that $\|f(\mathbf{x}) - g(\mathbf{x})\| < \epsilon$ for all $\mathbf{x} \in A$.

The theorem tells us that for (almost) any problem we may want to solve, there is a neural network that performs arbitrarily well. Unfortunately the theorem does not tell us how to find the parameters of such a network!

Recently, a result similar to the universal approximation theorem has been demonstrated. It says that the same kind of approximation can be obtained by using hidden layers with a limited width equals to the sum between the number of input and output neurons. This time is the depth to be unbounded.

In practice, the choice of the number and of the size of hidden layers depends on our capability to find the right parameters, that is, to make the network learn to solve the problem. Too many parameters cause overfitting. Too few make the network underfit. It has been empirically observed that deep neural networks (i.e. networks with many hidden layers) tend to achieve greater expressivity than shallow networks with the same amount of parameters. However, the training of deep networks is challenging from the numerical point of view. In the end, each problem requires a different tradeoff between depth and width of the neural network. Figure 9.6 compares networks of varying width and depth in their ability to solve a toy problem. At the two extremes we observe underfitting (top left) and overfitting (bottom right). The optimal configuration should be chosen somewhere in the middle.

9.2 Other kinds of neural networks

Feedforward networks, in particular the multi-layer perceptron, are probably the most widely used type of neural networks. However, many other architectures are used in different domains. Here we briefly describe some of these alternatives.

Convolutional Neural Networks (CNN) are feedforward networks in which some or all the linear layers are replaced by *convolutions*. Convolutions are linear operators in which the output depends only on a small number of input values that form a neighborhood. Weights are shared by neurons in the same layer, and they differ only in the neighborhood of neurons from the previous layer to which the weights are applied to. CNNs are used to process signals (such as images) where convolutions allow to achieve temporal or spatial invariance (for instance, a CNN for image recognition would recognize important structures in the image independently on where they are located). In CNNs non-linear activations are often paired with *pooling layers* that progressively reduce the temporal or spatial resolution of the signal, making it possible to balance the increase in local information produced by convolutions. CNN have been extremely successful in solving image recognition and other computer vision problems.

Recurrent Neural Networks (RNN) are networks where, unlike feedforward networks, there are loops of connections among neurons. These loops are

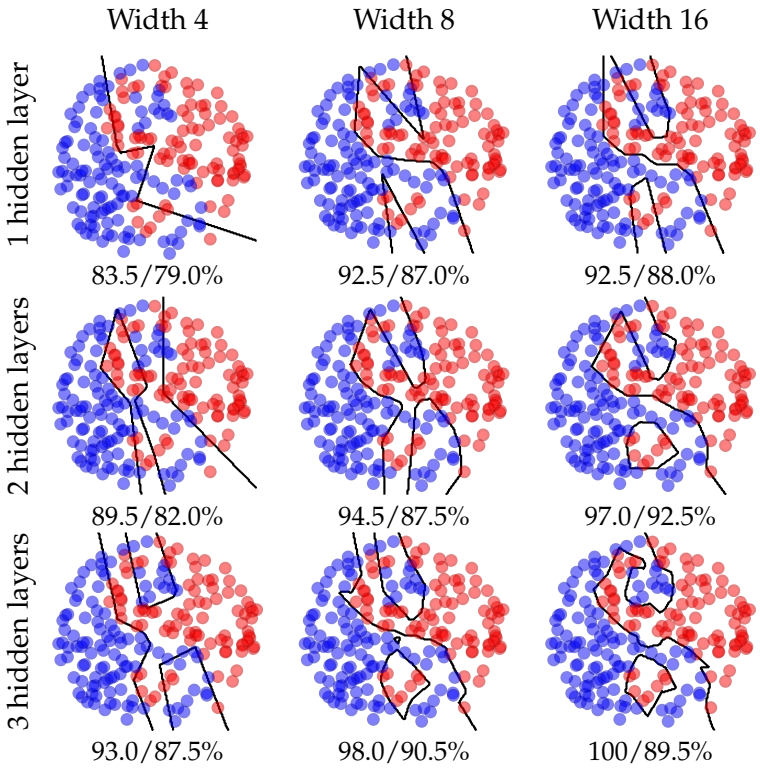


Figure 9.6: Decision boundary of a multi-layer perceptron, varying the number and size of hidden layers. Below each plot are reported the corresponding training and test accuracy (computed on other data from the same distribution).

used to store a memory of past computations. The network is run in steps, and part of the hidden activations are fed back into the network as additional input for the next step. This way the network can be used to process and/or produce sequences of data. RNNs are widely used in the domains of natural language processing, speech recognition etc.

Graph Neural Networks (GNN) are networks computing information over a graph. Computation can occur within each vertex in the graph, across the edges of the graph, or globally on all the information distributed over the graph. GNNs have been successfully used in applications where the information is well represented by a graph. For instance they have been used to analyze molecules, social networks etc.

Autoencoders are feedforward networks for unsupervised learning. They have an hourglass-like shape, with identical input and output layers, and with the hidden layers forming a bottleneck where the information is squeezed. Autoencoders are trained to make their output as close as possible to the in-

put. To make this happen, hidden neurons are forced to learn to find recurring patterns and common structures in the input data.

Generative Adversarial Networks (GAN) are used to generate new data that looks like the data in the training set. For instance, they have been used to generate images of faces of non-existing people, after learning from a training set of real face images. GANs consists in a pair of feedforward networks, a generator and a discriminator. The generator produces data from white noise, while the discriminator learns to recognize the output of the generator from real data taken from the training set. For training there is a single loss function that is maximized by one network and maximized by the other, as if the two were playing a game as adversarial.

9.3 Summary

In this lecture we introduced artificial neural networks. More in detail:

- we reviewed the perceptron as a model for an artificial neuron;
- we described the class of feedforward networks;
- we gave a mathematical model for the multi-layer perceptron;
- we described the backpropagation algorithm that allows to train feedforward networks;
- we discussed the depth and the width of multi-layer perceptrons;
- we gave a brief overview of other neural models used in machine learning.

Lecture 10

Training neural networks

Training neural networks poses new challenges with respect to other less flexible models. Unlike logistic regression and SVM, for instance, the loss function is not convex in the parameters of the model. This means that there may be *local minima*, that is, combinations of parameters for which no improvement is possible by moving around by a small step, even though we may be very far away from the global minimum we are looking for. Gradient descent may get stuck in one local minimum, as shown in the top-left plot of Figure 10.1.

One may get tempted of increasing the learning rate of gradient descent to "jump over" the local minima, but this way the algorithm would risk to miss also the global minimum. This phenomenon is called *overshooting* and is shown in the top-right plot of Figure 10.1. In the case of overshooting gradient descent would bounce around the minimum without being able to

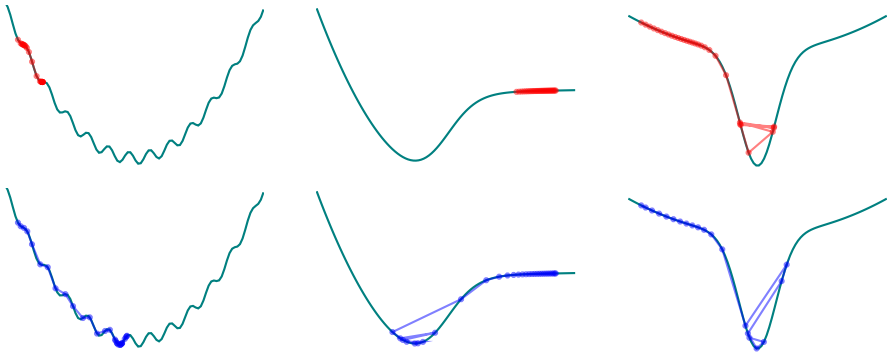


Figure 10.1: Comparison of Gradient Descent without (top row) and with (bottom row) momentum. From left to right, the plots show how momentum helps in dealing with local minima, plateaus and overshooting.

reach the bottom of the valley.

Another issue faced by gradient descent when dealing with non-convex functions is that it may get to regions that are almost flats (*plateaus*) where the gradient is almost null and progresses are very slow. Crossing the plateau requires a large learning rate and a large number of gradient descent steps (see the plot in the middle of Figure 10.1).

In practice, complicated convex functions with multiple local minima, plateaus, saddle points etc. make very difficult the use of plain gradient descent.

10.1 Momentum

Gradient descent can be made more adaptive if some kind of "memory" of past iterations is used to adjust the length of the step. When crossing a plateau, it would be better to take large steps and this could be triggered by observing that in plateaus all the updates point in the same direction. When in a narrow valley, it would be better to take small steps to avoid jumping outside the valley and to dump the oscillations around the bottom. Both cases can be implemented by introducing some *momentum* carrying extending the movement made in the past iterations.

Momentum can be easily understood taking inspiration by physics. Gradient descent with momentum reminds a ball falling down a mountain. The steepness of the terrain (i.e. the gradient) does not give the direction of the movement of the ball, but the direction of acceleration. The speed of the ball (i.e. the step taken by the algorithm) is the accumulation of the acceleration during time.

Each step of gradient descent with momentum consists in the following computations:

$$\begin{aligned} \mathbf{v}' &\leftarrow \alpha \mathbf{v} + \nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}), \\ \boldsymbol{\theta}' &\leftarrow \boldsymbol{\theta} - \eta \mathbf{v}', \end{aligned} \tag{10.1}$$

where $L(\boldsymbol{\theta})$ is the loss function to be minimized with respect to the parameters $\boldsymbol{\theta}$ and η is the learning rate. The step taken by the algorithm is proportional to the momentum term $\mathbf{v}$ that is updated in each step to incorporate the gradient. The coefficient $0 \leq \alpha < 1$ determines how much momentum is conserved between steps. With $\alpha = 0$ the method falls back to the version without momentum. With α close to one, each update influences many steps before its contribution vanish. In practice, α represents a sort of "friction" that limits the acceleration. Usually a large value of the coefficient is used. Typical values are $\alpha = 0.9$ or even $\alpha = 0.99$.

By iterating the computations (10.1) the algorithm computes a sequence momentum vectors $\mathbf{v}_0 \to \mathbf{v}_1 \to \cdots \to \mathbf{v}_t \to \ldots$ and a sequence of parameters $\boldsymbol{\theta}_0 \to \boldsymbol{\theta}_1 \to \cdots \to \boldsymbol{\theta}_t \to \ldots$ defined as:

$$\begin{aligned} \mathbf{v}_t &= \sum_{s=0}^{t-1} \alpha^s \nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}_{t-s-1}), \\ \boldsymbol{\theta}_t &= \boldsymbol{\theta}_0 - \eta \sum_{s=1}^{t} \mathbf{v}_s. \end{aligned} \tag{10.2}$$

The maximum acceleration due to momentum occurs when the gradient is constant $\nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}_t) = \mathbf{g}$:

$$\mathbf{v}_t = \sum_{s=0}^{t-1} \alpha^s \mathbf{g} = \frac{1-\alpha^t}{1-\alpha}\mathbf{g}, \tag{10.3}$$

which, for $t \to \infty$ converge to $\frac{1}{1-\alpha}\mathbf{g}$ which is $\frac{1}{1-\alpha}$ times larger than the step taken by simple gradient descent. For instance, for $\alpha = 0.9$ this corresponds to a factor 10, and for $\alpha = 0.99$ this corresponds to a factor 100.

An sample implementation of gradient descent with momentum is the following:

```
def gradient_descent_with_momentum(loss_grad, theta, lr=0.001,
    momentum=0.9, steps=100):
    v = np.zeros_like(theta)
    for _ in range(steps):
        v = v * momentum + loss_grad(theta)
        theta -= lr * v
    return theta
```

10.2 Initialization

For convex functions it is not important how the parameters are initialized. A perfectly good choice is to start with all the parameters to zero. In fact, the optimization algorithm will converge to the optimum independently on the starting point. For non convex functions, however, starting from zero is a very bad idea.

Recall from the previous lecture the formula for backpropagating derivatives in a multi-layer perceptron:

$$\boldsymbol{\delta}^{(l-1)} = (W^{(l)})^\intercal (\boldsymbol{\delta}^{(l)} \odot \mathbf{a}'(\mathbf{z}^{(l)})). \tag{10.4}$$

If all the weights are initialized to zero all the derivatives will also be zero. Therefore, backpropagation would not update the weights that will stay null. All zeros is a saddle point of the objective function.

It would also be bad to initialize all weights to the same non-zero value. In this case $\delta^{(l-1)}$ will have identical component, and consequently all the weights in the same layer will be updated by the same quantity, staying uniform. The initialization must "break the symmetry", otherwise all hidden neurons would stay the same and learning would be impossible.

A good solution is to randomly initialize the weights. For instance we can sample every weight from a Gaussian distribution with zero mean and σ_w^2 variance. However, we have to choose σ_w^2 carefully. Suppose that the features $x_0, x_1, \dots x_{n-1}$ are randomly generated to be statistically independent and to have σ_x^2 variance, then the components of the product $\mathbf{z} = W\mathbf{x}$ would have zero mean and the variance σ_z^2 would be:

$$\begin{aligned} \sigma_z^2 = \mathrm{Var}[z_i] = \mathrm{Var}[\sum_{j=0}^{n-1} W_{ij}x_j] = \sum_{j=0}^{n-1} \mathrm{Var}[W_{ij}]\mathrm{Var}[x_j] - \mathrm{E}[W_{ij}]^2\mathrm{E}[x_j]^2 \\ = \sum_{j=0}^{n-1} \sigma_w^2 \cdot \sigma_x^2 - 0 \cdot \mathrm{E}[x_j]^2 = n\sigma_w^2 \cdot \sigma_x^2, \end{aligned} \tag{10.5}$$

where we made use of the hypothesis that all variables are mutually independent. In practice the range of output values would be larger than the range of inputs by a factor $\sqrt{n\sigma_w^2}$ that is the ratio between the two standard deviations.

Repeating the process for multiple layers in a neural network would cause the output activations to explode when $\sqrt{n\sigma_w^2} > 1$. With smaller weights we would obtain the opposite effect, and activations would vanish to zero.

The same issue is even more evident during the backward step where it is referred to as the *vanishing gradient* problem (or exploding gradient). To make learning possible we have to set the magnitude of initial weights in a way that makes the outputs fall in the same range of values of the inputs. To do so we can initialize the weights to have zero mean and variance $\sigma_w^2 = 1/n$ (or, equivalently, we can take them with unit variance and then divide by $\sqrt{n}$).

This analysis ignores the activation functions. To take them into account some adjustment is required. With ReLU, for instance, the same reasoning leads to the *Kaiming initialization* technique, that consists in taking weights with zero mean and unit variance and divide them by $\sqrt{n/2}$. Other activation functions require different corrections, but Kaiming can be good enough to start training, after that initialization becomes less important. Biases are less problematic, and are simply initialized to zero.

10.3 Stochastic gradient descent

Recent achievements in the field of neural networks have been obtained by increasing the number of neurons that, in some cases, reached the tens of millions. The training of these huge networks requires similarly huge training sets. Gradient descent is hard to use in these cases. In fact, gradient descent must process all the training samples to compute the gradient of the average loss. This makes the computational cost of each optimization step linear in the size of the training set. The alternative to this strategy (that often is called *batch optimization*) is to use only part of the training set to compute the steps.

The gradient of the average loss is the average of the gradients of the loss L_i computed for each individual sample:

$$\nabla_{\boldsymbol{\theta}} \left(\frac{1}{m} \sum_{i=0}^{m-1} L_i(\boldsymbol{\theta}) \right) = \frac{1}{m} \sum_{i=0}^{m-1} \nabla_{\boldsymbol{\theta}} L_i(\boldsymbol{\theta}), \tag{10.6}$$

and the average gradient can be estimated even from a sub-sample of the training set. *Stochastic Gradient Descent* (SGD) extends this concept to the limit: each optimization step is computed by using a single training sample randomly taken from the training set.

Of course, sometimes the resulting update of the parameters will be in the wrong direction. However, most of the time the update will be approximately in the right direction and the cumulative effects of the updates will be a progressive improvement of the average loss function. Compared to batch optimization, the updates of stochastic optimization will be less accurate, but a lot faster to compute. The behavior of the algorithm will be unpredictable as it randomly updates the parameters. But in practice stochastic gradient descent converges to the solution more quickly than batch optimization.

One drawback of SGD is that it is a lot harder to monitor, since the loss function randomly oscillates up and down, making it difficult to decide when to stop the algorithm. In fact, once close to the minimum, the algorithm will bounce around it instead of stopping on top of it. A common improvement is to reduce the learning rate as the algorithm progresses towards the solution.

Another issue of SGD is that it may be difficult to exploit hardware capable of parallel processing. In fact, the processing of one training sample can start only when the processing of the previous one is terminated. A partial solution to these problems is to compute the optimization steps using *mini-batches* of training samples. In practice, this consists in computing the average gradient for a small sample (the mini-batch) of the training set (from ten to a couple of hundreds). Computation can be performed in parallel on the elements of the mini-batch. Moreover, mini-batches dampen the oscillations of the loss function. Beware, however, that large mini-batches slow

down the minimization of the average loss. A good practice is to use the smallest mini-batch size among those exploiting all the hardware available.

In practical implementations, instead of sampling the mini-batch, it is common to shuffle the training set and to form the mini-batches by taking the samples in that order. When all the training samples have been processed, the training set is shuffled again and the process is repeated. Every pass through the training set is called *epoch*, and multiple epochs are required to complete the training.

A possible implementation of stochastic gradient descent with mini-batches is the following (`loss_grad` is a function returning the gradient of the average loss computed on a mini-batch):

```
def stochastic_gradient_descent(X, Y, loss_grad, theta,
    batch_size, lr=0.001, epochs=10):
    m = X.shape[0]
    for epoch in range(epochs):
        # Random shuffle the training set
        perm = np.random.permutation(m)
        X = X[perm, :]
        Y = Y[perm]
        for i in range(0, m, batch_size):
            X_batch = X[i:i + batch_size, :]
            Y_batch = Y[i:i + batch_size]
            grad = loss_grad(X_batch, Y_batch, theta)
            theta -= lr * grad
    return theta
```

Adding moment to stochastic gradient descent is straightforward.

10.4 Example: digit classification

Optical character recognition is a good example of classification problem. Given an image of a character the goal is to identify its digital code. The MNIST data set is a popular benchmark of this type of problems [1]. It includes images of handwritten digits (one digit per image). Each image is a matrix of 28×28 pixels, each one representing the measured brightness in that particular location. Figure 10.2 shows a sample of the data set.

The data set include 60 000 training and 10 000 test images. The goal is to build a ten-class classifier with the lowest error rate. The problem is hard enough to show the limitations of many classification strategies. The lowest error rate reported in the literature is about 0.2%.

Neural networks have been used to address this problem. We will experiment with a multi-layer perceptron. The input layer of the network has

[1] `http://yann.lecun.com/exdb/mnist/`

Figure 10.2: A selection of images from the MNIST data set. Rows contains examples taken from the ten classes.

$28 \times 28 = 784$ neurons, one for each pixel in the images. The activation of input neurons will be a value representing the amount of ink in the corresponding pixel ($0 \rightarrow$ white, $1 \rightarrow$ black). The order we use to map the 28×28 grid of pixels into the 784-dimensional vector of activations is not relevant. We can concatenate the rows or the columns, or use any other scheme, the result will not change.

The output layer is composed of ten neurons, one for each class of digits. Their activations will be the estimated distribution of probability over the ten classes. We will try with zero to three hidden layers of 128, 64 and 32 neurons with a ReLU activations. A graphical representation of the variant with two hidden layers is shown in Figure 10.3.

The network has been trained with stochastic gradient descent ($\eta = 10^{-4}$, $\lambda = 10^{-5}$, 250 epochs, mini-batches of 100 samples). After each epoch the error rates on the training and the test set have been computed. They are reported in Figure 10.4. Some additional data is given in Table 10.1. It is clear that using hidden layers makes a big difference. Without them the model converges to a weak solution that misclassifies 7.67% of test examples. With hidden layers the error rate decreases to 2.11%. With hidden layers there is some overfitting. In fact, the training error decreases with the number of hidden layer, and the model with three hidden layers is able to classify correctly all training examples. In terms of test error, the best model is the

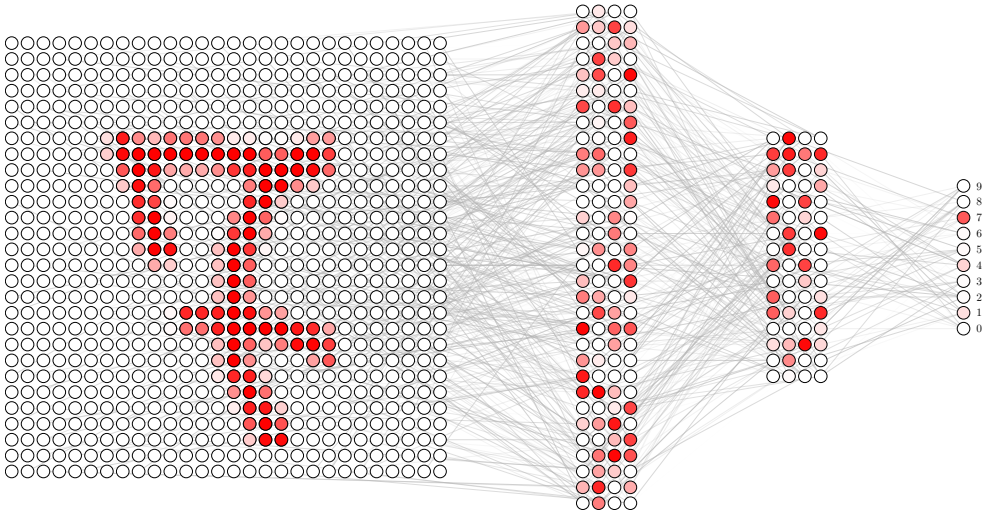


Figure 10.3: Scheme of a multi-layer perceptron for digit classification. The two-dimensional layout of neurons in layers is just for convenience of representation. For the same reason, only a fraction of the connections between neurons has been drawn.

Hidden layers	Number of neurons	Number of parameters	Training error (%)	Test error (%)
0	784 + 10	7850	7.17	7.67
1	784 + 128 + 10	101 770	0.57	2.11
2	784 + 128 + 64 + 10	109 386	0.01	2.22
3	784 + 128 + 64 + 32 + 10	111 146	0.00	2.20

Table 10.1: Error rates on the MNIST training and test sets, obtained by using a MLP with a different number of hidden layers.

one with a single hidden layer. However, more layers allow to reach a similar accuracy with fewer training epochs.

To understand why hidden layers are useful it is insightful to analyze the learned weights for the model without them (see Figure 10.5). Without hidden layers input neurons are directly connected to the output. Therefore, for each class there is a weight for each pixel in the image representing how much the ink on that location contributes positively or negatively to the likelihood that the whole image is of that class. For instance, ink in a central pixel would contribute positively to the classes 1, 2, 3 and 8 and negatively to the others. The contribution of a pixel to the a class is independent on the structures it makes with other pixels. For instance, it does not matter if it is part of a horizontal or a vertical stroke.

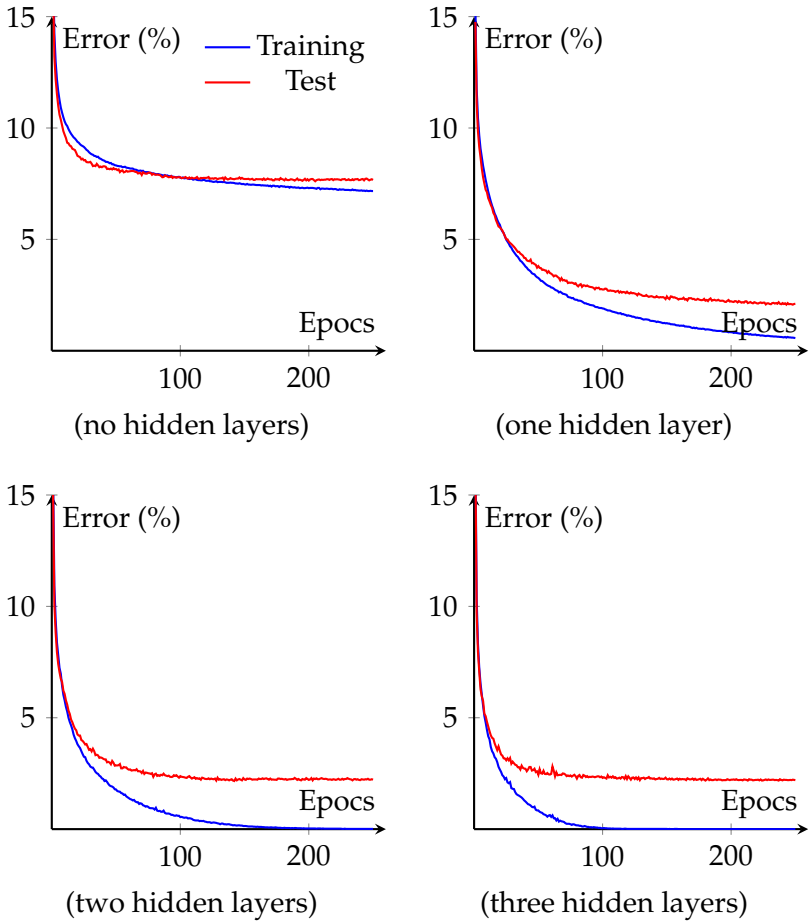


Figure 10.4: Error rates on the MNIST training and test sets, obtained by using zero to three hidden layers.

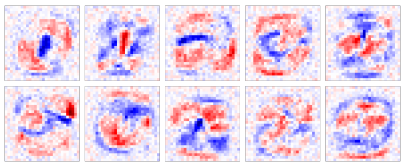


Figure 10.5: Weights of the MLP without hidden layers. The positive values are in red, the negatives in blue. Each group of weights represents the connections from the 784 input neurons to one of the ten output neurons.

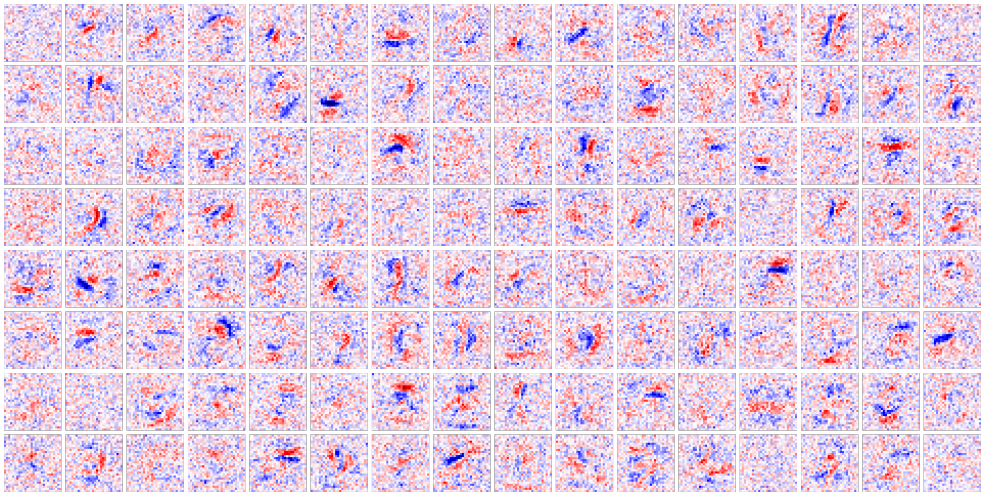


Figure 10.6: Weights of the MLP with one hidden layer. Each group of weights represents the connections from the 784 input neurons to one of the 128 hidden neurons. The positive values are in red, the negatives in blue.

Figure 10.6 shows, instead, the weights connecting input and hidden neurons of the model with a single hidden layer. The 128 hidden neurons learned to identify simple strokes located in a specific part of the image. The weights connecting the hidden and the output neurons will aggregate these intermediate features into class scores. For instance, there is a neuron specialized in recognizing arcs in the upper part of the image. The output neuron will aggregate all the evidences for a specific class. For instance, the neuron associated to the class 0 is positively connected to hidden neurons that get activated in presence of arcs in the upper or in the lower part of the image, and is negatively connected to those that react to central horizontal or vertical edges. This way it is possible to recognized different styles of writing the same digit, and to better tolerate exceptions and variations.

10.5 Summary

In this lecture we faced some advanced topics about optimization that are required to make possible the training of neural networks. In particular:

- we introduced gradient descent with momentum to address some of the issues that regular gradient descent presents when dealing with non-convex objective functions;

- we realized that initializing the weights of neural networks is not trivial

and that it requires some special technique such as Kaiming initialization;

- we described how Stochastic Gradient Descent (SGD) accelerates the training;
- we discussed the role of mini-batches in SGD.

The lecture also presented a complete case study in which we designed a neural network that solves digit recognition, a real-world classification problem.

Lecture 11

Convolutional Neural Networks

Machine learning has a long tradition as a powerful tool in the field of pattern recognition, where it is used to analyze signals such as sound waves, images etc. Multilayer perceptrons can be used for this, as we have seen in the MNIST example, but with important limitations. The major problem is that the number of parameters depends on the size of the input.

For MNIST we needed $28 \times 28 = 784$ input neurons. With, for instance, 128 neurons in the first hidden layer, we also need $784 \times 128 = 100\,352$ weights. For the recognition of larger images the number of parameters would explode. Suppose, for instance, the image to classify has size 200×200 pixels (which is still of moderate size) and that we want to have 1000 hidden neurons, then the number of parameters would be $200 \times 200 \times 1000 = 40\,000\,000$! And just for the first layer!

Since we cannot afford to connect all the input neurons to the those in the next layer, we must keep the most useful connections discarding the others. Past experience in computer vision suggests that simple shapes can be detected by combining the values of neighboring pixels.

Suppose, for instance, that you want to build a network for face recognition. To analyze faces you may want to detect salient facial features, such as the eyes, the nose, the mouth... These features occupy a limited portion of the image and you can look for them by considering a small group of pixels at a time.

Another important intuition is that the detection of the same simple shapes can be useful in every part of the input image. This suggests that it might be a good idea to repeat the same group of connections for all the input neurons. For instance, the same eye detector may be used to detect multiple eyes in different parts of an image.

Both intuitions can be implemented by replacing the linear operators in the multilayer perceptron with *convolutions*. The resulting neural networks are called *convolutional neural networks* and represent the state of the art for

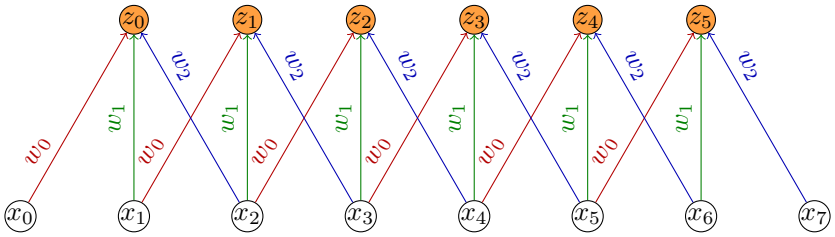


Figure 11.1: Convolution of a one-dimensional signal $\mathbf{x}$ with a filter $\mathbf{w}$ of size 3.

the recognition of images, and of other kind of signals.

11.1 Convolutions

The (discrete) convolution is an operator that transforms an input signal $\mathbf{x}$ into another signal $\mathbf{z}$. For now, let $\mathbf{x} = (x_0, x_1, \ldots, x_{n-1})$ be a one dimensional signal. Each output value z_u is obtained as a weighted sum of a group of k consecutive input values $x_u, x_{u+1}, \ldots, x_{u+k-1}$:

$$z_u = \sum_{s=0}^{k-1} w_s \cdot x_{u+s}, \tag{11.1}$$

where $\mathbf{w} = (w_0, w_1, \ldots, w_{k-1})$ is a vector of weights called *filter* or *kernel* (to not be confused with the kernel functions we encountered in past lectures). In the following we will denote convolutions with the $*$ symbol:

$$\mathbf{z} = \mathbf{x} * \mathbf{w}. \tag{11.2}$$

It easy to see that that convolution is a linear operator, which means that we have:

$$\begin{aligned} (\mathbf{x}_1 + \mathbf{x}_2) * \mathbf{w} &= \mathbf{x}_1 * \mathbf{w} + \mathbf{x}_2 * \mathbf{w}, \\ \mathbf{x} * (\mathbf{w}_1 + \mathbf{w}_2) &= \mathbf{x} * \mathbf{w}_1 + \mathbf{x} * \mathbf{w}_2, \\ \alpha(\mathbf{x} * \mathbf{w}) &= (\alpha\mathbf{x}) * \mathbf{w} = \mathbf{x} * (\alpha\mathbf{w}), \end{aligned} \tag{11.3}$$

for all inputs $\mathbf{x}$, $\mathbf{x}_1$, $\mathbf{x}_2$, all filters $\mathbf{w}$, $\mathbf{w}_1$, $\mathbf{w}_2$ and all scalars α.

Figure 11.1 graphically illustrates a convolution with a filter of size 3. The figure also makes evident three important properties of convolutions:

- each output value depends on a limited number of k neighboring input values (locality);
- the same k weights are used to compute all the output values in such a way that similar sequences in different parts of the input signals produces similar corresponding outputs (translation equivariance);

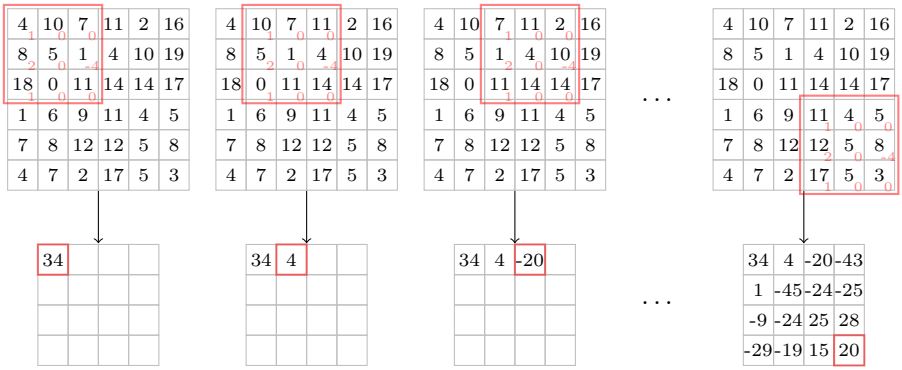


Figure 11.2: Two-dimensional convolution of a 6×6 image with the 3×3 filter (whose values are written in red) resulting in a 4×4 image.

- the number of parameters is independent on the size of the input signal.

All these seem useful properties that make convolutions better suited to deal with signals with respect to the linear operators in multilayer perceptrons.

Note that, according to the standard mathematical terminology, the operator defined in Equation (11.1) is called *cross correlation* that and convolutions are defined in a slightly different way, with the coefficients of the filter in reverse order. However, in machine learning the two operators are equivalent because the coefficients of the filters will be learned in the right order, independently on how the operator is defined. In machine learning the term convolution is more widely used than cross correlation.

11.2 Convolutions in two dimensions

Convolutions can be easily defined for two or more dimensions. In fact, two-dimensional convolutions are a very common operator in image processing and computer vision. Two-dimensional convolution is defined as follows:

$$z_{u,v} = \sum_{t=0}^{k-1} \sum_{s=0}^{k-1} w_{s,t} \cdot x_{u+s,v+t}, \tag{11.4}$$

where $\mathbf{x}$ is a real valued input image, $\mathbf{w}$ is a $k \times k$ filter and $\mathbf{z}$ is the result of the convolution. Note that non-square filters are possible but not very common. Two-dimensional convolutions can be computed as depicted in Figure 11.2: the coefficients of the filter are overlaid to the elements in the upper-left corner of the input. Each of these elements are multiplied by the corresponding coefficient, and the sum of the products is taken as the first output element. Then, the filter is moved one step to the right and the second

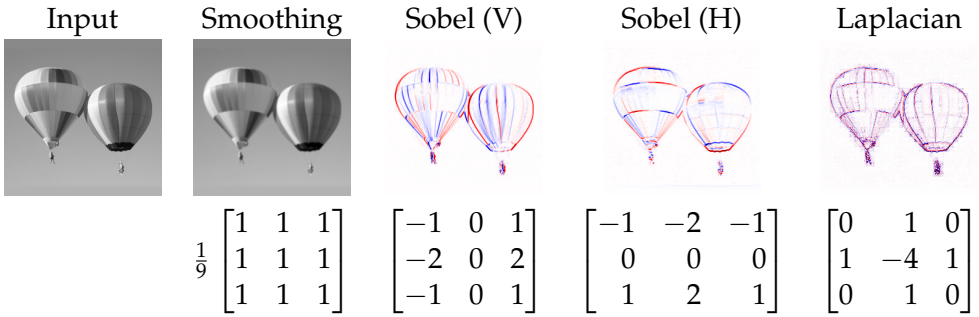


Figure 11.3: Examples of the results of convolutions of an image with some notable 3×3 filters. The last 3 have both positive (red) and negative (blue) values.

output element is computed. The process is repeated by placing the filter on the input image in each possible position until all the output elements have been computed.

What kind of transformations can be obtained through convolutions? In image processing convolutions are used to obtain a large variety of effects. Some of them are shown in Figure 11.3. When all the elements of the filter are positive the convolution removes some of the noise in the image and some of the details (we have a *smoothing* filter). Convolutions are also used to find the edges of the objects in the image; to do so the filter has to have positive values on one side and negative on the other, like for the two Sobel filters in the figure that detect horizontal and vertical filters. Similar filters can be defined to detect edges with different orientations. The last example is the Laplacian filter, which depends on the second derivatives of the image and produces both positive and negative values close to edges of any orientation.

Padding

One issue with convolutions as defined by Equation 11.4 is that the output is smaller than the input. In fact if the input has dimensions $h \times w$, a convolution with a $k \times k$ filter would result in an output of dimensions $(h-k+1) \times (w-k+1)$. A sequence of multiple convolutions, such as those we plan to use to build deep neural networks, would make the image disappear little by little. To contrast this phenomenon it is common to enlarge the input image by introducing some *padding*. Padding simply consists in adding a frame of additional elements (usually all set to zero) around the border of the image. There are three common strategies:

- in *valid convolutions* no padding is used and the output image has size $(h-k+1) \times (w-k+1)$;

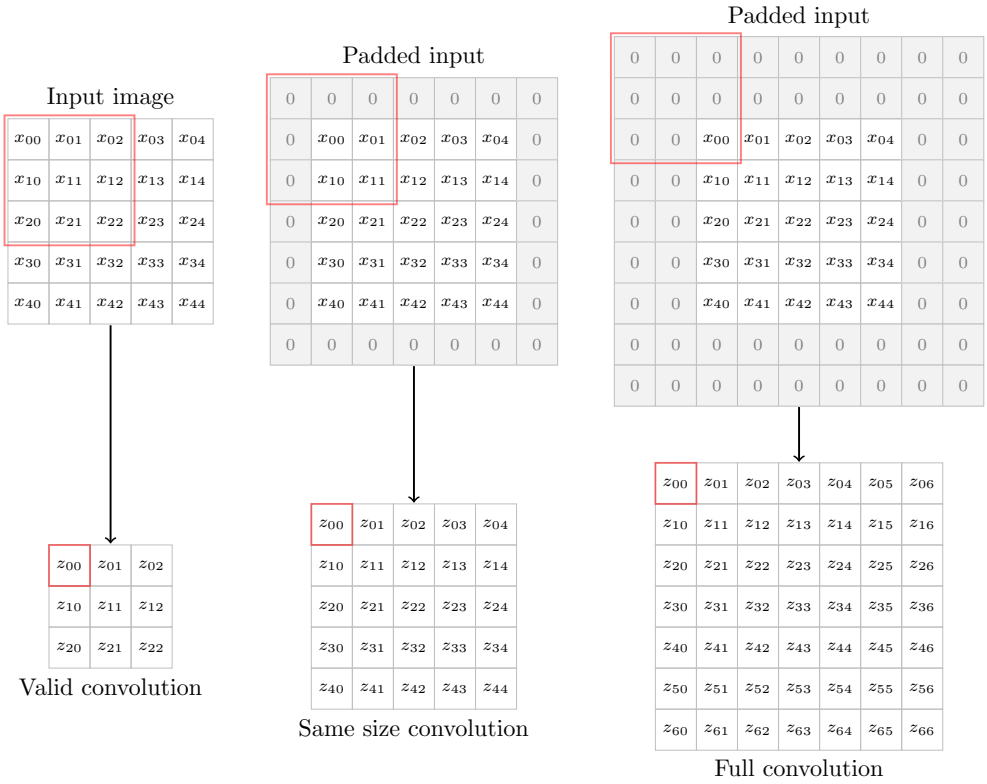


Figure 11.4: Three padding strategies for a 3×3 two-dimensional convolution of an input 6×6 image.

- in *same size convolutions* padding is added so that the size of input and output is the same. For odd values of k this corresponds to a padding of $(k-1)/2$ on each side;
- in *full convolutions* $k-1$ padding elements are added on each side and the output image grows to a size of $(h+k-1) \times (w+k-1)$;

Figure 11.4 illustrates the three strategies. In practice in valid convolutions we consider only positions in which the filter is completely inside the input image, while in full convolutions we consider positions in which there is any overlap. Same size are in between and are the most widely used in convolutional neural networks.

Multi-channel convolutions

So far we only considered two-dimensional signals of real-valued elements, such as gray-level images. There are many cases in which elements have multiple values. For instance, it is common to represent color images as 2D

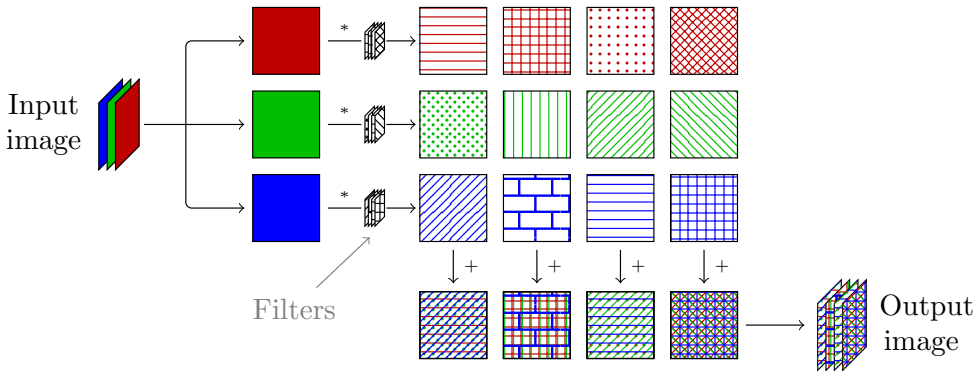


Figure 11.5: Convolution of a RGB image resulting in a four-channel image. The red, green and blue components are fed in four different single-channel convolutions each. Four channels are then formed by summing groups of three convolution results. They are finally stacked into a four-channel convolution.

signals in which pixels are triplets of values measuring the amount of red, green and blue (the so called RGB color space). We call each individual kind of information a *channel*, and we call the images with more than one channel *multi-channel* images. Color images, for instance, are multi-channel images with three channels.

Two-dimensional convolution can be easily extended to multiple channels as the sum of single-channel convolutions. Given a multi-channel image with n channels and n different filters the result of the multi-channel convolution is the convolution of the first channel by the first filter plus the convolution of the second channel by the second filter and so on and so forth. The result would be a single-channel image.

Multiple of these kind of convolutions can be combined to obtain a multi-channel to multi-channel convolution. Given a $h \times w \times n$ multi-channel image $\mathbf{x}$ and given a $k \times k \times p \times n$ multi-channel filter the result of the convolution $\mathbf{x} * \mathbf{w}$ is a $(h - k + 1) \times (w - k + 1) \times p$ multi-channel image $\mathbf{z}$ whose elements are computed as follows:

$$z_{u,v,d} = \sum_{c=0}^{n-1} \sum_{t=0}^{k-1} \sum_{s=0}^{k-1} w_{s,t,d,c} \cdot x_{u+s,v+t,c}, \tag{11.5}$$

assuming that it is a "valid" convolution, otherwise the the spatial dimensions need to be suitably adjusted.

An example of three-channels to four-channels convolution is depicted in Figure 11.5. The figure makes it clear how each output channel depends on all the input channels. The number of single-channel filters is given by the product between the number of input and output channels.

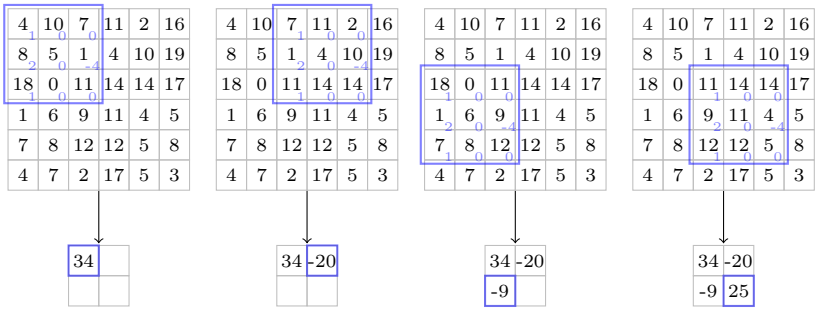


Figure 11.6: Single-channel strided convolution ($s = 2$) of a 6×6 images with the 3×3 filter (values are written in blue). The convolution is "valid" (i.e. no padding) and results in a 2×2 image.

Strided convolutions

Convolutions can be very time consuming, in particular when applied to large images. A simple way to reduce the computational burden consists in skipping some of the output pixels. In *strided convolutions* instead of moving the filter on the input by one pixel at a time, it is moved by s pixels. The number s is called *stride*. Most of the times the same stride is used for the horizontal and for the vertical directions. The result of using a stride $s > 1$ is that the number of output elements is reduced by a factor s^2. The formula for a multi-channel strided convolution is the following:

$$z_{u,v,d} = \sum_{c=0}^{n-1} \sum_{t=0}^{k-1} \sum_{r=0}^{k-1} w_{r,t,d,c} \cdot x_{s \cdot u+r, s \cdot v+t, c}, \tag{11.6}$$

with the only difference with respect to the standard convolution of the factors s in the indices of the input $\mathbf{x}$. Figure 11.6 shows an example of strided convolution.

11.3 Convolutional neural networks

Multi-channel convolutions are the main ingredient of Convolutional Neural Networks (CNN). This kind of neural networks represents the state-of-the-art solution to many computer vision problems. CNNs can be used to deal with any kind of signals, but most of the times are used to process images.

A CNN is a feed-forward neural network. Like multilayer perceptrons they are organized as a sequence of layers. However, between layers there are multi-channel convolutions instead of plain linear transformations. Since convolutions are linear operators, their combination is still a simple linear transformation. Activation functions follow each convolution to allow for a more complex behavior.

The activation of input neurons is taken directly from the input signal. For instance, in the case of a 100×100 color image we would have 30 000 input neurons. Given the activations of layer $l-1$, those of layer l are computed as follows:

$$z^{(l)}_{u,v,d} = b^{(l)}_d + \sum_{c=0}^{n-1} \sum_{t=0}^{k-1} \sum_{r=0}^{k-1} w^{(l)}_{r,t,d,c} \cdot x^{(l-1)}_{s\cdot u+r, s\cdot v+t, c}, \tag{11.7}$$

$$x^{(l)}_{u,v,d} = a(z^{(l)}_{u,v,d}), \tag{11.8}$$

Note that, differently from perceptrons, instead of having one bias for each neuron there is one bias for each channel. This way the number of parameters remains independent on the dimensions of the signal.

Equation (11.7) can also be expressed in vector form:

$$\mathbf{z}^{(l)} = \mathbf{x}^{(l-1)} * \mathbf{w}^{(l)} + \mathbf{b}^{(l)}, \tag{11.9}$$

where $*$ is used to denote the multi-channel convolution, and where the bias is intended to be added on a channel by channel basis.

Concerning the activation function, ReLU are the most widely used in convolutional neural networks, but others like the sigmoid and the hyperbolic tangent can be used as well.

Backpropagation

To train a convolutional neural network we can use the same backpropagation algorithm we defined for multilayer perceptrons. To do so we just have to find an expression representing the backpropagation through the convolution. For the sake of brevity, we will consider only the single-channel convolution in one dimension with a filter of size k:

$$z^{(l)}_u = \sum_{s=0}^{k-1} w^{(l)}_s \cdot x^{(l-1)}_{u+s}. \tag{11.10}$$

Suppose we already computed the derivatives of the loss function L with respect to all the logits $z^{(l)}_u$. Note that the activation $x^{(l-1)}_t$ is used to compute $z^{(l)}_{t-k+1}$, $z^{(l)}_{t-k+2}$, $\dots z^{(l)}_t$ by multiplying them by, respectively, $w^{(l)}_{k-1}$, $w^{(l)}_{k-2}$, $\dots$, $w^{(l)}_0$ (if this is not evident for you, you can see it more clearly in Figure 11.1). Therefore, we can use the chain rule of derivation as follows:

$$\frac{\partial L}{\partial x^{(l-1)}_t} = \sum_{r=0}^{k-1} \frac{\partial L}{\partial z^{(l)}_{(t-k+1)+r}} \cdot \frac{\partial z^{(l)}_{(t-k+1)+r}}{\partial x^{(l-1)}_t} = \sum_{r=0}^{k-1} \frac{\partial L}{\partial z^{(l)}_{(t-k+1)+r}} \cdot w^{(l)}_{k-1-r}, \tag{11.11}$$

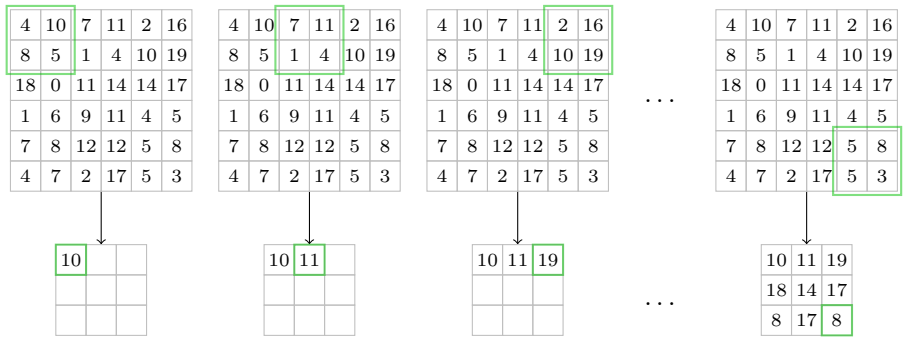


Figure 11.7: Two-dimensional 2×2 max pooling of a 6×6 image resulting in a 3×3 image.

which is just another convolution, but with the derivatives of $\mathbf{z}^{(l)}$ shifted by $k-1$ positions and with the weights in reverse order! Also some padding to the derivatives is needed to make everything works at the endpoints.

This result can be easily extended to multi-dimensional multi-channel convolutions, with different padding strategies and with strides. The fact that the derivatives of the convolution can be computed with another convolution makes backpropagation for convolutional networks very convenient, and even elegant.

Pooling

Convolutions in CNNs are used to identify interesting patterns in the signals. Channels in layers tend to specialize for the detection of a specific kind of patterns. These will be simple in the first layers and will become progressively more complex in later layers. Since the number of detectable patterns increases with the depth, also the number of channels is usually chosen to increase in the same way.

To compensate for the increase in the channels it is common to reduce the spatial dimensions. Strided convolutions can be used to obtain this effect in a very efficient way. An alternative is to include a *pooling* layer after all or some of the convolutions.

Pooling layers take groups of adjacent activations and summarize them with a single value. The most common kind of pooling is the *max pooling* which replaces the values with their maximum. For images pooling takes groups of $s \times s$ pixels, so the resulting number of elements is reduced by a factor of s^2. Pooling is applied independently across the channels. Figure 11.7 shows an example of 2×2 max pooling applied to a single-channel 6×6 image.

The intuition behind max-pooling is that, if convolutions learn to detect specific patterns, the pooling should preserve the maximum activation working as a sort of "OR logical gate". A high activation in a location after max pooling means that at least one of the corresponding input activations were high. So pooling preserves the meaning of the detection, but looses some precision in terms of spatial location (we don't know which input pixel has been chosen by the pooling operator).

An alternative to max pooling is *average pooling* which simply replaces groups of pixels with their spatial average.

Fully-connected layers

If we want to use a convolutional neural network for image classification, in the end we need to get to a single vector of k components, where k is the number of classes. In other words, at some point we must convert the spatial description of the content of the image, obtained by the convolutions with a global description that we can convert in a single vector of scores.

Many CNN after l convolutional layers take the $h \times w \times n$ multi-channel image $\mathbf{x}^{(l)}$ and turn it into a $h \cdot w \cdot n$-dimensional vector. The conversion requires no actual computation, and it is enough to rearrange all the values in a 1D vector. This operation is called *flattening* as it "flattens" the multi-channel image into a vector.

After flattening, the network will continue as a multilayer perceptrons with a few linear layers and non-linear activations. These linear layers in the context of a CNN are called *fully-connected layers* since, differently from convolutions, their neurons are connected to all the neurons in the previous layer.

Typically the last fully-connected layer yields the k-dimensional vector of scores that, after a final softmax, estimates the probabilities for the k classes.

11.4 Example: digit classification

The pvml library includes a class implementing a basic convolutional neural network. For the sake of brevity the code is omitted, but it is available online. It supports only multi-channel strided convolutions. ReLUs are used as activation functions. At the end a global average pooling is used to remove any residual spatial extent and softmax is applied to obtain class probabilities.

The training method uses stochastic gradient descent with momentum to minimize the average cross entropy. The following is a minimal script for training a CNN for handwritten digit recognition:

```
1 import pvml
2 import numpy as np
```

```
3
4  # Load the training set
5  Xtrain, Ytrain = pvml.load_dataset("mnist_train")
6  Xtrain = Xtrain.reshape(60000, 28, 28, 1) - 0.5
7
8  # Create and train the network
9  network = pvml.CNN([1, 12, 32, 48, 10], [7, 3, 3, 3], [2, 2, 1,
       1], [0, 0, 0, 0])
10 for epoch in range(20):
11     network.train(Xtrain, Ytrain, lr=1e-4, steps=600, batch=100)
12
13 # Load the test set
14 Xtest, Ytest = pvml.load_dataset("mnist_test")
15 Xtest = Xtest.reshape(10000, 28, 28, 1) - 0.5
16
17 # Evaluate the network
18 predictions, probs = network.inference(Xtest)
19 accuracy = (predictions == Ytest).mean()
20 print("Test␣accuracy:", accuracy * 100)
21 network.save("mnist_network.npz")
```

The network is created by passing four lists. The first list indicates the number of channels in each layer including the input layer (one in this case, since we are dealing with a gray-level image) and the output layer (where the channels corresponds to the number of classes). The second list indicates the size of the filter. In this case we have a 7×7 convolution followed by three 3×3 convolutions. The third list indicates the strides of the convolutions. The fourth list specifies the amount of padding to use for each convolution. For the network in the example the first convolutions have stride 2, while the other two are not strided ($s = 1$). It is common to design CNNs to increase the number of channels, while reducing the spatial dimensions. Here the strides and the lack of padding are used to reduce the data down to a 1×1 image with ten channels.

The train and the inference methods accept four dimensional arrays representing sets of images. For instance, the training set is represented by an array of dimensions $60\,000 \times 28 \times 28 \times 1$, since there are 60 000 training images of size 28×28. The last dimensions is one because these are gray-level images. Figure 11.8 shows how the network process the data.

Results

Training the network requires some time, but the results are quite impressive. The training curves are show in Figure 11.9. After 200 epochs the error on the test set is about 1.19% which is about 1% better with respect to the performance we obtained with the multilayer perceptron. Not only we almost halved the error rate, but we also saved on the number of parameters. Our

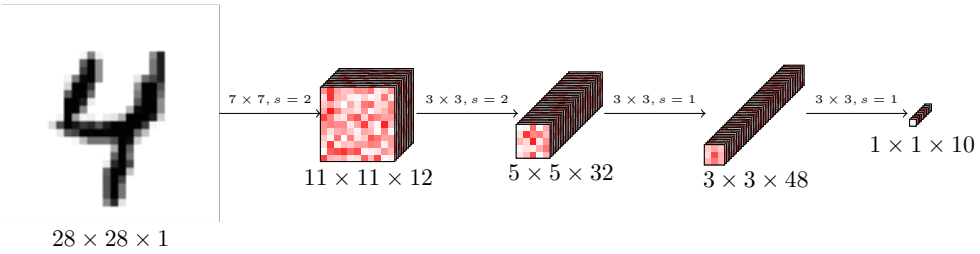


Figure 11.8: Representation of the processing of a CNN for digit recognition.

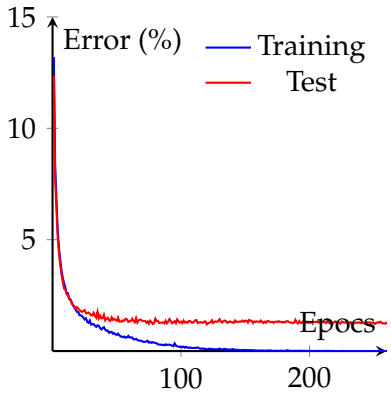


Figure 11.9: Error rates obtained by the CNN during training on the MNIST data set.

CNN, in fact, includes only $7 \times 7 \times 12 \times 1 + 3 \times 3 \times 32 \times 12 + 3 \times 3 \times 48 \times 32 + 3 \times 3 \times 48 \times 10 = 22\,290$ parameters which is a lot less than the 101 770 included in the MLP with a single hidden layer.

How does it work? To understand what the CNN learned we can look at the coefficients of the filters. Only the first convolutional layer can be easily interpreted: the filters computing the 12 channels in the first hidden layer are shown in Figure 11.10a. For some of them we can see positive coefficients on one side of the filters and negative coefficients on the opposite side. These filters implements edge detection at a given orientation.

We can understand the role of the channels in the hidden layers by looking at the images causing the highest activations. They are reported in Figure 11.11. Note how the 12 channels in the first layer react to simple shapes, such as horizontal (first channel) and vertical strokes (third channel). The channels in the second and third layer detect more complex shapes. In particular some of the channels in the third layer seem able to recognize a particular style of writing a particular digit.

The red boxes in the figure represent the part of the image causing the

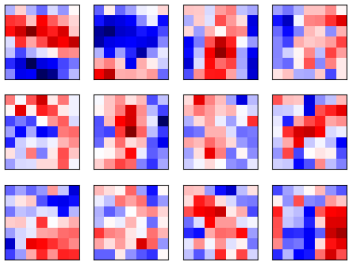


Figure 11.10: First layer of filters learned by the CNN. Positive values are painted in red, negative values in blue.

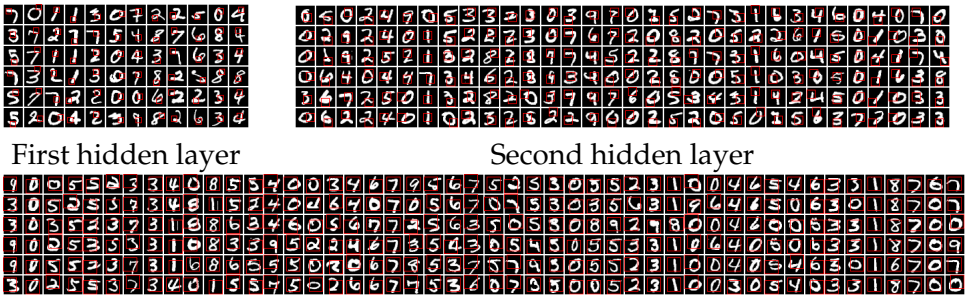


Figure 11.11: Top activations in the hidden layers of the CNN. Each column of the three images represents the top activations for the channels in the three hidden layers. The red boxes represent the part of the input image causing the high activation value.

activation. The input pixels on which a neuron depends is called the *receptive field* of the neuron. Note the receptive field grows with the depth of a layer. Ideally, the receptive field of the output layer should be the entire image (not represented in the figure).

11.5 Summary

In this lecture we introduced Convolutional Neural Networks (CNN) as a powerful architecture for the processing of signals. We discussed their main components:

- we defined the convolutional operator in one and two dimensions;
- we showed how convolutions can be extended to work with multiple channels;
- we introduced padding and strided convolutions;

- we also showed the other components typically used in CNNs, namely pooling and fully-connected layers.

Finally we shown how convolutional neural networks can be used for image recognition, the task for which they have been originally designed. We observed how CNN can easily outperform MLP both in terms of accuracy and in the number of parameters.

Lecture 12

CNN architectures

Convolutional neural networks (CNN) made it possible to successfully address problems in computer vision that were previously considered out of our reach. Nowadays, image recognition is almost considered as a "solved" problem. Even though current image recognition systems are still not able to display the richness with which humans understand what they see, they can achieve superhuman performance in many recognition tasks.

In order to improve the accuracy of their CNNs researchers developed techniques that proved to be useful also for other kind of neural networks. In this lecture we will review some of the most influential architectures that have used to train successful neural networks.

12.1 LeNet

LeNet is one of the earliest convolutional neural network ever designed. It is considered one of the most important elements that suggested that deep learning models were capable of learning to solve complex problems.

LeNet was designed in 1988 for handwritten digit recognition.[1] Its structure is depicted in Figure 12.1. LeNet paved the way for modern CNNs and its architecture presented most of the elements we find in modern architectures. The network includes three 5×5 convolutions followed by 2×2 average pooling and hyperbolic tangent as activation function. After the third convolution the 32×32 input gray-level image is reduced to a single pixel with 120 channels are processed by two fully-connected layers resulting in 10 ouput neurons. The fifth version of the network (LeNet5) in 1998 obtained a test error rate of 0.95% in classifying MNIST data.[2]

[1] Yann LeCun *et al.* "Backpropagation Applied to Handwritten Zip Code Recognition," Neural Computation. 1 (4): 541–551.

[2] Yann LeCun et al. "Gradient-Based Learning Applied to Document Recognition," Proceedings of the IEEE, 86(11):2278–2324, 1998.

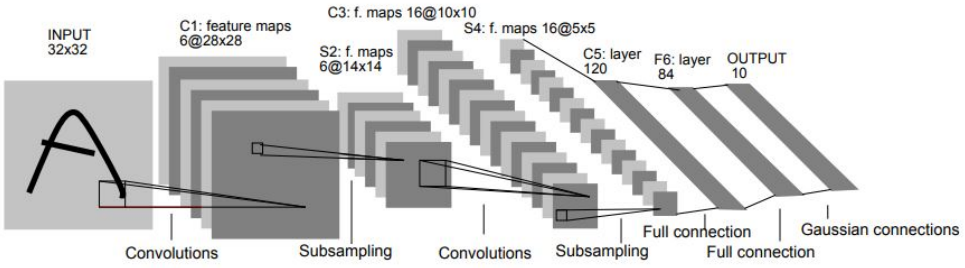


Figure 12.1: Architecture of the LeNet convolutional neural network for handwritten digits recognition.

12.2 AlexNet

The presentation of AlexNet in 2012 has been one of the most significant factors that determined the popularity of deep learning. At that time, in fact, general image recognition was considered a problem well beyond the capabilities of machine learning models. Researchers in computer vision were mostly engaged in designing robust features. SVMs were the *de-facto* standard classification model and neural networks were not considered. The network is named after its designer Alex Krizhevsky[3] from the University of Toronto.

The win of AlexNet, by a large margin, in the ImageNET Large Scale Image Recognition Challenge (ILSVRC) demonstrated that CNNs were not only feasible, but also more than promising. Given as input a 227×227 color image, the network computes a vector of probability estimates for the 1000 classes of the contest.

For that time it was a huge model with its $\sim$62 millions parameters, divided in five convolutional layers and three fully-connected layers. Training in reasonable time (a few days) was made possible by the use of two Graphical Processing Units (GPU).

The structure of AlexNet is depicted in Figure 12.2. The network included:

- a first convolution with a large 11×11 filter size with a stride of four;
- a 5×5 convolution without stride;
- other three 3×3 convolutions without stride;

[3] Alex Krizhevsky *et al.* "Imagenet classification with deep convolutional neural networks," Advances in Neural Information Processing systems, 2012.

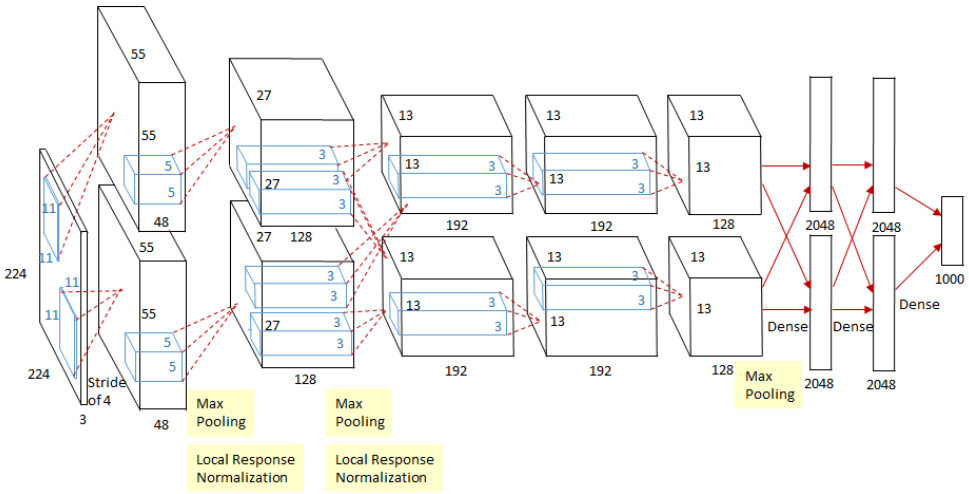


Figure 12.2: Architecture of AlexNet. It is divided in two parallel networks so that it can be executed on two GPUs.

- the resulting 6×6 image with 256 channels is flattened into a vector of 9216 components and then processed by three fully-connected layers of size 4096, 4096 and 1000;

- a final softmax computes the probability estimates.

Convolutions are followed by max-pooling layers and by local response normalization (a form of normalization in which the activation of a neuron inhibits that of its neighbors). Key components in the success of AlexNet were the use of ReLU activation functions (which greatly facilitate the training process) and a technique, called *dropout*, that increases the generalization capability of the network.

Dropout

Dropout is a simple but effective technique for the improvement of generalization in neural networks. It consists in randomly setting to zero the neurons with probability p (usually $p = 0.5$), and by multiplying the others by $1/(1-p)$ (so that the average value does not change). Dropout is typically used in the last hidden layers and makes it so the following layers learn to depend on multiple neurons and not just a few of them. With dropout the computation of important features is replicated by different neurons (perhaps with some variation), making them more robust to certain kind of distortions. Dropout is applied only during training and after that it is disabled.

12.3 VGG Networks

AlexNet represented a huge step forward in building automatic image recognition systems. However, some of the details of its architecture seemed rather *ad hoc* and not completely motivated. Moreover, the design of the network was limited by the specific hardware used for its training.

After the success of AlexNet, the Visual Geometry Group at the University of Oxford worked on a refinement of the structure of AlexNet by removing all unnecessary elements and by tuning the overall configuration to make it more accurate. They found out that:

- 3×3 convolution tend to be more effective than convolutions with larger filters. For instance, a sequence of two 3×3 convolutions obtains a similar effect of a single 5×5 convolution, but with less parameters (18 vs. 25 for a single channel), and even with an increase in expressivity if we put a ReLU between the two.
- local response normalization was not really effective;
- the deeper the network, the more accurate is the result (with limitations due to the computational cost).

The result was a set of networks of various depth (up to 19 layers) that made it possible to achieve a top-5 error rate of about 25% in the 2014 edition of the ILSVRC[4]. Among these, the most widely used was the variant with 16 layers (VGG-16) depicted in Figure 12.3. It includes a sequence of 13 convolutions with 3×3 filters and ReLU activation functions, sometimes followed by a $2\times$ max pooling. Then the networks continues as AlexNet with three fully-connected layers (4096, 4096 and 1000 neurons) followed by a final softmax. This network included more than 138 millions of parameters.

12.4 Inception

In those years many research groups were working on CNN architectures for image recognition. The winner of the 2014 ILSVRC competition was a network, called *inception*, designed by a team from Google[5]. The network is also called *GoogLeNet* as an homage to the LeNet, and through the years has been updated many times.

The first version was made as the combination of "inception modules". In one inception module different convolutions are applied in parallel to the

[4]Karen Simonyan and Andrew Zisserman, "Very deep convolutional networks for large-scale image recognition," 2014.

[5]Christian Szegedy *et al.* "Going deeper with convolutions," 2014.

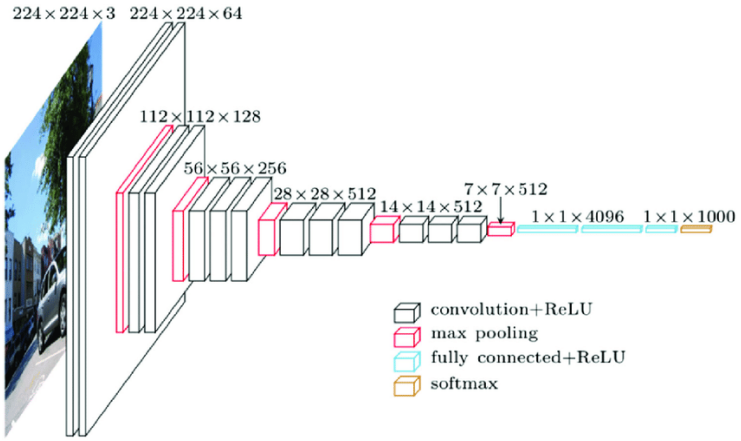


Figure 12.3: Architecture of the VGG-16 network. All convolutions were 3×3.

same input, and the results are then combined to form the input of the next module. More in detail, in each inception module there are four paths:

- one with just a 1×1 convolution. This processes one pixel at a time without any spatial transformation;
- one with a 3×3 convolution, preceded a 1×1 convolution that reduces the number of input channels, reducing also the combined number of parameters;
- one with a 5×5 convolution, preceded a 1×1 convolution (in later versions the 5×5 convolution has been replaced by a pair of 3×3 convolutions);
- one with a max pooling followed by a 1×1 convolution.

The rationale behind the inception module is that different kind of features may require different kind of processing. Instead of using a general purpose 5×5 convolution, the different paths can specialize in computing different kind of features. The result is that the total number of parameters is greatly reduced. As a result, the total number of parameters was relatively small (about 13 millions) despite the depth of the network (22 layers).

Another distinctive element of inception is the presence of three separate output layer. Two of these have the goal of facilitating the training of the first layers by complementing the gradients of the "true" output layer that risk to vanish because of the depth of the architecture. Figure 12.4 depicts the architecture of the network.

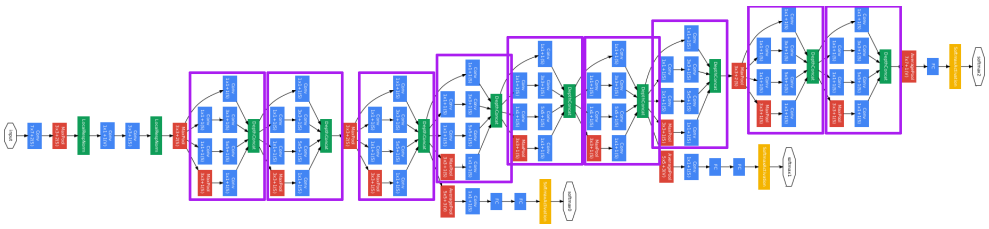


Figure 12.4: The first version of the inception network (GoogLeNet). The purple boxes highlight the inception modules. Note the three output layers attached at different depth levels.

12.5 ResNet

The results in image recognition made it clear that the depth of an architecture was a key element in determining its accuracy. One issue with very deep network is that they are hard to train. At some point we may expect that deep networks start to overfit. In practice, numerical issues posed more serious limitations. In particular, the gradient of the loss must reach the first convolutions with enough strength to make learning possible.

Suppose we have a network of depth D with a given training error. What we may expect from a network of depth $D+1$? The training error should decrease, or at least stay the same. In fact, in the worst case we expect that the additional layer learns to do nothing. In practice, after a given depth it is possible to measure an increase in the training error due to the difficulties of training.

The issue is that learning to do nothing is not as easy as we may expect. *Residual networks* are based on this intuition. They include *residual blocks* which combines a pair of convolutions as following:

$$\begin{aligned} \mathbf{x}^{(l+1)} &= \mathrm{ReLU}(\mathbf{x}^{(l)} * \mathbf{w}^{(l+1)} + \mathbf{b}^{(l+1)}), \\ \mathbf{x}^{(l+2)} &= \mathrm{ReLU}(\mathbf{x}^{(l+1)} * \mathbf{w}^{(l+2)} + \mathbf{b}^{(l+2)} + \mathbf{x}^{(l)}), \end{aligned} \tag{12.1}$$

with just a small difference (in red) with respect to how convolutions were normally used in CNNs. In the second step the unprocessed $\mathbf{x}^{(l)}$ is added before computing $\mathbf{x}^{(l+2)}$. If all the parameters are initialized with small values, then we have $\mathbf{x}^{(l+2)} \approx \mathbf{x}^{(l)}$, and the block is close to the identity function.

By concatenating residual blocks it is possible to build very deep networks that learns a lot more effectively than the previous architectures. This idea has been used to build the family of *ResNet* convolutional networks[6]. The version with 152 layers won the 2015 edition of the ILSVRC challenge with 3.57% of top-5 error rate. For the first time superhuman performance

[6] Kaiming He *et al.* "Deep Residual Learning for Image Recognition," 2015.

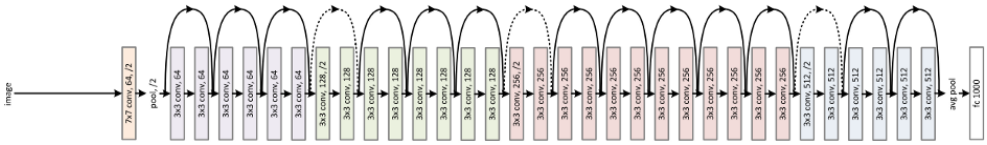


Figure 12.5: Architecture of ResNet-34.

were obtained for a complex image recognition task (humans exhibit about 5% of top-5 error on that data set). Figure 12.5 shows the architecture of the version of ResNet with 34 layers. Residual block are now widely used in many deep networks, even those that are not convolutional.

Batch normalization

In the same period a technique has been developed to speed up the training of deep neural networks. We know that mean-var scaling facilitate training by normalizing input data in a convenient way. In neural networks mean-var scaling is less effective because it only impacts the training of the parameters in the first layer. Even if we apply mean-var scaling, the input to all the other layers will not be normalized. We cannot simply apply mean-var scaling to the other layers, because the statistics of their inputs change during training.

Batch normalization assumes that the training algorithm is stochastic gradient descent with mini-batches (or some other similar algorithm). Mean-var scaling is applied before any layer (except the first one), by using mean and variance estimated on a single mini-batch of data, instead of using the whole training set. These are very noisy estimates, but empirically, they tend to be good enough to serve their purpose.

Training speed increases dramatically with batch normalization. After training, batch normalization is replaced by a true mean-var scaling with statistics computed on the whole training set. Batch normalization has been used with great success in ResNet and it is used today to train most deep neural networks.

12.6 Comparison

Figure 12.6 summarizes some information about various CNN architectures.

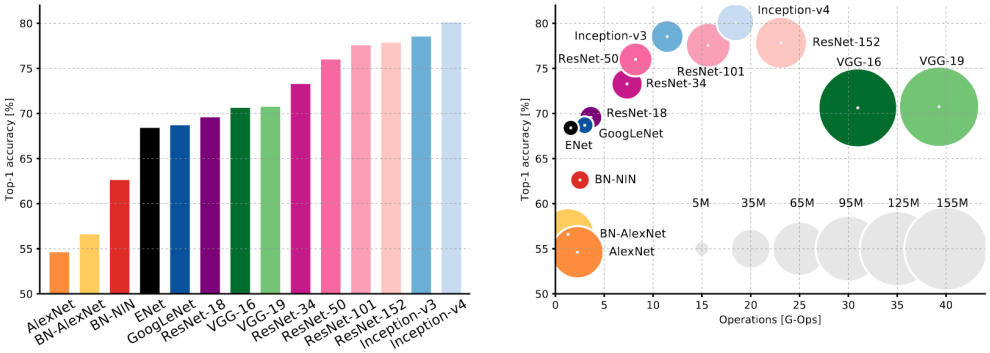


Figure 12.6: Comparison of various CNN architectures. ENet, NIN and recent versions of Inception were not discussed. BN-AlexNet refers to AlexNet with added Batch Normalization. The area of the discs is proportional to the number of parameters. Plots taken from Canziani *et al.* "An Analysis of Deep Neural Network Models for Practical Applications," 2017.

12.7 Other techniques

The design and training of Convolutional Neural Networks has been enriched by many techniques, methods and common practices. Here we will briefly review the most widely used.

Data augmentation

A good image recognition system is expected to be tolerant to many image distortions, including geometrical (rotation, shear...) and photometric (color, contrast...) transformations. A simple way to train a robust system is to apply random transformations to the training data. This can be done in two main ways: (i) before training by including in the training set multiple variations of each image; (ii) during training by applying a different random transformation when forming the mini-batch for each iteration of stochastic gradient descent.

The set of transformations considered depends on the specific application. The most commonly used are:

- random crop of a portion of the image;
- rotation (usually by a small angle);
- mirroring around the vertical axis;
- change in brightness;
- change in contrast;

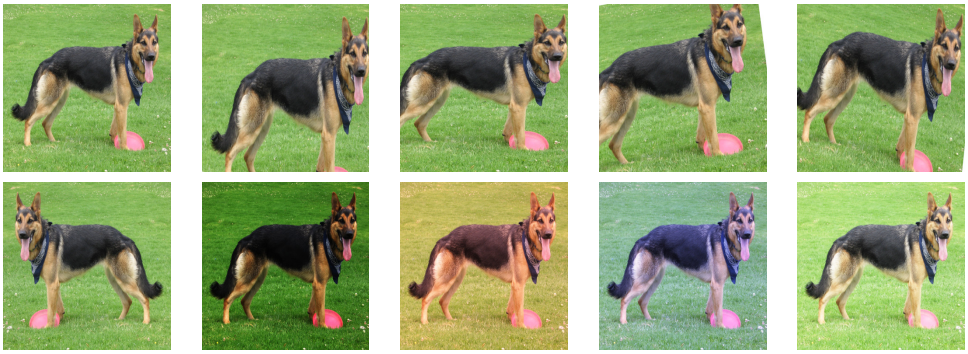


Figure 12.7: Examples of data augmentation applied to an input image. The original image is at the top left of the figure.

- small adjustment of colors;
- addition of some noise;
- ...

the important part is that these transformations should keep the subject of the image clearly recognizable. Figure 12.7 shows an example of data augmentation applied to an image. Note that after augmentation the images are scaled to a common size, which depends on the specific network architecture. Data augmentation is used only during training.

Transfer learning

In many case we cannot afford to train a deep neural network because we don't have enough data. AlexNet, VGG, Inception and ResNet all require hundreds of thousands or, even better, millions of labeled images for their training. When training data is scarce it is often possible to use these network by first training them for a different problem, and then to adapt them for the new task. In machine learning this approach is called *transfer learning*.

The most common way of doing transfer learning with CNNs is to pick an already trained network (often on ImageNet data), to remove the last fully connected layer, and to replace it with a new final layer having a number of output neurons equal to the number of classes of the new problem. Then the weights in the last layer are trained on a new training set, while the parameters in all the other layers are frozen. This strategy is depicted in Figure 12.8.

This approach works extremely well since, when the CNN is trained on a large and diverse training set, its first part learns to recognize patterns that

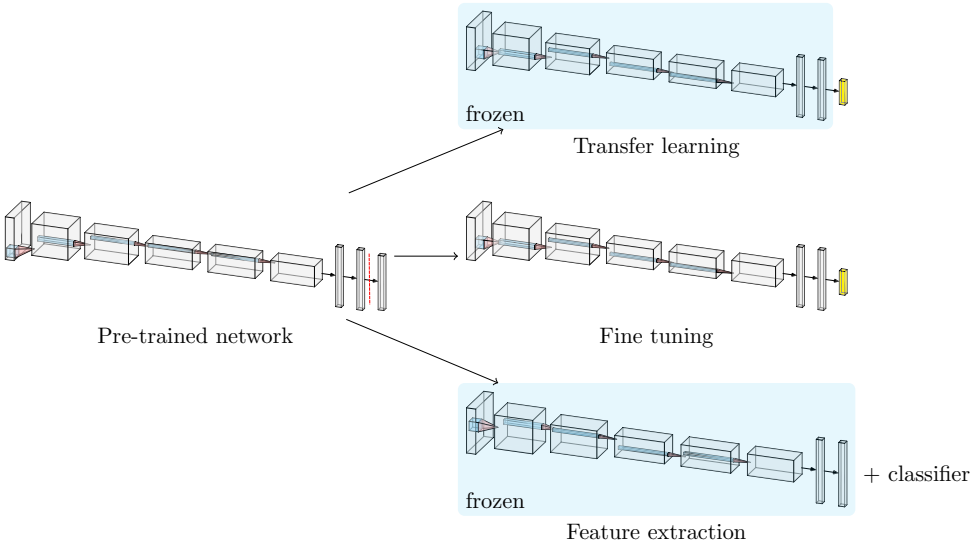


Figure 12.8: Illustration of transfer learning for a convolutional neural network.

are generally useful for most image recognition problems. Only the last layer is specific for the set of classes in the training set.

With this approach the original network is used as a feature extractor, and it has been empirically demonstrated that the last hidden layer is the best one for this purpose. A slightly different strategy consists in computing the features for the images of the new task, and to train with them a simple classifier, such as a linear SVM.

A further alternative consists, after the training of the new output layer, in continuing the training of the whole network with a small learning rate. This approach is called *fine tuning*.

12.8 Summary

In this lecture we revised some of the convolutional neural networks that made popular this kind of neural architecture. In particular we presented:

- LeNet, one of the earliest successful attempts;
- AlexNet, the one that popularized CNNs;
- VGG, which simplified the approach by removing the least useful components;
- Inception, which demonstrated how very good accuracy can be obtained with a limited number of parameters;

- ResNet, which showed how to effectively design very deep networks.

In this lecture we also presented several tricks and techniques that are used in this area, including dropout and batch normalization layers, data augmentation and transfer learning.

Lecture 13

Recurrent Neural Networks

In feed forward networks, like multilayer perceptrons and convolutional networks, the activation of each neuron is computed only once. This limits the behavior of the models, which are forced to execute the same sequence of operations for every input. Processing data of variable size with these models is very difficult, if not impossible.

Recurrent Neural Networks (RNN) are a class of networks especially designed to deal with variable-length sequences. They are "recurrent" because they have a backward connection that makes them reprocess as additional input their previous activations. This mechanism forms a loop in the layout of the neurons, that creates a sort of memory of the information processed for previous inputs. Because of the loop, these models do not fall in the category of feed forward networks.

RNNs process sequential data $\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_{T_x-1}$, where for each time step t there is an input vector $\mathbf{x}_t \in \mathbb{R}^{n_x}$. We use the word "time" to indicate the dimension along the sequence, even though it does not necessarily correspond to the physical concept of time (for instance we may have a sequence of elements varying in a linear space). The length T_x of the sequence is variable (different sequences may have different length), but the size n_x of the input vectors is fixed.

Depending on the problem, the output of the RNN can be a sequence of output vectors $\hat{\mathbf{y}}_0, \hat{\mathbf{y}}_1, \ldots, \hat{\mathbf{y}}_{T_y-1}$ (where it can be either $T_x = T_y$ or $T_x \neq T_y$), or a single vector, or both.

With RNNs we can address a large variety of problems. They are particularly popular in natural language processing, speech recognition, analysis of time series etc. The following is just a short list of examples:

- document classification: the input is a sequence of characters or a sequence of words, the output is a class label;

- language translation: input and output are sequences of words in different languages;
- music classification: the input is a sequence of notes, the output is a label indicating the genre of the composition;
- trigger word detection: the input is a sequence of sound features, the output is a sequence of labels $0/1$ indicating whether a given word has been pronounced or not;
- speech recognition: the input is a sequence of sound features, the output is a sequence of characters or words;
- image captioning: the input is an image (i.e. a single element) the output is a sequence of words;
- activity recognition: the input is a sequence of images the output is a label.

13.1 Basic RNN model

The RNN model is invoked multiple times, each time by feeding one of the element of the input sequence. At time t the vector $\mathbf{x}_t \in \mathbb{R}^n$ is taken as input, and the network outputs the vector $\hat{\mathbf{y}}_t \in \mathbb{R}^{n_y}$. The network also keeps a *state* vector $\mathbf{h}_t \in \mathbb{R}^{n_h}$, made of the activations of a set of n_h hidden neurons encoding the memory of the computations done in all the previous steps.

This idea can be implemented in different ways. In the basic RNN model the state vector $\mathbf{h}_t$ is obtained by combining the input $\mathbf{x}_t$ and the state at the previous step $\mathbf{h}_{t-1}$. This is done through two sets of weights: W_h is a $n_h \times n_x$ matrix representing the weights of the connections between input and hidden neurons, while U_h is a $n_h \times n_h$ matrix representing the connections internal to the hidden layer. The update rule for the hidden neurons is the following:

$$\mathbf{h}_t = \mathbf{a}_h(W_h\mathbf{x}_t + U_h\mathbf{h}_{t-1} + \mathbf{b}_h), \tag{13.1}$$

where $\mathbf{b}_h$ is a n_h-dimensional vector of biases, and $\mathbf{a}_h(\cdot)$ is an activation function (hyperbolic tangent and ReLU are common choices in this case). At the beginning the state is usually set to zero ($\mathbf{h}_{-1} = \mathbf{0}$).

The hidden state is also used to compute the output $\hat{\mathbf{y}}_t$:

$$\hat{\mathbf{y}}_t = \mathbf{a}_y(W_y\mathbf{h}_t + \mathbf{b}_y), \tag{13.2}$$

where W_y is a $n_y \times n_h$ matrix of weights connecting the hidden to the output layer, $\mathbf{b}_y$ is a vector of biases and $\mathbf{a}_y(\cdot)$ is the activation function. For instance,

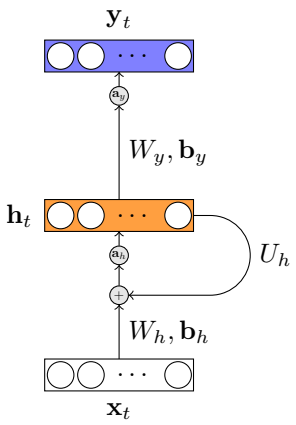


Figure 13.1: Graphical representation of a basic RNN model. All the connections between layers are fully-connected.

if we need a class label as output, we can use the softmax operator. Note that here we have one output vector for each input, therefore $T_x = T_y = T$

Figure 13.1 shows a graphical representation of a basic RNN model.

Summing up, the forward pass of a basic RNN is the following:

- set a null initial state $\mathbf{h}_{-1} \leftarrow \mathbf{0}$;
- for $t \in \{0, 1, \ldots, T-1\}$ do:
  - update the new state $\mathbf{h}_t \leftarrow \mathbf{a}_h(U_h\mathbf{h}_{t-1} + W_h\mathbf{x}_t + \mathbf{b}_h)$;
  - compute the output $\hat{\mathbf{y}}_t \leftarrow \mathbf{a}_y(W_y\mathbf{h}_t + \mathbf{b}_y)$.

Training a RNN

The training of a RNN is conceptually very similar to that of a feed forward network. Given a training set of input and output sequences we optimize the parameters W_h, U_h, $\mathbf{b}_h$, W_y, $\mathbf{b}_y$ to minimize a loss function L. To do this we can use stochastic gradient descent or similar algorithms.

The loss depends on the specific problem. For instance, if the expected output $y_0, \ldots, y_{T-1}$ is a sequence of class labels, then the actual output will be a sequence $\hat{\mathbf{y}}_0, \ldots, \hat{\mathbf{y}}_{T-1}$ of vectors of probability estimates, and we can use the average cross entropy as loss:

$$L = \frac{1}{T} \sum_{t=0}^{T-1} \sum_{j=0}^{n_y-1} -\bar{y}_{tj} \log \hat{y}_{tj}, \tag{13.3}$$

where $\bar{\mathbf{y}}_t$ is the one hot vector for the label y_t. Note that this expression represents the loss for a single training sample.

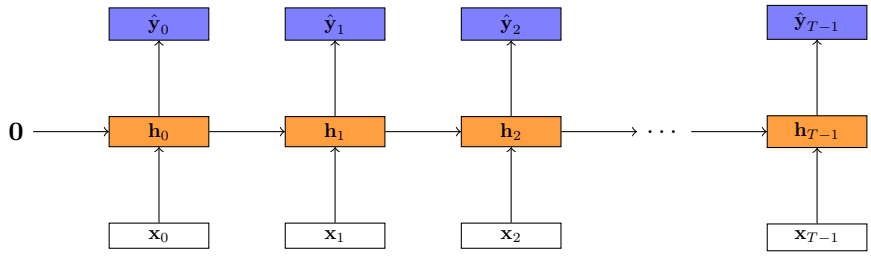


Figure 13.2: Unrolled representation of a RNN. The network can be seen as a sequence of identical modules that share the same parameters.

If the expected output is a single class label y (i.e. we have a standard classification problem), then we will compare it against the last output vector $\hat{\mathbf{y}}_{T-1}$:

$$L = \sum_{j=0}^{n_y-1} -\bar{y}_j \log(\hat{y}_{T-1,j}). \tag{13.4}$$

The backpropagation algorithm can be applied to RNNs to compute the derivatives that are needed by stochastic gradient descent to update the parameters. To see how it works in this case, it is better to "unroll" the RNN, as depicted in Figure 13.2. This way the network can be seen as a sequence of identical modules. In fact the resulting network looks similar to a multilayer perceptron with multiple inputs and outputs connected to the hidden layers. The main difference is that the layers here share the parameters, since weights and biases do not depend on the step t.

The exact math of backpropagation is a bit different from that used in multilayer perceptrons, but the basic idea is the same. The main difference is that the gradient of the loss with respect to, say, U_h depends on the derivatives with respect to all the $\mathbf{h}_t$.

For RNN the backpropagation starts from the last time steps and proceed back to $t = 0$. For this reason it is called *Back-Propagation Through Time* (BPTT).

RNN Backpropagation (optional)

The derivation of backpropagation equations for RNN models is similar to that for multiplayer perceptrons. Starting from the loss function derivatives are computed by following the operations involved in the network, but in the opposite order with respect to the forward pass.

It is useful to rewrite the forward pass step by step, making explicit the

computation of logits:

$$\mathbf{z}_t = U_h \mathbf{h}_{t-1} + W_h \mathbf{x}_t + \mathbf{b}_h, \tag{13.5}$$
$$\mathbf{h}_t = \mathbf{a}_h(\mathbf{z}_t), \tag{13.6}$$
$$\mathbf{u}_t = W_y \mathbf{h}_t + \mathbf{b}_y, \tag{13.7}$$
$$\hat{\mathbf{y}}_t = \mathbf{a}_y(\mathbf{u}_t). \tag{13.8}$$

We will consider the case of a element-by-element classification, so the activation $\mathbf{a}_y$ is the softmax operator, and the loss function is the average cross entropy between the estimates $\hat{\mathbf{y}}_t$ and the target labels, encoded as the on-hot vectors $\bar{\mathbf{y}}_t$:

$$L = \frac{1}{T} \sum_{t=0}^{T-1} \sum_{j=0}^{n_y-1} -\bar{y}_{tj} \log \hat{y}_{tj}. \tag{13.9}$$

Let δ_t^u be the gradient of L with respect to $\mathbf{u}_t$, and let δ_t^h and δ_t^z those with respect to $\mathbf{h}_t$ and $\mathbf{z}_t$. Starting from the end, we know from multinomial logistic regression that the derivative of the combination of a softmax and the cross entropy has a simple form:

$$\delta_t^u = \frac{1}{T}(\hat{\mathbf{y}}_t - \bar{\mathbf{y}}_t). \tag{13.10}$$

Note that $\mathbf{u}_t$ and $\mathbf{y}_t$ influence only the component of the loss for step t.

The hidden state $\mathbf{h}_t$, instead, impacts the loss in two ways: through the output $\hat{\mathbf{y}}_t$ and through the input to the next state $\mathbf{z}_{t+1}$. Therefore, its gradient will be the sum of two terms:

$$\delta_t^h = \mathcal{J}(\mathbf{u}_t, \mathbf{h}_t)\delta_t^u + \mathcal{J}(\mathbf{z}_{t+1}, \mathbf{h}_t)\delta_{t+1}^z = W_y^\intercal \delta_t^u + U_h^\intercal \delta_{t+1}^z, \tag{13.11}$$

where $\mathcal{J}(\cdot, \cdot)$ is the Jacobian matrix (which in both cases is just the transpose of the matrix of weights).

Finally, the gradient of the input $\mathbf{z}_t$ to the hidden state is simply obtained with the chain rule of derivation:

$$\delta_t^z = \mathbf{a}_h'(\mathbf{z}_t) \odot \delta_t^h, \tag{13.12}$$

where $\mathbf{a}_h'$ is the element-by-element derivative of the activation function and $\odot$ is the elmentwise product (this step can be found identical in the back-propagation for the multilayer perceptron). For the last step δ_T^z is set to zero.

At this point we can use the derivatives to compute the gradient for all the parameters in the RNN. To do so, we have to consider that weights and

biases are the same for all the time steps. The result is the following:

$$\nabla_{U_h} L = \sum_{t=0}^{T-1} \delta_{t+1}^z \cdot \mathbf{h}_t^\intercal, \qquad \nabla_{\mathbf{b}_h} L = \sum_{t=0}^{T-1} \delta_t^z, \tag{13.13}$$

$$\nabla_{W_y} L = \sum_{t=0}^{T-1} \delta_t^u \cdot \mathbf{h}_t^\intercal, \qquad \nabla_{\mathbf{b}_y} L = \sum_{t=0}^{T-1} \delta_t^u, \tag{13.14}$$

$$\nabla_{W_h} L = \sum_{t=0}^{T-1} \delta_t^z \cdot \mathbf{x}_t^\intercal. \tag{13.15}$$

Finally, the gradients can be used in stochastic gradient descent or other similar learning algorithms.

13.2 Language models

Language modeling is an interesting application of recurrent networks. In language modeling we want to build an estimator of the probability that a given sequence of symbols is a sentence of a given language. The model can be used in various ways. For instance, it can be used to correct spelling errors, or it can be used to propose to a user how to continue the message he or she is writing.

A successful use of language models is as auxiliary components for other models that produce text, such as in speech recognition, optical character recognition, image captioning... In these cases, language models can dramatically improve the quality of the output by removing ambiguities and by increasing the chance that the generated text is meaningful.

A language model estimates the probability

$$P(x_0, x_1, \ldots, x_{T-1}), \tag{13.16}$$

that if we pick a random sequence from some target language we draw exactly the sequence $x_0, x_1, \ldots, x_{T-1}$. In practice we expect that the model assigns large probabilities to common sequences, small probabilities to uncommon sequences, and zero to sequences that are not part of the language. For instance, for the English language we expect

$$P(\texttt{`the cat is on the roof'}) > P(\texttt{`the roof is on the cat'}) > P(\texttt{`on is roof the the cat'}) > P(\texttt{`si cta het on rofo eht'}).$$

The elements of the sequence are the particles forming the sentences of the language. For natural languages there are two common choices: each x_t is one character in an alphabet (n_x would be the size of the alphabet), or each x_t is a word in a vocabulary (n_x is the size of the vocabulary). Intermediate

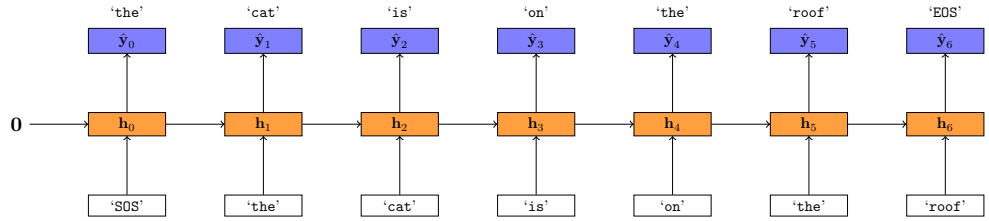


Figure 13.3: RNN for language modeling. During training the input words, or better their one-hot vector representations, are fed as input and the desired output is the next word in the sequence.

solutions are also possible, where each word is divided in a small number of *tokens*. Language models can be built also for other kind of languages, such as music, programming languages etc.

How can we build a RNN for language modeling? The basic idea is to use it to estimate the *conditional* probability distribution for the $(t+1)$-th element in the sequence, given all the previous ones:

$$P(x_{t+1}|x_0, x_1, \ldots, x_t). \tag{13.17}$$

In practice the model has to learn to predict the next element in the sequence, therefore the desired output is $y_t = x_{t+1}$, as depicted in Figure 13.3. More precisely, at step t the input to the network will be the one-hot vector encoding of x_t (assuming that we have a suitable vocabulary enumerating all possible words), and the output will be the vector $\hat{\mathbf{y}}_t$ assigning probability estimates to the possible continuations of the sequence. To signal the start and the end of the sequence it is convenient to add the special tokens `SOS` (Start Of Sequence) and `EOS` (End Of Sequence) as delimiters. Beside one-hot vectors, more sophisticated numerical encodings of the inputs exist.

In language modeling input and output sequences have the same length ($T_x = T_y = T$). Since the goal at each step is to guess a symbol, it is a classification problem, and the loss function is the average cross entropy.

Once the model has been trained it can be used to estimate the probability of sequences as follows:

$$P(x_0, x_1, \ldots, x_{T-1}) = P(x_0)P(x_1|x_0)p(x_2|x_0, x_1)\ldots P(x_{T-1}|x_0, x_1, \ldots, x_{T-2}), \tag{13.18}$$

where the joint probability has been factored as a product of the conditional probabilities estimated by the RNN (initially $P(x_0 = \texttt{SOS}) = 1$). The estimation algorithm is therefore the following:

- given the input $x_0, \ldots, x_{T-1}$ ($x_0 = \texttt{SOS}$, $x_{T-1} = \texttt{EOS}$):
- let $\hat{P} \leftarrow 1$;

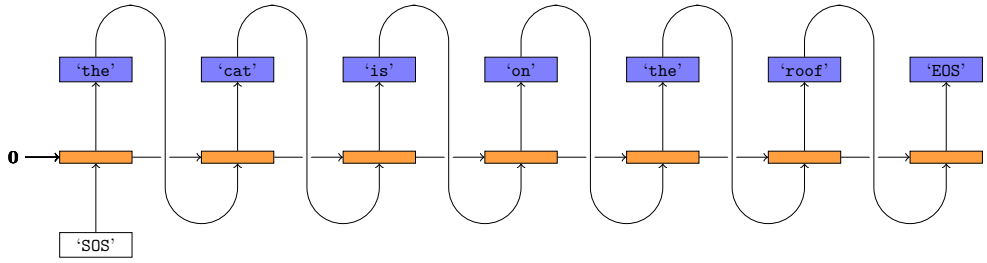


Figure 13.4: Language modeling generating a sentence. The input is the word generated at the previous step, starting with the 'SOS' token. The process continues until 'EOS' is generated.

- for $t \in \{0, 1, \ldots, T-2\}$ do:
  - compute the estimates $\hat{\mathbf{y}}_t \leftarrow RNN(x_t)$;
  - take the one corresponding to the next element in the sequence: $Q \leftarrow \hat{\mathbf{y}}_{t,x_{t+1}}$;
  - update the estimate: $\hat{P} \leftarrow \hat{P} \times Q$;
- output the estimate $\hat{P}$ of the joint probability for the input sequence.

A curious application of language models consists in using them to generate new sequences. To do this it is enough to start with $x_0 =$ SOS and then use the RNN to estimate the corresponding distribution $\hat{\mathbf{y}}_0$. Then x_1 is randomly selected according to $\hat{\mathbf{y}}_0$ and it is fed to the RNN to compute $\hat{\mathbf{y}}_1 \ldots$ and so on and so forth until EOS is selected (see Figure 13.4).

For instance, a language model built on all the works of Shakespeare generated the following text[1]:

```
PANDARUS:
Alas, I think he shall be come approached and the day
When little srain would be attain'd into being never fed,
And who is but a chain and subjects of his death,
I should not sleep.

Second Senator:
They are away this miseries, produced upon my soul,
Breaking and strongly should be buried, when I perish
The earth and thoughts of many states.

DUKE VINCENTIO:
Well, your wit is in the care of side and that.
```

[1] taken from a post on Andrej Karpathy's blog http://karpathy.github.io/2015/05/21/rnn-effectiveness/, where there are many other nice examples.

```
Second Lord:
They would be ruled after this chamber, and
my fair nues begun out of the fact, to be conveyed,
Whose noble souls I'll have the heart of the wars.

Clown:
Come, sir, I will make did behold your worship.

VIOLA:
I'll drink it.
```

Character- vs. word-level models

What should we use to represent sentences in natural languages? It is better to use sequences of words, or of characters? The answer it is not clear, and probably it depends on the specific task for which the model is built.

Models built at the character level have to learn to compose plausible words one character at a time. Therefore, word-level models are usually easier to train. However, building an exhaustive vocabulary is not feasible and at some point the model is going to encounter a word that is outside the vocabulary. To deal with this case, word-level models include a special `UNKNOWN` token. However, this trick may not always work well.

In short, character-level models are harder to train, but also more flexible. Modern language models often use intermediate encodings, in which words are divided in a small number of tokens.

13.3 Long Short Term Memory

One big issue with the basic RNN model is that it struggles to learn long-term dependencies. Consider the hidden states $\mathbf{h}_t$ and $\mathbf{h}_{t+\Delta t}$ at Δt steps of distance. Combining to equations (13.11) and (13.12) we see that the derivative of the loss with respect to $\mathbf{h}_t$ depends on a term with $U_h^\intercal$ applied to a multiple of the derivative with respect to $\mathbf{h}_{t+1}$. Continuing recursively, we can see that it will depend on a term with $(U_h^{\Delta t})^\intercal$ applied to a multiple of the derivative with respect to $\mathbf{h}_{t+\Delta t}$.

Even for relatively small values of Δt the elements of the matrix $U_h^{\Delta t}$ will tend to vanish to zero, or to explode to ∞. In fact, from the point of view of the backpropagation algorithm, the RNN is like a very deep multilayer perceptron, only with shared weights. For a multilayer perceptron 10 layers are enough to make a very deep network, but RNNs are expected to work with sequences that are a lot longer than that.

Long Short Term Memory (LSTM) is an evolution of the basic RNN model designed to address the problem of vanishing or exploding gradients. Thanks

to their structure LSTM networks can model dependencies in the data at a long distance.

The weakness in the basic RNN comes from the linear operation between states at consecutive time steps. LSTM introduces a new mechanism to carry information through the computation. A *cell state* vector $\mathbf{c}_t$ encode the memory of past computations. It has the same size of the hidden state $\mathbf{h}_t$, but it is computed in a way that makes it easy to preserve its values step by step.

Each step the cell state is updated (in the following we will denotes weights as $W_?$ or $U_?$, and biases as $\mathbf{b}_?$). First a new candidate state $\tilde{\mathbf{c}}_t$ is computed from the input $\mathbf{x}_t$ and the previous hidden state $\mathbf{h}_{t-1}$:

$$\tilde{\mathbf{c}}_t = \mathbf{a}_c(W_c\mathbf{x}_t + U_c\mathbf{h}_{t-1} + \mathbf{b}_c), \tag{13.19}$$

where $\mathbf{a}_c(\cdot)$ is an activation function (typically the hyperbolic tangent).

The part of the cell state that is replaced by the candidate state depends on a set of *gates*, which are just vectors of values in the $[0, 1]$ range that are used to let pass or block the information. The *input gate* $\mathbf{i}_t$ and the *forget gate* $\mathbf{f}_t$ are computed as follows:

$$\mathbf{i}_t = \sigma(W_i\mathbf{x}_t + U_i\mathbf{h}_{t-1} + \mathbf{b}_i), \tag{13.20}$$

$$\mathbf{f}_t = \sigma(W_f\mathbf{x}_t + U_f\mathbf{h}_{t-1} + \mathbf{b}_f), \tag{13.21}$$

where σ is the sigmoid activation function.

The new cell state is obtained as the old state (modulated by the forget gate) and the candidate state (modulated by the input gate):

$$\mathbf{c}_t = \mathbf{c}_{t-1} \odot \mathbf{f}_t + \tilde{\mathbf{c}}_t \odot \mathbf{i}_t, \tag{13.22}$$

where $\odot$ is the elementwise product. Note how easy it is for the network to learn to preserve information through the steps. It is enough to make the forget gate assume values close to one and the input gate close to zero. This property holds true also in the backward pass.

At this point the cell state is used to update the hidden state. This is done with another gate (the *output gate* $\mathbf{o}_t$) that modulates the value of the cell state:

$$\mathbf{o}_t = \sigma(W_o\mathbf{x}_t + U_o\mathbf{h}_{t-1} + \mathbf{b}_o), \tag{13.23}$$

$$\mathbf{h}_t = \mathbf{a}_h(\mathbf{c}_t) \odot \mathbf{o}_t, \tag{13.24}$$

where $\mathbf{a}_h(\cdot)$ is an activation function (in this case also the hyperbolic tangent is a common choice).

As a last operation, the output is computed from the hidden state (this part is the same of the basic RNN):

$$\hat{\mathbf{y}}_t = \mathbf{a}_y(W_y\mathbf{h}_t + \mathbf{b}_t). \tag{13.25}$$

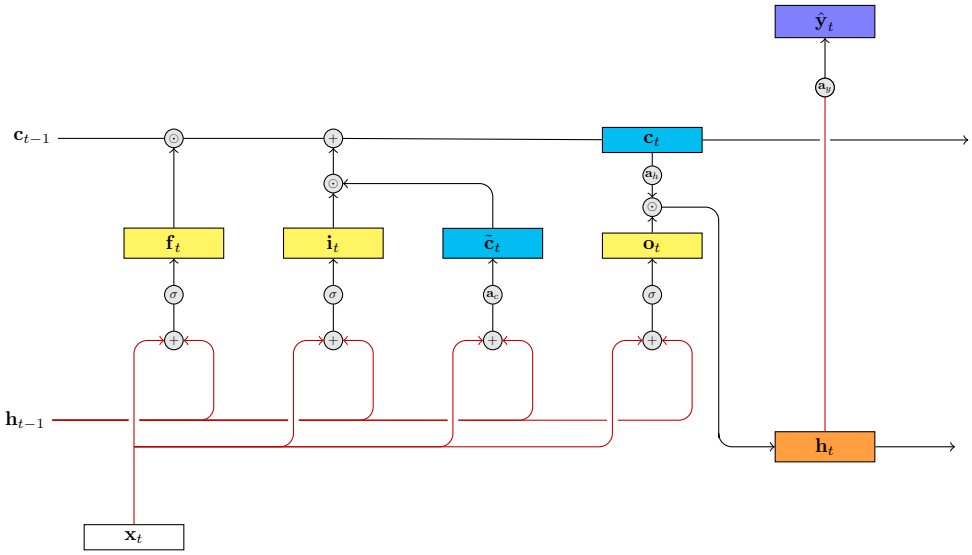


Figure 13.5: Graphical representation of the LSTM model. Connections in red are weighted by the parameters of the model. Usually $\mathbf{a}_c$ and $\mathbf{a}_h$ are the hyperbolic tangent activation function. The function $\mathbf{a}_y$ depends on the problem.

Figure 13.5 shows a scheme of the LSTM architecture.

LSTM networks are certainly more complicated than the basic RNN, but they are also significantly more powerful, and represent the state of the art in terms of recurrent neural models. Despite their complexity, they can be used as direct replacement of the basic RNN.

13.4 Gated Recurrent Unit

Many models have been proposed to simplify LSTM without losing its effectiveness. Among these the Gated Recurrent Unit (GRU) is the one that got closer to the objective.

In the GRU model there is not a separate cell state. The update of the hidden state depends on two gates, the *update gate* $\mathbf{z}_t$ and the *reset gate* $\mathbf{r}_t$:

$$\mathbf{z}_t = \sigma(W_z\mathbf{x}_t + U_z\mathbf{h}_{t-1} + \mathbf{b}_z), \tag{13.26}$$

$$\mathbf{r}_t = \sigma(W_r\mathbf{x}_t + U_r\mathbf{h}_{t-1} + \mathbf{b}_r). \tag{13.27}$$

The reset gate is used to modulate the previous state in the computation of the candidate new state $\tilde{\mathbf{h}}_t$:

$$\tilde{\mathbf{h}}_t = \mathbf{a}_h(W_h\mathbf{x}_h + U_h(\mathbf{r}_t \odot \mathbf{h}_{t-1}) + \mathbf{b}_h). \tag{13.28}$$

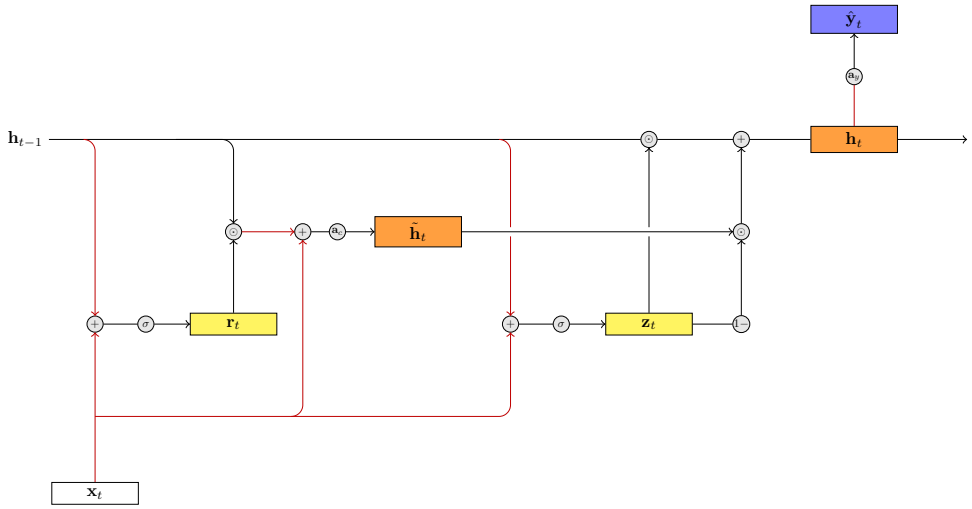


Figure 13.6: Graphical representation of the GRU model.

The update gate is used to mix the previous and the candidate state:

$$\mathbf{h}_t = \mathbf{z}_t \odot \mathbf{h}_{t-1} + (1 - \mathbf{z}_t) \odot \tilde{\mathbf{h}}_t. \tag{13.29}$$

Figure 13.6 summarizes the GRU architecture.

The behavior of GRU is similar to that of LSTM, but it uses less parameters and its implementation is a lot simpler. Most researchers, however, still prefer LSTM over GRU.

13.5 Multilayer RNNs

The complexity of a RNN is determined by the number of hidden neurons. The larger the hidden layer, the more complex the behavior that the network is able to learn. However, another way to increase the complexity of a RNN is to stack multiple recurrent layers.

For a network of depth D, at each time step we will have D hidden states $\mathbf{h}_t^{(0)}, \mathbf{h}_t^{(1)}, \ldots, \mathbf{h}_t^{(D-1)}$. The state $\mathbf{h}_t^{(0)}$ is computed from the input $\mathbf{x}_t$ and the previous state $\mathbf{h}_{t-1}^{(0)}$. For $l > 0$ the state $\mathbf{h}_t^{(l)}$ is computed from the current state of the layer below $\mathbf{h}_t^{(l-1)}$ and the previous state for the same layer $\mathbf{h}_{t-1}^{(l)}$. The topmost hidden state $\mathbf{h}^{(D-1)}$ is finally used to compute the output $\mathbf{y}_t$.

This architecture is depicted in Figure 13.7. The basic RNN model LSTM and GRU can all be stacked in multiple layers.

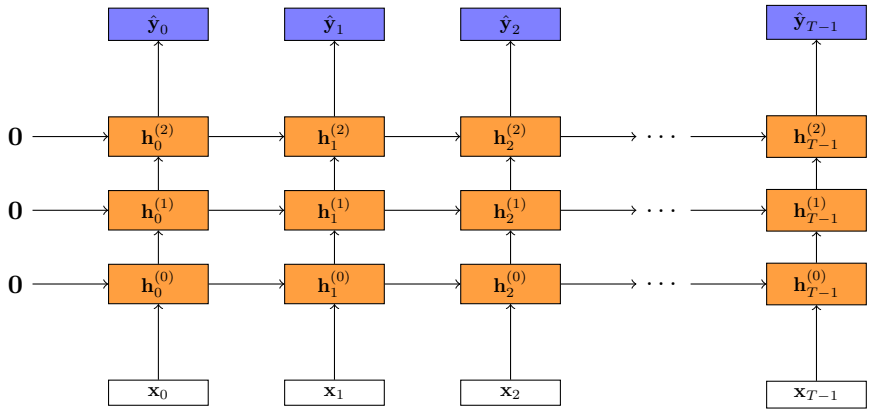


Figure 13.7: Diagram of a multilayer RNN with three hidden layers.

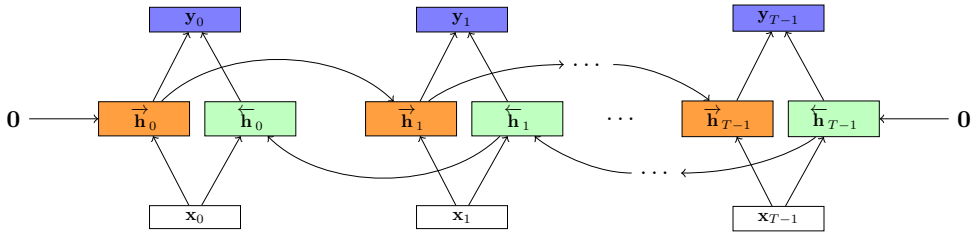


Figure 13.8: Diagram of a bidirectional RNN.

Bidirectional RNN

RNNs are asymmetric networks. The output at time t is determined by all the input values with $t' \leq t$, but not by the rest of the input sequence. Sometimes the knowledge of all the inputs allows to achieve significantly better accuracy. For instance, in optical character recognition knowing both the strokes to the left and to the right may greatly help in recognizing the character at the current location.

A simple but effective way of exploiting the information from both past and future samples is to stack two layers of hidden states. The resulting model is called *bidirectional RNN*. The state $\overrightarrow{\mathbf{h}}_t$ will be computed for increasing values of t, by combining $\mathbf{x}_t$ and $\overrightarrow{\mathbf{h}}_{t-1}$. The state $\overleftarrow{\mathbf{h}}_t$ will be computed for decreasing values of t, by combining $\mathbf{x}^t$ and $\overleftarrow{\mathbf{h}}_{t+1}$. Both $\overrightarrow{\mathbf{h}}_t$ and $\overleftarrow{\mathbf{h}}_t$ are used to compute the output $\hat{\mathbf{y}}_t$ or, are used as input for another bidirectional layer.

This architecture is depicted in Figure 13.8. The basic RNN model LSTM and GRU can all be used as basic blocks for this architecture.

13.6 Summary

In this lecture we introduced recurrent networks, a powerful neural architecture for the processing of sequential data. In particular we discussed:

- the basic RNN model, that is a simple implementation of the general strategy behind recurrent networks;
- language models built with RNNs;
- more complex models (LSTM and GRU) addressing weaknesses of the basic model (vanishing and exploding gradients);
- more powerful networks made of multiple stacked recurrent layers;
- bidirectional networks that process the data both in the forward and in the backward directions.

Lecture 14

Deep Reinforcement Learning

Supervised learning makes it possible to solve a large variety of problems. However, there are applications in which a training set of annotated samples is not available. In many cases, for instance, answers are not just right or wrong, but their goodness depends on their cumulative effect on the environment.

Reinforcement learning is a learning paradigm in which a model is trained to iteratively take decisions for the achievement of a specific goal. Learning is achieved by taking into account the feedback provided by the environment in which the decisions are executed. This scenario is typical of many application domains including robot control, investment strategies, game playing...

In the last years the combination of reinforcement learning algorithms with deep learning models allowed to obtain amazing results in these domains and today it is considered by many the most promising route towards the goal of "Artificial General Intelligence".

Reinforcement learning is a large area of study. This is just a short introduction to the topic in which the mathematical rigor is sacrificed for a more intuitive description.

14.1 Sequential decision making

The problem addressed by reinforcement learning is that of sequential decision making. It includes two main elements: an *agent* and an *environment*. At each discrete time step the environment provides *observations* to the agent. The agent performs *actions* that modify the environment. Every time an action is executed it affects the environment. It may also result in a feedback in the form of a numerical *reward*. The goal of the agent is to maximize the cumulative reward.

An example of this setup is represented in Figure 14.1. The environment

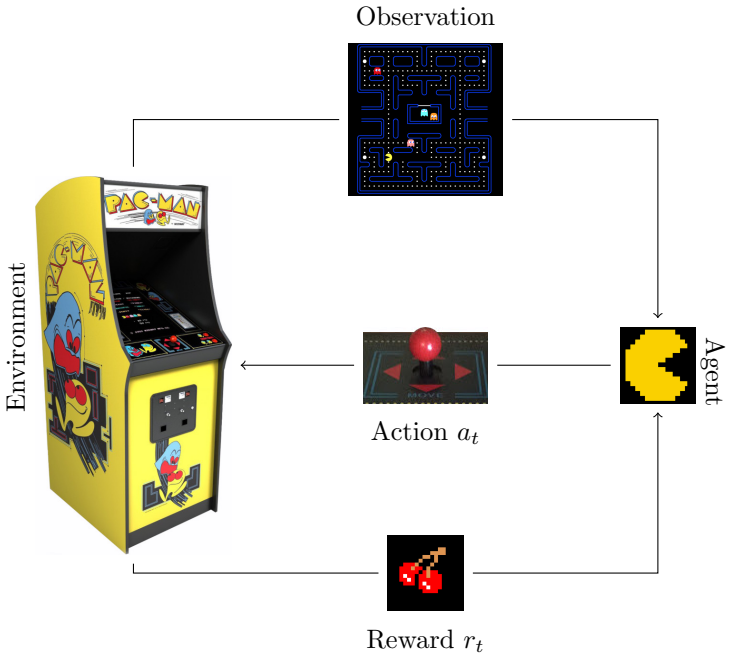


Figure 14.1: Graphical representation of a sequential decision process. At each time step the environment provides an observation to the agent, that uses it to select an action to perform. The environment sends back a numerical reward to the agent.

is the arcade machine, observations are the audio/video signals, the agent is the player piloting Pac-Man, the actions are the movements given with the joystick (up, down, left, right), the reward is the increment in score caused by an action.

The environment is characterized by a *state* $s_t \in \mathcal{S}$ which determines observations and rewards. The state is supposed to be *Markovian*, that is, it contains all the information needed to determine the future, making previous states irrelevant:

$$P(s_{t+1}|s_t) = P(s_{t+1}|s_0, s_1, \ldots, s_t). \tag{14.1}$$

The agent observes a representation of the current state and selects an action $a_t \in \mathcal{A}$. The execution of a_t generates a reward r_t and makes the environment transition to the next state s_{t+1}. From the point of view of the observer, the whole process can be described by the *history* H_t:

$$H_t = s_0, a_0, r_0, s_1, a_1, r_1, \ldots, s_{t-1}, a_{t-1}, r_{t-1}, s_t. \tag{14.2}$$

Note that:

- the state can be fully-observable or not;
- in general, the process is stochastic and the next state s_{t+1} depends on both the current state s_t and the action a_t:

$$s_{t+1} \sim T(s_t, a_t), \tag{14.3}$$

  where $T(\cdot, \cdot)$ is the probability distribution of the new state;

- similarly, the reward r_t follows the distribution $R(\cdot, \cdot)$:

$$r_t \sim R(s_t, a_t). \tag{14.4}$$

Deterministic processes can be seen as special cases of the more general stochastic formalization.

Policies and return

By selecting its actions the agent follows a strategy encoded in a *policy* π. The policy is a probability distribution over actions depending on the current state:

$$a_t \sim \pi(a|s_t). \tag{14.5}$$

This definition includes as a special case deterministic policies, which for each state assigns probability one to a single action. In this case we will simply write $a_t = \pi(s_t)$. Non-deterministic policies allow some randomness in the decision process. For instance they can model an agent that chooses the action by tossing a coin. This is a key property for some reinforcement learning methods.

The objective of sequential decision making is to make it so the agent follows a policy that maximizes the future rewards. Instead of considering the sum of the rewards, it is maximized the *return* G_t defined as:

$$G_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots = \sum_{i=0}^{+\infty} \gamma^i r_{t+i}, \tag{14.6}$$

where $\gamma \in [0, 1]$ is a *discount factor* that weights more rewards that are close in time than rewards in the far future. The return is preferred over the simple sum for a few of reasons: (i) it better reflects our limited confidence on future estimates (it plays the same role of interest rates in finance); (ii) it is mathematically well-behaved, as it is always finite for bounded rewards.

Summing up the problem of sequential decision making consists in finding the policy that maximizes the expected return:

$$\pi^\star = \arg\max_\pi \mathbb{E}[G_t], \tag{14.7}$$

where $\mathbb{E}[\cdot]$ denotes the expected value. Note that G_t depends only on future rewards. This means that in choosing a_t the agent should not take into account the rewards it already obtained.

14.2 Reinforcement learning

The framework described in the previous section is common to many fields, including control theory, planning, game theory, search algorithms, decision theory... Depending on the specific properties of the process problem (14.7) can be solved with different techniques. For instance, dynamic programming methods can effectively found the exact solution when there are a finite number of states, and the dynamics of the environment (state transitions and reward distributions) are known.

For many interesting problems, however, exactly modeling the environment is very hard because the number of states can be huge (or even infinite), and its dynamic can be largely unknown or too complex to be modeled. Reinforcement learning aims to deal with these cases, in which the prior knowledge about the problem is minimal or too hard to exploit. There is no direct supervision, and the only information available is the experience of the agent (sequences of states, actions and rewards).

In reinforcement learning there are three main aspects that the agent can learn:

- it can learn to behave effectively;
- it can learn to evaluate how good a state is;
- it can learn how the environment reacts to its actions.

The three approaches can be combined, and all result in a policy for the agent. The first is the more straightforward, since directly aims at learning the optimal policy. The second approach (learn to evaluate states) is the easiest to model and leads to elegant and effective algorithms. The third (learn the environment) is probably the most complicated and currently the less explored.

All three approaches are based on learning functions: from states to actions (learn to behave), from states to real values (learn to evaluate), from states to states (learn the environment). These functions will be represented by learnable models. In deep reinforcement learning they are deep neural networks (perceptrons, CNN, RNN...).

14.3 Value function learning

If we are able to evaluate states we can define policies that will likely take us to the good ones. A state is good for a given policy if the expected return is high for an agent that follows the policy starting from that state. The *state-value function* represents this concept.

The state-value function $v_\pi(s)$ for the policy π is the expected return that the agent will obtain if starting from state s it follows π:

$$v_\pi(s) = \mathbb{E}_\pi[G_t|s_t = s]. \tag{14.8}$$

Don't be fooled by the notation, the function is stationary, that is, it does not depend on a particular time step t.

The knowledge of the state-value function could be used to improve the policy: just take the action that maximizes the expected immediate reward plus the discounted expected value function from the next state. Unfortunately, to implement this we would need the knowledge of the distribution of the next state, which may be very difficult to model.

This approach works well, however, if we use another function, which is called the *action-value function*. The action-value function $q_\pi(s, a)$ is the expected return that an agent would obtain if, starting from state s, it takes action a and after that it continues by following the policy π.

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t|s_t = s, a_t = a]. \tag{14.9}$$

In practice the action-value function is like the state-value function, but with the first action fixed to a independently on the policy π. In fact, the state-value function can be expressed in terms of the action-value function:

$$v_\pi(s) = \sum_{a\in\mathcal{A}} \pi(a|s) q_\pi(s, a). \tag{14.10}$$

Equation (14.6) allows to split the action value function in an immediate reward plus the expected return at the next step:

$$\begin{aligned} q_\pi(s, a) &= \mathbb{E}_\pi[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \dots |s_t = s, a_t = a] \\ &= \mathbb{E}_\pi[r_t + \gamma(r_{t+1} + \gamma r_{t+2} + \dots)|s_t = s, a_t = a] \\ &= \mathbb{E}_{r\sim R(s,a)}[r] + \gamma \mathbb{E}_{s'\sim T(s,a)}[v_\pi(s')], \end{aligned} \tag{14.11}$$

where $r$and s' represent the reward and the new state obtained after performing action a in state s, and are taken from the (possibly unknown) distributions $R(s, a)$, and $T(s, a)$.

Given a policy π if we are able to compute its action-value function q_π we can improve it by using the *greedy* policy π':

$$\pi'(a|s) = \begin{cases} 1 & \text{if } a = \arg\max_{a'\in\mathcal{A}} q_\pi(s, a'), \\ 0 & \text{otherwise.} \end{cases} \tag{14.12}$$

The greedy policy is guaranteed to be at least as good of the original policy.

Optimal value functions

Our goal is to find the policy ensuring the highest expected return. To this purpose we can define the optimal value functions as those measuring the highest expected return:

$$v^\star(s) = \max_\pi v_\pi(s), \tag{14.13}$$

$$q^\star(s,a) = \max_\pi q_\pi(s,a), \tag{14.14}$$

with the obvious relationship:

$$v^\star(s) = \max_{a \in \mathcal{A}} q^\star(s,a). \tag{14.15}$$

By combining equations (14.11) and (14.15) we obtain the *Bellmann equation*:

$$q^\star(s,a) = \mathbb{E}_{r \sim R(s,a)}[r] + \gamma \mathbb{E}_{s' \sim T(s,a)}[\max_{a' \in \mathcal{A}} q^\star(s',a')]. \tag{14.16}$$

By solving the Bellmann equation we obtain the optimum action-value function whose greedy policy is the optimal policy:

$$\pi^\star(a|s) = \begin{cases} 1 & \text{if } a = \arg\max_{a' \in \mathcal{A}} q^\star(s,a'), \\ 0 & \text{otherwise.} \end{cases} \tag{14.17}$$

Q-learning

The real action-value function can be found only for small-scale problems, for instance those with a small number of states. In many cases it needs to be approximated by a parametric model. Moreover, the full state information is not always available to the agent which can only have access to its representation in terms of *features* $\mathbf{x}_t = f(s_t)$. Therefore, the problem becomes that of finding the best parameters $\boldsymbol{\theta}$ for a parametric model $q_{\boldsymbol{\theta}}$ approximating the optimal action value function:

$$q_{\boldsymbol{\theta}}(\mathbf{x}, a) \simeq q^\star(s,a), \tag{14.18}$$

with $\mathbf{x} = f(s)$.

Q-learning, one of the most successful reinforcement learning algorithm, tries to do exactly that. The basic idea is to turn the Bellmann equation in a loss function.

At time t we have the feature vector $\mathbf{x}_t$ representing the current state, the agent chooses and performs the action a_t according to a *behavior policy* π

(i.e. $a_t \sim \pi(a|\mathbf{x}_t)$), that can be different from the policy we are currently optimizing. The environment reacts by providing the reward r_t and the features for the new state $\mathbf{x}_{t+1}$. The loss function is then the square error with respect to the Bellmann equation:

$$L_t = \frac{1}{2}\left(r_t + \gamma \max_{a' \in \mathcal{A}} q_{\boldsymbol{\theta}}(\mathbf{x_{t+1}}, a') - q_{\boldsymbol{\theta}}(\mathbf{x}_t, a_t)\right)^2, \tag{14.19}$$

which can be used to update the parameters $\boldsymbol{\theta}$ with one step of gradient descent:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \eta(r_t + \gamma \max_{a' \in \mathcal{A}} q_{\boldsymbol{\theta}}(\mathbf{x_{t+1}}, a') - q_{\boldsymbol{\theta}}(\mathbf{x}_t, a_t))\nabla_{\boldsymbol{\theta}} q_{\boldsymbol{\theta}}(\mathbf{x}_t, a_t), \tag{14.20}$$

where η is the learning rate. Note that the gradient is taken for the term $q_{\boldsymbol{\theta}}(\mathbf{x}_t, a_t)$, but not for $q_{\boldsymbol{\theta}}(\mathbf{x_{t+1}}, a')$. This improves the stability of the algorithm.

On-line learning

One way to use Q-learning is to apply it on-line, that is, while the agent interacts with the environment. We use a loop in which the agent (i) observes the state, (ii) chooses an action according to the behavior policy, (iii) observes the reward and the new state, and finally (iv) uses all this information to update the parameters by applying (14.20).

If we use as behavior policy the greedy policy for the current $q_{\boldsymbol{\theta}}$ then the method may not be able to converge to a good solution, because it may get stuck repeating the same sub-optimal actions. To ensure that there is always some degree of exploration, we can use the *ϵ-greedy* policy π_ϵ instead. The ϵ-greedy policy is random with probability ϵ, otherwise is the same of the regular greedy policy:

$$\pi_\epsilon(a|s) = \frac{\epsilon}{|\mathcal{A}|} + (1-\epsilon)\pi(a|s), \tag{14.21}$$

where π is the greedy policy for $q_{\boldsymbol{\theta}}$. The use of the ϵ-greedy policy ensures that eventually all strategies are explored.

The following code sketches a possible Python implementation of Q-learning in the case of the linear model:

$$q_{W,\mathbf{b}}(\mathbf{x}, a) = W_a\mathbf{x} + b_a. \tag{14.22}$$

The code assumes that there is an object `environment` wit the methods: `environment.features()` that returns a feature vector representing the current state and `environment.make_actions(a)` that executes action `a` transitions in the next state and returns the reward.

```
def q_learning(environment, actions, lr, steps, gamma, epsilon):
  x = environment.features()
  W = np.zeros((actions, x.size))
  b = np.zeros(actions)
  q = W @ x + b
  for step in range(steps):
    if random.random() < epsilon:
      a = random.randrange(actions)
    else:
      a = q.argmax()
    r = environment.make_action(a)
    x_next = environment.features()
    q_next = W @ x_next + b
    update = -(r + gamma * q_next.max() - q[a])
    W[a, :] -= lr * update * x
    b[a] -= lr * update
    x = x_next
    q = q_next
  return W, b
```

Here the gradient with respect to the weights is the feature vector $\mathbf{x}$ and that with respect to the biases is one. In deep reinforcement learning a deep convolutional neural network is used to approximate the action-value function. In this case the gradient $\nabla_\theta q_\theta(\mathbf{x}, a)$ is computed by backpropagation.

Batch Q-learning

On-line learning is simple to implement and corresponds to our intuitive notion of learning, in which the agent learns continuously while it is immersed in the environment. However, it is not very efficient. In fact, each "event" experienced by the agent is used to update the weights only once. Depending on the problem, executing an action in the environment can be quite time consuming. For instance, in robotics it may require the actual control of a robot.

The fact that Q-learning considers different behavior and target policies makes it possible to follow an alternative strategy. It is called *batch learning* and consists in collecting a training set of events $\mathbf{x}_t, a_t, r_t, \mathbf{x}_{t+1}$ and in using stochastic gradient descent to optimize the parameters by minimizing the average loss. To form the training set we can use any behavior policy. For instance, we can just observe and record the actions of a different agent (like a human expert).

Note that the policy learned by the agent may be significantly different (even better) than the behavior policy.

Experience replay

The combination of on-line and batch learning make it possible to train the agent very effectively without requiring the collection of a training set. *Experience replay* is a technique which uses Q-learning to train an agent by using as behavior policy the policy previously learned by the agent.

In experience replay the events $\mathbf{x}_t, a_t, r_t, \mathbf{x}_{t+1}$ generated by the agent are not immediately used for training, but are stores in a data set. After each event, a sample of the past is drawn at random from the data set and used to update the parameters.

This approach has several advantages over on-line Q-learning. First, each event is used multiple times to update the parameters. Second, it avoids the correlations that are typical of consecutive events making the algorithm converge more smoothly. Experience replay, and similar techniques made it possible to use deep models in reinforcement learning, a result that was very difficult to achieve with standard on-line Q-learning.

14.4 Policy learning

Sometimes learning a value function can be harder than just learning a good policy. In fact, there are cases in which is relatively easy to predict the best action, but it is difficult to assess exactly how good or bad it is with respect to the alternatives. Another issue with Q-learning is that it always finds deterministic policies, while in some cases a stochastic policy may work better (e.g. in the rock-paper-scissor game!).

Q-learning presents an additional drawback: it needs to select the action with the highest action-value for the current state, and this maximization step can be hard when there are many possible actions, or when the action space is continuous (as in robotics, for instance).

Policy-based methods aim to approximate the optimal policy $\pi^\star$ with a parametric model $\pi_{\boldsymbol{\theta}}$. For instance, if there is a finite number of actions $\pi_{\boldsymbol{\theta}}$ can be a deep neural network ending with a softmax activation at the end ($\pi_{\boldsymbol{\theta}}$ must be a probability distribution over the actions).

Suppose that our objective $J(\boldsymbol{\theta})$ is to maximize the single-step reward r obtained by using the policy $\pi_{\boldsymbol{\theta}}$ starting from a random state s sampled from a distribution D:

$$J(\boldsymbol{\theta}) = \mathbb{E}_{(D,\pi_{\boldsymbol{\theta}},R)}[r] = \sum_{s\in\mathcal{S}} D(s) \sum_{a\in\mathcal{A}} \pi_{\boldsymbol{\theta}}(a|s)\mathbb{E}_{r\sim R(s,a)}[r]. \tag{14.23}$$

To optimize this objective we can use stochastic gradient descent (or, more

precisely, ascent), but we have to compute its gradient with respect to $\boldsymbol{\theta}$:

$$\begin{aligned}\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) &= \nabla_{\boldsymbol{\theta}} \left(\sum_{s \in \mathcal{S}} D(s) \sum_{a \in \mathcal{A}} \pi_{\boldsymbol{\theta}}(a|s) \mathbb{E}_{r \sim R(s,a)}[r] \right) \\ &= \sum_{s \in \mathcal{S}} D(s) \sum_{a \in \mathcal{A}} \nabla_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(a|s) \mathbb{E}_{r \sim R(s,a)}[r] \\ &= \sum_{s \in \mathcal{S}} D(s) \sum_{a \in \mathcal{A}} \pi_{\boldsymbol{\theta}}(a|s) \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a|s) \mathbb{E}_{r \sim R(s,a)}[r] \\ &= \mathbb{E}_{(D, \pi_{\boldsymbol{\theta}}, R)}[\nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a|s) r],\end{aligned} \tag{14.24}$$

where the logarithm is introduced by the following trick, valid for every positive function:

$$\nabla_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(a|s) = \pi_{\boldsymbol{\theta}}(a|s) \frac{\nabla_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(a|s)}{\pi_{\boldsymbol{\theta}}(a|s)} = \pi_{\boldsymbol{\theta}}(a|s) \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a|s). \tag{14.25}$$

The result in Equation (14.24) can be extended from the single-step reward to the total return $J(\boldsymbol{\theta}) = \mathbb{E}_{(D, \pi_{\boldsymbol{\theta}})}[G_t]$, resulting in the following:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \nabla_{\boldsymbol{\theta}} \mathbb{E}_{(D, \pi_{\boldsymbol{\theta}})}[G_t] = \mathbb{E}_{(D, \pi_{\boldsymbol{\theta}})}[\nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a|s) q_{\pi_{\boldsymbol{\theta}}}(s, a)], \tag{14.26}$$

which is the basis for the *policy-gradient* methods. A simple way to implement this is to use the return G_t as an unbiased estimate of $q_{\pi_{\boldsymbol{\theta}}}(s, a)$. This approach results in the *Reinforce* method. It consists in two phases: first let the agent acts in the environment recording features, actions and rewards until it gets in a terminal state (a state from which it cannot move out and where the reward is forever zero). Then compute the return at each time step, and apply one step of stochastic gradient ascent as follows:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \eta \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t|s_t) G_t, \tag{14.27}$$

which can be implemented in a few lines of Python code as sketched here below (the method `environment.terminal()` tells if a terminal state has been reached):

```
def reinforce(environment, lr, steps, init_theta, gamma):
  theta = init_theta
  for step in range(steps):
    actions = []
    features = []
    rewards = []
    while not environment.terminal():
      x = environment.features()
      prob = policy(theta, x)
      a = np.random.choice(np.arange(prob.size), p=prob)
      r = environment.make_action(a)
      actions.append(a)
```

```
      features.append(x)
      rewards.append(r)
    environment.reset()
    returns = [rewards[-1]]
    for r in rewards[-2::-1]:
      returns.append(r + returns[-1] * gamma )
    returns = returns[::-1]
    for x, a, g in zip(features, actions, returns):
      grad = grad_log_policy(theta, x, a)
      theta += lr * grad * g
```

Of course the functions `policy` and `grad_log_policy` must be adapted to the model used to represent π_θ and $\nabla_\theta \log \pi_\theta$.

The reinforce algorithm suffers some of the limitation of original on-line Q-learning: it can be unstable due to the correlation of events that are close in time and it uses each event just once.

Actor-critic

To address some of the issues with the Reinforce algorithm, instead of using G_t as an estimate for $q_{\pi_\theta}(s,a)$ we could use a parametric estimate $q_\omega(s,a)$. Following this intuition we get the *actor-critic* method, in which we have two models: the actor π_θ chooses the actions and the critic $q_\omega(s,a)$ evaluate them. This method is both value-based and policy-based. The algorithm updates the parameters of the actor with stochastic gradient ascent, and those of the critic in a way similar to that used in Q-learning:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \eta \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t|s_t) q_{\boldsymbol{\omega}}(s,a), \tag{14.28}$$

$$\boldsymbol{\omega} \leftarrow \boldsymbol{\omega} + \nu(r + \gamma q_{\boldsymbol{\omega}}(s',a') - q_{\boldsymbol{\omega}}(s,a)) \nabla_{\boldsymbol{\omega}} q_{\boldsymbol{\omega}}(s,a), \tag{14.29}$$

where η and ν are two separate learning rates for the two models.

```
def actor_critic(environment, lr1, lr2, steps, init_theta,
    init_omega, gamma):
  theta = init_theta
  omega = init_omega
  x = environment.features()
  prob = policy(theta, x)
  a = np.random.choice(np.arange(prob.size), p=prob)
  for step in range(steps):
    r = environment.make_action(a)
    x_next = environment.features()
    prob = policy(theta, x)
    a_next = np.random.choice(np.arange(prob.size), p=prob)
    grad = grad_log_policy(theta, x, a)
    theta += lr1 * grad * q(omega, x, a)
    delta = r + gamma * q(omega, x_next, a_next) - q(omega, x, a)
    omega += lr2 * delta * grad_q(omega, x, a)
```

14.5 Summary

In this lecture we gave a brief peek at reinforcement learning. It is a vast topic, and we have been just able to scratch its surface. Among the points covered there are:

- the basic framework for sequential decision making problems;
- methods based on value-function approximations like Q-learning;
- methods based on policy gradients (namely, the Reinforce method);
- the hybrid actor-critic method.