CentraleSupélec

Machine Learning

Last revision: February 15, 2024

Jérémy Fix

Hervé Frezza-Buet

Matthieu Geist

Contents

I Overview 9

1 Introduction 11
- 1.1 Datasets . . . 11
  - 1.1.1 Data sampling . . . 11
  - 1.1.2 Data conditionning . . . 14
- 1.2 Different learning problems... . . . 15
  - 1.2.1 Unsupervised learning . . . 15
  - 1.2.2 Supervised learning . . . 19
  - 1.2.3 Semi-supervised learning . . . 21
  - 1.2.4 Reinforcement learning . . . 21
- 1.3 Different learning strategies . . . 25
  - 1.3.1 Inductive Learning . . . 25
  - 1.3.2 Transductive learning . . . 25

2 The frequentist approach 27
- 2.1 Hypothesis spaces . . . 27
  - 2.1.1 Parametric and nonparametric hypothesis spaces . . . 27
  - 2.1.2 The linear case . . . 27
  - 2.1.3 Linear separability . . . 28
- 2.2 Risks . . . 29
  - 2.2.1 Loss functions . . . 29
  - 2.2.2 Real and empirical risks . . . 30
  - 2.2.3 Are good predictors good ? . . . 31
  - 2.2.4 Empirical risk minimization and overfitting . . . 31
- 2.3 Ensemble methods . . . 32
  - 2.3.1 Bagging . . . 32
  - 2.3.2 Random models . . . 32
  - 2.3.3 Boosting . . . 33

3 The Bayesian approach 35
- 3.1 Density of probability . . . 35
  - 3.1.1 Reminder . . . 35
  - 3.1.2 Joint and conditional densities of probability . . . 35
  - 3.1.3 The Bayes' rule for densities of probability . . . 36
- 3.2 Bayesian inference . . . 36
  - 3.2.1 The model . . . 36
  - 3.2.2 The parameter update . . . 37
  - 3.2.3 Update from bunches of data . . . 40
- 3.3 Bayesian learning for real . . . 41

4 Evaluation 43
- 4.1 Real risk estimation . . . 43
  - 4.1.1 Cross-validation . . . 43
  - 4.1.2 Real risk optimisation . . . 44
- 4.2 The specific case of classification . . . 44

4.2.1 Confusion matrix . . . 44
4.2.2 The specific case of bi-class problems . . . 46

II Concepts for Machine Learning 49

5 Risks 51
5.1 Controlling the risk . . . 51
5.1.1 The considered learning paradigm . . . 51
5.1.2 Bias-variance decomposition . . . 54
5.1.3 Consistency of empirical risk minimization . . . 54
5.1.4 Towards bounds on the risk . . . 57
5.1.5 To go further . . . 60
5.2 Classification, convex surrogates and calibration . . . 60
5.2.1 Binary classification with binary loss . . . 60
5.2.2 Cost-sensitive multiclass classification . . . 62
5.2.3 Calibration . . . 63
5.2.4 To go further . . . 63
5.3 Regularization . . . 64
5.3.1 Penalizing complex solutions . . . 64
5.3.2 Examples . . . 64
5.3.3 To go further . . . 65

6 Preprocessing 67
6.1 Selecting and conditioning data . . . 67
6.1.1 Collecting data . . . 67
6.1.2 Conditioning the data . . . 67
6.2 Dimensionality reduction . . . 69
6.2.1 Variable selection . . . 70
6.2.2 Principal Component Analysis (PCA) . . . 73
6.2.3 Relationship between covariance, Gram and euclidean distance matrices . . . 81
6.2.4 Principal component analysis in large dimensional space ($N \ll d$) . . . 83
6.2.5 Kernel PCA (KPCA) . . . 84
6.2.6 Further reading . . . 87
6.2.7 Manifold learning . . . 87

III Support vector machines 91

7 Introduction 93
7.1 Acknowledgment . . . 93
7.2 Objectives . . . 93
7.3 By the way, what is an SVM? . . . 93
7.4 How does it work? . . . 94

8 Linear separator 95
8.1 Problem Features and Notations . . . 95
8.1.1 The Samples . . . 95
8.1.2 The Linear Separator . . . 95
8.2 Separability . . . 95
8.3 Margin . . . 96

9 An optimisation problem **99**
9.1 The problem the SVM has to solve . . . 99
9.1.1 The separable case . . . 99
9.1.2 General case . . . 101
9.1.3 Relation with the ERM . . . 101
9.2 Lagrangian resolution . . . 102
9.2.1 A convex problem . . . 102
9.2.2 The direct problem . . . 102
9.2.3 The Dual Problem . . . 103
9.2.4 An intuitive view of optimization under constraints . . . 103
9.2.5 Back to the specific case of SVMs . . . 107

10 Kernels **111**
10.1 The feature space . . . 111
10.2 Which Functions Are Kernels? . . . 113
10.2.1 A Simple Example . . . 113
10.2.2 Conditions for a kernel . . . 114
10.2.3 Reference kernels . . . 114
10.2.4 Assembling kernels . . . 114
10.3 The core idea for SVM . . . 115
10.4 Some Kernel Tricks . . . 116
10.4.1 Data Normalization . . . 116
10.4.2 Centering and Reduction . . . 116
10.5 Kernels for Structured Data . . . 117
10.5.1 Document Analysis . . . 117
10.5.2 Strings . . . 117
10.5.3 Other Examples . . . 118

11 Solving SVMs **119**
11.1 Quadratic Optimization Problems with SMO . . . 119
11.1.1 General Principle . . . 119
11.1.2 Optimality Detection . . . 119
11.1.3 Optimisation Algorithm . . . 121
11.1.4 Numerical Problems . . . 121
11.2 Parameter Tuning . . . 121

12 Regression **123**
12.1 Definition of the Optimization Problem . . . 123
12.2 Resolution . . . 123
12.3 Examples . . . 125

13 Compedium of SVMs **127**
13.1 Classification . . . 127
13.1.1 C-SVC . . . 127
13.1.2 ν-SVC . . . 127
13.2 Regression . . . 127
13.2.1 ϵ-SVR . . . 128
13.2.2 ν-SVR . . . 128
13.3 Unsupervized Learning . . . 128
13.3.1 Minimal enclosing sphere . . . 128
13.3.2 One-class SVM . . . 129

IV Vector Quantization **131**

14 Introduction and notations for vector quantization **133**
14.1 An unsupervised learning problem . . . 133
14.1.1 Formalization as a dummy supervised learning problem . . . 133
14.1.2 Choosing the suitable loss function . . . 134
14.1.3 Samples . . . 135
14.2 Minimum of distortion . . . 135
14.2.1 Non unicity . . . 136
14.2.2 Sensitivity to the density of input samples . . . 136
14.2.3 Controlling the quantization accuracy . . . 137
14.3 Preserving topology . . . 140
14.3.1 Notations for graphs . . . 140
14.3.2 Masked Delaunay triangulation . . . 140
14.3.3 Structuring raw data . . . 143

15 Main algorithms **145**
15.1 K-means . . . 145
15.1.1 The Lloyd iteration . . . 145
15.1.2 The Linde-Buzo-Gray algorithm . . . 146
15.1.3 The online k-means . . . 147
15.2 Incremental neural networks . . . 147
15.2.1 Growing Neural Gas . . . 147
15.2.2 Growing Grid . . . 147
15.3 Self-Organizing Maps . . . 149
15.3.1 Principle . . . 149
15.3.2 Convergence issues . . . 150
15.3.3 Example . . . 151

V Neural networks **153**

16 Introduction **155**

17 Feedforward neural networks **159**
17.1 Single Layer perceptron . . . 159
17.1.1 Perceptron . . . 159
17.1.2 ADaptive LINear Elements . . . 166
17.1.3 Limitations . . . 168
17.1.4 Single layer perceptron . . . 168
17.2 Radial Basis Function networks (RBF) . . . 172
17.2.1 Architecture and training . . . 172
17.2.2 Universal approximation . . . 173
17.3 Multilayer perceptron (MLP) . . . 173
17.3.1 Architecture . . . 173
17.3.2 Learning : error backpropagation . . . 174
17.3.3 Universal approximator . . . 177
17.3.4 The need for using deep networks . . . 178
17.4 Generalization . . . 178
17.4.1 Regularization . . . 178
17.4.2 Learning procedure . . . 180
17.5 Optimization . . . 182
17.6 Convolutional neural networks : an early successful deep neural network . . . 182
17.7 Autoencoders . . . 183
17.8 Where is the problem with a deep neural network and how to alleviate it ? . . . 184
17.9 Success stories of Deep neural networks . . . 185

18 Recurrent neural networks **189**
18.1 Dealing with temporal data using feedforward networks . . . 189
18.2 General recurrent neural network (RNN) . . . 190
18.2.1 Architecture . . . 190
18.2.2 Real Time Recurrent Learning (RTRL) . . . 191
18.2.3 Backpropagation Through Time (BPTT) . . . 192
18.2.4 What about the initial state ? . . . 192
18.3 Echo state networks . . . 193
18.4 Long Short Term memory (LSTM) . . . 194
18.4.1 Architecture . . . 194
18.4.2 Example of applications . . . 196

19 Energy based models **199**
19.1 Hopfield neural networks . . . 199
19.1.1 Definition . . . 199
19.1.2 Example . . . 200
19.1.3 Training . . . 200
19.2 Restricted Boltzmann Machines . . . 201
19.2.1 RBM with binary units . . . 202
19.2.2 Training . . . 204

VI Ensemble methods **207**

20 Introduction **209**
20.1 Decision trees . . . 209
20.1.1 Basic idea . . . 209
20.1.2 Building regression trees . . . 211
20.1.3 Building classification trees . . . 212
20.1.4 More on trees . . . 213
20.2 Overview . . . 213

21 Bagging **215**
21.1 Bootstrap aggregating . . . 215
21.2 Random forests . . . 216
21.3 Extremely randomized trees . . . 217

22 Boosting **219**
22.1 AdaBoost . . . 219
22.1.1 Weighted binary classification . . . 219
22.1.2 The AdaBoost algorithm . . . 220
22.2 Derivation and partial analysis . . . 221
22.2.1 Forward stagewise additive modeling . . . 221
22.2.2 Bounding the empirical risk . . . 223
22.3 Restricted functional gradient descent . . . 224

VII Sequential Decision Making **229**

23 Bandits **231**
23.1 The stochastic bandit problem . . . 231
23.2 Optimism in the face of uncertainty . . . 232
23.3 The UCB strategy . . . 236
23.4 More on bandits . . . 238

24 Reinforcement learning **239**
24.1 Introduction . . . 239
24.2 Formalism . . . 240
24.2.1 Markov Decision Process . . . 240
24.2.2 Policy and value function . . . 241
24.2.3 Bellman operators . . . 241
24.3 Dynamic Programming . . . 243
24.3.1 Linear programming . . . 243
24.3.2 Value iteration . . . 244
24.3.3 Policy iteration . . . 245
24.4 Approximate Dynamic Programming . . . 246
24.4.1 State-action value function . . . 247
24.4.2 Approximate value iteration . . . 249
24.4.3 Approximate policy iteration . . . 251
24.5 Online learning . . . 255
24.5.1 SARSA and Q-learning . . . 255
24.5.2 The exploration-exploitation dilemma . . . 257
24.6 Policy search and actor-critic methods . . . 258
24.6.1 The policy gradient theorem . . . 259
24.6.2 Actor-critic methods . . . 260

Part I

Overview

Chapter 1

Introduction

Endowing machines with the ability of *learning* things may sound excessive or even inappropriate, since the common meaning of learning is a cognitive process exhibited by humans or animals. Learning implies being able to reuse in the future what has been learned in the past, requiring some memory capacities. Before the age of computers, memory was also a concept rather used for humans or animals. From the point of view of physicists, any system that keeps trace of its history is obviously endowed with memory and the process leading to the engram of that trace could naturally be called learning. Would we say that the solid matter inside of a boiled egg is the memory of the cooking stage ? Was the boiling itself a learning process ?

In this document, we will not enter into a semiotic analysis of learning. Instead, let us sketch out what learning is in the so-called "machine learning" field. This first part aims at outlining that field, highlighting the overall structure of a domain which has increased significantly during last decades.

1.1 Datasets

One intends to use machine learning techniques when s/he has to cope with data. Extracting information from a dataset in order to process further some new data that was not in the initial dataset can be thought as the core of machine learning. Every processing coming out of a machine learning approach is fundamentally data driven. Indeed, if the processing could have been designed from a specific knowledge about what it should do, then this knowledge could have been used for designing the process without any help of machine learning. For example, nobody uses collections of moon, earth and sun positions to predict eclipses, since the prediction process can be set up from Newton's laws. As opposed to this, there is not known algorithm (i.e. law) telling how to recognize which hand written digits are written in the zip-code area of a letter. For such problems, learning is to be considered, trying to set up a process that is grounded on a huge amount of data (zip-code scan images and their transcription into digits) to perform its recognition task.

1.1.1 Data sampling

The model

Let $\mathcal{Z}$ stands for the set where data are taken from. For example, let us consider the students of the Machine Learning class, represented by their weight in kilograms and height in centimeters. The data for this case are taken in $\mathcal{Z} \stackrel{\text{def}}{=} [20, 160] \times [50, 250]$ where the datum $z = (\texttt{weight}, \texttt{height}) \in \mathcal{Z}$ represents a specific student. In machine learning, data are considered to be distributed according to some *unknown* distribution. In other words, let us consider a random variable Z taking values in $\mathcal{Z}$ whose probability distribution is P_Z. Let us denote by p_Z the corresponding probability density, that is supposed to exist. In other words, for $A \subset \mathcal{Z}$,

$$\mathrm{P}\left(Z \in A\right) \stackrel{\text{def}}{=} \mathrm{P}_Z\left(A\right) \stackrel{\text{def}}{=} \int_A \mathrm{p}_Z\left(z\right) \mathrm{d}z$$

Taking a realization z of the random variable Z is denoted by $z \leftarrow\!\!- \mathrm{P}_Z$ in the following.

The random variable Z drives the sampling of a *dataset* $S = \{z_1, \cdots, z_i, \cdots, z_N\}$ such as $\forall i, z_i \leftarrow\!\!- \mathrm{P}_Z$, i.e the z_i are *independent and identically distributed*[1] (*i.i.d*). In the fake case of the students of the Machine Learning class, the dataset depicted in figure 1.1 is considered for further examples.

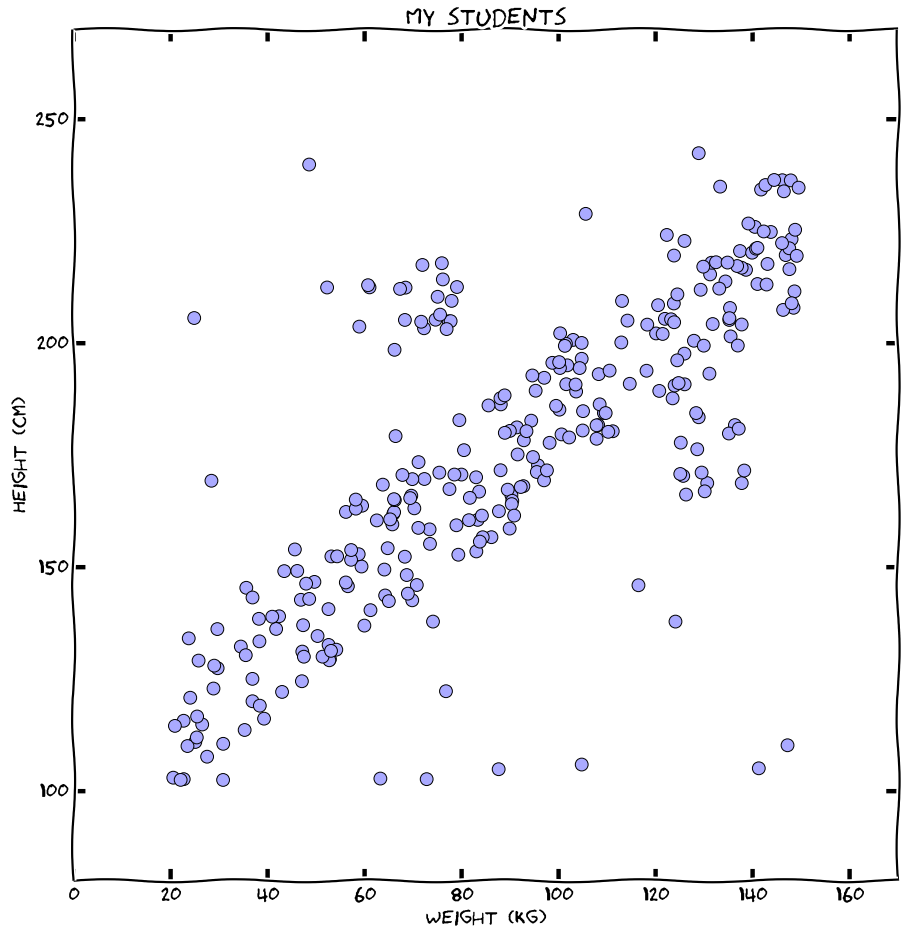


Figure 1.1: The dataset S of the weights and heights of the students attending the Machine Learning class (fake data). $|S| = N = 300$.

Sometimes, the generation of data (i.e. P_Z) comes with a supplementary process, that associates a label to the data. This supplementary labeling process is called the *oracle*, and it is defined as a conditional distribution. In this case, let us rename the data into inputs, renaming $\mathcal{Z}, z, Z, \mathrm{P}_Z$ into $\mathcal{X}, x, X, \mathrm{P}_X$. The label (or output) $y \in \mathcal{Y}$ given by the oracle to some input x results from a stochastic process as well. In other words, the oracle is defined as a conditional distribution $\mathrm{P}(Y \mid X)$, which is *unknown* as well. In this case, datasets are made of (x, y) pairs, sampled according to P_X and the oracle, i.e.

$$\begin{aligned} z &\overset{\text{def}}{=} (x, y) \\ \mathcal{Z} &\overset{\text{def}}{=} \mathcal{X} \times \mathcal{Y} \\ \mathrm{p}_Z(x, y) &\overset{\text{def}}{=} \mathrm{p}_{Y|X=x}(y)\mathrm{p}_X(x) \end{aligned} \tag{1.1}$$

From a computational point of view, the dataset S is as if it had been generated by algorithm 1. In the example of the Machine Learning class students, let us now add a supplementary Boolean

[1] In the theory of probabilities, instead of dealing with sample tossed from a probability distribution, a single random variable made of N identical and independent variables is rather considered, which is more rigorous than considering samples as done here.

attribute to each student, telling wether s/he belongs to the University Wrestling Team. Our dataset is now made of $((w, h), b)$, with (w, h) the weight and height and b set to `true` if the student belongs to the Wrestling Team. Such dataset is represented in figure 1.2. One can see that the sample distribution is the same as in figure 1.1, but some labels are added. Dealing with such data in the context of machine learning makes the assumption that the dataset can be obtained from algorithm 1, i.e. that whether a student belongs or not to the wrestling team can be somehow deduced from its weight and height.

Algorithm 1 `MakeDataSet`

$S = \emptyset$
for $i = 1$ **to** N **do**
  $x \leftarrow\!\!- \mathrm{P}_X$ // Toss the input.
  $y \leftarrow\!\!- \mathrm{P}_{Y \mid X=x}$ // Thanks to the oracle, toss the output from x.
  $S \leftarrow S \cup \{(x, y)\}$
end for
return S

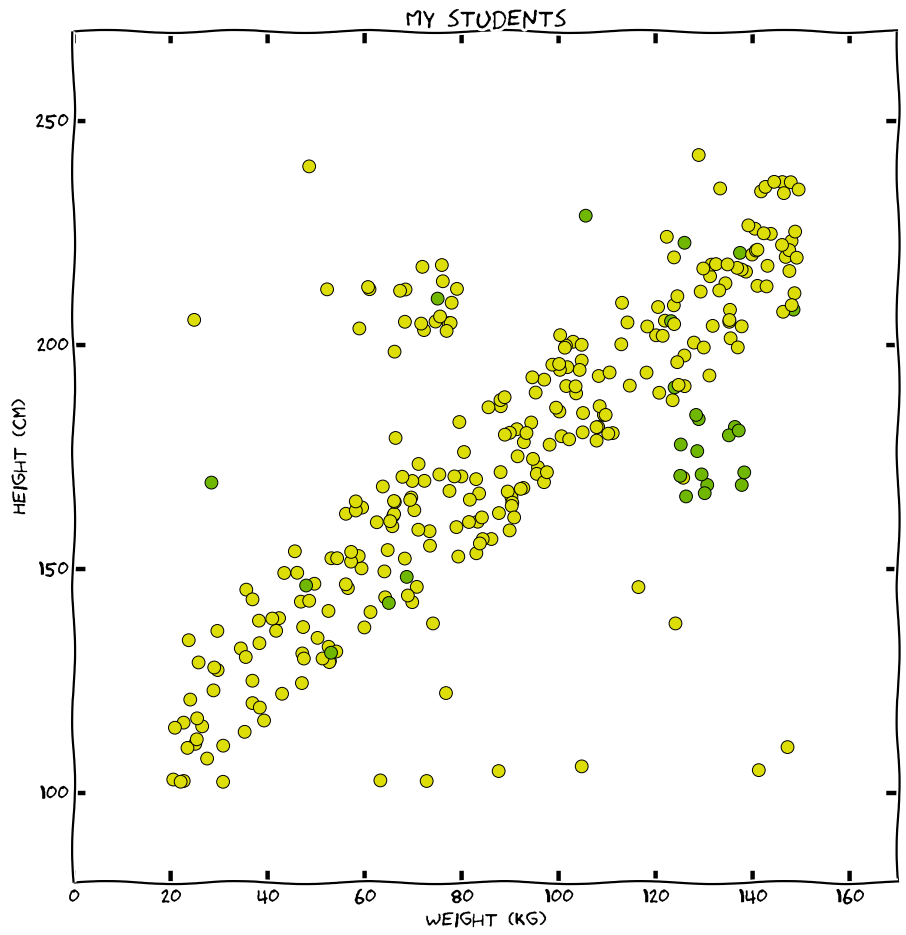


Figure 1.2: Weights and heights of the students attending the Machine Learning class. The ones belonging to the University Wrestling Team are plotted in green.

Ways of sampling

The way the data feed the learning process is important in practical applications of machine learning techniques. Indeed, some of them are restricted to a specific sampling strategy. Let us

overview the strategies here.

First strategy is *batch sampling*. It consists in providing the learning process with a batch of data, sampled once. The learning process will compute from this given dataset. The second strategy is *online sampling*. This consists is feeding the learning process with a continuous flow of successive data samples. In this case, each piece of data contributes to update the learning process. As opposed to the batch case, where one has to wait for the end of the dataset processing, an online learning process is therefore anytime. Some hybrid methods also exist. They are fed with a contunous flow of usually small datasets. The learning process then updates as each small dataset is submitted. Such datasets are sometimes referred to as *mini-batch*, and the update is called an *epoch*.

For the previously mentioned ways of sampling, the sampling was strictly the sampling of the random variable Z (i.i.d). When the data contains labels, the labelling process may be very expensive. In the case of skin cancer diagnostic for example, selecting a random picture of the back of some patient (i.e. input x) is easy, but labelling requires to ask an medical expert to analyze the picture in order to set the appropriate label y (cancer or not). In this case, one may have to select the pictures before submitting them to the expert. Indeed, if some picture is very similar to a previously labelled picture, the cost of asking the expert may not be worth it. This sample selection is called *active sampling*. It brakes the i.i.d nature of the samples[2], while i.i.d sampling is often required for theoretical convergence guaranties.

1.1.2 Data conditionning

Using machine learning for real problems requires to be able to find the suitable learning technique, but also to preprocess the data.

> *Data preprocessing in most cases is actually the place for introducing problem specific knowledge, as dedicated signal processing, suitable metrics, etc.*

Each datum z is indeed a set of attributes, i.e a set of key-value pairs. In the case of our students in figure 1.2, there are three attributes : `weight`, `height` and `is_wrestler`. Each datum is a specific values for *all* the attributes.

In practical cases, some values may be missing for some attributes, which may raise problems. Values may be also over-represented. In the example of the wrestler students in figure 1.2, as few of them actually belong to the wrestling team, considering blindly that a student do not belong to the team may not be a so bad prediction. To avoid this, one may try to balance the dataset, picking samples such that there is an equal amount of wrestler and not wrestler students. This may be done carefully, since the dataset is no more sampled according to algorithm 1, as discussed further in section 2.2.3.

The type of the attributes have to be considered carefully. Some attributes are numerical. This is the case for the weight and height in our example. Some other are categorial. This is the case for the `is_wrestler`. Some `nationality` attribute is also categorial, since there is no order between nationality values, they cannot be added, etc.

> *Many machine learning algorithms handle vectors, i.e. a set of scalar attributes. Representing categorial attribute values with numbers, for example using* $\texttt{USA} = 1, \texttt{Belgium} = 2, \texttt{France} = 3, \cdots$ *for* `nationality` *attribute, induces scalar operation on the values, whose semantics may be silly. In the given example,* $\texttt{France} > \texttt{USA}$ *and* `Belgium` *is an intermediate value between* `France` *and* `USA`.

Another problem that araises with vectorial data is the scales of the different attributes. In our student example, if weights were given in milligrams and heights in meter, the first dimension of the vectorial data (the weight) would live in a $[20000, 160000]$ range, while the second (the height) would live in a $[0, 3]$ range. With such scales, the point cloud of figure 1.1 would be a flat

[2] Indeed, they are still independent but the distribution changes.

horizontal line. In other words, the used learning algorithm would consider height as a constant, fluctuating slightly from one data sample to the other[3]. To avoid this, attribute values may be rescaled (usually *standardization*[4] is performed).

The high number of attributes in some data may also lead to an increase of the computation time, as well as a lack of generalization performance. To avoid this, some variable selection methods exist, that select, from all the attributes, a subset that appears to be useful for the learning. This is detailed in section 6.2.1.

Last, let us raise a warning. Be aware of the attribute meaning. For example, if in the student dataset, all wrestler are registered first, and if the line number in the datafile is an attribute, comparing this line number to the appropriate threshold (that is the number of wrestlers) leads to a perfect, but silly, learning. Such silly case may occur since toolboxes for machine learning usually provide scripts that take the data file as inputs and may use all attributes for doing their job. Be sure that datafiles do not need cleaning before feeding the toolbox.

1.2 Different learning problems...

Let us overview in this section the main classes of learning problems. Indeed, identifying the right nature of the problem to be learned drives the whole design of a learning procedure.

1.2.1 Unsupervised learning

Unsupervised learning consists in handling data that are not labelled. In other words, samples z are only tossed from a distribution P_Z, as in figure 1.1, and no oracle is involved in the process. With such a paradigm, the only thing that can be done is somehow describing the distribution.

Membership test

Relying on the samples in the dataset S, one can determine whether a point belongs or not to the distribution of the samples[5]. A membership test for a point could for example consist in finding the two closest samples to the tested point, measure the two distances from the point and the selected sample, average these distances, and compare the result to some threshold. Figure 1.3 shows the result. Once computed, the membership function can be used to say that a student with a 120cm height and a 100kg weight is very unlikely to be one of my students[6].

Identification of components

Another way to extract information from a distribution of samples is to recognize some components. For example, let us consider in our case lines as elementary components. One can describe the data as the main fitting lines, as in figure 1.4. Projecting high dimensional data to few lines, that are adjusted according to the distribution, may allow to express the high dimensional original data in a reduced space while preserving its variability.

Clustering

Last, some algorithms enable to represent the samples as a set of clusters, as in figure 1.5. This may help to set up an a posteriori identification of groups inside the data. Analyzing the groups/clusters in figure 1.5 may lead me to guess that there are three kind of students in my class. The examination of the small cluster on the right suggests that some students form a specific group. I may investigate on this... and discover that there are wrestlers in the classroom.

[3] e.g, the euclidian distance order of magnitude between two samples is mainly due to their weight difference.
[4] Rescaled such as mean is 0 and variance is 1.
[5] Is it likely that the point could have belonged to some dataset generated with the unknown P_Z ?
[6] I cannot know why. Do such students actually exist ? If yes, why do they not attend my lecture ? Keep in mind that the data is fake, and so are the conclusions we get from it.

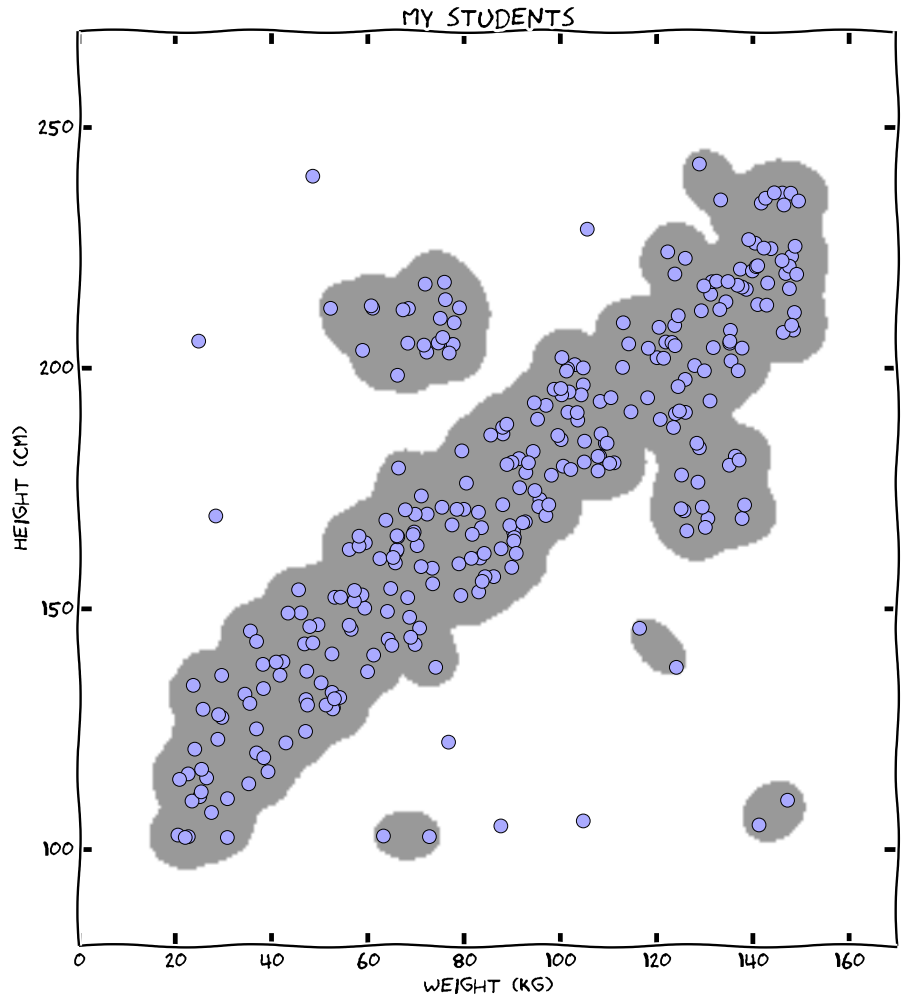


Figure 1.3: Weights and heights of the students attending the Machine Learning class. All 2D points are submitted to the membership test function described in the text (with threshold 7). The gray areas contain the points which passed the test.

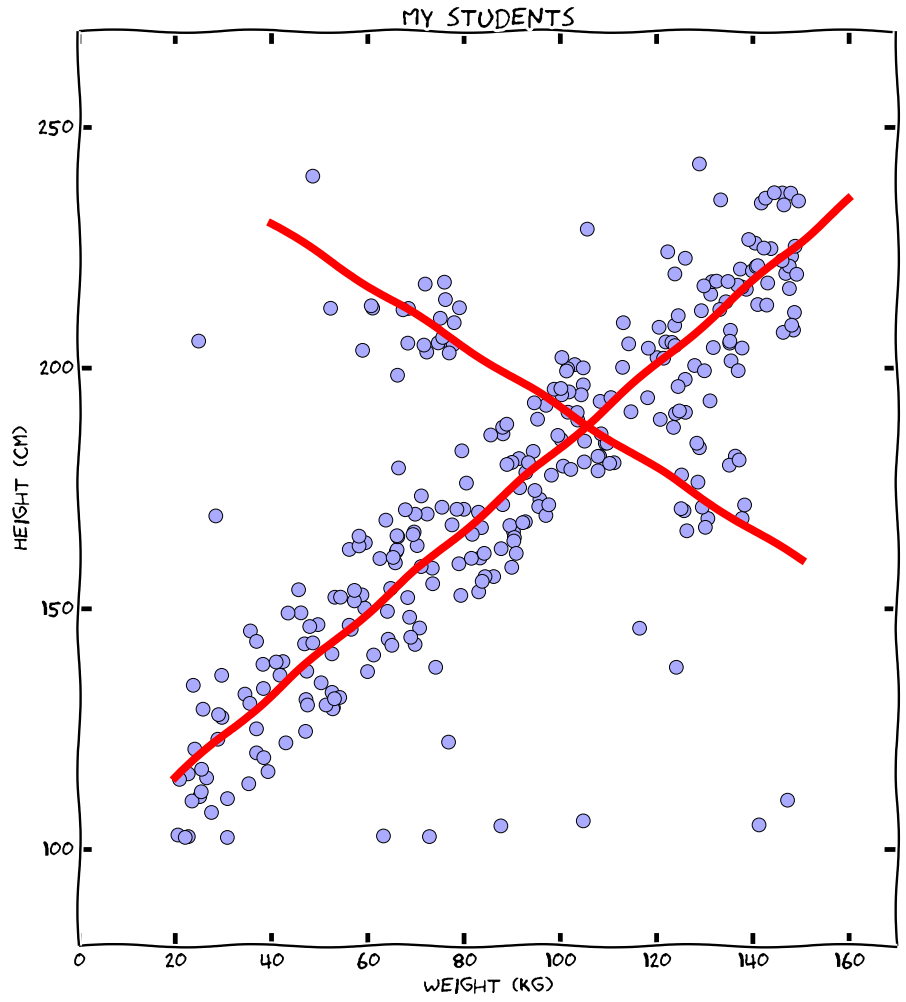


Figure 1.4: Weights and heights of the students attending the Machine Learning class, linear component analysis.

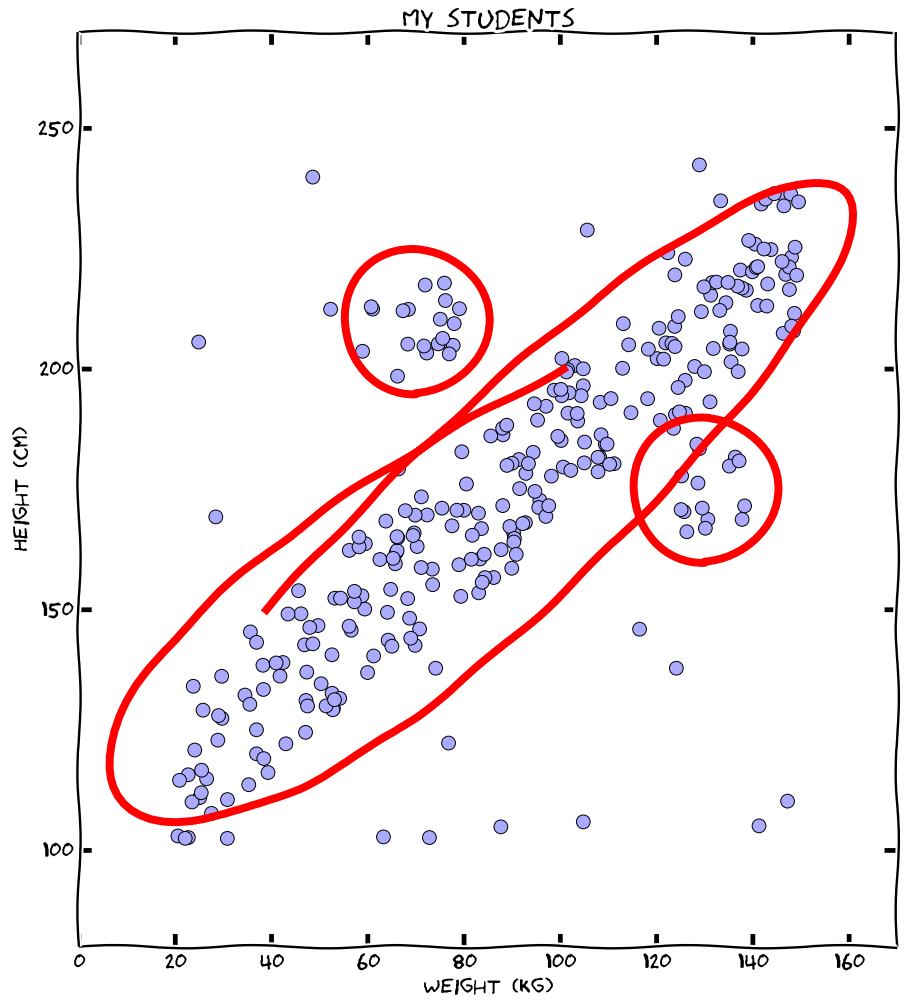


Figure 1.5: Weights and heights of the students attending the Machine Learning class, clustering.

1.2.2 Supervised learning

Supervised learning denotes learning problems where the data consists of an input and an output (also called a label). Each sample z is thus a pair (x, y), where y is supposed to be deductible from x by a process that needs to be set up by learning. This is what figure 1.2 illustrates. Let us recall[7] the notation $z \in \mathcal{Z}$, with $\mathcal{Z} = \mathcal{X} \times \mathcal{Y}$ here.

Bi-class classification

The most basic, but commonly addressed, supervised learning problem is two-class *classification*. A supervised learning problem is a *two-class classification* problem when $|\mathcal{Y}| = 2$, as in figure 1.2 where $\mathcal{Y} = \{\texttt{wrestler}, \texttt{non_wrestler}\}$. Learning consists in setting up a labelling process that tells wether a student is a wrestler or not, according to its weight and height. In other word, the learned labelling process should predict the label y of a incoming new data (x, y) from x only. A very classical example is the linear classifier. It consists of an hyperplane. A hyperplane splits the space into two regions. The linear classifier assigns one label to one region, and the other label to the other region. The learning consists in finding a suitable hyperplane, as in figure 1.6. When a new students enter the machine learning class, if the hyperplane is well placed[8], one can guess whether that student is a wrestler or not, just by identifying in which region the student is.

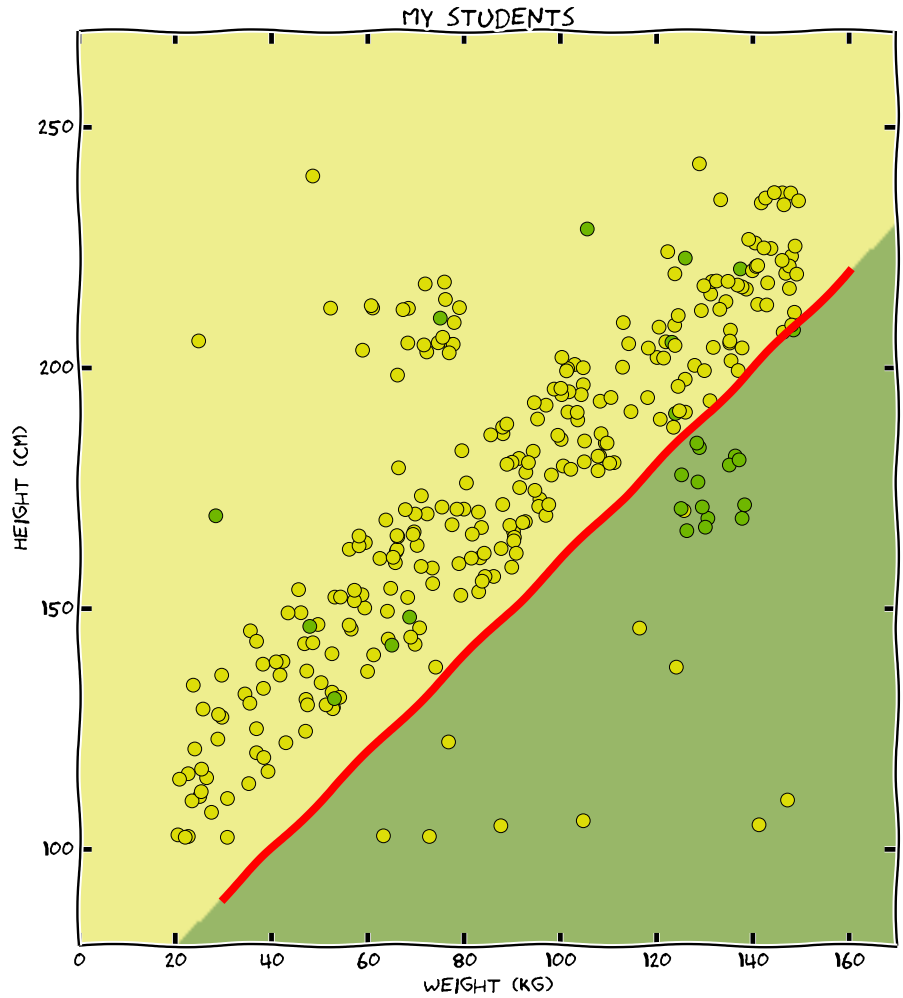


Figure 1.6: Weights and heights of the students attending the Machine Learning class, linear separation.

[7] See section 1.1.1.
[8] This is not always possible...

The case of the linear separation in figure 1.6 allows to introduce the concept of classification *score*. Even if the output has only two values, the hyperplane is defined by $ax + by + c = 0$. So for any (x, y) data, once a, b, c have been determined by learning, one can compute the scalar $s = ax + by + c$ and decide, according to the sign of s, in which region, green or yellow, the input (x, y) lives. In this case, the binary decision for labelling relies on the previous computation of a scalar. The higher s is, the more distant to the line (x, y) is. So a highly positive s means that the classifier says "strongly" that the student is a wrestler. In general, there are some classifiers that add a score to the label they produce. This can be a statistical score, or a geometrical score as for the hyperplane used in our example.

Multi-class classification

A *multi-class classification* problem is a problem where $\mathcal{Y}$ is finite (i.e. $|\mathcal{Y}| < \infty$). In the case of the students, the label could have been the name of the sport they practice, instead of only telling whether they practice wrestling or not. Even if some multi-class learning algorithms exist, there are also multi-class algorithms that combine two-class algorithms. This can be achieved by *one-versus-all* (or *one-versus-rest*) strategy (see algorithm 2 for learning, and then algorithm 3 for labelling a new input) or *one-versus-one* strategy (see algorithms 4 and 5). The former requires scores, as opposed to the latter. The latter requires more computation.

Algorithm 2 `One_vs_All`(learner, S)

classifiers $\leftarrow \emptyset$
for $y \in \mathcal{Y}$ **do**
  $S' \leftarrow \{(x, \mathbb{1}_{y'=y}) \mid (x, y') \in S\}$
  c $\leftarrow$ learner (S') // learner is a two-class learning algorithm, thus c is a two-class classifier.
  classifiers $\leftarrow$ classifiers $\cup \{(y, \text{c})\}$ // Let us store the classifier.
  // $s = \text{c}(x)$ is the score of x. The higher s is, the more confident is c in saying that x should be labelled with y.
end for
return classifiers

Algorithm 3 `One_vs_All_classifier`(classifiers, x)

$(y^\star, \text{c}^\star) \leftarrow \underset{(y,\text{c})\in\text{classifiers}}{\text{argmax}} \ \text{c}(x)$
return $y^\star$

Algorithm 4 `One_vs_One`(learner, S)

$L \leftarrow \{\{y, y'\} \mid (y, y') \in \mathcal{Y} \times \mathcal{Y},\ y \neq y'\}$
classifiers $\leftarrow \emptyset$
for $\{y, y'\} \in L$ **do**
  $S' \leftarrow \{(x, y'') \in S \mid y'' \in \{y, y'\}\}$ // We extract from S data with y or y' label.
  c $\leftarrow$ learner (S') // learner is a two-class learning algorithm, thus c is a two-class classifier.
  classifiers $\leftarrow$ classifiers $\cup \{\text{c}\}$
end for
return classifiers

Regression

A *regression* problem is a supervised learning problem where labels are continuous, as opposed to classification. The standard case is $\mathcal{Y} = \mathbb{R}$. For example, if the dataset of the students attending the machine learning class contains their weights and heights, as well as a Boolean label telling whether a student is over-weighted or not, and if this Boolean is the label to be predicted from weight and

Algorithm 5 `One_vs_One_classifier`(classifiers, x)

1: **return** $\underset{y\in\mathcal{Y}}{\text{argmax}} \sum_{c\in\text{classifiers}} \mathbb{1}_{c(x)=y}$ // $\sum\cdots$ is the number of classifiers that voted for class y.

height, the supervised learning problem is a classification problem, as presented previously. If now the BMI[9] of the student is given instead of the overweight flag, and if this index value has to be predicted from weight and height, the supervised learning problem becomes a regression.

The distinction between classification and regression may sound artificial, since it only denotes that $\mathcal{Y}$ is finite or continuous. From a mathematical point of view, such a distinction is irrelevant. Nevertheless, supervised learning methods are often dedicated to either regression or classification, which justifies the distinction between the two.

Last, let us mention the case where $\mathcal{Y} = \mathbb{R}^n$. This can be solved with n scalar regression problems, one for predicting each components of the label. When this is applied, each component prediction is learned separately from the others, whereas they may not be independent. Some methods like neural networks[10] handle multidimensional prediction as a whole, thus exploiting the eventual relations between the dimensions.

1.2.3 Semi-supervised learning

In some supervised learning problems, getting the label is expensive. This occurs when a human expert has to be hired for the labelling, or when the labelling requires a large amount of time. The consequence of this is that many labels are missing in the data. This is called a *semi-supervised learning* problem. Such data is illustrated in figure 1.7, taking the example of the students attending the machine learning class.

Considering semi-supervised learning makes the assumption that the labelling process is smooth: close inputs are likely to be labelled identically. In the figure 1.7, since green (wrestlers) dots are localized in the same region, I may infer that all students in that region are wrestlers. If I had to call one of the unlabelled students to ask him/her whether s/he a wrestler or not, I would rather choose to call students who lie at the border of the regions. If I can afford only two phone calls, I would call the two thickened students on the figure. Choosing actively which examples have to be labeled first is referred to as *active learning* in the domain.

1.2.4 Reinforcement learning

Until now, we have addressed learning paradigms for which data is given for being analyzed. The *reinforcement learning* paradigm rather concern control problems, and is thus very close to *optimal control*. Control is usually rather related to automation, and one may wonder why such concept is addressed in a machine learning context. Indeed, reinforcement learning addresses problems where the dynamics of the controlled system is unknown to the controller, as opposed to automation where the equations of that dynamics are given. Let us introduce reinforcement learning with an example.

A control problem

Let us play the famous weakest link game, used in the weakest link TV show. Here, a single player is considered. The player tries to get the highest amount of money in a delimited time. The game starts with a 20$ question. The player needs to answer it right to reach the next question. The next question is a 50$ one. If the player answers it right, s/he is asked the next question, which is 100$. The values for the questions are 20, 50, 100, 200, 300, 450, 600, 800 and 1000$. Each time the user reaches a new question, s/he has two options. First option is to try to answer it and go to next stage. Second option is to say "bank". In that case, the amount of money associated to the last question is actually won, and the game restarts from first stage.

[9] Body Mass Index
[10] Multi-layered perceptrons indeed.

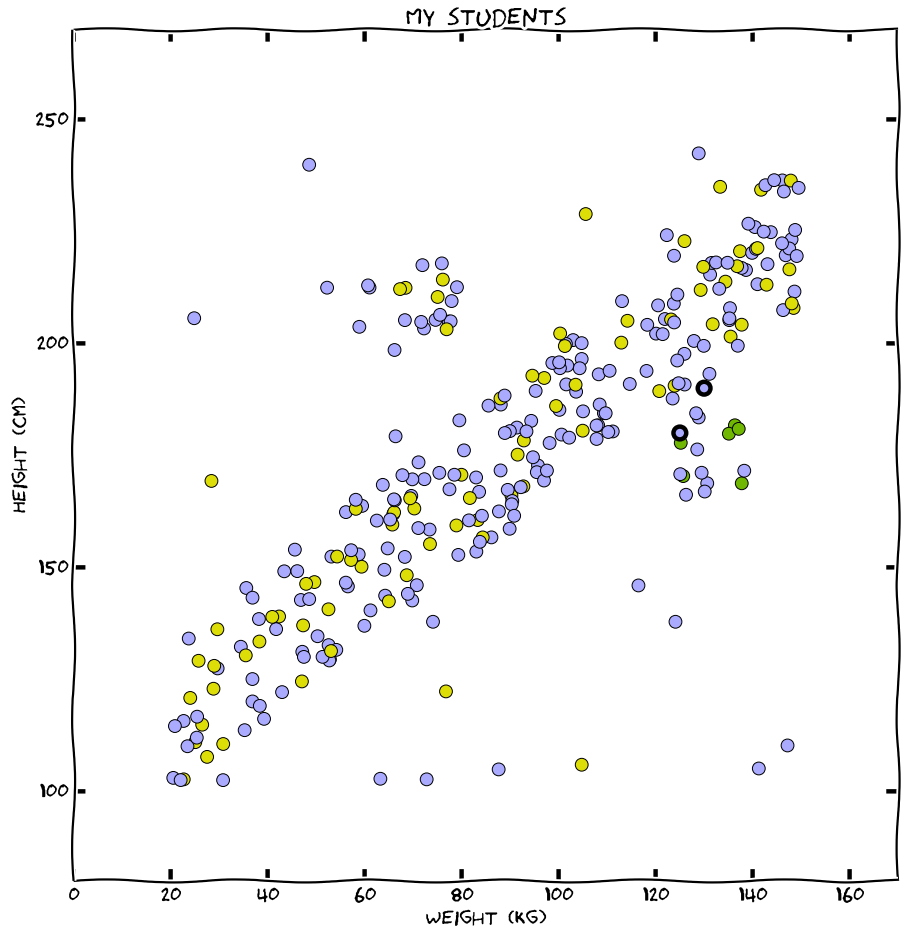


Figure 1.7: Weights and heights of the students attending the Machine Learning class. Blue dots are unlabelled data. Yellow dots are students that are known to be non-wrestlers, whereas green dots are those who are known to be wrestlers.

Let us model the game with a *Markovian Decision Process* (MDP). First element of the MDP is a state space S. Here $S = \{q_0, \cdots, q_i, \cdots, q_9\}$, i.e. the nine question levels of the game and the initial state. Second, let us denote by A the set of actions. Here $A = \{\text{answer}, \text{bank}\}$. Third is the transition matrix[11] T, defined such as $T^a_{s,s'}$ is the probability of reaching state s' from state s when performing action a. Last element is a reward matrix R, defined such as $R^a_{s,s'}$ is the reward expectation when the transition s, a, s' occurs.

For the weakest link game, the modeling is quite easy. Let us suppose that the probability of answering a question right is p. Reward is deterministic here. The following stands.

- $\forall s \in S,\ \forall s' \in S \setminus \{q_0\},\ T^{\text{bank}}_{s,s'} = 0$ and $T^{\text{bank}}_{s,q_0} = 1$. Banking makes a transition to q_0.
- $\forall i \in [0..8],\ T^{\text{answer}}_{q_i,q_{i+1}} = p$ and $T^{\text{answer}}_{q_i,q_0} = 1 - p$. Moreover, $T^{\text{answer}}_{q_9,q_0} = 1$, and other transition probabilities are null.
- $R^{\text{bank}}_{q_1,q_0} = 20$, $R^{\text{bank}}_{q_2,q_0} = 50$, $R^{\text{bank}}_{q_3,q_0} = 100$, $R^{\text{bank}}_{q_4,q_0} = 200$, $R^{\text{bank}}_{q_5,q_0} = 300$, $R^{\text{bank}}_{q_6,q_0} = 450$, $R^{\text{bank}}_{q_7,q_0} = 600$, $R^{\text{bank}}_{q_8,q_0} = 800$, $R^{\text{bank}}_{q_9,q_0} = 1000$ and $R^a_{s,s'} = 0$ otherwise. This is the reward profile.

Figure 1.8 illustrates the modeling of the game with a MDP. The purpose of reinforcement learning is to solve the MDP. It means finding a *policy* that allows the player to *accumulate* the maximal amount of reward. Such policy is called the *optimal policy*. A policy, in this context, is simply the function that tells for each state which action to do. It can be stochastic, but it can be shown that the optimal policy is indeed deterministic.

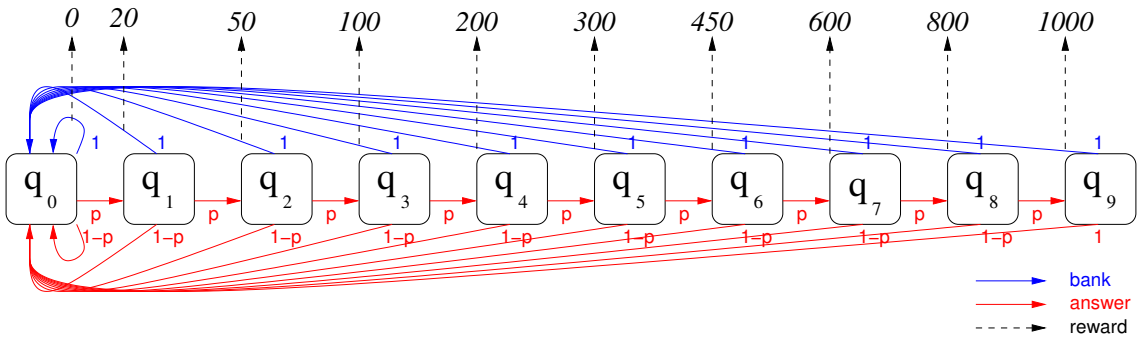


Figure 1.8: Modelling the weakest link game with a Markovian Decision Process. See text for details.

In the TV show, the users play during a fix duration, and they have to maximize their return (i.e. the sum of the money got when banking). In reinforcement learning, usually, no such duration is given. Time stress is modeled as a probability $1 - \gamma$, $\gamma \in [0, 1[$ to end the game at each action. This is not modelled directly in T and R. The optimal policy, that is what the reinforcement learning computes, take γ into account. If γ is high, one can hope reaching last states, and thus one may not bank for first questions. If γ is low, it is better to be Epicurian[12], i.e. to bank as soon as a question is answered correctly.

Resolution

Even for such a reduced problem, finding the right strategy is not obvious, even when the MDP (S, A, T, R, γ) is known. When everything is known, the problem of finding what to do in each state in order to accumulate the highest amount of reward (i.e. find the optimal policy) is addressed by optimal control, a field of automation. Reinforcement learning addresses this problem when T and R are unknown. To do so, the reinforcement learning methods often rely on inner supervized learning.

Figure 1.9 shows the optimal policy in different cases.

[11] It is a 3D tensor indeed....
[12] *carpe diem*

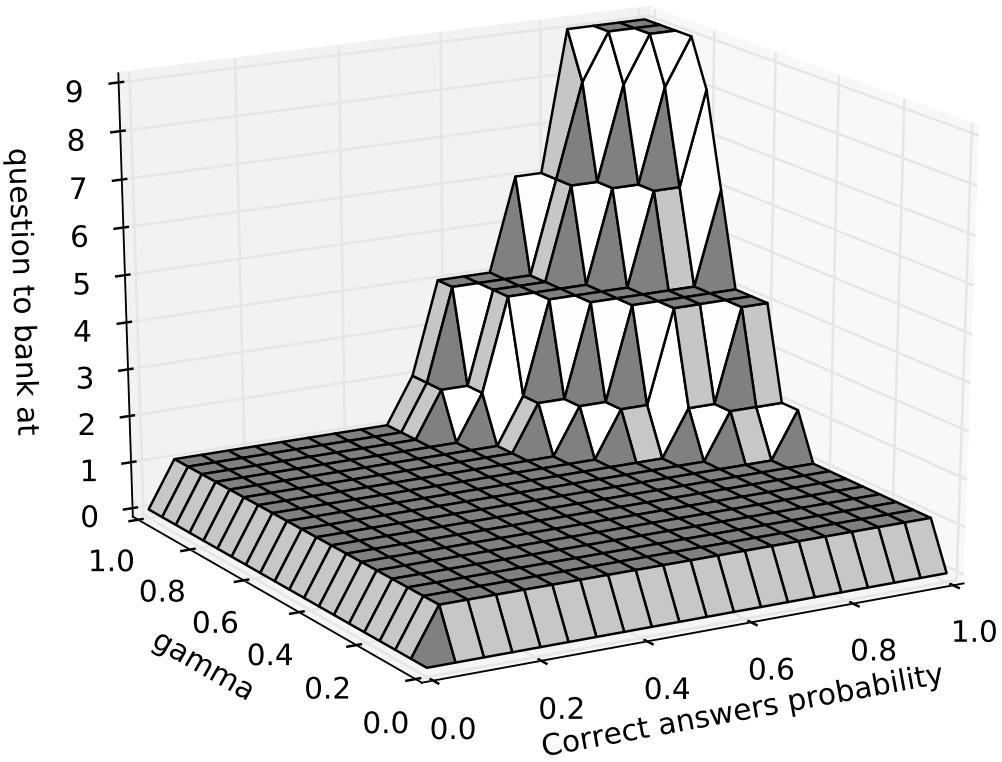


Figure 1.9: According to γ and p, the best policy consists in banking at a specific question level. This is what this plot illustrates. See text for details.

1.3 Different learning strategies

1.3.1 Inductive Learning

The main strategy for learning is to rely on the dataset for building up a predictor. Learning a rule from examples is called an induction. Therefore, computing a predictor from the data is an *inductive learning* process.

The way the predictor is induced is an *induction principle*. The principle that is mostly used is *empirical risk minimization* (ERM), as detailed further. Roughly speaking, it consists in finding a predictor that predicts best the labels in the dataset from their corresponding input. Will such a predictor be able to perform well on new data ? It would be nice that the answer is yes, meaning that the ERM principle has good *generalization* capabilities. Indeed, inductive principles that lead to predictors having poor generalization capabilities are useless, since the best they can do is to retrieve the labels of the data inputs while these labels are already known.

Controlling the generalization capabilities of inductive learning strategies is the core of the statistical analysis of main machine learning algorithms.

1.3.2 Transductive learning

The *transductive learning* strategy concerns supervized learning. Instead of learning a predictor from a dataset, transductive learning consists in embedding the dataset in the predictor. Then, when the predictor receives some inputs, it computes its label from the labels of the embedded dataset.

To sum up, setting up a predictor from the dataset, which is the learning stage, is nothing but storing the dataset. All the computation is done when a new data has to be labelled.

A famous example of transductive leaning in classification is the *k-nearest neighbours* algorithm (see algorithm 6), whose classification is depicted in figure 1.10.

Algorithm 6 `knn_predict`(k, S, x)

$C = \emptyset$.
for $i = 1$ **to** k **do**
  $(x^\star, y^\star) = \underset{(x',y')\in S}{\operatorname{argmin}} |x' - x|$ // Find the data with the input closest to x.
  $C \leftarrow C \cup \{(x^\star, y^\star)\}$
  $S \leftarrow S \setminus \{(x^\star, y^\star)\}$
end for
$y^\star = \underset{y\in\mathcal{Y}}{\operatorname{argmax}} \sum_{(x,y')\in C} \mathbb{1}_{y'=y}$ // $y^\star$ is the most frequent label in C.
return $y^\star$

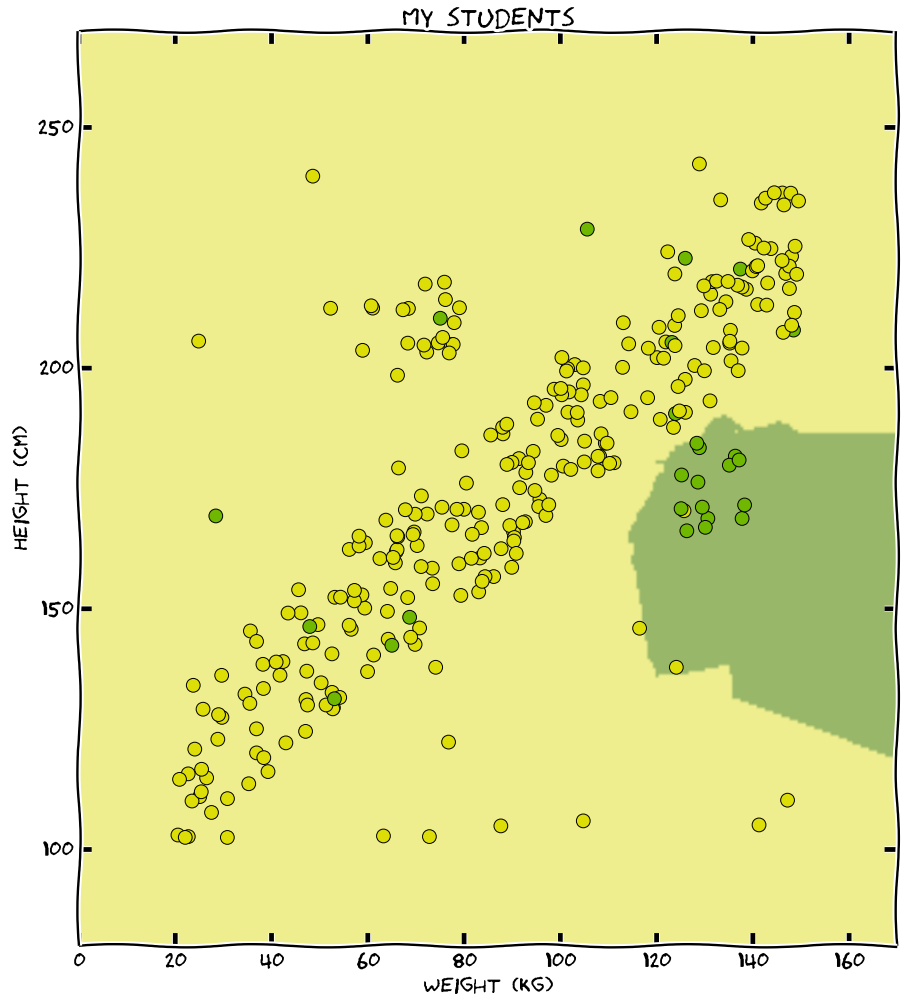


Figure 1.10: Weights and heights of the students attending the Machine Learning class. The colored areas correspond to the labels given by a KNN (with $k = 10$) relying on the dataset.

Chapter 2

The frequentist approach

The *frequentist approach* consists in relying on samples in order to estimate probabilities. It is opposed to the Bayesian approach where distributions of probabilities are rather manipulated (see chapter 3). Let us introduce here the main objects involved in machine learning when the frequentist approach is used. Supervised learning is mainly addressed here.

2.1 Hypothesis spaces

In the supervised learning case, inductive learning consists in building up a predictor, i.e a function $p \in \mathcal{Y}^{\mathcal{X}}$, that associates a label $y = p(x)$ to any input $x \in \mathcal{X}$. In practice, a particular algorithm is not able to search within the whole set of labelling functions $\mathcal{Y}^{\mathcal{X}}$, but only a subset $\mathcal{H} \subset \mathcal{Y}^{\mathcal{X}}$ of it, called the *hypothesis space*.

2.1.1 Parametric and nonparametric hypothesis spaces

Basically, hypothesis spaces are function sets. It is usual to define *parametric functions*, i.e. functions whose definition is related to some parameter. Let $\theta \in \Theta$ a parameter. Let $f \in \mathcal{Y}^{\Theta \times \mathcal{X}}$ a function such as $y = f(\theta, x)$. Considering θ as a parameter leads to define $f_\theta \in \mathcal{Y}^{\mathcal{X}}$ as $f_\theta(x) \stackrel{\text{def}}{=} f(\theta, x)$. The set $\mathcal{H} = \{f_\theta \mid \theta \in \Theta\} \subset \mathcal{Y}^{\mathcal{X}}$ is a *parametric hypothesis space* induced by f. Parametric hypothesis spaces are interesting since $\mathcal{H}$ is a "very small" subset of the full functional space $\mathcal{Y}^{\mathcal{X}}$. As a consequence, $\mathcal{H}$ can be explored conveniently thanks to an exploration of the parameter space Θ. This exploration enables to search for an optimal function within parametric hypothesis spaces, since it consists in searching some optimal $\theta^\star \in \Theta$. To do so, classical optimization techniques can be used, as the well-known gradient descent.

Even if parametric hypothesis spaces are commonly used in machine learning, some methods involve a *nonparametric hypothesis space*. In other words, $\mathcal{H} \subset \mathcal{Y}^{\mathcal{X}}$ can hardly be expressed as related to some parameter set Θ. For example, k-nearest neighbours (knn) predictors (see algorithm 6) have been presented in section 1.3.2. Each knn-predictor is parametrized by a set of samples. We could consider that Θ is the set of all finite subsets of $\mathcal{X} \times \mathcal{Y}$, which is very big, but usually, Θ is limited to $\mathbb{R}^n$. When Θ is so big, it is rather said that the hypothesis space is nonparametric. Another classical example of nonparametric hypothesis space is the set of all decision trees allowing to assign a label $y \in \mathcal{Y}$ to some input $x \in \mathcal{X}$.

2.1.2 The linear case

In machine learning, *linear functions* are commonly used, since many mathematical results are available for them. Linear functions are a prototypical case of parametric hypothesis space. Let us restrict here to *scalar linear functions* in $\mathbb{R}^n$. Here, Θ is $\mathbb{R}^n$ as well, since $f_\theta(x) \stackrel{\text{def}}{=} \theta^\intercal.x$. Using linear hypothesis spaces may appear restrictive, but there is a way to perform nonlinear computation with linear methods. This way consists in mapping the input set $\mathcal{X}$ to some other set Φ, thanks

to a *nonlinear* function $\varphi \in \Phi^{\mathcal{X}}$. In this case, $\Theta = \mathbb{R}^{\dim(\Phi)}$, $f_\theta(x) \stackrel{\text{def}}{=} \theta^{\mathsf{T}}.\varphi(x)$ and the hypothesis set $\mathcal{H} \stackrel{\text{def}}{=} \left\{ f_\theta \;\middle|\; \theta \in \mathbb{R}^{\dim(\Phi)} \right\} \subset \mathcal{Y}^{\mathcal{X}}$.

When such a projection is used, $\mathcal{X}$ is called the *ambient space* and Φ is called the *feature space*. Usually, $\dim(\Phi) \gg \dim(\mathcal{X})$, meaning that using the non linear projection φ consists in applying linear methods in high dimension in order to get an overall non-linear processing. This has dramatic consequences, as illustrated next.

2.1.3 Linear separability

Let us consider a bi-class supervised learning problem, with $\mathcal{X} = \mathbb{R}^n$ and $\mathcal{Y} = \{\bullet, \circ\}$. A linear separator in $\mathbb{R}^n$ is defined by $(\theta, b) \in \mathbb{R}^n \times \mathbb{R}$. It associates to some input $x \in \mathcal{X}$ the label $\circ$ when $\theta^{\mathsf{T}}.x + b \geq 0$, and $\bullet$ otherwise[1]. In other words, the associated label depends only on which side of the hyperplane $\theta^{\mathsf{T}}.x + b = 0$ the input x actually stands.

Let us now consider a dataset $S = \{(x_1, y_1), \cdots, (x_i, y_i), \cdots, (x_N, y_N)\} \subset \mathcal{X} \times \mathcal{Y}$. It is said to be *linearly separable* if some $(\theta, b) \in \mathbb{R}^n \times \mathbb{R}$ exists such as, for all the samples in S, the separator defined from (θ, b) gives the right label.

> *Linear separability is crucial in machine learning. Keep in mind that a collection of $n+1$ random points in $\mathbb{R}^n$, given random binary labels, is likely to be separable. In other words, a learning problem consisting in separating $n+1$ points in $\mathbb{R}^n$ is easy, and thus not interesting...*

Let us illustrate this in $\mathbb{R}^2$ with 3 points in figure 2.1. Of course, if all points were aligned, separation would have been impossible.

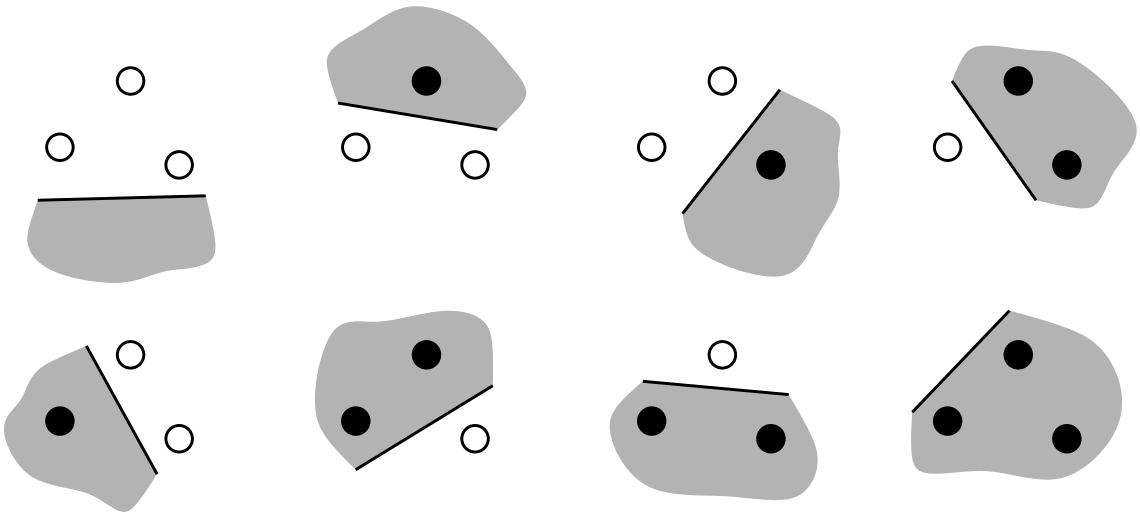


Figure 2.1: For theses points in $\mathbb{R}^2$, any labelling makes the obtained dataset linearly separable. This is true for most point configurations. This remarks can be extended to $n+1$ points in $\mathbb{R}^n$.

Let us now consider the labeled points S in figure 2.2-left. They are obviously not linearly separable. In this case, the trick is to project $\mathcal{X}$ into an ambient space Φ, as mentioned in previous section, thanks to some function φ. In order to name components, $x \in \mathcal{X} = \mathbb{R}^2$ is denoted by $\mathbf{x} = (\mathbf{x}^1, \mathbf{x}^2)$. Let us use

$$\varphi(\mathbf{x}) = \begin{pmatrix} \mathbf{x}^1 \\ \mathbf{x}^2 \\ (\mathbf{x}^1)^2 + (\mathbf{x}^2)^2 \end{pmatrix} \tag{2.1}$$

[1] If we denote $\theta' = (\theta, b) \in \mathbb{R}^{n+1}$ and $x' = (x, 1) \in \mathbb{R}^{n+1}$, the expression $\theta^{\mathsf{T}}.x + b$ rewrites as $\theta'^{\mathsf{T}}.x'$. This is usually done in order to avoid a specific formulation for the offset b in mathematical expressions.

Let us define $\varphi(S) = \{(\varphi(x), y) \mid (x, y) \in S\}$. Figure 2.2-middle shows that $\varphi(S)$ can be separated. The separation frontier in the ambient space $\mathcal{X}$ is obtained from a reverse projection of $\varphi(\mathcal{X}) \cap \{\theta^{\intercal}.x + b = 0\}$ in the feature space[2]. This is how φ enables to perform non-linear separation in the input space, while a linear separator is involved in the feature space.

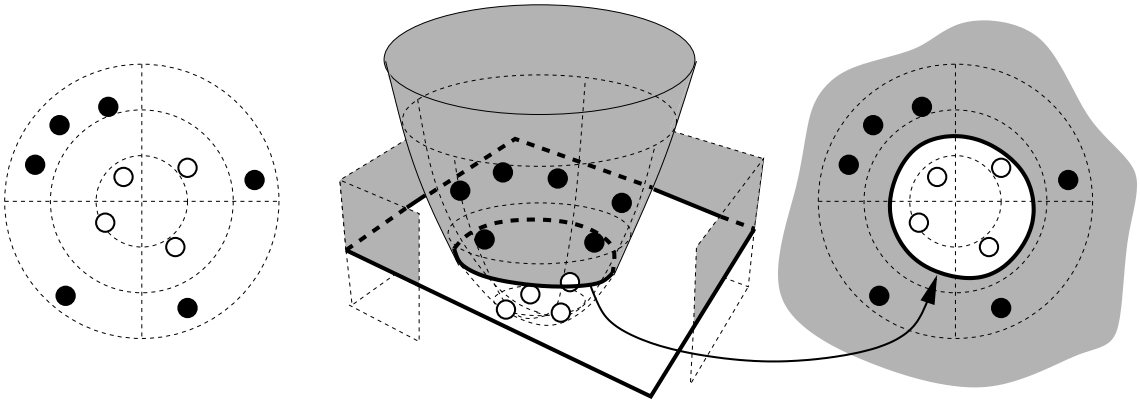


Figure 2.2: On the left, the labeling of 10 points in $\mathbb{R}^2$ is not linearly separable. The same points projected in $\mathbb{R}^3$ by φ, defined by equation (2.1), become separable by a hyperplane, as middle of the figure shows. The half-space over the hyperplane correspond to black labels. It is shaded. The paraboloid is the projection of the whole $\mathbb{R}^2$ in $\mathbb{R}^3$ by φ. The part of the paraboloid above the hyperplane is darken as well. It correspond to regions labeled as black. On the right, points are represented back into $\mathbb{R}^2$, as on the left, but the plane area which projects into the dark half-space in $\mathbb{R}^3$ is darken. It can be seen that the linear separation in $\mathbb{R}^3$ (the feature space) corresponds to a non-linear separation in $\mathbb{R}^2$ (the ambient space), since φ is a non-linear projection.

The trick of projecting a dataset into a high-dimension space, so that the projected points become linearly separable, seems powerful. Indeed, it has its drawback. In figure 2.2, the projection has been chosen carefully. This cannot been done in non artificial situation. In real cases, we would rather have built a dataset-agnostic high dimensional projection into $\mathbb{R}^n$, with $n \gg 9$ in order to make our 10 points very easily separable, and then set up a separator to make the decision. This cannot be represented, as opposed to what we did for the middle part of figure 2.2 where φ projects in $\mathbb{R}^3$. Nevertheless, the right part of figure 2.2 can still be sketched out, even if the feature space cannot be plotted. The figure that we will obtain may be like figure 2.3... The separation has poor generalization capabilities, and can hardly be used to predict the label of new incoming samples.

Once again, this drawback occurs since it is easy to separate points in high dimensions, and so separation becomes useless. This is referred to as the *curse of dimensionality*.

2.2 Risks

Let us consider here supervised learning and the definitions given in section 1.1.1. We would like to find some predictor (or hypothesis) $h \in \mathcal{H}$ such as the label $y = h(x)$ given by h to some input x is likely to be the label associated to that x if it is sampled by algorithm 1, i.e. if that x were labeled by the oracle. This is what a "good" predictor is supposed to do. The concept of *risk* aims at defining and measuring the quality of a predictor.

2.2.1 Loss functions

In order to evaluate predictors, a *loss function*, denoted by $L \in (\mathbb{R}^+)^{\mathcal{Y} \times \mathcal{Y}}$, needs to be defined for comparing the label returned by some predictor to the one it should have returned. For example, one can use the *binary loss* defined as

$$L(y, y') = \begin{cases} 1 & \text{if } y \neq y' \\ 0 & \text{otherwise} \end{cases} \tag{2.2}$$

[2] $(\theta, b) \in \mathbb{R}^3 \times \mathbb{R}$ here.

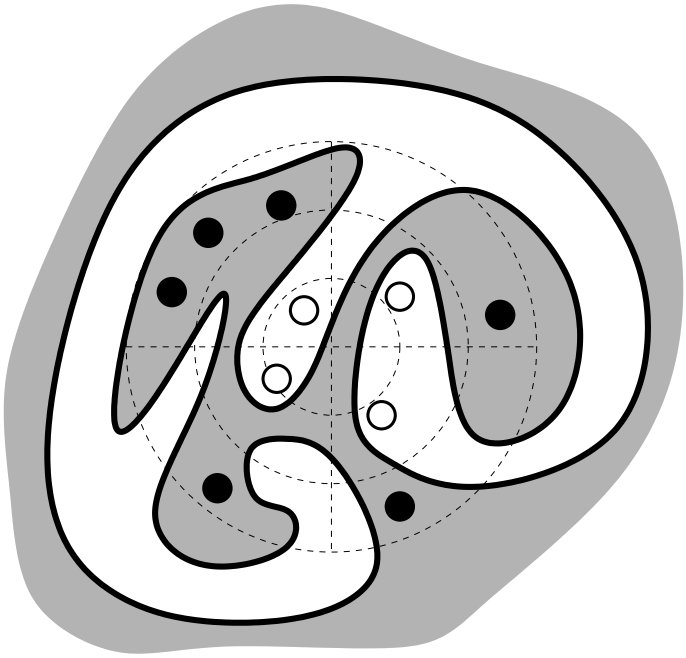


Figure 2.3: The curse of dimensionality. See text for detail.

The binary loss is suitable for finite label set $\mathcal{Y}$, i.e. classification, but it may be rough when $\mathcal{Y}$ is continuous, i.e. regression. When $\mathcal{Y} = \mathbb{R}$, one can use the *quadratic loss* defined as

$$L\left(y, y'\right) = \left(y - y'\right)^2$$

2.2.2 Real and empirical risks

Once some loss function is chosen, the idea is to measure the quality of some predictor $h \in \mathcal{H}$. If $(x, y) \dashleftarrow \mathrm{P}_Z$, according to equation (1.1) or algorithm 1 (page 13), we would like that $L\left(h\left(x\right), y\right) \approx 0$. It means that the predictor should behave as the oracle does, *for the inputs that are likely to be tossed*.

The *real risk* of h, denoted by $\mathcal{R}\left(h\right)$ is thus the expectation of the loss, for the data sampled according to algorithm 1, i.e.

$$\mathcal{R}\left(h\right) \overset{\text{def}}{=} \int_{\mathcal{X}\times\mathcal{Y}} L\left(h\left(x\right), y\right) \mathrm{p}_Z\left(z\right) \mathrm{d}z, \text{ with } z = (x, y)$$

Unfortunately, the real risk cannot be computed, since the oracle p_Z is unknown. Nevertheless, it can be estimated. The classical method for estimating an expectation is the computation of the average over a collection of samples. The dataset S actually contains samples tossed from P_Z, as algorithm 1 shows. The following average, called the *empirical risk* of h computed from S, denoted by $\mathcal{R}^S_{\text{emp}}\left(h\right)$ or $\mathcal{R}_N\left(h\right)$ $(N = |S|)$, thus estimates the real risk.

$$\mathcal{R}^S_{\text{emp}}\left(h\right) \overset{\text{def}}{=} \frac{1}{|S|} \sum_{(x,y)\in S} L\left(h\left(x\right), y\right)$$

The criterion that measures how well some hypothesis h imitates the labeling performed by the oracle is the real risk. It cannot be computed. The empirical risk may be an estimation of it. Is this estimation reliable ? Can we consider that a predictor which has a low empirical risk on some dataset has a low real risk in general ? This is the core question of supervised machine learning.

2.2.3 Are good predictors good ?

I have at my disposal a daily record seismographs in France, and I have the list of the days where an earthquake occurred in France. I want to predict from that data whether there will be an earthquake or not this afternoon, knowing the seismograph record of this morning.

Without any expertise in geophysics, I can predict that there will be no earthquake this afternoon... ignoring the seismograph data of the day. I behave as a constant function, always telling that no earthquake will occur. Both my empirical risk and real risk are very low, so I am a good earthquake predictor, from a statistical point of view. Making prediction errors on very infrequent inputs do not alter the real risk.

Another example of such good predictors is recognizing the digit 0 among randomly chosen hand-written digits. Once again, always answering that a digit is not a 0 leads to a 90% real risk (with the binary loss) if the digits are uniformly distributed over the ten digits.

> *A good predictor is a predictor that minimizes the real risk. Good predictors may be uninteresting, especially when the distribution of labels is strongly unbalanced.*

If I really want to build up an alert system for earthquakes, I would need to balance the dataset with 50% non-earthquake, taken randomly in the huge mass of non-earthquake days, and 50% earthquake data (made of all, but few, earthquake data I actually have). If there is only, on the balance dataset, 1% of cases for which I predict earthquake while it actually not occurs (false positive detection), this 1% may trigger frequent false alerts in the daily use of my detector. In other words, after the balancing process, the database has lost is statistical significance for a real life usage.

> *If data is very unbalanced, another available solution to the prediction problem is to collect the regular data, i.e. the seismographs of non-earthquake days here, and to learn a membership test by an unsupervized method, as introduced in section 1.2.1. Exceptional situations, as earthquakes, may be labeled as non-members by this test.*

2.2.4 Empirical risk minimization and overfitting

When a dataset S is given, and when some hypothesis space $\mathcal{H}$ is determined to extract some good predictor from, one very common *induction principle* is to find the hypothesis that succeeds best in predicting the labels of the dataset. This inductive principle is the *empirical risk minimization* (*ERM*). In other words, the idea is to find

$$\widehat{h}_S \stackrel{\text{def}}{=} \operatorname*{argmin}_{h \in \mathcal{H}} \mathcal{R}^S_{\text{emp}}(h)$$

where $\mathcal{R}^S_{\text{emp}}(h)$ is the empirical risk of h computed on S. The function $\widehat{h}_S$ is also denoted by h_N ($N = |S|$). When $\mathcal{H}$ is a parametric function set (see section 2.1.1), one can use a gradient descent in the parameter space to find $\widehat{h}_S$.

Some problem arise if $\mathcal{H}$ gathers complex functions, i.e. $\mathcal{H}$ is rich. Indeed, one can find in $\mathcal{H}$ many hypotheses h for which $\mathcal{R}^S_{\text{emp}}(h) = 0$. Such hypotheses fit the dataset perfectly good.

In general cases, when a perfect fit to the data occurs, one should suspect an *overfitting* situation, where the function $\widehat{h}_S$ that has been found performs a *learning by heart*. For example, this is what happened in figure 2.3, since the non-linear decision region gives the right label for all the ten points, whereas it will certainly not generalize this good labelling for new data. Overfitting occurs when $\mathcal{H}$ is rich enough to contain functions that can learn big datasets by heart.

Let us sum up the problem. The best predictor available in $\mathcal{H}$ *is* $h^\star = \operatorname{argmin}_{h\in\mathcal{H}} \mathcal{R}(h)$. *It cannot be computed. If* $\mathcal{R}(h^\star) = 0$, *it can be said that* $\mathcal{H}$ *contains a function that behaves like the unknown oracle. If it is not null,* $\mathcal{R}(h^\star)$ *is called the* inductive bias, *revealing that* $\mathcal{H}$ *is not rich enough to fit the oracle better than this*[a]. *If, for some dataset* S, $\mathcal{R}^S_{\text{emp}}\left(\widehat{h}_S\right) \approx 0$, *an overfitting has to be suspected, since it may happen that* $\mathcal{R}\left(\widehat{h}_S\right)$ *is high whereas* $\widehat{h}_S$ *behaves perfectly on the dataset. The low empirical risk is then misleading. On the contrary, if* $\mathcal{R}^S_{\text{emp}}\left(\widehat{h}_S\right)$ *is high for some dataset* S, $\mathcal{H}$ *seems not rich enough to fit the problem. In this case* $\mathcal{R}^S_{\text{emp}}\left(\widehat{h}_S\right)$ *may be a good approximation of* $\mathcal{R}\left(\widehat{h}_S\right)$. *Moreover, in this case as well,* $\mathcal{R}\left(\widehat{h}_S\right)$ *may approximate the inductive bias* $\mathcal{R}(h^\star)$, *meaning that the empirical risk minimization, from which* $\widehat{h}_S$ *is otained, is a good inductive principle (*$\widehat{h}_S$ *behaves as well as* $h^\star$*). In order to reduce the inductive bias, one may prefer having rich hypothesis spaces... but then, this richness allows for the* $\widehat{h}_S$ *found on some* S *to vary from one dataset to another, since learning is a learning by heart that just fits the given dataset without generalizing the labeling process. In other words, the variance here means that* $\widehat{h}_S$ *can be completely different from* $\widehat{h}_{S'}$ *even when* S *and* S' *slightly differ. Choosing the appropriate level of richness for* $\mathcal{H}$ *is difficult, it is refer to as the* bias-variance trade-off.

[a] It may also be due to some randomness in the oracle itself... Indeed, the inductive bias is not exactly $\mathcal{R}(h^\star)$, see paragraph 5.1.2 for details.

2.3 Ensemble methods

Let us say here few words about *ensemble learning* for supervised learning. The core idea is to handle a set of predictors instead of a single one. Once learning has occurred, all predictors propose a label for an incoming new input. The proposed labels are used to compute the label actually assigned to the new input. For example, in case of classification, each predictor proposition can be viewed has a vote for one of the available labels. The label which gets the maximum number of votes is then the one assigned to the input. In case of regression, the average value of all the proposed labels can be used to compute the label given finally to the input. Of course, when some data sample S is given, learning several predictors may lead to different ones if the process is non deterministic somewhere. The point is that even if each predictor performs just slightly better than a random labelling, the merged prediction can actually benefit from the variability of the predictor and be quite good.

2.3.1 Bagging

First place where randomness can be introduced is the sample set. For training each predictor, one can build up a dataset from S by sampling P data from S uniformly, with replacement. If the learning process is very sensitive to the data (some samples can be duplicated since replacement occur, some others may be missing, ...), the predictors got from this process may show variability. Nevertheless that is counterbalanced with the merging procedure described previously to get a good overall prediction. Making datasets this way is called *bootstrapping*, and relying on this to set up many predictors, whose predictions are merged, is called *bagging*.

2.3.2 Random models

The second place for adding randomness in the building up of a predictor collection is the predictor set up itself. Indeed, instead of using a heavy and accurate optimization process from the data in S, one can set up an almost-completely-random predictor... It means randomly generated predictors that behave slightly better than random. It is interesting when this generation saves computation

time (compared to an accurate optimization process). As for bagging, the merging of the predictions compensates for the weaknesses of each single predictor output. Dataset bootstrapping can also be used for training random models, adding more variability.

2.3.3 Boosting

Boosting is a specific case in ensemble methods since it does not rely on randomness. Indeed, what is actually boosted in the *boosting* method is a weak predictor learning algorithm. To do the boosting trick, the weak learning algorithm has to be able to handle weighted datasets. A weighted dataset is a dataset where each sample is associated with a weight reflecting the importance given to that sample in the resulting predictor construction.

Boosting is an iterative process. First, equal weights are given to all samples in S. A first predictor is learned from this. Then, the weights are reconsidered in order to be increased for the samples badly labeled by the previously learned predictor. A second predictor is learned from the dataset with this new weight distribution.... and so on. The boosting theory gives formulas for weighting the predictions of each constructed predictors in order to set up a final predictor as a weighted sum of the individual predictions.

Chapter 3

The Bayesian approach

This chapter introduces the basics of Bayesian inference on a fake example, in order to build up the general scheme of Bayesian approaches of machine learning. The mathematics here aim at being intuitive rather than providing a rigorous definition of probabilities. The latter is grounded on the measure theory that is not addressed here.

3.1 Density of probability

3.1.1 Reminder

Let us remind here that a random variable X taking values in $\mathcal{X}$ is described by its probability distribution P_X. Let $A \subset \mathcal{X}$ a set of values, the probability that the random variable X "takes" a value in this set is $\mathrm{P}_X(A)$. Moreover, let us suppose that a density of probability can be associated to the probability distribution. It is a function $\mathrm{p}_X \in (\mathbb{R}^+)^{\mathcal{X}}$ such as $\mathrm{P}_X(A) = \int_A \mathrm{p}_X(x)\mathrm{d}x$. In the following, random variables are described thanks to the associated density of probability, that is supposed to exist.

3.1.2 Joint and conditional densities of probability

Let us suppose now that occurring events are pairs of values $(x, y) \in \mathcal{X} \times \mathcal{Y}$. Let us denote by $X.Y$ a random variable taking values $(x, y) \in \mathcal{X} \times \mathcal{Y}$. The density of probability associated with this variable is $\mathrm{p}_{X.Y} \in (\mathbb{R}^+)^{(\mathcal{X}\times\mathcal{Y})}$. This represents the occurrences of pairs (x, y) in the world modeled by $X.Y$. If $\mathrm{p}_{X.Y}(x, y)$ is higher, the pair of values which are close to (x, y) are more "probable[1]". Figure 3.1 illustrates this as well as the incoming definitions.

Let us now suppose that only the x component of incoming events is of interest. The random variable describing its occurrence is denoted by X and it takes its values in $\mathcal{X}$. The associated density of probabilities can be computed as:

$$\forall x \in \mathcal{X},\ \mathrm{p}_X(x) \overset{\text{def}}{=} \int_{\mathcal{Y}} \mathrm{p}_{X.Y}(x, y)\mathrm{d}y$$

Here, the random variable X is obtained[2] by a *marginalization* of the variable $X.Y$. We can obtain Y the same way. It can also be said that $X.Y$ is a *joint variable* of X and Y.

If now, we are only considering the occurring pairs such as $x = x_0$. The "probability" of such pairs to occur is the "probability" of pairs (x, y) knowing that $x = x_0$, in other words the "probability" of y knowing that $x = x_0$. This is how conditional densities of probability can be defined.

[1] As both $\mathcal{X}$ and $\mathcal{Y}$ are usually continuous, the probability some (x, y) to occur *exaclty* if null, of course, even when the density $\mathrm{p}_{X.Y}(x, y)$ for that pair is high. Saying that (x, y) is "probable" is thus abusive, this is why it is quoted.

[2] Here, its density of probability is obtained. It describes the random variable when it exists.

$$\forall y \in \mathcal{Y},\ \mathrm{p}_{Y|X=x_0}(\mathbf{y}) \overset{\text{def}}{=} \frac{\mathrm{p}_{X.Y}(x_0,\mathbf{y})}{\int_{\mathcal{Y}} \mathrm{p}_{X.Y}(x_0,y')\mathrm{d}y'} = \frac{\mathrm{p}_{X.Y}(x_0,\mathbf{y})}{\mathrm{p}_X(x_0)} \tag{3.1}$$

In equation 3.1, the argument y is highlighted, in order to stress that $\mathrm{p}_{Y|X=x_0}(y)$ is a function of y. This will be not recalled in the following.

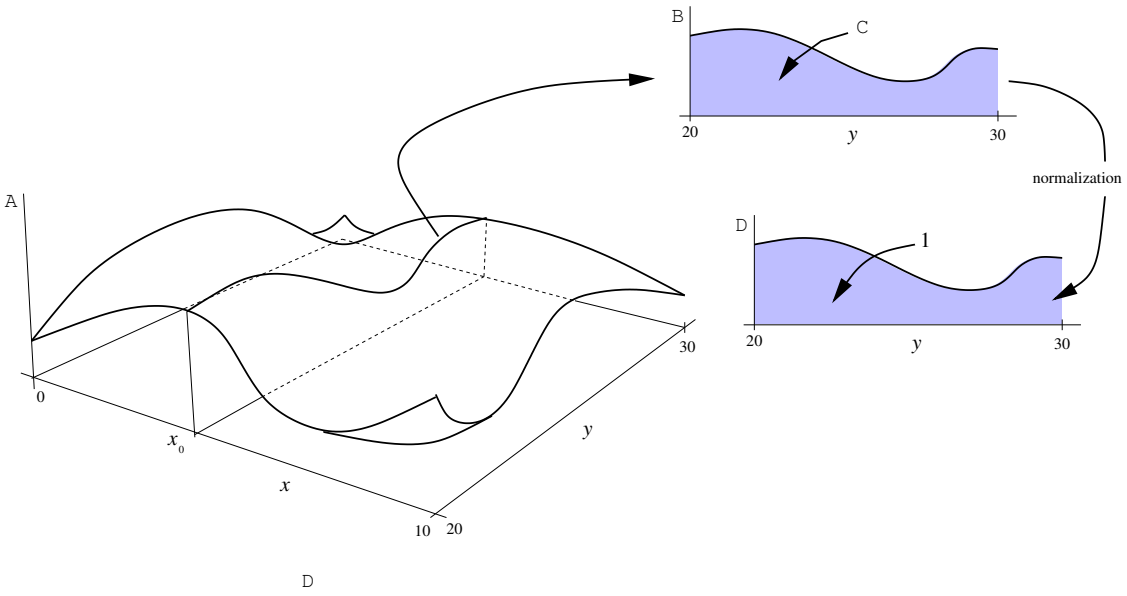


Figure 3.1: Joint and conditional densities of probability. $\mathcal{X} = [0, 10]$ and $\mathcal{Y} = [20, 30]$. In the figure, $\mathtt{A} = \mathrm{p}_{X.Y}(x, y)$, $\mathtt{B} = \mathrm{p}_{X.Y}(x_0, y)$, $\mathtt{C} = \mathrm{p}_X(x_0)$ and $\mathtt{D} = \mathrm{p}_{Y|X=x_0}(y)$.

3.1.3 The Bayes' rule for densities of probability

From equation (3.1), it can be derived straightforwardly that

$$\forall (x, y) \in \mathcal{X} \times \mathcal{Y},\ \mathrm{p}_{Y|X=x}(y) \times \mathrm{p}_X(x) = \mathrm{p}_{X.Y}(x, y) = \mathrm{p}_{X|Y=y}(x) \times \mathrm{p}_Y(y)$$

which leads to the following expression of the *Bayes' rule*, expressed for densities of probability:

$$\forall (x, y) \in \mathcal{X} \times \mathcal{Y},\ \mathrm{p}_{Y|X=x}(y) = \frac{\mathrm{p}_{X|Y=y}(x) \times \mathrm{p}_Y(y)}{\mathrm{p}_X(x)} \tag{3.2}$$

It is very similar to the more usual Bayes' rule with probabilities, i.e. $\mathrm{P}(A \mid B) = \frac{\mathrm{P}(B \mid A) \times \mathrm{P}(A)}{\mathrm{P}(B)}$, but components of the formulas here are functions (the densities) rather than scalars (the probabilities).

The Bayes' rule for densities of probability is the core mechanism for Bayesian learning, as illustrated next.

3.2 Bayesian inference

Let us illustrate what *Bayesian inference* is from a fake example.

3.2.1 The model

As usually in machine learning (see introduction section 1.1.1), the data is modelled by a random variable Z providing values in $\mathcal{Z}$. As for the frequentist approach (see section 2.1.1), the idea is to infer from data samples a parametric model that fits them. This parametric model, as in frequentist approach as well, is given a priori.

For the sake of further densities of probability plotting, let us consider that models have a scalar parameter $\theta \in \Theta = [0, 1]$. For a specific value θ, we consider the data to be samples $z \in \mathcal{Z} = [0, 1]$ of the random variable m_θ whose density of probability $\mathrm{p}_{m_\theta}(z)$ is defined in figure 3.2.

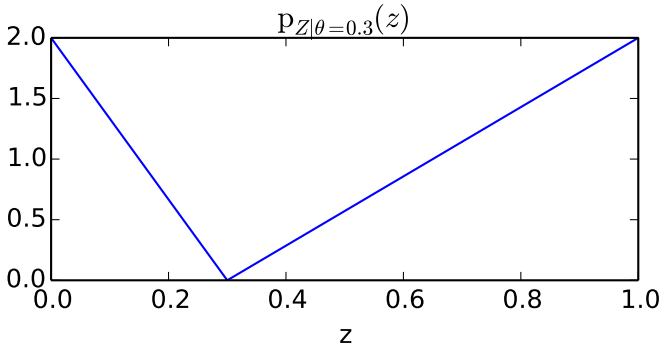


Figure 3.2: The density of probability $\mathrm{p}_{m_\theta}(z)$ for $\theta = .3$.

This density represents the distribution of data for a specific θ. Let us now introduce a random variable T with values in Θ... this is the trick of Bayesian inference. Its density $\mathrm{p}_T(\theta)$ represents the values that the parameter θ is likely to have. This is set a priori to any kind of distribution. Introducing the random variable T gives the density of probability p_{m_θ} a conditional flavor. Indeed, let us consider that

$$\forall z \in \mathcal{Z},\ \mathrm{p}_{m_\theta}(z) \overset{\text{def}}{=} \mathrm{p}_{Z|T=\theta}(z).$$

We have thus modeled a situation where getting a data sample consists in first tossing $\theta \leftarrow\!\!- \mathrm{P}_T$ and second tossing $z \leftarrow\!\!- \mathrm{P}_{m_\theta}$. Once again, the first toss sounds artificial... but this is the trick, as next section shows.

From this modeling of data generation, the joint density of probability can be computed easily from the Bayes' rule (see equation (3.2)).

$$\forall (z, \theta) \in \mathcal{Z} \times \Theta,\ \mathrm{p}_{Z.T}(z, \theta) = \mathrm{p}_{Z|T=\theta}(z) \times \mathrm{p}_T(\theta) = \mathrm{p}_{m_\theta}(z) \times \mathrm{p}_T(\theta) \tag{3.3}$$

Figures 3.3 and 3.4 show the joint density of probability for different p_T.

3.2.2 The parameter update

In the previous section, we have described probabilities for a given model m_θ as well as an a priori P_T. Let us consider p_T to be the one in figure 3.4 (i.e. parabolic), in order to stress that this choice is arbitrary and artificial. These probabilities do not reflect any reality without being related to real data. Let us suppose here that the hidden process generating the data is actually $\mathrm{P}_Z = m_{0.7}$. Let us sample a new data $z \leftarrow\!\!- m_{0.7}$ from it. From equation 3.2, i.e. the Bayes' rule for densities of probability, the following can be written.

$$\forall \theta \in \Theta,\ \mathrm{p}_{T|Z=z}(\theta) = \frac{\mathrm{p}_{Z|T=\theta}(z) \times \mathrm{p}_T(\theta)}{\mathrm{p}_Z(z)} \tag{3.4}$$

Note that here, the data z is a parameter. The variable is θ, so $\mathrm{p}_{T|Z=z}$ is a density of probability over Θ. Indeed, it tells how θ is distributed under our a priori hypotheses, knowing that the data z has been observed.

In equation (3.4), $\mathrm{p}_T(\theta)$ is called the *prior*. It can be computed since it is given a priori. The value $\mathrm{p}_Z(z)$ is a normalization constant. It is the "probability" of the occurrence of that data when the situation is modelled as we do... the way the data is actually sampled is not considered. $\mathrm{p}_Z(z)$ can be computed here numerically[3] from the joint density of probability given by equation (3.3). The density of probability $\mathrm{p}_{Z|T=\theta}(z)$ is known since it is our model, i.e. $\mathrm{p}_{m_\theta}(z)$, as already stated.

[3] This is a marginalization computed from an integral $\mathrm{p}_Z(z) = \int_\Theta \mathrm{p}_{Z.T}(z, \theta)\mathrm{d}\theta$.

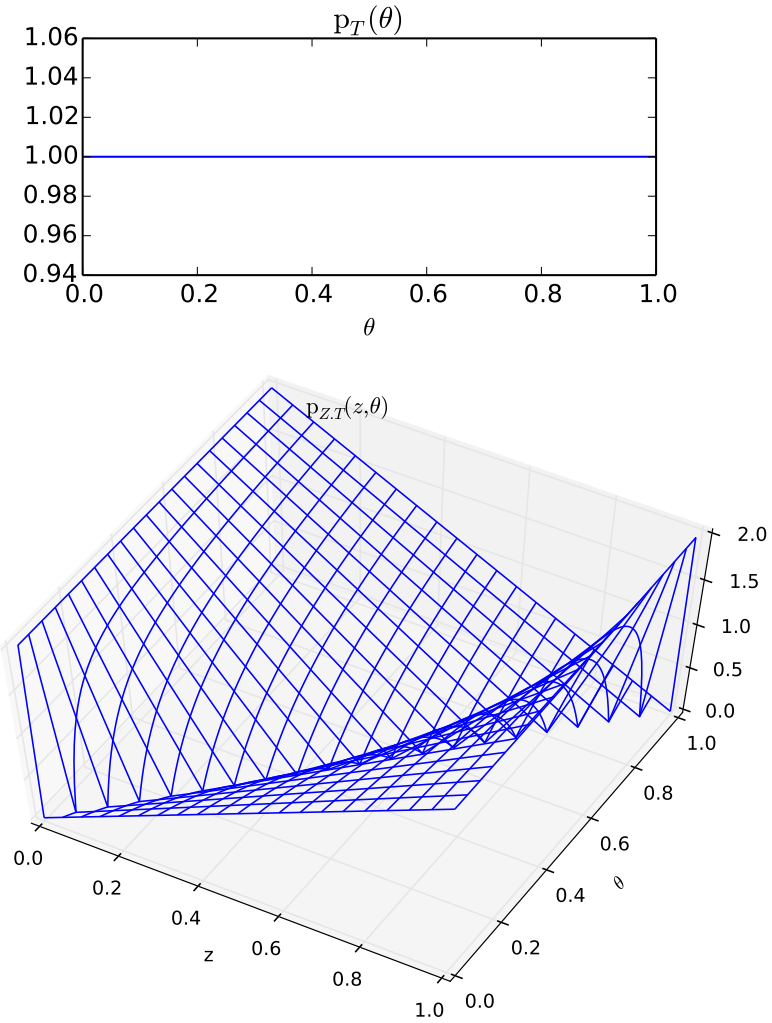


Figure 3.3: Joint distribution of parameter θ and data z. The distribution p_T (uniform here) is given a priori, as well as $p_{Z|T}$ which is the one defined in figure 3.2.

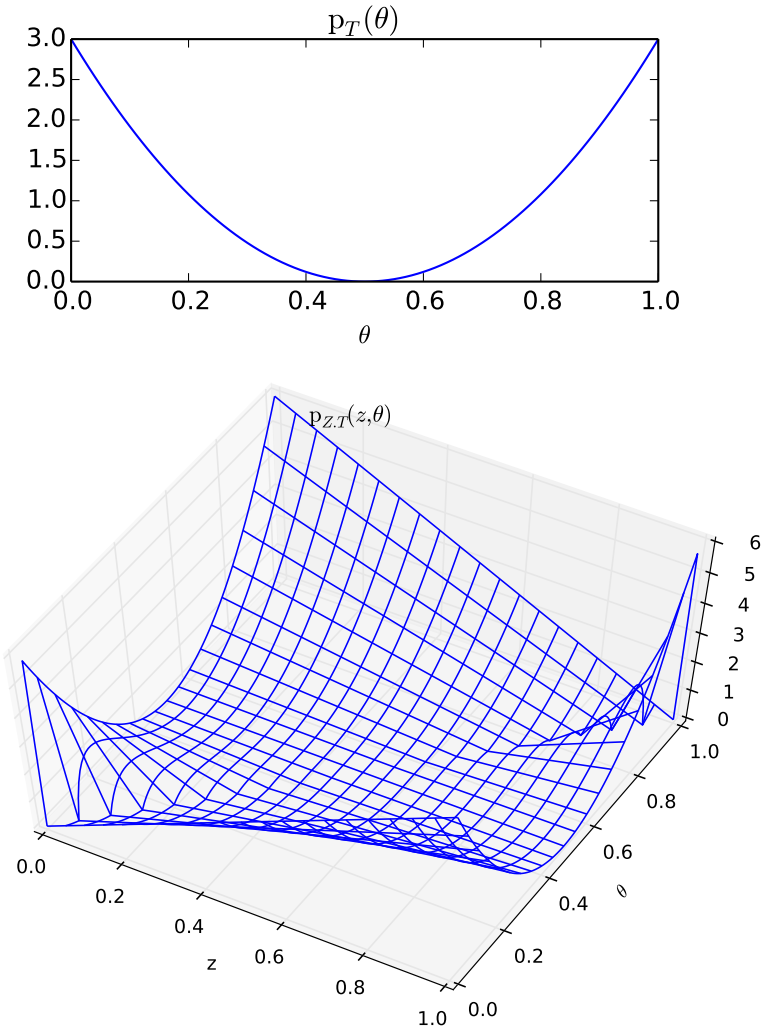


Figure 3.4: Joint distribution of parameter θ and data z. The distribution p_T (parabolic here) is given a priori, as well as $p_{Z|T}$ which is the one defined in figure 3.2.

The *Bayesian inference* consists in updating the prior $\mathrm{p}_T(\theta)$ so that it is now the function $\mathrm{p}_{T|Z=z}(\theta)$ that has been computed. Using that new prior changes the situation that we model. Indeed, this prior has somehow considered the data z that has been provided. *Bayesian learning* then consists in repeating this update for each new data sample that is tossed. This is illustrated in figure 3.5, where it can be seen that $\mathrm{p}_T(\theta)$ gets more and more focused, as the data samples are provided, to the value $\theta = 0.7$ which is actually the parameter we used for tossing data samples. The shape of the distribution reflects the *uncertainty* concerning the estimated value for that parameter.

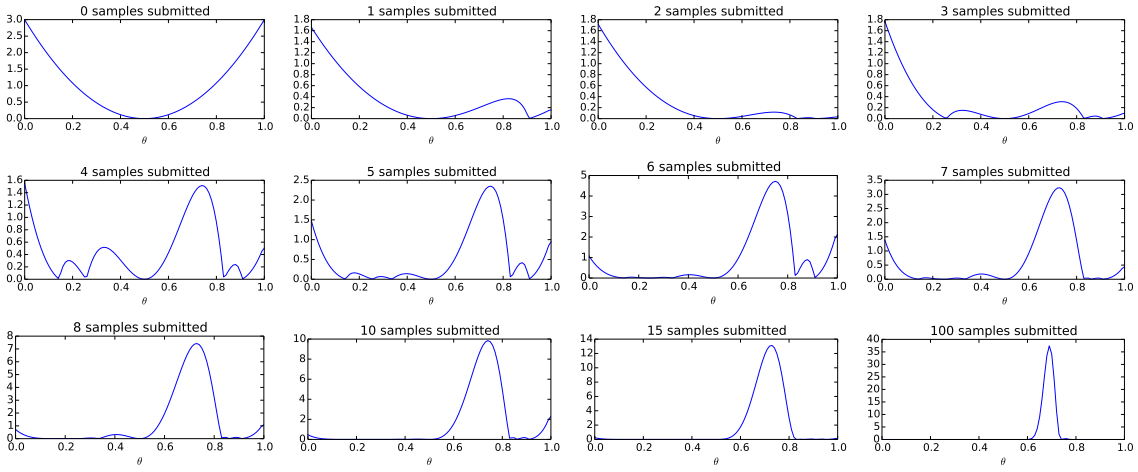


Figure 3.5: Bayesian inference. Each plot is the prior $\mathrm{p}_T(\theta)$ after the mentioned number of samples provided.

3.2.3 Update from bunches of data

Let us consider now two data samples z_2 and z_1, tossed *independently* from the unknown process. They are a samples of a random variable $K = (Z, Z)$. Since samples are independent, and since each one is linked to m_θ, the following stands:

$$\mathrm{p}_{K|T=\theta}(z_2, z_1) = \mathrm{p}_{m_\theta}(z_2) \times \mathrm{p}_{m_\theta}(z_1) = \mathrm{p}_{Z|T=\theta}(z_2) \times \mathrm{p}_{Z|T=\theta}(z_1)$$

So the Bayesian update (equation (3.4)), when a pair of *independent* samples is given, is the following:

$$\begin{aligned}
\forall \theta \in \Theta,\ \mathrm{p}_{T|K=z_2,z_1}(\theta) &= \frac{\mathrm{p}_{K|T=\theta}(z_2, z_1) \times \mathrm{p}_T(\theta)}{\mathrm{p}_K(z_2, z_1)} \\
&= \frac{\mathrm{p}_{Z|T=\theta}(z_2) \times \mathrm{p}_{Z|T=\theta}(z_1) \times \mathrm{p}_T(\theta)}{\mathrm{p}_Z(z_2) \times \mathrm{p}_Z(z_1)} \\
&= \frac{\mathrm{p}_{Z|T=\theta}(z_2)}{\mathrm{p}_Z(z_2)} \times \frac{\mathrm{p}_{Z|T=\theta}(z_1) \times \mathrm{p}_T(\theta)}{\mathrm{p}_Z(z_1)} \\
&= \frac{\mathrm{p}_{Z|T=\theta}(z_2) \times \left(\dfrac{\mathrm{p}_{Z|T=\theta}(z_1) \times \mathrm{p}_T(\theta)}{\mathrm{p}_Z(z_1)} \right)}{\mathrm{p}_Z(z_2)}
\end{aligned}$$

The last equation is exactly the successive application of equation 3.4 to z_1 and then z_2. This can be generalized to any number of samples. As a consequence, the result of successive applications of the Bayesian inference, when samples are mutually independent, is the density of probability of the parameter, knowing the whole bunch of samples, whatever the order of submission, and knowing the initial prior. The effect of the latter may vanish when the number of samples is high.

3.3 Bayesian learning for real

In real problems, the model depends on $\theta \in \mathbb{R}^n$, so both m_θ and the prior $p_T(\theta)$ are multidimensional functions. Moreover, one often wants to compute the update analytically, with probability densities whose form enables an analytical computation (e.g. multidimensional Gaussian densities, Dirichlet densities, etc...). This raises a problem. Having a prior of a certain kind do not guaranty that it will still be of the same kind after an update. Indeed, in figure 3.5, the prior was initially parabolic, but it takes a non-parabolic shape after the very first update, and it becomes similar to a Gaussian-like shape in the end. It would be nice, when the computation is run fully analytically[4], to handle a kind of density that is stable when equation (3.4) is applied. To ensure this, the model should be a *conjugate prior* of the prior. For example, when the model is Gaussian, and if the prior is also Gaussian, the new prior will still be Gaussian[5].

Having a model that is a conjugate prior is very restrictive. To get rid of the limitation in the shapes of the probability densities, once can use Monte Carlo sampling[6].

Last, let us mention that Bayesian updates are intrinsically online methods (see section 1.1.1), since update is made each time a sample is presented. Equation (3.2) can be adapted to take a bulk of samples instead of a single one. Some of the Bayesian approaches of machine learning are addressed in the companion course *Statistical Models (for Machine Learning)*.

[4] Numerical evaluations were used to process figure 3.5.
[5] and the new mean and variance can be computed analytically !
[6] See MCMC sampling (Andrieu et al., 2003).

Chapter 4

Evaluation

4.1 Real risk estimation

As previously mentioned in paragraph 2.2.4, measuring the performance of some predictor on the dataset that has been used to train it may lead to a very optimistic estimation of its real performance. The extreme case is *overfitting*, when the predictor behaves perfectly on the trained data while it is indeed very bad in general.

The real risk, which is the reliable measure of a predictor performance, cannot be computed, as opposed to the empirical risk... that cannot be trusted ! Performance evaluation has thus to be done carefully.

4.1.1 Cross-validation

Overfitting occurs when the hypothesis $\widehat{h}_S$ computed from the dataset S sticks to that set. In this case, $\mathcal{R}^S_{\text{emp}}\left(\widehat{h}_S\right) \approx 0$, this is the fitting to the training data which is expected by any inductive principle, but $\mathcal{R}^{S'}_{\text{emp}}\left(\widehat{h}_S\right)$ is high for another dataset S', meaning that the fitting to S is actually an overfitting.

Let us call S the *training set*, since learning consists in building $\mathcal{R}^S_{\text{emp}}$ from it. One can detect overfitting by computing the empirical risk of $\widehat{h}_S$ on another dataset S', called the *test set*. Of course, both S and S' are sampled i.i.d. according to P_Z (see algorithm 1).

Usually, only a single dataset is available and one can do better than only evaluate the empirical risk on a train set. This is the idea of the *k-fold cross-validation* procedure described by algorithm 7 that provides an estimation of the real risk of $\widehat{h}_S$ by a generalization of the use of only two training and test sets described so far. This is illustrated on figure 4.1.

Algorithm 7 `cross_validation`(k, S, α)

1: // α is a learning algorithm
2: Split S into a partition $\{S_1, \cdots, S_i, \cdots, S_k\}$ such as $\forall i, |S_i| \approx |S|/k$
3: **for** $i = 1$ **to** k **do**
4: Train from $S'_i = S \setminus S_i$ using α and get the predictor $\widehat{h}_{S'_i} = \alpha(S'_i)$. // $S \setminus S_i$ is the training set.
5: Compute $R_i = \mathcal{R}^{S_i}_{\text{emp}}\left(\widehat{h}_{S'_i}\right)$. // S_i is the test set.
6: **end for**
7: **return** $\frac{1}{k}\sum_{i=1}^{k} R_i$ // This is an estimation of $\mathcal{R}\left(\widehat{h}_S\right)$.

When $k = |S|$, the k-fold cross-validation is referred to as a *leave-one-out cross-validation*. It requires a lot of computation, but is can be usefull if few data is available.

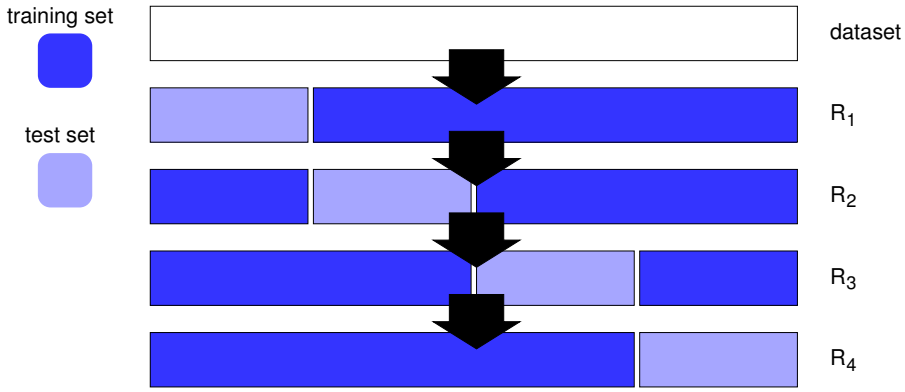


Figure 4.1: 4-fold cross-validation. See text and algorithm 7 for detail.

> *The least thing that can be done when applying supervized machine learning is to evaluate the method with cross-validation.*

4.1.2 Real risk optimisation

In real situation, machine learning algorithms often depend on parameters that can be tuned. One should choose the parameter configuration for which the real risk is minimal. It can be approximated by choosing the parameter configuration for which the estimated real risk is minimal.

This very common situation can lead to estimation errors. Let us introduce notations to illustrate the problem. Let α_θ an algorithm whose parameters are θ. For a given θ, one can compute h_θ from a dataset S, and evaluate its real risk $\mathcal{R}(h_\theta) \approx R_\theta = \texttt{cross_validation}(k, S, \alpha_\theta)$. Since R_θ can be computed, one can also compute

$$\theta^\star = \underset{\theta \in \Theta}{\operatorname{argmin}} R_\theta$$

and then, applying $\alpha_{\theta^\star}$ on S gives a predictor $h_{\theta^\star}$, which is the one returned by the whole process.

The error that could be made with this approach is considering that $R_{\theta^\star}$, computed during the optimization process, is an estimation of the real risk of the optimization process. Indeed, it estimates the real risk of using our learning algorithm with parameter $\theta^\star$, but not the real risk of the whole optimization process since the value of $\theta^\star$ is not independent from S.

In this case, apart from dividing S into training and test sets for cross-validation, an extra *validation set* S' should be used to measure $\mathcal{R}^{S'}_{\text{emp}}(h_{\theta^\star})$ and estimate the real risk of our optimization process. As we have extended training an testing sets to a whole cross-validation process, the validation set can be extended to a cross-validation as well... To do so, let us denote by `metalearn` the algorithm described above, reminded in algorithm 8.

This algorithm's real risk can be estimated by cross-validation, as any other, by simply calling $\texttt{cross_validation}(k, S, \texttt{metalearn})$. This process involves two nest levels of cross-validation (see figure 4.2), which generalize the use of train, test and validation sets.

4.2 The specific case of classification

4.2.1 Confusion matrix

The performance measurement for classification problems is, as for any supervised learning problem, the real risk. It is usually computed by using the binary loss (see equation (2.2)) so that the real risk is the percentage of errors that can be expected from the predictor/classifier.

Nevertheless, it can be relevant to get into more details for analyzing errors. To do so, the *confusion matrix* is usually computed. It consists in using a test set[1] and report in a matrix the

[1] It is better to use a set that has not been used for training the predictor.

Algorithm 8 `metalearn`(S)

```
// α is a learning algorithm with parameters θ, k is a constant.
R* = +∞
for θ ∈ Θ do
    // Consider operation research techniques if Θ cannot be iterated
    Compute R_θ = cross_validation (k, S, α_θ).
    if R_θ < R* then
        R* ← R_θ
        θ* ← θ
    end if
end for
Use α_θ* to train on S and get a predictor h_θ* = α_θ* (S).
return h_θ*
```

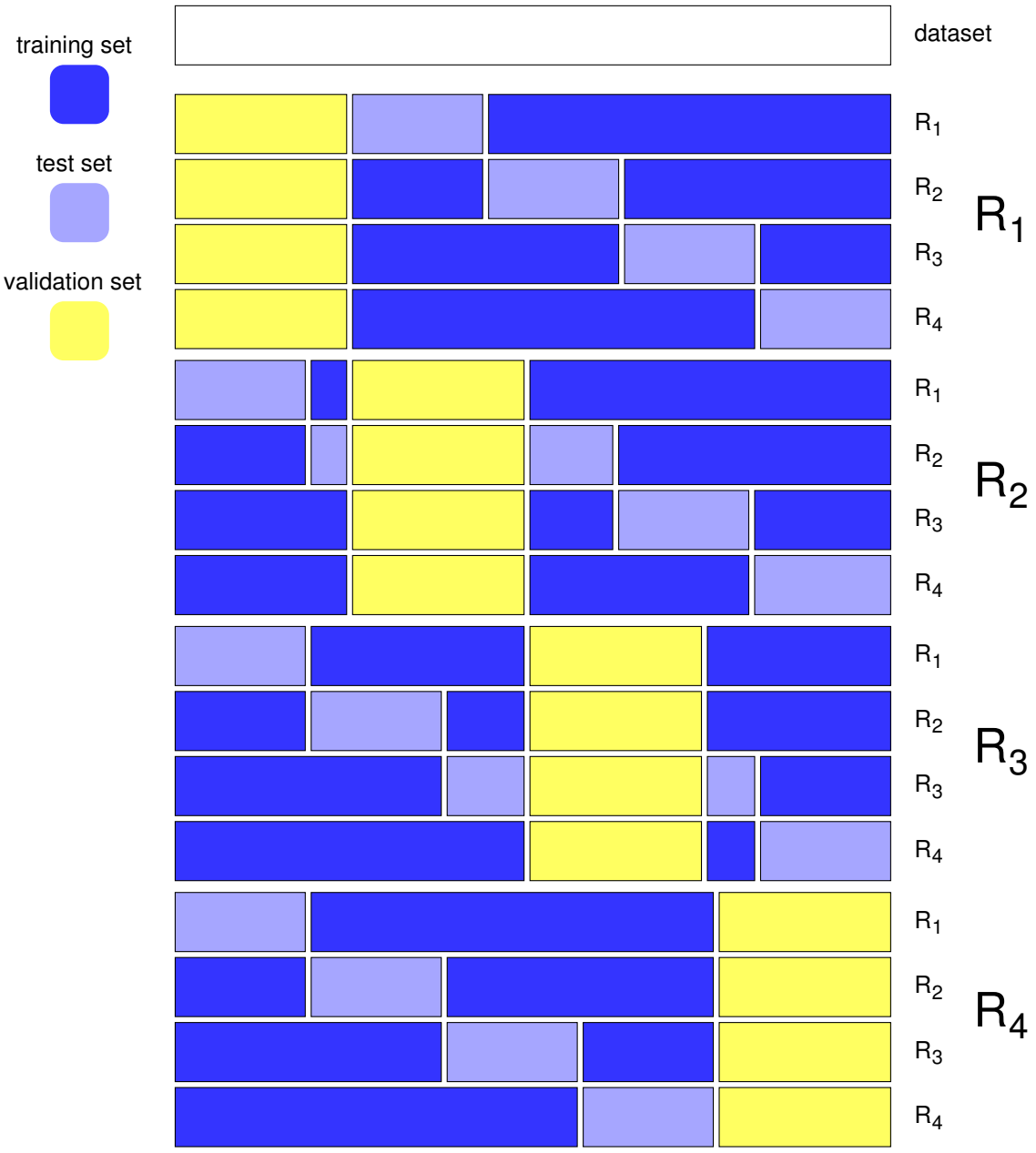


Figure 4.2: Cross-validation of algorithm 8 involves a nested cross-validation. See text for detail.

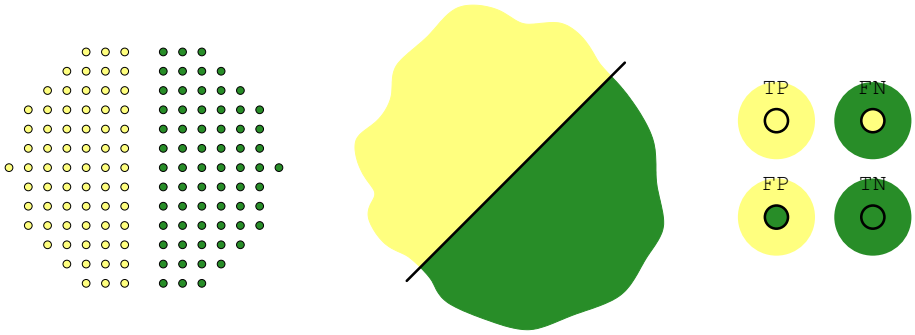


Figure 4.3: On the left, a label dataset is depicted. Positively labeled samples are painted in pale yellow, negatively labeled ones are painted in green. The middle of the figure shows a linear separator. It separates the plane into sub-regions. The sub-region corresponding to positive labels is painted in yellow as well, the negative sub-region is depicted in green. On the right, a recall of the confusion matrix coefficients with the colored graphical notation.

responses of the classifier, as in table 4.1. In this example, when a sample belongs to class `C` it is systematically given a `C` label by the predictor. Nevertheless, the `C` label is also given by the predictor when real class is not a `C`.

		predicted			
		A	B	C	D
real	A	13	2	2	3
	B	1	25	4	0
	C	0	0	40	0
	D	2	0	1	7

Table 4.1: Confusion matrix for some classifier computed from a dataset S with $|S| = 100$ data samples. The classes can be `A`, `B`, `C` or `D`. Numbers in the matrix are the number of samples satisfying the condition (their sum is 100).

The confusion matrix can also be meaningful when the problem is *cost sensitive*. It means that errors may have different weights/costs according to their nature. For example, predicting that a patient has no cancer while s/he actually has one is not the same as predicting a cancer while the patient is sane... Some cost matrix C can be associated to the confusion matrix, where C_{ij} is the cost of predicting class i while the real class is j. Usually, the C_{ii} are null.

4.2.2 The specific case of bi-class problems

When $|\mathcal{Y}| = 2$, the classification problem is bi-class, which is very common. The two classes are often referred to as positive and negative classes, i.e. $\mathcal{Y} = \{\texttt{P}, \texttt{N}\}$, since the problem usually consists in detecting something (positive means detected). The coefficients of the confusion matrix in this case have specific names, as table 4.2 shows. Let us depict these values by a linear separation example in figure 4.3.

		predicted	
		P	N
real	P	TP	FN
	N	FP	TN

Table 4.2: Confusion matrix for some bi-class classifier. Class `P` is the positive class and class `N` the negative one. The coefficient names are *true positives* (TP), *true negatives* (TN), *false positives* (FP), *false negatives* (FN).

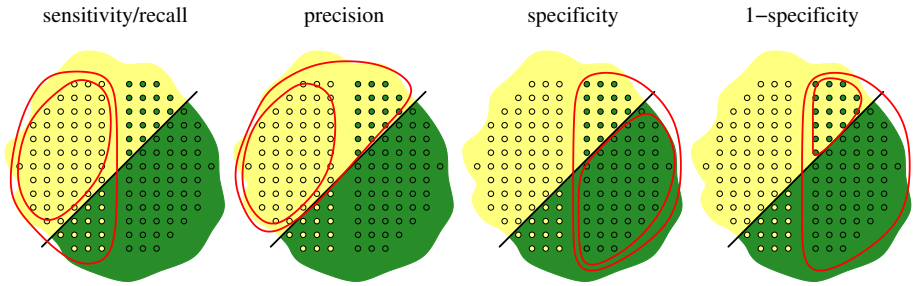


Figure 4.4: The different concepts are illustrated in the case of a linear separator. Each concept is a ratio (see text), represented by two read area. The central one is the numerator, and the surrounding one the denominator.

Lots of measures are based on the confusion matrix coefficients. Main ones are *sensitivity* or *recall* ($\frac{TP}{TP+FN}$), *specificity* ($\frac{TN}{FP+TN}$), *precision* ($\frac{TP}{FP+TP}$). Those definitions can be depicted graphically as well, as in figure 4.4.

It is also common to plot a predictor performance in a *ROC space*. That space is a chart depicted in figure 4.5.

> *It allows to situate the performances independently from the unbalance between the classes.*

Last, one can summarize the trade-off between sensitivity and precision with the *f-score* (or *f1-score*) that is computed as

$$f = 2 \times \frac{\text{sensitivity} \times \text{precision}}{\text{sensitivity} + \text{precision}}$$

The f-score is a way to merge sensitivity and precision into a single number, as shown by the chart in figure 4.6.

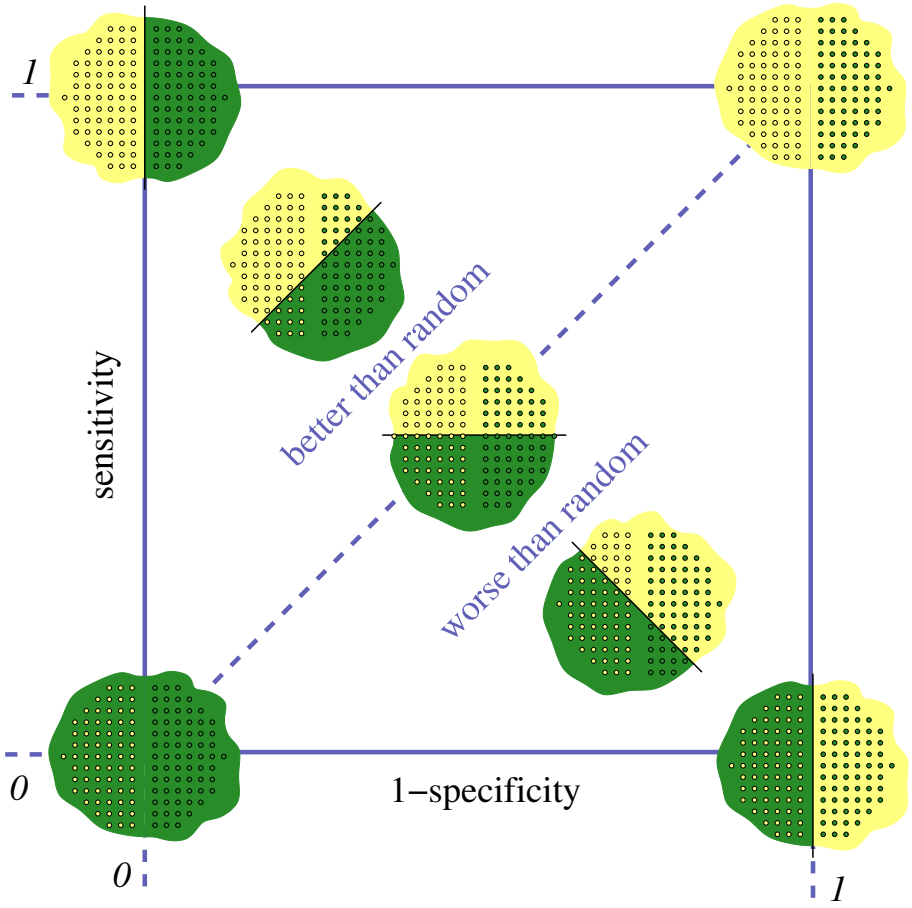


Figure 4.5: ROC space. See text for detail.

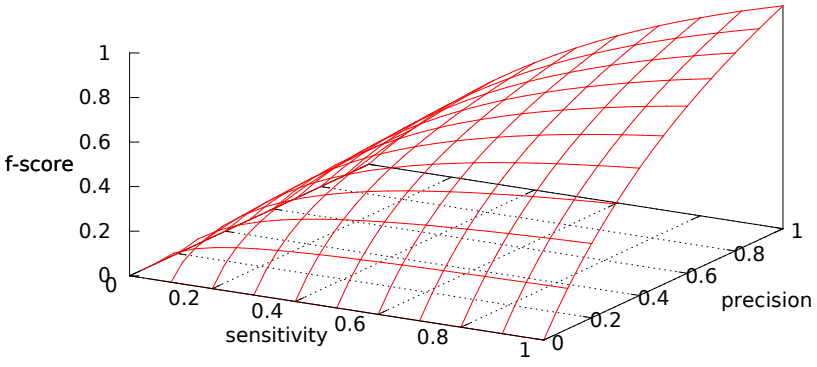


Figure 4.6: F-score. See text for detail.

Part II

Concepts for Machine Learning

Chapter 5

Risks

In machine learning, one wants to learn something from data. The "something" to be learnt is usually quantified thanks to the central notion of risk. However, the risk is an asymptotical concept, and learning is practically done by the optimization of an empirical counterpart of it. In section 5.1, we provide a brief introduction to statistical learning theory. Notably, we will provide (partial) answers to questions such as:

- does minimizing an empirical risk asymptotically lead to minimizing the risk?
- to what extent the minimized empirical risk is a good approximation of the risk?

These questions are respectively related to the consistency of machine learning and to overfitting.

In the case of classification, the natural (empirical) risk (based on the binary loss) is a difficult thing to optimize. Therefore, it is customary to minimize a convex surrogate (or proxy) instead. In section 5.2, we motivate the introduction of these surrogates, exemplify some of them and discuss their calibration (in other words, is it consistent to minimize them instead of the risk?).

As will be shown in section 5.1, overfitting is heavily related to the capacity (or richness) of the considered hypothesis spaces. In section 5.3, we briefly introduce the concept of regularization, which is a modification of the risk that penalizes complex solutions (the aim being to avoid overfitting by restricting the search space).

5.1 Controlling the risk

This section provides a brief introduction to *statistical learning theory*. We start by formalizing the learning problem, focusing mainly on supervised learning.

5.1.1 The considered learning paradigm

To formalize the learning problem, we assume that we have:

- a random generator of vectors $x \in \mathcal{X}$, sampled i.i.d. (independently and identically distributed) from a distribution $\mathrm{P}(x)$, *fixed* but *unknown*;
- an *oracle* that for each input x provides an output $y \in \mathcal{Y}$, sampled according to the conditional distribution $\mathrm{P}(y \mid x)$, also *fixed* but *unknown*. If $\mathcal{Y} = \mathbb{R}$, we're facing a regression problem, whereas if $\mathcal{Y}$ is a finite set, we're facing a classification problem;
- a machine that can implement a set of functions, this set being called the *hypothesis space* $\mathcal{H} = \{f : \mathcal{X} \to \mathcal{Y}\} \subset \mathcal{Y}^{\mathcal{X}}$.

In a first try, supervised learning can be framed as follows: pick $f \in \mathcal{H}$ that predict "the best" the responses of the oracle. The choice of f must be done based on a dataset $\mathcal{D} = \{(x_i, y_i)_{1 \leq i \leq n}\}$ of n examples sampled i.i.d. from the joint distribution $\mathrm{P}(x, y) = \mathrm{P}(x)\mathrm{P}(y \mid x)$. If these examples are fixed in practice (the dataset is given beforehand), they should really be understood here as

i.i.d. random variables. All quantities computed using these samples are therefore also random variables.

Before stating more precisely what "the best" means formally, we give some examples of hypothesis spaces:

linear predictions: write $\mathcal{Y} = \mathbb{R}$ (regression) and $\mathcal{X} = \mathbb{R}^p$, the hypothesis space is

$$\mathcal{H} = \left\{ f_{\alpha,\beta} : \mathcal{X} \to \mathbb{R}, \quad f_{\alpha,\beta}(x) = \alpha^\top x + \beta, \alpha \in \mathbb{R}^p, \beta \in \mathbb{R} \right\}.$$

In this case, searching for a function $f \in \mathcal{H}$ amounts to search for the related parameters α and β. If $\mathcal{Y} = \{-1, 1\}$ (binary classification, see section 5.2 for multiclass classification), we can define similarly the following space (writing sgn the operator that gives the sign of a scalar):

$$\mathcal{H} = \left\{ f_{\alpha,\beta} : \mathcal{X} \to \{-1, 1\}, \quad f_{\alpha,\beta}(x) = \operatorname{sgn}(\alpha^\top x + \beta), \alpha \in \mathbb{R}^p, \beta \in \mathbb{R} \right\};$$

Radial Basis Function Networks (RBFN): the underlying idea here is that many functions can be represented as a mixture of Gaussians. Given d vectors $\mu_i \in \mathbb{R}^p$ (the centers of the Gaussians) and d symmetric and positive definite matrices Σ_i (variance matrices) chosen *a priori*, the hypothesis space is

$$\mathcal{H} = \left\{ f_{\alpha,\beta} : x \to \sum_{i=1}^{d} \alpha_i \exp\left(\frac{1}{2} (x - \mu_i)^\top \Sigma_i^{-1} (x - \mu_i) \right) + \beta \right\}.$$

Each Gaussian function is generally called a basis function. Using the same sign trick as before, this can be used to build an hypothesis space for classification;

linear parameterization: In an abstract way, write $\phi : \mathcal{X} \to \mathbb{R}^d$ a vector function concatenating predefined basis functions ($\phi(x)$ is usually called the feature vector and $\phi_j(x)$, its jth component, is the jth basis function), we have the following hypothesis space:

$$\mathcal{H} = \left\{ f_\alpha : x \to \alpha^\top \phi(x), \alpha \in \mathbb{R}^d \right\}. \tag{5.1}$$

Again, this can be modified to form an hypothesis space for binary classification;

nonlinear parametrization: In all above examples, the dependency on the parameters is linear. This is not necessarily the case, consider an RBFN where the weight of each basis function, but also the mean and variance of each basis function, has to be learnt. Another classical example is when predictions are made using an artificial neural network, see part V;

Reproducing Kernel Hilbert Space (RKHS): Let $\{(x_i, y_i)_{1 \leq i \leq n}\}$ be the dataset and let K be a Mercer kernel[1], the hypothesis space can be written as

$$\mathcal{H} = \left\{ f_\alpha : x \to \sum_{i=1}^{n} \alpha_i K(x, x_i), \alpha \in \mathbb{R}^n \right\}.$$

Notice that, contrary to the previous examples, the hypothesis depends here on the dataset of learning samples (through the use of the inputs x_i). The related approach is usually called non-parametric. Notice also that $\mathcal{H}$ is not the RKHS, but a subset of it, see part III for details.

Now, we still need to precise formally what we mean by predicting "the best". To do so, we introduce the notion of *loss function*. A loss function $L : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}_+$ measures how two outputs are similar. It allows quantifying locally the quality of the prediction related to a predictor $f \in \mathcal{H}$: $L(y, f(x))$ measures the error between the response y of the oracle for a given input x and the prediction $f(x)$ of the machine for the same input. Here are some examples of loss functions:

[1] Roughly speaking, this is the functional generalization of a symmetric and positive definite matrix, see part III for details.

ℓ_2-loss: it is defined as

$$L(y, f(x)) = (y - f(x))^2.$$

It is the typical loss used for regression.

ℓ_1-loss: it is defined as

$$L(y, f(x)) = |y - f(x)|.$$

It is of interest in a regression setting when there are outliers (because outliers are less penalized with the ℓ_1-loss), for example;

binary loss: it is defined as

$$L(y, f(x)) = \mathbb{I}_{\{y \neq f(x)\}} = \begin{cases} 1 & \text{if } f(x) \neq y \\ 0 & \text{else} \end{cases}. \tag{5.2}$$

It is the ideal (but unpractical, see section 5.2) loss for classification.

If the loss function quantifies locally the quality of the prediction, we need a more global measure of this quality. This is quantified by the *risk*, formally defined as the expected loss:

$$\begin{aligned} \mathcal{R}(f) &= \int_{\mathcal{X}\times\mathcal{Y}} L(y, f(x)) \mathrm{dP}(x, y) \\ &= \mathrm{E}\left[L(Y, f(X))\right]. \end{aligned}$$

Ideally, supervised learning would thus consists in minimizing the risk $\mathcal{R}(f)$ under the constraints $f \in \mathcal{H}$, giving the solution

$$f_0 = \operatorname*{argmin}_{f\in\mathcal{H}} \mathcal{R}(f).$$

Unfortunately, recall that the joint distribution $\mathrm{P}(x, y)$ is unknown, so the risk cannot even be computed for a given function f. However, we have a partial information about this distribution, through the dataset $\mathcal{D} = \{(x_i, y_i)_{1\leq i\leq n}\}$. It is natural to define the *empirical risk* as

$$\mathcal{R}_n(f) = \frac{1}{n}\sum_{i=1}^{n} L(y_i, f(x_i)).$$

A supervised learning algorithm will therefore minimize this empirical risk, providing the solution

$$f_n = \operatorname*{argmin}_{f\in\mathcal{H}} \mathcal{R}_n(f).$$

This is called the *empirical risk minimization* (or ERM for short). Notice that f_n is a random quantity (as it depends on the dataset, which is a collection of i.i.d. random variables).

To sum up, given a dataset (assumed to be sampled i.i.d. from an unknown joint distribution), given an hypothesis space (chosen by the practitioner) and given a loss function (also chosen by the practitioner, depending on the problem at hand), a supervised learning algorithm minimizes the empirical risk, constructed from the dataset, instead of ideally the risk. The natural questions that arise are:

- does f_n converges to f_0 (and with what type of convergence)? If it is not the case, machine learning cannot be consistent;
- given n samples and the chosen hypothesis space (and also depending on the considered loss function), how close is f_n to f_0?

These are the questions we study next. Before, we discuss briefly the *bias-variance decomposition* (a central notion in machine learning).

5.1.2 Bias-variance decomposition

Recall that f_0 and f_n are respectively the minimizers of the risk $\mathcal{R}$ and of the empirical risk $\mathcal{R}_n$, when the search is constrained to the hypothesis space $\mathcal{H}$. Write f_* the minimizer of the risk in the unconstrained case:

$$f_* = \operatorname*{argmin}_{f \in \mathcal{Y}^{\mathcal{X}}} \mathcal{R}(f), \quad \mathcal{R}_* = \mathcal{R}(f_*).$$

The term $\mathcal{R}_*$ is the best one can hope (notice that it is not necessary equal to zero, it depends on the noise of the oracle). In some cases, we can express analytically the minimizer f_*, but the corresponding solution cannot be computed, the underlying distribution being unknown. For example, with an ℓ_2-loss, one can show that $f_*(x) = \mathrm{E}[y \mid x] = \int_{\mathcal{Y}} y \mathrm{dP}(y \mid x)$, that cannot be computed (the conditional distribution being only observed through data).

Consider the following decomposition:

$$\underbrace{\mathcal{R}(f_n) - \mathcal{R}_*}_{\text{error}} = \underbrace{\mathcal{R}(f_0) - \mathcal{R}_*}_{\text{bias}} + \underbrace{\mathcal{R}(f_n) - \mathcal{R}(f_0)}_{\text{variance}}.$$

Each of these terms is obviously positive. The term $\mathcal{R}(f_n) - \mathcal{R}_*$ is the error of using f_n (computed from the data) instead of the best (but unreachable) choice f_*. The term $\mathcal{R}(f_0) - \mathcal{R}_*$ is called the **bias**. It is a deterministic term that compares the best solution one can find in the hypothesis space $\mathcal{H}$ to the best possible solution (without constraint). This term therefore depends on the choice of $\mathcal{H}$, but not on the data used. The richer the hypothesis space (the more functions belonging to it), the smaller can be this term. The term $\mathcal{R}(f_n) - \mathcal{R}(f_0)$ is called the **variance**. It is a stochastic term (through the dependency to data) that should go to zero as as the number of samples goes to the infinity. In the next sections, we will see that this is a necessary condition for the ERM to be consistent, and that we can bound it (in probability) with a function depending on the richness of $\mathcal{H}$ and on the number of samples.

5.1.3 Consistency of empirical risk minimization

In the following sections, we abstract a little bit the notations. We write $z = (x, y)$ an example and $Q(z, f) = L(y, f(x))$ (and $\mathrm{P}(z) = \mathrm{P}(x, y)$ the joint distribution). Consequently, the risk and its empirical counterpart are now:

$$\mathcal{R}(f) = \mathrm{E}[Q(Z, f)] \tag{5.3}$$

$$\text{and } \mathcal{R}_n(f) = \frac{1}{n} \sum_{i=1}^{n} Q(z_i, f). \tag{5.4}$$

This allows lightening the notations and gives more generality to the following results[2].

Let f be a *fixed* function of $\mathcal{H}$. The risk (5.3) is the expectation of the random variable $L(Y, f(X))$ and the empirical risk (5.4) is the empirical expectation of i.i.d. samples of the same random variable[3]. In the probability theory, the convergence of an empirical expectation is given by the laws of large numbers (there are many of them, depending on required assumptions and on the considered convergence). Here, we will focus on convergence in probabilities. Therefore, the weak law of large numbers states that, for the *fixed* $f \in \mathcal{H}$ chosen beforehand (before seing the data), we have:

$$\mathcal{R}_n(f) \xrightarrow[n\to\infty]{P} \mathcal{R}(f) \Leftrightarrow \forall \epsilon > 0, \mathrm{P}(|\mathcal{R}(f) - \mathcal{R}_n(f)| > \epsilon) \xrightarrow[n\to\infty]{} 0.$$

Notice that we cannot replace f by f_n (the minimizer of the empirical risk) in the above expression, because f_n depends on the data and is therefore itself a random quantity.

[2]For example, it applies also to some unsupervised learning algorithms. Let $x_1, \ldots, x_n$ be i.i.d. samples from an unknown distribution $\mathrm{P}(x)$ to be estimated. Let $\{p_\alpha(x), \alpha \in \mathbb{R}^d\}$ be a set of parameterized densities. Consider the (one-parameter) loss function $L(p(x, \alpha)) = -\log p_\alpha(x)$ and the related risk $\mathcal{R}(p_\alpha) = \mathrm{E}[-\log p_\alpha(x)]$ (this corresponds to maximizing the likelihood of the data). This case is also handled by the notations z and Q.

[3]We recall that it is important to understand that, given this statistical model of supervised learning, the dataset is a random quantity (so is the empirical risk), even if in practice the dataset is given beforehand and imposed. Imagine that we could repeat the experience (of generating the dataset). The dataset would be different, but still sampled i.i.d. from the same joint distribution.

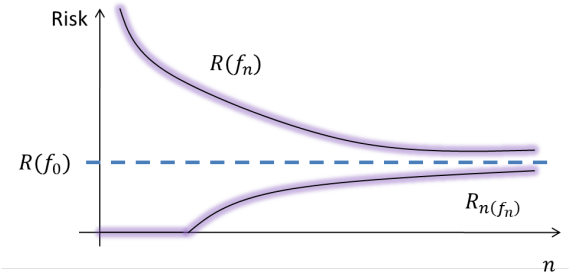


Figure 5.1: An illustration of the convergence behavior of the risk and its empirical counterpart. Consider for example a regression problem where one tries to fit a polynomial to data. With too few points, the empirical risk will be zero while the risk will be high. With an increasing number of data points, both risks should converge to the same quantity.

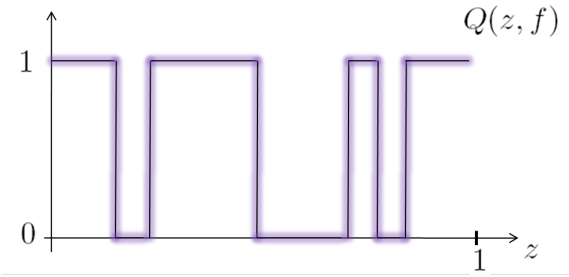


Figure 5.2: The considered hypothesis space for showing that limits in Def. 5.1 are not equivalent.

For the ERM to be consistent, we require that f_n converges to f_0 in some sense. What we are interested in is the quality of the solution, quantified by the risk. So, the convergence to be studied is the one of the (empirical) risk of f_n to the risk of f_0. This gives a first definition of consistency of ERM.

Definition 5.1 (Classic consistency of the ERM principle)**.** *We say the the ERM principle is consistent for the set of functions $Q(z, f)$, $f \in \mathcal{H}$, and for the distribution $\mathrm{P}(z)$ if the following convergences occur:*

$$\mathcal{R}(f_n) \xrightarrow[n\to\infty]{P} \mathcal{R}(f_0) \text{ and } \mathcal{R}_n(f_n) \xrightarrow[n\to\infty]{P} \mathcal{R}(f_0).$$

Notice that the two limits are not equivalent, as illustrated in Fig. 5.1. To show this more formally, assume that $z \in (0, 1)$ and consider $\mathcal{H}$ the space of functions such that $Q(z, f) = 1$ everywhere except on a finite number of intervals of cumulated length ϵ, as illustrated in Fig. 5.2. Assume that $\mathrm{P}(z)$ is uniform on $(0, 1)$. Then, for any $n \in \mathbb{N}$, $\mathcal{R}_n(f_n) = 0$ (pick the function with n intervals centered in the $z_1, \ldots, z_n$ datapoints). On the other hand, for any $f \in \mathcal{H}$, $\mathcal{R}(f) = 1 - \epsilon$. Therefore, $\mathcal{R}(f_0) - \mathcal{R}_n(f_n) = 1 - \epsilon$ does not converge to zero while $\mathcal{R}(f_0) - \mathcal{R}(f_n) = 0$ does.

The problem with this definition of consistency is that it encompasses straightforward cases of consistency. Assume that for a set of functions $Q(z, f)$, $f \in \mathcal{H}$, the ERM is not consistent. Now, let ϕ be a function such that $\phi(z) < \inf_{f\in\mathcal{H}} Q(z, f)$ and add the corresponding function to $\mathcal{H}$ (see Fig. 5.3). With this extended set, the ERM becomes consistant: For any distribution and any sampled dataset, the empirical risk is minimized with $\phi(z)$, which is also the argument that minimize the risk. This is a case we would like to avoid, motivating a more strict definition of consistency.

Definition 5.2 (Strict *consistency of the ERM principle*)**.** *Let $Q(z, f)$, $f \in \mathcal{H}$ be a set of function and $P(z)$ a distribution. For $c \in \mathbb{R}$, define $\mathcal{H}(c)$ the set*

$$\mathcal{H}(c) = \{f \in \mathcal{H} : \mathcal{R}(f) = \int Q(z, f)\mathrm{dP}(z) \geq c\}.$$

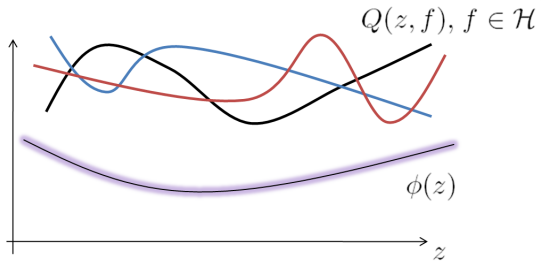


Figure 5.3: Triviality of the classic consistence.

The principle of ERM is said to be strictly consistent (for the above set of functions and distribution) if for any $c \in \mathbb{R}$, we have

$$\inf_{f \in \mathcal{H}(c)} \mathcal{R}_n(f) \xrightarrow[n \to \infty]{P} \inf_{f \in \mathcal{H}(c)} \mathcal{R}(f).$$

With this definition, the function $\phi(z)$ used in the previous explanation will be removed from the set $\mathcal{H}(c)$ for a large enough value of c. The fact that this definition of strict consistency implies the definition of classic consistency (the converse being obviously false) is not straightforward, but it can be demonstrated.

This notion of consistency is fundamental in machine learning. If it is not satisfied, minimizing an empirical risk has no sense (so, roughly speaking, machine learning would be useless). What we would like is a (necessary and) sufficient condition to have a strict consistency, that is a convergence in probabilities of the quantities of interest. The weak law of large numbers states that for any function f in $\mathcal{H}$, we have this convergence. However, this is not sufficient. As f_n is a random quantity, there is no way to know what function will minimize the empirical risk, so the standard law of large numbers do not apply. However, assume that we have a *uniform* convergence in probabilities, in the sense that

$$\sup_{f \in \mathcal{H}} |\mathcal{R}(f) - \mathcal{R}_n(f)| \xrightarrow[n \to \infty]{P} 0 \Leftrightarrow \forall \epsilon > 0, \mathrm{P}\left(\sup_{f \in \mathcal{H}} |\mathcal{R}(f) - \mathcal{R}_n(f)| > \epsilon \right) \xrightarrow[n \to \infty]{} 0, \tag{5.5}$$

which is much stronger than the weak law of large numbers (see the next section for a more quantitative discussion). With such a uniform convergence, the ERM principle is strictly consistent (convergence occurs in the worst case, so it occurs for f_n: $|\mathcal{R}(f_n) - \mathcal{R}_n(f_n)| \leq \sup_{f \in \mathcal{H}} |\mathcal{R}(f) - \mathcal{R}_n(f)|$). This is the base of a fundamental result of statistical learning theory.

Theorem 5.1 (Vapnik's key theorem)**.** *Assume that there exists two constants a and A such that for any function $Q(z, f)$, $f \in \mathcal{H}$, and for a given distribution $\mathrm{P}(z)$, we have*

$$a \leq \mathcal{R}(f) = \int Q(z, f) \mathrm{dP}(z) \leq A.$$

Then, the following assertion are equivalents:

1. *for the distribution $\mathrm{P}(z)$, the ERM principle is strictly consistent for the set of functions $Q(z, f)$, $f \in \mathcal{H}$;*
2. *for the distribution $\mathrm{P}(z)$ there is a one-sided uniform convergence over the set of functions $Q(z, f)$, $f \in \mathcal{H}$,*

$$\forall \epsilon > 0, \mathrm{P}\left(\sup_{f \in \mathcal{H}} (\mathcal{R}(f) - \mathcal{R}_n(f))_+ > \epsilon \right) \xrightarrow[n \to \infty]{} 0,$$

where $(x)_+ = \max(x, 0)$.

Notice that the theorem requires only a one-sided uniform convergence (contrary to the two-sided uniform convergence of Eq. (5.5)). This is because we want to minimize the risk, and we do not care about its maximization. Next, we discuss this result in a more quantitative way and provide sufficient conditions on the *structure* of the hypothesis space for the ERM principle to be consistant for *any* distribution.

5.1.4 Towards bounds on the risk

In this section, we restrict the learning paradigm by assuming that for any z and any f, $Q(z, f) \in \{0, 1\}$. This corresponds notably to the case of classification with the binary loss. The results presented next can be extended to a more general case (up to additional technical difficulties), but this restriction simplifies the discussion.

We have seen in the preceding section that consistency of the ERM principle has to do with a notion of uniform convergence, that somehow extends the weak law of large numbers. We start by providing a quantitative (that is, non-asymptotical) version of this law.

Theorem 5.2 (*Hoeffding's inequality*)**.** *Let $X_1, \ldots, X_n$ be i.i.d. random variables, bounded in (0,1) and of common mean $\mu = \mathrm{E}[X_1]$. Then:*

$$\forall \epsilon > 0, \quad \mathrm{P}\left(\left|\frac{1}{n}\sum_{i=1}^{n} X_i - \mu\right| > \epsilon\right) \leq 2e^{-2n\epsilon^2}.$$

Obviously, the weak law of large numbers is a corollary of this result. This is called a concentration inequality: it states how the empirical mean (which is a random variable) concentrates around its expectation. Such concentration inequalities are the base of what is called *PAC (Probably Approximately Correct)* analysis. The preceding result can be equivalently written as

$$\mathrm{P}\left(\left|\frac{1}{n}\sum_{i=1}^{n} X_i - \mu\right| \leq \epsilon\right) > 1 - 2e^{-2n\epsilon^2}.$$

Write $\delta = 2e^{-2n\epsilon^2} \Leftrightarrow \epsilon = \sqrt{\frac{\ln \frac{2}{\delta}}{2n}}$. The Hoeffding's inequality can equivalently be stated as: with probability at least $1 - \delta$, we have

$$\left|\frac{1}{n}\sum_{i=1}^{n} X_i - \mu\right| \leq \sqrt{\frac{\ln \frac{2}{\delta}}{2n}}.$$

So the result is probably (of probability at least $1 - \delta$) approximately (the error being at most $\sqrt{\frac{\ln \frac{2}{\delta}}{2n}}$) correct.

This can be directly applied to our problem (recall Eqs (5.3) and (5.4), that correspond respectively to an expectation and to and empirical expectation). Let $f \in \mathcal{H}$ be a function chosen beforehand, then with probability at least $1 - \delta$ we have

$$|\mathcal{R}(f) - \mathcal{R}_n(f)| \leq \sqrt{\frac{\ln \frac{2}{\delta}}{2n}}.$$

Unfortunately, this cannot be extended directly to uniform convergence. Indeed, probabilities are about measuring sets. What Hoeffding says, when applied to the risk, is that the measure of the set of datasets satisfying $|\mathcal{R}(f) - \mathcal{R}_n(f)| > \epsilon$ is at most δ. Now, this set of dataset depends on the function f of interest. Take another function f', the corresponding set of datasets will be different, so the measure of both sets of datasets (corresponding respectively to f and f', such that the inequality of interest is satisfied for both function) is no longer bounded by δ, but by 2δ. This is illustrated in Fig. 5.4. We write this idea more formally now.

Assume that $\mathcal{H}$ is a finite set, such that $\mathrm{Card}\,\mathcal{H} = h$. We can write

$$\begin{aligned}
\mathrm{P}\left(\sup_{f\in\mathcal{H}} |\mathcal{R}(f) - \mathcal{R}_n(f)| > \epsilon\right) &= \mathrm{P}\left(\bigcup_{f\in\mathcal{H}} \{|\mathcal{R}(f) - \mathcal{R}_n(f)| > \epsilon\}\right) \text{ (by def. of the sup)} \\
&\leq \sum_{f\in\mathcal{H}} \mathrm{P}\left(|\mathcal{R}(f) - \mathcal{R}_n(f)| > \epsilon\right) \text{ (union bound)} \\
&\leq 2he^{-2n\epsilon^2} \text{ (by Hoeffding on each term of the sum).}
\end{aligned}$$

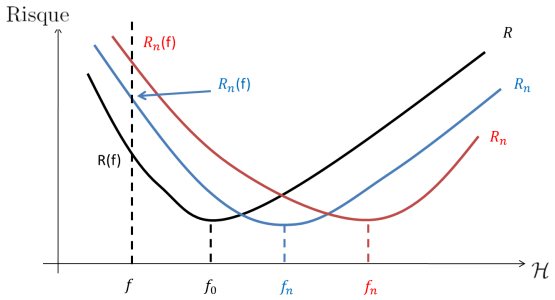


Figure 5.4: Here, $\mathcal{R}$ is the risk and $\mathcal{R}_n$ is the empirical risk for two different datasets (sampled from the same law). For a given function $f \in \mathcal{H}$, the fluctuation (or variation) of $\mathcal{R}_n(f)$ around $\mathcal{R}(f)$ is controlled by the Hoeffding inequality. However, f_n depends on the data set and the fluctuation of the associated empirical risk cannot be controlled by Hoeffding.

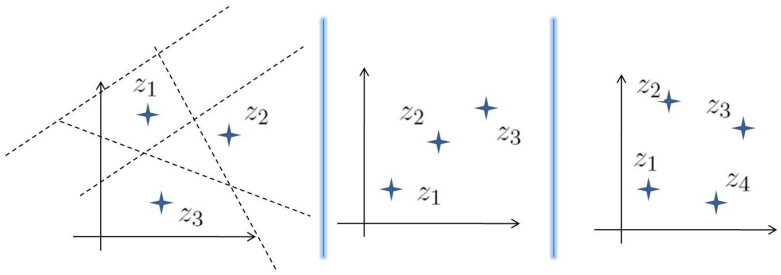


Figure 5.5: Idea for counting functions.

In other words, we can say that with probability at least $1 - \delta$, we have

$$|\mathcal{R}(f_n) - \mathcal{R}_n(f_n)| \leq \sup_{f \in \mathcal{H}} |\mathcal{R}(f) - \mathcal{R}_n(f)| \leq \sqrt{\frac{\ln \frac{2h}{\delta}}{2n}}.$$

Moreover, we have just shown that, if the hypothesis space is a finite set, then the ERM principle is strictly consistent, for any distribution (no specific assumption has been made about $\mathrm{P}(z)$, apart from the fact that the related random variables are bounded; this means that it will work for any—classification here—problem).

Unfortunately, even for the simplest cases exemplified before, the hypothesis space is uncountable and this method (Hoeffding with a union bound) does not apply. Yet, the underlying fundamental idea is good, it is just the way of counting functions which is not smart enough. Next we introduce the *Vapnik-Chervonenkis dimension*, which is a measure of the **capacity** (the way of counting functions in a smart way) of an hypothesis space, among others (*e.g.*, Rademacher averagers, shattering dimension, and so on).

Recall that we are focusing here on the case of classification. The basic idea for counting functions in a smart way is as follows: as we work with empirical risks, there is no difference between two functions that provide the same empirical risk, so they should not be counted twice. Consider the examples provided in Fig. 5.5, with 3 to 4 labels. Suppose that the classifiers (the functions of the hypothesis space) make their predictions based on a separating hyperplan (see the left figure). If two classifiers predict the same labels for the provided examples, they will have the same risk and will not be distinguishable. In the left figure, there are 8 different classifiers from this viewpoint. In the middle figure, there are only 6 different classifiers (points are aligned). In the right figure, there are 8 possible classifiers (all configurations of the labels are not possible given a linear separator). Generally speaking, given any hypothesis space and n data points, there are a maximum of 2^n possible classifiers (still relatively to the value of the associated empirical risk). We have just seen in the simple example of Fig. 5.5 that if there are more than 3 points, a

linear separator will not be able to provide all possibilities. This rough idea is formalized by the following result.

Theorem 5.3 (Vapnik & Chervonenkis, Sauer & Shelah)**.** *Define $G^{\mathcal{H}}(n)$ the growth function of a set of functions $Q(z,f)$, $f \in \mathcal{H}$, as*[4]

$$\begin{aligned} G^{\mathcal{H}}(n) &= \ln\left(\sup_{z_1,\dots,z_n\in\mathcal{Z}} \left(N^{\mathcal{H}}(z_1,\dots,z_n)\right)\right) \\ \text{with } N^{\mathcal{H}}(z_1,\dots,z_n) &= \mathrm{Card}(Q_{z_1,\dots,z_n}) \\ \text{and } Q_{z_1,\dots,z_n} &= \left\{\begin{pmatrix} Q(z_1,f) & \dots & Q(z_n,f)\end{pmatrix}^\top : f\in\mathcal{H}\right\}. \end{aligned}$$

The growth function satisfies one of the two properties:

1. *either $G^{\mathcal{H}}(n)$ is linear,*
$$G^{\mathcal{H}}(n) = n\ln 2, \forall n \in \mathbb{N}^*;$$
2. *or $G^{\mathcal{H}}(n)$ is sub logarithmic after a given rank,*
$$G^{\mathcal{H}}(n)\begin{cases} = n\ln 2 & \text{if } n \le h \\ \le h(1+\ln\frac{n}{h}) & \text{if } n > h \end{cases},$$
with h the greater integer such that $G^{\mathcal{H}}(n) = n\ln 2$.

In the first case, the Vapnik-Chervonenkis dimension is infinite, $d_{VC}(\mathcal{H}) = \infty$. In the second case, we have $d_{VC}(\mathcal{H}) = h$.

Alternatively, we can say that the Vapnik-Chervonenkis dimension of a set of indicator functions $Q(z,f)$, $f \in \mathcal{H}$ is the maximum number h of vectors $z_1,\dots,z_h$ which can be separated in all 2^h possible ways using functions of this set (*shattered* by this set of functions). It plays the role of the cardinal of this set (while being much smarter). This dimension depends on the structure of the hypothesis space, but not on the distribution of interest (not on $\mathrm{P}(z)$, so not on the specific problem at hand). It can be estimated or computed in many cases. For example, if classification is done thanks to a linear separator in $\mathbb{R}^d$, then we have $d_{\mathrm{VC}}(\mathcal{H}) = d+1$. This say that the Vapnik-Chervonenkis dimension is a measure of the **capacity** of the space (and again, it is not the sole).

The following result can be proven.

Theorem 5.4 (Bound on the risk)**.** *Let δ be in $(0,1)$. With probability at least $1-\delta$, we have*

$$\forall f \in \mathcal{H}, \quad \mathcal{R}(f) \le \mathcal{R}_n(f) + 2\sqrt{\frac{2}{n}\left(d_{VC}(\mathcal{H})\ln\frac{2en}{d_{VC}(\mathcal{H})} + \ln\frac{2}{\delta}\right)}, \tag{5.6}$$

with e being the exponential number (the mathematical constant).

This being true for any f, it is also true for f_n, the minimizer of the empirical risk. This gives a bound on the risk, based on the number of samples and on the capacity of $\mathcal{H}$. Notably, this tells that if $n \gg d_{\mathrm{VC}}(\mathcal{H})$, then the error term is small and we do not have problems of overfitting (and conversely, if we do not have enough samples, the empirical risk will be far away from the risk, leading to an overfitting problem). A direct (and important) corollary of this result is that the empirical risk minimization is strictly consistent if $d_{\mathrm{VC}}(\mathcal{H}) < \infty$.

[4] $Q_{z_1,\dots,z_n}$ is the set of all possible combinations of labels that can be computed given the functions in $\mathcal{H}$ for the data points $z_1,\dots,z_n$, $N^{\mathcal{H}}(z_1,\dots,z_n)$ is the cardinal of this set of points, bounded by 2^n, and $G^{\mathcal{H}}(n)$ is the logarithm of the supremum of these cardinals for arbitrary datasets (bounded by $n\ln 2$).

5.1.5 To go further

We have provided a short (and dense) introduction to statistical learning theory. For a deeper introduction, the interested student can follow the optional course "*apprentissage statistique*" or read the associated course material by Geist (2015a) (in french). Most of the material presented in the above sections comes from the book by Vapnik (1998), who did a seminal work in statistical learning theory. Vapnik (1999) provides a much shorter (than the book) introduction. A seminal work regarding PAC analysis is the one of Valiant (1984). Other references that might be of interest (the list being far from exhaustive and disordered, the last reference focusing more on concentration inequalities) are (Bousquet et al., 2004; Cucker and Smale, 2001; Evgeniou et al., 2000; Hsu et al., 2014; Györfi et al., 2006; Boucheron et al., 2013). For a general analysis of support vector machines (to be presented in Part III), see Guermeur (2007).

5.2 Classification, convex surrogates and calibration

In this section, we focus on the problem of classification. In this case, the input set $\mathcal{X}$ is usually a subspace of $\mathbb{R}^p$ and the output space is $\mathcal{Y} = \{1, \dots, K\}$, a finite set of labels. The natural loss for classification is the binary loss (5.2), defined as $L(y, f(x)) = \mathbb{I}_{\{y \neq f(x)\}}$, which gives the risk[5]

$$\mathcal{R}(f) = \mathrm{E}\left[L(Y, f(X))\right] = \mathrm{E}\left[\mathbb{I}_{\{y \neq f(x)\}}\right] = \mathrm{P}\left(Y \neq f(X)\right).$$

In other words, minimizing this risk corresponds to minimizing the probability of predicting a wrong label. That is why the binary loss is the natural loss for classification. The associated empirical risk is

$$\mathcal{R}_n(f) = \frac{1}{n} \sum_{i=1}^{n} \mathbb{I}_{\{y_i \neq f(x_i)\}}.$$

There are two problems with this risk:

1. with this formulation, the hypothesis space should satisfies $\mathcal{H} \subset \{1, \dots, K\}^{\mathcal{X}}$, and it is quite hard to design (for example through a parameterization) a space of functions that output labels (much harder than designing space of functions that output reals);
2. the resulting optimization problem (minimizing the empirical risk) is really hard to solve (not smooth, not convex, and so on).

Therefore, it is customary to minimize a *surrogate* (or proxy) to this risk of interest.

5.2.1 Binary classification with binary loss

We start by discussing the case of binary classification. Without loss of generality, assume that $\mathcal{Y} = \{-1, 1\}$, which will be more convenient. Recall the hypothesis space of linear parameterizations (5.1),

$$\mathcal{H} = \left\{ f_\alpha : x \to \alpha^\top \phi(x), \alpha \in \mathbb{R}^d \right\},$$

with $\phi : \mathcal{X} \to \mathbb{R}^d$ a predefined feature vector. For any x, $f_\alpha(x)$ is a scalar, while we would like to output an element of $\{-1, 1\}$. This can be done using the sign trick briefly mentioned before. Write $\mathcal{G}$ the following hypothesis space:

$$\mathcal{G} = \{g_\alpha : x \to \mathrm{sgn}(f_\alpha(x)), f_\alpha \in \mathcal{H}\} \subset \{-1, 1\}^{\mathcal{X}}.$$

We have just designed a (simple) hypothesis space for classification (through the space $\mathcal{H}$). Considering the binary loss, this leads to the following optimization problem

$$\min_{\alpha \in \mathbb{R}^d} \mathcal{R}_n(g_\alpha) = \min_{\alpha \in \mathbb{R}^d} \frac{1}{n} \sum_{i=1}^{n} \mathbb{I}_{\{y_i \neq \mathrm{sgn}(f_\alpha(x))\}} = \min_{\alpha \in \mathbb{R}^d} \frac{1}{n} \sum_{i=1}^{n} \mathbb{I}_{\{y_i \neq \mathrm{sgn}(\alpha^\top \phi(x))\}}.$$

[5] We recall that in probabilities, for an event A depending on a random variable Z, we have that $\mathrm{E}\left[\mathbb{I}_{\{A\}}\right] = \int \mathbb{I}_{\{A\}} \mathrm{dP}\left(z\right) = \int_A \mathrm{dP}\left(z\right) = \mathrm{P}\left(A\right)$.

	$\varphi(t)$ for $t \in \mathbb{R}$
hinge loss	$\max(0, 1-t)$
truncated least-squares	$(\max(0, 1-t))^2$
least-squares	$(1-t)^2$
exponential loss	e^{-t}
sigmoid loss	$1-\tanh(t)$
logistic loss	$\ln(1+e^{-t})$

Table 5.1: Some classic loss functions for binary classification, see also Eq. (5.7).

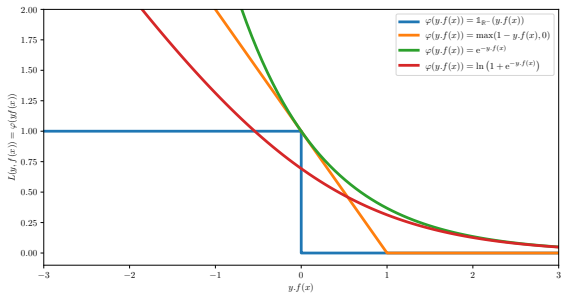


Figure 5.6: Some plots of classic loss functions for binary classification, see also Eq. (5.7).

The question is: how to solve this optimization problem? It is not convex (a very nice property in optimization), not even smooth, so there is no easy answer.

A solution is to introduce a surrogate to this risk, that works on f_α instead of g_α, and that solves (approximately) the same problem. We give a simple example now. The rational is to introduce a loss function such that the loss is low when y and $f(x)$ have the same sign, and high in the other case. Consider the following risk and its empirical counterpart:

$$\mathcal{R}(f_\alpha) = \mathrm{E}\left[e^{-Yf_\alpha(X)}\right] \text{ and } \mathcal{R}_n(f_\alpha) = \frac{1}{n}\sum_{i=1}^{n} e^{-y_i f_\alpha(x_i)}.$$

We have that $\mathcal{R}(f_\alpha) \geq 0$. To minimize this risk, we should set f_α such that $\mathrm{sgn}(f_\alpha(x_i)) = \mathrm{sgn}(y_i)$ and with a high (ideally infinite) absolute value. Therefore, minimizing this risk makes sense, from a classification perspective. Moreover, as here f_α is linear (in the parameters), one can easily show that $\mathcal{R}_n(f_\alpha)$ is a smooth (Lipschitz) and convex function. This allows using a bunch of optimization algorithms to solve the related problem, with strong guarantees to compute the global minimum.

This is called the exponential loss. More generally, write $L(y, f(x)) = \varphi(yf(x))$ the loss for a convenient and well-chosen function $\varphi : \mathbb{R} \to \mathbb{R}$, we have the following generic surrogate to binary classification:

$$\mathcal{R}(f_\alpha) = \mathrm{E}\left[\varphi(Yf_\alpha(X))\right] \text{ and } \mathcal{R}_n(f_\alpha) = \frac{1}{n}\sum_{i=1}^{n} \varphi(y_i f_\alpha(x_i)). \tag{5.7}$$

In Tab. 5.1 and Fig. 5.6, we provide some classic loss functions. Using a convex surrogate allows for solving a convex optimisation problem when minimizing the empirical risk. In other word, given some dataset $S = \{(x_1, y_1), \cdots, (x_i, y_i), \cdots, (x_N, y_N)\}$, the functional $\mathcal{R}_n(f)$ is convex according to f. Let us recall that

$$\mathcal{R}_n(f) = \frac{1}{N}\sum_{(x_i,y_i)\in S} L(x_i, y_i) = \frac{1}{N}\sum_{(x_i,y_i)\in S} \varphi(y_i.f(x_i))$$

is a sum of positive terms. Therefore, the convexity of $\mathcal{R}_n(f)$ can be deduced from the convexity of each of the $\varphi(y_i.f(x_i)) \stackrel{\text{def}}{=} K_i(f)$ terms according to f. This comes naturally from the convexity of φ, i.e.

$$\forall \lambda \in [0,1],\ \varphi(\lambda x + (1-\lambda)x') \leq \lambda\varphi(x) + (1-\lambda)\varphi(x')$$

	$\psi(s)$ for $s \in \mathbb{R}$
hinge loss	$\max(0, 1+s)$
truncated least-squares	$(\max(0, 1+s))^2$
least-squares	$(1+s)^2$
exponential loss	e^s
logistic loss	$\ln(1+e^s)$

Table 5.2: Some loss functions for cost-sensitive multiclass classification, see also Eq. (5.12).

since it enables to derive that

$$\begin{aligned} K_i\left(\lambda f + (1-\lambda)f'\right) &= \varphi\left(y_i(\lambda f + (1-\lambda)f')\left(x_i\right)\right) \\ &= \varphi\left(\lambda y_i f\left(x_i\right) + (1-\lambda)y_i f'\left(x_i\right)\right) \\ &\leq \lambda\varphi\left(y_i f\left(x_i\right)\right) + (1-\lambda)\varphi\left(y_i f'\left(x_i\right)\right) \\ &= \lambda K_i\left(f\right) + (1-\lambda)K_i\left(f'\right) \end{aligned}$$

which shows that $\mathcal{R}_n\left(f\right)$ is convex according to f. As linear functions are considered here, i.e $f(x) = f_\alpha(x) = \alpha^\top \phi(x)$, the convexity of $\mathcal{R}_n\left(f\right)$ according to f allows for asserting the convexity of $\mathcal{R}_n\left(f_\alpha\right)$ according to the parameters α, since the composition of a convex function and an affine function is convex.

Knowing what surrogate to choose is not an easy question, they have different theoretical guarantees, they lead to more or less easy to optimize problems, and so on. Notice that once one has solved $\alpha_n = \operatorname{argmin}_{\alpha\in\mathbb{R}^d} \mathcal{R}_n(f)$, for the empirical risk given in Eq. (5.7), the decision rule (the solution to the classification problem) is $g_{\alpha_n}(x) = \operatorname{sgn}(f_{\alpha_n}(x))$.

5.2.2 Cost-sensitive multiclass classification

We consider now the general cost-sensitive multiclass classification problem. Let $\mathcal{G} \subset \{1, .., K\}^{\mathcal{X}}$ be a hypothesis space of classifiers. The general risk considered here is

$$\mathcal{R}(g) = \mathrm{E}\left[c(X, g(X), Y)\right],$$

where $c(x, g(x), y)$ is the cost[6] of assigning label $g(x)$ to input x when the oracle provide the response y. For example, the binary loss corresponds to $c(x, g(x), y) = \mathbb{I}_{\{y\neq g(x)\}}$. Cost-sensitive classification might be of interest for some applications. Assume that one wants to predict if a patient has a given disease or not, given a set of physiological measurements. If the patient is predicted to be ill, further (more expensive) tests are conducted to be sure, and the patient is treated. If the patient is not predicted to be ill, nothing is done. However, is the prediction is false, the patient dies, as no treatment is given to him. Predicting that a patient is not ill while he is has a higher cost than the converse. Moreover, the result of the patient not being cured while he's ill might depend on the patient also (for example, if he is young and sportive, he won't die), so this cost might depend on the input too.

The empirical risk to be minimized is

$$\mathcal{R}_n(g) = \frac{1}{n}\sum_{i=1}^{n} c(x_i, g(x_i), y_i).$$

We have the same problem as before: designing the space $\mathcal{G}$ is difficult and the resulting optimization problem is hard, if even possible. Consider again the hypothesis space of linear parameterizations (5.1),

$$\mathcal{H} = \left\{f_\alpha : x \to \alpha^\top \phi(x), \alpha \in \mathbb{R}^d\right\},$$

From this, we would like to design a space $\mathcal{G} \subset \{1, .., K\}^{\mathcal{X}}$. Write $f_{\alpha_1,\ldots,\alpha_K} : \mathcal{X} \to \mathbb{R}^K$ the function defined as

$$f_{\alpha_1,\ldots,\alpha_K}(x) = \begin{pmatrix} f_{\alpha_1}(x) & \ldots & f_{\alpha_K}(x) \end{pmatrix}^\top \text{ such that } \forall 1 \leq k \leq K, f_{\alpha_k} \in \mathcal{H}.$$

[6] There should be no cost for assigning the correct label, that is $c(x, y, y) = 0$.

In other words, for each label k, we define a function $f_{\alpha_k} \in \mathcal{H}$ which can be interpreted as the *score* of this label. For a given input, the predicted label is the one with the higher score. The corresponding space of classifiers is

$$\mathcal{G} = \left\{ g_{\alpha_1,\dots,\alpha_K} : x \to \underset{1\leq k\leq K}{\operatorname{argmax}} f_{\alpha_k}(x), \forall 1 \leq k \leq K : f_{\alpha_k} \in \mathcal{H} \right\}.$$

There remains to define a surrogate to the risk of interest that operates on the functions f_{α_k} instead if $g_{\alpha_1,\dots,\alpha_K}$. Let $\psi : \mathbb{R} \to \mathbb{R}$ one of the function defined on Tab. 5.2, we consider the following surrogate:

$$\mathcal{R}_n(f_{\alpha_1,\dots,\alpha_n}) = \frac{1}{n}\sum_{i=1}^{n}\sum_{k=1}^{K} c(x_i,k,y_i)\psi(f_{\alpha_k}(x_i)). \tag{5.12}$$

For the functions of Tab. 5.2, if the functions f_{α_k} are linearly parameterized, the resulting empirical risk is convex. Consider for example the exponential loss, $\psi(s) = e^s$. For the loss $\sum_{k=1}^{K} c(x_i,k,y_i)\exp(f_{\alpha_k}(x_i))$ to be small, we should have $f_{\alpha_{k=y_i}}(x_i) > f_{\alpha_{k\neq y_i}}(x_i)$, which shows informally that this surrogates makes sense (generally, the constraint $\sum_{k=1}^{K} f_{\alpha_k}(x) = 0$ is added). Said otherwise, minimizing this surrogate loss should push for smaller scores for labels with larger cost. Consequently, the classifier that selects the label with maximal score should incur a small cost.

Notice that the surrogate (5.12) does not generalizes the one of Eq. (5.7): if $c(x, f(x), y) = \mathbb{I}_{\{y\neq f(x)\}}$ and if $K = 2$, both expressions are not equal. Notice also that it is not sole solution. For example, consider the following surrogate:

$$\mathcal{R}_n(f_{\alpha_1,\dots,\alpha_n}) = \frac{1}{n}\sum_{i=1}^{n}\sum_{k=1}^{K} c(x_i,k,y_i)e^{f_{\alpha_k}(x_i)-f_{\alpha_{y_i}}(x_i)}. \tag{5.13}$$

It is also a valid surrogate. Knowing what surrogate to use in the cost-sensitive multiclass classification case is a rather open question.

5.2.3 Calibration

Defining a convex surrogate loss is a common technique to reduce the computational cost of learning a classifier. However, if the resulting problem is more amenable to efficient optimization, it is unclear wether minimizing the surrogate loss results in a good accuracy (that is, if it will minimize the original loss of interest). The study of this problem is known as *calibration*. For example, write $\mathcal{R}$ the risk and $\mathcal{R}_\varphi$ its convex surrogate. The question is: if we want to have a suboptimality gap ϵ for the risk $\mathcal{R}$, how small should be the suboptimality gap for $\mathcal{R}_\varphi$? If there exists a positive-valued function δ (called a calibration function) such that

$$\mathcal{R}_\varphi(f) \leq \delta(\epsilon) \Rightarrow \mathcal{R}(f) \leq \epsilon,$$

the surrogate loss is said to be calibrated with respect to the primary loss. A deeper study of calibration is beyond the scope of this course material, but the interested student can look at the references provided next.

5.2.4 To go further

Using a convex surrogate, such as the hinge loss in binary classification (Cortes and Vapnik, 1995), is a common technique in machine learning. Bartlett et al. (2006) studies the calibration of surrogates provided in Eq. (5.7) (see also the work of Steinwart (2007) for a more general treatment). The surrogate of Eq. 5.12 has been introduced by Lee et al. (2004) in the cost-insensitive case and extended to the cost-sensitive case by Wang (2013). Ávila Pires et al. (2013) study its calibration. The surrogate of Eq. (5.13) has been proposed by Beijbom et al. (2014), the motivation being to introduce "guess-averse" estimators. An alternative to convex surrogate is to use "smooth surrogates" (with no problem of calibration, at the cost of convexity), see the work of Geist (2015b).

5.3 Regularization

We have seen in Sec. 5.1 that the number of samples should be large compared to the capacity (or richness) of the considered hypothesis space. If one has not enough samples or a too rich hypothesis space, the problem of overfitting might appear and one should choose a smaller hypothesis space. However, it is not necessarily an easy task. For example, consider a linear parameterization: how should one choose beforehand the basis functions to be removed? When working with an RKHS (roughly speaking, when using the kernel trick, see Part III), the hypothesis space is implicitly defined through a kernel (and with a Gaussian kernel, commonly used with support vector machines—see Part III again—, the corresponding Vapnik-Chervonenkis dimension is infinite) and it cannot be shrinked explicitly easily. A solution to this problem is to regularize the risk.

5.3.1 Penalizing complex solutions

Let $\mathcal{H}$ be an hypothesis space and $\mathcal{R}_n$ be the empirical risk of interest. Let $\Omega : \mathcal{H} \to \mathbb{R}$ be a function that penalizes the complexity of a candidate solution. Regularization amounts to solving the following optimization problem:

$$J_n(f) = \mathcal{R}_n(f) + \lambda\Omega(f),$$

where λ is called the regularization factor (it is a free parameter) and allows setting a compromise between the minimization of the empirical risk and the quality of the solution. Informally, this can also been seen as solving the following optimization problem:

$$\min_{f\in\mathcal{H}:\Omega(f)\leq\eta} \mathcal{R}_n(f),$$

for some value of the (free) parameter η. Instead of searching for a minimizer of the empirical risk in the whole hypothesis space, we're only looking for solutions of a maximum complexity (as measured by Ω) of η. How to solve the corresponding optimization problem obviously depends on the risk and on the measure of complexity of solutions. The theoretical analysis of the related solutions also depends on these instantiations. We give some examples of possible choices for Ω in the next section.

5.3.2 Examples

For simplicity, we consider here a space of parameterized functions:

$$\mathcal{H} = \{f_\alpha : \mathcal{X} \to \mathbb{R}, \alpha \in \mathbb{R}^d\}.$$

We give some classic regularization terms:

ℓ_2-penalization: it is defined as

$$\Omega(f_\alpha) = \|\alpha\|_2^2 = \sum_{j=1}^{d} \alpha_j^2$$

and is also known as Tikhonov regularization. It is often used with a linear parameterization and an ℓ_2-loss, providing the regularized linear least-squares (that admit an analytical solution);

ℓ_0-penalization: this is sometime called the norm of sparsness, or ℓ_0-norm, even if it is not a norm. It is defined as

$$\Omega(f_\alpha) = \|\alpha\|_0 = \text{Card}\left(\{j \in \{1,\dots,d\} : \alpha_j \neq 0\}\right).$$

The more coefficients are different from zero (the less sparse is the solution), the more penalized is f_α. Notice that solving the corresponding optimization problem is intractable in general;

ℓ_1-penalization: it is defined as

$$\Omega(f_\alpha) = \|\alpha\|_1 = \sum_{j=1}^{d} |\alpha_j|.$$

This is often uses as a proxy for the ℓ_0 norm, as it also promotes sparse solutions.

We do not provide more examples, but there exist much more way of penalizing an empirical risk.

5.3.3 To go further

A (not always convenient) alternative to regularization is structural risk minimization[7] (Vapnik, 1998). For a risk based on the ℓ_2-loss and a linear parameterization, ℓ_2-penalization has been proposed by Tikhonov (1963) and ℓ_1-penalization by Tibshirani (1996a) (it is known as LASSO, least absolute shrinkage and selection operator, an efficient method for solving it being proposed by Efron et al. (2004b)). Under the same assumptions (that is, a linear least-squares problem), see Hsu et al. (2014) for an analysis of ℓ_2-penalization and Bunea et al. (2007) for ℓ_1-penalization (among many others). For regularization in support vector machines (and the link between margin maximization and risk minimization), see for example Evgeniou et al. (2000).

[7] Roughly speaking, assume that you have an increasing set of hypothesis spaces, $\mathcal{H} = \bigcup_k \mathcal{H}_j : \mathcal{H}_1 \subset \cdots \subset \mathcal{H}_k \ldots$. One can minimize the empirical risk for each subspace end evaluates the corresponding bound on the risk (see for example Eq. (5.6)), and choose the solution with the smaller upper-bound on the risk.

Chapter 6

Preprocessing

6.1 Selecting and conditioning data

6.1.1 Collecting data

There might be different starting points in machine learning depending on your objectives. If your aim is to solve a particular problem with particular data, you necessarily have to begin with collecting samples and, if you work on a classification problem, to label all the collected samples. However, if your aim is rather on developing new machine learning techniques, then it is relevant to work on popular benchmarks on which other techniques have already been tried. There are various datasets that are commonly used for this second aspect of developing a new machine learning algorithm. While reading machine learning papers you might certainly notice the use of the following datasets :

- the UCI Machine Learning Repository (http://archive.ics.uci.edu/ml/) which is hosting a large collection of datasets such as the IRIS dataset for classification (150 samples with 4 features characterizing flowers of 3 classes), the diabete dataset (442 samples with 10 features and a target to regress)
- the MNIST dataset (http://yann.lecun.com/exdb/mnist/) with 28×28 black and white images of handwritten digits, with a total of 70000 samples
- face image databases, various links to databases are provided at http://www.face-rec.org/databases/

There are actually plenty of datasets that can be used, a lot of them being listed on the Wikipedia page https://en.wikipedia.org/wiki/List_of_datasets_for_machine_learning_research.

There are also some online platforms, such as Kaggle[1], on which competitions are proposed with associated datasets.

6.1.2 Conditioning the data

Dealing with ordinal and categorical features

The machine learning algorithms we consider in this document work with vectorial and numerical inputs. It turns out that some datasets (e.g. medical) contain features that are not numerical but rather ordinal or categorical. In these two situations, the variable is actually a label rather than a number. The difference between ordinal and categorical data is that ordinal data are naturally ordered (e.g. ratings as excellent, very good, good, fair, poor), while categorical features have no natural ordering (e.g. countries France, United States, Spain, ...).

Let us now consider different encoding schemes of ordinal and categorical features. Suppose we have a feature being the rating of a movie with three possible values : excellent, fair and poor.

[1] https://www.kaggle.com

In order to keep the ordering of the values when encoding the feature, one may use an increasing number with the following encoding :

Ordinal Feature value	poor	fair	excellent
Numerical feature value	-1	0	1

When the dataset contains categorical features, we cannot use the same encoding as for ordinal features simply because we otherwise induce an ordering of the feature values that is not in the data. Let us consider a dataset of people with a feature indicating their nationality : $\mathcal{F} = \{American, Spanish, German, French\}$. Sometimes, the feature value might appear as an integer because someone decided to encode categorical features with a number, e.g. American=0, Spanish=1, German=2, French=3 so that the fact that a feature is actually categorical might not immediately pop out. One possible encoding scheme for handling such categorical features is to use a so-called one-hot encoding. The idea is actually to create $|\mathcal{F}|$ features, each except one being equal to zero.

Categorical feature value	American	Spanish	German	French
Numerical feature values	$[1, 0, 0, 0]$	$[0, 1, 0, 0]$	$[0, 0, 1, 0]$	$[0, 0, 0, 1]$

Working with text documents

Text documents are specific in the sense that they are only symbolic. In order to feed a machine learning algorithm which requires feature vectors as input, one has to convert these text documents into feature vectors. One way to do so is to use the *bag-of-words* representation. In the bag-of-word model, we take all the vocabulary inside the corpus you have, keeping only the words as case-insensitive and dropping out the various punctuation elements. You then build up a vector with the size of the vocabulary as the number of dimensions; Then you count the occurrence of each word and fill in the vector accordingly. Therefore, each text document gets represented as a feature vector. The bag of words encoding can be extended to take into account patterns of words in the documents with *n-grams* as you would do, for example, when computing Gabors on an image. N-grams are patterns of n consecutive words. Clearly, as we are working with text documents that have some grammatical structure, you would never encounter all the possible n-grams you could imagine from a given vocabulary. If you were to known the true distribution of text documents, it would be a manifold of lower dimension than the vector space you can compute with the n-grams and therefore, your input vectors would be much sparser than when using bag-of-words but at the same time focusing on patterns of words which can be more informative.

Imputation of missing values

It might be that, for some reasons, the dataset contains entries with some missing features. This can be indicated in various ways depending on the dataset. For example, the Heart Disease dataset[2] has some entries with missing features indicated as the special value -9, some other datasets might use a blank space to indicate a missing value. One easy way to deal with missing values is actually to drop the entries which have at least one missing feature but this might dramatically decrease the number of data in your dataset. One may rather consider imputing the missing values, i.e. filling in the missing feature with something. Several possibilities can be considered to define which value to put in the missing slots as for example the mean, median or most frequent feature value over the whole dataset,

Feature scaling

Given a set $\{x_0, \cdots, x_i, \cdots, x_{N-1}\} \in \mathbb{R}^d$ of input vectors, it is usual that the variables do not vary in the same ranges. There are several machine learning algorithms that compute Euclidean distances in input space, such as k-nearest neighbors for clustering or support vector machines with radial kernels. In this situation, if one dimension is in range, say, some order of magnitudes larger than the others, it will dominate the computation of the distance and therefore hardly take into account the other dimensions. There are also some cases where the learning algorithm can

[2] http://archive.ics.uci.edu/ml/datasets/Heart+Disease

converge faster if the features are normalized. An example of this situation is when you perform gradient descent of, for example, a linear regression model with a quadratic cost[3]. If the features are normalized, the loss will be much more circular symmetric than if they are not. In this case, a gradient descent would point toward the minimum and the algorithm would converge faster. As we shall see later, for example when speaking about neural networks, we will minimize cost functions with penalties on the parameters we are looking for. This penalty will actually ensure that we do not overfit the training set. If we do not standardize the data, the penalty will typically penalize much more the dimensions which cover the largest range which is not a desired effect.

There are various ways to ensure that the features belong to the same range. Let us denote by $x_{i,j}$ the j-th feature of the input x_i, you might :

- scale each features so that they belong to the range $[0, 1]$:

$$\forall i \in [0, N-1], \forall j \in [0, d], {x'}_{i,j} = \frac{x_{i,j} - \min_{k \in [0,N-1]} x_{k,j}}{\max_{k \in [0,N-1]} x_{k,j} - \min_{k \in [0,N-1]} x_{k,j}}$$

- standardize each features so that they have zero mean and unit variance :

$$\forall i \in [0, N-1], \forall j \in [0, d], {x'}_{i,j} = \frac{x_{i,j} - \overline{x_{.,j}}}{\sigma_j}$$

$$\overline{x_{.,j}} = \frac{1}{N} \sum_{k=0}^{N-1} x_{k,j}$$

$$\sigma_j = \sqrt{\frac{1}{N} \sum_{k=0}^{N-1} (x_{k,j} - \overline{x_{.,j}})^2}$$

Remember that feature scaling is performed based on the training data. This has two implications. The first is that you must remember the parameters of the scaling so that if you apply the learned predictor on a new input, you actually make use of the same scaling than the one used for learning the predictor. Also, as the scaling parameters are computed from the training set, you should in principle include the scaling step in the whole pipeline that has to be cross-validated.

6.2 Dimensionality reduction

Typical data we encounter in machine learning live in a large dimensional space. This does not necessarily mean that the data span the whole input space. These may rather occupy a space of lower dimension or at least the data might be projected in such a low dimensional space without disrupting the relationships between the data. Discovering this space of lower dimension has many interests in machine learning. If we focus on less than three dimensions, we can then afford visualizing the dataset we are working with. We may also dramatically decrease the size of the input vectors we are working with and therefore speed up algorithms working on these data. Finally, projecting our fixed number of input vectors into a lower dimensional space and then learn from these fewer features can help fighting against the *curse of dimensionality* especially when the number of original features is large. In this case, your data can quickly get sparse in the input space and it is therefore hard to properly generalize. For example, if one has N input vectors living in a d dimensional space and suppose that, in order to be efficient, our predictor requires to distinguish between 10 values in each dimension, you end up with $10^d S$ different configurations to distinguish, a number that grows exponentially with the number of features.

Optimally transforming original data that leave in a high dimensional space into a lower dimensional space is the focus of dimensionality reduction methods. This broad definition gives the essence of dimensionality reduction. There are still some elements to be defined such as what we mean by "optimally" and which are the type of transformations that we consider. Reducing the dimension of some input data can be done by selecting a subset of the original dimensions, which is coined the term *feature selection*, or by computing new features from the original dimensions

[3] which is considered just for illustrative purpose as it can be solved analytically

which is coined the term *feature extraction*. In the next sections we focus on three specific feature extraction methods, namely Principal Component Analysis (PCA), Kernel PCA and Locally Linear Embedding. These are derived from different definitions of optimality and transformations applied to the data.

6.2.1 Variable selection

Forewords The elements in this section are mainly the integration of a former document. That document(Geist, 2014) was written in French and is here translated in English. In this section, we provide a quick overview on variable selection. The interested reader is referred to recent reviews on this topic(Guyon and Elisseeff, 2003; Somol et al., 2010).

Taxonomy Suppose we are given input vectors $\{x_0, \cdots, x_i, \cdots, x_{N-1}\} \in \mathbb{R}^d$ and an output $\{y_0, \cdots, y_i, \cdots, y_{N-1}\}$ to predict. It might be, for some reason, that all the input dimensions are not necessary to predict the output. To take a dummy example to fix the ideas, if one dimension of all the input vectors is constant while the target to predict is varying, this dimension does not bring any information for a predictor to predict the output. The problem of finding a subset of the original dimensions that is hopefully sufficient to produce a good or even better predictor of the output is referred to as *variable selection*. There are three main approaches to variable selection : *filters*, *wrappers* and *embedded*. In the two first approaches, variables are selected or dropped out according to a heuristic for the filters (e.g. correlation or mutual information between the dimensions and the output) and according to an estimation of the real risk for the wrappers (which implies computing several predictors minimizing a given loss). The filters are clearly less computationally expensive than the wrappers but also less theoretically grounded. The embedded method introduces a penalty term in the loss to be minimized (e.g. L1 penalty) which tends to take into account fewer than the available dimensions in the computed predictor.

The LASSO (*Least Absolute Shrinkage and Selection Operator*, (Tibshirani, 1996b)) algorithm is one example of embedded method. It considers a ℓ_1 penalty to a linear least square loss[4]. The resulting loss reads :

$$\mathcal{R}^S_{\text{emp}}(\theta) + \lambda|\theta|_1 = \frac{1}{N}\sum_{i=0}^{N-1}(y_i - \theta^{\mathsf{T}}x_i)^2 + \lambda|\theta|_1$$

The ℓ_1 penalty term will promote sparse predictors, i.e. predictors in which the parameter vector θ will have null coefficients, and therefore performs variable selection.

Searching in the variable space Let us formalize the variable selection problem. The inputs are vectors $x \in \mathbb{R}^d$. A variable of x is one of its d components, which we denote x_j : $x = (x_0 x_1 ... x_{d-1})^{\mathsf{T}}$. We define a set of k out of the d variables as $\mathcal{X}_{\sigma,k}$:

$$\mathcal{X}_{\sigma,k} = \left\{x_j : j \in \sigma^{-1}(1), \sigma : [|1..d|] \to \{0,1\}, \left|\sigma^{-1}(1)\right| = k\right\}$$

The function σ indicates, for each variable, if we should keep it or not. We denote by $\mathcal{X}_d$ the set with all the variables. There are $\frac{d!}{k!(d-k)!}$ possible $\mathcal{X}_{\sigma,k}$ sets with k variables and the search space contains actually 2^d sets. We suppose, and this will be detailed latter on, that we have a score function J which gives the quality $J(\mathcal{X}_{\sigma,k})$ of the k selected variables. Some examples of score functions will be given in the next paragraphs. Let us introduce some quantities :

- the individual relevance S_0 defined as :

$$S_0(x_i) = J(\{x_i\}), 1 \leq i \leq d$$

- given a set $\mathcal{X}_{\sigma,k}$, the relevance $S^-_{\mathcal{X}_{\sigma,k}}$ of the variables $x_j \in \mathcal{X}_{\sigma,k}$ is defined as :

$$\forall x_j \in \mathcal{X}_{\sigma,k}, S^-_{\mathcal{X}_{\sigma,k}}(x_j) = J(\mathcal{X}_{\sigma,k}) - J(\mathcal{X}_{\sigma,k} \setminus \{x_j\})$$

[4] the regressor is linear in the intput $f(\theta, x) = \theta^{\mathsf{T}}x$

- given a set $\mathcal{X}_{\sigma,k}$, the relevance $S^+_{\mathcal{X}_{\sigma,k}}$ of the variables $x_j \in \mathcal{X}_d \setminus \mathcal{X}_{\sigma,k}$ is defined as :

$$\forall x_j \in \mathcal{X}_d \setminus \mathcal{X}_{\sigma,k}, S^+_{\mathcal{X}_{\sigma,k}}(x_j) = J(\mathcal{X}_{\sigma,k} \cup \{x_j\}) - J(\mathcal{X}_{\sigma,k})$$

We can now define few additional notions which allow to estimate how good it is to add or remove a variable. Given a set of variables $\mathcal{X}_{\sigma,k}$

- $x_j \in \mathcal{X}_{\sigma,k}$ is the *most relevant variable* iif

$$x_j = \underset{x_k \in \mathcal{X}_{\sigma,k}}{\operatorname{argmax}} S^-_{\mathcal{X}_{\sigma,k}}(x_k)$$

- $x_j \in \mathcal{X}_{\sigma,k}$ is the *less relevant variable* iif

$$x_j = \underset{x_k \in \mathcal{X}_{\sigma,k}}{\operatorname{argmin}} S^-_{\mathcal{X}_{\sigma,k}}(x_k)$$

- $x_j \in \mathcal{X}_d \setminus \mathcal{X}_{\sigma,k}$ is the *next most relevant variable* iif

$$x_j = \underset{x_k \in \mathcal{X}_d \setminus \mathcal{X}_{\sigma,k}}{\operatorname{argmax}} S^+_{\mathcal{X}_{\sigma,k}}(x_k)$$

- $x_j \in \mathcal{X}_d \setminus \mathcal{X}_{\sigma,k}$ is the *next less relevant variable* iif

$$x_j = \underset{x_k \in \mathcal{X}_d \setminus \mathcal{X}_{\sigma,k}}{\operatorname{argmin}} S^+_{\mathcal{X}_{\sigma,k}}(x_k)$$

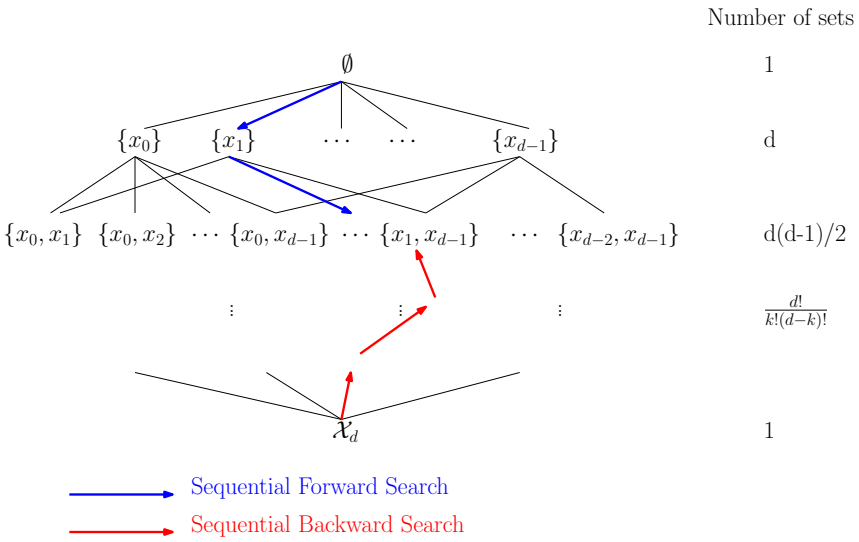


Figure 6.1: Forward and Backward sequential search (SFS, SBS). See the text for details.

We can now introduce different strategies for building a set of variable $\mathcal{X}_{\sigma,k}$. Starting from an empty set of variable $\mathcal{X} = \emptyset$, one can be greedy with respect to the score and add at each step the *next most relevant variable* until getting the desired number of variables. This strategy is called *Sequential Forward Search* (SFS). Another possibility is to start from the full set $\mathcal{X} = \mathcal{X}_d$ and drop out variables by removing the *less relevant variable* until getting a desired number of variables. This strategy is called *Sequential Backward Search*(SBS). These are the most classical strategies, illustrated on figure 6.1.

The SFS and SBS are heuristics and not necessarily optimal. It might be for example, following the SFS strategy, that an added variable renders a previously added variable unnecessary. In general, there is no guarantee that the SFS and SBS strategies find the optimal set of variables among the exponentially growing number of configurations 2^d. There are other strategies that have been proposed in the literature as the *Sequential Floating Forward Search* (SFFS), *Sequential Floating Backward Search* (SFBS) introduced in (Pudil et al., 1993) which combines forward and backward steps. This algorithm has been extended in (Somol et al., 1999) and additional variants are presented in (Somol et al., 2010).

Filter and Wrapper methods Having introduced strategies for building up a set of variables $\mathcal{X}_{\sigma,k}$, we still need to define the score function J. The filter methods use heuristics in the definition of J while wrappers define the score function J from an estimation of the real risk of the predictor.

A first example of filter heuristic is the *Correlation-based feature selection* (CSF) (Hall, 1999; Guyon and Elisseeff, 2003). It looks for a set of variables that balance the correlation between the variables and the output to predict y and the correlation between the variables themselves. The strategy is to keep features highly correlated with the output, yet uncorrelated to each other. Namely, given a training set $\{(x_i, y_i), 0 \leq i \leq N-1\}$ the score J of a set $\mathcal{X}_{\sigma,k}$ is defined as :

$$
\begin{aligned}
J_{\texttt{CSF}}\left(\mathcal{X}_{\sigma,k}\right) &= \frac{k\overline{r}(\mathcal{X}_{\sigma,k}, y)}{\sqrt{k(k-1)\overline{r}(\mathcal{X}_{\sigma,k}, \mathcal{X}_{\sigma,k})}} \\
\overline{r}(\mathcal{X}_{\sigma,k}, y) &= \frac{1}{k} \sum_{i \in \sigma^{-1}(1)} r(x_i, y) \\
\overline{r}(\mathcal{X}_{\sigma,k}, \mathcal{X}_{\sigma,k}) &= \frac{1}{k(k-1)} \sum_{i,j \in \sigma^{-1}(,) i \neq j} r\left(x_i, x_j\right)
\end{aligned}
$$

with r a measure of correlation, for example :

$$
\begin{aligned}
r\left(x_i, y\right) &= \frac{\frac{1}{N}\sum_{k=0}^{N-1} x_{k,i} y_k}{\left(\frac{1}{N}\sum_{k=0}^{N-1}(x_{k,i})^2\right)\left(\frac{1}{N}\sum_{k=0}^{N-1}(y_k)^2\right)} \\
r\left(x_i, x_j\right) &= \frac{\frac{1}{N}\sum_{k=0}^{N-1} x_{k,i} x_{k,j}}{\left(\frac{1}{N}\sum_{k=0}^{N-1}(x_{k,i})^2\right)\left(\frac{1}{N}\sum_{k=0}^{N-1}(x_{k,j})^2\right)}
\end{aligned}
$$

where $x_{j,i}$ denotes the i-th component of the input x_j. One can use other correlation measures, such as mutual information(Hall, 1999). Other filter heuristics have been proposed and some of them are reviewed in(Zhao et al., 2008).

The wrappers use an estimation of the real risk as the score function J. Denote $x^\sigma \in \mathbb{R}^k$ the vector for which only the k components in $\mathcal{X}_{\sigma,k}$ are retained. Suppose we have a training set $\{(x_i, y_i), 0 \leq i \leq N-1\}$ and a validation set $\{(x'_i, y'_i), 0 \leq i \leq M-1\}$. Let us denote $\hat{f}$ the predictor obtained from a given learning algorithm on the training set $\{(x_i^\sigma, y_i), 0 \leq i \leq N-1\}$. The score $J\left(\mathcal{X}_{\sigma,k}\right)$ can be defined as :

$$
J\left(\mathcal{X}_{\sigma,k}\right) = \frac{1}{M}\sum_{i=0}^{M-1} L\left(y'_i, \hat{f}\left(x'^{\sigma}_i\right)\right)
$$

with L a given loss (usually strongly dependent on the considered learning algorithm). Other estimation of the real risk could have been used, such as the K-fold cross-validation or bootstrapping.

Embedded methods The embedded methods make use of the specifics of the learning algorithm you consider and are therefore usually dependent on this chosen algorithm. For example, the support vector machines (SVM) can be considered as an embedded approach as they choose the support vectors among the training set (in which case, basis functions are selected rather than variables by themselves). LASSO (Tibshirani, 1996b) is a learning algorithm which has a ℓ_1 penalty term constraining weights on some of the variables to be zero. Related to LASSO, some other methods use a similar penalty as LARS (Efron et al., 2004a) or elastic net (Zou and Hastie, 2003). Regression trees[5](Breiman et al., 1983) also have internal mechanisms for variable selection.

Another approach, which we might rather consider as a model selection approach (in the sense that it selects an hypothesis space, which is actually identical to variable selection when the model is linear), is called complexity regularization. It consists in adding a penalty term to the empirical risk to be minimized, this penalty term being input data dependent (Barron, 1988; Bartlett et al., 2002; Wegkamp, 2003; Lugosi and Wegkamp, 2004). The rationale is to get Vapnik-Chervonenkis

[5]See chapter 20

type bounds (see chapter5) but actually dependent on the considered data and therefore, allowing a better estimation of the risk for selecting a model, with theoretical guarantees. This type of approach is complex and mathematically involved and will not be detailed further.

6.2.2 Principal Component Analysis (PCA)

Problem statement: Finding an affine subspace minimizing the reconstruction error

Suppose we are given N input vectors $\{x_0, \cdots, x_i, \cdots, x_{N-1}\} \in \mathbb{R}^d$. We want to summarize these d dimensional vectors by only r dimensions with $r < d$ (but typically $r \ll d$), each of these dimensions being a linear combination of the original dimensions. We consider a situation a bit more restrictive where we want the dimensions to be orthogonal to each other. This problem is known as the *Principal Component Analysis* problem and was introduced by (Pearson, 1901). It can be defined[6] as finding an affine transformation of the data so that the error we make by reconstructing the data from their projections is minimized. As an example, consider the case depicted on Fig. 6.2 where we have data in $d = 2$ dimensions and we are trying to summarize these data by a single number, i.e. we are looking for a line on which to project the data. This is like if the line we are looking for was connected to the data with little springs. Each spring pulls the line so that it ultimately reaches an equilibrium. For this one dimensional case, the line is defined by its origin w_0 and direction w_1.

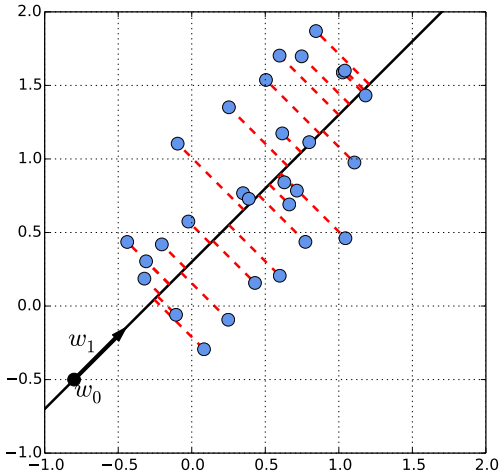


Figure 6.2: Given a set of data points, we seek a line on which to project the data so that the sum of the norms of the vectors (red dashed) connecting the data points to their projection is minimized. This is the problem we solve when we are looking for the first principal component of the data points.

Formally, the problem can be defined as :

$$\min_{\{w_0,\cdots,w_j,\cdots,w_r\}\in\mathbb{R}^d} \sum_{i=0}^{N-1} \left| x_i - w_0 - \sum_{j=1}^{r} (w_j^\intercal.(x_i - w_0))w_j \right|_2^2 \quad \text{subject to } w_i^\intercal.w_j = \delta_{i,j} \tag{6.1}$$

Let us denote by $\mathbf{W} \in \mathcal{M}_{d,r}(\mathbb{R})$ the matrix in which the columns are the vectors $\{w_1, \cdots, w_i, \cdots, w_r\}$:

$$\mathbf{W} = \left(\begin{array}{c|c|c|c|c} w_1 & \cdots & w_i & \cdots & w_r \end{array} \right)$$

[6]There exists other equivalent ways to derive the principal components such as finding projection axis maximizing the variance of the projected data. We go back on this equivalence later in this section.

The constrained optimization problem (6.1) can be written in matrix form as :

$$\min_{w_0 \in \mathbb{R}^d, \mathbf{W} \in \mathcal{M}_{d,r}(\mathbb{R})} \sum_{i=0}^{N-1} \left| x_i - w_0 - \mathbf{W}.\mathbf{W}^\intercal.(x_i - w_0) \right|_2^2 \text{ subject to } \mathbf{W}^\intercal.\mathbf{W} = \mathbf{I}_r$$

$$\min_{w_0 \in \mathbb{R}^d, \mathbf{W} \in \mathcal{M}_{d,r}(\mathbb{R})} \sum_{i=0}^{N-1} \left| (\mathbf{I}_d - \mathbf{W}.\mathbf{W}^\intercal).(x_i - w_0) \right|_2^2 \text{ subject to } \mathbf{W}^\intercal.\mathbf{W} = \mathbf{I}_r \tag{6.2}$$

The above matrix form could have be written directly if we have in mind that, given $\{w_1, \cdots, w_j, \cdots, w_r\}$ are orthogonal unit vectors, $\mathbf{W}$ is the matrix of an orthogonal projection on the r dimensional subspace generated by $\{w_1, \cdots, w_j, \cdots, w_r\}$. The norm of the residual $x - \hat{x}$, where $\hat{x} = \mathbf{W}.\mathbf{W}^\intercal x$ is the projection of x on the subspace generated by $\{w_1, \cdots, w_j, \cdots, w_r\}$, is $|(\mathbf{I} - \mathbf{W}.\mathbf{W}^\intercal).x|_2$.

Before expanding a little bit the expression inside (6.2), we first note that :

$$(\mathbf{I}_d - \mathbf{W}.\mathbf{W}^\intercal)^\intercal = (\mathbf{I}_d - \mathbf{W}.\mathbf{W}^\intercal)$$

Second, as $\mathbf{W}^\intercal.\mathbf{W} = \mathbf{I}_r$, therefore :

$$(\mathbf{W}.\mathbf{W}^\intercal)^2 = \mathbf{W}.\mathbf{W}^\intercal\mathbf{W}\mathbf{W}^\intercal = \mathbf{W}.\mathbf{W}^\intercal$$

The matrix $\mathbf{W}.\mathbf{W}^\intercal$ is therefore idempotent. For any idempotent matrix M, we also know that $\mathbf{I}-\mathbf{M}$ is idempotent as $(\mathbf{I}-\mathbf{M})^2 = \mathbf{I}-2\mathbf{M}+\mathbf{M}^2 = \mathbf{I}-\mathbf{M}$. Putting all the previous steps together, we end with :

$$(\mathbf{I}_d - \mathbf{W}.\mathbf{W}^\intercal)^\intercal.(\mathbf{I}_d - \mathbf{W}.\mathbf{W}^\intercal) = (\mathbf{I}_d - \mathbf{W}.\mathbf{W}^\intercal)$$

We are now ready for expanding eq. (6.2) :

$$\begin{aligned}
\sum_{i=0}^{N-1} \left| (\mathbf{I}_d - \mathbf{W}.\mathbf{W}^\intercal).(x_i - w_0) \right|_2^2 &= \sum_{i=0}^{N-1} (x_i - w_0)^\intercal (\mathbf{I}_d - \mathbf{W}.\mathbf{W}^\intercal)^\intercal (\mathbf{I}_d - \mathbf{W}.\mathbf{W}^\intercal)(x_i - w_0) \\
&= \sum_{i=0}^{N-1} (x_i - w_0)^\intercal (\mathbf{I}_d - \mathbf{W}.\mathbf{W}^\intercal)(x_i - w_0)
\end{aligned}$$

Deriving w_0

We now solve eq. (6.2) with respect to w_0 by computing the derivative with respect to w_0 and setting it to zero[7].

$$\begin{aligned}
\frac{d}{dw_0} \sum_{i=0}^{N-1} \left| (\mathbf{I}_d - \mathbf{W}.\mathbf{W}^\intercal).(x_i - w_0) \right|_2^2 &= -2 \sum_{i=0}^{N-1} (\mathbf{I}_d - \mathbf{W}.\mathbf{W}^\intercal)(x_i - w_0) \\
&= -2(\mathbf{I}_d - \mathbf{W}.\mathbf{W}^\intercal) \sum_{i=0}^{N-1} (x_i - w_0) \\
\frac{d}{dw_0} \sum_{i=0}^{N-1} \left| (\mathbf{I}_d - \mathbf{W}.\mathbf{W}^\intercal).(x_i - w_0) \right|_2^2 = 0 &\Leftrightarrow (\mathbf{I}_d - \mathbf{W}.\mathbf{W}^\intercal) \sum_{i=0}^{N-1} (x_i - w_0) = 0 \\
&\Leftrightarrow (\mathbf{I}_d - \mathbf{W}.\mathbf{W}^\intercal)(\frac{1}{N} \sum_{i=0}^{N-1} x_i - w_0) = 0
\end{aligned}$$

The vectors $\mathbf{u}$ satisfying $(\mathbf{I}_d - \mathbf{W}.\mathbf{W}^\intercal).\mathbf{u} = 0$ are the vectors belonging to the subspace generated by the column vectors of $\mathbf{W}$, i.e. by the vectors $\{w_1, \cdots, w_j, \cdots, w_r\}$. This actually makes sense if one thinks of how an affine subspace is defined : the origin w_0 of the affine subspace can be translated by any linear combination of the vectors $\{w_1, \cdots, w_j, \cdots, w_r\}$ and we still

[7] For computing the derivative, we note that for any vectors $\mathbf{x}$, matrix $\mathbf{A}$, and vector functions $\mathbf{u}(\mathbf{x}), \mathbf{v}(\mathbf{x})$, $\frac{d\mathbf{u}^\intercal \mathbf{A} \mathbf{v}}{d\mathbf{x}} = \frac{d\mathbf{u}}{d\mathbf{x}} \mathbf{A}\mathbf{v} + \frac{d\mathbf{v}}{d\mathbf{x}} \mathbf{A}^\intercal \mathbf{u}$.

get the same affine subspace. For the 1D example on Fig. 6.2, this means translating w_0 along w_1. Finally, we note that the value of the function (6.2) to be minimized is the value for any $w_0 = \frac{1}{N}\sum_{i=0}^{N-1} x_i + h, h \in \text{span}\{w_1, \cdots, w_j, \cdots, w_r\}$:

$$\begin{aligned} & \forall h \in \text{span}\{w_1, \cdots, w_j, \cdots, w_r\}, (\mathbf{I}_d - \mathbf{W}.\mathbf{W}^\intercal).h = 0 \\ \Rightarrow \quad & \forall h \in \text{span}\{w_1, \cdots, w_j, \cdots, w_r\}, \\ & (\mathbf{I}_d - \mathbf{W}.\mathbf{W}^\intercal).(x_i - \frac{1}{N}\sum_{i=0}^{N-1} x_i - h) = (\mathbf{I}_d - \mathbf{W}.\mathbf{W}^\intercal).(x_i - \frac{1}{N}\sum_{i=0}^{N-1} x_i) \end{aligned}$$

We can also note that the optimization problem defined by eq. (6.2) is the same whatever w_0 as soon as $w_0 = \frac{1}{N}\sum_{i=0}^{N-1} x_i + h, h \in \text{span}\{w_1, \cdots, w_j, \cdots, w_r\}$. So let us take $w_0 = \frac{1}{N}\sum_{i=0}^{N-1} x_i = \bar{x}$, which means that the data get centered by the sample mean of $\{x_0, \cdots, x_i, \cdots, x_{N-1}\}$ before being projected. If we look back to the example drawn on fig 6.2, we clearly see that we may have moved w_0 along w_1 without changing the line on which the data are projected. In general, the fact that w_0 is defined up to a $h \in \text{span}\{w_1, \cdots, w_j, \cdots, w_r\}$ simply means that the origin of the hyperplane $\text{span}\{w_1, \cdots, w_j, \cdots, w_r\}$ can be defined up to a translation within this hyperplane.

Deriving the principal components vectors $\{w_1, \cdots, w_j, \cdots, w_r\}$

We rewrite the optimization problem eq. (6.2) as :

$$\min_{\mathbf{W} \in \mathcal{M}_{d,r}(\mathbb{R})} \sum_{i=0}^{N-1} \left|(\mathbf{I}_d - \mathbf{W}.\mathbf{W}^\intercal).(x_i - \bar{x})\right|_2^{\,2} \text{ subject to } \mathbf{W}^\intercal.\mathbf{W} = \mathbf{I}_r \tag{6.3}$$

For simplicity, let us denote $\tilde{x_i} = x_i - \bar{x}$ and let us expand a little bit the inner term of eq (6.3) :

$$\left|(\mathbf{I}_d - \mathbf{W}.\mathbf{W}^\intercal).\tilde{x_i}\right|_2^{\,2} = \tilde{x_i}^\intercal(\mathbf{I}_d - \mathbf{W}.\mathbf{W}^\intercal)^\intercal\tilde{x_i} = \tilde{x_i}^\intercal\tilde{x_i} - \tilde{x_i}^\intercal\mathbf{W}.\mathbf{W}^\intercal\tilde{x_i}$$

Also, one may notice the following :

$$\sum_{i=0}^{N-1} \tilde{x_i}^\intercal\mathbf{W}\mathbf{W}^\intercal\tilde{x_i} = \sum_{i=0}^{N-1} (\mathbf{W}^\intercal\tilde{x_i})^\intercal(\mathbf{W}^\intercal\tilde{x_i}) = \sum_{i=0}^{N-1}\sum_{j=1}^{r}(w_j^\intercal\tilde{x_i})^2 = \sum_{j=1}^{r}\sum_{i=0}^{N-1}(\tilde{x_i}^\intercal w_j)^2 = \sum_{j=1}^{r} w_j^\intercal\tilde{\mathbf{X}}\tilde{\mathbf{X}}^\intercal w_j$$

where $\tilde{\mathbf{X}} = (\tilde{x}_1|\cdots|\tilde{x}_{N-1})$, i.e. the matrix with columns $\tilde{x_i}$. This matrix is the so-called *sample covariance matrix*. Therefore the minimization problem (6.3) is equivalent to the following maximization problem :

$$\max_{\mathbf{W} \in \mathcal{M}_{d,r}(\mathbb{R})} \sum_{j=1}^{r} w_j^\intercal\tilde{\mathbf{X}}\tilde{\mathbf{X}}^\intercal w_j \text{ subject to } \mathbf{W}^\intercal\mathbf{W} = \mathbf{I}_r \tag{6.4}$$

Now, to begin simply and drive our intuition, we shall have a look to the solution when we are looking for only one principal component vector. In this case, the optimization problem reads :

$$\max_{w_1 \in \mathbb{R}^d} w_1^\intercal\tilde{\mathbf{X}}\tilde{\mathbf{X}}^\intercal w_1 \text{ subject to } w_1^\intercal w_1 = 1$$

To solve this constrained optimization problem, we introduce the associated Lagrangian :

$$\mathcal{L}(w_1, \lambda_1) = w_1^\intercal\tilde{\mathbf{X}}\tilde{\mathbf{X}}^\intercal w_1 + \lambda_1(1 - w_1^\intercal w_1)$$

We now compute the gradients of the Lagrangian with respect to w_1 and λ_1 :

$$\begin{aligned} \frac{\partial\mathcal{L}}{d\lambda_1} &= 1 - w_1^\intercal w_1 \\ \frac{\partial\mathcal{L}}{dw_1} &= 2\tilde{\mathbf{X}}\tilde{\mathbf{X}}^\intercal w_1 - 2\lambda_1 w_1 \end{aligned}$$

Looking for the critical points, i.e. where the gradient vanishes, we get :

$$\frac{\partial \mathcal{L}}{d\lambda_1} = 0 \Rightarrow w_1^\intercal w_1 = 1$$
$$\frac{\partial \mathcal{L}}{dw_1} = 0 \Rightarrow \tilde{\mathbf{X}}\tilde{\mathbf{X}}^\intercal w_1 = \lambda_1 w_1$$

We therefore conclude that w_1 is an eigen vector of the sample covariance matrix $\tilde{\mathbf{X}}\tilde{\mathbf{X}}^\intercal$ with corresponding eigen-value λ_1. Now, the question is which eiven vector should we consider ? Well, to answer that question, we just need to look at what it means that w_1 is an eigenvector for our term we wish to maximize (6.4):

$$w_1^\intercal \tilde{\mathbf{X}}\tilde{\mathbf{X}}^\intercal w_1 = w_1^\intercal(\lambda_1 w_1) = \lambda_1 w_1^\intercal w_1 = \lambda_1$$

which is maximized for the largest eigen-value of the sample covariance matrix. So, to conclude this first part :

> *The first principal component vector is a normalized eigenvector associated with the largest eigen value of the sample covariance matrix* $\tilde{\mathbf{X}}\tilde{\mathbf{X}}^\intercal$

Now let us have a look to the problem of finding two components. The problem to be solved is :

$$\max_{w_1,w_2\in\mathbb{R}^d} w_1^\intercal \tilde{\mathbf{X}}\tilde{\mathbf{X}}^\intercal w_1 + w_2^\intercal \tilde{\mathbf{X}}\tilde{\mathbf{X}}^\intercal w_2 \text{ subject to } w_i^\intercal w_j = \delta_{i,j}$$

One first thing we can note is that the sum of the variances of the projections can be written a slightly different way taking into account the fact that w_1 and w_2 are orthogonal, i.e. $w_1^\intercal w_2 = 0$. The residuals of the orthogonal projection of the data $\tilde{\mathbf{X}}$ over the vector w_1 is $(\mathbf{I}_d - w_1 w_1^\intercal)\tilde{\mathbf{X}}$ and :

$$\begin{aligned} w_2^\intercal((\mathbf{I}_d - w_1 w_1^\intercal)\tilde{\mathbf{X}})((\mathbf{I}_d - w_1 w_1^\intercal)\tilde{\mathbf{X}})^\intercal w_2 &= w_2^\intercal(\mathbf{I}_d - w_1 w_1^\intercal)\tilde{\mathbf{X}}\tilde{\mathbf{X}}^\intercal(\mathbf{I}_d - w_1 w_1^\intercal)w_2 \\ (\mathbf{I}_d - w_1 w_1^\intercal)w_2 &= w_2 - (w_1 w_1^\intercal)w_2 \\ &= w_2 - w_1(w_1^\intercal w_2) \\ &= w_2 \\ \Rightarrow w_2^\intercal((\mathbf{I}_d - w_1 w_1^\intercal)\tilde{\mathbf{X}})((\mathbf{I}_d - w_1 w_1^\intercal)\tilde{\mathbf{X}})^\intercal w_2 &= w_2^\intercal \tilde{\mathbf{X}}\tilde{\mathbf{X}}^\intercal w_2 \end{aligned}$$

Therefore the variance of the data projected on w_2 equals the variance of the residuals, after projection on w_1, projected on w_2. We can then proceed iteratively and our greedy algorithm would lead to select the normalized eigenvectors associated with the largest eigenvalues of the sample covariance matrix (we finish justifying the correctness of .

> *If the matrix* $\tilde{\mathbf{X}}\tilde{\mathbf{X}}^\intercal$ *has* r ***distinct eigenvalues****, our solution* $\{w_1, \cdots, w_k, \cdots, w_r\}$ *are the* r ***normalized eigenvectors*** *associated with the* ***largest eigenvalues*** *of* $\tilde{\mathbf{X}}\tilde{\mathbf{X}}^\intercal$.

The solution to the PCA is not unique. Indeed, if w_1 is an eigenvector of $\mathbf{M}$, then $-w_1$ is also an eigenvector of $\mathbf{M}$. In terms of the principal components, this means that they are at least defined up to a sign. For example, on the illustration 6.2, we draw w_1 but we may have considered $-w_1$ as well. Also, if the matrix $\tilde{\mathbf{X}}\tilde{\mathbf{X}}^\intercal$ has eigenvalues with a multiplicity larger than 1, since the eigenvectors of an eigenvalue λ of any positive-definite symmetric matrix $\mathbf{M}$ of multiplicity k_λ engenders a subspace of dimensionality k_λ, any basis of this subspace are elements of the solution to the PCA problem.

Is the greedy algorithm optimal ?

There is a tricky step in the previous section because we still need to justify that the greedy algorithm above actually leads to an optimal solution. In the context of our optimization problem, we indeed used above a greedy algorithm where we begin by finding the first vector on which to project the data and then proceed by looking at the second vector on which to project the residues and so on. In general, there is no reason that a greedy algorithm finds the optimal solution to an optimization problem but in our case, this is indeed the case. Indeed, we shall demonstrate that whichever set of projection vectors, eigenvectors associated to the largest eigenvalues of the sample covariance matrix are solutions to the maximization problem given by equation (6.4).

Theorem 6.1. *For any symmetric positive semi-definite matrix* $\mathbf{M} \in \mathcal{M}_{d,d}(\mathbb{R})$*, denote* $\{\lambda\}1d$ *its eigenvalues with* $\lambda_1 \geq \lambda_2 \cdots \geq \lambda_d \geq 0$*. For any set of* $r \in [|1, d|]$ *orthogonal unit vectors,* $\{\mathbf{v}_1, \cdots, \mathbf{v}_j, \cdots, \mathbf{v}_r\}$*, we have :*

$$\sum_{j=1}^{r} v_j^T \mathbf{M} v_j \leq \sum_{j=1}^{r} \lambda_j$$

And this upper bound is reached by eigenvectors associated with the largest eigenvalues of $\mathbf{M}$

Proof. Suppose we have r orthogonal unit vectors $\{\mathbf{v}_1, \cdots, \mathbf{v}_j, \cdots, \mathbf{v}_r\}$ on which we project the data. We want to compute the maximal value of $\sum_{j=1}^{r} v_j^T \mathbf{M} v_j$.

Given the matrix $\mathbf{M}$ is real symmetric, there exists a basis of $\mathbb{R}^d$ of eigenvectors. Let us denote this basis $\{e_1, \cdots, e_i, \cdots, e_d\}$ and the associated eigenvalues $\{\lambda_1, \cdots, \lambda_i, \cdots, \lambda_d\}$. Denote $\beta_{i,j} = e_i^T v_j$ the coordinates of v_j in our basis :

$$\forall j \in [|1, r|], v_j = \sum_{i=1}^{d} \beta_{i,j} e_i$$

From which it follows :

$$\begin{aligned}
\sum_{j=1}^{r} v_j^T \mathbf{M} v_j &= \sum_{j=1}^{r} (\sum_{i=1}^{d} \beta_{i,j} e_i)^T \mathbf{M} (\sum_{i=1}^{d} \beta_{i,j} e_i) \\
&= \sum_{j=1}^{r} (\sum_{i=1}^{d} \beta_{i,j} e_i)^T (\sum_{i=1}^{d} \beta_{i,j} \mathbf{M} e_i) \\
&= \sum_{j=1}^{r} (\sum_{i=1}^{d} \beta_{i,j} e_i)^T (\sum_{i=1}^{d} \beta_{i,j} \lambda_i e_i) \\
&= \sum_{j=1}^{r} (\sum_{i=1}^{d} \beta_{i,j} e_i^T)(\sum_{i=1}^{d} \beta_{i,j} \lambda_i e_i) \\
&= \sum_{j=1}^{r} \sum_{i=1}^{d} \lambda_i \beta_{i,j}^2 = \sum_{i=1}^{d} \lambda_i \sum_{j=1}^{r} \beta_{i,j}^2
\end{aligned} \tag{6.5}$$

The next question is about upper bounding $\sum_j \beta_{i,j}^2$. To do so, we extend the set of orthogonal unit vectors $\{\mathbf{v}_1, \cdots, \mathbf{v}_j, \cdots, \mathbf{v}_r\}$ with $d - r$ vectors u_j to form a basis $\{v_1, \cdots, v_r, u_1, \cdots, u_{d-r}\}$ of $\mathbb{R}^d$. We can express the eigenvectors e_i in this basis as :

$$\forall i \in [|1, d|], e_i = \sum_{j=1}^{r} \beta_{i,j} v_j + \sum_{j=1}^{d-r} e_i^T u_j u_j$$

The eigenvectors are unit vectors, from which it follows :

$$\forall i, 1 = |e_i|_2^{\,2} = \sum_{j=1}^{r} \beta_{i,j}^2 + \sum_{j=1}^{d-r} (e_i^T u_j)^2$$

This leads to the inequality : $\forall i, \sum_{j=1}^{r} \beta_{i,j} \leq 1$, which can be injected in equation (6.5) :

$$\sum_{j=1}^{r} v_j^T \mathbf{M} v_j = \sum_{i=1}^{d} \lambda_i \sum_{j=1}^{r} \beta_{i,j}^2 \leq \sum_{i=1}^{d} \lambda_i$$

If the projection vectors are eigenvectors $\{\mathbf{w}_1, \cdots, \mathbf{w}_j, \cdots, \mathbf{w}_r\}$ with associated eigenvalues $\{\lambda_j, \cdots, \lambda_1, \cdots, \lambda_r\}$, we have :

$$\sum_{j=1}^{r} \mathbf{w}_j^T \mathbf{M} \mathbf{w}_j = \sum_{j=1}^{r} \lambda_j \mathbf{w}_j^T \mathbf{w} = \sum_{j=1}^{r} \lambda_j$$

Which concludes the proof.

□

It remains to apply the previous algorithm to the symmetric positive semi-definite matrix $\tilde{\mathbf{X}}\tilde{\mathbf{X}}^T$ to conclude that eigenvectors associated with the largest eigenvalues are indeed a solution to our optimization problem.

What is the fraction of sample variance that we keep ?

If we keep only r eigenvalues, we may wonder how much of the sample variance we keep. The sample variance of the data points is the sum of the diagonal elements of the sample covariance matrix (i.e. its trace). Also, given the sample covariance matrix is real symetric, its trace equals the sum of its eigenvalues. Indeed, for any matrix $\mathbf{M}$ and orthogonal matrix $\mathbf{P}$, we have :

$$\operatorname{Tr} \mathbf{P}^{-1}\mathbf{M}\mathbf{P} = \sum_{i,k,l} \mathbf{P}_{i,l}^{-1} \mathbf{M}_{l,k} \mathbf{P}_{k,i} = \sum_{k,l} (\mathbf{P}\mathbf{P}^{-1})_{k,l} \mathbf{M}_{k,l} = \sum_{k,l} \delta_{k,l} \mathbf{M}_{k,l} = \sum_{k} \mathbf{M}_{k,k} = \operatorname{Tr} \mathbf{M}$$

and therefore, denoting σ the sample variance of the data :

$$\sigma = \operatorname{Tr} \tilde{\mathbf{X}}\tilde{\mathbf{X}}^{\intercal} = \sum_{i=0}^{N-1} \lambda_i$$

Finally, considering only r eigenvalues, the sample variance of the projected data equals the fraction $\frac{\sum_{i=0}^{r-1} \lambda_i}{\sum_{i=0}^{N-1} \lambda_i}$ of the original sample variance.

Using the Singular Value Decomposition to compute the PCA

In order to compute the eigenvectors of $\tilde{\mathbf{X}}\tilde{\mathbf{X}}^{\intercal}$, i.e. the vectors $\{w_1, \cdots, w_j, \cdots, w_r\}$, there is no need to compute the matrix product and make the eigen value decomposition of the product but one can rather make use of some decomposition of the matrix $\tilde{\mathbf{X}}$ such as the Singular Value Decomposition. As we shall see, the Singular Value Decomposition also allows to directly compute the principal components of our data, without having to project explicitely each sample on the principal component vectors. Indeed, we know that there exist an orthogonal matrix $\mathbf{U} \in \mathcal{M}_{d,d}(\mathbb{R})$ ($\mathbf{U}\mathbf{U}^{\intercal} = \mathbf{U}^{\intercal}\mathbf{U} = \mathbf{I}_d$), a diagonal matrix $\mathbf{D} \in \mathcal{M}_{d,N}(\mathbb{R})$ with non-negative real numbers on the diagonal and an orthogonal matrix $\mathbf{V} \in \mathcal{M}_{N,N}(\mathbb{R})$ ($\mathbf{V}\mathbf{V}^{\intercal} = \mathbf{V}^{\intercal}\mathbf{V} = \mathbf{I}_N$) such that :

$$\begin{aligned} \tilde{\mathbf{X}} &= \mathbf{U}\mathbf{D}\mathbf{V}^{\intercal} \\ \Rightarrow \tilde{\mathbf{X}}\tilde{\mathbf{X}}^{\intercal} &= \mathbf{U}\mathbf{D}\mathbf{V}^{\intercal}\mathbf{V}\mathbf{D}^{\intercal}\mathbf{U}^{\intercal} = \mathbf{U}\mathbf{D}\mathbf{D}^{\intercal}\mathbf{U}^{\intercal} = \mathbf{U}\mathbf{D}\mathbf{D}^{\intercal}\mathbf{U}^{-1} \end{aligned}$$

We recognize the diagonalization of $\tilde{\mathbf{X}}\tilde{\mathbf{X}}^{\intercal}$, the eigen vectors of $\tilde{\mathbf{X}}\tilde{\mathbf{X}}^{\intercal}$ being the column vectors of $\mathbf{U}$. In the singular value decomposition, we suppose that the diagonal elements of $\mathbf{D}$ are ordered by decreasing magnitude and the implementation usually behave that way. Therefore, the vectors $\{w_1, \cdots, w_j, \cdots, w_r\}$ we are looking for are the first r column vectors of $\mathbf{U}$: $\mathbf{W} = \begin{pmatrix} \mathbf{U}_1 | & \mathbf{U}_2 | & \cdots | & \mathbf{U}_r \end{pmatrix}$. The principal components are the projections of the data over the axes

$\{w_1, \cdots, w_j, \cdots, w_r\} = \{\mathbf{u}_1, \cdots, \mathbf{u}_j, \cdots, \mathbf{u}_r\}$, i.e. $\mathbf{W}^\intercal\mathbf{X}$. Let us compute the projection of our samples over the principal component vectors :

$$\mathbf{U}^\intercal\tilde{\mathbf{X}} = \mathbf{U}^\intercal\mathbf{U}\mathbf{D}\mathbf{V}^\intercal = \mathbf{D}\mathbf{V}^\intercal$$

with $\mathbf{D}\mathbf{V}^\intercal \in \mathcal{M}_{d,N}(\mathbb{R})$. The principal components are therefore the first r rows of $\mathbf{D}\mathbf{V}^\intercal$.

Algorithm 9 gives all the steps for performing a SVD based PCA. An example of application of this algorithm on artificial data is shown on Fig. 6.3a as well as applying the PCA on the 5000 samples from the handwritten MNIST dataset on Fig. 6.3b. From the example over the MNIST dataset, one can observe that the linear projection revels some isolated clusters (for the digits 0 and 1) but others remain interleaved. Do not be misleaded, PCA is an unsupervised algorithm as it does not take into account the labels for finding the projections. The labels are added to the figures after the PCA is performed and it turns out that it reveals that some classes are isolated. Using only two principal components, the PCA captures 4.48% of the variance, computed as $\frac{\lambda_0+\lambda_1}{\sum_i \lambda_i}$ where, λ_i are the eigenvalues of the sample covariance matrix, ordered by decreasing magnitude.

> *If the centered data are stacked as the columns of $\tilde{\mathbf{X}}$. From the SVD of $\tilde{\mathbf{X}} = \mathbf{U}\mathbf{D}\mathbf{V}^\intercal$ (the eigenvalues being ordered by decreasing magnitude on the diagonal of $\mathbf{D}$), one finds the r first principal components as the r first rows of $\mathbf{D}\mathbf{V}^\intercal$ and the projection vectors as the r first columns of $\mathbf{U}$.*

Algorithm 9 Algorithm for computing r principal components.

Input: $\{x_0, \cdots, x_i, \cdots, x_{N-1}\}$ // The original data in $\mathbb{R}^d$
Output: $\{y_0, \cdots, y_i, \cdots, y_{N-1}\}$ // The projected data in $\mathbb{R}^r$
1: Compute the mean of the data $\tilde{x} = \frac{1}{N}\sum_{i=0}^{N-1} x_i$
2: Center the data $\forall i, \tilde{x}_i = x_i - \tilde{x}$
3: Stack the data in the columns of $\tilde{\mathbf{X}} \in \mathcal{M}_{d,N}(\mathbb{R})$
4: Compute the SVD of $\tilde{\mathbf{X}} = \mathbf{U}\mathbf{D}\mathbf{V}^\intercal$
5: The principal components are the first r rows of $\mathbf{D}\mathbf{V}^\intercal$: $\begin{pmatrix} y_0 | & \cdots | & y_{N-1} \end{pmatrix} = (\mathbf{D}\mathbf{V}^\intercal)[:r, :]$
6: The projections vectors are the first r column vectors of $\mathbf{U}$.

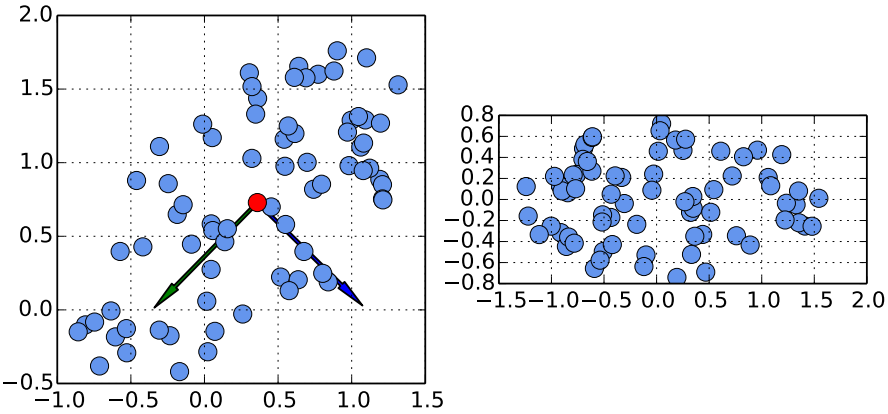
Finding an affine subspace maximizing the variance of the projections

There is a common alternative for deriving the principal components that leads to an equivalent algorithm than the one presented in the previous section. This alternative consists in finding axis on which to project the data so that the variance of the projections is maximized. The optimization problem is therefore introduced in a slightly different way than when we optimized the reconstruction error. To simplifiy the presentation, we here suppose that the data are **centered**. The optimization problem can then be written as :

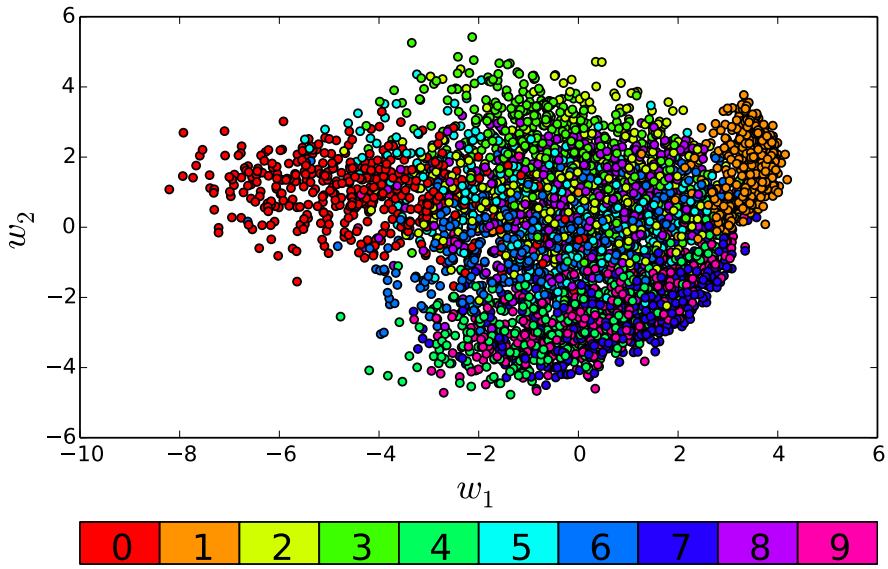
$$\max_{\{w_1,\cdots,w_j,\cdots,w_r\}\in\mathbb{R}^d} \sum_{j=1}^{r} \frac{1}{N-1} \sum_{i=0}^{N-1} \left|w_j^\intercal.\tilde{x}_i\right|_2^2 \text{ subject to } w_i^\intercal.w_j = \delta_{i,j}$$

We can rewrite a little the inner term :

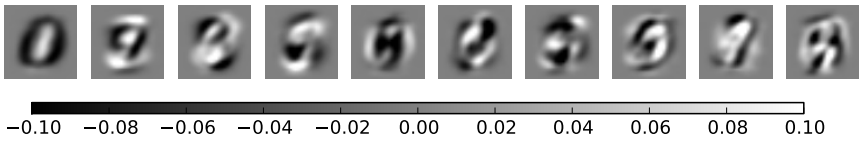
$$\begin{aligned}
\sum_{j=1}^{r} \frac{1}{N-1} \sum_{i=0}^{N-1} \left|w_j^\intercal.\tilde{x}_i\right|_2^2 &= \sum_j \frac{1}{N-1} \sum_i w_j^\intercal \tilde{x}_i \tilde{x}_i^\intercal w_j = \sum_j w_j^\intercal (\frac{1}{N} \sum_i \tilde{x}_i \tilde{x}_i^\intercal) w_j \\
&= \sum_j \frac{1}{N-1} w_j^\intercal \tilde{\mathbf{X}}\tilde{\mathbf{X}}^\intercal w_j \\
&= \frac{1}{N-1} \sum_{i=0}^{N-1} \tilde{x}_i^\intercal \mathbf{W}\mathbf{W}^\intercal \tilde{x}_i
\end{aligned}$$



(a) The PCA of a set of datapoints in $\mathbb{R}^2$. There are $N = 72$ datapoints in $d = 2$ dimensions. On the left, the original datapoints are shown. The mean of the data w_0 is shown as the red dot. The data are centered and stacked in the rows of $\tilde{\mathbf{X}}$. From the SVD decomposition of $\tilde{\mathbf{X}} = \mathbf{U}\mathbf{D}\mathbf{V}^\mathrm{T}$, the two columns of $\mathbf{V}$ are extracted and shown as the green and blue arrows. The projection of the datapoints on the two principal vectors $\mathbf{DU}$ is shown on the figure on the right.



(b) The PCA applied to 1000 samples from the MNIST handwritten digits dataset. Each colored point represents one 28×28 image, the color indicating the associated label. Do not be misleaded, the labels are not used for the PCA which is an unsupervised technique. The labels are used after applying the PCA, just to get an idea of how the digits are clustered. The two first components computed from 5000 samples capture actually only 4.48% of the variance.



(c) The 10 first principal vectors when applying PCA to 5000 sampes of the MNIST handwritten digits dataset. All the 28×28 vectors are normalized and have been saturated in the range [-0.10, 0.10]. We better appreciate for exemple why the 0 and 1 have, respectively, a negative and positive first component.

We first recognize the sample covariance matrix of the data :

$$\mathbf{\Sigma} = \frac{1}{N-1}\sum_{i=0}^{N-1} \tilde{x}_i \tilde{x}_i^\intercal = \frac{1}{N-1}\sum_{i=0}^{N-1} (x_i - \tilde{x})(x_i - \tilde{x})^\intercal$$

and also the same optimization problem than (6.4) which we found when defining the PCA as minimizing the reconstruction error. We can then conclude that the optimal projection vectors are the eigen vectors of the sample covariance matrix $\Sigma = \frac{1}{N}\tilde{\mathbf{X}}\tilde{\mathbf{X}}^\intercal$ associated with the largest r eigenvalues.

6.2.3 Relationship between covariance, Gram and euclidean distance matrices

Sample covariance matrix. The *sample covariance matrix* of a set of vectors $\{x_0, \cdots, x_i, \cdots, x_{N-1}\} \in \mathbb{R}^d$ is the $d \times d$ matrix $\mathbf{\Sigma}$ defined as :

$$\mathbf{\Sigma} = \frac{1}{N-1}\sum_{i=0}^{N-1} (x_i - \overline{x})(x_i - \overline{x})^\intercal = \frac{1}{N-1}\mathbf{X}\mathbf{X}^\intercal$$
$$\overline{x} = \frac{1}{N}\sum_{i=0}^{N-1} x_i$$

where $\mathbf{X} = [x_0 - \overline{x} | x_1 - \overline{x} | \cdots | x_{N-1} - \overline{x}]$, i.e. the vectors $x_i - \overline{x}$ are the column vectors of $\mathbf{X}$. The sample covariance matrix is symmetric : $\mathbf{\Sigma}^\intercal = \mathbf{\Sigma}$. The eigenvalues of any sample covariance matrix are non-negative. For any eigenvalue-eigenvector pair $\lambda, \mathbf{v}$ of $\mathbf{\Sigma}$, we have :

$$\lambda\mathbf{v} = \mathbf{\Sigma}\mathbf{v} = \frac{1}{N-1}\mathbf{X}\mathbf{X}^\intercal\mathbf{v}$$
$$\Rightarrow \lambda|\mathbf{v}|^2 = \lambda\mathbf{v}^\intercal\mathbf{v} = \frac{1}{N-1}\mathbf{v}^\intercal\mathbf{X}\mathbf{X}^\intercal\mathbf{v} = \frac{1}{N-1}(\mathbf{X}^\intercal\mathbf{v})^\intercal(\mathbf{X}^\intercal\mathbf{v}) = \frac{1}{N-1}\left|\mathbf{X}^\intercal\mathbf{v}\right|^2$$

from which it follows $\lambda \geq 0$.

Gram matrix. The *Gram matrix* of a set of vectors $\{x_0, \cdots, x_i, \cdots, x_{N-1}\} \in \mathbb{R}^d$ is the $N \times N$ matrix $\mathbf{G}$ containing the scalar products of the vectors x_i, namely $G_{ij} = x_i^\intercal . x_j$:

$$\mathbf{G} = \begin{bmatrix} x_0^\intercal . x_0 & x_0^\intercal . x_1 & \cdots & x_0^\intercal . x_{N-1} \\ \vdots & \vdots & \vdots & \vdots \\ x_{N-1}^\intercal . x_0 & x_{N-1}^\intercal . x_1 & \cdots & x_{N-1}^\intercal . x_{N-1} \end{bmatrix} = \mathbf{X}^\intercal\mathbf{X}$$

where $\mathbf{X} = [x_0 | x_1 | \cdots | x_{N-1}]$, i.e. the vectors x_i are the column vectors of $\mathbf{X}$. The gram matrix is symmetric : $\mathbf{G}^\intercal = \mathbf{G}$.

Euclidean distance matrix. The *Euclidean distance matrix* of a set of vectors $\{x_0, \cdots, x_i, \cdots, x_{N-1}\} \in \mathbb{R}^d$ is the $N \times N$ matrix $\mathbf{D}$ whose elements are the squared euclidean distances between the vectors x_i, namely $D_{ij} = |x_i - x_j|^2$:

$$\mathbf{D} = \begin{bmatrix} 0 & |x_0 - x_1|^2 & \cdots & |x_0 - x_{N-1}|^2 \\ \vdots & & \ddots & \vdots \\ |x_{N-1} - x_0|^2 & |x_{N-1} - x_1|^2 & \cdots & 0 \end{bmatrix}$$

The Euclidean distance matrix $\mathbf{D}$ is symmetric and has zeros in the main diagonal : $\mathbf{D}^\intercal = \mathbf{D}$, $\forall i, D_{ii} = 0$.

Relationship between euclidean distance and gram matrices Let us now detail how the covariance, gram and euclidean distance matrices are related. The euclidean distances can be expressed from scalar products only :

$$\forall i,j, |x_i - x_j|^2 = (x_i - x_j)^\intercal.(x_i - x_j) = x_i^\intercal.x_i + x_j^\intercal.x_j - 2x_i^\intercal x_j$$

Therefore, the euclidean distance matrix and gram matrix are related by :

$$\Rightarrow \forall i,j, \mathbf{D}_{ij} = \mathbf{G}_{ii} + \mathbf{G}_{jj} - 2\mathbf{G}_{ij}$$

Actually, the euclidean distance matrix can be built from the gram matrix but the converse is not true : the same euclidean distance matrix can be built from different configuration of the input vectors and different gram matrices. For example :

$$\mathbf{X} = \begin{bmatrix} 1 & \frac{1}{2} \\ 0 & \frac{\sqrt{3}}{2} \end{bmatrix}, \mathbf{D} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, \mathbf{G} = \begin{bmatrix} 1 & \frac{1}{2} \\ \frac{1}{2} & 1 \end{bmatrix}$$

$$\mathbf{X} = \begin{bmatrix} \sqrt{2} & \frac{3}{2\sqrt{2}} \\ 0 & \sqrt{\frac{7}{8}} \end{bmatrix}, \mathbf{D} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, \mathbf{G} = \begin{bmatrix} 2 & \frac{3}{2} \\ \frac{3}{2} & 2 \end{bmatrix}$$

More generally, the distance matrix is invariant to translation while the gram matrix is not. Indeed :

$$\begin{aligned} \forall \mathbf{x},\mathbf{y},\mathbf{c}, |(\mathbf{x}+\mathbf{c}) - (\mathbf{y}+\mathbf{c})| &= |\mathbf{x}-\mathbf{y}| \\ (\mathbf{x}+\mathbf{c})^\intercal.(\mathbf{y}+\mathbf{c}) &= \mathbf{x}^\intercal.\mathbf{y} + \mathbf{c}^\intercal.\mathbf{y} + \mathbf{x}^\intercal.\mathbf{c} + \mathbf{c}^\intercal.\mathbf{c} \end{aligned}$$

It is actually sufficient to define an origin for a set of vectors so that their Gram matrix can be deduced from the distance matrix. We can indeed recover the gram matrix of the vectors $x_i - \frac{1}{N}\sum_j x_j$ from the distance matrix and one can show that it is given by :

$$\mathbf{G} = -\frac{1}{2}\mathbf{HDH}$$

where $\mathbf{H}$ is a so-called centering matrix defined by :

$$\mathbf{H} = \mathbf{I} - \frac{1}{N}\mathbf{e}\mathbf{e}^\intercal$$

with $\mathbf{e}$ a vector full of 1, i.e. $\mathbf{H}_{i,j} = \delta_{i,j} - \frac{1}{N}$. The above transformation leading from $\mathbf{D}$ to $\mathbf{G}$ is called *double centering*.

Relationship between the covariance and gram matrices In order to stress an interesting relationship between the covariance and gram matrices that will be used in the next section for extending PCA to kernel PCA, we need to make a bit of linear algebra.

Lemma 6.1. $\forall \mathbf{A} \in \mathbb{R}^{n\times m}, \ker(\mathbf{A}) = \ker(\mathbf{A}^\intercal\mathbf{A})$

Proof. Let us consider $\mathbf{A} \in \mathbb{R}^{n\times m}$. It is clear that

$$\forall \mathbf{x} \in \mathbb{R}^m, \mathbf{A}\mathbf{x} = 0 \Rightarrow \mathbf{A}^\intercal\mathbf{A}\mathbf{x} = 0$$

Therefore $\ker(\mathbf{A}) \subseteq \ker(\mathbf{A}^\intercal\mathbf{A})$.

Now take $\mathbf{x} \in \mathbb{R}^m$, such that $\mathbf{x} \in \ker(\mathbf{A}^\intercal\mathbf{A})$. Then :

$$\begin{aligned} \mathbf{A}^\intercal\mathbf{A}\mathbf{x} = 0 &\Rightarrow \mathbf{x}^\intercal\mathbf{A}^\intercal\mathbf{A}\mathbf{x} = 0 \\ &\Leftrightarrow (\mathbf{A}\mathbf{x})^\intercal\mathbf{A}\mathbf{x} = 0 \\ &\Leftrightarrow |\mathbf{A}\mathbf{x}|_2^{\,2} = 0 \\ &\Leftrightarrow \mathbf{A}\mathbf{x} = 0 \end{aligned}$$

Therefore $\ker(\mathbf{A}^\intercal\mathbf{A}) \subseteq \ker(\mathbf{A})$ which ends the proof. □

We now remind the rank-nullity theorem, in the context of matrices but which could have been stated in a more general way with linear applications.

Theorem 6.2 (Rank-nullity)**.** $\forall \mathbf{A} \in \mathbb{R}^{n\times m}, \text{rk}(\mathbf{A}) + \dim(\ker(\mathbf{A})) = m.$

We can demonstrate a theorem linking the rank of the covariance and gram matrices.

Theorem 6.3. $\forall \mathbf{A} \in \mathbb{R}^{n\times m}, \text{rk}(\mathbf{A}^\intercal\mathbf{A}) = \text{rk}(\mathbf{A}\mathbf{A}^\intercal) \leq \min(n, m)$

Proof. By applying the lemma 6.1 and the rank nullity theorem :

$$\begin{aligned}\forall \mathbf{A} \in \mathbb{R}^{n\times m}, &\text{rk}(\mathbf{A}) + \dim(\ker(\mathbf{A})) = m = \text{rk}\left(\mathbf{A}^\intercal\mathbf{A}\right) + \dim\left(\ker\left(\mathbf{A}^\intercal\mathbf{A}\right)\right)\\ \Rightarrow &\text{rk}(\mathbf{A}) = \text{rk}\left(\mathbf{A}^\intercal\mathbf{A}\right) \text{ and } \text{rk}(\mathbf{A}) \leq m\end{aligned}$$

Applying the lemma 6.1 and the rank nullity theorem to the matrix $\mathbf{A}^\intercal$:

$$\begin{aligned}\forall \mathbf{A} \in \mathbb{R}^{n\times m}, &\text{rk}\left(\mathbf{A}^\intercal\right) + \dim\left(\ker\left(\mathbf{A}^\intercal\right)\right) = n = \text{rk}\left(\mathbf{A}\mathbf{A}^\intercal\right) + \dim\left(\ker\left(\mathbf{A}\mathbf{A}^\intercal\right)\right)\\ \Rightarrow &\text{rk}\left(\mathbf{A}^\intercal\right) = \text{rk}\left(\mathbf{A}\mathbf{A}^\intercal\right) \text{ and } \text{rk}(\mathbf{A}) \leq n\end{aligned}$$

We also know that the column rank and row ranks are equal and therefore $\text{rk}(\mathbf{A}) = \text{rk}(\mathbf{A}^\intercal)$ which ends the proof. □

Given the sample covariance matrix $\mathbf{\Sigma} = \frac{1}{N-1}\mathbf{X}\mathbf{X}^\intercal$ has the same rank than $(N-1)\Sigma = \mathbf{X}\mathbf{X}^\intercal$ and given the gram matrix $\mathbf{G} = \mathbf{X}^\intercal\mathbf{X}$, applying the previous theorem leads to :

$$\text{rk}(\mathbf{\Sigma}) = \text{rk}\left(\mathbf{X}\mathbf{X}^\intercal\right) = \text{rk}\left(\mathbf{X}^\intercal\mathbf{X}\right) = \text{rk}(\mathbf{G}) \leq \min(n, m)$$

Given that the covariance and gram matrices have the same rank, they have the same number of nonzero eigenvalues[8]. There is actually an even stronger property which is :

Lemma 6.2 (Eigenvalues of the covariance and gram matrices)**.** *The nonzero eigenvalues of the scaled covariance matrix* $(N-1)\mathbf{\Sigma} = \mathbf{X}\mathbf{X}^\intercal$ *and gram matrix* $\mathbf{G} = \mathbf{X}^\intercal\mathbf{X}$ *are the same :*

$$\{\lambda \in \mathbb{R}^*, \exists \mathbf{v} \neq \mathbf{0}, (N-1)\mathbf{\Sigma}\mathbf{v} = \lambda\mathbf{v}\} = \{\lambda \in \mathbb{R}^*, \exists \mathbf{v} \neq \mathbf{0}, \mathbf{G}\mathbf{v} = \lambda\mathbf{v}\}$$

Proof. Consider a nonzero eigenvalue $\lambda \in \mathbb{R}^*$ of $\mathbf{X}\mathbf{X}^\intercal$. There exists $\mathbf{v} \neq \mathbf{0}$, $\mathbf{X}\mathbf{X}^\intercal\mathbf{v} = \lambda\mathbf{v}$. Left-multiplying by $\mathbf{X}^\intercal$ gives $\mathbf{X}^\intercal\mathbf{X}\mathbf{X}^\intercal\mathbf{v} = \lambda\mathbf{X}^\intercal\mathbf{v}$. Denoting $\mathbf{w} = \mathbf{X}^\intercal\mathbf{v}$, we have $\mathbf{X}^\intercal\mathbf{X}\mathbf{w} = \lambda\mathbf{w}$. If $\mathbf{w} = \mathbf{X}^\intercal\mathbf{v} = 0$, we have $\lambda\mathbf{v} = \mathbf{X}\mathbf{X}^\intercal\mathbf{v} = \mathbf{X}\mathbf{w} = 0$ and therefore $\mathbf{v} = 0$ as $\lambda \neq 0$ which is in contradiction with our hypothesis. So, necessarily, $\mathbf{w} \neq 0$, and $(\lambda, \mathbf{X}^\intercal\mathbf{v})$ is an eigenvalue-eigenvector pair of $\mathbf{X}^\intercal\mathbf{X} = \mathbf{G}$. We have therefore demonstrated the inclusion

$$\{\lambda \in \mathbb{R}^*, \exists \mathbf{v} \neq \mathbf{0}, (N-1)\mathbf{\Sigma}\mathbf{v} = \lambda\mathbf{v}\} \subseteq \{\lambda \in \mathbb{R}^*, \exists \mathbf{v} \neq \mathbf{0}, \mathbf{G}\mathbf{v} = \lambda\mathbf{v}\}$$

Since the two sets have the same dimension, there are actually equal. □

During the demonstration, we also showed that if $(\lambda, \mathbf{v}) \in \mathbb{R} \times \mathbb{R}^d$ is an eigenvalue-eigenvector pair of $\mathbf{X}\mathbf{X}^\intercal$, then $(\lambda, \mathbf{X}^\intercal\mathbf{v}) \in \mathbb{R} \times \mathbb{R}^N$ is an eigenvalue-eigenvector pair of $\mathbf{X}^\intercal\mathbf{X}$. Conversely if $(\lambda, \mathbf{w}) \in \mathbb{R} \times \mathbb{R}^N$ is an eigenvalue-eigenvector pair of $\mathbf{X}^\intercal\mathbf{X}$, then $(\lambda, \mathbf{X}\mathbf{w}) \in \mathbb{R} \times \mathbb{R}^d$ is an eigenvalue-eigenvector pair of $\mathbf{X}\mathbf{X}^\intercal$.

6.2.4 Principal component analysis in large dimensional space ($N \ll d$)

Suppose we are given N input vectors $\{x_0, \cdots, x_i, \cdots, x_{N-1}\} \in \mathbb{R}^d$. Suppose also that the number of dimensions d is much larger than the number of input vectors N which we denote $d \gg N$. This is the case for example when we are working with a small dataset of images. As detailed in the previous section, PCA operates with the following steps :

1. center the input vectors : $\overline{x} = \frac{1}{N}\sum_{i=0}^{N-1} x_i$, $\tilde{x}_i = x_i - \overline{x}$

[8] any real symmetric matrix $\mathbf{A}$ can be diagonalized $\mathbf{A} = \mathbf{O}\mathbf{D}\mathbf{O}^\intercal$. The rank is the dimension of the space engendered by the columns or the rows of $\mathbf{A}$ which has the same dimension than the space engendered by the columns or the rows of $\mathbf{D}$ and therefore equals the number of nonzero eigenvalues of $\mathbf{A}$.

2. stack the centered vectors in the matrix $\tilde{\mathbf{X}} = [\tilde{x}_0|\tilde{x}_1|\cdots|\tilde{x}_{N-1}]$
3. compute the r normalized eigenvectors $\mathbf{v}_j$ associated with the r largest eigenvalues of the matrix $\tilde{\mathbf{X}}\tilde{\mathbf{X}}^{\mathsf{T}} \in \mathbb{R}^{d\times d}$
4. project your data on the r normalized eigenvectors $\mathbf{v}_j$

When the number of features d increases, the covariance matrix can become quite large and the computation of the eigenvectors can be cumbersome. In light of the lemma 6.2, it turns out that the eigenvectors of $\tilde{\mathbf{X}}\tilde{\mathbf{X}}^{\mathsf{T}}$ can be computed from the eigenvectors of $\tilde{\mathbf{X}}^{\mathsf{T}}\tilde{\mathbf{X}} \in \mathbb{R}^{N\times N}$. In case $N \ll d$, it is much less expensive to compute these eigenvectors. We can therefore state another equivalent way of computing PCA :

1. center the input vectors : $\overline{x} = \frac{1}{N}\sum_{i=0}^{N-1} x_i$, $\tilde{x}_i = x_i - \overline{x}$
2. stack the centered vectors in the matrix $\tilde{\mathbf{X}} = [\tilde{x}_0|\tilde{x}_1|\cdots|\tilde{x}_{N-1}]$
3. compute the r normalized eigenvectors $\mathbf{w}_j \in \mathbb{R}^N$ associated with the r largest eigenvalues λ_j of the matrix $\tilde{\mathbf{X}}^{\mathsf{T}}\tilde{\mathbf{X}} \in \mathbb{R}^{N\times N}$
4. project your data on the r normalized[9] eigenvectors $\frac{\tilde{\mathbf{X}}\mathbf{w}_j}{|\tilde{\mathbf{X}}\mathbf{w}_j|} = \frac{1}{\sqrt{\lambda_j}}\tilde{\mathbf{X}}\mathbf{w}_j$

We can reformulate a little bit this procedure by making use only of scalar products, and this will turn out to be very useful for deriving a non-linear extension of PCA :

1. center the input vectors : $\overline{x} = \frac{1}{N}\sum_{i=0}^{N-1} x_i$, $\tilde{x}_i = x_i - \overline{x}$
2. compute the r normalized eigenvectors $\mathbf{w}_j \in \mathbb{R}^N$ associated with the r largest eigenvalues λ_j of the Gram matrix $\mathbf{G} = \begin{bmatrix} \tilde{x}_0.\tilde{x}_0 & \cdots & \tilde{x}_0.\tilde{x}_{N-1} \\ \vdots & \vdots & \vdots \\ \tilde{x}_{N-1}.\tilde{x}_0 & \cdots & \tilde{x}_{N-1}.\tilde{x}_{N-1} \end{bmatrix}$
3. project your data on the r normalized eigenvectors $\frac{\tilde{\mathbf{X}}\mathbf{w}_j}{|\tilde{\mathbf{X}}\mathbf{w}_j|} = \frac{1}{\sqrt{\lambda_j}}\tilde{\mathbf{X}}\mathbf{w}_j : \forall \mathbf{v} \in \mathbb{R}^d, \frac{1}{\sqrt{\lambda_j}}\tilde{\mathbf{X}}\mathbf{w}_j.\mathbf{w} = \frac{1}{\sqrt{\lambda_j}}\mathbf{w}_j.\begin{bmatrix} \tilde{x}_0.\mathbf{w} \\ \vdots \\ \tilde{x}_{N-1}.\mathbf{w} \end{bmatrix}$

6.2.5 Kernel PCA (KPCA)

Given a set of N vectors $\{x_0, \cdots, x_i, \cdots, x_{N-1}\} \in \mathbb{R}^d$, suppose we have a, possibly non linear, mapping φ of our input space into a so called *feature space* F :

$$\begin{array}{rcl} \varphi : \mathbb{R}^d & \to & \Phi \\ x & \mapsto & X \end{array}$$

Typically, F can be a vector space of larger dimension (even infinite) than the input space, e.g. $F = \mathbb{R}^m, m \gg d$. Let us assume for now (but we come back soon on this point) that the transformed data are centered, i.e.

$$\frac{1}{N}\sum_{i=0}^{N-1} \varphi(x_i) = 0$$

Let us now perform a linear PCA on the transformed data $\{\varphi(x_0), \cdots, \varphi(x_i), \cdots, \varphi(x_{N-1})\}$. To do so, we will follow the procedure given in the previous section when working only with scalar products :

[9] $\left|\tilde{\mathbf{X}}\mathbf{w}_j\right|^2 = ({}^{\mathsf{T}}\tilde{\mathbf{X}}\mathbf{w}_j)\tilde{\mathbf{X}}\mathbf{w}_j = \mathbf{w}_j^{\mathsf{T}}\tilde{\mathbf{X}}^{\mathsf{T}}\tilde{\mathbf{X}}\mathbf{w}_j = \lambda_j\mathbf{w}_j^{\mathsf{T}}\mathbf{w}_j = \lambda_j$

1. center the input vectors $\{\varphi(x_0), \cdots, \varphi(x_i), \cdots, \varphi(x_{N-1})\}$: there is nothing to do since we consider, for now, that the mapped vectors are centered (at the end of the section, we come back to the case the mapped vectors are not centered)
2. compute the r normalized eigenvectors $\mathbf{w}_k \in \mathbb{R}^N$ associated with the r largest eigenvalues λ_k of the Gram matrix $\mathbf{G} \in \mathbb{R}^{N \times N}$:

$$\mathbf{G} = \begin{bmatrix} \varphi(x_0).\varphi(x_0) & \varphi(x_0).\varphi(x_1) & \cdots & \varphi(x_0).\varphi(x_{N-1}) \\ \varphi(x_1).\varphi(x_0) & \varphi(x_1).\varphi(x_1) & \cdots & \varphi(x_1).\varphi(x_{N-1}) \\ \vdots & \vdots & \vdots & \vdots \\ \varphi(x_{N-1}).\varphi(x_0) & \varphi(x_{N-1}).\varphi(x_1) & \cdots & \varphi(x_{N-1}).\varphi(x_{N-1}) \end{bmatrix}$$

3. project your data, say $\mathbf{y}$, on the r normalized eigenvectors $\frac{1}{\sqrt{\lambda_k}}\varphi(\mathbf{X})\mathbf{w}_k$

Let us have a look of what the last point, the projection of the vector to get the component, looks like :

$$\forall \mathbf{v} \in \mathbb{R}^d, (\frac{1}{\sqrt{\lambda_k}}\varphi(\mathbf{X})\mathbf{w}_k).\varphi(\mathbf{v}) = \frac{1}{\sqrt{\lambda_k}}\mathbf{w}_k.\begin{bmatrix} \varphi(x_0).\varphi(\mathbf{v}) \\ \varphi(x_1).\varphi(\mathbf{v}) \\ \vdots \\ \varphi(x_{N-1}).\varphi(\mathbf{v}) \end{bmatrix}$$

Therefore, the k-th principal component is computed solely from scalar products between the vectors mapped into the feature space Φ. Actually, the only thing we need to compute when performing the PCA in the feature space is scalar products between vectors in feature space since the Gram matrix, from which we extract the eigenvectors, are also computed only from scalar products in the feature space. We never need to compute explicitely the feature vectors $\varphi(v) \in \Phi$ and only need to evaluate scalar products between two elements of Φ. This algorithm is known as the Kernel PCA(Scholkopf et al., 1999).

The fact that all we need to compute is scalar products in the feature space implies that we can employ the so called *kernel trick* which implicitly defines the mapping function φ from the definition of a so-called *kernel* function k. Not every function k is a kernel as it does not always correspond to the scalar product in a feature space. However, there are some conditions, known as the *Mercer's theorem*, which determine when a function k is actually a kernel. This is explained in detail in the chapter 10. For our purpose, we just introduce briefly some known kernels :

- the polynomial kernel $\mathrm{k}_d(x, x') = (x.x' + c)^d$
- the gaussian, or RBF, kernel $\mathrm{k}_{\mathrm{rbf}}(x, x') = \exp\left(-\frac{|x - x'|^2}{2\sigma^2}\right)$

It can be shown that the gaussian kernel actually projects the data into an infinite dimension space. The reader is referred to the chapter 10 for more details on kernels.

The last point that is not yet solved is : what about feature vectors that are actually not centered in the feature space ? One can actually work with uncentered feature vectors if we change the kernel :

$$\begin{aligned} \forall i, j, &\left(\varphi(x_i) - \frac{1}{N}\sum_{p=0}^{N-1}\varphi(x_p)\right).\left(\varphi(x_j) - \frac{1}{N}\sum_{p=0}^{N-1}\varphi(x_p)\right) \\ &= \varphi(x_i).\varphi(x_j) - \frac{1}{N}\sum_{p=0}^{N-1}\varphi(x_i).\varphi(x_p) - \frac{1}{N}\sum_{p=0}^{N-1}\varphi(x_j).\varphi(x_p) + \frac{1}{N^2}\sum_{p=0}^{N-1}\sum_{t=0}^{N-1}\varphi(x_p).\varphi(x_t) \\ &= \mathrm{k}(x_i, x_j) - \frac{1}{N}\sum_{k=0}^{N-1}(\mathrm{k}(x_i, x_k) + \mathrm{k}(x_k, x_j)) + \frac{1}{N^2}\sum_{p=0}^{N-1}\sum_{t=0}^{N-1}\mathrm{k}(x_p, x_t) \end{aligned}$$

Therefore we can introduce the kernel $\tilde{\mathrm{k}}$ which takes as input the input vectors $\{x_0, \cdots, x_i, \cdots, x_{N-1}\} \in \mathbb{R}^d$ and computes the scalar product between the centered feature vectors. It can be shown(Scholkopf et al., 1999) that the associated Gram matrix $\tilde{\mathbf{K}}$ is defined as :

$$\tilde{\mathbf{K}} = (\mathbf{I}_N - \frac{1}{N}\mathbf{1})\mathbf{K}(\mathbf{I}_N - \frac{1}{N}\mathbf{1})$$

where $\mathbf{I}_N$ is the identity matrix and $\mathbf{1}$ is the square $N \times N$ matrix with all entries set to 1.

Let us now apply K-PCA to the MNIST dataset. The figure6.4 illustrates the two first principal components of 5000 digits from the MNIST dataset using a RBF kernel with a variance $\sigma = 4.8$ which corresponds to the mean euclidean distance in the image space between the considered digits and their closest neighbor. This non-linear projection captures 7% of the variability of the original data compared to the linear PCA (fig 6.3b) which captured only around 5%. At the time of writting this section, it is not completely clear to which extent the kernel PCA brings any improvement over the PCA when applied to the MNIST dataset. However, there are some datasets for which k-PCA appears superior to PCA in feature extraction and the reader is referred for example to (Scholkopf et al., 1999) in which k-PCA and linear PCA are compared in the context of extracting features feeding a classifier and where it is shown that extracting features with k-PCA leads to better classification performances.

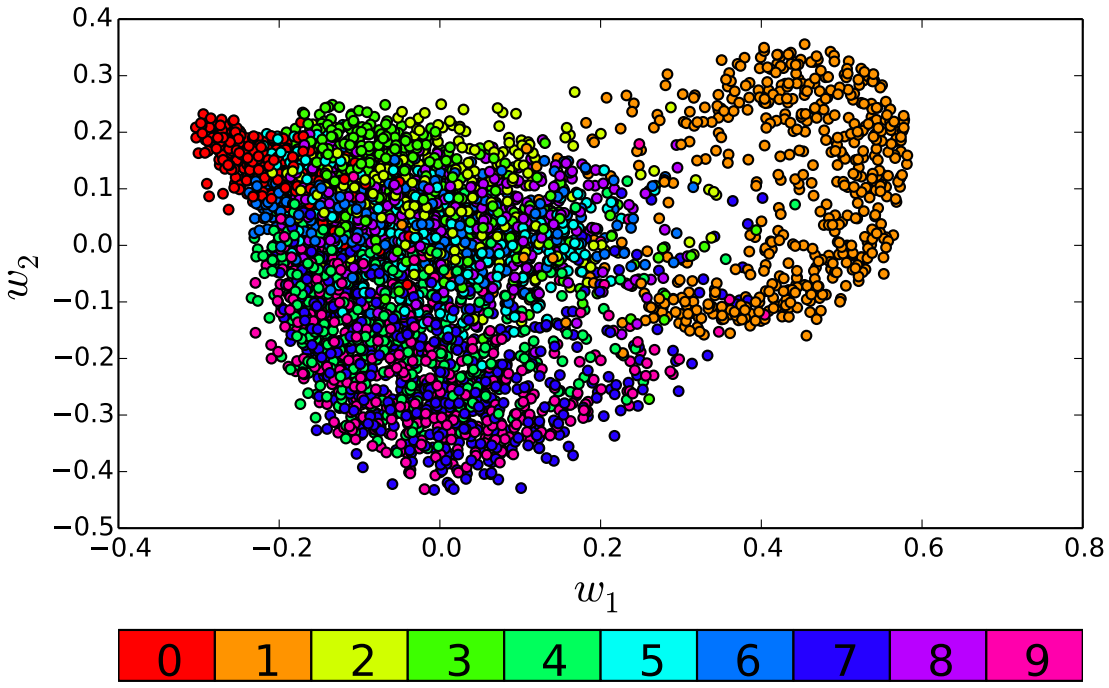


Figure 6.4: The Kernel PCA applied to 5000 samples from the MNIST handwritten digits dataset using a RBF kernel with $\sigma = 4.8$, corresponding to the mean euclidean distance between the considered images and their closest neighbor. Each colored point represents one 28×28 image, the color indicating the associated label. Do not be misleaded, the labels are not used for the k-PCA which is an unsupervised technique. The labels are used after applying the k-PCA, just to get an idea of how the digits are clustered. The two first components computed from 5000 samples capture approximately 7% of the variance.

6.2.6 Further reading

There are a lot of various methods for performing dimensionality reduction. The PCA and kernel PCA both try to preserve the variance of the original data. The kernel PCA is rather time consuming to compute and the Kernel PCA might actually require a lot of principal components to ensure capturing most of the data variance. A Sparse Kernel PCA algorithm proposes to overcome this limitation (Tipping, 2001). The PCA and its variants should really be seen as compressing methods in the sense that these techniques try to extract fewer features than the original number of features, in order to be able to reconstruct as best as possible the original data.

6.2.7 Manifold learning

Some other algorithms rely on a different problem statement. Rather than trying to minimize the reconstruction error of the data, one might try to preserve the distances between the input vectors. These methods really seek to keep the topology of the dataset while transforming it into a smaller dimensional space; these methods are refered to as *manifold learning* methods. In that respect, they can really be used to visualize the data, for exemple, on a 2D screen. With this formulation, we really seak a lower dimensional representation of the original data ensuring that the pairwise distances are preserved and therefore that the layout of the data in 2D is as similar as possible to the layout of the data in the original large dimensional space. This leads to a family of algorithms like the multidimensional scaling (MDS) algorithm(Cox and Cox, 2000), Isomap(Tenenbaum et al., 2000), Sammon mapping(Sammon, 1969), Locally Linear Embedding(Roweis and Saul, 2000), Stochastic Neighborhood Embedding (SNE)(Hinton and Roweis, 2002) and t-SNE (van der Maaten and Hinton, 2008). The reader is referred to (Lee and Verleysen, 2007) where many non linear dimensionality reduction techniques are reviewed. Below, we develop a little bit the t-SNE method, one of the recently and sucessful manifold learning method.

t-Stochastic Neighborhood Embeeding (t-SNE)

Suppose we are given N input vectors $\{x_0, \cdots, x_i, \cdots, x_{N-1}\} \in \mathbb{R}^d$ for which we can define a similarity p_{ij}. We are looking for N output vectors $\{y_0, \cdots, y_i, \cdots, y_{N-1}\} \in \mathbb{R}^r$, with $r \ll d$ so, that denoting q_{ij} the similarity between the i-th and j-th points y_i y_j in the low dimensional space, the similarities p_{ij} and q_{ij} are as close as possible. The t-SNE algorithm relies on specific choices for computing the similarites p_{ij}, q_{ij} and measuring the discrepancy between them. In SNE, the similarities between points in the original high dimensional space are computed as the conditional probability $p_{j/i}$ that x_j would be picked as the neighbor of x_i if the neighbor of x_i were selected according to a normal distribution centered on x_i. from Gaussians centered on the datapoints :

$$\forall i, j, p_{j/i} = \frac{\exp(-\frac{|x_i - x_j|_2}{2{\sigma_i}^2})}{\sum_{k \neq i} \exp(-\frac{|x_i - x_k|_2}{2{\sigma_i}^2})}$$

Note that from this asymmetric definition, within SNE, the fact x_i is picked as the neighbor of x_j does not imply that x_j is picked as the neighbor of x_i. While in SNE, the similarity is directly taken as $p_{ij} = p_{j/i}$, in t-SNE, the similarities are symmetrized :

$$p_{ij} = \frac{p_{j/i} + p_{i/j}}{2N}$$

The similarities in the low dimensional space could have been defined similarly. However, as argued in (van der Maaten and Hinton, 2008), using a gaussian for defining the similarities push too much constraints on the locations of the projections in the low dimensional space. They indeed propose to use a t-Student distribution with 1 degree of freedom which leads to define the similarities q_{ij} as :

$$\forall i, j, q_{ij} = \frac{(1 + |y_i - y_j|_2{}^2)^{-1}}{\sum_{k \neq l}(1 + |y_k - y_l|_2{}^2)^{-1}}$$

The t-Student's distribution with one degree of freedom (or Cauchy distribution) has a heavier tail and allows to push apart a little more the datapoints in the low-dimensional space.

Once the similarities have been defined, it remains to introduce the criteria to be optimized. The dissimilarity between the similarities p_{ij} and q_{ij} can be estimated with the Kullback-Leibler divergence and reads :

$$C = \sum_{i,j} p_{ij} \log\left(\frac{p_{ij}}{q_{ij}}\right)$$

One can then minimize C with respect to the points y_i in the low dimensional space by performing a gradient descent (with momentum) of C with respect to the y_i. The gradient reads (see (van der Maaten and Hinton, 2008)) :

$$\frac{\partial C}{dy_i} = 4 \sum_j (p_{ij} - q_{ij})(1 + |y_i - y_j|_2{}^2)^{-1}(y_i - y_j)$$

The complexity of the algorithm is quadratic because you need to compute all the pairwise distances. In (van der Maaten, 2014), an improvement of the complexity of the algorithm is introduced with an approximation. It is based on the following idea which makes the evaluation of the gradient faster: looking at the gradient for one point y_i, one can see it as a sum of influences or forces which push or pull the point y_i. When some points are far away from y_i, one can approximate their individual forces by a single one originating from the center of mass of these points. By approximating the individual contributions of several far points into a single one, one can use the Barnes-Hutt approximation used in physics and improves the complexity from $O(N^2)$ to $O(N \log N)$. Applying t-SNE on 5000 digits of MNIST is shown on fig. 6.5.

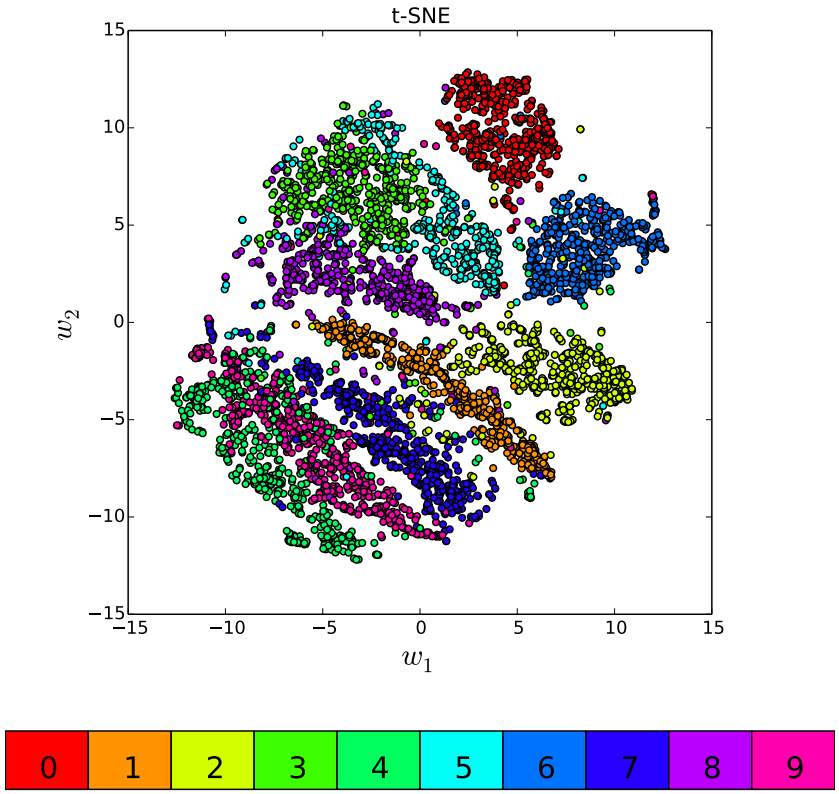


Figure 6.5: The t-SNE applied to 5000 samples from the MNIST handwritten digits dataset, using 2 components and a perplexity of 40(van der Maaten and Hinton, 2008). Each colored point represents one 28×28 image, the color indicating the associated label. Do not be misleaded, the labels are not used for the t-SNE which is an unsupervised technique. However, it turns out that t-SNE nicely clusters the different classes.

Part III

Support vector machines

Chapter 7

Introduction

7.1 Acknowledgment

The elements in the whole part III is mainly the integration of a former document. That document was written in French and translated into English by Cédric Pradalier. The text here thus reuses that translation.

The reader can refer to (Cristanini and Shawe-Taylor, 2000; Shawe-Taylor and Cristanini, 2004; Vapnik, 2000) for a more exhaustive view of what is presented here.

7.2 Objectives

This document part aims at providing a practical introduction to Support Vector Machines (SVM). Although this document presents an introduction to the topics and the foundations of the problem, we refer the reader interested in more mathematical or practical details to the documents cited in the bibliography. Nevertheless, there should be enough material here to get an "intuitive" understanding of SVMs, with an engineering approach allowing a quick and grounded implementation of these techniques.

SVMs involve a number of mathematical notions, among which the theory of generalization – only hinted at here –, optimization and kernel-based machine learning. We will only cover here the aspects of these theoretical tools required to understand what are SVMs.

7.3 By the way, what is an SVM?

An SVM is a machine learning algorithm able to identify a separator. So, the essential question is: what is a separator? Let's consider a finite set of vectors in $\mathbb{R}^n$, separated into two groups, or in other words, into two classes. The membership of a vector to a group or an other is defined by a label with a label "group 1" or "group 2". Building a separator falls down to building a function that takes as input a vector of our set and can then output its membership. SVMs provide a solution to this problem, as would a simple recording of the vector memberships. However, with SVM we expect good properties of generalization: should we consider another vector, not in the initial set, the SVM will probably be able to identify its membership to one of the two groups, based on the membership of the initial set of vectors.

The notion of separator can be extended to situations where the SVM is estimating a regression. In this case, the label associated to a given vector is no longer a discrete membership but a real value. This leads to a different problem since we no longer try to assess to which group a vector belongs (e.g. group 1 or group 2), but we want to estimate how much a vector is "worth".

7.4 How does it work?

The core idea is to define an optimization problem based on the vectors for which we know the class membership. Such a problem could be seen as "optimize such and such value while making sure that ... ". There are two main difficulties: the first is to define the right optimization problem. This notion of "right" problem is what refers to the mathematical theory of generalization, and make SVMs a somewhat difficult-to-approach tool. The second difficulty is to solve this optimization problem once defined. There we enter the realm of algorithmic subtleties from which we will only consider the SMO algorithm.

Chapter 8

Linear separator

We will fist consider the case of a simple (although ultimately not so simple) separator: the linear separator. In practice, this is the core of SVMs, even if they also provide much more powerful separators than the ones we're going to cover in this chapter.

8.1 Problem Features and Notations

8.1.1 The Samples

As mentioned in introduction, we are considering a finite set of labelled vectors. In our specific case, we will refer to the set of given labelled vectors as the set of *samples* S, containing N elements.

$$S = \{(x_1, y_1), \cdots, (x_i, y_i), \cdots, (x_N, y_N)\}.\ \forall i, (x_i, y_i) \in \mathcal{X} \times \mathcal{Y}$$

The membership of a vector to a class is represented here with the value $y \in \mathcal{Y} = \{-1, 1\}$, which will simplify the expressions later. Let us consider $\mathcal{X} = \mathbb{R}^n$ as the input set in the following.

8.1.2 The Linear Separator

The dot product of two vectors will be denoted $x.y$. Using this notation, we can define the *linear separator* $h_{w,b},\ (w, b) \in \mathcal{X} \times \mathbb{R}$ with the following equation:

$$h_{w,b}(x) = w.x + b$$

This separator does not exclusively output values in $\mathcal{Y} = \{-1, 1\}$, but we will consider that when the results of $h_{w,b}(x)$ is positive, the vector x belongs to the same class than the samples labelled $+1$, and that if the result is negative, the vector x belongs to the same class as the samples labelled -1.

Before digging deeper in this notion of linear separator, let's point out that the equation $h_{w,b}(x) = 0$ defines the separation border between the two classes, and that this border is an affine hyperplane in the case of a linear separator.

8.2 Separability

Let's consider again our sample set S, and let's separate it into two sub-set according to the value of the label y. We define:

$$\begin{aligned} S^+ &= \{(x, y) \in S \mid y = 1\} \\ S^- &= \{(x, y) \in S \mid y = -1\} \end{aligned}$$

Stating that S is *linearly separable* means that there exists $w \in \mathcal{X}$ and $b \in \mathbb{R}$ such that:

$$\begin{aligned} h_{w,b}(x) > 0 \quad \forall x \in S^+ \\ h_{w,b}(x) < 0 \quad \forall x \in S^- \end{aligned}$$

This is not always feasible. There can be label distributions over the vectors in S that make S non linearly separable. In the case of samples taken in the $\mathcal{X} = \mathbb{R}^2$ plane, stating that the sample distribution is linearly separable means that we can draw a line (thus an hyperplane) such that the samples of the $+1$ class are on one side of this border and those of the -1 class on the other side.

8.3 Margin

For the following, let's assume that S is linearly separable. This rather strong hypothesis will be relieved later, but for now it will let us introduce some notions. The core idea of SVM is that, additionally to separating the samples of each class, it is necessary that the hyperplane cuts "right in the middle". To formally define this notion of "right in the middle" (cf. figure 8.1), we introduce the *margin*.

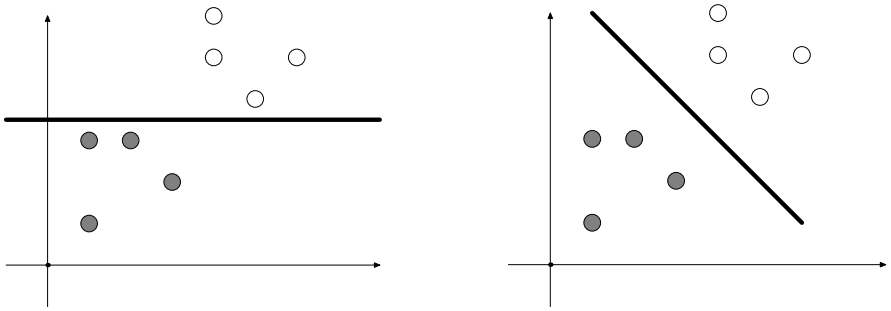


Figure 8.1: The same samples (class -1 or $+1$ is marked with different colors) are, on both figures, separated by a line. The notion of margin allows to qualify mathematically the fact the separation on the right figure is "better" than the one on the left.

Using figure 8.2, we can already make the following observations. First the curves defined by equation $h_{w,b}(x) = C$ are parallel hyperplanes and w is normal to these hyperplanes. The parameter b expresses a shift of the separator plane, i.e. a translation of $h_{w,b}(x)$. The norm $|w|$ of w affects the level set $h_{w,b}(x) = C$. The larger $|w|$, the more compressed the level set will be.

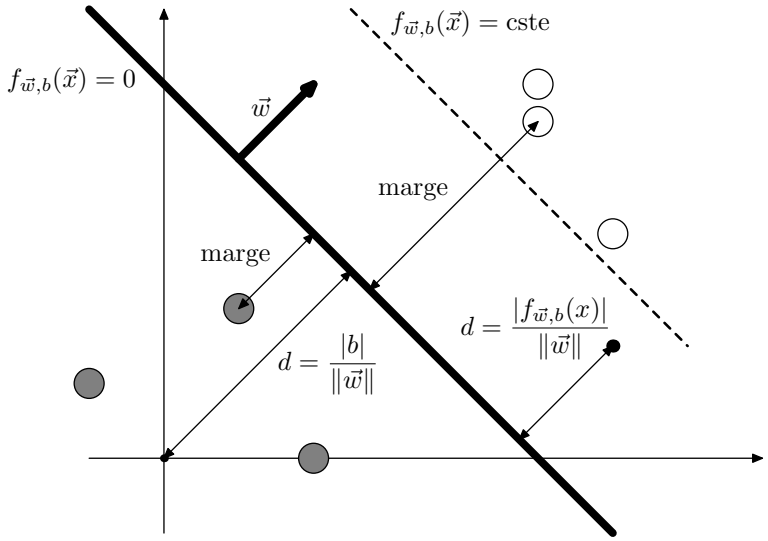


Figure 8.2: Definition of a separator $h_{w,b}(x)$. The values on the graph are the values of the separator at the sample points, not the Euclidian distances. If the Euclidian distance of a point x to the separation border is d, $|h_{w,b}(x)|$ on that point is $d|w|$.

When looking for a given separating border, we are faced with an indetermination regarding

the choice of w and b. Any vector w not null and orthogonal to the hyperplane will do. Once this vector chosen, we determine b such that $b/|w|$ is the oriented measure[1] of the distance from the origin to the separating hyperplane.

The margin is defined with respect to a separator $h_{w,b}$ and a given set of samples S. We will denote $\gamma_{h_{w,b}}(S)$ this margin. It is defined from the function $\gamma_{h_{w,b}}(x,y)$ computed from each sample (x,y), also called margin, but rather sample margin. This latter margin is:

$$\gamma_{h_{w,b}}(x,y) = y \times h_{w,b}(x) \tag{8.1}$$

Since $y \in \{-1,1\}$ and a separator puts samples with label $+1$ on the positive side of its border and those of class -1 on the negative side, the sample margin is, up to the norm of w, the distance from a sample to the border. The (overall) margin, for all the samples, is simply the minimum of all margins:

$$\gamma_{h_{w,b}}(S) = \min_{(x,y)\in S} \gamma_{h_{w,b}}(x,y) \tag{8.2}$$

Coming back to the two cases of figure 8.1, it seems clear that on the right side – the better separator –, the margin $\gamma_{h_{w,b}}(S)$ is larger, because the the border cuts further from the samples. Maximizing the margin is most of the work of an SVM during the learning phase.

[1] The direction of w defines the positive direction.

Chapter 9

An optimisation problem

9.1 The problem the SVM has to solve

9.1.1 The separable case

In continuity with chapter 8, let's assume that we are still given a sample set S that is separable with a linear separator. If the separator effectively separate S, with on the positive side, all the $+1$-labelled samples, and on the negative side, the ones labelled -1, then all the sample margin are positive (cf. eq. (8.1)). When one of the margin is negative, then the separator is not correctly separating the two classes although it should be possible since we assume the sample set linearly separable. In this incorrect case, $\gamma_{h_{w,b}}(S)$ is negative (cf. eq. (8.2)). Thus, maximising the margin means first that we separate (positive margin) and then that we separate well (maximal margin).

A separator with maximal margin is such that it makes the margin of the sample with the smallest margin larger than the margin of the sample with the smallest margin for all the other possible separators.

Let's consider this sample of smallest margin. In fact, there can be more than one (with equal margins), belonging to either the positive or the negative classes. Actually, considering figure 9.1, there are necessarily one sample of the positive class and one sample of the negative class that constrain this margin, and the separating border has to cut right in the middle. We can note as well that only these examples constrain the separating hyperplane, and that we could remove all the others from the sample set without changing the maximal-margin separator. This is why these samples are named *support vectors*.

Note that the separating hyperplane with maximal margin is defined up to a constant scale, and that we can chose this scale such that the support vectors lie on the level set with value $+1$ and -1. Figure 9.2 illustrates this point. In this particular case, the distance from the support vectors to the separating plane – i.e. the margin – is simply $1/|w|$.

Starting from this observation, let's consider the case where S is separable. Figure 9.3 depicts two separators such that all the samples lie not only on the right side ($\gamma_{h_{w,x}}(x,y) > 0$), but also outside of the band defined by the level set -1 and $+1$ ($\gamma_{h_{w,x}}(x,y) > \pm 1$).

The width of the band delimited by the level set -1 and $+1$ is $2/|w|$. To find the separator of maximal margin, it is then only necessary to search the separator for which $|w|$ is minimal among the separators analoguous to those from figure 9.3, that is among all those that verify, for all samples, $\gamma_{h_{w,x}}(x,y) > 1$. Minimising $|w|$ reduces to widening the band $-1/+1$ until it "blocks" against the support vectors, which brings us back to the situation depicted in figure 9.2.

The hyperplane with maximal margin, for a sample set $S = \{(x_1, y_1), \cdots, (x_i, y_i), \cdots, (x_N, y_N)\}$, is thus the solution to the following optimization problem, in which the coefficient $1/2$ is just added to simplify the calculus of the derivative in the coming sections:

$$\begin{aligned} \text{find} \ \underset{w,b}{\text{argmin}} \quad & \frac{1}{2} w.w \\ \text{subject to} \quad & y_i(w.x_i + b) \geq 1, \ \forall (x_i, y_i) \in S \end{aligned}$$

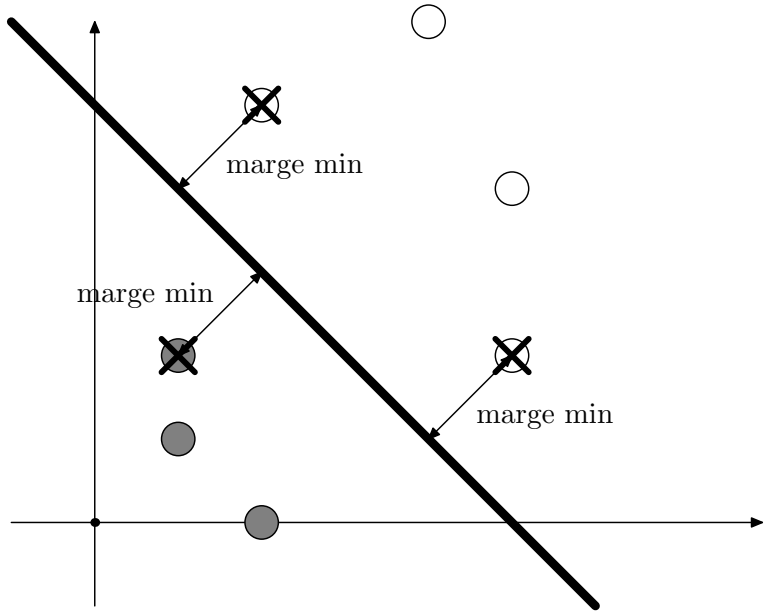


Figure 9.1: The separator with maximal margin as a border which is defined with at least one sample of each class. The *Support Vectors* are marked with a cross.

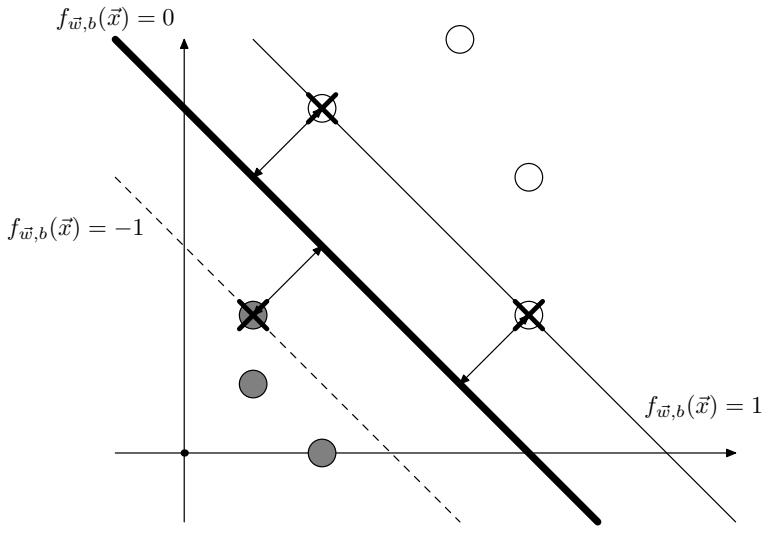


Figure 9.2: The separator in this figure has the same border as the one from figure 9.1, so it separates the samples with the same quality. However, the difference is that the level set $h_{w,b}(x) = 1$ and $h_{w,b}(x) = -1$ contains the support vectors. To achieve this, it was necessary to modify the norm of w, and adjust b accordingly.

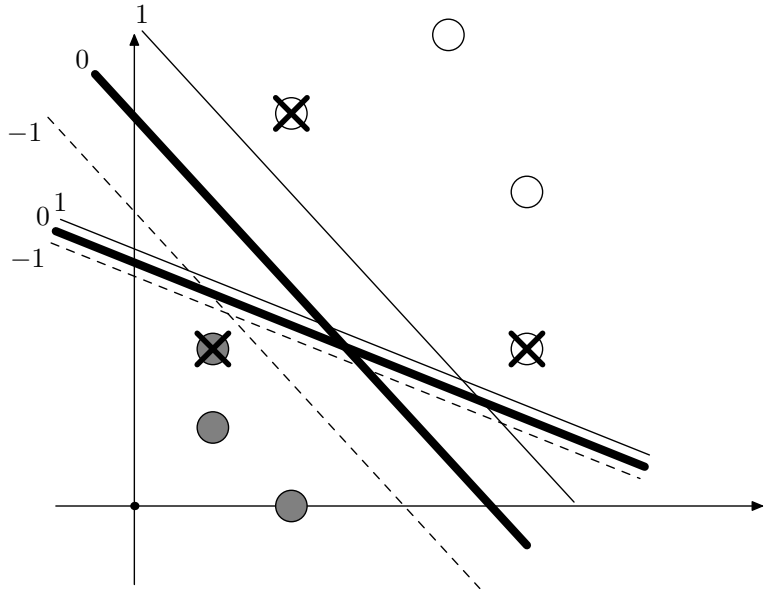


Figure 9.3: Both separators in this figure make the margin of all samples greater than 1. For both of them, the width of the bands is $2/|w|$, where w is the term appearing in $h_{w,b}(x) = w.x + b$. Consequently, the "most vertical" separator on this figure has a vector w with a smaller norm than the other.

9.1.2 General case

For the general case where S is not separable, the solution consist in authorizing some of the samples to have a margin smaller than one, or even negative. To this end, we will transform the constraint $y_i(w.x_i + b) \geq 1$ into $y_i(w.x_i + b) \geq 1 - \xi_i$, with $\xi_i \geq 0$. Obviously, doing that without constraints leads to unwanted effects, since we could minimize $w.w$ down to zero by selecting large enough ξ_i. To avoid this, we will add the variables ξ_i, names *slack variables* in the minimization problem so as to prevent them to become exceedingly large. This will limit the influence of the samples violating the separability on the separating solution to the optimization problem. In summary, the optimization problem becomes the following, with C a positive parameter defining the tolerance of the SVM to incorrectly separated samples:

$$\begin{array}{rl} \text{find } \underset{w,b,\xi}{\operatorname{argmin}} & \frac{1}{2} w.w + C \sum_i \xi_i \\ \text{subject to} & \left\{ \begin{array}{l} y_i(w.x_i + b) \geq 1 - \xi_i, \ \forall (x_i, y_i) \in S \\ \xi_i \geq 0, \ \forall i \end{array} \right. \end{array} \tag{9.1}$$

9.1.3 Relation with the ERM

We have introduced the SVMs as a learning process which maximizes the margin for separating the two classes. This formulation is different from minimizing an empirical risk, whereas the empirical risk minimization (ERM) is the induction principle which is legitimated theoretically, as introduced in chapter 5.

The margin optimization process described here could be though as another induction principle, since it does not rely on minimizing the total amount of errors performed on the learning dataset. Nevertheless, the margin optimization can be described as an ERM process thanks to the formulation of a specific loss function.

Let us recall notations. We have a decision function $h(x) = w.x + b$ which associates a scalar to the input. The sign of this scalar should be the same as the associated label $y \in \{-1, 1\}$, in other words, the value $yh(x)$ should be positive for correctly classified data. The loss associated to a data sample (x, y) is $L(h(x), y)$. Let us recall (see paragraph 5.2.1) the *hinge loss* as

$$L_{\text{hinge}}(h(x), y) = \max(0, 1 - yh(x)).$$

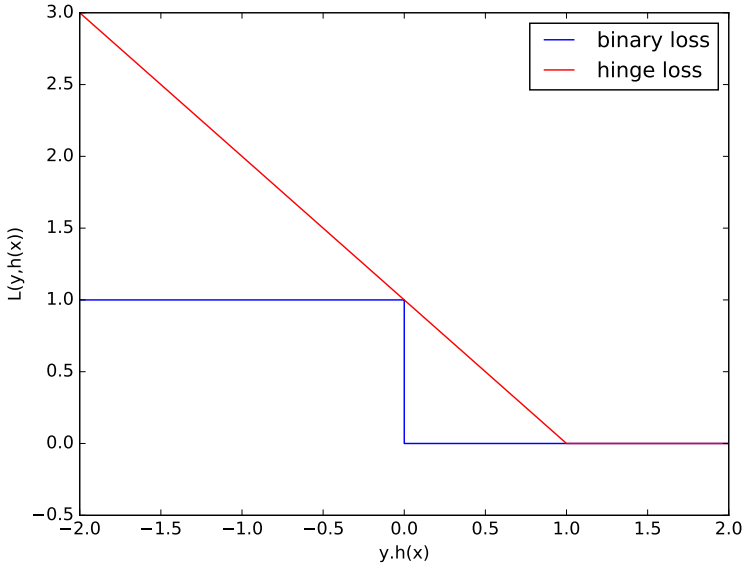


Figure 9.4: Binary and hinge losses.

This loss approximates the binary loss, as figure 9.4 shows. Let us now reconsider equation (9.1). The two constraints can be rewritten as as

$$\xi_i \geq \max\left(0, 1 - y_i(w.x_i + b)\right)$$

For a given (w, b), the minimization of the objective function in equation (9.1) is improved when the sum is minimal, i.e when each ξ_i is minimal. Therefore, as the slack variables should be minimal, we can use the previous expression as an equality before injecting it in the objective function without changing the optimal solution. So the minimization problem of equation (9.1) can be rewritten as

$$\text{find} \underset{w,b}{\operatorname{argmin}} \frac{1}{2} w.w + C \sum_i \max\left(0, 1 - y_i h\left(x_i\right)\right)$$

which is

$$\text{find} \underset{w,b}{\operatorname{argmin}} \sum_i L_{\text{hinge}}\left(h\left(x_i\right), y_i\right) + \lambda w.w.$$

This last formulation is an actual ERM problem, with an additional regularization term $\lambda w.w$.

9.2 Lagrangian resolution

9.2.1 A convex problem

In the following, we will assume that the SVM optimization problem is convex, that is to say, this optimization has a single global optimum and no local optima. This statement, not justified here, is essential because the problem convexity is what ensure the convergence of the SVM algorithms towards **the** optimal solution.

9.2.2 The direct problem

The intent in this document is not to introduce *optimization theory*, but to bring in the absolute minimum to understand the relation with the SVMs. In section 9.1.2, we defined an *optimization problem* in the following form:

$$\begin{aligned} \text{find} \underset{k}{\operatorname{argmin}} \quad & f(k) \\ \text{subject to} \quad & g_i(k) \leq 0, \ 1 \leq i \leq n \end{aligned}$$

Solving this problem require defining the following function, called problem's *Lagrangian*, which involves the constraints multiplied by coefficients $\alpha_i \geq 0$. These coefficients are named *Lagrange multipliers*. The constraints g_i are affine.

$$\mathcal{L}(k, \alpha) = f(k) + \sum_{1 \leq i \leq n} \alpha_i g_i(k)$$

In this case, the theory says that the vector $k^\star$ minimizing $f(k)$ while respecting the constraints must satisfy that $\mathcal{L}$ has a saddle point at $(k^\star, \alpha^\star)$. At this point, $\mathcal{L}$ is a minimum for k and a maximum for α:

$$\forall k,\ \forall \alpha \geq 0,\ \mathcal{L}(k^\star, \alpha) \leq \mathcal{L}(k^\star, \alpha^\star) \leq \mathcal{L}(k, \alpha^\star)$$

The following stands at the optimum

$$\frac{\partial}{\partial k}\mathcal{L}(k^\star, \alpha^\star) = 0$$

whereas $\frac{\partial}{\partial \alpha}\mathcal{L}(k^\star, \alpha^\star)$, that should be null as well at the saddle point, may be not defined (see the top-right frame in figure 9.5)

These conditions are sufficient to define the optimum if the Lagrangian is a convex function, which will be the case for SVMs. See Cristanini and Shawe-Taylor (2000) for supplementary mathematical justifications.

The challenge is that writing these conditions does not always lead easily to a solution to the initial problem. In the case of SVMs, solving the dual problem will result in an easier (although not easy) solution.

9.2.3 The Dual Problem

The *dual problem* is the result of inserting the optimality constraints given by the terms $\frac{\partial}{\partial k}\mathcal{L} = 0$ in the expression of $\mathcal{L}$. This expression will only involve the multipliers, but in a maximization problem under an other set of constraints.

This comes from the saddle shape of the Lagrangian around the optimum: the optimum is a minimum along k, but a maximum along α (cf. figure 9.5). Injecting $\frac{\partial}{\partial k}\mathcal{L} = 0$ in $\mathcal{L}(k, \alpha)$ is equivalent to defining a function $\theta(\alpha)$ which for α expresses the minimum value of the Lagrangian, that is to say the minimal value of $\mathcal{L}$ resulting from setting α and minimizing by playing on k. It remains then to maximize $\theta(\alpha)$ by playing on α, which is the dual optimization problem.

The advantage of the dual problem in the context of SVMs will become clearer as soon as we leave the generalities of optimization theory and come back to its application to SVMs.

9.2.4 An intuitive view of optimization under constraints

This section will give an intuitive view of the theory of optimization under constraints. However, the impatient reader may accept these theorems and directly jump to the next section. Thanks to Arnaud Golinvaux[1] for his contribution to the following.

Optimization under equality constraints

Let's start from the following optimization problem, while keeping in mind figure 9.6.

$$\begin{aligned} \text{find} \quad & \underset{k}{\text{argmin}} \quad f(k) \\ \text{subject to} \quad & g(k) = 0 \text{ where } g(k) = (g_i(k))_{1 \leq i \leq n} \end{aligned}$$

Let's consider the solution $k^\star$ of our optimization problem. This solution, due to the constraints, can be different from the minimum K of the objective function (see figure 9.6). The constraints in g are a function of k. In principle, we're only looking for $k^\star$ in the kernel of g, i.e. amongst the k such that $g(k) = 0$.

[1] 3rd year student at Supélec in 2012

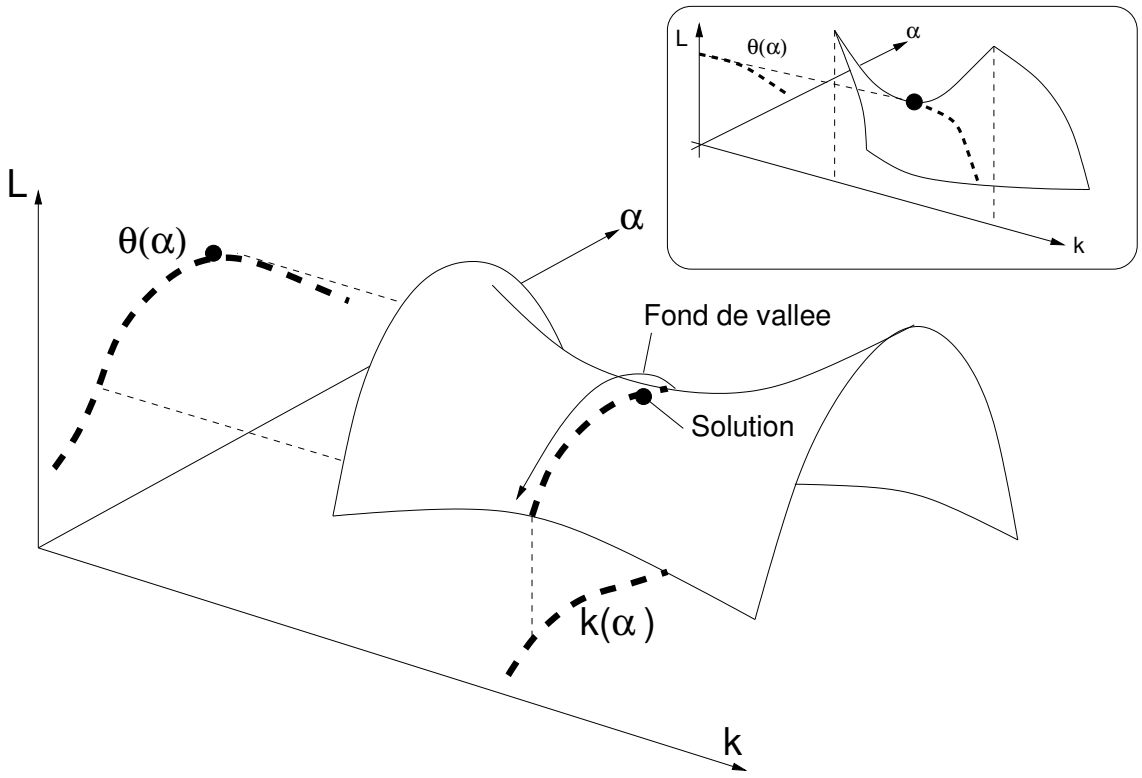


Figure 9.5: Transition from the primal problem to the dual problem. The Lagrangian $\mathcal{L}(k, \alpha)$ is saddle-shaped around the solution of the problem. The "bottom of the valley", i.e. the minima along k, is represented on the saddle by a dashed line. The equation $\frac{\partial}{\partial k}\mathcal{L} = 0$ enables us to link k and α, so as to express k as a function of α, denoted $K(\alpha)$. This link is the projection of the valley to the "horizontal" plane. Injecting this relation into $\mathcal{L}$ gives a function $\mathcal{L}(K(\alpha), \alpha) = \theta(\alpha)$. This function is the objective function of the dual problem, which we're trying to maximize, as shown in the figure.

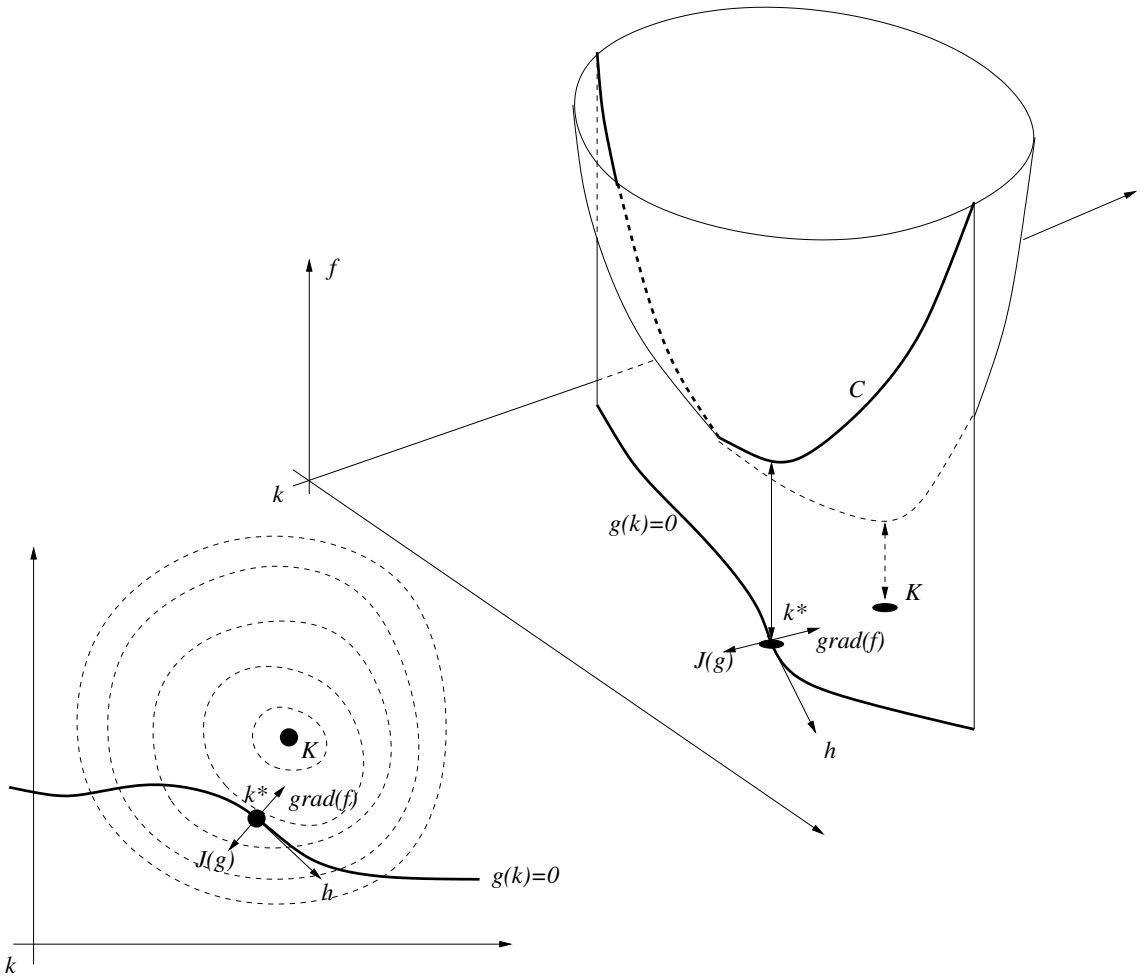


Figure 9.6: See text.

Staying in the kernel (i.e. on the bold curve in the figure) leads to the following consequence around the solution $k^\star$. Let h an infinitesimal displacement around the optimum, such that $k^\star + h$ is still in the kernel of g. We have:

$$\begin{array}{lcl} g(k^\star) & = & 0 \\ g(k^\star + h) & = & 0 \\ g(k^\star + h) & = & g(k^\star) + \mathrm{J}g|_k\,(k^\star)\,.h \end{array}$$

Where $\mathrm{J}g|_k\,(k_0)$ represents the Jacobian matrix of g with respect to k, taken at k_0. We can immediately deduce that:

$$\mathrm{J}g|_k\,(k^\star)\,.h = 0$$

A displacement h satisfying the constraints around the solution is thus included in the vector space defined by the kernel of the Jacobian matrix:

$$h \in \ker\left(\mathrm{J}g|_k\,(k^\star)\right)$$

Let's consider now the consequence of this displacement regarding our objective function f. By linearizing around $k^\star$ while using the constraint-satisfying displacement h above, we have:

$$f(k^\star + h) = f(k^\star) + \nabla f|_k\,(k^\star)\,.h$$

As a reminder, the gradient is a degenerated Jacobian when the function is scalar instead of vector-valued. Being around $k^\star$ means that $f(k^\star)$ is mininal, so long as our displacements respect the constraints (in bold on figure 9.6, i.e. the curve C). So $\nabla f|_k\,(k^\star)\,.h >= 0$. However, similarly to h, $-h$ satisfies the constraints as well, since the set $\ker\left(\mathrm{J}g|_k\,(k^\star)\right)$ of h satisfying the constraints is a matrix kernel and as such a vector space. Hence, we have as well $-\nabla f|_k\,(k^\star)\,.h >= 0$. From this, we can deduce that

$$\nabla f|_k\,(k^\star)\,.h = 0, \forall h \in \ker\left(\mathrm{J}g|_k\,(k^\star)\right)$$

In other words, $\nabla f|_k\,(k^\star)$ is in the vector sub-space E orthogonal to $\ker\left(\mathrm{J}g|_k\,(k^\star)\right)$. But this space E happens to be the one spanned by the column vectors of the matrix $\mathrm{J}g|_k\,(k^\star)$. Hence, we can confirm that $\nabla f|_k\,(k^\star)$ is a linear combination of these column vectors.

But, since g is the vector of the n scalar contraints g_i,

$$\mathrm{J}g|_k\,(k^\star) = \left[\nabla g_1|_k\,(k^\star)\,,\, \nabla g_2|_k\,(k^\star) \cdots \nabla g_n|_k\,(k^\star)\right]$$

so these column vectors are the gradients of each of the constraints with respect to the parameters. As a result, we have:

$$\exists(\alpha_1, \cdots, \alpha_n) \in \mathbb{R} : \nabla f|_k\,(k^\star) + \sum_{i=1}^{n} \alpha_i\, \nabla g_i|_k\,(k^\star) = 0$$

Hence, the idea to set the following Lagrangian:

$$\mathcal{L}\,(k, \alpha) = f(k) + \sum_{i=1}^{n} \alpha_i g_i(k)$$

which gradient with respect to k must be null at the constrained optimum.

The case of inequality constraints

Let's consider now the following optimization problem:

$$\begin{array}{rcl} \text{find } \underset{k}{\text{argmin}} & & f(k) \\ \text{subject to} & & g_i(k) \leq 0,\ 1 \leq i \leq n \end{array}$$

The idea is to associate to each constraint $g_i(k) <= 0$ a new scalar parameter y_i. We group the y_i of inequality constraints into a vector y. We set $g'_i(k, y) = g_i(k) + {y_i}^2$. An optimization problem

with inequality constraints thus become a problem with equality constraints, but with additional parameters.

$$\begin{aligned} \text{find} \ \underset{k,y}{\text{argmin}} \quad & f(k) \\ \text{subject to} \quad & g'(k,y) = 0 \end{aligned}$$

The trick is that, necessarily, the new parameters in y cannot influence the objective function $f(k)$. Be aware though, that every that was written earlier on the parameter k must now be applied to the union of k and y in this new problem. Following the discussion above, we can set the following Lagrangian:

$$\mathcal{L}(k,y,\alpha) = f(k) + \sum_{i=1}^{n} \alpha_i (g_i(k) + {y_i}^2)$$

The gradient of the Lagrangian with respect to the parameters (now k and y) must be null. It is thus null if we consider it with respect to k and with respect to y.

By differentiating with respect to k, we still have:

$$\nabla f|_k (k^\star) + \sum_{i=1}^{n} \alpha_i \ \nabla g_i|_k (k^\star) = 0$$

By differentiating along y_i, we get $2\alpha_i y_i = 0$, which means either $\alpha_i = 0$ or $y_i = 0$. But $y_i = 0$ is equivalent, from the definition of the new constraints g', to $g_i(k) = 0$.... and the k we're referring to is the one defined by the other derivative of the Lagrangian, i.e. $k^\star$. This is why, at the optimum, we have $\alpha_i g_i(k^\star) = 0$ (these are the condition KKT that will be of use later). These conditions are important because the value of α_i at the optimum let us know whether the constraint g_i is saturated or not. A non-zero value of α_i corresponds to a saturated constraint (equality).

The second derivative of the Lagrangian with respect to y_i happens to be α_i. Intuitively[2], if we refer to curve C in figure 9.6, and by keeping in mind that the parameters y_i are now part of the parameters k in this figure, we can clearly see that the curvature of C is positivie (upward), and thus that the second derivative is positive. Hence, $\alpha_i \geq 0$.

In practice, we finally ended up completely suppressing any reference to y_i in the expression of our conditions of interest. Thus, we can avoid bringing in these variables in the Lagrangian, and although it contains inequality constraints, we can keep the following Lagrangian for our problem:

$$\begin{aligned} & \mathcal{L}(k,\alpha) = f(k) + \sum_{i=1}^{n} \alpha_i g_i(k) \\ & \nabla \mathcal{L}|_k (k^\star, \alpha) = 0 \\ & \forall i, \ \alpha_i \geq 0 \\ & \forall i, \ \alpha_i g_i(k^\star) = 0 \end{aligned}$$

9.2.5 Back to the specific case of SVMs

Let's come back the to the problem defined in section 9.1.2, and let's denote with α_i and μ_i the coefficients relative to the two types of constraints. After having rewritten these constraints so as to make inequalities appear ($\leq$) and match the form seen in section 9.2.2, we can define the Lagrangian of our problem as:

$$\begin{aligned} \mathcal{L}(w,b,\xi,\alpha,\mu) &= \frac{1}{2} w.w + C \sum_{i}^{N} \xi_i - \sum_{i}^{N} \alpha_i (y_i (w.x_i + b) + \xi_i - 1) - \sum_{i}^{N} \mu_i \xi_i \\ &= \frac{1}{2} w.w + \sum_{i}^{N} \xi_i (C - \alpha_i - \mu_i) + \sum_{i}^{N} \alpha_i - \sum_{i}^{N} \alpha_i y_i (w.x_i + b) \\ & \forall i, \ \alpha_i \geq 0 \\ & \forall i, \ \mu_i \geq 0 \end{aligned}$$

[2]To develop this proof correctly, please refer to a math text book

The two types of constraints of our problem are inequalities (cf. section 9.1.2). The theory says then that if a constraint is saturated, that is to say if it is actually an equality, then its Lagrange multiplier is not null. When it is a strict inequality, the multiplier is null. So, for a constraint $g_i(...) \leq 0$ for which the associated multiplier would be k_i, we have either $k_i = 0$ and $g_i(...) < 0$, or $k_i > 0$ and $g_i(...) = 0$. These two cases can be summarised into a single expression $k_i g_i(...) = 0$. This expression is named suplementary condition de KKT[3]. In our problem, we can then express six KKT conditions: the constraints, the multipliers positive sign and the suplementary KKT conditions:

$$\forall i,\ \alpha_i \geq 0 \qquad \text{(KKT1)}$$

$$\forall i,\ \mu_i \geq 0 \qquad \text{(KKT2)}$$

$$\forall i,\ \xi_i \geq 0 \qquad \text{(KKT3)}$$

$$\forall i,\ y_i(w.x_i + b) \geq 1 - \xi_i \qquad \text{(KKT4)}$$

$$\forall i,\ \mu_i \xi_i = 0 \qquad \text{(KKT5)}$$

$$\forall i,\ \alpha_i(y_i(w.x_i + b) + \xi_i - 1) = 0 \qquad \text{(KKT6)}$$

This being defined, let us set to zero the partial derivative of the Jacobian with respect to the terms that are not Lagrange multipliers:

$$\frac{\partial}{\partial w}\mathcal{L} = 0 \Rightarrow w = \sum_i^N \alpha_i y_i x_i \qquad \text{(L1)}$$

$$\frac{\partial}{\partial b}\mathcal{L} = 0 \Rightarrow \sum_i^N \alpha_i y_i = 0 \qquad \text{(L2)}$$

$$\frac{\partial}{\partial \xi_i}\mathcal{L} = 0 \Rightarrow \forall i,\ C - \alpha_i - \mu_i = 0 \qquad \text{(L3)}$$

Equation L1, injected into the expression of the Lagrangian, let us remove the term w. Equation L3 let us remove the μ_i as well. L2 let us eliminate b, which is now in $\mathcal{L}$ multiplied by a null term. After these substitutions, we now have a Lagrangian expression that depends only of the α_i. We will now maximize it by playing on these α_i, knowing that injecting L1, L2 and L3 already guarantee that we have a minimum with respect to w, ξ and b. This is the dual problem. The constraints on this problem can be inferred from the constraints on the α_i resulting from the equations KKTi. Using L3, KKT2 and KKT3, we can show the following by considering the two cases resulting from KKT5:

- either $\xi_i = 0, \mu_i > 0$, and then $0 \leq \alpha_i < C$,
- or $\xi_i > 0, \mu_i = 0$, and then $\alpha_i = C$ according to L3.

The constraints on the α_i are thus $0 \leq \alpha_i \leq C$ and L2. Hence, we must solve the following optimization problem, dual from our initial problem, to find the Lagrange multipliers α_i.

$$\text{find } \underset{\alpha}{\text{argmax}} \quad \sum_i \alpha_i - \frac{1}{2}\sum_j^N \sum_i^N \alpha_j \alpha_i y_j y_i x_j.x_i$$

$$\text{subject to} \quad \begin{cases} \sum_i \alpha_i y_i = 0 \\ \forall i,\ 0 \leq \alpha_i \leq C \end{cases}$$

Additionally, from the two cases mentioned above, we can also deduce that $\xi_i(\alpha_i - C) = 0$. This means that accepting a badly separated sample x_i ($\xi_i \neq 0$) is equivalent to using its α_i with a maximum value of C.

One interesting aspect of this expression of the dual problem is that it only involves the samples x_i, or more precisely only their dot products. This will be useful later when moving away from

[3]Karush-Kuhn-Tucker

linear separators. Furthermore, the vector of the separating hyperplane being defined by L1, it is the result of contributions from all the samples x_i, with a value of α_i. However, these values, after optimization, could be found to be zero for many samples. Such samples will thus have no influence on the definition of the separator. Those that remain, that is those for which α_i is non-zero, will be named support vectors, because they are the ones that define the separating hyperplane.

Solving the dual problem is not trival. So far, we only defined the problem. In particular, b has now disappeared from the dual problem and we will have to work hard[4] to find it back once this problem solved. We'll discuss this point further in chapter 11.

Let's complete this chapter with an example of linear separation, where the separator is the solution of the optimization problem we've defined earlier. The samples that effectively influence the expression L1 with a non-zero coefficient α_i, i.e. the support vectors, are marked with a cross in figure 9.7.

The separation is thus defined with the following equation:

$$h_{w,b}(x) = \left(\sum_i^N \alpha_i y_i x_i\right).x + b$$

which we will rather write as follows, to only involve the samples through their dot products:

$$h_{w,b}(x) = \sum_i^N \alpha_i y_i x_i.x + b \tag{9.2}$$

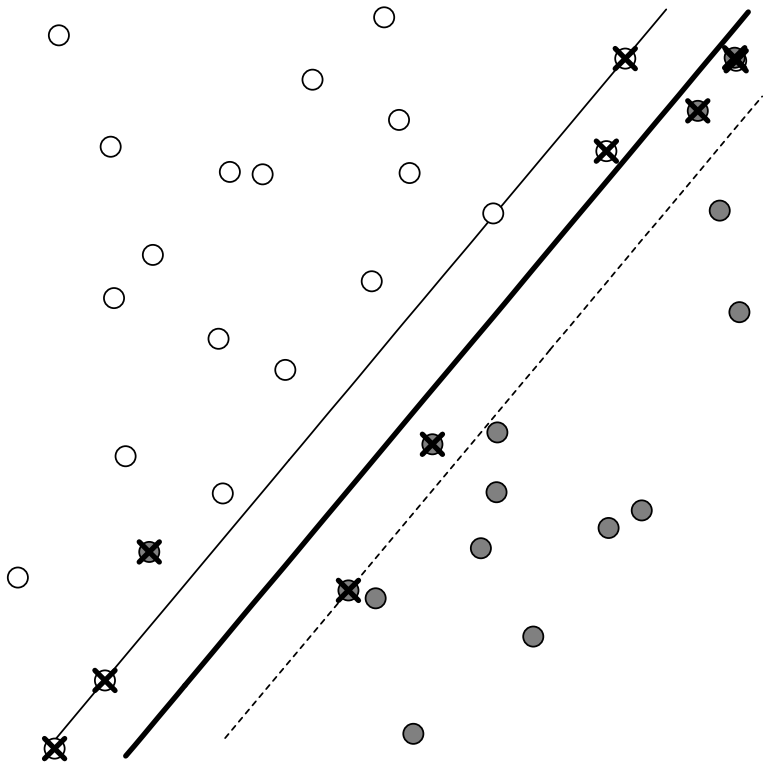


Figure 9.7: Hyperplane resulting from the resolution of the optimization problem from section 9.1.2. The separation border $h_{w,b}(x) = 0$ is the bold curve. The curve $h_{w,b}(x) = 1$ is shown as a thin line, and $h_{w,b}(x) = -1$ in a dashed line. The *support vectors* are marked with crosses.

[4]See section 11.1.2 page 121

Chapter 10

Kernels

The main interest of kernels in the context of SVM is that everything we will written in next chapters on linear separation also applies readily to non-linear separation once we bring *kernel*s in, so long as we do it right.

10.1 The feature space

Let's imagine that a set of samples S, labelled with -1 or $+1$ according to their membership as before, but not linearly separable. The methods we've seen up to last chapter will obviously work, but they will provide a low-quality solution and a lot of the samples will be support vectors (cf. figure 10.1).

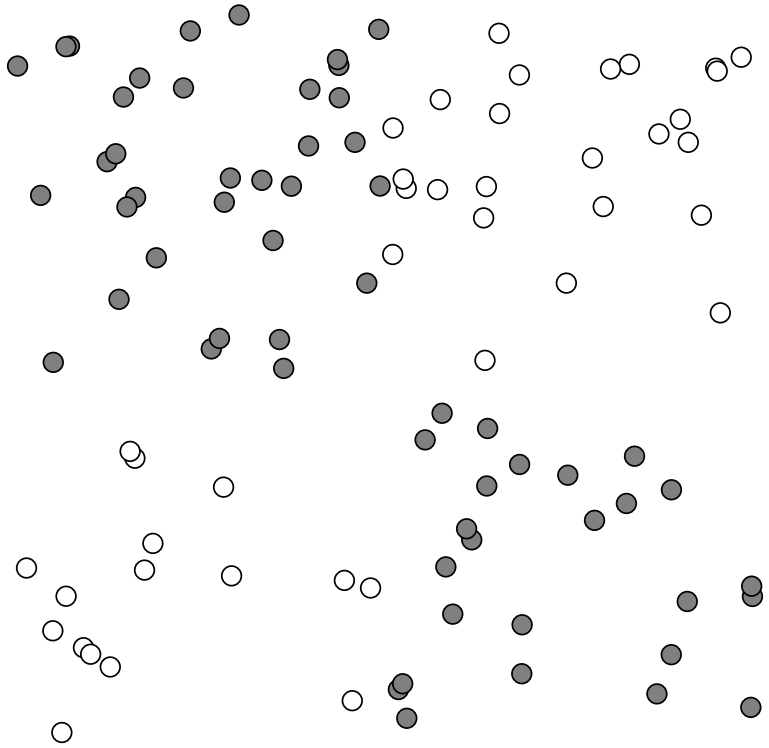


Figure 10.1: Non linearly separable samples.

In order to build a better separation of the samples, a solution is to project the samples into a different space[1], and to implement a linear separation in this space where it will hopefully work better.

[1] Often of very high dimension

Let φ be this projection. We have:

$$\varphi(x) = \begin{pmatrix} \varphi^1(x) \\ \varphi^2(x) \\ \varphi^3(x) \\ \vdots \\ \varphi^n(x) \end{pmatrix}$$

Obviously, the functions φ^i are not necessarily linear. Furthermore, we can have $n = \infty$! So, if we use the approaches seen in the previous chapters 9 and apply them in the feature space, that is to say, if we work with the following sample set with binary labels ($\mathcal{Y} = \{-1, 1\}$):

$$\varphi(S) = \{(\varphi(x_1), y_1) \cdots (\varphi(x_i), y_i) \cdots (\varphi(x_N), y_N)\}$$

instead of

$$S = \{(x_1, y_1), \cdots, (x_i, y_i), \cdots, (x_N, y_N)\}$$

then, we just have to perform a linear separation on the corpus $\varphi(S)$. Using equation L1, we get a separator w and a value for b. Now, to decide the class of a new vector x, we could compute $\varphi(x)$ and apply the separator on $\varphi(x)$ to find out its class membership, -1 or $+1$.

In practice, we will avoid computing explicitly $\varphi(x)$ by noting that in the optimization problem defined in chapter 9 only involves the samples through dot products of pairs of samples.

Let's denote $\mathrm{k}(x, x')$ the product $\varphi(x).\varphi(x')$. Working on corpus $\varphi(S)$ is equivalent to working on corpus S with the algorithms of chapter 9, but replacing every occurence of $\bullet.\bullet$ with $\mathrm{k}(\bullet, \bullet)$.

So far, the interest of kernels should not be obvious, because to compute $\mathrm{k}(x, x')$, we still need to apply its definition, that is to project x and x' in the feature space and to compute, in the feature space, their dot product.

However, the trick, known as the *kernel trick*, is that we will actually avoid performing this projection because we will compute $\mathrm{k}(x, x')$ in an other way. Actually $\mathrm{k}(x, x')$ is a function that we will choose, making sure that there exists, in theory, a projection φ into a space that we will not even try to describe. By this way, we will compute directly $\mathrm{k}(x, x')$ each time the algorithm from chapter 9 refers to a dot product, and that's all. The projection into the huge feature space will be kept implicit.

Let's take an example. Consider

$$\mathrm{k}(x, x') = \exp\left(-\frac{|x - x'|^2}{2\sigma}\right)$$

It is well known that this function corresponds to the dot product of the projection of x and x' into an infinite dimension space. The optimization algorithm that will use this function, also known as kernel, will compute a linear separation in this space while maximizing the margin, without having to perform any infinite loop to compute the products by multiplying terms of the projected vectors two by two!

The separation function is then directly inspired from equation (9.2) page 109, once the optimal α_i found and b computed,

$$\mathrm{sep}(x) = \sum_i^N \alpha_i y_i \mathrm{k}(x_i, x) + b$$

knowing that many of the terms of the sum are zero if the problem is actually separable. In our case, the separator can then be rewritten as:

$$\mathrm{sep}(x) = \sum_i^N \alpha_i y_i \exp\left(-\frac{|x_i - x|^2}{2\sigma}\right) + b$$

The level set $\mathrm{sep}(x) = 0$ define the separation border between the classes, and the level set $\mathrm{sep}(x) = 1$ and $\mathrm{sep}(x) = -1$ represents the margin. Figure 10.2 depicts the result of the algorithm given in chapter 9 with our kernel function.

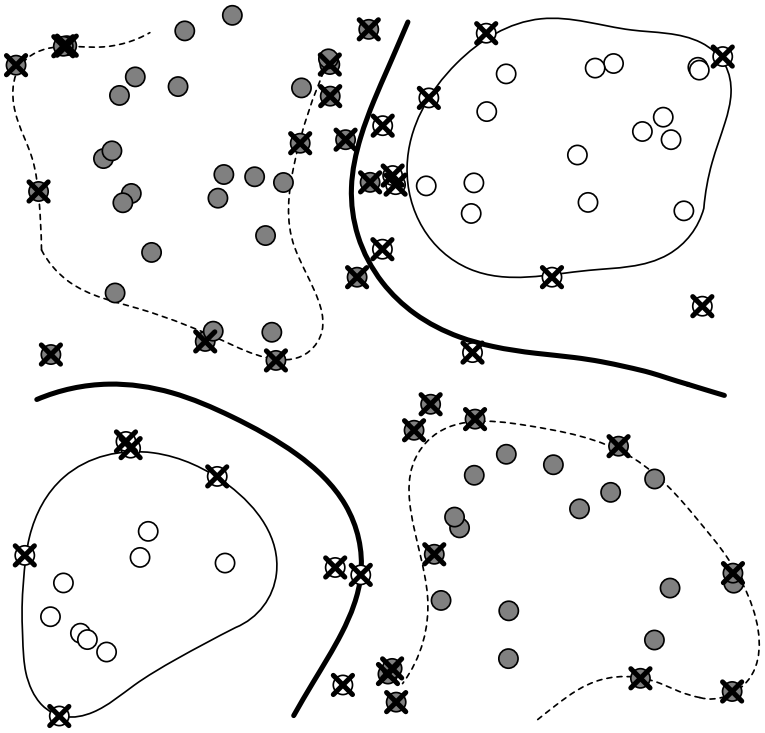


Figure 10.2: Solution of the optimization problem from section 9.1.2 on the corpus from figure 10.1, but with Gaussian kernels. The separating borders $\text{sep}(x) = 0$ is depicted with a bold line and the level set $\text{sep}(x) = 1$ and $\text{sep}(x) = -1$ respectively with the dashed and thin lines. The *support vectors* are marked with crosses.

10.2 Which Functions Are Kernels?

Obviously, it would be too easy if any function of two vectors were a kernel. For a function to be a kernel, there must exist a projection into a feature space such that the function output the same result as the dot product of the projected vectors. Although we don't have to instantiate this projection (as seen before), we have to make sure it exists. The convergence of the SVM is at stake, because this convergence is only ensured when the problem is convex as mentioned in paragraph 9.2.1, and that requires the kernel not to be any random function.

10.2.1 A Simple Example

We consider now the following function to compare two vectors:

$$\mathrm{k}(x, x') = (x.x' + c)^2$$

Is it a kernel? If so, what is the corresponding projection? One way to prove it, is to exhibit the dot product of the projected vectors.

$$\begin{aligned}
(x.x' + c)^2 &= \left(\sum_i \mathbf{x}^i \mathbf{x}'^i + c\right)^2 \\
&= \sum_{i,j} \mathbf{x}^i \mathbf{x}'^i \mathbf{x}^j \mathbf{x}'^j + 2c \sum_i \mathbf{x}^i \mathbf{x}'^i + c^2 \\
&= \sum_{i,j} (\mathbf{x}^i \mathbf{x}^j)(\mathbf{x}'^i \mathbf{x}'^j) + \sum_i (\sqrt{2c}\mathbf{x}^i)(\sqrt{2c}\mathbf{x}'^i) + (c)(c)
\end{aligned}$$

Hence, we can deduce that the projection into a space where our kernel is a dot product, is the

function that combines 2 by 2 the components of the vector x:

$$\varphi(x) = \begin{pmatrix} \left(\mathbf{x}^1\right)^2 \\ \mathbf{x}^1\mathbf{x}^2 \\ \mathbf{x}^1\mathbf{x}^3 \\ \cdots \\ \mathbf{x}^n\mathbf{x}^{n-1} \\ \left(\mathbf{x}^n\right)^2 \\ \sqrt{2c}.\mathbf{x}^1 \\ \sqrt{2c}.\mathbf{x}^2 \\ \cdots \\ \sqrt{2c}.\mathbf{x}^n \\ c \end{pmatrix}$$

10.2.2 Conditions for a kernel

There exists mathematical conditions, named *Mercer's theorem*, that will determine whether a function is a kernel or not, without having to build the projection into the feature space. In practice, we have to ensure that for all sample set with length N, the matrix $\left(\mathrm{k}\left(\mathbf{x}^i, \mathbf{x}^j\right)\right)_{1 \leq i,j \leq N}$ is positive definite[2]. We will not delve further into these conditions because in most cases, we will rather build kernels from existing kernels.

10.2.3 Reference kernels

In this section, we describe two commonly used kernels. The first one is the the polynomial kernel:

$$\mathrm{k}_d(x, x') = (x.x' + c)^d$$

It corresponds to a projection $\Phi(x)$ in a feature space where each component $\phi_i(x)$ is a product of components of x with a degree lower than d (a monomial). The separator computed from this kernel is a polynomial with degree d, whose terms are components of x. The larger the constant c, the more importance is given to the high-order terms. With $c = 1$ and $d = 3$, figure 10.3 depicts the result of the separation.

The second kernel we will introduce here is the Gaussian kernel, also known as RBF[3], mentionned earlier:

$$\mathrm{k}_{\mathrm{rbf}}(x, x') = \exp\left(-\frac{|x - x'|^2}{2\sigma}\right)$$

This kernel corresponds to a projection into an infinite dimension space. However, in this space, *all the points are projected on the hypersphere with radius* 1. This can be easily seen from $|\varphi(x)|^2 = \mathrm{k}(x, x) = \exp(0) = 1$.

10.2.4 Assembling kernels

Once we know some kernels, here are some results leading to the definition of other kernels.
Let:

- k_1, k_2, k_3 three kernel functions.
- f a real valued function.
- Φ a function that project the vectors in another vector space.
- B a semi-definite positive matrix.
- p a polynomial with positive coefficients.

[2] i.e. all its eigenvalues are strictly positive.
[3] Radial Basis Function, a name coming from RBF neural networks which are generalized by SVMs using this kernel.

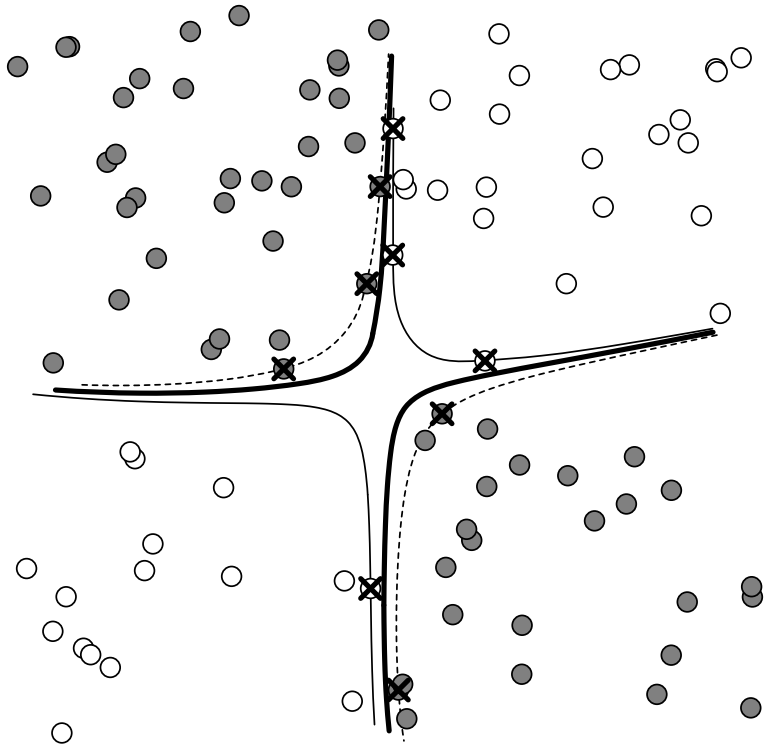


Figure 10.3: Solution of the optimization problem of section 9.1.2 over the corpus from figure 10.1, but with a polynomial kernel with degree 3. The separating borders $\text{sep}(x) = 0$ is depicted with a bold line and the level set $\text{sep}(x) = 1$ and $\text{sep}(x) = -1$ respectively with the dashed and thin lines. The *support vectors* are marked with crosses.

- α a positive number.

then, the following function k are also kernels:

$$\mathrm{k}(x, x') = \mathrm{k}_1(x, x') + \mathrm{k}_2(x, x')$$

$$\mathrm{k}(x, x') = \alpha \mathrm{k}_1(x, x')$$

$$\mathrm{k}(x, x') = \mathrm{k}_1(x, x')\,\mathrm{k}_2(x, x')$$

$$\mathrm{k}(x, x') = f(x)\, f(x')$$

$$\mathrm{k}(x, x') = \mathrm{k}_3(\Phi(x), \Phi(x'))$$

$$\mathrm{k}(x, x') = x^{\mathrm{T}} B x'$$

$$\mathrm{k}(x, x') = p(\mathrm{k}_1(x, x'))$$

$$\mathrm{k}(x, x') = \exp(\mathrm{k}_1(x, x'))$$

10.3 The core idea for SVM

Now that we have added the *kernel trick* to our tool-box, we can work with very high dimension spaces, without having to really enter them. However, a linear separator – and also a linear regression – is actually simplified by the projection of the samples in very high dimension spaces... In return, such a easy-to-build separation is usually meaningless. In other words, it is easy to learn by heart, that is to learn a function that will not generalize to new samples. This is what is usually referred to as "curse of dimensionality". By maximizing the margin, SVMs are still efficient in these high-dimension space and are not effected by this curse. By projecting into the feature space to use a algorithm maximizing the margin, we obtain a good separability while keeping good generalization capabilities. This is the core idea of SVMs.

10.4 Some Kernel Tricks

The ability to compute a dot product between vectors is sufficient to implement more operations than could be expected, and these results can be obtained without having to explicitly project the vectors into the feature space. Here are some examples.

10.4.1 Data Normalization

The norm of a feature vector is given by:

$$|\varphi(x)| = \sqrt{\varphi(x).\varphi(x)} = \sqrt{\mathrm{k}(x,x)}$$

Hence, we can very easily work on normalized data... in the feature space! In practice, the dot product is:

$$\frac{\varphi(x)}{|\varphi(x)|}.\frac{\varphi(x')}{|\varphi(x')|} = \frac{\varphi(x).\varphi(x')}{|\varphi(x)||\varphi(x')|} = \frac{\mathrm{k}(x,x')}{\sqrt{\mathrm{k}(x,x)\,\mathrm{k}(x',x')}}$$

So, we just need to use the right side of the above expression as a new kernel, built upon a kernel k, to work with normalized vectors in the feature space corresponding to k. Denoting $\bar{\mathrm{k}}$ the normalized kernel, we simply have:

$$\bar{\mathrm{k}}(x,x') = \frac{\mathrm{k}(x,x')}{\sqrt{\mathrm{k}(x,x)\,\mathrm{k}(x',x')}}$$

We can even quite easily compute distance, in feature space, between the projections of two vectors:

$$|\varphi(x) - \varphi(x')| = \sqrt{\mathrm{k}(x,x) - 2\mathrm{k}(x,x') + \mathrm{k}(x',x')}$$

10.4.2 Centering and Reduction

For some algorithms[4], it is more efficient to center (substract the center of gravity) and reduce the samples (divide by their variance). This is something we can achieve in the feature space as well. As a reminder, in the following N is the number of samples in our training set.

Let's start by computing the sample variance, which will let us reduce them should we want to.

$$\mathrm{var} = \frac{1}{N}\sum_{i=1}^{N}\left|\varphi(x_i) - \frac{1}{N}\sum_{j=1}^{N}\varphi(x_j)\right|^2 = \cdots = \frac{1}{N}\sum_{i=1}^{N}\mathrm{k}(x_i,x_i) - \frac{1}{N^2}\sum_{i,j=1}^{N}\mathrm{k}(x_i,x_j)$$

To work on centered and reduced samples, we use the kernel $\hat{\mathrm{k}}$ defined as follows:

$$\begin{aligned}\hat{\mathrm{k}}(x,x') &= \frac{\varphi(x) - \frac{1}{N}\sum_{j=1}^{N}\varphi(x_j)}{\sqrt{\mathrm{var}}}.\frac{\varphi(x') - \frac{1}{N}\sum_{j=1}^{N}\varphi(x_j)}{\sqrt{\mathrm{var}}} \\ &= \frac{1}{\mathrm{var}}\left(\mathrm{k}(x,x') - \frac{1}{N}\sum_{i=1}^{N}\mathrm{k}(x,x_i) - \frac{1}{N}\sum_{i=1}^{N}\mathrm{k}(x',x_i) + \frac{1}{N^2}\sum_{i,j=1}^{N}\mathrm{k}(x_i,x_j)\right)\end{aligned}$$

Note that these kernels are very computationally expensive, even in comparison with SVMs which tend to be computationally heavy algorithms with simple kernels. This is the type of situation where one would rather pre-compute and store the kernel values for all the sample pairs in the data-base.

[4] SVMs are not the only one to take advantage of the kernel trick

10.5 Kernels for Structured Data

In practice, as soon as we have data in a vector space, their dot product is a kernel function. So, one way to use SVMs is to express the sample as "vectors". In the situations we are considering, these vectors live in a very high dimension space... It is not necessarily practical, even if possible, to project them into a feature space. Nevertheless, the ability of SVMs to maintain their generalization power even in high-dimension spaces, where samples tend to be "far apart" is of essential importance in this section.

All the approaches alluded to here come from Shawe-Taylor and Cristanini (2004), where many other techniques are also presented.

10.5.1 Document Analysis

One way to process documents with statistical methods is to consider them as *bags of words*. To this end, we define a dictionary of N words $m_1, \cdots, m_N$, as well as the following function φ, mapping a document d to a vector $\varphi(d)$:

$$\begin{array}{rccl} \varphi : & \text{Documents} & \to & \mathbb{N}^N \\ & d & \mapsto & \varphi(d) = (f(m_1, d), \cdots, f(m_N, d)) \end{array}$$

where $f(m, d)$ is the number of occurences of word m in document d. For a set $\{d_l\}_l$ of l documents, the document term matrix D, where line i is given by vector $\varphi(d_i)$, let us define a dot product (hence a kernel) over the documents. In practice, $\mathrm{k}(d_i, d_j)$ is given by the coefficient (i, j) of $DD^\intercal$. We can mitigate the fact that documents will have different length by using a normalized kernel.

Furthermore, we can tune this kernel by injecting some a-priori semantic knowledge. For instance, we can define a diagonal matrix R where each diagonal value corresponds to the importance of a given word. We can also define a matrix P of semantic proximity for which coefficient $p_{i,j}$ represents the semantic proximity of words m_i and m_j. The semantic matrix $S = RP$ let us create a kernel taking advantage of this knowledge:

$$\mathrm{k}(d_i, d_j) = \varphi(d_i) S S^\intercal \varphi(d_j)$$

10.5.2 Strings

Character strings have received a lot of attention in computer science, and many approaches have been designed to quantify the similarity between two strings. In particular, SVMs are one of the machine learning techniques used to process DNA sequences where string similarity is essential. This section provide an example of such function.

Let's consider the case of the *p-spectrum kernel*. We intend to compare two strings, probably of different length, by using their common sub-strings of length p. Let Σ be an alphabet, we denote Σ^p the set of strings of length p built on Σ and $s_1 s_2$ the concatenation of s_1 and s_2. We also note $|A|$ the number of element in a set A. We can then define the following expression, for $u \in \Sigma^p$:

$$\varphi_u^p(s) = |\{(v_1, v_2) \ : s = v_1 u v_2\}|$$

For a string s, we get one $\varphi_u^p(s)$ per possible sub-string u. $\varphi_u^p(s)$ is zero for most $u \in \Sigma^p$. Thus, we are projecting a string s on a vector space with $|\Sigma|^p$ dimensions and the components of the vector $\varphi^p(s)$ are the $\varphi_u^p(s)$. We can finally define a kernel with a simple dot product:

$$\mathrm{k}(s, t) = \varphi^p(s).\varphi^p(t) = \sum_{u \in \Sigma^p} \varphi_u^p(s) \varphi_u^p(t)$$

Let's explicit this kernel with $p = 3$ and the following strings: `bateau`, `rateau`, `oiseau`, `croise ciseaux`. The elements of Σ^3 leading to non-zero components are `ate`, `aux`, `bat`, `cis`, `cro`, `eau`, `ise`, `ois`, `rat`, `roi`, `sea`, `tea`. The lines in table 10.1 are the non-zero components of the vectors $\varphi^p(s)$.

We can then represent as a matrix the values of the kernel for every pair of words, as shown in table 10.2.;

	ate	aux	bat	cis	cro	eau	ise	ois	rat	roi	sea	tea
bateau	1		1			1						1
rateau	1					1			1			1
oiseau						1	1	1			1	
croise					1		1	1		1		
ciseaux		1		1		1	1				1	

Table 10.1: *3-spectrum* of words bateau, rateau, oiseau, croise ciseaux.

k	bateau	rateau	oiseau	croise	ciseaux
bateau	4	3	1	0	1
rateau	3	4	1	0	1
oiseau	1	1	4	2	3
croise	0	0	2	4	1
ciseaux	1	1	3	1	5

Table 10.2: *3-spectrum* des mots bateau, rateau, oiseau, croise ciseaux.

10.5.3 Other Examples

We can also design recursive kernels, which will let us define the "dot product" of two structured objects (e.g. symbolic objects), such as graphs or trees. The subtlety and the difficulty consist in producing a number out of two structure objects while making sure that the function evaluating this number respects Mercer's property.

Chapter 11

Solving SVMs

11.1 Quadratic Optimization Problems with SMO

11.1.1 General Principle

There are several methods to solve the optimization problem defined in section 9.1.2. One of them is the *SMO*[1] algorithm. Even for this approach, several refinement can be found in the litterature.

The core idea is to start from the KKT conditions of the dual problem defined page 108. We need to move in the α_i space in order to maximize the objective function. The SMO algorithm consist in moving by changing pairs of α_i. Equation L2, presented in 9.2.5, shows that if we keep all α_i but two constant, the two we can change are linearly dependent: we can express one of them as a linear function of the other. This let us rewrite the objective function as a function of only a single α_i, and to compute its gradient.

The algorithm stops when some conditions, characterizing the optimality, are satisfied.

In practice, this technique is rich in algorithmic tricks to optimally chose the pairs of α_i and increase the convergence rate. As such it tends to be rather difficult to implement. In the following, we're presenting the technique presented in Keerthi et al. (1999), which is an improvement on the initial SMO algorithm presented in Platt (1998).

11.1.2 Optimality Detection

To detect the optimality, the idea is to start again from the dual problem defined page 108. This a minimization problem (we're multiplying the objective function by -1 for that) under three constraints (the second one having two inequalities). Even if this problem is already the result of Lagrangian resolution, we can use the Lagrangian technique again to solve it... but we're not running in circle, because this Lagrangian will help us express optimality conditions that will become the stopping criteria for an algorithm. And this algorithm solves directly the dual problem defined page 108. So, this Lagrangian is[2]:

$$\mathcal{L}(\alpha, \delta, \mu, \beta) = \frac{1}{2}\sum_{j}^{N}\sum_{i}^{N}\alpha_j\alpha_i y_j y_i x_j.x_i - \sum_{i}^{N}\alpha_i - \sum_{i}^{N}\delta_i\alpha_i + \sum_{i}^{N}\mu_i(\alpha_i - C) - \beta\sum_{i}^{N}\alpha_i y_i$$

For the sake of notation simplicity, we define:

$$F_i = \sum_{j}^{N}\alpha_i y_i x_i.x_j - y_i$$

We can then express the KKT conditions, that is to say the zeroing of the the partial derivatives of the Lagrangian. We can also express the additional KKT conditions which are that when a multiplier is zero, then the constraint is not saturated and when it is non-zero, then the constraint

[1] Sequential Minimal Optimization

[2] Be careful, the α_i are now primal parameters and the δ_i, μ_i and the parameter β are the Lagrange multipliers.

is saturated. Thus, the product of the constraint and its multiplier is zero at the optimum without the two factors being zero at the same time (see page 108 for a reminder). The multipliers are also all positives.

$$\forall\, 1 \leq i \leq N,\ \begin{cases} \dfrac{\partial}{\partial \alpha_i}\mathcal{L} = (F_i - \beta)y_i - \delta_i + \mu_i &= 0 \\ \delta_i \alpha_i &= 0 \\ \mu_i(\alpha_i - C) &= 0 \end{cases}$$

These conditions get simpler when they are written in the following way, distinguishing three cases according to the value of α_i.

Case $\alpha_i = 0$ **:** So $\delta_i > 0$ and $\mu_i = 0$, thus

$$(F_i - \beta)y_i \geq 0$$

Case $0 < \alpha_i < C$ **:** So $\delta_i = 0$ and $\mu_i = 0$, thus

$$(F_i - \beta)y_i = 0$$

Case $\alpha_i = C$ **:** So $\delta_i = 0$ and $\mu_i > 0$, thus

$$(F_i - \beta)y_i \leq 0$$

Since $y_i \in \{-1, 1\}$, we can separate the value of i according to the sign of $F_i - \beta$. This let us define the following sets of indices:

$$\begin{aligned} I_0 &= \{i : 0 < \alpha_i < C\} \\ I_1 &= \{i : y_i = 1, \alpha_i = 0\} \\ I_2 &= \{i : y_i = -1, \alpha_i = C\} \\ I_3 &= \{i : y_i = 1, \alpha_i = C\} \\ I_4 &= \{i : y_i = -1, \alpha_i = 0\} \\ I_{\text{sup}} &= I_0 \cup I_1 \cup I_2 \\ I_{\text{inf}} &= I_0 \cup I_3 \cup I_4 \end{aligned}$$

Then:

$$\begin{aligned} i \in I_{\text{sup}} &\Rightarrow \beta \leq F_i \\ i \in I_{\text{inf}} &\Rightarrow \beta \geq F_i \end{aligned}$$

We can then define the following bounds on these set:

$$\begin{aligned} b_{\text{sup}} &= \min_{i \in I_{\text{sup}}} F_i \\ b_{\text{inf}} &= \max_{i \in I_{\text{inf}}} F_i \end{aligned}$$

In practice, when we will iterate the algorithm, we will have $b_{\text{sup}} \leq b_{\text{inf}}$ as long as we haven't reached the optimum, but at the optimum:

$$b_{\text{inf}} \leq b_{\text{sup}}$$

We can reverse this condition, and say that we haven't reached the optimum as long as we can find two indices, one in I_{sup} and the other in I_{inf}, which violate the condition $b_{\text{inf}} \leq b_{\text{sup}}$. Such an indice pair defines a *violation* of the optimality conditions:

$$(i, j) \text{ such that } i \in I_{\text{sup}} \text{ and } j \in I_{\text{inf}} \text{ is a violation if } F_i < F_j \tag{11.1}$$

Equation (11.1) is theoretic, since on a real computer, we will never have, numerically, $b_{\text{inf}} \leq b_{\text{sup}}$ at the optimum. We will satisfy ourselves with defining this conditions "up to a bit", say $\tau > 0$. In other words, the approximative optimality condition is:

$$b_{\text{inf}} \leq b_{\text{sup}} + \tau \tag{11.2}$$

Equation (11.1), which defines when a pair of indices violates the optimality condition, is then modified accordingly:

$$(i,j) \text{ such that } i \in I_{\text{sup}} \text{ et } j \in I_{\text{inf}} \text{ is a violation if } F_i < F_j - \tau \tag{11.3}$$

The criterion (11.3) will be tested to check whether we need to keep running the optimization algorithm, or if can consider that the optimum has been reached.

Before terminating this paragraph, finally aimed at presenting the stopping criteria of the algorithm we will define in the following sections, let's point out that, at the optimum, $b_{\text{sup}} \approx b_{\text{inf}} \approx \beta$... and that this value is also the b of our separator! In section 9.2.5 page 109, we lamented that a closed-form solution for b was, until now, not made available by the Lagrangian solution.

11.1.3 Optimisation Algorithm

The principle of the SMO optimization starts by noting that equation L2 (cf. 9.2.5) links the α_i and that their sum, weighted by the y_i, is constant. So, if we only allow two of these coefficients α_{i_1} and α_{i_2} to changes (in $[0,C]^2$), keeping the other constant, we have: $L2 \Rightarrow \alpha_{i_1} y_{i_1} + \alpha_{i_2} y_{i_2} + \text{const} = 0$. With $\alpha_{i_1} = (-\text{const} - \alpha_{i_2} y_{i_2})/y_{i_1}$, we can write the objective function of the dual problem (cf. page 108) as a function of α_{i_1} only. We can then compute the point $\alpha_{i_1}^\star$ for which this function is maximal[3]. We then update $\alpha_{i_1} \leftarrow \alpha_{i_1}^\star$ and we can obtain α_{i_2}. It is nevertheless necessary to make sure to control these variations so as to make sure $(\alpha_{i_1}, \alpha_{i_2})$ stays in $[0,C]^2$. Then, we select another pair of coefficient to update, etc.

So, most of the problem is how to chose the pairs (i_1, i_2). This is a thorny problem, whose solution influences how quickly we reach the maximum. The solution proposed in Keerthi et al. (1999) consist in considering all i_1, to find out whether $i_1 \in I_{\text{sup}}$ or $i_1 \in I_{\text{inf}}$. For the selected i_1, i_2 will be the index belonging to the other set, for which the bound b_{sup} or b_{inf}, according to which set contains i_1, is reached. By this way, we work on the pair that violates the most the optimality constraints. More refinements can be found in detail in the pseudo-code presented in Keerthi et al. (1999).

Numerically, the update of a pair can lead to negligible changes of the pair $(\alpha_{i_1}, \alpha_{i_2})$, say smaller in absolute value than a threshold θ. If all the updates generated by a traversal of the samples are negligible, then we can stop the algorithm (stopping case #1). Furthermore, when we find out, via equation (11.2), that the optimality is reached, then we also stop the algorithm (stopping case #2).

11.1.4 Numerical Problems

Stopping case #1 depends on the value of θ, whereas case #2 depends on τ. But it is completely possible to stop du to case #1, without having reached optimality... This often occurs, for instance, when the optimization is playing on the 15th decimal.

To mitigate this, we need to choose the parameters τ and θ. A good thing to check when the algorithm stops, is the ratio of pairs of coefficients which violate the optimality criterion. When this ratio is high, we need to reduce θ. When, on the other hand, the solution seems obviously wrong whereas none of the pair still violates the optimality criterion, then the value of τ is probably too high.

11.2 Parameter Tuning

The choice of parameters best fitted to a given problem is far from intuitive. In the case of the classification problem we have seen until now, the parameter C needs to be determined. Similarly, when we work, for instance, with Gaussian kernels, the parameter σ must be included as well.

To this end, a brute-force approach consist in trying many values of the parameters[4], and to measure the generalization error of the SVM with cross-validation. Finally, we will select the parameters for which the generalization error is minimal.

[3] It is a quadratic function for which we just need to zero the derivative.

[4] *Grid search method*

Chapter 12

Regression

Until now, we only considered the problem of separating a corpus of samples in two classes, according to their labels -1 or $+1$. Regression consist in using labels with values in $\mathbb{R}$ and to search for a function that will map a vector to its label, based on the samples in the corpus.

12.1 Definition of the Optimization Problem

Similarly to previous sections, we will start by considering the case of a linear regression and we will later generalize to other regression by replacing dot product with kernels.

The regression we are presenting here takes its root in the fact that a linear separator $h_{w,b}$ is good when it fits well all the samples, up to an error $\epsilon > 0$. In other words:

$$\forall i,\ |(w.x_i + b) - y_i| \leq \epsilon$$

Clearly, this constraint is to strong in general and in practice we will optimize an objective function that authorize some of the samples to violate the constraint. This is depicted in figure 12.1.

In a process similar to that of a linear separator, we reach the definition of the following optimization problem for a regression:

$$\begin{array}{rl} \text{find } \underset{w,b,\xi,\xi'}{\text{argmin}} & \frac{1}{2}w.w + C\sum_i^N(\xi_i + \xi'_i) \\ \text{subject to} & \begin{cases} y_i - w.x_i - b \leq \epsilon + \xi_i,\ \forall (x_i, y_i) \in S \\ w.x_i + b - y_i \leq \epsilon + \xi'_i,\ \forall (x_i, y_i) \in S \\ \xi_i, \xi'_i \geq 0,\ \forall i \end{cases} \end{array}$$

12.2 Resolution

Solving this optimization problem is once again easier after switching to the dual problem, as was the case for linear separator in chapter 9. Let α_i and α'_i the multipliers for the two first constraints of our optimization problem. The vector w of the separator is given by:

$$w_{\alpha,\alpha'} = \sum_i^N(\alpha_i - \alpha'_i)x_i$$

with α_i and α'_i solution of the following dual problem:

$$\begin{array}{rl} \text{find } \underset{\alpha,\alpha'}{\text{argmax}} & \sum_i^N y_i(\alpha_i - \alpha'_i) - \epsilon\sum_i^N(\alpha_i + \alpha'_i) - \frac{1}{2}w_{\alpha,\alpha'}.w_{\alpha,\alpha'} \\ \text{subject to} & \begin{cases} \sum_i^N(\alpha_i - \alpha'_i) = 0 \\ \alpha_i, \alpha'_i \in [0, C],\ \forall i \end{cases} \end{array}$$

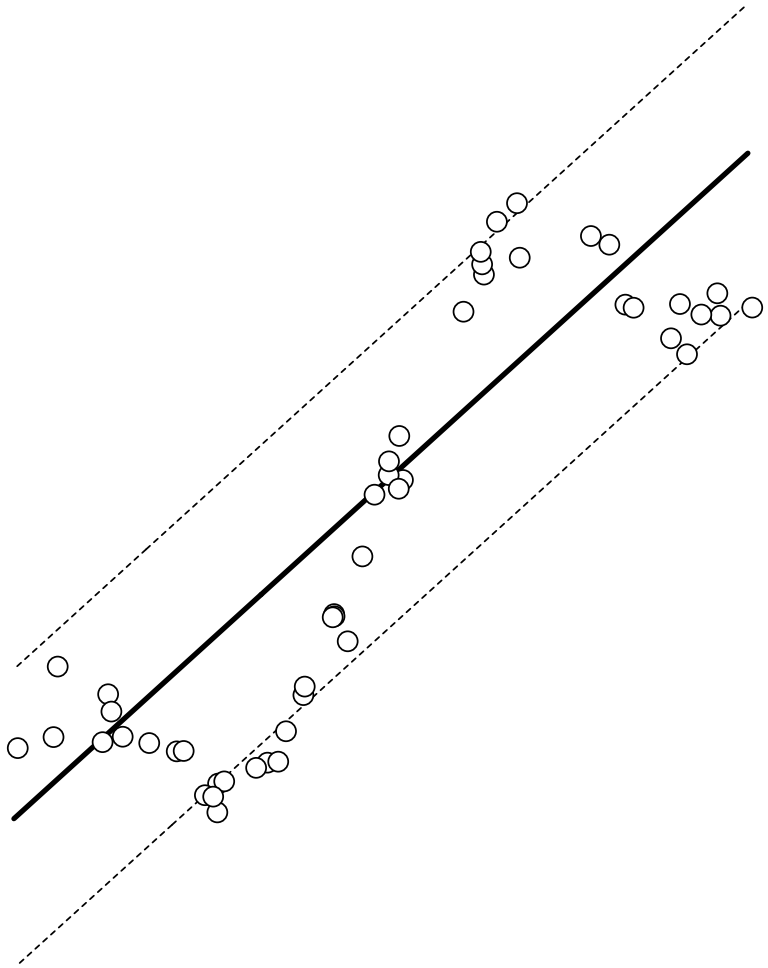


Figure 12.1: Linear regression. The white dots have an abscissa of x_i, one-dimension vector, and for ordinate y_i. The dashed band represents the set of acceptable distance to the separator, $|w.x + b - y| \leq \epsilon$, and not many samples are out of this set.

Once again, it "just" remains to apply an algorithm that will search for the maximum of this dual problem. Approaches similar to the SMO algorithm exist but lead to algorithm relatively hard to implement.

12.3 Examples

First, using a linear kernel, it has to be notices that the condition $\alpha_i, \alpha'_i \in [0, C]$ limits the value taken by the computed predictor, since for $w_{\alpha,\alpha'}$, we have the coeffcients $\alpha_i - \alpha'_i \in [-C, C]$. This limit is therefore related to C. The influence of C is illustrated on figure 12.2.

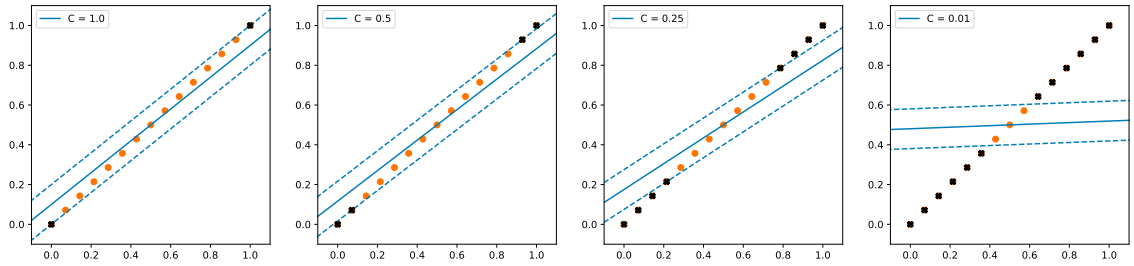


Figure 12.2: The influence of C on an ϵ-SVR. Black crosses are placed on support vectors.

Be careful ! With a Gaussian kernel, the regression formula is $h(x) = \sum_i^N a \exp\left(-\frac{(x_i - x)^2}{2\sigma^2}\right) + b$, *with* $a = \alpha_i - \alpha'_i$, *so* $-C \leq a \leq C$. *As each Gaussien is at most 1, each term of the sum is at most in* $[-C, C]$. *If the Gaussian do not overlap much, only few terms of the sum (let say at most k terms) are significantly non null for any* x. *So* $-kC \leq h(x) \leq kC$ *reasonably stands. You may have thus to select* C *and* σ *(k depends on* σ *and how your samples are distributed) such has the* $[-kC, kC]$ *contains the values you want to predict.*

Figure 12.3 demonstrates the use of this type of SVM for regression in the case of 1D vectors, for different kernels. Figure 12.4 gives an example in the case of 2D vectors.

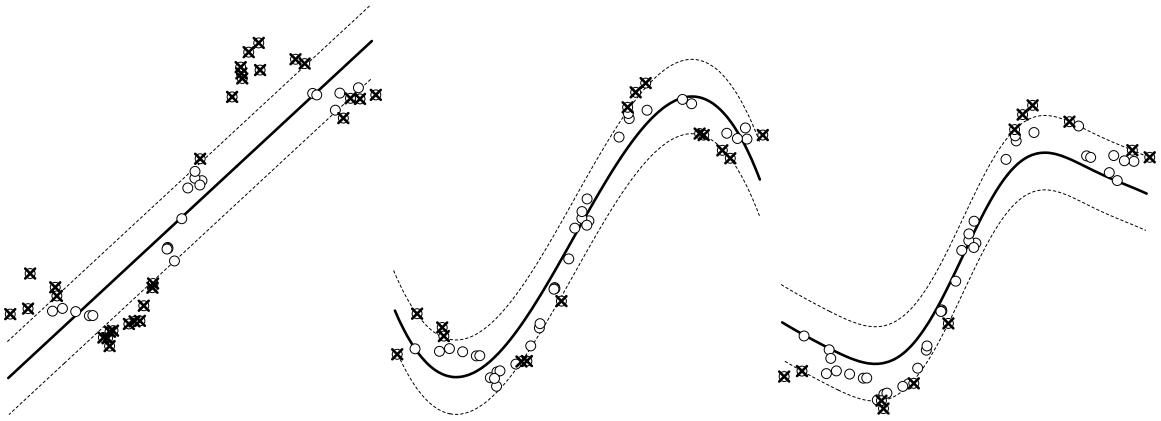


Figure 12.3: Regression on 1D vectors, similar to figure 12.1. Left: using a standard dot product. Middle: using a 3rd degree polynomial kernel. Right: using a Gaussian kernel. The support vectors are marked with a cross. They are vectors x_i for which a pair of (α_i, α'_i) is non zero. They are the one constraining the position of the regression curve. The tolerance $-\epsilon, +\epsilon$ is represented with dashed lines.

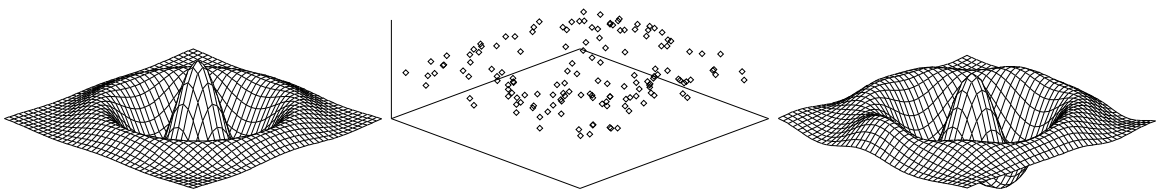


Figure 12.4: Left: the function $z = f(x, y) = exp(-2.5(x^2 + y^2) * cos(8 * \sqrt{x^2 + y^2})$ that we used to generate the samples. Middle: 150 samples, obtained by randomly drawing x and y and defining vector $x_i = (x, y)$ with a label $y_i = f(x, y) + \nu$, with ν drawn uniformly from $[-0.1, 0.1]$. Right: result of the regression on these samples, with a Gaussian kernel with variance $\sigma = 0.25$ and a tolerance $\epsilon = 0.05$.

Chapter 13

Compedium of SVMs

Ultimately, the principle of the approaches seen so far is always the same: we define a quadratic optimization problem that can be solved using only dot products of pairs of samples.

13.1 Classification

These approaches are called *SVC* for *Support Vector Classification*.

13.1.1 C-SVC

The *C-SVC* the approach we've seen so far. The optimization problem is only given here as a reminder:

$$\begin{aligned} \text{find } \underset{w,b,\xi}{\text{argmin}} \quad & \frac{1}{2} w.w + C \sum_i \xi_i \\ \text{subject to} \quad & \begin{cases} y_i \left(w.x_i + b \right) \geq 1 - \xi_i, \ \forall i \\ \xi_i \geq 0, \ \forall i \end{cases} \end{aligned}$$

13.1.2 ν-SVC

The problem of a $C-SVM$ is that C, which define when to use *slack variables* ξ_i, does not depend on the number of samples. In some cases, we might want to define the number of support vectors based on the number of samples instead of giving an absolute value. The parameter $\nu \in]0, 1]$ is linked to the ratio of examples that can be used as support vectors[1]. In a C-SVM, we always force the samples to be located outside the band $[-1, 1]$. Here, we chose a band $[-\rho, \rho]$, and we adjust ρ to obtain the desired ratio of support vectors. This defines a *ν-SVC* problem.

$$\begin{aligned} \text{find } \underset{w,b,\xi,\rho}{\text{argmin}} \quad & \frac{1}{2} w.w - \nu\rho + \frac{1}{N} \sum_i \xi_i \\ \text{subject to} \quad & \begin{cases} y_i \left(w.x_i + b \right) \geq \rho - \xi_i, \ \forall i \\ \xi_i \geq 0, \ \forall i \\ \rho \geq 0 \end{cases} \end{aligned}$$

Expressing the objective function however is not so simple and how will ν define the ratio of samples used as support vectors is far from obvious. This can be justified by looking at the KKT conditions of this optimization problem Schölkopf et al. (2000).

13.2 Regression

These approaches are named *SVR* for *Support Vector Regression*.

[1] This ratio tends towards ν when we have many samples.

13.2.1 ϵ-SVR

The *ϵ-SVR* is the approach we presented earlier. The optimization problem is given here as a reminder.

$$\begin{aligned} \text{find} \underset{w,b,\xi,\xi'}{\text{argmin}} \quad & \frac{1}{2} w.w + C \sum_i (\xi_i + \xi'_i) \\ \text{subject to} \quad & \begin{cases} y_i - w.x_i - b \leq \epsilon + \xi_i, \ \forall i \\ w.x_i + b - y_i \leq \epsilon + \xi'_i, \ \forall i \\ \xi_i, \xi'_i \geq 0, \ \forall i \end{cases} \end{aligned}$$

13.2.2 ν-SVR

Similarly to the ν-SVC, the purpose here is to modulate the width ϵ of the ϵ-SVR according to a parameter $\nu \in]0, 1]$. The objective is to define the number of samples outside of a tube with radius ϵ around the regression function as a ratio ν of the total number of samples[2].

$$\begin{aligned} \text{find} \underset{w,b,\xi,\xi',\epsilon}{\text{argmin}} \quad & \frac{1}{2} w.w + C(\nu\epsilon + \frac{1}{N} \sum_i (\xi_i + \xi'_i)) \\ \text{subject to} \quad & \begin{cases} y_i - w.x_i - b \leq \epsilon + \xi_i, \ \forall i \\ w.x_i + b - y_i \leq \epsilon + \xi'_i, \ \forall i \\ \xi_i, \xi'_i \geq 0, \ \forall i \\ \epsilon \geq 0 \end{cases} \end{aligned}$$

As was the case for ν-SVC, the justification of this *ν-SVR* formulation of the optimization problem can be found in Schölkopf et al. (2000) and is a consequence of the KKT conditions resulting from this problem.

13.3 Unsupervized Learning

SVM can also be used to perform unsupervized machine learning (distribution analysis). In this case, we seeking to build a novelty detector. We ask the SVM to describe the samples as clusters of points. Once these clusters have been defined, any sample falling outside of any cluster would be deemed not to have been generated by the same distribution as the others. This different sample is then the result of a new phenomenon. This particularly practical when we have many examples of our phenomenon, but not many counter-examples. For instance, in the case of the analysis of an EEG signal to predict epileptic crisis, the data-set mostly describe states that are not forebearing a crisis, because crisis are, thankfully, much less frequent than the normal state. One way to detect a crisis is to observe when a signal falls outside of the *perimeter* of the normal signals.

13.3.1 Minimal enclosing sphere

The idea behind these techniques is to include the samples $x_1, \cdots, x_i$ in minimal-radius sphere, called the *minimal enclosing sphere*. Obviously, if we're using a strict constraint, the first noisy sample far from the other will prevent the sphere from sticking to the bulk of the data-set. In practice, we might want to search for the smallest sphere containing $\alpha\%$ of the data. This problem is NP-complete...

We can again use the trick of the *slack variables* that can give an approximate solution to the full NP-complete problem:

$$\begin{aligned} \text{find} \underset{\omega,r,\xi}{\text{argmin}} \quad & r^2 + C \sum_i \xi_i \\ \text{subject to} \quad & \begin{cases} |x_i - \omega|^2 \leq r^2 + \xi_i, \ \forall i \\ \xi_i \geq 0, \ \forall i \end{cases} \end{aligned}$$

Using the Langrangian solution, we obtain a formula giving r and an indicator function whose value is 1 outside the sphere, both results fortunately using only dot products (cf. figure 13.1).

[2] At least, in practice, the ratio of samples outside a tube with radius ϵ around the function tend towards ν when the number of samples is high.

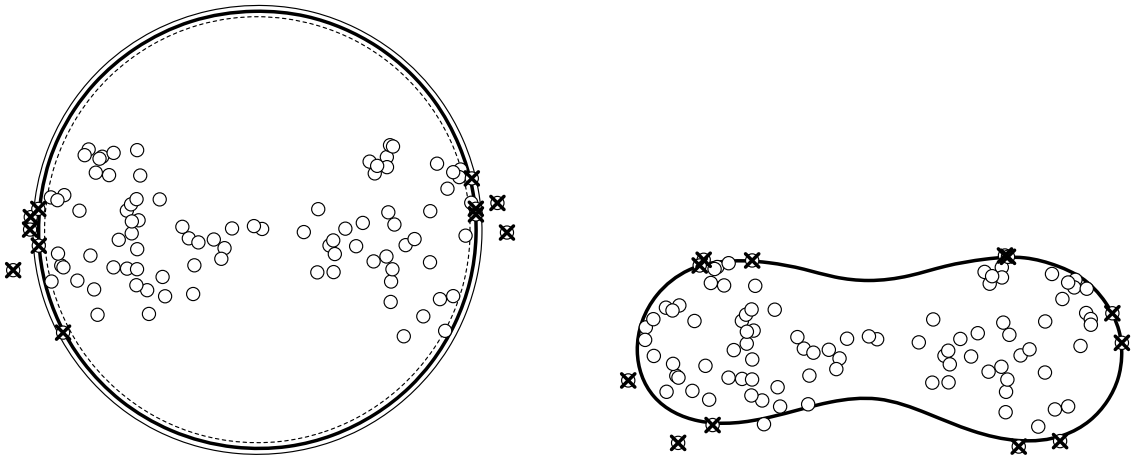


Figure 13.1: Minimal enclosing sphere. Left: using the standard dot product. Right: using a Gaussian kernel with $\sigma = 3$. In both cases, $C = 0.1$ and the samples are randomly drawn in a 10×10 square.

As before, we can find a ν-version of this problem, in order to control the number of support vectors and thus the ratio of samples lying outside the sphere Shawe-Taylor and Cristanini (2004). The optimization problem is the same as before, setting $C = 1/\nu N$ and then multiplying by ν the objective function[3]. Once again, the reason why this value of C leads to ν being effectively linked with the ratio of samples outside the sphere can be found by analysing the KKT conditons.

$$\begin{aligned} \text{find} \underset{\omega,r,\xi}{\text{argmin}} \quad & \nu r^2 + \frac{1}{N}\sum_i \xi_i \\ \text{subject to} \quad & \begin{cases} |x_i - \omega|^2 \leq r^2 + \xi_i, \ \forall i \\ \xi_i \geq 0, \ \forall i \end{cases} \end{aligned}$$

13.3.2 One-class SVM

This case is a bit peculiar, although very much used. The goal is to find an hyperplane satisfying two conditions. The first one is that the origin must lie on the negative side of the plane with the samples all on the positive side, up to the *slack variables*, as usual. The second condition is that this hyperplane must be as far as possible from the origin. In general, the advantage of this setup is not obvious. The samples could set in a spherical pattern around the origin and no such hyperplane would exist. In practice, the *one-class SVM* is relevant for radial kernels such as Gaussian kernels. Remember that these kernels project the vectors onto an hypersphere centered n the origin (see paragraph 10.2.3). The origin is thus "naturally" far from the samples for this type of kernels and the samples clustered on a part of the hypersphere. We can isolate this part by pushing the hyperplane as far as possible from the origin (cf. figure 13.2).

$$\begin{aligned} \text{find} \underset{w,b,\xi,\rho}{\text{argmin}} \quad & \frac{1}{2} w.w - \rho + \frac{1}{\nu N}\sum_i \xi_i \\ \text{subject to} \quad & \begin{cases} w.x_i \geq \rho - \xi_i, \ \forall i \\ \xi_i \geq 0, \ \forall i \end{cases} \end{aligned}$$

Once again, the parameter ν is linked to the ratio of samples on the negative side of the hyperplane. This is justified in Schölkopf et al. (2001) using arguments on the KKT conditions of the problem.

[3] which does not change anything to the result.

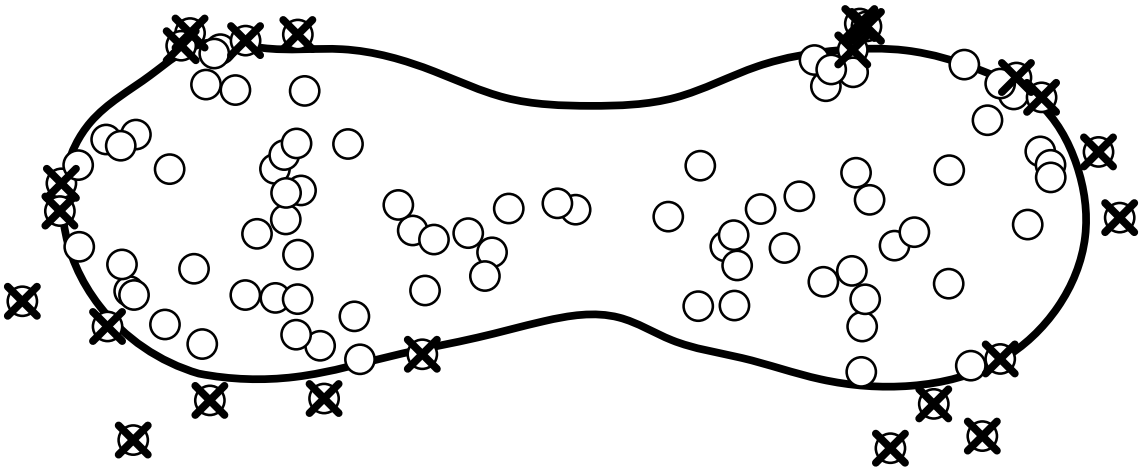


Figure 13.2: One class SVM. The samples are the same as in figure 13.1. We use a Gaussian kernel with $\sigma = 3$ and $\nu = 0.2$, which means that around 20% of the samples are outside the selected region.

Part IV

Vector Quantization

Chapter 14

Introduction and notations for vector quantization

14.1 An unsupervised learning problem

14.1.1 Formalization as a dummy supervised learning problem

The *vector quantization* techniques belong to the class of unsupervised learning methods. A synthetic overview of these methods can be found in (Fritzke, 1997). Here, let us rather introduce vector quantization as a degenerated case of supervised learning, where a specific set of hypotheses $\mathcal{H}$ is used.

Let us consider a random variable X whose values are in $\mathcal{X}$, from which the learning dataset is sampled. There is no labeling here, since only the distribution of inputs is of interest. Let us consider a dummy labeling process, i.e. an oracle, that is the identity function. This oracle is the conditional distribution $\mathrm{P}(y \mid x) = \delta_{x,y}$, and the random variable providing the samples with a supervised learning flavor is thus (X, X). Let us consider loss function $L \in (\mathbb{R}^+)^{\mathcal{X}\times\mathcal{X}}$. The quadratic loss is usually considered in vector quantization, but this is not mandatory.

Let us approximate the oracle (the identity...) with a hypothesis $h \in \mathcal{H} \subset \mathcal{X}^{\mathcal{X}}$. The set of hypothesis $\mathcal{H}$ is taken as the following set of functions

$$\mathcal{H} = \left\{ h_\Omega,\ \Omega \in \mathcal{P}_{\mathbb{N}^\star}(\mathcal{X}) \;\middle|\; h_\Omega(x) = \operatorname*{argmin}_{\omega\in\Omega} L(\omega, x) \right\}$$

where $\mathcal{P}_{\mathbb{N}^\star}(\mathcal{X})$ is the set of finite subsets of $\mathcal{X}$. In other words, h_Ω is defined according to a set of values $\Omega = \{\omega_0, \cdots, \omega_i, \cdots, \omega_K\}$. It computes its output as the ω_i which is the closest to the input. The ω_is are called the *prototypes*. All hypotheses in $\mathcal{H}$ are not required to use the same number of prototypes for their computation.

Let us define the *distortion* induced by a set of prototypes Ω as $\mathcal{R}(h_\Omega)$. It is the expectation, when a sample x is taken according to X, of the error made by assimilating x to its closest prototype in Ω. It can be approximated by an empirical risk $\mathcal{R}^S_{\mathrm{emp}}(h_\Omega)$ measured on a dataset $S = \{x_1, \cdots, x_i, \cdots, x_N\}$, actually viewed here as $S = \{(x_1, x_1), \cdots, (x_i, x_i), \cdots, (x_N, x_N)\}$. This empirical risk is considered here as the distortion induced by h_Ω on the data.

Since we allow for arbitrarily large Ωs in this definition, it is obvious that, when the values of X are bounded, having a huge set Ω of prototypes uniformly spread over the values taken by X enables to have $\mathcal{R}(h_\Omega)$ as small as wanted. Therefore, as opposed to real supervised learning, the goal here is not only to reduce the distortion, but rather to have a minimal distortion when only few prototypes (i.e. $|\Omega| < K$) are allowed. In this case, the prototypes for which this minimal distortion is obtained are "well spread" over the data.

Handling a discrete collection of few prototypes gives the method its name of vector quantization.

14.1.2 Choosing the suitable loss function

In the previous definition, where a dummy supervised learning is used to describe a unsupervised learning problem, the loss function L is crucial.

> *This is the place for adding the semantics of the problem in the algorithms. Even if the vast majority of references in the literature use* $\mathcal{X} = \mathbb{R}^n$ *and* $L(x, x') = (x - x').(x - x')$*, the central role of the loss has to be kept in mind when vector quantization is applied to real problems.*

Let us take the example of handwritten digits recognition where inputs are digits, provided as a gray-scaled 8×8 images. In this case, $\mathcal{X} = [0, 1]^{64}$, where 0 stands for white and 1 for black. Let us consider the three inputs x_1, x_2 and x_3 depicted in the figure 14.1. Using the default loss function mentioned above, the following stands: $L(x_1, x_2) = L(x_2, x_3) = L(x_1, x_3) = 20$. In other words, $x_1 x_2 x_3$ forms an equilateral triangle in $[0, 1]^{64}$. An appropriate design should have lead to $L(x_1, x_2) < L(x_1, x_3)$, since samples x_1 and x_2 look very similar to each other. Figure 14.2 shows the inadequacy of the ℓ_2 norm as well.

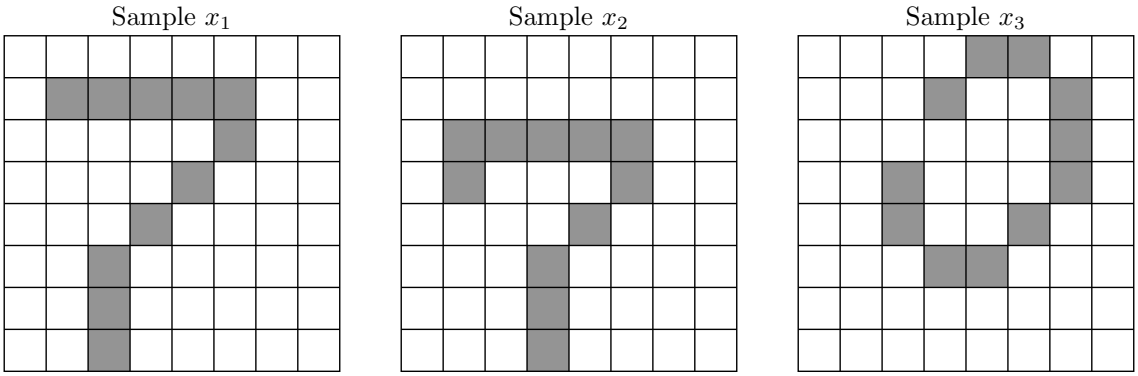


Figure 14.1: Three digit inputs. Each digit is made of 11 black pixels. Each pair of digits is such that the digits have only one black pixel in common.

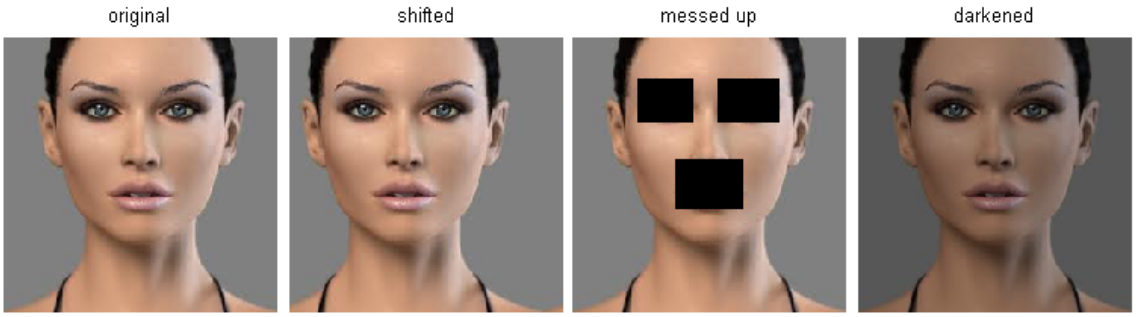


Figure 14.2: The original image is on the left. The three other images are respectively obtained from it by a shift, the adding of black rectangles and a darkening. In each of these three cases, the pixel-wise ℓ_2 distance to the original image is the same, while the visual alteration we experiment as observers is not. The illustration is taken from http://cs231n.github.io/classification/.

14.1.3 Samples

Voronoï subsets

In real cases, the random variable X is unknown. It is supposed to drive the production of the dataset $S = \{x_1, \cdots, x_i, \cdots, x_N\}$. Since h_Ω in $\mathcal{H}$ consists in returning the prototype which is the closest to the given argument, gathering the samples according to the labels given by h_Ω is meaningful. This leads to the definition[1] of *Voronoï subsets* as follows:

$$\begin{aligned} \mathcal{V}_\Omega^S(\omega) &\stackrel{\text{def}}{=} \left\{ x \in S \;\middle|\; \operatorname*{argmin}_{\omega' \in \Omega} L(\omega', x) = \omega \right\} \\ \forall (\omega, \omega') \in \Omega^2,\ \omega \neq \omega' &\Rightarrow \mathcal{V}_\Omega^S(\omega) \cap \mathcal{V}_\Omega^S(\omega') = \emptyset \\ \bigcup_{\omega \in \Omega} \mathcal{V}_\Omega^S(\omega) &= S \end{aligned}$$

As the Voronoï subsets form a partition of S, the empirical distortion $\mathcal{R}_{\text{emp}}^S(h_\Omega)$ can be decomposed on each of them:

$$\begin{aligned} \mathcal{R}_{\text{emp}}^S(h_\Omega) &= \frac{1}{N} \sum_{x \in S} L(x, h_\Omega(x)) \\ &= \frac{1}{N} \sum_{\omega \in \Omega} \sum_{x \in \mathcal{V}_\Omega^S(\omega)} L(x, \omega) \\ &= \frac{1}{N} \sum_{\omega \in \Omega} V_\Omega^S(\omega) \\ \text{where } V_\Omega^S(\omega) &\stackrel{\text{def}}{=} \sum_{x \in \mathcal{V}_\Omega^S(\omega)} L(x, \omega) \end{aligned}$$

Let us call $V_\Omega^S(\omega)$ the *Voronoï distortion* caused by ω, since it is the contribution of the samples "around" ω to the global distortion $\mathcal{R}_{\text{emp}}^S(h_\Omega)$. The relevance of Voronoï distortion in the control of vector quantization algorithms has been introduced in (Frezza-Buet, 2014), it is detailed in forthcoming paragraphs.

Modeling the sample set as the result of a rejection sampling process

Let us consider that $\mathcal{X}$ is bounded, in order to allow for choosing some x uniformly from it, i.e. $x \leftarrow\!\!- \mathcal{U}_\mathcal{X}$. Let us define some *density function* $p \in [0,1]^\mathcal{X}$, such as $p(x) / \int_\mathcal{X} p(x)\,\mathrm{d}x$ is the density of probability of the random variable X.

Even if, in real situations, S is given, let us model it as the result of the rejection sampling process described in algorithm 10. The point is that $|S| \leq M$. Let us rename S as S^M when we want to stress that the data set is obtained from a rejection sampling process involving M attempts.

The point to be noticed is that, for a given set of prototypes Ω and a given density function p, the Voronoï distortions are approximately proportional to the number of attempts in the rejection sampling process.

$$\forall \omega \in \Omega,\ V_\Omega^{S^{k \times M}}(\omega) \approx k \times V_\Omega^{S^M}(\omega) \tag{14.8}$$

14.2 Minimum of distortion

Let us suppose in this section that the number of prototypes is constant, i.e. $|\Omega| = \kappa \in \mathbb{N}$. The goal of vector quantization techniques is to find the prototypes for which the distortion induced on some data set is minimal. Let us call $\Omega_\kappa^\star$ the result.

$$\Omega_\kappa^\star \in \operatorname*{argmin}_{\Omega \in \mathcal{P}_\kappa(\mathcal{X})} \mathcal{R}_{\text{emp}}^S(h_\Omega) \tag{14.9}$$

[1] The argmin operator is supposed to return a single element in the definition, which may be false theoretically. This case is not addressed for the sake of clarity, since this is not a big issue in real cases.

Algorithm 10 Computation of S.

$S \leftarrow \emptyset$ // Start with an empty set.
for $i \leftarrow 1$ **to** M **do**
  // Let us consider M attempts to add a sample in S.
  $x \xleftarrow{\mathcal{U}} \mathcal{X}$, $u \xleftarrow{\mathcal{U}} [0,1[$ // Choose a random place x in $\mathcal{X}$.
  if $u < p(x)$ **then**
    // The test will pass with a probability $p(x)$.
    $S \leftarrow S \cup \{x\}$ // x is kept (i.e. not rejected).
  end if
end for
return S

where $\mathcal{P}_\kappa(\mathcal{X})$ denotes the set of all κ-sized subsets of $\mathcal{X}$.

14.2.1 Non unicity

The fact that $\Omega_\kappa^\star$ is not unique may occur if the dataset configuration is symmetrical, as illustrated on figure 14.3.

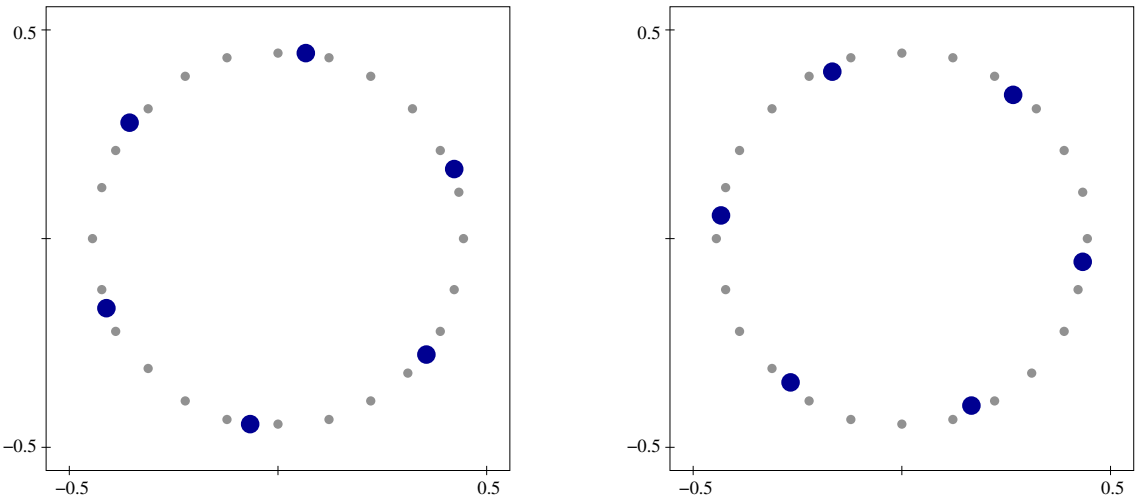


Figure 14.3: Here, $\mathcal{X} = [-0.5, 0.5]^2$. S is represented by smaller dots, while Ω is the larger ones. Left and right figures show two distinct optimal configurations for $\kappa = 6$.

14.2.2 Sensitivity to the density of input samples

Let us experiment some computation of the Voronoï distortion when $\Omega_\kappa^\star$ is reached. In the figures, $\Omega_\kappa^\star$ is computed by the algorithm 15 presented in section 15.1. In the experiment, $\mathcal{X} = [-0.5, 0.5]^2$ and $\kappa = 500$ are used. The dataset is built from algorithm 10, using $N = 50000$. On the figures, only 5000 inputs are actually displayed, for the sake of clarity. First experiment consists in using $p = 1$. On such a uniform distribution, figure 14.4 shows that prototypes are uniformly spread over the surface of $\mathcal{X}$ (left). The values of $V_{\Omega_\kappa^\star}^{S^{50000}}(\omega)$ for the κ prototypes ω are also represented (middle) as well as their histogram (right). It is observed experimentally that the Voronoï distortions is almost equally shared between the prototypes, when the positions of prototypes minimize the global distortion. Note that fluctuations in the Voronoï distortion values can be observed, in spite of the huge number of input samples, suggesting that this heterogeneity is not a discretization effect but reflects the nature of the $\Omega_\kappa^\star$ for that input configuration[2].

Let us now observe the effect of density variations in the examples, using the density function p shown in figure 14.5 to set up the sample set. This leads to the figure 14.6. It shows that, in spite

[2] As far as the author knows, no mathematical results are available to support or contradict this observation.

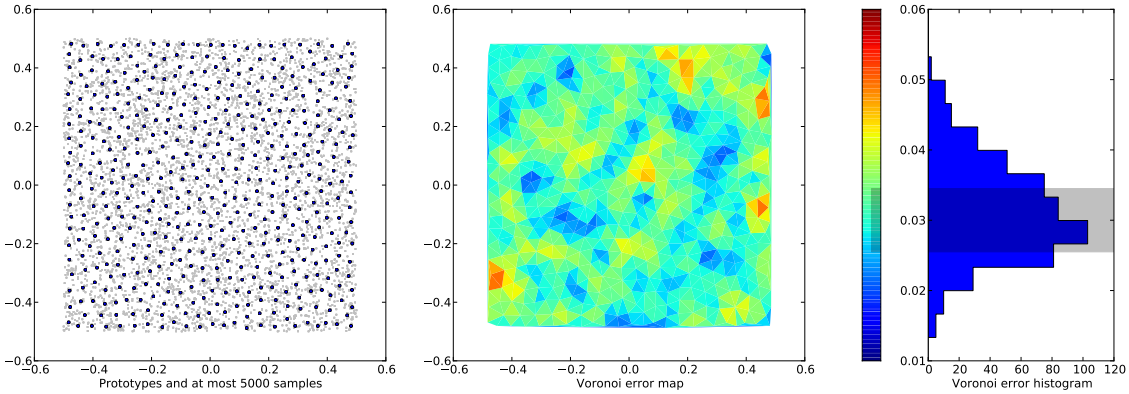


Figure 14.4: Vector quantization of a uniform distribution by $\kappa = 500$ prototypes. S^M is represented by smaller dots, while $\Omega^\star_\kappa$ is the larger ones. The darken range in the histogram is the shortest interval containing 50% of the values.

of some small fluctuations, the Voronoï distortions is almost equally shared between the prototypes as well when $\Omega^\star_\kappa$ is reached.

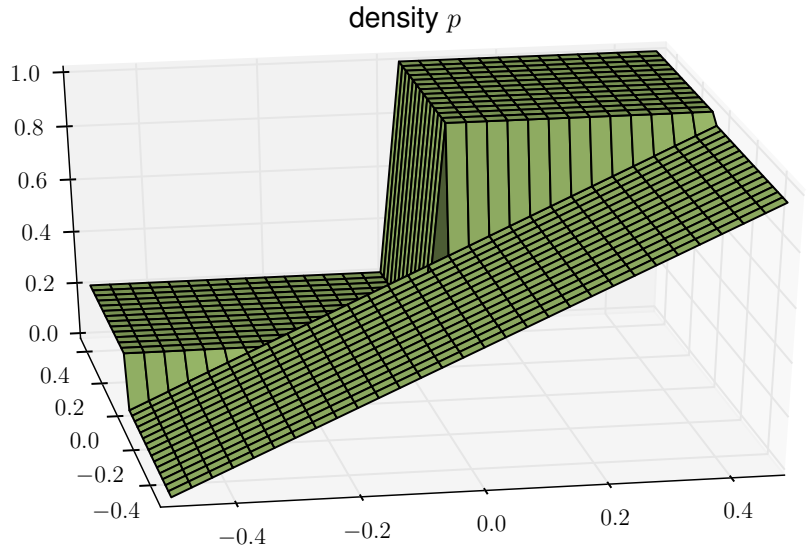


Figure 14.5: Non uniform density function.

The value of the equal distortion share, i.e the center of the darken range in figures 14.4-right and 14.6-right, is obtained from a large amount of weak sample contributions around prototypes where p is high, while it is obtained from a smaller amount of stronger sample contributions around prototypes where p is low.

> *In other words, the organization of the prototypes when optimality is achieved is sensitive to the density of the samples: prototypes are sparser where the density is lower.*

14.2.3 Controlling the quantization accuracy

It has been shown in the previous section that, when the minimum of distortion is reached by a set of prototypes, the Voronoï distortion is shared, almost equally, between the prototypes. The actual

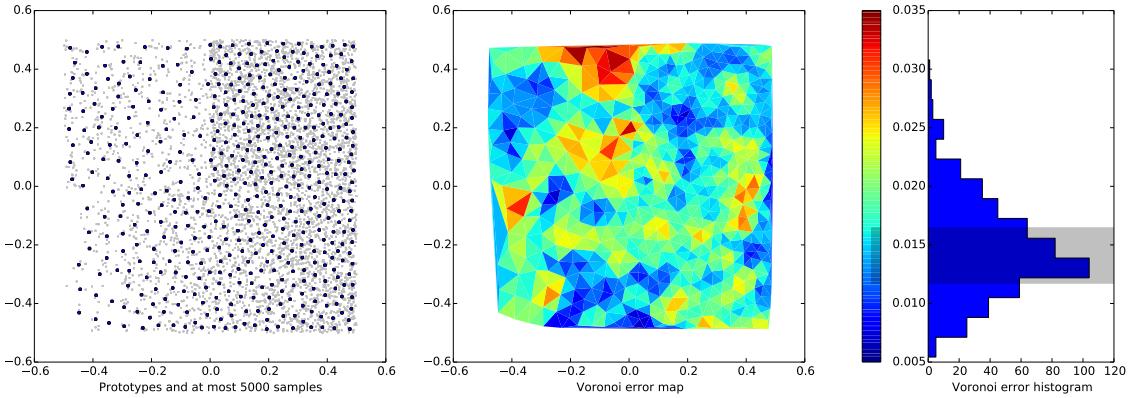


Figure 14.6: Same as figure 14.4, using an non uniform density function p.

value of this share (e.g 0.03 in figure 14.4, 0.013 in figure 14.6) appears to give a hint about the quantization accuracy: the more numerous the prototypes are, the lower is the share for each one. Let us use the value of this share to control the quantization accuracy. The samples are supposed to be tossed according to algorithm 10, and thus S is rather referred to as S^M.

Equation (14.8) states that $V_{\Omega_\kappa^\star}^{S^M}(\omega)$ is proportional to M. Let us denote by T the proportional coefficient.

$$\forall \omega \in \Omega_\kappa^\star,\ V_{\Omega_\kappa^\star}^{S^M}(\omega) \approx MT,\ T \in \mathbb{R}^+ \tag{14.10}$$

The coefficient T can be viewed as a *target*, determined in advance in order to set the accuracy of the quantization. Once T is fixed, a *targeted vector quantization process* consists of setting the appropriate number of prototypes κ, knowing M, such as each of the measured $V_{\Omega_\kappa^\star}^{S^M}(\omega)$, $\omega \in \Omega_\kappa^\star$ is close to the value MT. This is what the `VQ-T` algorithm does (see algorithm 11). The use of the *shortest confidence interval* allows to extract the "main" values from a collection (Guenther, 1969). The implementation of algorithm 11 is naive since it can easily be improved by a dichotomic approach.

Algorithm 11 `VQ-T`(T, S^M)

$\kappa \leftarrow 1$ // Start with a single prototype, $[\cdots]$ denotes lists.
repeat
  Compute $\Omega_\kappa^\star$ according to eq. (14.9) // Use your favorite VQ algorithm here
  $(a, b) =$ `shortest_confidence_interval` $\left(\left[V_{\Omega_\kappa^\star}^{S^M}(\omega)\right]_{\omega \in \Omega_\kappa^\star}, \delta\right)$ // Use $\delta = .5$
  if $T.M < a$ **then**
    $\kappa \leftarrow \kappa + 1$
  else if $T.M > b$ **then**
    $\kappa \leftarrow \kappa - 1$
  end if
until $T.M \in [a, b]$
return $\Omega_\kappa^\star$

The value of T can be chosen by trial and errors, but considering a geometrical interpretation is worth it. Let us consider $\mathcal{X} = [-0.5, 0.5]^2$, the quadratic loss $L(x, \omega) = (x - \omega)^2$ and a uniform distribution (i.e. $p = 1$). Let us suppose that, in such a situation, the desired quantization accuracy consists of $\kappa^\star$ prototypes. The prototypes are the elements of $\Omega_{\kappa^\star}^\star$ obtained from a vector quantization algorithm. The quantization accuracy in figure 14.4 where $p = 1$ actually corresponds to $\kappa^\star = 500$. Let us consider the *Voronoï tessellation* induced by the prototypes. The Voronoï tessellation is the partition of $\mathcal{X}$ into $\kappa^\star$ cells $C_\omega \overset{\text{def}}{=} \left\{x \in \mathcal{X} \,\middle|\, h_{\Omega_{\kappa^\star}^\star}(x) = \omega\right\}$. Let us also consider $S_\omega \overset{\text{def}}{=} \left\{x \in S \,\middle|\, h_{\Omega_{\kappa^\star}^\star}(x) = \omega\right\}$ similarly. It can be considered that the Voronoï

Algorithm 12 `shortest_confidence_interval`(V, δ)

Require: $\delta \in [.5, 1]$ // $V = [v_i]_{0 \leq i < N}$ is the list of N values.
1: $l \leftarrow \texttt{int}\,(\delta.N)$ // `int` (x) stands for the closest integer to x
2: sort V in an increasing order
3: // r is set greater than any possible interval length.
4: $r \leftarrow v_{N-1} - v_0 + 1$
5: **for** $k \leftarrow 0$ **to** $N - l$ **do**
6: // Values from k to $k + l$ are a fraction δ of all the values.
7: // r' is the width of the range of these values.
8: $r' \leftarrow v_{k+l-1} - v_k$
9: **if** $r' < r$ **then**
10: $r \leftarrow r',\ j \leftarrow k$ // The minimum found is saved.
11: **end if**
12: **end for**
13: **return** (v_j, v_{j+l-1})

tessellation divides the area of $\mathcal{X}$, which is 1 here, into $\kappa^\star$ parts with the same surface, i.e $1/\kappa^\star$ each. Let us approximate the shape of each cell C_ω by a circle centered at ω. The radius r is such that the surface of the disk is the area of the cell, i.e. $\pi r^2 = 1/\kappa^\star$, i.e $r^2 = 1/\pi\kappa^\star$. The quadratic momentum μ of the disk[3] is $\pi r^4/2$. The variance μ is the momentum divided by the disk area, i.e. $\mu = \big(\pi r^4/2\big)/\pi r^2 = r^2/2 = 1/2\pi\kappa^\star$. The variance ν of any S_ω approximates the variance μ. By definition, $\nu = V_\Omega^{S^M}(\omega)/|S_\omega|$. As p is uniform, there are exactly M samples in S^M and they are equally shared in the S_ω. Therefore, $|S_\omega| \approx M/\kappa^\star$. So the variance can be re-written as $\nu \approx \kappa^\star V_\Omega^{S^M}(\omega)/M$. Identifying μ with ν leads to $\mu \approx \kappa^\star V_\Omega^{S^M}(\omega)/M$ and thus $V_\Omega^{S^M}(\omega) \approx \mu M/\kappa^\star$. Identifying the latter expression with equation (14.10) leads to $T = \frac{\mu}{\kappa^\star}$. As $\mu = \frac{1}{2\pi\kappa^\star}$, we have

$$T = \frac{1}{2\pi\kappa^{\star 2}}$$

In the case of figure 14.4, since $\kappa^\star = 500$, we have $T = 6.37 \times 10^{-7}$ and thus, from equation (14.10) with $M = 50000$, $V_\Omega^{S^M}(\omega) \approx 50000 \times 6.37 \times 10^{-7} \approx 0.0318$. This value actually lies between the darken range in figure 14.4.

Let us now apply the algorithm 11 with the density p depicted in figure 14.5, with $M = 50000$ and $T = 0.0318$. This leads to $\kappa = 343$. The configuration is displayed in figure 14.7. Comparing the upper-right regions of figures 14.7 and 14.6 shows that in figure 14.7, the accuracy is similar to the desired one, i.e. figure 14.4.

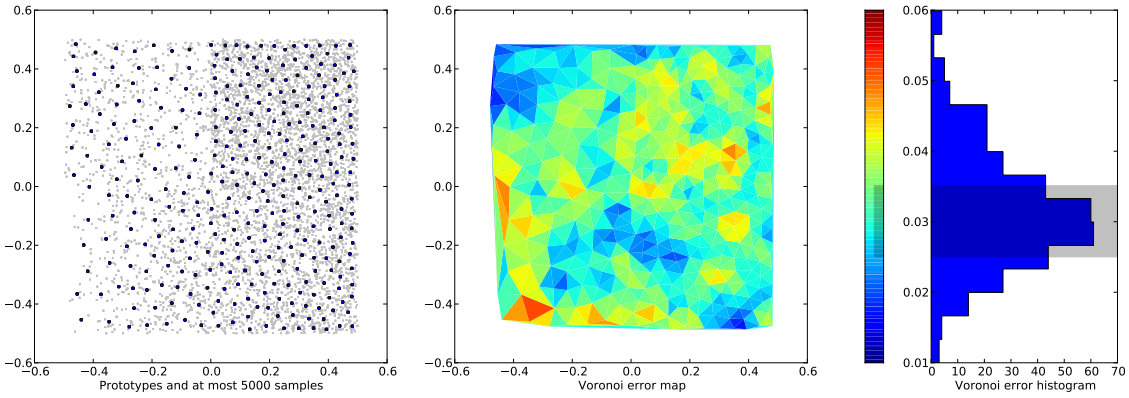


Figure 14.7: Same as figure 14.6 except that $\kappa = 343$.

[3]The quadratic momentum is $\int_{\text{disk}} |x - G|^2 \mathrm{d}x$, with G the center.

14.3 Preserving topology

The input space $\mathcal{X}$ has a topology induced by the loss function L, since it can usually allow for the definition of neighborhoods in $\mathcal{X}$. Nevertheless, the actual distribution of inputs in $\mathcal{X}$ may not span the entire $\mathcal{X}$, but may rather lie on a manifold. For example, if $\mathcal{X} = \mathbb{R}^2$, the inputs may lie on a circle, that is a 1-D sub-manifold of $\mathbb{R}^2$. Preserving topology consists in building up a structure around the prototypes in Ω in order to reflect the topology of the manifold where the samples are extracted from. This structure is a graph, where the prototypes are the vertices, and an edge between two prototypes mean that they are actually neighbours within the manifold. Lots of vector quantization algorithm handle the prototypes as a graph.

14.3.1 Notations for graphs

Let us here set up notation for graphs, since all the vector quantization algorithms presented next consists in handling a graph. A graph consists of vertices and edges. The vertices are a finite set $\mathbb{V} = \left\{v_1, \cdots, v_i, \cdots, v_{|\mathbb{V}|}\right\} \subset \mathbb{N}$ containing vertex identifiers. The edges are non oriented and connect distinct vertices. An edge, here, is thus a 2-sized set of vertices. Let us denote an edge $\{v, v'\} = \{v', v\}$ by $\mathrm{e}_{v \leftrightarrow v'}$ or $\mathrm{e}_{v' \leftrightarrow v}$. Let $\mathbb{E}$ be the finite set of the edges of the graph.

Let us define the set of the neighbors of a vertex v and the corresponding edges as

$$\begin{aligned} \mathcal{N}(v) &= \{v' \in \mathbb{V} \mid \mathrm{e}_{v \leftrightarrow v'} \in \mathbb{E}\} \\ \mathcal{E}(v) &= \{\mathrm{e}_{a \leftrightarrow b} \in \mathbb{E} \mid a = v \text{ or } b = v\} \end{aligned}$$

Extending the graph with a new vertex simply consists in adding a new element (that was not in $\mathbb{V}$) in $\mathbb{V}$.

One notation issue comes from the need to anchor values to both vertices and edges. In other words, we need these objects to have "attributes", in the programming sense of the term. This can be represented by functions. For example, as prototypes are handled by vertices in the algorithms, a function $\mathrm{proto} \in \mathcal{X}^{\mathbb{V}}$ is defined, such as $\omega = \mathrm{proto}(v)$ is the prototype hosted by v. Edges may handled an age (integer). In that case, the function $\mathrm{age} \in \mathbb{N}^{\mathbb{E}}$ enables to define the age $\mathrm{age}(\mathrm{e}_{v \leftrightarrow v'})$ of the edge $\mathrm{e}_{v \leftrightarrow v'}$. Some other attributes/functions may be used further.

Last, attribute affectations means that the function is changed. It is denoted by $\leftarrow$. Changing the prototype handled by a vertex is thus denoted by $\mathrm{proto}(v) \leftarrow \omega$. It means that a new proto is now considered. It is identical to the previous one, except that from now, for the value v, $\mathrm{proto}(v) = \omega$.

The notation for hypothesis space can be set from vertices, rather than from prototypes, for the sake of clarity in algorithms. Indeed, Let us denote :

$$h_{\mathbb{V}}(x) = \underset{v' \in \mathbb{V}}{\operatorname{argmin}} L(\mathrm{proto}(v'), x) \tag{14.13}$$

We will allow the following writings, without ambiguity:

$$\begin{aligned} \omega &= h_{\Omega}(x) \\ v &= h_{\mathbb{V}}(x) \end{aligned}$$

14.3.2 Masked Delaunay triangulation

We have already mentioned the *Voronoï tessellation* earlier. The tessellation is a partition of the space into convex regions, delimited by hyperplanes, so that all the points in a cell are closer to one prototype. Figure 14.8 illustrates this in the $\mathbb{R}^2$ case.

The *Delaunay triangulation* can be obtained as the dual of the Voronoï tessellation as follows: if two cells share a common boundary hyperplane, the prototypes of that cells are connected in the Delaunay triangulation. This is what figure 14.9, on top, shows. Let us notice that the input samples are not uniformly distributed in $\mathcal{X} = \mathbb{R}^2$, since they lie within a two-piece distribution. In Martinez and Schulten (1994), the interest of the *masked Delaunay triangulation* has been highlighted, since it reflects the topology of the inner manifold. The masked Delaunay triangulation is

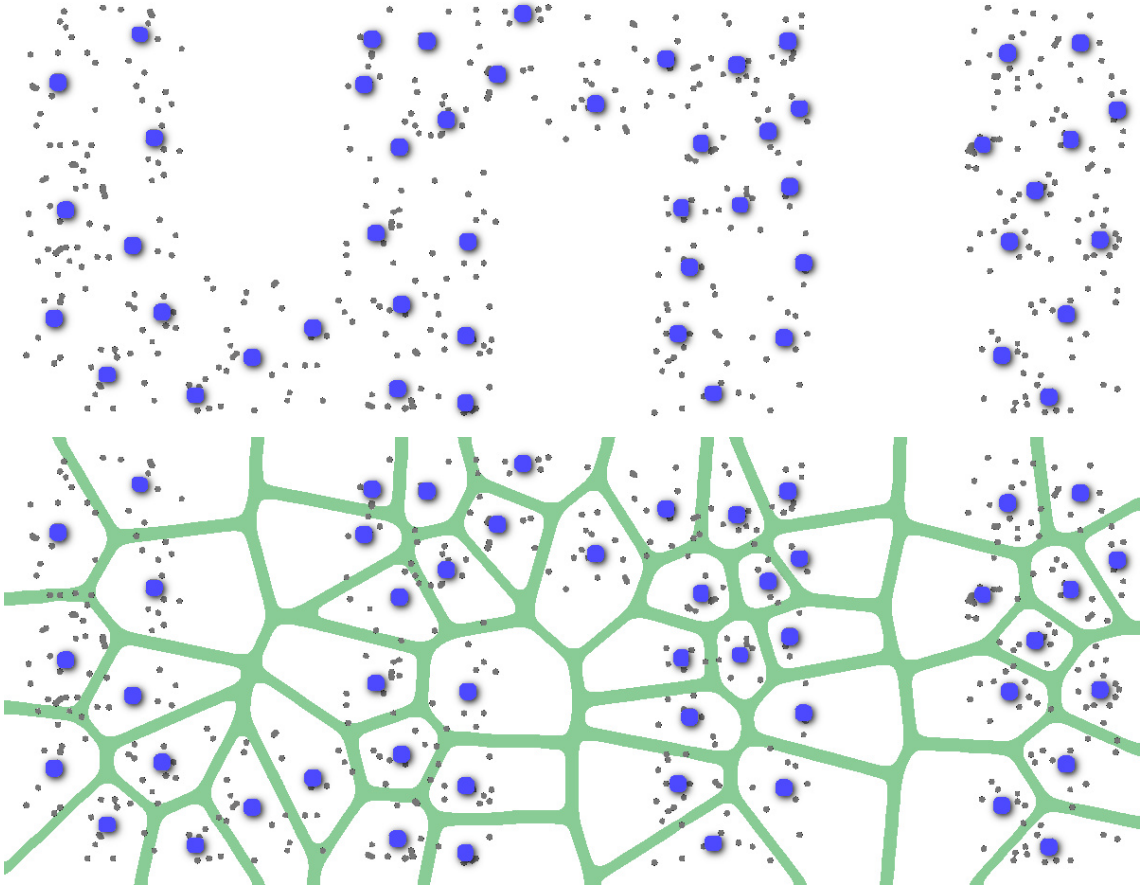


Figure 14.8: Top: vector quantization of the input samples (grey dots) by 50 prototypes (blue dots). Bottom: Voronoï tessellation (green). Each cell is the region where points are closer to the central prototypes that to other ones.

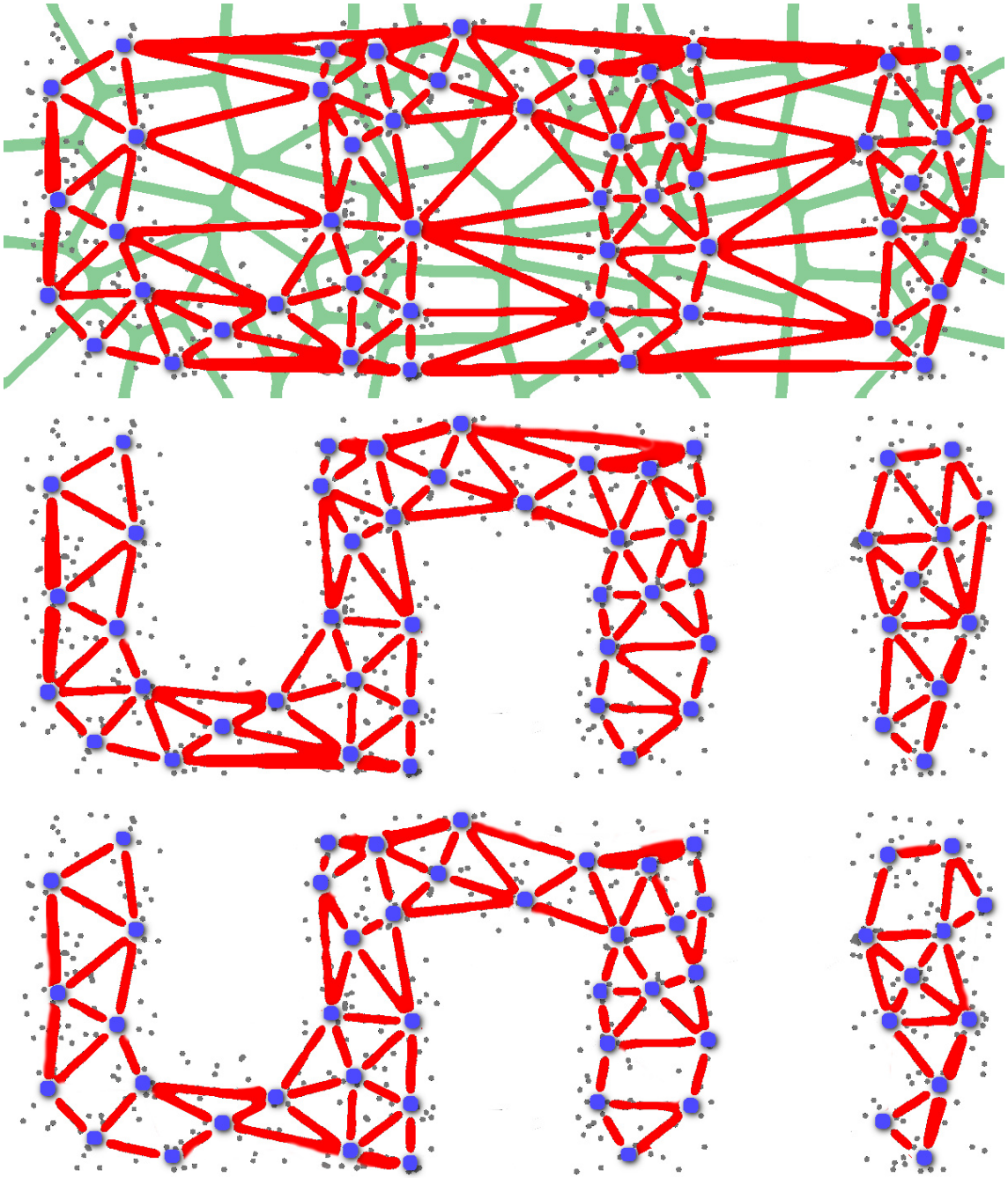


Figure 14.9: Top: Delaunay triangulation (Voronoï tessellation dual). Middle: Masked Delaunay triangulation. Bottom: Approximated masked Delaunay triangulation obtained by CHL (see algorithm 13).

a sub graph of the Delaunay triangulation that only has edges covering "well" the manyfold (see. middle of figure 14.9).

Computing Delaunay triangulations geometrically is feasible, but determining, from a set of samples, which edges actually belong to the masked Delaunay triangulation is not obvious. Moreover, many vector quantization algorithm build the masked Delaunay triangulation incrementally. In that context, the very simple competitive Hebbian learning algorithm (Martinez and Schulten, 1994) is the basis of these algorithms (see. algorithm 13). It leads to the graph in the bottom of figure 14.9. As one can see, some edges are missing, since the graph obtained is not a triangulation.

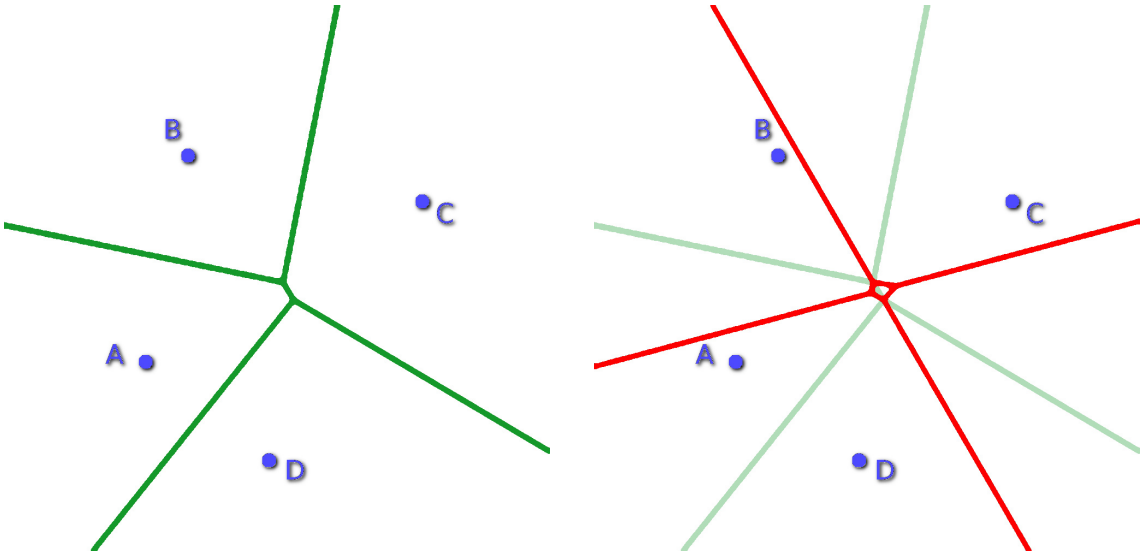


Figure 14.10: Left: Voronoï tesselation for nearly co-cyclic points. Right: The second order Voronoï tesselation.

Missing edges often correspond to four points that almost lie on the same circle. Such points are depicted in figure 14.10. On the left, the Voronoï tessellation is showed for prototypes A, B, C, D. Regions A and B, B and C, C and D, D and A, as well as A and C, share a common edge. The Delaunay triangulation is made of the 5 segments $[AB]$, $[BC]$, $[CD]$, $[DA]$, $[AC]$. On the right plot in figure 14.10, the *second order Voronoï tessellation* is depicted (in red). Each point in a cell in that plot have the same two closest prototypes. In other words, in the `CHL` procedure (algorithm 13), if a sample tossed belong to one of the second order Voronoï cells, the corresponding edge of the Delaunay triangulation is created. It can be seen in figure 14.10-right that the creation of the edge AC is very unlikely with `CHL`, since the corresponding region, i.e the central cell, is tiny for almost co-circular points. This explains the missing of some of the edges when comparing middle and bottom plots in figure 14.9.

14.3.3 Structuring raw data

The main power of vector quantization is its ability to bridge the gap between analogical and symbolic worlds. In other words, the input data are tossed according to some input distribution, which is a continuous object, whereas the result of the process is a graph[4], which is a symbolic object. This enables to *talk* about the distribution with symbolic term as "the distribution contains two separated parts", "the distribution is made of two cycles", etc. Putting words onto analogical input received by sensors is what human do when they speak about whet they perceive and vector quantization can be viewed as a step in that direction for artificial systems.

Let us illustrate this on the example of digit recognition. When someone recognizes a shape as the digit 9, s/he can explicit that the shape is a nine because it is made of a cycle with a tail pending on the right. So if we consider the distribution of the pixels belonging to the shape of

[4]A set of prototypes, without any edges, is a graph as well.

Algorithm 13 Competitive Hebbian Learning, CHL(Ω, n)

Require: $\mathbb{V}$ is set of vertices hosting prototypes, $|\mathbb{V}| \geq 2$.

$\mathbb{E} \leftarrow \emptyset$, $i \leftarrow 0$.
while $i < n$ **do**
  $i \leftarrow i + 1$
  Sample some x.
  find $v = h_{\mathbb{V}}(x)$ // See eq. (14.13)
  find $v' = h_{\mathbb{V} \setminus \{v\}}(x)$
  if $\mathrm{e}_{v \leftrightarrow v'} \notin \mathbb{E}$ **then**
    $i \leftarrow 0$
    $\mathbb{E} \leftarrow \mathbb{E} \cup \{\mathrm{e}_{v \leftrightarrow v'}\}$
  end if
end while
return $\mathbb{E}$

a nine in a picture, a topology preserving vector quantization may lead to a graph from which a cycle and a tail can be extracted (see figure 14.11).

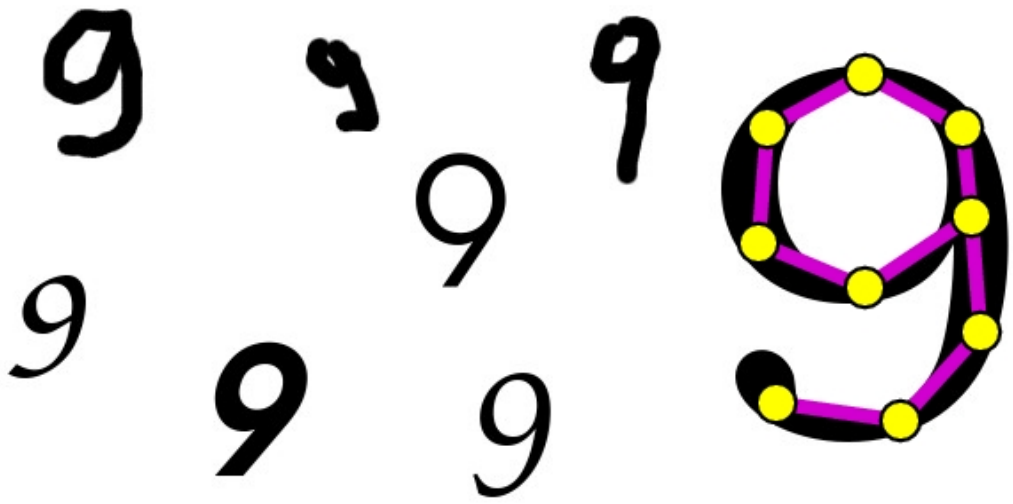


Figure 14.11: The pixels of the digit 9 can be structured as a cycle with pending tail on the right.

Chapter 15

Main algorithms

In this chapter, main vector quantization techniques are presented. Nevertheless, the reader should keep in mind that lots of variations of those algorithms are available in the literature. An overview of vector quantization algorithms can also be found in Fritzke (1997).

The reader is invited to refer to paragraph 14.3.1 for notations related to graphs.

15.1 K-means

15.1.1 The Lloyd iteration

The *Lloyd iteration*, named from Lloyd (1982), is the core process of the distortion minimisation used in the k-mean algorithm. Let us consider a dataset $S = \{x_1, \cdots, x_i, \cdots, x_N\}$ that is to be clustered in k groups. The clustering consists in giving to each sample x_i in S a group number, i.e. a label denoted by $l_i \in \{1, 2, \cdots, k\}$. The labelling $l \in \{1, 2, \cdots, k\}^{\{1,2,\cdots,|S|\}}$ is a function that will be successively updated. Let us consider a set of prototypes $\Omega = \{\omega_1, \cdots, \omega_p, \cdots, \omega_k\}$. Like the labels associated to the samples, the prototypes will be updated as well. So let rather consider Ω as a function $\omega \in \mathcal{X}^{\{1,2,\cdots,k\}}$, such as ω_p is the p-th prototype in the set Ω.

The Lloyd iterations consist of searching both the labelling function and the set of prototype sequentially, i.e. to iterate a serie (l^t, ω^t). in order to find a function that minimizes the distortion. The Lloyd iteration specified by algorithm 14 updates first l, and then ω.

Algorithm 14 The Lloyd iteration

Require: A current set of prototype ω^{t-1}.

1: $\forall\ 1 \leq i \leq |S|\,,\ l_i^t = \underset{1 \leq p \leq k}{\operatorname{argmin}} \left(x_i - \omega_p\right)^2$ // x_i is labelled with the index of the closest prototype.

2: $\forall\ 1 \leq p \leq k,\ \omega_p^t = \dfrac{\sum_{i \in C_p} x_i}{|C_p|}$ where $C_p \overset{\text{def}}{=} \left\{i \mid l_i^t = p\right\}$

The line 1 of algorithm 14 consists in associating each input sample with the closest prototype. The label of an input sample is thus the index of its closest prototype. Only the labelling of the prototypes changes at this step. The line 2 actually updates the prototype such as each prototype is the barycenter of the samples that have been associated to it at previous step. At the next iteration, since the prototypes have changed, the labelling performed at line 1 may change at its turn.

In order to see what happens when successive Lloyd iterations are performed, let us associate an energy $\mathcal{E}^S(l, \omega)$ to any (l, ω), defined by equation (15.1), keeping in mind that it is a pair of functions.

$$\mathcal{E}^S(l, \omega) \overset{\text{def}}{=} \sum_{1 \leq i \leq |S|} \left(x_i - \omega_{l_i}\right)^2 = \sum_{1 \leq p \leq k} \sum_{i \in C_p} \left(x_i - \omega_p\right)^2 \tag{15.1}$$

Let us show that both steps of the Lloyd iteration actually decrease the energy. In algorithm 14, applying line 1 changes the energy from $\mathcal{E}^S\left(l^{t-1},\omega^{t-1}\right)$ to $\mathcal{E}^S\left(l^t,\omega^{t-1}\right)$. This is obviously a decrease of energy, i.e. $\mathcal{E}^S\left(l^{t-1},\omega^{t-1}\right)\geq\mathcal{E}^S\left(l^t,\omega^{t-1}\right)$, since the labelling is define so as to associate each input to the closest prototype. If $l^{t-1}\neq l^t$, we have indeed a strict decrease.

After this first step, as each sample is associated to its closest prototype, the energy is the distortion introduced page 133, in the specific case of the quadratic loss.

$$\frac{\mathcal{E}^S\left(l^t,\omega^{t-1}\right)}{|S|}=\mathcal{R}^S_{\text{emp}}\left(h_{\Omega^{t-1}}\right),\ \text{where}\ \Omega^{t-1}=\left\{\omega_1^{t-1},\cdots,\omega_k^{t-1}\right\}$$

Then, applying line 2 of algorithm 14 changes the energy from $\mathcal{E}^S\left(l^t,\omega^{t-1}\right)$ to $\mathcal{E}^S\left(l^t,\omega^t\right)$. It is a decrease as well. Indeed, each cluster C_l contributes to the global energy by a value $\sum_{i\in C_p}\left(x_i-\omega_p\right)^2$ (see the right part of equation (15.1)). Taking ω_p as the barycenter of the samples involved in the summation leads to the minimization of this sum[1]. This is what line 2 of algorithm 14 actually does, so the energy is actually decreased.

As ω^t is deduced from l^t (see line 2 of algorithm 14), let us rename $\mathcal{E}^S\left(l^t,\omega^t\right)$ as $\mathcal{E}^S\left(l^t\right)$. The serie $\mathcal{E}^S\left(l^t\right)$ decreases with t. It is indeed defined on a finite space, since the number of labelling l of the samples is finite[2]. The serie thus converges towards a minimum. We can also state that the labelling is stable, using a reductio ad absurdum. Indeed, let us suppose that, when the convergence is reached, the serie passes by several *distinct* l^t. As these labelling are distinct, line 1 of algorithm 14 leads to a strict decrease of energy, so the convergence is not reached. As a consequence, once the minimal is reached, the labelling is stable.

As a conclusion, iterating algorithm 14 can be done until the labelling do not change (i.e. $l^t=l^{t+1}$). This will happen. Then, ω^{t+1} reaches a *local* minimum of the distortion defined at page 133. This convergence result is the core of the k-mean algorithm defined in the newt section.

15.1.2 The Linde-Buzo-Gray algorithm

The *k-means* algorithms (Lloyd, 1982; Linde et al., 1980; Kanungo et al., 2002) is certainly the most famous vector quantization algorithm, since it is implemented in any numerical processing framework. It considers a set of samples a priori and computes the position of k prototypes so that they minimize the distortion (see algorithm 15). The idea is to update the prototypes so that each of them is the mean of the samples that lies in its Voronoï region.

It is batch, since it works for a set given as one bulk of data, and k is a parameter that has to be determined by the user. The line 6 of algorithm 15 consists of cloning some existing prototypes. Here, cloning a vertex means creating a new vertex hosting a random prototype which is very close to the prototype of the initial vertex. The new vertex is added into $\mathbb{V}$. The reaching of stopping condition has be proven, but the result may be a local distortion minimum. Figures 14.4, 14.6 and 14.7 are obtained by the use of this algorithm.

Algorithm 15 k-means

1: Sample $S=\{x_1,\cdots,x_i,\cdots,x_N\}$ according to p.
2: Compute ω_1 as the mean of S.
3: $\mathbb{V}=\{v\}$ such as $\text{proto}\left(v\right)=\omega_1$ // Let us start with a single vertex
4: **while** $|\mathbb{V}|<k$ **do**
5: Select randomly $n=\min\left(k-|\mathbb{V}|,|\mathbb{V}|\right)$ vertices in $\mathbb{V}$.
6: Clone these n vertices (the clones are added in $\mathbb{V}$).
7: **repeat**
8: $\forall\, x\in S,\ \text{label}\left(x\right)\leftarrow h_{\mathbb{V}}\left(x\right)$
9: $\forall\, v\in\mathbb{V},\ \text{proto}\left(v\right)\leftarrow\text{mean}_{\{x\in S\ |\ \text{label}(x)=v\}}\,x$
10: **until** No label change has occurred. // This loop made of Lloyd's iterations always ends.
11: **end while**

[1] Given $\{x_1,\cdots,x_n\}$ and a value g, the error $E\left(g\right)=\sum_i\left(x_i-g\right)^2$ is minimal when g is the barycenter of the x_is. The proof is obtained by computing $\frac{\partial}{\partial g}E\left(g\right)=0$.

[2] There are $k^{|S|}$ ways of labelling the input samples.

15.1.3 The online k-means

There is an online version of the Linde-Buzo-Gray algorithm (MacQueen, 1967), that is not really useful for actually processing data. Nevertheless, the structure of the algorithms presented further reflects the one of this online k-means. Algorithm 16 is very easy to program.

Algorithm 16 k-means online

1: Sample $\Omega = \{\omega_1, \cdots, \omega_i, \cdots, \omega_k\}$ according to p.
2: Make $\mathbb{V} = \{v_1, \cdots, v_i, \cdots, v_k\}$ such as $\forall i,\ \text{proto}(v_i) = \omega_i$.
3: **while true do**
4: Sample x according to p.
5: Determine $v^\star = h_{\mathbb{V}}(x)$. // Competition.
6: $\text{proto}(v^\star) \leftarrow \text{proto}(v^\star) + \alpha(x - \text{proto}(v^\star))$ // learning, $\alpha \in]0, 1]$.
7: **end while**

There is no real stopping criterion. After a while, the prototypes hosted by the vertices in $\mathbb{V}$ are placed in order to minimize the distortion[3]. Line 5 selects the vertex whose prototype is the closest to the input sample. This stage is a *competition*. Line 6 says that the winning vertex $v^\star$ is *the only one* whose prototype is modified, consecutively to the computation of the current input. This is called a *winner-take-all* learning rule.

The update of the winning vertex (line 6) is performed by a low past first order recursive filter. This computes each $\text{proto}(v)$ as the mean of the inputs sample for which v hosted the closest prototype. The same idea motivates the Linde-Buzo-Gray algorithm, as stated previously. Increasing α make the prototypes shake, whereas smaller α leads to more stable positions. A good compromise could be to use a large $\alpha = 0.1$ for first steps, allowing the prototypes to roughly take their positions, and then use a much more smaller $\alpha = 0.005$.

15.2 Incremental neural networks

One issue with vector quantization is to find the appropriate number of vertices. In k-means, this number is determined a priori. On the contrary, in incremental neural networks algorithms, the idea is to increase the number of vertices until some stopping criterion is achieved.

15.2.1 Growing Neural Gas

The *growing neural gas* (GNG) was proposed in Fritzke (1995a). It consists of handling a graph of prototypes, by successively cloning the current vertices and update their connections thanks to the competitive Hebbian learning presented in section 14.3.2. Successive stages of the GNG evolution are shown in figure 15.1, the GNG procedure is algorithm 17.

In order to control the number of prototypes and adjust it when the input distribution changes, a GNG-T algorithm was proposed (Frezza-Buet, 2014) as an extension of GNG, taking advantages of the accuracy control introduced in section 14.2.3.

15.2.2 Growing Grid

With GNG-inspired algorithms, the topology of the manifold where the input samples are lying is retrieved by competitive Hebbian learning, as figure 15.1 shows. Another approach consists of maintaining the graph as a *growing grid* (Fritzke, 1995b), thus ignoring the actual topology of the data. The relevance of forcing the topology is detailed in section 15.3, where self-organizing maps are introduced. Here, the idea is to start with four prototypes, connected as a square, and add a full row or a full line when growth of the network is required. The connections are created when the line (or the row) is added, and it is not updated then, as opposed to GNG. Figure 15.2 shows the successive stages of the Growing Grid evolution.

[3] Algorithm 16 is indeed a stochastic gradient descent.

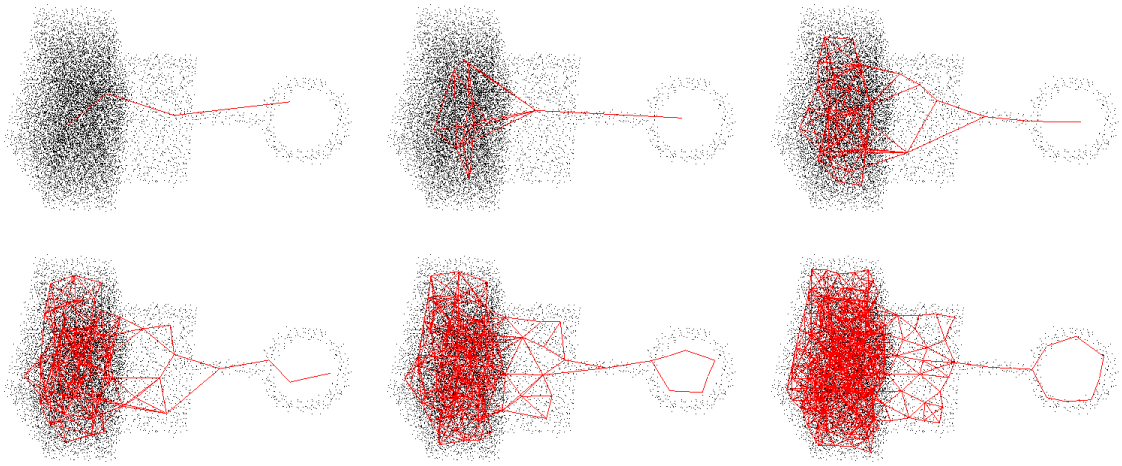


Figure 15.1: Successive evolution steps of GNG from a 3D input sample distribution. Dots are the input samples. The graph is represented as a red grid, showing edges that intersect at vertices. Each is placed v at the position of $\text{proto}\,(v)$.

Algorithm 17 Growing Neural Gas

Choose randomly $(\omega, \omega') \in \mathcal{X}^2$, set up $\mathbb{V} = \{v, v'\}$ such as $\text{proto}\,(v) = \omega$ and $\text{proto}\,(v') = \omega'$.
$\text{dist}\,(v) \leftarrow 0,\ \text{dist}\,(v') \leftarrow 0$ // New vertices have not accumulated local distortion yet.
$i \leftarrow 0$ // this counts the samples.
$\mathbb{E} \leftarrow \emptyset$ // Empty edge set at start.
while stopping criterion not met **do**
  Sample some x according to p, $i \leftarrow i + 1$.
  find $v = h_{\mathbb{V}}\,(x)$
  find $v' = h_{\mathbb{V}\setminus\{v\}}\,(x)$
  if $\mathrm{e}_{v\leftrightarrow v'} \notin \mathbb{E}$ **then**
    $\mathbb{E} \leftarrow \mathbb{E} \cup \{\mathrm{e}_{\omega\leftrightarrow\omega'}\}$ // An edge is added if it did not exist already
  end if
  $\text{age}\,(\mathrm{e}_{v\leftrightarrow v'}) \leftarrow 0$ // The age of the edge is set (or reset) to 0.
  $\text{dist}\,(v) \leftarrow \text{dist}\,(v) + L\,(\text{proto}\,(v), x)$ // The local distortion is accumulated for the winner.
  $\text{proto}\,(v) \leftarrow \text{proto}\,(v) + \alpha(x - \text{proto}\,(v))$ // learning, $\alpha \in]0, 1]$.
  $\forall v' \in \mathcal{N}\,(v),\ \text{proto}\,(v') \leftarrow \text{proto}\,(v') + \zeta\alpha(x - \text{proto}\,(v'))$ // weaker learning, $\zeta \in]0, 1]$. $\zeta = 0.1$ is ok.
  $\forall \mathrm{e}_{v\leftrightarrow v''} \in \mathcal{E}\,(v),\ \text{age}\,(\mathrm{e}_{v\leftrightarrow v''}) \leftarrow \text{age}\,(\mathrm{e}_{v\leftrightarrow v''}) + 1$ // edges emanating from v get older.
  Remove edges older than $a_{\max}$. If a vertex has no more edges after these removals, suppress it.
  if i is a multiple of λ **then**
    find $v = \text{argmax}_{v''\in\mathbb{V}}\,\text{dist}\,(v'')$ // v is the vertex with the highest accumulated distortion.
    find $v' = \text{argmax}_{v''\in\mathcal{N}(v)}\,\text{dist}\,(v'')$ // v' is the neighbor of v with the highest accumulated distortion.
    create a new v'' such as $\text{proto}\,(v') = (\text{proto}\,(v) + \text{proto}\,(v'))/2$ and add it in $\mathbb{V}$.
    $\mathbb{E} \leftarrow \mathbb{E} \setminus \{\mathrm{e}_{v\leftrightarrow v'}\}$.
    $\mathbb{E} \leftarrow \mathbb{E} \cup \{\mathrm{e}_{v\leftrightarrow v''}, \mathrm{e}_{v''\leftrightarrow v'}\}$.
    $\text{dist}\,(v) \leftarrow \text{dist}\,(v) - \gamma\text{dist}\,(v)$ // $\gamma \in]0, 1]$.
    $\text{dist}\,(v') \leftarrow \text{dist}\,(v') - \gamma\text{dist}\,(v')$
    $\text{dist}\,(v'') \leftarrow (\text{dist}\,(v) + \text{dist}\,(v'))/2$.
  end if
  $\forall v \in \mathbb{V},\ \text{dist}\,(v) \leftarrow \text{dist}\,(v) - \gamma\text{dist}\,(v)$
end while

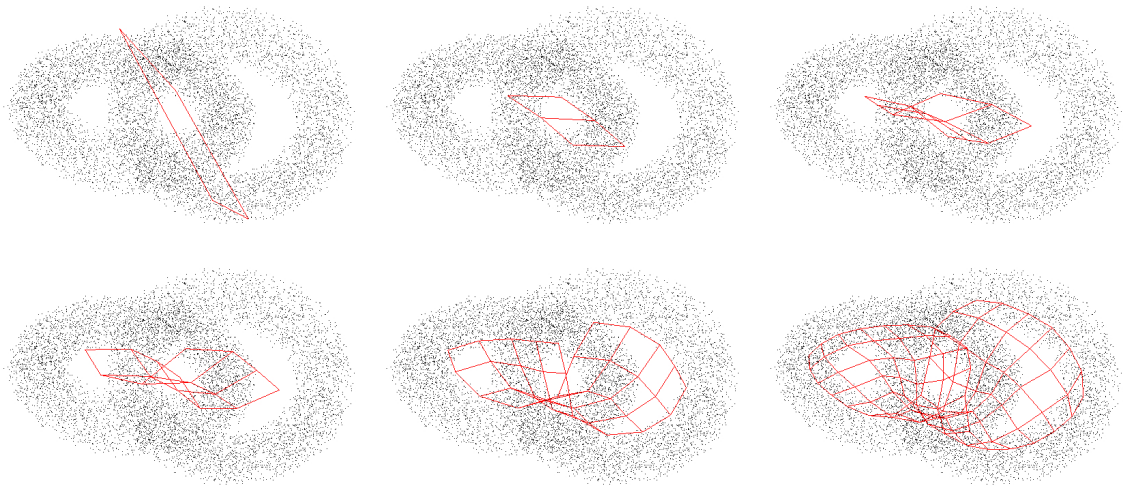


Figure 15.2: Successive evolution steps of Growing Grid from a 3D input sample distribution. Drawing convention is the one of figure 15.1.

15.3 Self-Organizing Maps

15.3.1 Principle

A *self-organizing map* (Kohonen, 1989, 2013) is an vector quantization algorithm where the prototypes are organized as a graph *a priori*. The graph is kept constant during all the process, as opposed to incremental networks presented previously. Even if the graph used for self-organizing maps is a 2D grid, the algorithm can be implemented with any kind of graph.

With self-organizing maps (SOM), two distances are involved. The first is the loss L used to match prototypes against inputs. The second one is a graph distance, denoted by ν, that is a distance *in the graph* separating two vertices. For example, $\nu(v, v')$ can be the number of edges of the shortest path from v to v'.

Given this, the self-organizing maps can be introduced (see. algorithm 18). Note that it is very close to algorithm 16.

Algorithm 18 Self-Organizing Maps

1: Let $(\mathbb{V}, \mathbb{E})$ be a graph, given a priori. // The proto $(v), v \in \mathbb{V}$ can be initialized randomly.
2: Let ν a graph distance induced by the edge set $\mathbb{E}$.
3: **while true do**
4: Sample x according to p.
5: Determine $v^\star = h_{\mathbb{V}}(x)$. // Competition.
6: $\forall v \in \mathbb{V}$, proto $(v) \leftarrow$ proto $(v) + \alpha h(\nu(v^\star, v))(x -$ proto $(v))$ // learning, $\alpha \in]0, 1]$.
7: **end while**

As for algorithm 16, line 5 performs a competition, i.e. the selection of the vertex whose prototype it the closest to the current input. However, the learning stage, i.e. line 6, slightly changes in algorithm 18. Indeed, learning is applies to *all* the prototypes. The strength of learning is determined by the term $\alpha h(.)$. It corresponds to a modulation of the learning rate α. The function $h \in [0, 1]^{\mathbb{R}^+}$ has to be a decreasing function such that $h(0) = 1$. As $h(\nu(v^\star, v))$ is used at line 6, the modulation is the highest when $\nu(v^\star, v) = 0$, i.e. when $v = v^\star$ is considered. The modulation decreases for the neighbors of $v^\star$ *in the graph*, since $\nu(v^\star, v)$ is still high for them. For prototypes v that are far, *in the graph*, from $v^\star$, $\nu(v^\star, v)$ is weaker and the learning as no significant effect. To sum up, as opposed to algorithm 16, $v^\star$ is not the only prototype that is modified by the current input sample, since its close neighbors also learn. This is called a *winner-take-most* learning scheme.

Note that a function h such that $h(0) = 1$ and $h(.) = 0$ otherwise makes algorithm 18 be identical to algorithm 16.

15.3.2 Convergence issues

The convergence of Self-organizing maps has not been proved in the general case even if it works fine in practical cases (Cottrell et al., 1998). Nevertheless, one has to clearly understand what SOM do actually compute, and the effects of the function h. Let us illustrate this on an example in $\mathcal{X} = \mathbb{R}^2$, where the density p has a coronal shape. The graph is a grid $\mathbb{V} = \{v_{i,j}\}_{1 \leq i,j \leq 20}$. The edges are those of a rectangular mesh, sketched on the figures. Nevertheless, the edges only influence the algorithm through ν. Here, we use for the sake of simplicity $\nu(v_{i,j}, v_{i',j'}) = \sqrt{(i-i')^2 + (j-j')^2}$. The function h is defined in equation (15.2).

$$h(\nu) = \begin{cases} 1 - \frac{\nu}{r} & \text{if } \nu \leq r \\ 0 & \text{otherwise} \end{cases} \tag{15.2}$$

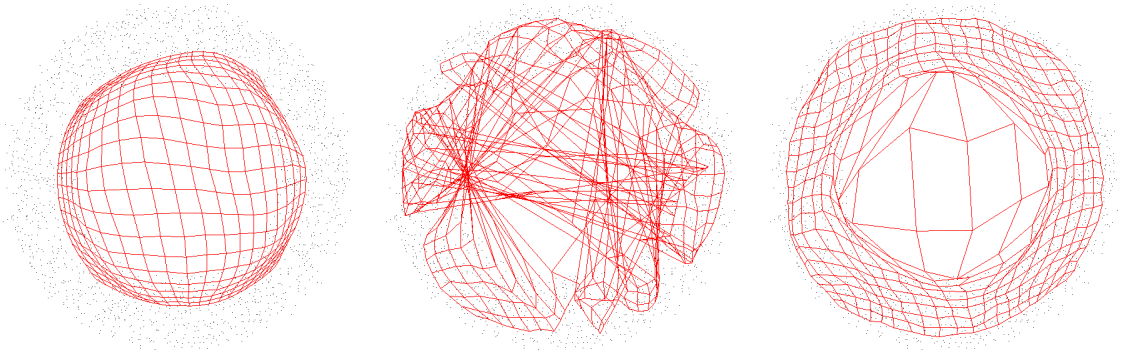


Figure 15.3: SOM applied to a coronal input distribution. Drawing convention is the one of figure 15.1. See text for details .

Figure 15.3 shows the results. In figure 15.3-left, $r = 15$ is used (see equation (15.2)). A wide area of prototypes around the winner actually learn. This has an averaging effect, all the prototypes tend to be attracted to the mean of the input samples. Nevertheless, it can be observed that the grid is "unfolded". In figure 15.3-middle, $r = 3$ is used. The averaging effect is weaker, and the prototypes cover the input sample distribution better. The drawback is that the map is "folded". In figure 15.3-right, we used first $r = 15$, and then $r = 3$. This leads to both an unfolded and nicely covering prototype distribution. Nevertheless, as figure 15.3-right shows, some prototypes (the middle ones in the figure) lie outside the distribution, because the map elasticity pulls them in opposite directions. Such prototypes are sometime called *dead units*.

> *The h function in the literature is often presented as a Gaussian with a slowly decreasing variance, which complicates the formulation of the SOM algorithm. Indeed, simpler h can be used, and the progressive decay can be reduce to few values, the first ones with a wide h expansion and the last ones with a narrower h.*

Once the map is correctly unfolded over the input samples, the following stands : two prototypes that are close according to ν are also close according to L. The reverse is wrong (see figure 15.4).

> *Be careful with the input sampling. It has to be random. For example, in figure 15.3-right, submitting examples line bye line, from the top to the bottom and from the left to the right within each line, would have lead to a bad unfolding.*

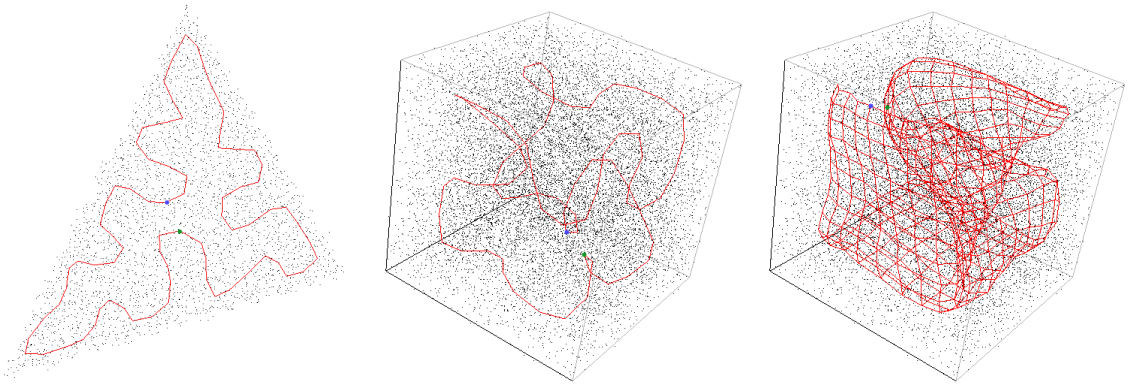


Figure 15.4: When the input manifold dimension is higher than the one of the map. On the figures, the blue and green prototypes are close according to L but far according to ν. Left and middle: The graph is a ring. Right: the graph is a grid. Drawing convention is the one of figure 15.1.

15.3.3 Example

Let us consider the case of written digits. Input samples are 28×28 gray-scaled images, where digits are written. Input samples are thus $x \in \mathcal{X} = [0, 255]^{784}$. As a loss function L, we do not use directly the Euclidian distance in $\mathbb{R}^{784}$ (see. section 14.1.2). Rather, when we compare $\text{proto}(v)$ to x, we first blur the images and then compute the Euclidian distance between the blurred objects.

In this example, the input sample lie in a manifold in $\mathbb{R}^{784}$. Visualizing this manifold is not easy. As we force the topology to be 2D, since we use a grid for connecting the vertices, we are able to represent the hosted prototypes as a 2D grid on a screen, displaying at each grid position the prototypical image. Recognition can be performed as follows: Ask an expert to label each vertex according to its prototype (their number is finite). When a handwritten digit needs to be labeled, find the vertex hosting the closest prototype in the map and give its label to the input.

Another, and maybe more fundamental, aspect of the map in figure 15.5 is that is represents the distribution of all the input digits over the surface of the screen, trying to place the prototypes such as the ones that are close on the screen are actually close digits in $\mathbb{R}^{784}$. This is an example of using self-organizing maps as non-linear projections for visualizing high dimensional data.

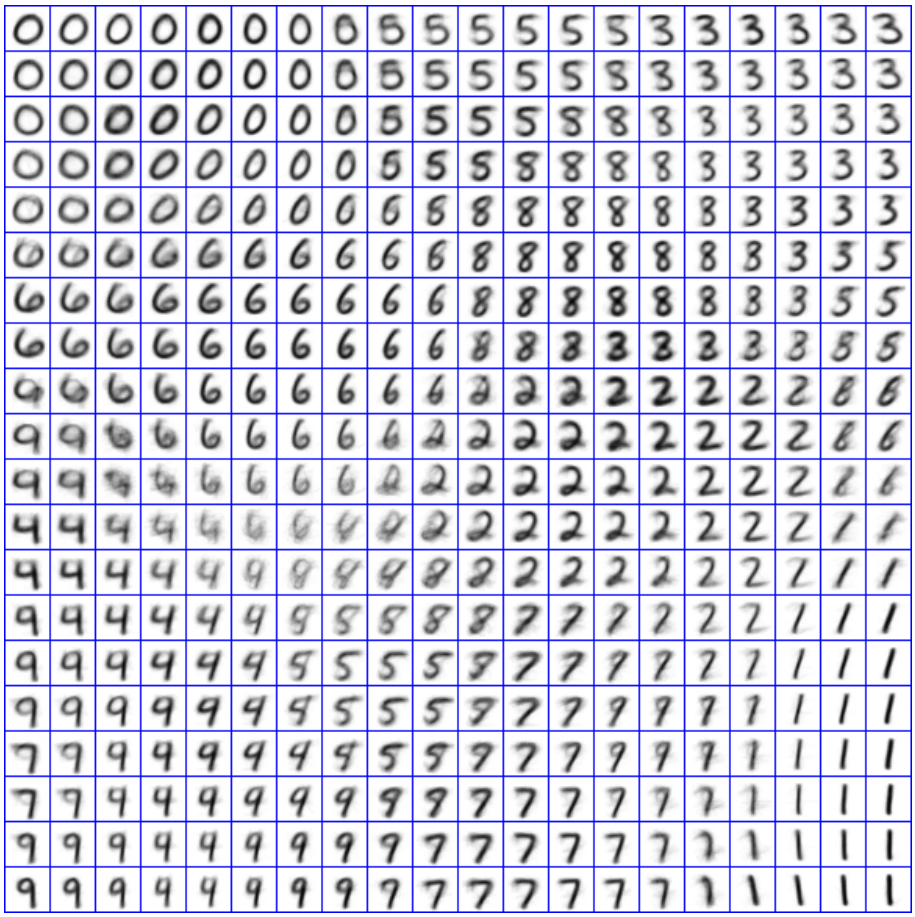


Figure 15.5: Handwritten digits mapping with a self-organizing map.

Part V

Neural networks

Chapter 16

Introduction

What is a neural network ? A *neural network* is basically a set of interconnected units (i.e. a graph of units), having inputs and outputs, which compute by themselves a pretty simple function of their inputs. The idea of studying a network of interconnected units perfoming a rather basic computation originates from (McCulloch and Pitts, 1943) which introduces a simple model of a neuron with several inputs x_i feeding the neuron through weighted connections of weight w_i. The weighted sum of the input contributions $\sum_i w_i x_i$ provides the pre-activation of the neuron from which its output is computed with a heaviside transfer function $h(x) = \mathbb{1}_{x \geq 0}$:

$$y = h(\sum_i w_i x_i)$$

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

The neuron model of (McCulloch and Pitts, 1943) was not equipped with learning rules allowing to adapt its weights. As we shall see in the next chapters, various improvements were found ultimately leading to trainable neural networks.

Even if the first motivation was to model how the brain works, we shall prefer speaking about units rather than neurons as speaking about neurons tend to insist too much on a relationship with biological neurons. Definitely, biological neurons inspired (and still inspire) the design of neural networks but neural networks can be considered as a specific structure of predictors in machine learning on their own without having to refer to any biological motivations to justify their study.

If we denote $\mathbf{x}$ the inputs of a unit and y its output, a prototypical neural network unit links the inputs to the output through a non-linear function f applied to a linear combination of the inputs :

$$a = \mathbf{w}^\intercal \mathbf{x} + b$$
$$y = f(a)$$

where f is a so-called transfer function, $\mathbf{w}$ a set of weights, b a bias and a the pre-activation which is introduced for convenience. The transfer function linking the pre-activation and the output (or activation) of the unit can take different forms and below is a list of some commonly chosen transfer functions :

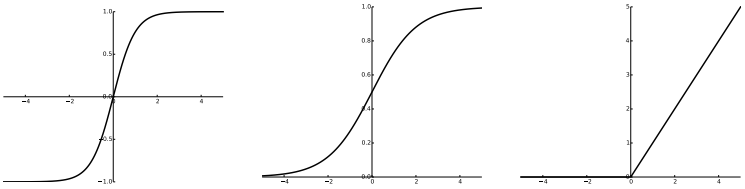
- hyperbolic tangent : $f(a) = \tanh(a)$
- sigmoid[1] : $f(a) = \frac{1}{1+\exp(-a)}$
- rectified linear unit (ReLu) : $f(a) = \max(a, 0) = [a]^+$
- softplus : $f(a) = \log(1 + \exp(a))$

[1] also called logistic or squashing function

While the hyperbolic tangent and sigmoid were common choices for the transfer function, it turns out that the softplus and rectified linear units bring up interesting results in terms of performances of the learned predictor and the speed of learning(Nair and Hinton, 2010; Zeiler et al., 2013). The ReLu is really quick to evaluate contrary to transfer functions involving exponentials! It also behaves quite favourably when having to derive it as we shall see latter in the chapter. These transfer functions are plotted on figure 16.1. There are also population-based transfer functions where the output of a unit actually depends on the pre-activation of a collection of other units. A popular example is the softmax function. If we consider a population of units for which we denote a_i the pre-activations and y_i the outputs, the softmax computes the outputs as :

$$\forall i, y_i = \frac{\exp(a_i)}{\sum_j \exp(a_j)}$$

The softmax is especially used in the context of learning a classifier as the softmax transfer function constrains the activations to lie in the range $[0, 1]$ and to sum up to 1. The softmax also induces a competition among the interconnected units : if one unit raises its activation and due to the normalization constraint, it necessarily induces a drop of the activation of at least one of the other units.



a) b) c)

Figure 16.1: Classical transfer functions; a) hyperbolic tangent $f(a) = \tanh(a)$, b) sigmoid $f(a) = \frac{1}{1+\exp(-a)}$, c) rectified linear unit $f(a) = \max(a, 0) = [a]^+$, d) softplus : $f(a) = \log(1 + \exp(a))$.

Up to now, we especially focused on the type of computation that a single unit is performing. The topology of the network is also a distinguishable feature of neural networks (fig 16.2). Some neural networks are acyclic or feedforward; you can, say, identify the leaves with the inputs and the root with the output if we take the convention that information flows from the leaves up to the root. In general, we can group the units into layers and therefore speak about the input layer, the output layer, and the hidden layers in between. The hidden layer are so called because these contain the units for which you actually do not know the value while the inputs and outputs are provided by the datasets in a supervised learning problem. Actually, nothing prevents you from considering an architecture where the units have connections to a layer that is not the next one with so called *skip-layer connections*. In particular, if one knows that the output contains some linear dependencies on the input, it could be beneficial to add such skip layer connections. These connections do not actually enhance the expressiveness of the architecture but slightly push the network into the right direction when learning comes into play.

When the data has a hierarchical structures, some neural networks such as recursive and recurrent neural networks are more appropriate. With recursive neural networks, the same network is evaluated on children to compute a representation of a parent. The children can actually be inputs from the dataset or could also be some parent representation. Recursive neural networks are appropriate when dealing with data that have actually a hierarchical structure such as in natural language processing. In a recurrent neural network, cycles within the network are introduced. These cycles produce a memory effect in the network as the activations of the units depend not only on the current input but also on the previous activations within the network. This type of network is particularly suited for datasets such as time series.

Having briefly sketched what are neural networks, in the next chapters, we go in details through a variety of neural network architectures especially focusing on how we train them, i.e. how we find optimal parameters given a regression or classification problem and what they are good for. Classical books on neural networks include (Bishop, 1995; Hertz et al., 1991). (Schmidhuber, 2015; Bengio et al., 2013) recently reviewed the history of the ideas in the neural network community

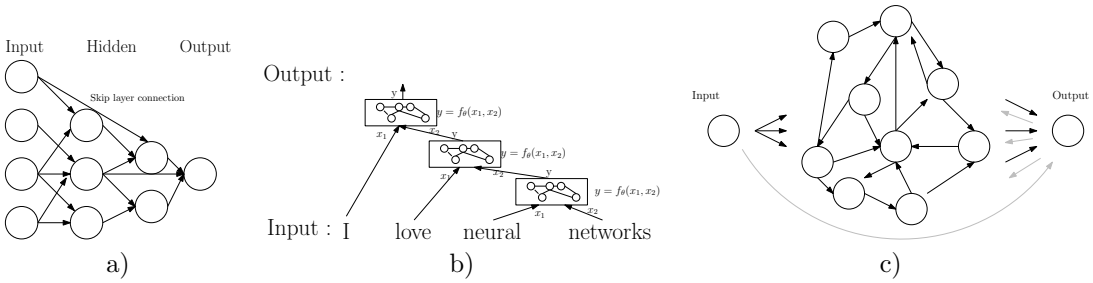


Figure 16.2: a) A *feedforward neural network* is an acyclic graph of units. b) A *recursive neural network* is applied recursively to compute the parent representation of two childrens, these two childrens being possibly parent nodes themselves. c) A *recurrent neural network* is fed with a sequence of inputs, the network itself possibly containing cycles.

and pointed out as well recent trends in the field. There is a also the book of (Bengio et al., 2015) that is, at the time of writting these lines, a work in progress written by researchers from the university of Montreal (Y. Bengio), one the top leading research group in neural networks with the university of Toronto (G. Hinton) and the IDSIA research group (J. Schmidhuber). The online book of Michael Nielsen (`http://neuralnetworksanddeeplearning.com/`) is also a good reference.

Chapter 17

Feedforward neural networks

17.1 Single Layer perceptron

17.1.1 Perceptron

Architecture

The perceptron was introduced in (Rosenblatt, 1962). The simple perceptron he introduced is depicted on fig. 17.1. It is made of an input layer, an association layer and an output layer[1]. The association layer A computes predefined functions of the inputs while the connections between the association layer and output layer are trainable (or plastic). Given an input $\mathbf{x}$, we denote $\Phi_j(\mathbf{x})$ the activities in the association layer, the basis functions Φ_j being predefined.

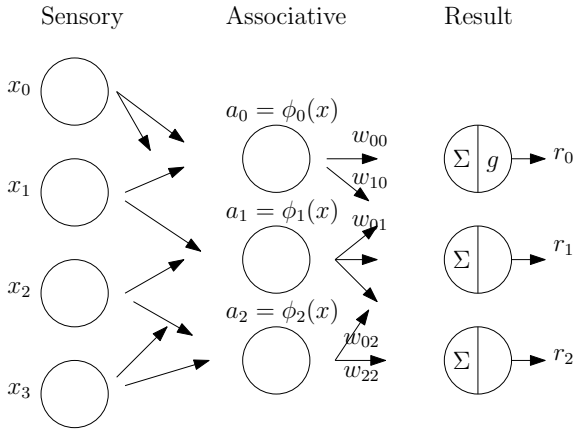


Figure 17.1: A perceptron is an acyclic graph with an input or sensory layer x, an association layer a and an output or result layer r. The association layer activities are computed with predefined basis functions ϕ_i. The weights between the associative and result are trainable (or plastic).

The outputs of the perceptron are computed in two steps : 1) a linear combination of the activities in the associative layer defining the preactivation, 2) a transfer function g applied on the pre-activation :

$$\begin{aligned}\forall i, \mathbf{r}_i &= g\left(\sum_j w_{j,i}\mathbf{a}_i + \mathbf{b}_i\right) \\ &= g(\sum_j w_{j,i}\phi_i(\mathbf{x}) + \mathbf{b}_i)\end{aligned}$$

[1] Actually, F. Rosenblott introduces the layer as a Sensory layer, Association layer and Result layer and build up architectures from these S, A, R layers.

Let us denote n_s, n_a, n_r the number of units in, respectively, the sensory, associative and result layers. Introducing the vector of basis functions outputs $\phi(\mathbf{x}) = \begin{bmatrix} \phi_0(\mathbf{x}) \\ \phi_1(\mathbf{x}) \\ \vdots \end{bmatrix} \in \mathbb{R}^{n_a}$, the weight matrix $\mathbf{W} \in \mathbb{R}^{n_a \times n_r}$ with $\mathbf{W}_{i,j} = w_{i,j}$ where $w_{i,j}$ is the weight connecting the associative unit j to the output unit i, and the bias vector $\mathbf{b} = \begin{bmatrix} b_0 \\ b_1 \\ \vdots \end{bmatrix} \in \mathbb{R}^{n_r}$, the computation of the perceptron can be written compactly in matrix form :

$$\mathbf{r} = g\left(\mathbf{W}^{\mathrm{T}}\phi(\mathbf{x}) + \mathbf{b}\right)$$

where g is applied element-wise. You can write the above formula even more compactly if one adds an extra constant basis function $\phi_b(\mathbf{x}) = 1$ with extra weights to encompass the bias vector $\mathbf{b}$. We would then consider a weight matrix in $\mathbb{R}^{n_a+1 \times n_r}$ and the vector of basis functions would contain $n_a + 1$ entries with one entry set to 1.

The perceptron was introduced in the context of binary classification in which case the transfer function g is taken to be the sign function :

$$g(x) \begin{cases} -1 & \text{if} \quad x < 0 \\ +1 & \text{if} \quad x \geq 0 \end{cases}$$

As a last note to finish the introduction of the perceptron, it is actually a big reduction to summarize the contribution of (Rosenblatt, 1962) to the study of the S-A-R architecture for binary classification as he also studied variants of this architecture and the interested reader is referred to the original book[2]. In the next sections, we present how one can learn the weights between the association and result layers.

Learning : the perceptron algorithm

Given a classification problem with a set of input/output pairs $(x_i, y_i) \in \mathbb{R}^d \times \{-1, 1\}$, we want to learn a perceptron defined by :

$$\mathbf{y} = g\left(\mathbf{W}^{\mathrm{T}}\phi(\mathbf{x})\right)$$
$$g(x) \begin{cases} -1 & \text{if} \quad x < 0 \\ +1 & \text{if} \quad x \geq 0 \end{cases}$$

In this setting, $\phi(\mathbf{x}) \in \mathbb{R}^{n_a+1}, \phi(\mathbf{x}) = \begin{bmatrix} 1 \\ \phi_0(\mathbf{x}) \\ \phi_1(\mathbf{x}) \\ \vdots \end{bmatrix}$ and, since there is a single output $\mathbf{W} \in \mathbb{R}^{n_a+1}$ as well. The *perceptron learning rule* operates online and updates, after each sample $(\mathbf{x}_i, \mathbf{y}_i)$, the weights according to :

$$\mathbf{W} = \begin{cases} \mathbf{W} & \text{if the input is correctly classified, i.e. } g\left(\mathbf{W}^{\mathrm{T}}\phi(\mathbf{x}_i) + \mathbf{b}\right) = y_i \\ \mathbf{W} + \phi(\mathbf{x}) & \text{if the input is incorrectly classified as -1, i.e. } g\left(\mathbf{W}^{\mathrm{T}}\phi(\mathbf{x}_i) + \mathbf{b}\right) = -1 \text{ and } y_i = +1 \\ \mathbf{W} - \phi(\mathbf{x}) & \text{if the input is incorrectly classified as +1, i.e. } g\left(\mathbf{W}^{\mathrm{T}}\phi(\mathbf{x}_i) + \mathbf{b}\right) = +1 \text{ and } y_i = -1 \end{cases}$$

The perceptron learning rule can be actually compactly written as :

$$\mathbf{W} = \begin{cases} \mathbf{W} & \text{if the input is correctly classified, i.e. } g\left(\mathbf{W}^{\mathrm{T}}\phi(\mathbf{x}_i) + \mathbf{b}\right) = y_i \\ \mathbf{W} + y_i\phi(\mathbf{x}) & \text{otherwise} \end{cases}$$

[2] http://catalog.hathitrust.org/Record/000203591

Geometrical interpretation

We can have a geometrical understanding of how perceptrons and its learning rule work. Suppose we have a set of transformed input vectors $\phi(\mathbf{x}_i) \in \mathbb{R}^{n_a+1}$ and associated labels $y_i \in \{-1, 1\}$. Consider the space $\mathbb{R}^{n_a+1}$ in which belong the transformed inputs $\phi(\mathbf{x}_i)$ as well as the weights of the perceptron $\mathbf{w}$. We can associate an hyperplane to each of the transformed input vector $\phi(\mathbf{x}_i)$ defined by the following equations :

$$\mathbf{v}^{\mathrm{T}}\phi(\mathbf{x}_i) = 0$$

This hyperplane splits the space $\mathbb{R}^{n_a+1}$ into two regions : one in which $\mathbf{v}^{\mathrm{T}}\phi(\mathbf{x}_i) < 0$ and one in which $\mathbf{v}^{\mathrm{T}}\phi(\mathbf{x}_i) \geq 0$. Consider the case an input is correctly classified (fig. 17.2). If the input vector is positive ($y_i = 1$), then it means that both the weight vector $\mathbf{w}$ and transformed input $\phi(\mathbf{x}_i)$ belong to the same half space. If the input vector is negative ($y_i = -1$), then it means that the weight vector $\mathbf{w}$ and the transformed input $\phi(\mathbf{x}_i)$ do not belong to the same half space.

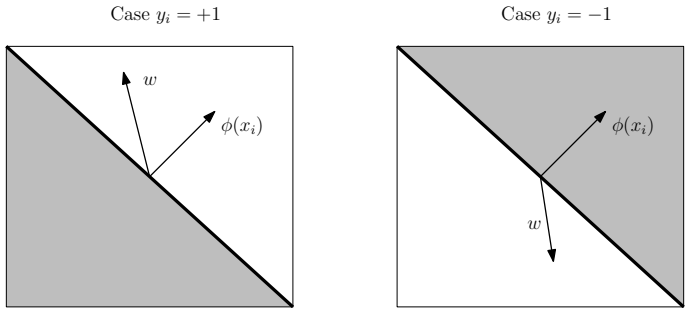


Figure 17.2: Case when a perceptron correctly classify an input $\mathbf{x}_i$. If the input is positive ($y_i = 1$), both the weight and transformed input belong to the same half space. If the input is negative ($y_i = -1$), the weight vector and transformed input do not belong to the same half space. The grey region indicates the half space in which a weight vector would misclassify the input.

Now, consider the case the perceptron is misclassifying an input (fig. 17.3). If the input is positive ($y_i = 1$), the weight vector and transformed input do not belong to the same half space. In order to correctly classify the input, they should actually belong to the same half-space. In this case, the perceptron learning rule is updating the weights as $\mathbf{w} + \phi(\mathbf{x}_i)$ which brings, at least in our example, the weight vector in the correct half space. If the input is negative ($y_i = -1$) both the weight and transformed input belong to the same half space while they should not in order to correctly classify the input. In this case, the perceptron learning rule is updating the weights as $\mathbf{w} - \phi(\mathbf{x}_i)$ which brings, at least in our example, the weight vector in the complementary half space where it should lie in order to correctly classify the input.

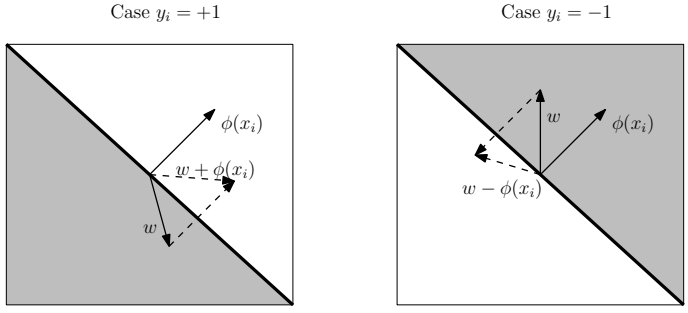


Figure 17.3: Case when a perceptron misclassify an input $\mathbf{x}_i$. If the input is positive ($y_i = 1$), the weight vector and transformed input do not belong to the same half space. If the input is negative ($y_i = -1$) both the weight and transformed input belong to the same half space. The grey region indicates the half space in which a weight vector actually misclassify the input.

Let us now consider the case we have two inputs $\mathbf{x}_1$ and $\mathbf{x}_2$, respectively positive and negative ($y_1 = 1$, $y_2 = -1$). Drawing the hyperplanes defined by $\mathbf{v}^{\mathrm{T}}\phi(\mathbf{x}_1) = 0$, $\mathbf{v}^{\mathrm{T}}\phi(\mathbf{x}_2) = 0$ delineate a

subspace of $\mathbb{R}^{n_a+1}$ in which a weight vector should be in order to correctly classify the two input vectors, a cone of feasible solutions. It is actually not necessary that such a region exists. Indeed, if, for example, we add an extra input $\mathbf{x}_3$ to the two inputs example we just considered, such that $\mathbf{x}_3 = \mathbf{x}_2 - \mathbf{x}_1$, and setting $y_3 = +1$ vanishes the cone of feasible solutions. We will go back to this question of feasibility in section 17.1.3.

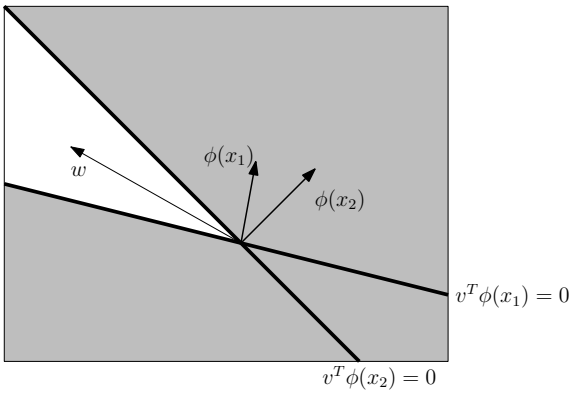


Figure 17.4: Considering two inputs $\mathbf{x}_1$ and $\mathbf{x}_2$ respectively positive and negative ($y_1 = 1$, $y_2 = -1$). In order to correctly classify the two inputs, the weight vector must to the white region, the cone of feasible solutions.

Linear separability

As we shall see in the next sections, the perceptron algorithm cannot only solve a particular class of classification problems which are called linearly separable.

Definition 17.1 (Linear separability)**.** *A binary classification problem* $(x_i, y_i) \in \mathbb{R}^d \times \{-1, 1\}, i \in [1..N]$ *is said to be* linearly separable *if there exists* $\mathbf{w} \in \mathbb{R}^d$ *such that :*

$$\forall i, \operatorname{sign}\left(\mathbf{w}^{\mathrm{T}} x_i\right) = y_i$$

with $\forall x < 0, \operatorname{sign}(x) = -1, \forall x \geq 0, \operatorname{sign}(x) = +1$.

The exact values of output, whether $\{0, 1\}$ or $\{-1, 1\}$ does not actually really care in the above definition. To illustrate the notion of linear separability, consider a binary classification problem with binary inputs, i.e. binary expressions. Consider two inputs x_1, x_2 in $\{0, 1\}$ and one output $y \in \{-1, 1\}$. The boolean expressions $and(x_1, x_2)$ or $or(x_1, x_2)$ are both linearly separable as shown on fig. 17.5.

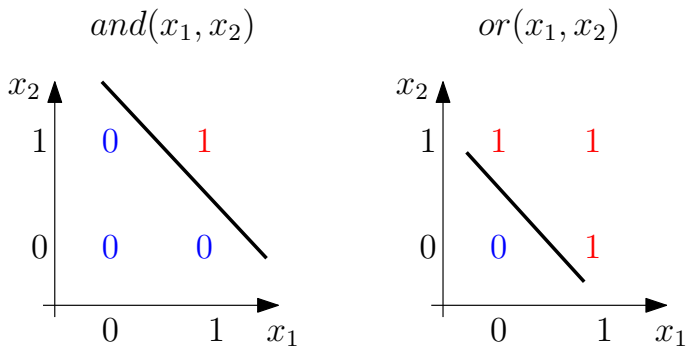


Figure 17.5: The AND and OR boolean functions are linearly separable as a line in the input space can be defined in order to place the positive inputs on one side and the negative inputs on the other side.

Not all the binary classification problems are linearly separable. One famous example is the XOR function depicted on fig. 17.6 for which there is no way to define a line separating the positive and negative inputs.

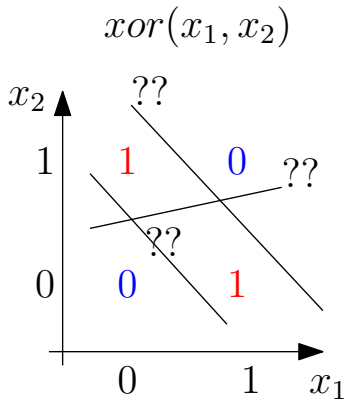


Figure 17.6: The XOR boolean function is not linearly separable as no line can be defined to split the positive and negative inputs.

One may wonder how many linearly separable functions with discrete inputs and outputs exist or even generalize and wonder about the probability that a randomly picked classification problem with real inputs is linearly separable. Actually, it turns out that all depends on the ratio between the number of data points N and the dimensionality of the input d. If $N < d$, any labelling of the inputs can be linearly separated. The probability of getting a linearly separable problem then quickly drops as the number of samples gets larger than the number of dimensions (Cover, 1965).

Perceptron learning rule convergence theorem

In case a classification problem is linearly separable, the perceptron learning rule can be shown to converge to a solution in a finite number of steps. Without loss of generality, we will consider a problem linearly separable in the input space. When introducing the perceptron, we mentioned using transformed inputs by introducing basis functions ϕ_i and we could consider a linearly separable classification problem in the transformed input space. However, as the basis functions were predefined, it is absolutely equivalent to consider that a problem is linearly separable in an input space whatever is this input space ("raw" or transformed). The *perceptron convergence theorem* states :

Theorem 17.1 (Perceptron convergence theorem)**.** *A classification problem* $(x_i, y_i) \in \mathbb{R}^d \times \{-1, 1\}, i \in [1..N]$ *is linearly separable (def 17.1) if and only if the perceptron learning rule converges to an optimal solution in a finite number of steps.*

Proof. Consider a linearly separable binary classification problem $(x_i, y_i) \in \mathbb{R}^d \times \{-1, 1\}, i \in [1..N]$. By definition, there exists $\hat{\mathbf{w}}$ such that :

$$\forall i, \hat{\mathbf{w}}^\intercal \mathbf{x}_i = y_i \Rightarrow \forall i, \hat{\mathbf{w}}^\intercal \mathbf{x}_i y_i > 0$$

Necessarily, $|\hat{\mathbf{w}}|_2 > 0$. Let us denote $\mathbf{w}^t$ the weight after having updated t misclassified inputs and $\mathbf{x}^t, \mathbf{y}^t$ the t-th misclassified input/output and we suppose that there exists an infinite sequence of misclassified input/output pairs[3]; otherwise, the proof ends immediately. For any $t > 0$, since the input/output pair $\mathbf{x}^t, \mathbf{y}^t$ was misclassified with the weights $\mathbf{w}^{t-1}$, it means $({}^\intercal\mathbf{x}^{t-1})\mathbf{x}^t\mathbf{y}^t < 0$. The

[3]at least one of the input/output pairs is considered infinitely many times

sequence of weights $\mathbf{w}^t$ after k updates using the perceptron learning rule will be :

$$\begin{aligned}\mathbf{w}^1 &= \mathbf{w}^0 + \mathbf{y}^1\mathbf{x}^1\\ \mathbf{w}^2 &= \mathbf{w}^1 + \mathbf{y}^2\mathbf{x}^2\\ &\vdots\\ \mathbf{w}^k &= \mathbf{w}^{k-1} + \mathbf{y}^k\mathbf{x}^k\end{aligned}$$

Taking $k > 0$ and summing all the above equations lead to $\mathbf{w}^k - \mathbf{w}^0 = \sum_{i=1}^{k} \mathbf{y}^i\mathbf{x}^i$. Let us compute the scalar product by $\hat{\mathbf{w}}$ (one solution to the linear separation) :

$$\hat{\mathbf{w}}^\intercal(\mathbf{w}^k - \mathbf{w}^0) = \sum_{i=1}^{k} \mathbf{y}^i\hat{\mathbf{w}}^\intercal\mathbf{x}^i$$

Since the problem is by hypothesis linearly separable, then $\forall i, \mathbf{y}^i\hat{\mathbf{w}}^\intercal\mathbf{x}^i > 0$. Let us denote $t_m = \min_{i\in[1,N]} \mathbf{y}^i\hat{\mathbf{w}}^\intercal\mathbf{x}^i > 0$. Therefore, we end up with :

$$\begin{aligned}\hat{\mathbf{w}}^\intercal(\mathbf{w}^k - \mathbf{w}^0) &\geq kt_m > 0\\ t_m &= \min_i \mathbf{y}^i\hat{\mathbf{w}}^\intercal\mathbf{x}^i > 0\end{aligned}$$

Reminding the Cauchy-Schwartz inequality[4], we get :

$$\begin{aligned}|\hat{\mathbf{w}}|_2\left|\mathbf{w}^k - \mathbf{w}^0\right|_2 &\geq \left|\hat{\mathbf{w}}^\intercal(\mathbf{w}^k - \mathbf{w}^0)\right| \geq kt_m\\ \Rightarrow \left|\mathbf{w}^k - \mathbf{w}^0\right|_2 &\geq \frac{kt_m}{|\hat{\mathbf{w}}|_2}\\ \Rightarrow \left|\mathbf{w}^k\right|_2 &\geq -\left|\mathbf{w}^0\right|_2 + \frac{kt_m}{|\hat{\mathbf{w}}|_2}\end{aligned}$$

Note $\left(\frac{t_m}{|\hat{\mathbf{w}}|_2}\right)$ is a constant dependent only on the dataset and a fixed solution $\hat{\mathbf{w}}$. Therefore, $\left|\mathbf{w}^k\right|_2$ is lower bounded by a linear function in the number of misclassified input/output pair k. This is a first point. Let us now focus on upper bounding the norm of $\mathbf{w}^k$:

$$\begin{aligned}\forall k > 0, \mathbf{w}^k &= \mathbf{w}^{k-1} + \mathbf{y}^k\mathbf{x}^k\\ \Rightarrow \left|\mathbf{w}^k\right|_2^{\,2} &= \left|\mathbf{w}^{k-1}\right|_2^{\,2} + \left|\mathbf{y}^k\mathbf{x}^k\right|_2^{\,2} + 2(\mathbf{w}^{k-1})^\intercal\mathbf{y}^k\mathbf{x}^k\end{aligned}$$

Remind that the input/output pair $\left(\mathbf{x}^k, \mathbf{y}^k\right)$ is the k-th misclassified input/output pair, meaning $(\mathbf{w}^{k-1})^\intercal\mathbf{x}^k\mathbf{y}^k < 0$ and therefore :

$$\begin{aligned}\forall k > 0, \left|\mathbf{w}^k\right|_2^{\,2} &< \left|\mathbf{w}^{k-1}\right|_2^{\,2} + \left|\mathbf{y}^k\mathbf{x}^k\right|_2^{\,2}\\ \Rightarrow \forall t, \left|\mathbf{w}^k\right|_2^{\,2} - \left|\mathbf{w}^{k-1}\right|_2^{\,2} &< \left|\mathbf{y}^k\mathbf{x}^k\right|_2^{\,2}\\ \Rightarrow \left|\mathbf{w}^k\right|_2^{\,2} - \left|\mathbf{w}^0\right|_2^{\,2} &= \sum_{i=0}^{k-1}(\left|\mathbf{w}^{i+1}\right|_2^{\,2} - \left|\mathbf{w}^i\right|_2^{\,2}) < \sum_{i=0}^{k-1}\left|\mathbf{y}^{i+1}\mathbf{x}^{i+1}\right|_2^{\,2}\\ \Rightarrow \left|\mathbf{w}^k\right|_2^{\,2} &< \left|\mathbf{w}^0\right|_2^{\,2} + kt_M\end{aligned}$$

with $t_M = \max_{i\in[1,N]} \left|\mathbf{y}^{i+1}\mathbf{x}^{i+1}\right|_2^{\,2}$. The latter implies $\left|\mathbf{w}^k\right|_2 < \sqrt{\left|\mathbf{w}^0\right|_2^{\,2} + kt_M}$. That is the second point. We therefore demonstrated that :

$$\begin{aligned}\forall k, -\left|\mathbf{w}^0\right|_2 + \frac{kt_m}{|\hat{\mathbf{w}}|_2} \leq \left|\mathbf{w}^k\right|_2 &< \sqrt{\left|\mathbf{w}^0\right|_2^{\,2} + kt_M}\\ t_m &= \min_{i\in[1,N]} \mathbf{y}^i\hat{\mathbf{w}}^\intercal\mathbf{x}^i > 0\\ t_M &= \max_{i\in[1,N]} \left|\mathbf{y}^{i+1}\mathbf{x}^{i+1}\right|_2^{\,2}\end{aligned}$$

[4]For any vector space E with a scalar product (a pre-Hilbert space), denoted $(u.v)$, then $|(u.v)|^2 \leq (u.u)(v.v)$

In the lower bound, we have a linearly increasing function of k. In the upper bound, we have an increasing function in $\sqrt{k}$. Necessarily, there is a finite value of the number of misclassified input/output pairs k for which the two curves cross after what the inequality cannot hold anymore which raises a contradiction and leads to the conclusion that there cannot be an infinite sequence of misclassified input/output pairs and therefore the perceptron algorithm is converging. Therefore, we demonstrated that if the classification problem is linearly separable, then the perceptron learning rule is converging in a finite number of updates.

The other implication is straightforward. If the perceptron is converging in a finite number of steps, it means that after say k updates, no mistakes are performed and, given the definition of the perceptron, this implies that the classification problem is linearly separable.

Given the equivalence, we can then also state that, in case the classification problem is not linearly separable, the perceptron algorithm will never converge since, otherwise, the classification problem would have been linearly separable. □

More on perceptrons

While we demonstrated the convergence of the perceptron learning rule, we did not speak that much about the rate of convergence. The learning rule we consider and associated algorithm which would peak each input/output pairs one after the other is not the algorithm that provides the fastest rate of convergence. There are variants of the perceptron learning rule with improved rate of convergence (Gallant, 1990; Muselli, 1997; Soheili and Pena, 2013; Soheili, 2014).

There are also extensions of the perceptron using kernels. As one may note, the weights of the perceptron are always a weighted (by the labels) sum of the input samples :

$$\mathbf{w} = \sum_{i \in \mathcal{I}} y_i x_i$$

where $\mathcal{I}$ is the set of misclassified inputs that we encounter during learning. At some point in time, in order to test the prediction of the perceptron, we simply compute the dot product of the weights by the vector $\mathbf{x}$ to test :

$$\mathbf{w}.\mathbf{x} = \sum_{i \in \mathcal{I}} y_i x_i.\mathbf{x}$$

and test for the sign of $\mathbf{w}.\mathbf{x}$ to decide whether $\mathbf{x}$ belongs to the positive or negative class. Given the computation are expressed only from dot products, one can extend the algorithm using kernels as in (Freund and Schapire, 1999). Given a mapping function φ of our input space into a so called *feature space* Φ :

$$\begin{array}{rcl} \varphi : \mathbb{R}^d & \rightarrow & \Phi \\ x & \mapsto & X \end{array}$$

The weight vector would then be expressed in the feature space :

$$\mathbf{w} = \sum_{i \in \mathcal{I}} y_i \varphi(x_i)$$

As before, testing an input $\mathbf{x}$ (which is also a step during learning) would imply computing the dot product of the weights by the input, now projected in the feature space $\varphi(x)$:

$$\mathbf{w}.\varphi(\mathbf{x}) = \sum_{i \in \mathcal{I}} y_i \varphi(x_i).\varphi(\mathbf{x}) = \sum_{i \in \mathcal{I}} y_i \mathrm{k}(x_i, \mathbf{x})$$

where k is a kernel (see chapter10 for more details). For example, we show on fig 17.7 an example of binary classification, using RBF kernels with a variance $\sigma = 0.3$, where the perceptron is

trained with the perceptron learning rule. Each class contains 100 samples and convergence was actually obtained by iterating only two times on the training set. Please note that the point of this illustration is to actually to illustrate the application of the perceptron and it is clear that such a classifier does not possess a large margin around the classes which might be revealed by a bad generalization. However, the interested reader can read (Freund and Schapire, 1999) where the voted-perceptron algorithm is introduced, a modification of the perceptron algorithm with guaranteed margins.

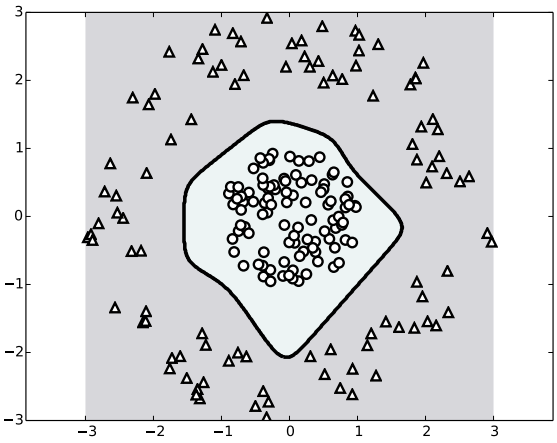


Figure 17.7: Application of the perceptron learning rule with RBF kernels ($\sigma = 0.3$) with 100 samples for both the positive and negative classes. Convergence was obtained in two iterations over the training set.

17.1.2 ADaptive LINear Elements

At the same time (Rosenblatt, 1962) introduced the perceptron, (Widrow and Hoff, 1962) introduced a very similar single layer architecture knonw as the ADaptive LInear Elements (*ADALINE*). While the architecture is similar, the learning algorithm is different. Rather than using the perceptron learning rule, the ADALINE network is trained with the Least-Mean-Square (LMS) algorithm. Suppose we are given a training set for a binary classification problem $S = \{(x_i, y_i) \in \mathbb{R}^d \times \{-1, 1\}, i \in [1..N]\}$, the LMS algorithm looks for weights $\mathbf{w}^\star \in \mathbb{R}^{d+1}$ minimizing the empirical risk :

$$\mathcal{R}^S_{\text{emp}}(\mathbf{w}) = \frac{1}{N}\sum_{i=1}^{N} L(y_i, f_{\mathbf{w}}(x_i)) = \frac{1}{N}\sum_{i=1}^{N} |y_i - f_{\mathbf{w}}(x_i)|_2^{\,2}$$
$$f_{\mathbf{w}}(x) = \mathbf{w}^{\mathsf{T}} \begin{bmatrix} 1 \\ x \end{bmatrix}$$

There are two possibilities to solve this optimization problem. The first possibility is a batch method where all the samples are considered and this least mean square problem can actually be solved analytically by computing its derivative with respect to $\mathbf{w}$ and setting it to zero.

$$\frac{d\mathcal{R}^S_{\text{emp}}}{d\mathbf{w}_j}(\mathbf{w}) = 0$$
$$\Leftrightarrow -\frac{2}{N}\sum_{i=1}^{N}(y_i - \mathbf{w}^{\mathsf{T}} \begin{bmatrix} 1 \\ x_i \end{bmatrix}) \begin{bmatrix} 1 \\ x_i \end{bmatrix} = 0$$
$$\Leftrightarrow \sum_{i=1}^{N}(\mathbf{w}^{\mathsf{T}} \begin{bmatrix} 1 \\ x_i \end{bmatrix}) \begin{bmatrix} 1 \\ x_i \end{bmatrix} = \sum_{i=1}^{N} y_i \begin{bmatrix} 1 \\ x_i \end{bmatrix}$$

Let us now introduce the vector $\mathbf{y}$ with $\mathbf{Y}_i = y_i$, $\mathbf{X} = \begin{bmatrix} 1 & 1 & \cdots & 1 \\ x_1 & x_2 & \cdots & x_N \end{bmatrix}$. We can then rewrite $\sum_{i=1}^{N} yi \begin{bmatrix} 1 \\ x_i \end{bmatrix} = \mathbf{XY}$. For the left-hand side term, let us write $\mathtt{x}_i = \begin{bmatrix} 1 \\ x_i \end{bmatrix}$:

$$\begin{aligned}
\forall j \in [1, d+1], (\sum_{i=1}^{N} (\mathbf{w}^\mathtt{T} \begin{bmatrix} 1 \\ x_i \end{bmatrix}) \begin{bmatrix} 1 \\ x_i \end{bmatrix})_j &= \sum_{i=1}^{N} \sum_{k=1}^{d+1} \mathbf{w}_k (\mathtt{x}_i)_k (\mathtt{x}_i)_j \\
&= \sum_{k=1}^{d+1} \mathbf{w}_k \sum_{i=1}^{N} \mathbf{X}_{k,i} \mathbf{X}_{j,i} \\
&= \sum_{k=1}^{d+1} \mathbf{w}_k \sum_{i=1}^{N} \mathbf{X}_{j,i} {\mathbf{X}^\mathtt{T}}_{i,k} \\
&= \sum_{k=1}^{d+1} \mathbf{w}_k (\mathbf{X}\mathbf{X}^\mathtt{T})_{j,k} = \sum_{k=1}^{d+1} (\mathbf{X}\mathbf{X}^\mathtt{T})_{j,k} \mathbf{w}_k
\end{aligned}$$

And therefore :

$$\sum_{i=1}^{N} (\mathbf{w}^\mathtt{T} \begin{bmatrix} 1 \\ x_i \end{bmatrix}) \begin{bmatrix} 1 \\ x_i \end{bmatrix} = (\mathbf{X}\mathbf{X}^\mathtt{T})\mathbf{w}$$

Finally, the solution to the least square reads :

$$(\mathbf{X}\mathbf{X}^\mathtt{T})\mathbf{w} = \mathbf{X}\mathbf{y} \tag{17.1}$$

which is known as the *normal equations*. If the matrix $\mathbf{X}\mathbf{X}^\mathtt{T}$ is not singular, the solution to the least square problem is :

$$\mathbf{w} = (\mathbf{X}\mathbf{X}^\mathtt{T})^{-1}\mathbf{X}\mathbf{y}$$

and $(\mathbf{X}\mathbf{X}^\mathtt{T})^{-1}\mathbf{X} \in \mathbb{R}^{(d+1)\times N}$ is actually the *Moore-Penrose pseudo-inverse* of $\mathbf{X}$. In case the matrix $\mathbf{X}\mathbf{X}^\mathtt{T}$ is not invertible, there is not unicity of the solution to the least square problem. One can then find the solution $\mathbf{w}$ with the minimal norm. It turns out that this can be computed from the *Singular Value Decomposition* (SVD) of $\mathbf{X}$. The SVD of $\mathbf{X} \in \mathbb{R}^{(d+1)\times N}$ is $\mathbf{X} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^\mathtt{T}$ with $\mathbf{U} \in \mathbb{R}^{(d+1)\times(d+1)}$ and $\mathbf{V} \in \mathbb{R}^{N\times N}$ two orthogonal matrices ($\mathbf{U}^{-1} = \mathbf{U}^\mathtt{T}$, $\mathbf{V}^{-1} = \mathbf{V}^\mathtt{T}$) and $\mathbf{\Sigma}$ is a diagonal matrix with non-negative elements (some can be equal to zeros depending on the rank of the matrix $\mathbf{X}$). The *minimal norm solution* to the least square problem is then defined by (17.2) (Lawson and Hanson, 1974).

$$\mathbf{w} = (\mathbf{V}\mathbf{\Sigma}^{+}\mathbf{U}^\mathtt{T})\mathbf{y} \tag{17.2}$$

with :

$$\mathbf{\Sigma}^{+}_{i,i} = \begin{cases} \frac{1}{\mathbf{\Sigma}_{i,i}} & \text{if } \mathbf{\Sigma}_{i,i} \neq 0 \\ 0 & \text{otherwise} \end{cases}$$

It might not be convenient to solve the optimization problem in a single shot as it requires to compute the pseudo-inverse matrix which grows with the number of samples. Also, the previous method is batch and requires all the samples to be available to compute the optimal solution for the weights. An alternative is to update the parameters $\mathbf{w}$ online, one sample at a time. One simple approach is then to compute the gradient of the loss and to perform a so-called *steepest descent* or *gradient descent*. The derivative can be taken considering the whole training set (*gradient descent*) or only one sample at a time (*Stochastic Gradient Descent* - SGD). If we consider one sample at a time to make the updates online, it reads :

$$\forall i, \nabla_{\mathbf{w}} L\left(y_i, f_{\mathbf{w}}\left(x_i\right)\right) = \frac{d}{d\mathbf{w}} \left| y_i - f_{\mathbf{w}}\left(x_i\right) \right|_2^2 = -2 \frac{df_{\mathbf{w}}}{d\mathbf{w}}\left(x_i\right)\left(y_i - f_{\mathbf{w}}\left(x_i\right)\right) = -x_i\left(y_i - f_{\mathbf{w}}\left(x_i\right)\right)$$

We can then update the weights according to a fraction of the steepest descent. Therefore, at time t, after observing the input/output pair x_i, y_i, the weights would be updated according

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \frac{\alpha}{2}\nabla_{\mathbf{w}} L\left(y_i, f_{\mathbf{w}}\left(x_i\right)\right) = \mathbf{w}_t + \alpha x_i(y_i - f_{\mathbf{w}}\left(x_i\right))$$

This is actually the so-called *delta rule* or Widrow-Hoff rule as considered by (Widrow and Hoff, 1962). Even if (Widrow and Hoff, 1962) originally considered binary classification with its architecture using a linear transfer function and a quadratic loss, as we will see in section17.1.4, different combinations of transfer function and loss are considered depending on the type of problem to be solved (regression or classification).

17.1.3 Limitations

As explained in the previous section, any neural network in which only the last layer contains trainable weights with a binary transfer function can only solve linearly separable binary classification problems. One of the famous example that had a strong negative impact on the research efforts in neural networks is the XOR binary function. The XOR classification problem with two inputs x_1, x_2 is indeed not linearly separable in the x_1, x_2 space. However, if we transform the inputs and work in the $x_1\overline{x_2}, \overline{x_1}x_2$ space, the problem becomes linearly separable (fig. 17.8). However, the question is to determine how the inputs should be transformed so that the problem becomes linearly separable. In other words, how one might learn appropriate features computed from the inputs so that a classification (or regression) problem becomes solvable.

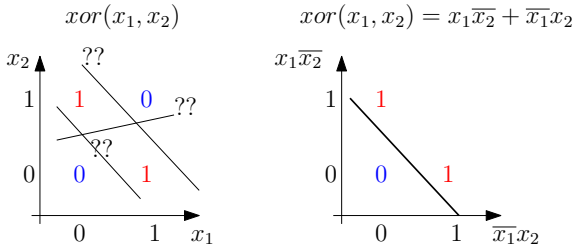


Figure 17.8: The XOR boolean function $x_1 \oplus x_2$ is not linearly separable in the x_1, x_2 space but becomes linearly separable when projected into the $x_1\overline{x_2}, \overline{x_1}x_2$ space.

In section 17.1.1, we saw an example of perceptron with appropriately chosen basis functions which performs a non-linear classification. In the section 17.2, we study a particular type of "single-layer" neural network, the radial basis function networks, in which appropriate choose of features computed from the inputs allow to solve non-linear regression. Actually, the limitation of these networks is not that the perceptron can only represent linearly separable problems; the true question is how to learn the appropriate features and this is what we will in the section on multilayer perceptrons.

17.1.4 Single layer perceptron

In the previous sections, we introduced both the perceptron and Adaline networks from an historical perspective in the sense that our presentation sticks to the architecture introduced respectively by (Rosenblatt, 1962) and (Widrow and Hoff, 1962). We now inspect the question of single layer neural networks from a different perspective by considering which architecture one might use in order to solve regression or classification problems.

Regression

Suppose we are given a monodimensional regression problem $S = \{(x_i, y_i) \in \mathbb{R}^d \times \mathbb{R}, i \in [1..N]\}$. In that case, one would use a linear transfer function $g(x) = x$ and a quadratic loss, i.e.:

$$L\left(y, h_{\mathbf{w}}\left(x\right)\right) = \left|y - f_{\mathbf{w}}\left(x\right)\right|_2^2$$
$$f_{\mathbf{w}}\left(x\right) = \mathbf{w}^{\mathrm{T}}\begin{bmatrix}1\\x\end{bmatrix}$$

The empirical risk to be minimized therefore reads :

$$\mathcal{R}^S_{\text{emp}} = \frac{1}{N}\sum_{i=1}^{N}\left|y_i - \mathbf{w}^{\mathrm{T}}\begin{bmatrix}1\\x_i\end{bmatrix}\right|_2^2$$

As detailed in section 17.1.2, the empirical risk can be minimized in batch mode, using all the training set and the optimal weights $\mathbf{w}^\star \in \mathbb{R}^{d+1}$ are given by solving a linear least square problem and given by the equations (17.1) or (17.2) depending on whether or not $\mathbf{X}\mathbf{X}^{\mathrm{T}}$ is invertible, with $\mathbf{X} = \begin{bmatrix}1 & 1 & \cdots & 1\\ x_1 & x_2 & \cdots & x_N\end{bmatrix}$. We remind the previous results for completeness :

$$\mathbf{w}^\star = \begin{cases}(\mathbf{X}\mathbf{X}^{\mathrm{T}})^{-1}\mathbf{X}\mathbf{y} & \text{if } \mathbf{X}\mathbf{X}^{\mathrm{T}} \text{ is invertible}\\ (\mathbf{V}\mathbf{\Sigma}^{+}\mathbf{U}^{\mathrm{T}})\mathbf{y} & \text{otherwise; the minimal norm solution}\end{cases}$$

The second possibility to optimize for the weights $\mathbf{w}$ is to perform learning online with the stochastic gradient descent. You can perform gradient using one sample at a time (stochastic gradient), all the samples (batch gradient)[5] or mini-batch gradient considering only a part of the samples at every iteration. For the stochastic gradient descent, given some initial weights $\mathbf{w_0}$, the update rule is :

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \frac{\alpha}{2}\nabla_{\mathbf{w}} L\left(y_i, f_{\mathbf{w}}\left(x_i\right)\right) = \mathbf{w}_t + \alpha x_i(y_i - f_{\mathbf{w}}\left(x_i\right))$$

where α is learning rate to be defined (pretty small if fixed, i.e. $\alpha \approx 10^{-2}, 10^{-3}$, or adaptive as we will see later in this chapter). Mini-batches can be meaningful if you use parallel processors (e.g. GPUs) as you actually compute the gradient for several samples with the same weights and can then use more efficiently the parallelism of the hardware.

Binary classification

Let us now consider binary classification problems : we are given $S = \{(x_i, y_i) \in \mathbb{R}^d \times \{0, 1\}, i \in [1..N]\}$. For learning a classifier, one can actually devise several architectures and associated learning algorithms but some are more appropriate than others. The first option we consider is to use the logistic transfer function[6] $g(x) = \frac{1}{1+\exp(-x)}$ which allows to interpret the output as the conditional probability of belonging to one of the class (as $g(x) \in [0, 1]$) given an input. In this situation the quadratic loss is not appropriate (see at the end of this paragraph why) and the cross-entropy loss should be preferred :

$$L\left(y, \hat{y}\right) = -y\ln\left(\hat{y}\right) - (1-y)\ln\left(1-\hat{y}\right)$$
$$\mathcal{R}^S_{\text{emp}}\left(\mathbf{w}\right) = -\frac{1}{N}\sum_{i=1}^{N}(y_i\ln\left(f_{\mathbf{w}}\left(x_i\right)\right) + (1-y_i)\ln\left(1-f_{\mathbf{w}}\left(x_i\right)\right))$$
$$f_{\mathbf{w}}\left(x\right) = g(\mathbf{w}^{\mathrm{T}}\begin{bmatrix}1\\x_i\end{bmatrix})$$
$$g(x) = \frac{1}{1+\exp(-x)}$$

[5] Please note that this is actually meaningless to perform a batch gradient in this situation as the optimal weights can be analytically solved. For multilayer perceptrons, it makes much more sense.

[6] In practice, (LeCun et al., 1998) suggests to use a scaled hyperbolic tangent transfer function $g(x) = 1.7159\tanh\left(0.6666x\right)$

As $y, \hat{y} \in \{0,1\}$, we can note that $L(y,\hat{y}) \geq 0$. Also, $\forall y \in \{0,1\}, L(y,\hat{y}) = 0 \Leftrightarrow \hat{y} = y$. We will now compute the gradient of the loss with respect to the weights. A few preliminaries will be helpful :

$$\begin{aligned}\forall x, g'(x) &= \frac{\exp(-x)}{(1+\exp(-x))^2} = \frac{1}{1+\exp(-x)}(1 - \frac{1}{1+\exp(-x)}) = g(x)(1-g(x)) \\ \forall y, \forall \hat{y}, \frac{\partial L}{d\hat{y}}(y,\hat{y}) &= \frac{\hat{y}-y}{\hat{y}(1-\hat{y})}\end{aligned}$$

Let us denote $\mathtt{x}_i = \begin{bmatrix} 1 \\ x_i \end{bmatrix}$. We then get :

$$\begin{aligned}\forall i, \frac{\partial L}{d\mathbf{w}}\left(y_i, g\left(\mathbf{w}^\mathtt{T}\mathtt{x}_i\right)\right) &= \mathtt{x}_i g'\left(\mathbf{w}^\mathtt{T}\mathtt{x}_i\right) \frac{\partial L}{d\hat{y}}\left(y_i, g\left(\mathbf{w}^\mathtt{T}\mathtt{x}_i\right)\right) \\ &= \mathtt{x}_i(g\left(\mathbf{w}^\mathtt{T}\mathtt{x}_i\right) - y_i)\end{aligned}$$

So, updating the weights with the stochastic gradients lead to :

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \alpha \nabla_\mathbf{w} L\left(y_i, f_\mathbf{w}\left(x_i\right)\right) = \mathbf{w}_t + \alpha \mathtt{x}_i (y_i - g\left(\mathbf{w}^\mathtt{T}\mathtt{x}_i\right))$$

which is actually very similar to the update when considering a linear transfer function with a quadratic loss for the regression problem we considered previously. The transfer function is taken to be the logistic function $\sigma(x) = \frac{1}{1+\exp(-x)}$. If we were to use, for example, the hyperbolic tangent $\tanh x = \frac{\exp(x)-\exp(-x)}{\exp(x)+\exp(-x)}$ one has to adapt the loss accordingly taking into account the fact that the hyperbolic tangent is linearly linked to the logistic as $\tanh(x) = 2\sigma(2x) - 1$. The outputs must also be defined in $\{-1, 1\}$.

What is going on if, rather than the cross entropy loss, we take the quadratic loss but still with the logistic transfer function ? Performing the computation, we get :

$$\forall i, \frac{d}{d\mathbf{w}} \left| y_i - g\left(\mathbf{w}^\mathtt{T}\mathtt{x}_i\right) \right|_2^{\,2} = -2x_i(y_i - g\left(\mathbf{w}^\mathtt{T}\mathtt{x}_i\right))g'\left(\mathbf{w}^\mathtt{T}\mathtt{x}_i\right)$$

We see that the gradient of the quadratic cost does keep a $g'\left(\mathbf{w}^\mathtt{T}\mathtt{x}_i\right)$ term which was cancelled out when using the cross-entropy loss. The issue we then encounter is when, for an input x_i, we get $g\left(\mathbf{w}^\mathtt{T}\mathtt{x}_i\right) \approx 0$ or $g\left(\mathbf{w}^\mathtt{T}\mathtt{x}_i\right) \approx 1$ where the derivative of the logistic function is close to zero. This is the case for example when an input is misclassified and the initial weights sufficiently strong to bring the logistic function in its saturated part. In this case, the gradient is really flat and it will take quite a long time for the parameters to escape from this region.

Multiclass classification

In the case of a classification problem with c classes $S = \{(x_i, y_i) \in \mathbb{R}^d \times \{0, \cdots c-1\}, i \in [1..N]\}$, we would encode the output with the 1-of-c or one-hot encoding, i.e. the size of the output $\mathbf{y}$ is number of classes and we set $\mathbf{y}_i = \delta_{i,y_i}$. We can then devise two architectures. The first one is take **sigmoidal transfer function** for the output layer and use the **cross-entropy loss** applied

to all the predicted output and output dimensions :

$$
\begin{aligned}
L\left(y,\hat{y}\right) &= -\sum_{k=0}^{c-1}(y_k \ln\left(\hat{y}_k\right) + (1-y_k)\ln\left(1-\hat{y}_k\right)) \\
\mathcal{R}_{\text{emp}}^{S}\left(\mathbf{W}\right) &= -\frac{1}{N}\sum_{i=1}^{N} L\left(y_i, f_{\mathbf{W}}\left(x_i\right)\right) \\
f_{\mathbf{W}}\left(x\right) &= g(\mathbf{W}^{\intercal}\begin{bmatrix}1\\x_i\end{bmatrix}) = \begin{bmatrix} g\left(\mathbf{w}_0^{\intercal}\begin{bmatrix}1\\x_i\end{bmatrix}\right) \\ g\left(\mathbf{w}_1^{\intercal}\begin{bmatrix}1\\x_i\end{bmatrix}\right) \\ \vdots \\ g\left(\mathbf{w}_{c-1}^{\intercal}\begin{bmatrix}1\\x_i\end{bmatrix}\right) \end{bmatrix} \\
g(x) &= \frac{1}{1+\exp(-x)}
\end{aligned}
$$

where g is applied element-wise and $\mathbf{W}$ is now a $d+1 \times c$ matrix with the weights to each of the c output units in its columns. One can then verify that the derivative of the loss with respect to any weight $\mathbf{w}_{k,j}$ reads :

$$
\frac{\partial L\left(\mathbf{y}, f_{\mathbf{W}}\left(x\right)\right)}{\partial \mathbf{w}_{k,j}} = (-\mathbf{y}_k + g\left(\mathbf{w}_k^{\intercal}\begin{bmatrix}1\\x_i\end{bmatrix}\right))\mathbf{x}_j
$$

In this case, we cannot interpret the outputs as any discrete probability distribution as they are not normalized. If one wants to interpret the outputs as the conditional probability over the labels given the inputs, we can guarantee that the outputs are in the range $[0,1]$ and sums up to 1 by using the **soft-max transfer function**. Denoting $\mathbf{W}$ the weight matrix where the j-th column $\mathbf{W}_{.,j} = \mathbf{w}_j$ contains the weights from the input to the j-th output, given an input x_i it is handful to introduce the notation :

$$
\mathbf{a}_j = \mathbf{w}_j^{\intercal} x_i
$$

The predicted outputs then read :

$$
\forall j \in [0, c-1], \hat{y}_j = \frac{\exp(\mathbf{a}_j)}{\sum_k \exp(\mathbf{a}_k)}
$$

In this case, the appropriate loss is the **negative log-likelihood loss** defined as :

$$
L\left(y,\hat{y}\right) = -\log(\hat{y}_y)
$$

This supposes that y is the class number. In case y is encoding the class with the 1-of-c or one hot encoding, then you just get the cross-entropy loss $-\sum_{k=0}^{c-1} y_k \log(\hat{y}_k)$. If we write the empirical risk function of the parameter matrix $\mathbf{W}$, denoting $\mathbf{w}_j$ its j-th column :

$$
\begin{aligned}
\mathcal{R}_{\text{emp}}^{S}\left(\mathbf{W}\right) &= -\frac{1}{N}\sum_{i=1}^{N} L\left(y_i, f_{\mathbf{W}}\left(x_i\right)\right) \\
&= -\frac{1}{N}\sum_{i=1}^{N}\sum_{k=0}^{c-1} \mathbb{1}_{y_i=k} \log\left(\frac{\exp(\mathbf{w}_k^{\intercal} x_i)}{\sum_{l=0}^{c-1}\exp(\mathbf{w}_l^{\intercal} x_i)}\right)
\end{aligned}
$$

Here, the derivatives are a bit more tedious to compute. Let us compute some intermediate steps :

$$\forall \mathbf{x}, \forall k, j \forall i \neq k, \frac{\partial}{\partial \mathbf{w}_{i,j}} \log\left(\frac{\exp(\mathbf{w}_k^\intercal \mathbf{x})}{\sum_{l=0}^{c-1}\exp(\mathbf{w}_l^\intercal \mathbf{x})}\right) = -\mathbf{x}_j \frac{\exp(\mathbf{w}_i^\intercal \mathbf{x})}{\sum_{l=0}^{c-1}\exp(\mathbf{w}_l^\intercal \mathbf{x})} = -\mathbf{x}_j [f_{\mathbf{W}}(\mathbf{x})]_i$$

$$\forall \mathbf{x}, \forall k, j, \frac{\partial}{\partial \mathbf{w}_{k,j}} \log\left(\frac{\exp(\mathbf{w}_k^\intercal \mathbf{x})}{\sum_{l=0}^{c-1}\exp(\mathbf{w}_l^\intercal \mathbf{x})}\right) = \mathbf{x}_j - \mathbf{x}_j \frac{\exp(\mathbf{w}_i^\intercal \mathbf{x})}{\sum_{l=0}^{c-1}\exp(\mathbf{w}_l^\intercal \mathbf{x})} = \mathbf{x}_j - \mathbf{x}_j [f_{\mathbf{W}}(\mathbf{x})]_k$$

$$\Rightarrow \forall \mathbf{x}, \forall i, k, j, \frac{\partial}{\partial \mathbf{w}_{i,j}} \log\left(\frac{\exp(\mathbf{w}_k^\intercal \mathbf{x})}{\sum_{l=0}^{c-1}\exp(\mathbf{w}_l^\intercal \mathbf{x})}\right) = \mathbf{x}_j(\delta_{i,k} - [f_{\mathbf{W}}(\mathbf{x})]_i)$$

From these, we deduce the derivatives of the loss :

$$\forall y, \forall \mathbf{x}, \forall i, j, \frac{\partial L}{\partial \mathbf{w}_{i,j}}(y, f_{\mathbf{W}}(\mathbf{x})) = \mathbf{x}_j([f_{\mathbf{W}}(\mathbf{x})]_i - \delta_{y,i})$$

It can be seen that this is actually a generalization of the binary classification with a sigmoidal output and a cross-entropy loss. If we apply the above formula with two outputs but just focusing on the output of class 1 (the positive class), and noting $\forall y \in \{0,1\}, \delta_{y,1} = y$, we get :

$$\forall y \in \{0,1\}, \forall \mathbf{x}, \forall j, \frac{\partial L}{\partial \mathbf{w}_j}(y, f_{\mathbf{W}}(\mathbf{x})) = \mathbf{x}_j([f_{\mathbf{W}}(\mathbf{x})]_1 - y)$$

with $[f_{\mathbf{W}}(\mathbf{x})]_1 = \frac{\exp(\mathbf{w}_1^\intercal \mathbf{x})}{\exp(\mathbf{w}_0^\intercal \mathbf{x}) + \exp(\mathbf{w}_1^\intercal \mathbf{x})} = \frac{1}{1+\exp(-(\mathbf{w}_1 - \mathbf{w}_0)^\intercal \mathbf{x})}$. In the case of multiple classes, the softmax output introduces competition between the different classes which is not the case with logistic outputs. The outputs get normalized and one can interpret the outputs $\hat{\mathbf{y}}_j$ as the conditional probability $\mathrm{P}(y = j \mid x)$.

17.2 Radial Basis Function networks (RBF)

17.2.1 Architecture and training

The radial basis function network (Broomhead and Lowe, 1988) is a type of perceptron in which the basis functions are radial $\phi_j(x) = \exp(-\frac{(x-c_j)^2}{\sigma_j^2})$. In its simplest form, the centers and standard deviations are fixed. To illustrate the RBF networks, consider a single dimensional regression problem $S = \{(x_i, y_i) \in \mathbb{R}^d \times \mathbb{R}, i \in [1..N]\}$. The regressor is defined as :

$$\forall x \in \mathbb{R}^d, f_{\mathbf{w}}(x) = \sum_{k=0}^{K-1} \mathbf{w}_k \phi_k(\mathbf{x}) = \mathbf{w}^\intercal \phi(\mathbf{x})$$

where $\phi(\mathbf{x}) = \begin{bmatrix} \phi_0(\mathbf{x}) \\ \phi_1(\mathbf{x}) \\ \vdots \\ \phi_{K-1}(\mathbf{x}) \end{bmatrix}$, considering K basis functions with one constant function, e.g. $\phi_0(\mathbf{x}) = 1$.
The regression problem can be solved by looking for the weight vector $\mathbf{w}$ minimizing the empirical risk[7] with a quadratic loss :

$$L(y, f_{\mathbf{w}}(x)) = |y - f_{\mathbf{w}}(x)|_2^{\,2}$$
$$f_{\mathbf{w}}(x) = \mathbf{w}^\intercal \phi(\mathbf{x})$$

The empirical risk to be minimized therefore reads :

$$\mathcal{R}_{\text{emp}}^S = \frac{1}{N}\sum_{i=1}^{N} \left|y_i - \mathbf{w}^\intercal \phi(\mathbf{x})\right|_2^{\,2}$$

[7] we shall latter come back on the question of generalization

As we already saw in the previous section, this least square minimization problem can be solved analytically or iteratively with a steepest descent. Analytically, the optimal weights read :

$$\mathbf{w}^{\star} = \begin{cases} \left(\phi\left(\mathbf{X}\right)\phi\left(\mathbf{X}\right)^{\mathrm{T}}\right)^{-1}\mathbf{X}\mathbf{y} & \text{if } \phi\left(\mathbf{X}\right)\phi\left(\mathbf{X}\right)^{\mathrm{T}} \text{ is invertible} \\ (\mathbf{V}\mathbf{\Sigma}^{+}\mathbf{U}^{\mathrm{T}})\mathbf{y} & \text{otherwise; the minimal norm solution} \end{cases}$$

How do we define the centers and standard deviations of the basis functions ? There are actually several possibilities(Schwenker et al., 2001; Peng et al., 2007; Han and Qiao, 2012). The simplest is to pick randomly $K-1$ centers from the inputs and compute a common standard deviations as the mean of the distances between the selected inputs and their closest selected neighbors. We then train/compute the optimal weights to minimize the risk. Another possibility is to apply a clustering algorithm (e.g. k-means) to identify good candidates for the centers and compute the standard deviation as before. Then, after this unsupervised learning step, one would learn in a supervised manner the optimal weights. These are two-phase training algorithms for training RBF (Schwenker et al., 2001). Another possibility is to train the RBF in three phases(Schwenker et al., 2001). The two first phases consists in initializing the centers and standard deviations of the kernels with some clustering algorithm and then compute the optimal weights directly or with a steepest descent. The third phase consist in adapting all the parameters (weights, centers, standard deviations) using a steepest descent. One can actually compute the gradients of the loss with respect to the weights, centers and standard deviations(Schwenker et al., 2001; Bishop, 1995) :

$$\begin{aligned}
L\left(y, f_{\mathbf{w}}\left(x\right)\right) &= \left|y - f_{\mathbf{w}}\left(x\right)\right|_2^{\,2} \\
f_{\mathbf{w}}\left(x\right) &= \mathbf{w}^{\mathrm{T}}\phi\left(\mathbf{x}\right) \\
\forall k, \phi\left(\mathbf{x}\right)_k &= \exp\left(-\frac{\left|\mathbf{x}-\mathbf{c}_k\right|_2}{2{\sigma_k}^2}\right) \\
\forall k, j, \frac{\phi\left(\mathbf{x}\right)_k}{\partial \mathbf{c}_j} &= \delta_{k,j}\phi\left(\mathbf{x}\right)_k \frac{\mathbf{x}-\mathbf{c}_k}{{\sigma_k}^2} \\
\forall k, j, \frac{\phi\left(\mathbf{x}\right)_k}{\partial \sigma_j} &= \delta_{k,j}\phi\left(\mathbf{x}\right)_k \frac{\left|\mathbf{x}-\mathbf{c}_k\right|_2^{\,2}}{{\sigma_k}^3} \\
\frac{\partial L\left(y, f_{\mathbf{w}}\left(x\right)\right)}{\partial \mathbf{w}} &= -2\phi\left(\mathbf{x}\right)\left(y - f_{\mathbf{w}}\left(x\right)\right) \\
\forall k, \frac{\partial L\left(y, f_{\mathbf{w}}\left(x\right)\right)}{\partial \mathbf{c}_k} &= -2\left(y - f_{\mathbf{w}}\left(x\right)\right)\frac{\partial f_{\mathbf{w}}\left(x\right)}{\partial \mathbf{c}_k} = -2\left(y - f_{\mathbf{w}}\left(x\right)\right)\mathbf{w}_k\phi\left(\mathbf{x}\right)_k\frac{\mathbf{x}-\mathbf{c}_k}{{\sigma_k}^2} \\
\forall k, \frac{\partial L\left(y, f_{\mathbf{w}}\left(x\right)\right)}{\partial \sigma_k} &= -2\left(y - f_{\mathbf{w}}\left(x\right)\right)\frac{\partial f_{\mathbf{w}}\left(x\right)}{\partial \sigma_k} = -2\left(y - f_{\mathbf{w}}\left(x\right)\right)\mathbf{w}_k\phi\left(\mathbf{x}\right)_k\frac{\left|\mathbf{x}-\mathbf{c}_k\right|_2^{\,2}}{{\sigma_k}^3}
\end{aligned}$$

Some other algorithms for optimizing both the weights and basis function parameters can be found in(Peng et al., 2007; Han and Qiao, 2012).

17.2.2 Universal approximation

It can be shown that any sufficiently smooth function can be approximated arbitrarily well with a RBF network with a "sufficient" number of kernels. The interested reader is referred to (Park and Sandberg, 1991; Hartman Eric J. et al., 1990).

17.3 Multilayer perceptron (MLP)

17.3.1 Architecture

A multi-layer perceptron is build from one input layer, one output layer and several (≥ 1) hidden layers as depicted on fig 17.9, with $L-1$ hidden layers. The transfer function for the hidden layer output layers are usually taken to be different and are respectively denoted g and f. For example, for a classification problem, whatever the transfer function in the hidden layers, we usually take the softmax transfer function for the output layer which guarantees the outputs $y_i^{(L)}$ define a discrete

probability distribution. For a regression problem, the transfer function of the output is taken as a linear function $f(x) = x$ while non linearities are introduced in the hidden layers.

Let us introduce some notations :

- $\mathbf{w}_{ij}^{(l)}$ the weight between the j-th unit of layer $l-1$ and the i-th unit of layer l,
- $\mathbf{a}_i^{(l)}$ the pre-activation of the unit i of layer l,
- $\mathbf{y}_i^{(l)}$ the output of the unit i of layer l

Every unit computes its pre-activation as a linear combination of its inputs. For simplicity of the notations, we denote $\mathbf{y}_i^{(0)} = x_i$ and $\mathcal{I}_l$ the set of indices of the units in layer $l \in [0, L]$. Remember that, in order to take into the bias (offset) in the linear combination of the input, each layer $l \in [0, L-1]$ has one unit with a constant output equal to 1. This means for example, that if the inputs are taken from $\mathbb{R}^d$, the input layer contains actually $d+1$ units. The computation within the network read :

$$\text{Preactivations:} \forall l \in [1, L], \forall i \in \mathcal{I}_l, \mathbf{a}_i^{(l)} = \sum_{j \in \mathcal{I}_{l-1}} \mathbf{w}_{ij}^{(l)} \mathbf{y}_j^{(l-1)}$$

$$\forall l \in [1, L], \mathbf{a}^{(l)} = \mathbf{W}^{(l)} \mathbf{y}^{(l-1)}$$

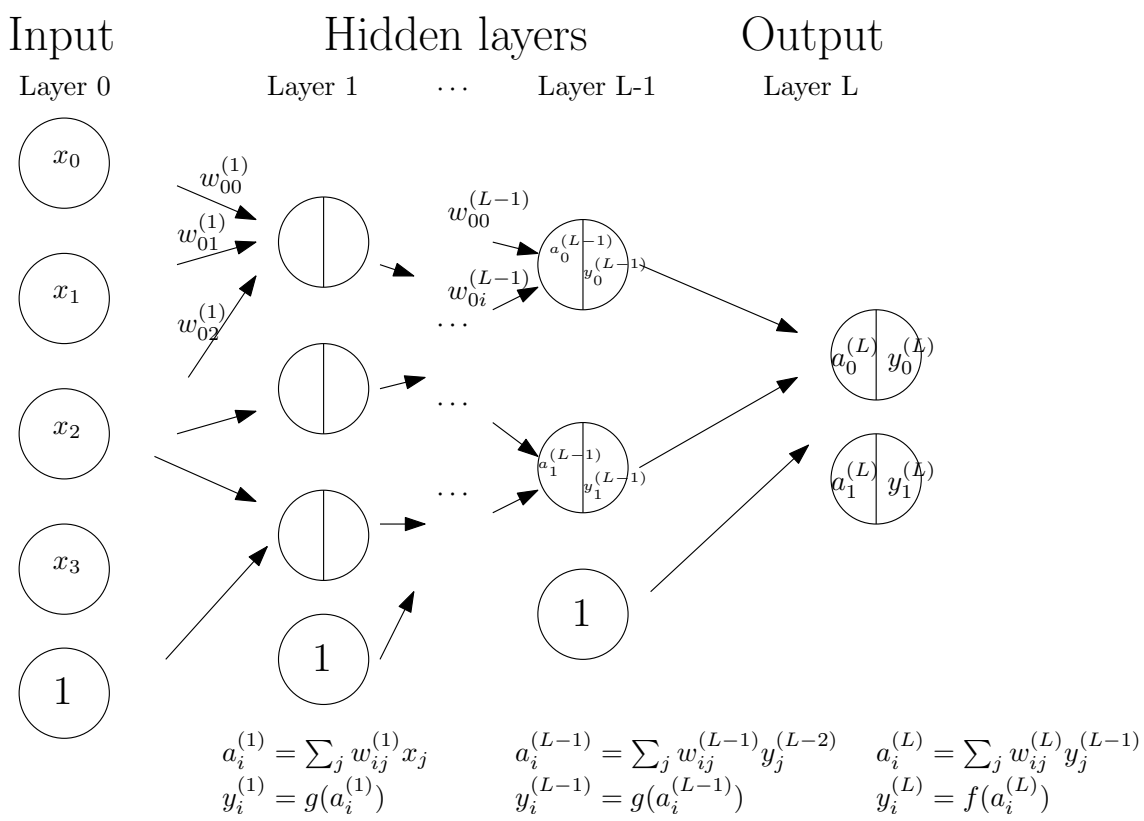


Figure 17.9: A multilayer perceptron is built from an input and output layer with several hidden layers in between. Each layer other than the output is extended with a unit of constant output 1 for the bias.

17.3.2 Learning : error backpropagation

Now that we have introduced the architecture, we need to devise an algorithm to optimize the parameters (the weights) of the neural network and we make a step into the domain of nu-

merical optimization. We can actually resort to any optimization algorithms such as derivative-free optimization algorithms (e.g. line-search, Brent's method(Brent, 1973), black-box optimization such as CMAE-ES(Hansen, 2006), Particle Swarm Optimization(Engelbrecht, 2007; Eberhart and Kennedy, 1995),...), optimization algorithms that make use of the gradient (steepest descent(Werbos, 1981; Rumelhart et al., 1986), natural gradient (Amari, 1998a), conjugate gradient (Johansson et al., 1991), ..) or algorithms that use of the second order derivatives (Hessian) but sometimes only approximating it as in (Martens, 2010). We go back on this topic of optimization algorithms in the section 17.5. For now, let us consider error backpropagation which is historically a major breakthrough in the neural network community as it brought the ability to learn multilayer neural networks(Werbos, 1981; Rumelhart et al., 1986).

We consider architectures for which an appropriate combination of loss function and output transfer function have been chosen. As we saw in section17.1.4, it means :

- for a regression problem with a vectorial output, a linear transfer function and a quadratic loss[8] :

$$f(\mathbf{a}) = \mathbf{a},\ L(\mathbf{y}, \hat{\mathbf{y}}) = \tfrac{1}{2}|\mathbf{y} - \hat{\mathbf{y}}|_2^{\ 2}$$

- for a multi-class classification problem, a softmax output transfer function and the negative log-likelihood loss :

$$f(\mathbf{a}) = \tfrac{1}{\sum_k \exp(a_k)}\left[\exp(a_0) \quad \exp(a_1) \quad \cdots \quad \exp(a_{c-1})\right]^\intercal,\ L(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_k \mathbf{y}_k \log(\hat{\mathbf{y}}_k)$$

Starting from some initial weight and bias vector $\mathbf{w}$, its update following the steepest descent reads :

$$\mathbf{w} = \mathbf{w} - \alpha \nabla_{\mathbf{w}} L$$

Let us compute the derivatives of the loss with respect to a weight from the last hidden layer to the output layer. Denoting $\mathbf{a}^{(\mathbf{L})} = \mathbf{W}^{(L)}\mathbf{y}^{(L-1)}$ the pre-activations of the output layer where $\mathbf{W}^{(L)}$ is the weight matrix from the last hidden layer to the output layer ($\mathbf{W}^{(\mathbf{L})}{}_{i,j}$ is the weight from the hidden unit j to the output unit i), the predicted output can be written as $\hat{\mathbf{y}} = f\left(\mathbf{a}^{(\mathbf{L})}\right)$. In order to compute the gradient to respect to any weight, we shall apply the *chain rule*; in case of a weight $\mathbf{w}_{i,j}^{(L)}$ between the j-th hidden unit and i-th output unit, the gradient of the loss reads :

$$\forall i,j,\ \frac{\partial L(\mathbf{y}, \hat{\mathbf{y}})}{\partial \mathbf{w}_{i,j}^{(L)}} = \sum_k \frac{\partial \mathbf{a}_k^{(L)}}{\partial \mathbf{w}_{i,j}^{(L)}} \frac{\partial L(\mathbf{y}, \hat{\mathbf{y}})}{\partial \mathbf{a}_k^{(L)}}$$

It is convenient to introduce the notation $\mathbf{\Delta}_k^{(L)} = \frac{\partial L(\mathbf{y}, \hat{\mathbf{y}})}{\partial \mathbf{a}_k^{(L)}}$. Therefore, the above formula is written as :

$$\forall i,j,\ \frac{\partial L(\mathbf{y}, \hat{\mathbf{y}})}{\partial \mathbf{w}_{i,j}^{(L)}} = \sum_k \frac{\partial \mathbf{a}_k^{(L)}}{\partial \mathbf{w}_{i,j}^{(L)}} \mathbf{\Delta}_k^{(L)}$$

Whether in the regression or classification, the preactivations are computed as the product of the weight matrix times the output of layer $L-1$: $\mathbf{a}_k^{(L)} = \sum_i \mathbf{w}_{ki}^{(L)} \mathbf{y}_i^{(L-1)}$. Therefore :

$$\frac{\partial \mathbf{a}_k^{(L)}}{\partial \mathbf{w}_{i,j}^{(L)}} = \delta_{i,k} \mathbf{y}_j^{(L-1)}$$

And so, the derivative of the loss reads :

$$\forall i,j,\ \frac{\partial L(\mathbf{y}, \hat{\mathbf{y}})}{\partial \mathbf{w}_{i,j}^{(L)}} = \sum_k \frac{\partial \mathbf{a}_k^{(L)}}{\partial \mathbf{w}_{i,j}^{(L)}} \mathbf{\Delta}_k^{(L)} = \sum_k \delta_{i,k} \mathbf{y}_j^{(L-1)} \mathbf{\Delta}_k^{(L)} = \mathbf{y}_j^{(L-1)} \mathbf{\Delta}_i^{(L)}$$

[8] the $\frac{1}{2}$ in the quadratic loss is introduced to get similar formula than in the classification case. By the way, this is just a scaling factor

We now need to explicit the term $\boldsymbol{\Delta}_i^{(L)}$ which is the derivative of the loss with respect to the pre-activations. The computation are actually similar to the one carried out in section 17.1.4 and repeated here for completeness :

- in case of a regression :

$$\boldsymbol{\Delta}_k^{(L)} = \frac{\partial L(\mathbf{y}, \hat{\mathbf{y}})}{\partial \mathbf{a}_k^{(L)}} = \frac{1}{2}\frac{\partial |\mathbf{y} - \hat{\mathbf{y}}|_2^{\,2}}{\partial \mathbf{a}_k^{(L)}} = \frac{1}{2}\frac{\partial |\mathbf{y} - \mathbf{y}^{(L)}|_2^{\,2}}{\partial \mathbf{a}_k^{(L)}} = \frac{1}{2}\sum_i \frac{(\mathbf{y}_i - \mathbf{y}_i^{(L)})^2}{\partial \mathbf{a}_k^{(L)}} = -\sum_i (\mathbf{y}_i - \mathbf{y}_i^{(L)})\frac{\partial \mathbf{y}_i^{(L)}}{\partial \mathbf{a}_k^{(L)}}$$

$$\begin{aligned}
\frac{\partial \mathbf{y}_i^{(L)}}{\partial \mathbf{a}_k^{(L)}} &= \frac{\partial}{\partial \mathbf{a}_k^{(L)}} f\left(\mathbf{a}_i^{(L)}\right) = \delta_{k,i} \\
\Rightarrow \boldsymbol{\Delta}_k^{(L)} &= -\sum_i (\mathbf{y}_i - \mathbf{y}_i^{(L)})\frac{\partial \mathbf{y}_i^{(L)}}{\partial \mathbf{a}_k^{(L)}} = -\sum_i (\mathbf{y}_i - \mathbf{y}_i^{(L)})\delta_{k,i} \\
&= -(\mathbf{y}_k - \mathbf{y}_k^{(L)}) \\
\Rightarrow \forall i,j, \frac{\partial L(\mathbf{y}, \hat{\mathbf{y}})}{\partial \mathbf{w}_{i,j}^{(L)}} &= \mathbf{y}_j^{(L-1)} \boldsymbol{\Delta}_i^{(L)} = -\mathbf{y}_j^{(L-1)}(\mathbf{y}_i - \mathbf{y}_i^{(L)})
\end{aligned}$$

- in case of a classification, denoting $c(x)$ the class of the input x (i.e. $\forall j, \mathbf{y}_i = \delta_{i,c(x)}$ using 1-of-c encoding of the desired output) :

$$\begin{aligned}
\boldsymbol{\Delta}_k^{(L)} = \frac{\partial L(\mathbf{y}, \hat{\mathbf{y}})}{\partial \mathbf{a}_k^{(L)}} &= -\sum_i \mathbf{y}_i \frac{\partial}{\partial \mathbf{a}_k^{(L)}} \log\left(\mathbf{y}_i^{(L)}\right) = -\sum_i \frac{\mathbf{y}_i}{\mathbf{y}_i^{(L)}}\frac{\partial \mathbf{y}_i^{(L)}}{\partial \mathbf{a}_k^{(L)}} \\
\frac{\partial \mathbf{y}_i^{(L)}}{\partial \mathbf{a}_k^{(L)}} &= \frac{\partial}{\partial \mathbf{a}_k^{(L)}} \frac{\exp(\mathbf{a}_i^{(L)})}{\sum_l \exp(\mathbf{a}_l^{(L)})} = \frac{\delta_{i,k}\exp(\mathbf{a}_i^{(L)})\sum_l \exp(\mathbf{a}_l^{(L)}) - \exp(\mathbf{a}_i^{(L)})\exp(\mathbf{a}_k^{(L)})}{\left(\sum_l \exp(\mathbf{a}_l^{(L)})\right)^2} \\
&= \frac{\exp(\mathbf{a}_i^{(L)})}{\sum_l \exp(\mathbf{a}_l^{(L)})}(\delta_{i,k} - \frac{\exp(\mathbf{a}_k^{(L)})}{\sum_l \exp(\mathbf{a}_l^{(L)})}) \\
&= \mathbf{y}_i^{(L)}(\delta_{i,k} - \mathbf{y}_k^{(L)}) \\
\Rightarrow \boldsymbol{\Delta}_k^{(L)} &= -\sum_i \frac{\mathbf{y_i}}{\mathbf{y}_i^{(L)}}\frac{\partial \mathbf{y}_i^{(L)}}{\partial \mathbf{a}_k^{(L)}} = -\sum_i \mathbf{y}_i(\delta_{i,k} - \mathbf{y}_k^{(L)}) = -(\delta_{c(x),k} - \mathbf{y}_k^{(L)}) \\
&= -(\mathbf{y}_k - \mathbf{y}_k^{(L)}) \\
\Rightarrow \forall i,j, \frac{\partial L(\mathbf{y}, \hat{\mathbf{y}})}{\partial \mathbf{w}_{i,j}^{(L)}} &= \mathbf{y}_j^{(L-1)} \boldsymbol{\Delta}_i^{(L)} = -\mathbf{y}_j^{(L-1)}(\mathbf{y}_i - \mathbf{y}_i^{(L)})
\end{aligned}$$

We now turn to the computation of the derivatives with respect to a weight or bias afferent to a unit in layer $L-1$:

$$\begin{aligned}
\forall i,j, \frac{\partial L(\mathbf{y}, \hat{\mathbf{y}})}{\partial \mathbf{w}_{i,j}^{(L-1)}} &= \sum_k \frac{\partial L(\mathbf{y}, \hat{\mathbf{y}})}{\partial \mathbf{a}_k^{(L-1)}}\frac{\partial \mathbf{a}_k^{(L-1)}}{\mathbf{w}_{i,j}^{(L-1)}} = \sum_k \frac{\partial L(\mathbf{y}, \hat{\mathbf{y}})}{\partial \mathbf{a}_k^{(L-1)}}\delta_{i,k}\mathbf{y}_j^{(L-2)} = \frac{\partial L(\mathbf{y}, \hat{\mathbf{y}})}{\partial \mathbf{a}_i^{(L-1)}}\mathbf{y}_j^{(L-2)} \\
&= \mathbf{y}_j^{(L-2)} \boldsymbol{\Delta}_i^{(L-1)} \\
\boldsymbol{\Delta}_i^{(L-1)} = \frac{\partial L(\mathbf{y}, \hat{\mathbf{y}})}{\partial \mathbf{a}_i^{(L-1)}} &= \sum_k \frac{\partial L(\mathbf{y}, \hat{\mathbf{y}})}{\partial \mathbf{a}_k^{(L)}}\frac{\partial \mathbf{a}_k^{(L)}}{\partial \mathbf{y}_i^{(L-1)}}\frac{\partial \mathbf{y}_i^{(L-1)}}{\partial \mathbf{a}_i^{(L-1)}} = \sum_k \boldsymbol{\Delta}_k^{(L)}\mathbf{w}_{ki}^{(L)} g'(\mathbf{a}_i^{(L-1)}) \\
&= g'(\mathbf{a}_i^{(L-1)})\sum_k \boldsymbol{\Delta}_k^{(L)}\mathbf{w}_{ki}^{(L)}
\end{aligned}$$

If we look at the structure of $\mathbf{\Delta}_i^{(L-1)}$, it is basically the $\mathbf{\Delta}$ term of the next layer weighted by weights of the projection from the unit i to the next layer, everything premultiplied by the derivative of the hidden layer transfer function. Here comes the name "error backpropagation" . The error term $\mathbf{\Delta}_i^{(L)}$ on the output layer is propagated backward in the previous layer. This process is recursive and backpropagation would go downward down to the input layer. We did not detail the derivative of the hidden layer transfer function as it is specific to the network you consider :

- rectified linear units[9] : $g(x) = \max x0$, $g'(x) = \mathbb{1}_{x>0}$
- logistic units : $g(x) = \frac{1}{1+\exp(-x)}$, $g'(x) = g(x)(1 - g(x))$
- hyperpoblic tangent units : $g(x) = \tanh(x)$, $g'(x) = 1 - g(x)^2$

17.3.3 Universal approximator

A single hidden layer neural network with a linear output unit can approximate any continuous function arbitrarily well, given it has a enough hidden units (Hornik et al., 1990; Cybenko, 1989; Hornik et al., 1989). We propose to illustrate with a single dimensional input and output the ability of single hidden layer neural networks to approximate a smooth function. Taking for example a logitstic transfer function :

$$\phi_i(x) = \frac{1}{1 + \exp(-\alpha(x - c_i))}$$

where we just introduce the bias. In our notations, the term inside the exponential is really $\begin{bmatrix} -\alpha c_i \\ \alpha \end{bmatrix}^{\mathrm{T}} \begin{bmatrix} 1 \\ x \end{bmatrix}$. We can then combine two of such sigmoids with different centers and different gain α. Some examples are drawn on fig 17.10. Combining arbitrarily close sigmoids, one can actually build up bell shape functions.

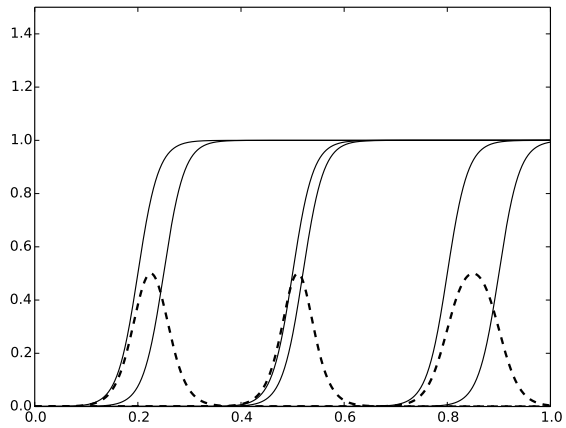


Figure 17.10: By combining two sigmoids with arbitrarily close centers, one can actually build up arbitrarily local bell shape functions which can then be weighted in order to produce any smooth function. The full line plots are six sigmoids which are then grouped by pairs and the difference of these pairs are plotted with dashed lines. For generating the plot, the centers of the pairs are $\{0.2, 0.25\}, \{0.5, 0.52\}, \{0.8, 0.9\}$ with a gain $\alpha = 50$.

Intuitively we reach the point that we can define a bunch of pairs of sigmoids (outputs from the hidden layer) that will create local functions which can then be weighted with the weights from the hidden layer to the output in order to approximate any continuous function. The formal proofs are given in the reference(Hornik et al., 1990; Cybenko, 1989; Hornik et al., 1989).

[9] in which case, one should speak about subgradient

17.3.4 The need for using deep networks

In the previous section, we established that neural networks with a single hidden layer can approximate arbitrarily well any well behaving functions. This result actually convinced people that neural networks is a good class of predictors. However, it turns out that the theorem does not state the size of the hidden layer and it turns out that this number can be actually pretty large(Mhaskar and Micchelli, 1995). Why should we consider a deep (large number of hidden layers) rather than a shallow (few number of hidden layers) neural network ? By using many layers, we can compose many times non-linear functions and can achieve a targeted non-linear mapping without having to consider very large layers and actually, if we were to use few layers we would need a large number of units (Håstad and Goldmann, 1991; Håstad, 1986; Delalleau and Bengio, 2011; Pascanu et al., 2013). These results show that, for some functions, a shallow network may need a number of hidden units growing exponentially with the input while a deep architecture requires a linearly growing number of hidden units. There are also, now, several empirical results indicating the superiority of deep networks (Bengio et al., 2007). These results were permitted after new techniques for training deep neural networks were developed as we will see in the next chapter.

17.4 Generalization

So far, we only discussed about minimizing the empirical risk given some data. However, if only minimize the empirical risk we will definitely (hopefully) perform well on the training set but the generalization performance will be usually bad. This is actually especially true when the dataset is limited. It turns out that some recent works working on very large datasets (or some works augmenting the original dataset by transforming the original dataset as (Ciressan et al., 2012)) do not encounter this issue as working with a very large dataset can be understood as working with an infinite stream of different inputs. However, if the dataset is not "sufficiently" large with respect to the number of degrees of freedom of the network, the network might overfit the training set and perform very badly on data it has not seen in the training set. In this section, we review some popular methods allowing to get hopefully good generalization performances by counterbalancing the minimization of the empirical risk and penalizing models that are too "complex".

To take an example of overfitting. Imagine that you have some 1D data to regress. Suppose that the unknown model is actually linear in the input, say $y = \alpha x$ but obviously, you do not know the model that generated the data as this is what you want to learn from samples, say N samples x_i, y_i. If you were to minimize the empirical risk, you can simply consider the Lagrange polynomials :

$$f(x) = \sum_{i=0}^{N-1} y_i \frac{\prod_{j\in[0,N-1],j\neq i}(x - x_j)}{\prod_{j\in[0,N-1],j\neq i}(x_i - x_j)}$$

This regressor gets perfect fit to the samples if we were to estimate the empirical risk as it is actually null. To avoid too much math, simply suppose that your data are slightly noisy. It is clear that the lagrange polynomial will not result in a linear function of the input since such a linear function would not get a null empirical risk and the lagrange polynomial is actually perfectly fitting the data. You would get higher order monomes which might lead to bad generalization on unseen input/output since you are actually fitting both the data and the noise that you would like to filter out.

17.4.1 Regularization

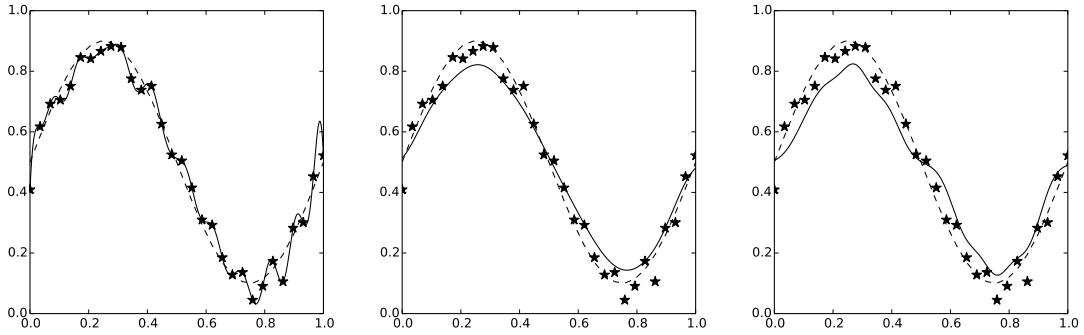
So far, we just speak about minimizing the empirical risk. However, this is not really the quantity of interest. More relevant is the minimization of the real risk which we usually do not have access to[10]. Usually the issue that we may encounter when only focusing on the empirical risk is overfitting where we would perfectly perform on the training set but badly performed on data that were not

[10]there is less and less true as the datasets we are working with are growing, the abundant amount of data may actually prevent overfitting and discard the need for regularizing the neural networks.

present in the dataset, i.e. we would have a bad generalization error. For example, on figure17.11, we generated data from a sine with normally distribution noise :

$$h(x) = 0.5 + 0.4\sin(2\pi x) + \epsilon, \epsilon \sim \mathcal{N}(0, 0.05)$$

Using a RBF with one kernel per sample it turns out the optimal solution to the least square regression is clearly overfitting the data as shown by the learned predictor plotted in solid line on fig. 17.11a.



a) b) c)

Figure 17.11: With 30 samples generated according to eq.17.4.1, and building a RBF with one kernel per sample with a standard deviation $\sigma = 0.05$, fitting the RBF without any penalty leads to an overfitted regressor (a) while fitting the RBF with a weight decay penalty $\lambda = 2$ or L1 norm penalty ($\alpha = 0.005$) provides a better generalization (b,c). The dashed line indicates the noise-free data. Original example from (Bishop, 1995).

Weight decay : L2 norm penalty

The weight decay, or L2 norm penalty consists in adding an extra term to the cost function to be minimized which depends on the norm of the learned weights, **without the biases**. You would then minimize the cost function :

$$J(\mathbf{w}) = L(\mathbf{w}) + \frac{\lambda}{2}\sum_{k=1}^{d+1} {\mathbf{w}_k}^2$$

Performing a gradient descent on this extended cost function is actually simply adding a linear term in the gradient :

$$\begin{aligned}\nabla_{\mathbf{w}} J &= \nabla_{\mathbf{w}} L + \lambda \mathbf{w} \\ \mathbf{w} &\leftarrow \mathbf{w} - \alpha(\nabla_{\mathbf{w}} L + \lambda \mathbf{w}) = (1 - \alpha\lambda)\mathbf{w} - \alpha\nabla_{\mathbf{w}} L\end{aligned}$$

Note that if the predictor is linear and the cost quadratic, adding a L2 penalty simply adds $\lambda\mathbf{I}$ to the $\mathbf{X}\mathbf{X}^\intercal$ matrix to inverse to compute the optimal solution (actually $\lambda\mathbf{I}$ with the first diagonal element set to zero to avoid regularizing the bias). Note also that the bias is not included in the regularization. The L2 penalty will actually enforce the weights to keep a low norm, it will bring $\mathbf{w}$ closer to 0. One may see the L2 penalty as a brake to activate the non-linearities of your network. If one has in mind the RBF network with $f_{\mathbf{w}}(x) = \sum_{k=0}^{d+1} \mathbf{w}_k \phi_k(x)$ we see that if the norm of $\mathbf{w}$ is low, it will tend to prevent activating the non-linearities. An example of RBF with a L2 penalty is shown on fig 17.11b. With this example in mind, we might understand why it is not a good idea to penalize the bias term. The bias term is the mean component of your data. To see this, we can rewrite the cost function to be minimized by expliciting the bias :

$$\underset{\mathbf{w}}{\operatorname{argmin}} \sum_{i=1}^{N} \left| y_i - \mathbf{w}_0 - \sum_{k\geq 1} \mathbf{w}_k \phi_k(x_i) \right|_2^2$$

If you now compute the gradient with respect to $\mathbf{w}_0$ and set it to zero, you will find :

$$\mathbf{w}_0 = \frac{1}{N}\sum_{i=1}^{N} y_i - \sum_{k\geq 1} \mathbf{w}_k \left(\frac{1}{N}\sum_{i=1}^{N} \phi_k\left(x_i\right) \right)$$

It is usually a good idea to standardize the inputs[11] in which case the second term vanishes and you recover the mean of the outputs for the optimal bias. This is one reason why you should not regularize the bias. The penalty should only affect the activation of the non linearities which are the main causes of overfitting.

There is also another idea which helps intuiting why weight decay helps for generalization. Weight decay tends to bring the weights closer to zero. It turns out that if we have a logistic (or a tanh) transfer function, when the weights get small, the preactiviations tend to be where the logistic is almost linear. Therefore, if the weights are constrained to be small, each layer is actually a linear layer and the whole layers of the multilayer perceptrons collapse to a single linear hidden layer. The weights will bring the logistic functions in their saturated part only if it is actually sufficiently decreasing the loss with respect to the weight decay amplitude. That way, we understand weight decay as a penalty on the complexity, richness or expressiveness of a multilayer perceptron.

Sparse weights : L1 norm penalty

The L1 norm penalty consists in adding an extra term to the cost function to be minimized which depends on the absolute value of weights, **without the biases**. You would then minimize the cost function :

$$J\left(\mathbf{w}\right) = L\left(\mathbf{w}\right) + \lambda \sum_{k=1}^{d+1} \left|\mathbf{w}_k\right|$$

Performing a gradient descent on this extended cost function is actually simply adding a term in the gradient which depends on the sign of the components of $\mathbf{w}$:

$$\begin{aligned} \nabla_{\mathbf{w}} J &= \nabla_{\mathbf{w}} L + \lambda \text{sign}\left(\mathbf{w}\right) \\ \mathbf{w} &\leftarrow \mathbf{w} - \alpha(\nabla_{\mathbf{w}} L + \lambda \mathbf{w}) = \mathbf{w} - \alpha\lambda \text{sign}\left(\mathbf{w}\right) - \alpha \nabla_{\mathbf{w}} L \end{aligned}$$

On figure17.12, we give an illustration(Hastie et al., 2009) that helps understanding the influence L1 norm penalty. With the hypothesis that our loss is quadratic, the L1 norm penalty tends to favor solutions that are more aligned with the axis than the L2 penalty which leads to weights that are sparse (i.e. more components get equal to 0). On figure 17.11c, a RBF is regressed with 30 gaussian kernels and a L1 penalty with $\alpha = 0.005$. The linear regression with L1 penalty is solved with the LASSO-Lars algorithm[12]. It turns out that of the 30 basis functions, only 10 get activated. The norm of the optimal solution of the predictor plotted on the figure is actually around $\left|\mathbf{w}^\star\right|_2 \approx 0.57$, actually quite close to the norm of the optimal solution with the L2 penalty $\left|\mathbf{w}^\star\right|_2 \approx 0.61$ but the solution is much sparser.

17.4.2 Learning procedure

The two methods introduced in the previous section add extra terms to the cost to be minimized. In this section, we actually consider variations of the learning procedure.

Dropout

Dropout is a regularizing technique introduced in (Srivastava et al., 2014) and illustrated on fig 17.13. The motivation is actually quite interesting. It is based on the idea of avoiding co-adaptation. It means that, in order to enforce the hidden units of a MLP to learn sound and

[11] a gradient descent has better performance because the cost function is more circular. If you do not standardize the inputs, the cost function might be elongated and the gradient descent would take longer to converge

[12] implemented using the Lasso-Lars algorithm in scikit-learn `http://scikit-learn.org`.

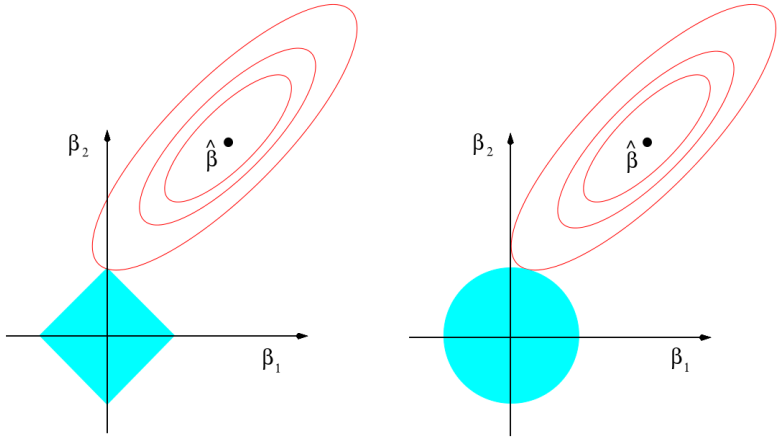


Figure 17.12: The L1 penalty tends to promote solutions that have few non null components. Indeed, compared to the L2 penalty, the optimal solution with the L1 penalty will be more axis aligned. The figure is taken from (Hastie et al., 2009).

robust features, one will actually discard some of its feedforward inputs during training. Discarding is controlled by a binary gate tossed for each input connection following a Bernoulli distribution of parameter p. By doing so, the units can hardly compensate their failures with the help of the others (co-adaptation) as they tend to work with a random subset of other units. When testing the full network, the probability used to drop out units is multiplying the contribution of the unit therefore averaging the contributions. The authors report in (Srivastava et al., 2014) that using $p = 20\%$ or $p = 50\%$ significantly improved the generalization performance of various architectures.

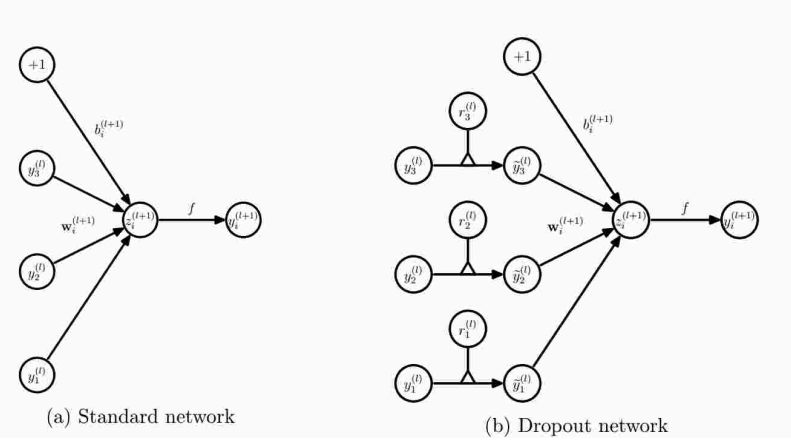


Figure 17.13: Dropout consists in discarding some units during training with a given probability p taken as 20% or 50% as suggested by (Srivastava et al., 2014).Figure taken from (Srivastava et al., 2014)

Early stopping

Early stopping consists, in its most naive implementation, in tracking in parallel with the training error, a validation error computed on a subset of the inputs not in the training set (say, 10% of the data). In an ideal situation, one would observe error function of training epoch that look like on figure 17.14a. Initially, both the errors on the training and validation set decrease. At some point in time, however, while the training error goes on decreasing, the validation starts increasing (see (Wang et al., 1993) for a theoretical analysis in a simplified case). This point in time should actually be the point where learning should be stopped in order to avoid overfitting. In practice,

the errors do not strictly follow this ideal picture (see fig. 17.14b)(Prechelt, 1996). One can however still monitor the performances of the neural network on the validation set during training and select at the end of the training period, the weights that led to the lowest error on the validation set.

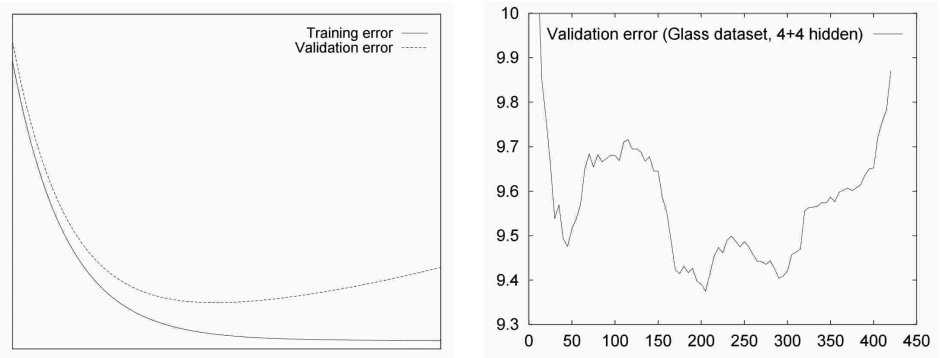


Figure 17.14: a) Ideal training and validation curves which clearly indicate when to stop learning. b) Real cases might actually be much more fluctuating. Images from (Prechelt, 1996)

17.5 Optimization

Several optimization techniques that are not actually specific to neural networks turn out to converge faster than the classical (stochastic) gradient descent. The interested reader is referred to (Bengio et al., 2015), Chap. 8, for an in depth presentation of the aspects such as momentum, first and second order methods (conjugate gradients, hessian free optimizers(Martens, 2010), saddle free optimizers(Dauphin et al., 2014)). There are actually extensive research on understanding the landscape of the cost function we encounter with neural networks and designing specific optimization techniques that take these aspects into account.

17.6 Convolutional neural networks : an early successful deep neural network

One the oldest deep neural network that worked very well was the Neocognitron (Fukushima, 1980) and the convolutional neural networks (Lecun et al., 1998) which both rely on similar principles. These were introduced for classifying images. The basic idea to build up the convolutional neural networks is to take into account the structure of the images, and especially the fact that filters extracting sound features from the local patches on the images are translation invariant. To get this point, imagine you want to detect specific orientations in a gray-scale image using signal processing techniques. You would certainly end up using gabor filters. Actually, you would perform the convolution of your input image with the gabor filter just because the way to detect a specific orientation (computing the gabor on a local patch of the image) is independent of the location where you want to detect that orientation; that is the meaning of translation invariance in the extraction of features. Therefore, the idea of convolutional neural networks is, instead of considering fully connected networks with completely independent weights, to learn filters that are then used to perform convolution over the input (image or hidden layer). This technique of constraining weights of several units to be same is known as *weight sharing*. One example of convolutional neural network is shown on fig 17.15.

A convolutional neural network is built from a stack of convolution and pooling layers. Then at the really top of the stack, some fully connected layers are added. The convolution layers are simply performing a local matrix product between a small patch of the input and a so called filter which is few units in width and height. This filter is actually applied over the whole image as you would perform a usual convolution : this is where the invariance in the feature extraction operates. The feature kernels are actually tensors with the width and height of the filter as well as a depth.

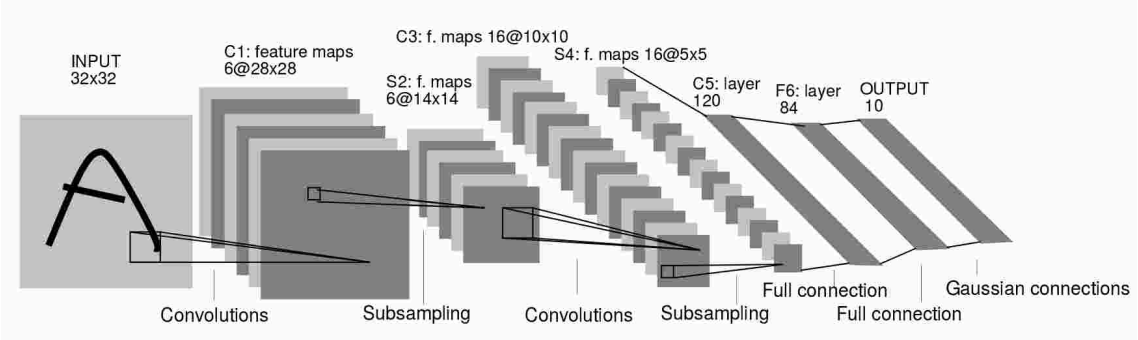


Figure 17.15: A convolutional neural network as introduced in (Lecun et al., 1998). The first layers compute convolutions on their input with several trainable filters. This weight sharing dramatically decrease the number of weights to learn by actually exploiting a fundamental structure of images : the extraction of features from images is translation invariant.

For example, the first convolution layer applied to a RGB image has filters of depth 3. If k filters are computed from the input image, the next convolution layer will have a depth of k. After the convolution layer, one finds a pooling layer. A pooling layer introduces another translation invariance. In its original work, (Lecun et al., 1998) considers subsampling which reads :

$$\mathbf{y}_i^{(l)} = \tanh\left(\beta \sum_{j \in \mathrm{RF}_i} \mathbf{y}_j^{(l-1)} + b\right)$$

Subsampling consists in computing the average of the convolutional layer outputs over a local patch that we call the receptive field. It turns that another pooling operation known as max-pooling works significantly better in practice(Scherer et al., 2010). Max-pooling consists in computing the max rather the average within the receptive field of a unit :

$$\mathbf{y}_i^{(l)} = \max_{j \in \mathrm{RF}_i} \left(\mathbf{y}_i^{(l-1)}\right)$$

Such convolutional neural networks with non-overlapping max-pooling layers appear to be very effective(Ciresan et al., 2011b). In (Simard et al., 2003), the authors present some "good" choices for setting up a convolutional neural network which turn out to work well in practice in terms of initialization of the weights, of the organization of convolutive and fully-connected layers. One additional point the authors present is *data augmentation* which consist in applying distorsions on the training set images in order to feed the network with much more inputs than if we just considered the original dataset. Hopefully, the data augmentation technique should provide additional sensible inputs which mimic the availability of an infinite dataset and therefore might discard the need to regularize the network.

17.7 Autoencoders

Autoencoders Autoencoders (or also known as Diabolo networks) are a specific type of neural network where the objective is to train a network that is able to reconstruct its input. A simple single layer autoencoder is represented on fig 17.16. Usually, the hidden layers have the shape of a bottleneck which smaller and smaller layers with the aim that the input xx will actually get compressed in a so-called code $\mathbf{c}$ being sufficiently informative to allow the reconstruction $\mathbf{x}'$ to be close to the input $\mathbf{x}$. In the simple autoencoder of fig 17.16, the equations read :

$$\mathbf{c} = \sigma\left(\mathbf{W}\mathbf{x} + \mathbf{b}\right)$$
$$\mathbf{x}' = \mathbf{W}'\mathbf{c} + \mathbf{b}'$$

where σ is a transfer function (e.g. logistic). One may constrain the decoding weights $\mathbf{W}'$ to be equal to the transpose of the coding weights $\mathbf{W}$ to decrease the number of parameters to train.

If one is using a linear transfer function and a quadratic cost, one then seek to minimize the reconstruction error from some low dimensional projections which is exactly what PCA is doing. However, when non linear transfer function are used, the auto-encoders do not at all behave as a PCA(Japkowicz et al., 2000) and can be used to extract useful non linear features from the inputs and autoencoders turn out to be effective architectures for performing non linear dimensionality reduction(Hinton et al., 2006).

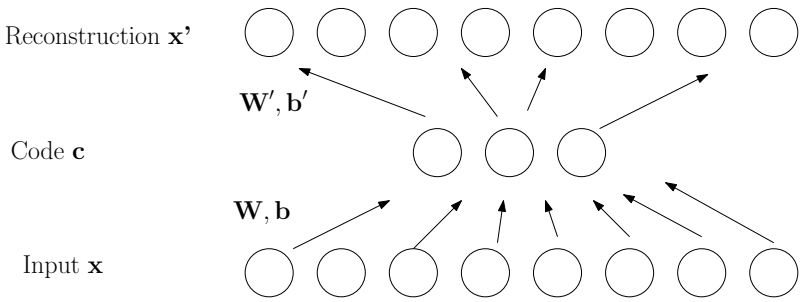


Figure 17.16: A simple single hidden layer autoencoder. The input $\mathbf{x}$ is going through a bottleneck to create a code $\mathbf{c}$ from which we seek to build up a reconstruction $\mathbf{x}'$ as similar as possible to the original input $\mathbf{x}$.

Actually, one may even use a hidden layer with the same number of even more units than the input and still get sensible hidden units that are not merely learning the identity function(Bengio et al., 2007; Ranzato et al., 2006) if the architecture is appropriately regularized (early stopping or L1 norm penalty). In (Hinton et al., 2006), a deep autoencoder with three hidden layers between the input and code layer is introduced to perform dimensionality reduction. The authors also present a way to efficiently train such deep architectures. Variants of the autoencoders where noise, acting as a regularizer, is introduced in the hidden layers are presented in (Vincent et al., 2008). Injecting noise enforce the network to learn robust features and prevent the autoencoder to simply learn the identity function when large hidden layers are considered. These autoencoders are called *denoising autoencoders*. In (Vincent et al., 2008), the authors also introduce the *stacked denoising autoencoder* which are merely a stack of encoders trained iteratively. A first single layer denoising autoencoder is trained. Then, the learned code is used as the input for training a second denoising autoencoder and so on and so forth.

17.8 Where is the problem with a deep neural network and how to alleviate it ?

We already introduced two deep architectures : the convolutional neural networks and the deep autoencoders. Actually, the convolutional neural networks were the only successful deep neural networks until recently. The reason is that the convolutional neural networks are not fully connected. Their shared weights are constrained by several patches of the images. When fully connected feedforward neural networks were considered, they were hard to train. One reason was raised by (Hochreiter, 1991). He actually identified that training is essentially difficult with gradient descent because the gradient tends either to vanish or to blow up as it gets backpropagated in the first layers. Actually, this issue of *vanishing gradient* is shared with the recurrent neural networks. The recurrent neural networks will be introduced in the next chapter but let us just mention few words on them. While feedforward neural networks are acyclic, reccurrent neural networks contain cycle. One way to train them is to actually see each time step of the activities as a layer of a feedforward neural network. Using backpropagation on this neural network actually propagates the gradient through the history of the activities of the network and is called backpropagation through time (BPTT). If several epochs are considered, we actually build up a feedforward neural network with several layers. Gradient descent in recurrent neural networks is known to be difficult (Bengio et al., 1994).

Around 2006, it was found that the issue of gradient vanishing can be alleviated by an appropriate initialization of the network parameters(Hinton et al., 2006; Bengio et al., 2007). A similar idea was already introduced in (Schmidhuber, 1992). The idea is to train in an unsupervised way the feedforward (or recurrent) neural networks. One such method relies on autoencoders that we introduced in a previous section. Remember that autoencoders seek to learn a useful code, useful in the sense that it can be sparse and allows to reconstruct the original data. Robust and sparse features are then extracted from the input image. This unsupervised learning seems to bring the weights of the neural network in a region much more favorable for fine tuning with stochastic gradient descent. One may find additional elements on why training deep networks is difficult in (Glorot and Bengio, 2010).

17.9 Success stories of Deep neural networks

Several recent works push forward the use of neural networks for classification and regression problems, to name a few: image recognition (Ciressan et al., 2012; Krizhevsky et al., 2012), image to text translation (Karpathy and Li, 2014), speech recognition (Graves et al., 2013; Deng and Platt, 2014; Deng et al., 2013; Yu and Deng, 2015), speech translation (Sutskever et al., 2014), drug activity prediction (Dahl et al., 2014). In this section, we review some of these success stories especially focusing on the architectures and training procedures involved.

One famous example of successful deep neural network that we already presented in section 17.6 are the convolutional neural networks LeNet of (Lecun et al., 1998). Recently, it is still deep convolutional neural networks that rank first on the MNIST dataset (Ciressan et al., 2012). In (Ciressan et al., 2012), the authors actually present results on other datasets (e.g. CIFAR, Traffic signs, NORB, chinese characters) but we concentrate here on the MNIST dataset. As a reminder, the MNIST dataset has 10 classes of 28×28 black and white handwritten digits with a training set of 60.000 images and a test set of 10.000 images. In (Ciressan et al., 2012), the authors trained multiple convolutional networks with the same architecture depicted on fig 17.17. The authors applied small distortions to the original dataset (shrinkage) to build up 35 datasets on which they trained a convolutional neural network separately (during training, the images are further distorted before each epoch). They then averaged the response of the 35 neural networks for the final classification. The final architecture is entitled Multicolumn Convolutional Deep Neural Network (MCDNN). The final architecture has up to a million parameters.

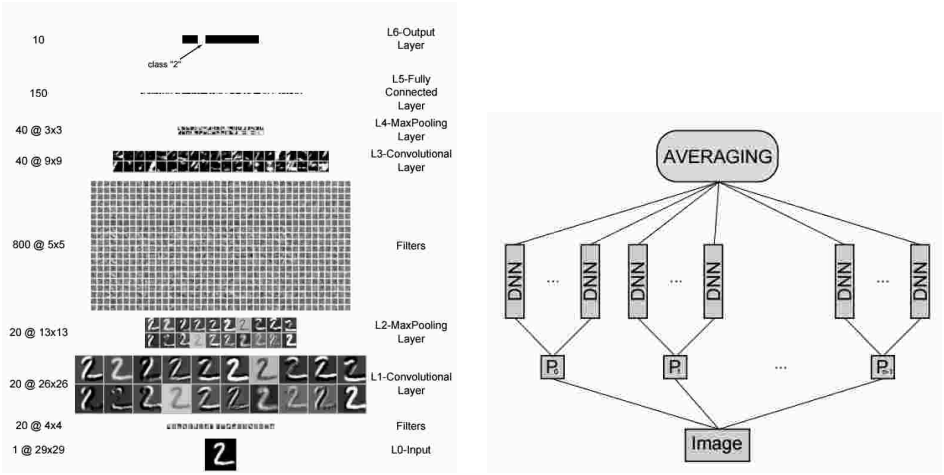


Figure 17.17: a) Convolutional neural network trained on the MNIST dataset by (Ciressan et al., 2012). Applying small distorsions on the original data, the authors build up 35 datasets on which a convolutional neural network is trained separately and then averaged as in (b). Images from(Ciressan et al., 2012).

In different of their works (Cireşan et al., 2010; Ciresan et al., 2011a; Ciressan et al., 2012), the authors use, as a non-linear activation function within the hidden layers of the MLPs(Cireşan et al.,

2010) or within the fully connected and convolutional layers the convolution networks (Ciressan et al., 2012), a scaled hyperbolic tangent suggested by (LeCun et al., 1998) :

$$g(x) = 1.7159\tanh(0.6666x)$$

For the classification output, a softmax is considered. Interestingly, learning is performed using the "good old on-line back-propagation"(Cireşan et al., 2010) without momentum or any other tricks except an exponentially decreasing learning rate schedule and an initialization of the weights uniformly in [-0.05, 0.05]. Note however that is is allowed because fast GPU implementations permit to train the networks during several epochs. As stated by the authors, one convolutional neural network such as on figure17.17a took 14 hours to train on GPUs which would have easily taken a month on a CPU. As reported by the authors, the final architecture ranks first on the MNIST classification :

Network architecture	Misclassification on the test set	Reference
CNN	0.70 %	(Lecun et al., 1998)
CNN	0.40 %	(Simard et al., 2003)
CNN	0.39 %	(Ranzato et al., 2006)
MLP	0.35 %	(Cireşan et al., 2010)
CNN committee	0.27 %	(Ciresan et al., 2011a)
MCDNN	0.23 %	(Ciressan et al., 2012)

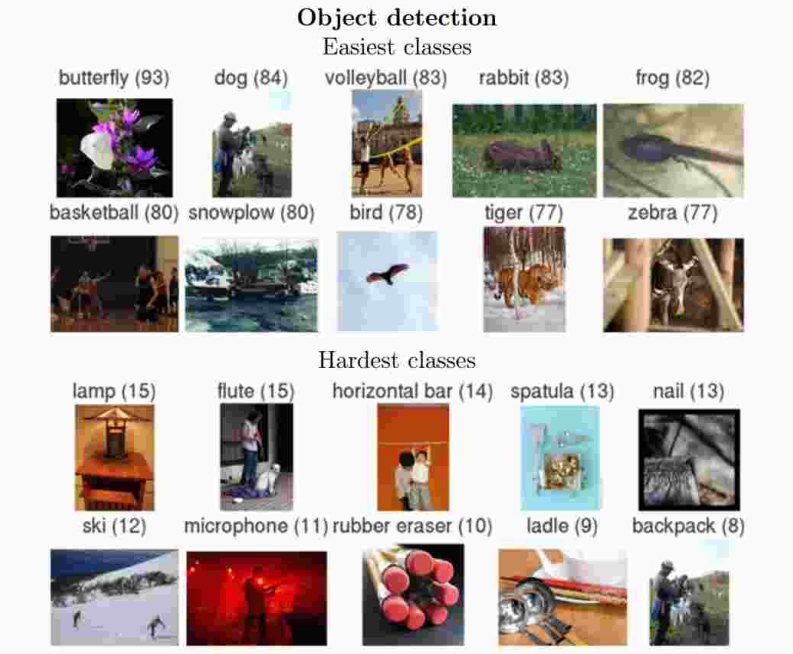


Figure 17.18: Examples of classes from the ImageNet dataset.(Russakovsky et al., 2014).

The SuperVision deep convolutional neural network of (Krizhevsky et al., 2012) ranks first in the ImageNet classification problem(Russakovsky et al., 2014). The 2012 ImageNet challenge consisted in classifying 1000 different categories of objects. The training set was made of 1.2 million image, the validation set of 50.000 images and the test set contained 100.000 images. Some examples of the ImageNet dataset are shown on fig 17.18. The supervision network of (Krizhevsky et al., 2012) is built from 7 hidden layers : 5 convolutional layers and finally 2 fully connected layers. The output layer uses a soft-max transfer function. The hidden layers have a rectified linear transfer function. The total number of parameters to learn reach 60 millions. With so many parameters to learn, the authors proposed to extract random patches of 224×224 pixels from the 256×256 pixels images in order to augment the dataset. Learning uses stochastic gradient descent with dropout (probability of 0.5 toset the output of a hidden unit to 0), momentum of 0.9, weight decay of 0.0005. The weights are initialized according to a specific scheme detailed in (Krizhevsky

et al., 2012) but basically relying on normally distributed weights and unit or zero biases depending on the layer. The learning rate is adapted through a heuristic which consists in dividing it by 10 when the validation error stops improving. According to the authors, it took about a week to train the network on two GPUs involving 90 epochs of the while dataset of 1.2 million images. Recently, (Krizhevsky, 2014) introduces a new way to make training of convolutional neural networks on GPUs faster.

Chapter 18

Recurrent neural networks

18.1 Dealing with temporal data using feedforward networks

Suppose we want to learn a predictor for which the current decision depends on the current input and on the previous inputs. One can actually solve such a task with a feedforward neural network by extending the input layer with, say, n parts each fed with one sample from $x_t, x_{t-1}, x_{t-2}, \cdots x_{t-n+1}$. The input of such a network is called a tapped delay line and the feedforward network built from such an input, a *time-delay neural network* (TDNN)(Waibel et al., 1989).

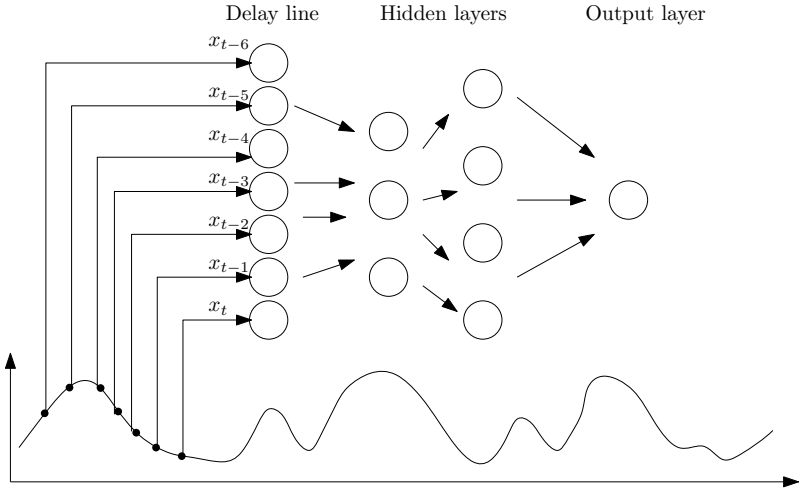


Figure 18.1: A time-delay neural network (TDNN) takes the history into account for its decision by being fed by a sliding window over the input. The main limitation of such a network is that the maximal amount of previous inputs the network can integrate is fixed in the length of the delay line.

The main issue with such a network is that the history that can be used to make the prediction is dependent on the predefined length of the delay line. In addition, the TDNN is using separate weight vectors to extract the information from the samples of the different time steps which is not always optimal, especially if one needs to extract the same piece of information from the input but at different time steps. This means we introduce several weights that must be trained to perform the same work and this might impair generalization.

Rather than relying on a predefined time window over the input, a recurrent neural network can learn to capture and integrate the sufficient amount of information from past inputs in order to make a correct decision.

18.2 General recurrent neural network (RNN)

18.2.1 Architecture

The feedforward neural networks introduced in the previous chapter form an acyclic graph. When ones allow for cycles in the connection graph, one builds up *recurrent neural networks* such as the one depicted on figure 18.2.

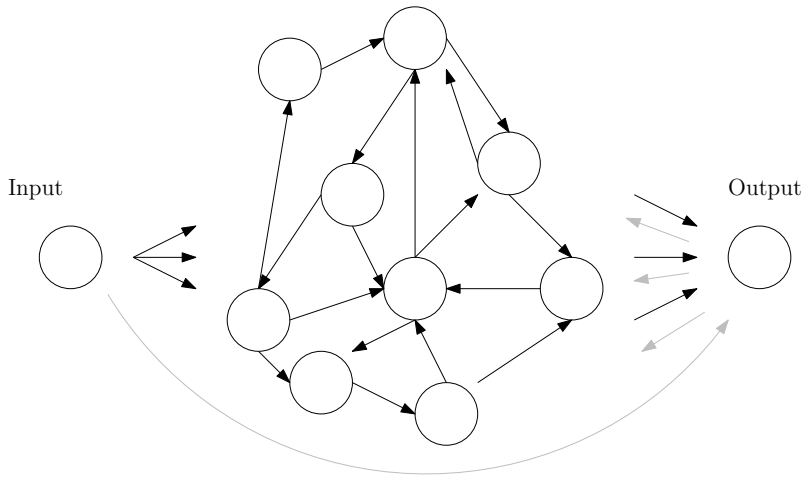


Figure 18.2: A recurrent neural network is a neural network with cycles. The outputs may or may not be fed back to the network. The output is influenced by the hidden units and may also receive direct excitation from the input.

Recurrent neural networks are particularly well suited when working with datasets where the decision requires some form of memory such as for example when working with time series (e.g. speech signals) and hopefully, the recurrent neural network will learn the dependency in time to correctly predict the current output. Depending on the application, it might be required that the output feeds back to the hidden layers as illustrated on fig. 18.2. To describe the computations within a recurrent neural network, we use the same notations as in (Jaeger, 2002) :

- $u_i, i \in [1, K]$ denote the input unit activities ,
- $x_j, j \in [1, N]$ denotes the hidden or internal state activities,
- $y_k, k \in [1, L]$ denotes the output unit activities.

The units are interconnected with weighted connected connections denoted :

- $\mathbf{W}^{in} \in \mathbb{R}^{N \times K}$ the weight matrix from the inputs u_i to the hidden units x_j
- $\mathbf{W}^{back} \in \mathbb{R}^{N \times L}$ the weight matrix from the outputs y_k to the hidden units x_j
- $\mathbf{W} \in Real^{N \times N}$ the weight matrix between the hidden units x_j
- $\mathbf{W}^{out} \in \mathbb{R}^{L \times (K+N)}$ the weight matrix between the input and hidden units to the output units

The biases of the hidden and output units are denoted $\mathbf{b}^x$ and $\mathbf{b}^y$. Similarly to the case of feedforward neural networks, we may use different transfer functions for the hidden units and the output units. Therefore, we introduce the transfer function f and f^{out} for respectively the hidden and output units. We can now write down the equations that rule the activities within the recurrent neural network. The network is supposed to be initialized to some initial state $\mathbf{x}(0) = \mathbf{x}_0, \mathbf{y}(0) = \mathbf{y}_0$:

$$\begin{aligned} \forall t > 0, \mathbf{x}(t) &= f\left(\mathbf{W}^{in}\mathbf{u}(t) + \mathbf{W}\mathbf{x}(t-1) + \mathbf{W}^{back}\mathbf{y}(t-1) + \mathbf{b}^x\right) \\ \mathbf{y}(t) &= f^{out}\left(\mathbf{W}^{out}\begin{bmatrix}\mathbf{u}(t)\\ \mathbf{x}(t)\end{bmatrix} + \mathbf{b}^y\right) \end{aligned} \tag{18.1}$$

18.2.2 Real Time Recurrent Learning (RTRL)

In (Williams and Zipser, 1989), the authors introduced the real time recurrent learning algorithm which is a gradient descent method applied to recurrent neural networks. We will follow in part the presentation given in (Jaeger, 2002). We consider an epoch of T timesteps. A loss is introduced between the outputs $\mathbf{y}(t), t \in [1,T]$ and the desired outputs $\mathbf{d}(t), t \in [1,T]$:

$$L = \sum_{t=1}^{T} L^t = \sum_{t=1}^{T} L(\mathbf{y}(t), \mathbf{d}(t))$$

In order to perform a (possibly stochastic) gradient descent of the loss, we need to compute the derivative of the loss with respect to the weights and biases. All the weights, whether feeding the hidden or output units, contribute to the loss. Without loss of generality, we formulate the derivative with respect to a weight we denote $\mathbf{w}_{k,l}$:

$$\forall \mathbf{w}_{k,l}, \frac{\partial L}{\partial \mathbf{w}_{k,l}} = \sum_{t=1}^{T} \sum_{i=1}^{L} \frac{\partial L^t}{\partial y_i} \frac{\partial y_i(t)}{\partial \mathbf{w}_{k,l}}$$

This requires to compute the derivative of the output activities with respect to the weights and biases. Let us just explicit the derivatives with respect to the weights. From eq. (18.1), these derivatives read :

$$\begin{aligned}
\frac{\partial \mathbf{y}(t)}{\partial \mathbf{w}_{k,l}} &= f^{out\prime}(a_i(t)) \left[\frac{\partial \mathbf{W}^{out}}{\partial \mathbf{w}_{k,l}} \begin{bmatrix} \mathbf{u}(t) \\ \mathbf{x}(t) \end{bmatrix} + \mathbf{W}^{out} \begin{bmatrix} \frac{\partial \mathbf{u}(t)}{\partial \mathbf{w}_{k,l}} \\ \frac{\partial \mathbf{x}(t)}{\partial \mathbf{w}_{k,l}} \end{bmatrix} \right] \\
&= f^{out\prime}(a_i(t)) \left[\frac{\partial \mathbf{W}^{out}}{\partial \mathbf{w}_{k,l}} \begin{bmatrix} \mathbf{u}(t) \\ \mathbf{x}(t) \end{bmatrix} + \mathbf{W}^{out} \begin{bmatrix} \mathbf{0} \\ \frac{\partial \mathbf{x}(t)}{\partial \mathbf{w}_{k,l}} \end{bmatrix} \right] \\
a_i(t) &= \mathbf{W}^{out} \begin{bmatrix} \mathbf{u}(t) \\ \mathbf{x}(t) \end{bmatrix} + \mathbf{b}^y
\end{aligned}$$

We cannot much further without specifying with respect to which weight the derivative is actually computed. Depending on the weight with respect to which the derivative is computed, one would actually get various formula. For example :

$$\frac{\partial \mathbf{y}(t)}{\partial \mathbf{w}^{out}_{k,l}} = f^{out\prime}(a_i(t)) \left[\frac{\partial \mathbf{W}^{out}}{\partial \mathbf{w}^{out}_{k,l}} \begin{bmatrix} \mathbf{u}(t) \\ \mathbf{x}(t) \end{bmatrix} + \mathbf{W}^{out} \begin{bmatrix} \mathbf{0} \\ \frac{\partial \mathbf{x}(t)}{\partial \mathbf{w}^{out}_{k,l}} \end{bmatrix} \right]$$

The matrix $\frac{\partial \mathbf{W}^{out}}{\partial \mathbf{w}^{out}_{k,l}}$ is full of zero with a single 1 line k, column l. If one computes the derivative with respect to a weight feeding the hidden layer, the term $\frac{\partial \mathbf{W}^{out}}{\partial \mathbf{w}_{k,l}}$ vanishes but the other remains. For example :

$$\begin{aligned}
\frac{\partial \mathbf{y}(t)}{\partial \mathbf{w}^{in}_{k,l}} &= f^{out\prime}(a_i(t)) \left[\frac{\partial \mathbf{W}^{out}}{\partial \mathbf{w}^{in}_{k,l}} \begin{bmatrix} \mathbf{u}(t) \\ \mathbf{x}(t) \end{bmatrix} + \mathbf{W}^{out} \begin{bmatrix} \mathbf{0} \\ \frac{\partial \mathbf{x}(t)}{\partial \mathbf{w}^{in}_{k,l}} \end{bmatrix} \right] \\
&= f^{out\prime}(a_i(t)) \mathbf{W}^{out} \begin{bmatrix} \mathbf{0} \\ \frac{\partial \mathbf{x}(t)}{\partial \mathbf{w}^{in}_{k,l}} \end{bmatrix}
\end{aligned}$$

Whatever the weight we consider, the derivatives of the output activities require to compute the derivatives of the hidden layer with respect to the weights as well. Similarly to the derivatives of the output layer activities, one would derive eq (18.1) with respect to the weight and end up with some formula that we can summarize as :

$$\frac{\partial \mathbf{x}(t)}{\partial \mathbf{w}^{in}_{k,l}} = g\left(\frac{\partial \mathbf{x}(t-1)}{\partial \mathbf{w}^{in}_{k,l}}, \frac{\partial \mathbf{y}(t-1)}{\partial \mathbf{w}^{in}_{k,l}}, \mathbf{x}(t-1), \mathbf{y}(t-1), \mathbf{u}(t), \mathbf{W}^{in}, \mathbf{W}, \mathbf{W}^{back}, \mathbf{b}^x, f \right)$$

Note that the derivatives of the output activities $\mathbf{y}(t)$ with respect to some hidden layer weights were dependent on the derivative of the hidden activities $\mathbf{y}(t)$ at time t which themselves are computed from the derivatives of the hidden and output activities at time $t-1$ which overall implies that all the derivatives at time t can be computed from the derivatives at time $t-1$. As the initial state is independent from the weights, the initial conditions read :

$$\frac{\partial \mathbf{x}(0)}{\partial \mathbf{w}_{k,l}} = \mathbf{0}$$
$$\frac{\partial \mathbf{y}(0)}{\partial \mathbf{w}_{k,l}} = \mathbf{0}$$

To compute the gradient of the loss, we need to compute the derivative of all the hidden and output units ($N+L$ units), for every time step (T steps), with respect to every weight ($N^2+2N.L+K.N$ weights) which actually leads to a very expensive computational cost which is the main drawback of this method compared to over methods such as the Backpropagation through time presented in the next section. If the number of hidden units dominates the number of inputs and outputs, the time complexity of one step is an order of N^4. However, RTRL is a forward differentiation method meaning that the derivatives of the loss are computed at each time step and therefore the parameters can be updated online, i.e. at each time step.

18.2.3 Backpropagation Through Time (BPTT)

Backpropagation through time was introduced in (Werbos, 1988). Contrary to RTRL which is a forward differentation algorithm, BPTT requires a forward and backward pass similar to the backpropagation applied in the context of feedforward neural networks. Actually, BPTT is simply the backpropagation applied to the recurrent neural network unfolded in time, where the network at one time step is seen as a layer in a feedforward network of depth T, the number of time steps of an epoch. The unfolding in time of the network is shown on figure 18.3 from (Sutskever, 2013).

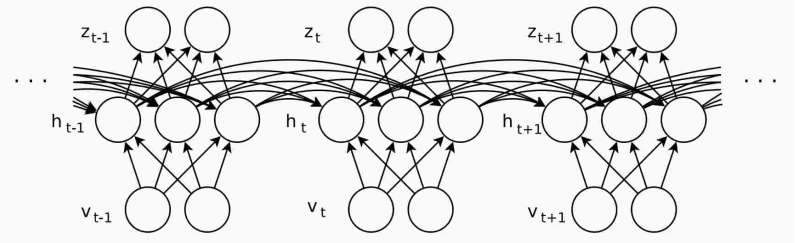


Figure 18.3: Backpropagation through time is backpropagation applied to the recurrent neural network unfolded in time where the network at each time step is considered as one layer of a feedforward neural network of depth T. The notations of the illustration differ slightly from the ones used in the previous section. The figure is from (Sutskever, 2013).

Derivation of the BPTT algorithm is an application of backpropagation to the unfolded network and the reader interested in the equation is referred to (Sutskever, 2013; Jaeger, 2002). Contrary to the RTRL algorithm, BPTT is a batch algorithm which requires to compute the full forward pass before backpropagating the gradient of the loss. However, it is computationally less expensive than RTRL; if the number of hidden units dominate the number of inputs, the time complexity of one epoch requires an order of N^2 operations for the forward and backward pass and therefore the time complexity of one epoch is an order of $T.N^2$.

18.2.4 What about the initial state ?

Because the recurrent neural networks are recurrent, we need to define an initial state for the recurrent neurones, i.e. the afferent units to the ones for which computing the activities at time t requires some activities at time $t-1$. This includes the hidden units but also the output units

when these are fed back to the hidden units. One could could set the initial activities of these units arbitrarily to 0. However, this is not guaranteed to be the optimal choice. Another possibility is to treat the initial state as a variable to be optimized and one can therefore, in a gradient descent method, computes the derivatives of the loss with respect to the initial state and follow the negative gradient.

18.3 Echo state networks

Echo state networks (Jaeger, 2004) are a particular type of recurrent neural networks in which the recurrent weights are predefined and fixed and one only seeks to learn the projection from the hidden layer to the output. An echo state network is depicted on fig 18.4.

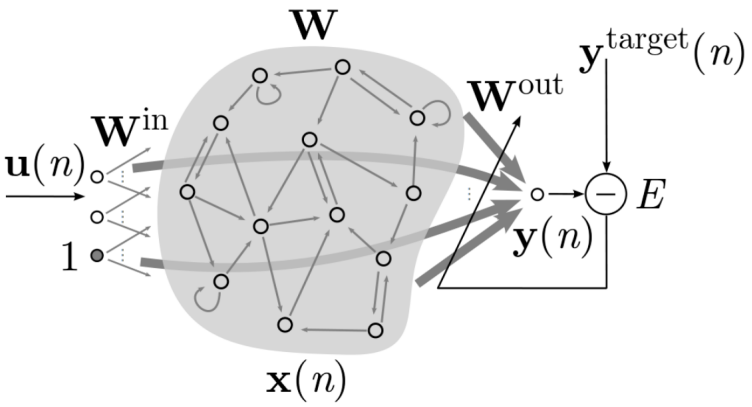


Figure 18.4: In an echo state network (ESN), the recurrent weights are predefined and fixed; only the weights from the hidden to output layers are trained. The figure is from (Lukosevicius, 2012).

The description of echo state networks follows (Lukosevicius, 2012). The hidden units in the ESN are leaky integrators and the output unit activities are linear with respect to the hidden (and possibly input) units. The output units are called readout units. From eq.(18.1), with a linear output transfer function $f^{out}(x) = x$, no feedback connections from the output to the hidden layer $\mathbf{W}^{back} = 0$, the update equations then read :

$$\mathbf{x}(n) = (1-\alpha)\mathbf{x}(n-1) + \alpha \tanh\left(\mathbf{W}^{in}\mathbf{u}(n) + \mathbf{W}\mathbf{x}(n) + \mathbf{b}^{x}\right)$$
$$\mathbf{y}(n) = \mathbf{W}^{out}\begin{bmatrix}\mathbf{u}(n)\\ \mathbf{x}(n)\end{bmatrix} + \mathbf{b}^{y}$$

This is the network one would consider for a regression problem : the fixed predefined recurrent network extracts features from the input stream and a linear readout learns to map the features to the output to regress. In the context of a classification problem, one would use a non linear output transfer function such as the softmax which constrains the output activities to lie in [0,1] and to sum up to 1. Learning the weights from the recurrent network (or reservoir) to the output is not the biggest issue with ESN. In the case of a single output, if the sequence is not too long and the hidden layer not too large, one could compute the optimal weights from the Moore Penrose pseudo inverse, e.g. $\mathbf{w}^{out} == \mathbf{X}\mathbf{X}^{\intercal^{-1}}\mathbf{X}\mathbf{y}$ where $\mathbf{X}$ gathers the hidden states during all time steps and $\mathbf{y}$ the sequence of outputs to predict. One could also apply online learning to the readout weights. Regularization of the readout weights introduced with feedforward neural networks (e.g. L2 penalty) apply in this context as well. The interested reader is referred to (Lukosevicius, 2012) for more information on this.

The size of the reservoir (hidden layer) is usually taken to be as big as possible, hopefully enriching the hidden representation with which the readout is computed. In (Triefenbach et al., 2010), the authors make use of reservoir with 20.000 hidden units.

One big issue with ESN is to be able to define the input to hidden weight matrix $\mathbf{W}^{in}$, the recurrent hidden weight matrix $\mathbf{W}$ and the leaking rate α. The hidden recurrent weight matrix

is usually generated as a sparse matrix as it turns out that it generates in practice better results than dense matrix and numerical computation libraries can compute efficiently operations with sparse matrices which then speed up the evaluation of the network. The input matrix $\mathbf{W}^{in}$ is a dense matrix. Various distributions are used to generate the coefficients of the matrices such as a uniform or gaussian distribution. It is usually advised to scale the hidden recurrent weights $\mathbf{W}$ so that its spectral radius (largest eigenvalue in magnitude) is strictly smaller than 1. It is not always the case that a spectral radius of 1 is optimal (Lukosevicius, 2012). The spectral radius has an influence on the fading of the influence of the inputs on the reservoir activities. If one thinks of the update of the reservoir as a repeated application of the weight matrix $\mathbf{W}$, using a small spectral radius tends to vanish more quickly (exponentially) the input that got integrated at some time step by the reservoir. The leak factor α of the leaky integrator influences how quickly the dynamic of the reservoir evolves. If the input or output time series evolve quickly in few time steps and the leak factor is set too close to 1, the dynamic of the reservoir will not be fast enough as it keeps a strong inertia with its previous state. We will not go further in details on how to setup a reservoir as various elements can be found in (Lukosevicius, 2012; Jaeger, 2002). Actually, all the previous details seem to favor a careful design of the input and hidden layers of the ESN and it seems that much simpler (more constrained) architectures still perform favourably as presented in (Rodan and Tiňo, 2011). To finish this section on ESN, Mantas Lukoševičius is providing source codes on his website (`http://minds.jacobs-university.de/mantasCode`) with implementations of ESN in various programming languages.

18.4 Long Short Term memory (LSTM)

18.4.1 Architecture

Recurrent neural networks share the same vanishing/exploding gradient issue that we encounter with feedforward neural networks because, by nature, recurrent neural networks are deep networks and the issue of vanishing/exploding gradient brakes a recurrent neural network in its ability to capture long term dependencies, i.e. to remember some view of the inputs presented a long time before and required for the current prediction. The analysis carried out in (Hochreiter, 1991) for feedforward neural networks led the same author to introduce in (Hochreiter and Schmidhuber, 1997) a specific recurrent neural networks architecture which does not face the vanishing/exploding gradient issue. The LSTM architecture is built with a specific unit called the memory cell and depicted on fig 18.5. The idea behind the memory cell is to be able to keep a stored information unperturbed during an arbitrary long time period. This helps in learning long term dependencies, i.e. dependency between an input at some time $x_{t-\tau}$ and the output y_t for "long" time delays τ. To do so, the memory cell input and output are controlled by gates which are themselves dependent on the activities of the other units within the network. The original LSTM memory cell introduced by (Hochreiter and Schmidhuber, 1997) is depicted on fig 18.5).

The input and output gates modulate the entrance and "release"[1] of information. This basic memory unit was further extended in (Gers et al., 2000) which introduces a forget gate. Indeed, modification of the content memorized in a memory unit in a LSTM unit is done by imposing a new input and opening the input gate. The motivation of the forget gate introduced in (Gers et al., 2000) is to modify the recurrent pathway gain from $f_t = 1.0$ in the original unit to a smaller positive value (hopefully $f_t = 0.0$ to get a full reset) so that the stored information vanishes.

With the input, output and forget gates, the full equations(Graves, 2013) of the LSTM unit read :

$$
\begin{aligned}
i_t &= \sigma(\mathbf{W}_{xi}x_t + \mathbf{W}_{hi}h_{t-1} + \mathbf{W}_{ci}c_{t-1} + b_i) \\
f_t &= \sigma(\mathbf{W}_{xf}x_t + \mathbf{W}_{hf}h_{t-1} + \mathbf{W}_{cf}c_{t-1} + b_f) \\
c_t &= f_t c_{t-1} + i_t \tanh(\mathbf{W}_{xc}x_t + \mathbf{W}_{hc}h_{t-1} + b_c) \\
o_t &= \sigma(\mathbf{W}_{xo}x_t + \mathbf{W}_{ho}h_{t-1} + \mathbf{W}_{co}c_{t-1} + b_o) \\
h_t &= o_t \tanh(c_t)
\end{aligned}
$$

[1] this is not strictly a release as the information is not dropped from the memory unit when the output gates open but rather influences the other units

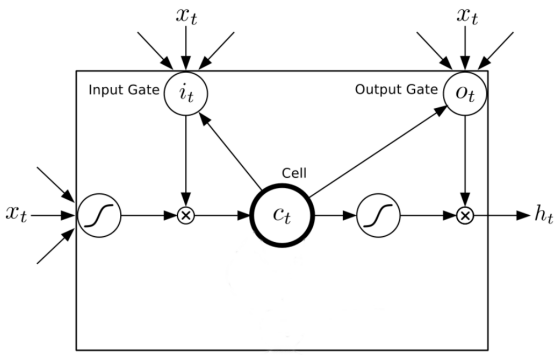


Figure 18.5: The original LSTM memory cell introduced in (Hochreiter and Schmidhuber, 1997) is able to memorize an information. This information is protected from input perturbations provided by the other units within the network with an input. The other units are protected from the influence of the memory cell by an output gate. The network has to learn when to memorize and release a piece of information. LSTM memory units can be arranged to build up memory cells which share their input and output gates. Figure adapted from (Gers et al., 2000).

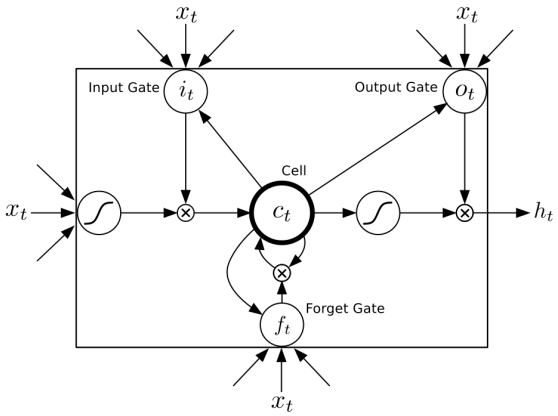


Figure 18.6: Modified LSTM unit with a forget gate which modulates the gain of the recurrent memory feedback pathway which was originally set to 1.0. A full reset would be obtained with a gain $f_t = 0$. The figure is from (Gers et al., 2000).

The above equations simply state that all the hidden and input units contribute to the input, output and forget gating of a unit as well as to define the potential new input to store in the cell. In order to memorize an input, the input gate has to be closed $i_t \approx 0$ and the forget gate open $f_t \approx 1$. To replace the content of the memory cell, it is sufficient to close the forget gate $f_t \approx 0$ and to open the input gate $i_t \approx 1$. In the original LSTM unit, the forget factor was always set to 1.0 and therefore, replacing the content of the memory cell would require some amount of time as, when $f_t = 1$, we get $c_t = c_{t-1} + c^*$. Training of such a network can be performed by applying the algorithms such as Real Time Recurrent Learning or Backpropagation Through Time we introduced in the previous sections.

In some situations as for example in speech processing, it might be helpful to consider both past and future (to some extent) inputs in order to classify the current input. For example, when one is speaking, there are co-articulating effects where the next phoneme to be pronounced influences the end of the previous one. In this context, (Graves and Schmidhuber, 2005) introduced bidirectional LSTM which consists in two LSTM networks processing the input one in the forward direction and the other in the backward direction. The classification at a given can then influenced by both the past and future contexts. There have been also successful works on unsegmented data (Graves et al., 2006) where the network is directly fed with the continuous signals (e.g. the full speech signal) rather than with chunks (e.g. segmented phonemes) that have been segmented in a preprocessing step.

18.4.2 Example of applications

Recurrent neural networks, especially LSTM based neural networks, are one of the most well performing methods on benchmarks involving sequential data(Schmidhuber, 2015). One example of application domain is in dialog systems with benchmarks on, for example, speech synthesis or speech recognition which were usually addressed using Hidden Markov Models. There are also some applications on handwritten texts where the objective is to transcribe a handwritten text into a computer string. In these contexts, bidirectional LSTM networks were applied successfully and reached state of the art performances(Graves et al., 2013; Graves and Schmidhuber, 2005; Graves et al., 2006). Bidirectional LSTM are made of two LSTM networks, one fed with the signal from its beginning and the other fed with the signal from the end. The point of using bidirectional LSTM is to improve the predictor by taking into account both the past context and the future context in order to take into account coarticulation. Coarticulation is the fact that the way a phoneme is pronounced is dependent on the previous and next phonemes to be pronounced. Actually, the same phenomena occurs when one is handwritting. LSTM are also extremely performant to work on unsegmented data. A classic approach to translate a speech signal is to segment it in phonemes that are each individually classified. That might be feasible but actually turn out to be unnecessary with LSTM RNN. When one wants to transcribe a handwritten word, the same issue occurs as the letters are glued together in the word.

Other applications of recurrent neural networks focus on building up generative models(Graves, 2013; Sutskever et al., 2011). In (Sutskever et al., 2011), the authors train a specific RNN architecture that they call the multiplicative RNN where the recurrent connections are gated by learned functions of the inputs. The author train their network to a character prediction task where the network learns to predict characters. Then, the authors apply the network for generating sentences ! The idea is feed the network with a seed, i.e. a small piece of text and then take the most probable output (character) and to feed it back into the network so that it considers it as the next input and then to iterate. Some samples of generated texts are given in (Sutskever et al., 2011) where it is interesting to note that the network was able to capture some sense of English with a pretty good grammar. In (Graves, 2013), the authors train a LSTM network to predict and enerate texts. The author also train LSTM RNN on a handwritting task where the dataset consist in x,y coordinates of a pen used to write a sentence as well as its up/down position. The network is sucessfully trained to predict the next position of the pen. The author goes even further by actually synthesizing handwritten sentences from computer strings. A demo is available on the website of Alex Graves (`http://www.cs.toronto.edu/~graves/handwriting.html`) and illustrated on fig.18.7.

Figure 18.7: A handwritten sentence generated by the recurrent neural network of (Graves, 2013) and fed with the sentence "A LSTM network generating handwritten sentences".

Recently, it has been proposed to combine deep feedforward networks (convolutional neural networks) with recurrent neural networks (bidirectional LSTM) in order to produce captions of images (Karpathy and Li, 2014).

Chapter 19

Energy based models

19.1 Hopfield neural networks

19.1.1 Definition

Hopfield networks are reccurent neural networks with binary threshold units and symmetric connections. If we denote $s_i \in \{-1, 1\}$ the state of neuron i, b_i its biais and $w_{ij} = w_{ji}$ the weight of connections between neurons i and j, the update rule for the state s_i is simply:

$$\forall i, s_i(t+1) = \begin{cases} 1 & \text{if } \sum_j w_{ij} s_j(t) + b_i > 0 \\ s_i(t) & \text{if } \sum_j w_{ij} s_j(t) + b_i = 0 \\ -1 & \text{otherwise} \end{cases} \tag{19.1}$$

We further suppose that there can be self-connections but that the weights of self-connections are restricted to be positive :

$$\forall i, w_{ii} \geq 0$$

One can then define the following energy (*lyapunov*) function[1]:

$$E = -\sum_i s_i b_i - \frac{1}{2} \sum_{i,j \neq i} s_i s_j w_{ij} - \sum_i w_{ii} s_i \tag{19.2}$$

We can rewrite the energy to isolate the terms in which the state of a specific neuron is involved :

$$\forall k, E_k = -s_k b_k - w_{kk} s_k - \sum_{j \neq k} w_{kj} s_k s_j - \frac{1}{2} \sum_{i \neq k} \sum_{j \neq i, j \neq k} s_i s_j w_{ij} - \sum_{i \neq k} s_i b_i - \sum_{i \neq k} w_{ii} s_i$$

From this expression, we can compute the *energy gap*, i.e. the difference in energy when the neuron k is in state 1 and when the neuron k is in state 0 :

$$\begin{aligned} \forall k, \Delta E_k &= E_k(s_k = 1) - E_k(s_k = -1) \\ &= 2(-b_k - w_{kk} - \sum_{j \neq k} w_{kj} s_j) \end{aligned}$$

When updating neuron k, if its state was $s_k(t) = 1$, and the update makes it turn off $s_k(t+1) = -1$, according to eq. (19.1) this means that $\sum_j w_{kj} s_j(t) + b_k = \sum_{j \neq k} w_{kj} s_j + w_{kk} + b_k < 0$. Therefore, $\Delta E_k > 0$. The update produces a modification of the energy of $-\Delta E_k < 0$ and therefore the energy is strictly decreasing. If the neuron was in state $s_k(t) = -1$ and the update makes it turn on $s_k(t) = 1$, this mean that $\sum_j w_{kj} s_j(t) + b_k = \sum_{j \neq k} w_{kj} s_j(t) + b_k > 0$. Given that $-w_{kk} \leq 0$, we have $\Delta E_k < 0$. The update leads to an increase of ΔE_k of the energy and therefore the energy is again strictly decreasing. In case the neuron does not change its state, the energy is constant.

[1] It shall be noted that this formulation of the energy function must be modified if we allow the neuron to change its state when the net input equals 0, in this case, see the work of Floreen et Orponen *Complexity issues in Discrete Hopfield Networks.*

Therefore, **sequential updates make energy function eq.(19.2) decreasing**. The energy function is in general strictly decreasing and is constant if and only if the state of the neuron does not change. Given that there is a finite number of states, we can conclude that the network will converge in a finite number of iterations, the fixed point being a local minima of the energy function.

19.1.2 Example

We consider a Hopfield network with 100 binary neurons with states in $\{-1, 1\}$. The weights are symmetrix and randomly generated in $[-1, 1]$. Self-connections are restricted to be positive (random in $[0, 1]$). The biases are randomly generated in $[-1, 1]$. For each iteration, we randomly choose one of the productive rules (if any), the updates being stopped whenever there is no more productive rule to apply. The energy function of the number of productive rules applied is shown on figure 19.1. On this example, it took 97 iterations before reaching a minimum of the energy function.

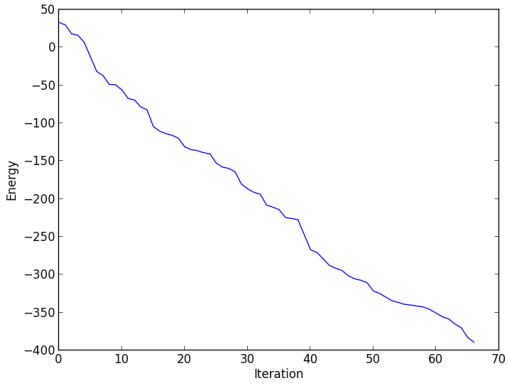


Figure 19.1: Evolution of the energy of a 1D Hopfield network function of the number of productive rules applied

19.1.3 Training

Hopfield suggested that such a network could be used as a memory where patterns to be memorized would be local minima of the energy function. When the network starts from a state *close* to one minimum it will eventually relax to it. This means that, say, a picture might be completely reconstructed from only a subpart of it.

To store a pattern p in a hopfield network, we need to ensure that this pattern p is a minimum of the energy function (19.2) which we can do by a gradient descent of the energy function:

$$\begin{aligned} \Delta w_{ii} &= \alpha p_i \\ \Delta w_{ij} &= \alpha p_i p_j \\ \Delta b_i &= \alpha p_i \end{aligned}$$

If all the patterns to be stored are available, we can set the weights and biases to :

$$\begin{aligned} \mathbf{W} &= \frac{1}{N} \sum_i p_i p_i^T \\ \mathbf{b} &= \frac{1}{N} \sum_i p_i \end{aligned}$$

where p_i is a pattern considered as a column vector.

Example We consider a hopfield neural networks with 100 neurons, their states being either -1 or 1. We first consider a single pattern to be memorized. This pattern is composed of four segments alternatively set to -1 and 1. We show on figure 19.2 the evolution of the states once the weights have been set according to the learning rule (the batch version).

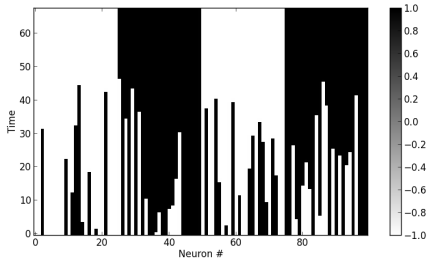


Figure 19.2: Evolution of a Hopfield network trained to store the pattern $(-1,-1,...1,1,..-1,-1,...,1,1)$

It is not shown here, but the learning rule above is somehow specific to the states $-1, 1$. If we use the same learning rule with states $0, 1$, and run the same example, we may keep random values in the domain where the pattern is 0 as the weights and biaises for these neurons equal zero and therefore their state does not leave their initial value.

19.2 Restricted Boltzmann Machines

Boltzmann Machines (Hinton and Sejnowski, 1986) and restricted Boltzmann Machines are stochastic generative neural networks. The feedforward and recurrent neural networks presented in the previous chapters are discriminative neural networks : in a classification context, they learn the conditional probability of the label given the input. The Boltzmann machine and its restricted version are generative models that seek to model the joint probability distribution of the labels and inputs. While discriminiative models can only output the label given an input, the generative models can output the label as well as some input samples by conditioning the learned joint probability by some prior over the labels or over the inputs. A Boltzmann machine is a single layer stochastic neural network. With binary activations, the activation of a unit is sampled from a probability distribution parametrized by the weighted sum of their inputs (plus the bias). In a restricted Boltzmann machine[2], the units are divided into two subgroups so-called visible and hidden units. One motivation in this distinction between hidden and visible units comes when one wants to model some data generated by some unknown processes (some hidden causes). The only thing we get is some observation of the system (the visible units) and we would like to infer the hidden causes of these observations

Denoting v_i the activation of the visible units and h_i the activation of the hidden units, we introduce the energy function associated to a state $\mathbf{v}, \mathbf{h}$ by eq. (19.9).

$$E(v,h) = \sum_i v_i b_i^v + \sum_i h_i b_i^h + \sum_{i<j} v_i h_j w_{ij}^{hv} + \sum_{i<j} v_i v_j w_{ij}^v + \sum_{i<j} h_i h_j w_{ij}^h \tag{19.9}$$

The probability of a joint configuration h, v is defined as :

$$p(v,h) = \frac{e^{-E(v,h)}}{\sum_{u,g} e^{-E(u,g)}}$$

The term $\sum_{u,g} e^{-E(u,g)}$ is called the partition function. The probability of a visible state or of a

[2]historically introduced in (Smolensky, 1986), it was popularized by G. Hinton and collaborators who devised efficient learning algorithms such as the contrastive divergence algorithm(Carreira-Perpiñán and Hinton, 2005)

hidden state is given by :

$$\begin{aligned} p(v) &= \frac{\sum_g e^{-E(v,g)}}{\sum_{u,g} e^{-E(u,g)}} \\ p(h) &= \frac{\sum_u e^{-E(u,h)}}{\sum_{u,g} e^{-E(u,g)}} \end{aligned}$$

In the remaining of this section, we give some elements with respect to restricted boltzmann machines with binary units and the way they can be trained. We refer the reader to (Hinton, 2012) for more details on training RBM. The RBM are not restricted to deal with binary units and variants with real valued activations have been proposed. RBM have been applied successfully to classification where the hidden units can feed a feedforward network to discriminate the class or the label as well as the input can be used as the visible part of the RBM. Finally, RBM have been stacked to build up deep belief networks and deep boltzmann machines (Hinton, 2009; Salakhutdinov and Hinton, 2009).

19.2.1 RBM with binary units

A Restricted Botlzmann Machine is made of two layers called visible and hidden. The visible layer is what is observed from a phenomena while the hidden layer is the hidden causes that generates the observations. Each layer is a set of units denoted $v_i, i \in [0..n-1]$, $h_j, j \in [0..m-1]$ which have binary values $\forall i, v_i \in \{0,1\}, \forall j, h_j \in \{0,1\}$. The visible units are connected to all the hidden units and the hidden units are connected to all the visible units but there is no connection within a layer which gives the name *restricted* to this type of Botlzmann machines. The lack of intra-layer connection will make possible to analytically derive update rules for the network. Such a network is represented on figure 19.3.

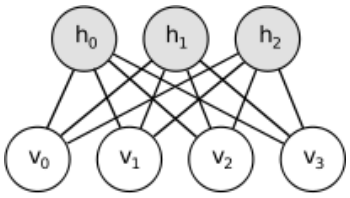


Figure 19.3: Graphical representation of a restricted Boltzmann machine with $n = 4$ visible units and $m = 3$ hidden units.

Let's denote b^v, b^h the biases of respectively the visible and hidden units and w the weight matrix between the visible and hidden units. The weights are symmetric in the sense that if w_{ij} is the weight between the visible unit i and hidden unit j, the hidden unit$j$ is also connected to the visible unit i with the weight w_{ij}. We define an energy function $E(v,h)$ which depends on the state of the visible and hidden units as :

$$E(v,h) = -\sum_i v_i b_i^v - \sum_j h_j b_j^h - \sum_i \sum_j v_i h_j w_{ij} \tag{19.12}$$

From this energy function, we can define a probability over the states of the network as :

$$p(\mathbf{v} = v, \mathbf{h} = h) = \frac{1}{Z} e^{-E(v,h)}$$

where Z is a normalization factor, called the partition function and defined as :

$$Z = \sum_{v,h} e^{-E(v,h)}$$

The random variables $\mathbf{v}$ et $\mathbf{h}$ are discrete and the marginal of $\mathbf{v}$ is defined as :

$$\begin{aligned} p(\mathbf{v}=v) &= \sum_h p(\mathbf{v}=v|\mathbf{h}=h)p(\mathbf{h}=h) \\ &= \sum_h p(\mathbf{v}=v,\mathbf{h}=h) \\ &= \frac{1}{Z}\sum_h e^{-E(v,h)} \end{aligned}$$

We now introduce the *free energy* (which will renders the derivations easier) as :

$$\mathcal{F}(v) = -\log(\sum_h \exp(-E(v,h)))$$

We can then rewrite the marginal of $\mathbf{v}$ and the partition function in terms of the free energy :

$$\begin{aligned} Z &= \sum_v \exp(-\mathcal{F}(v)) \\ p(\mathbf{v}=v) &= \frac{\exp(-\mathcal{F}(v))}{\sum_{v'}\exp(-\mathcal{F}(v'))} \end{aligned}$$

With the expression of the energy function in equation (19.12) the free energy can be written as :

$$\begin{aligned} \mathcal{F}(v) &= -\log(\sum_h \exp(-E(v,h))) \\ &= -\log(\sum_h \exp(\sum_i v_i b_i^v)\exp(\sum_j h_j b_j^h + \sum_i\sum_j w_{ij}v_i h_j)) \\ &= -\log(\prod_i \exp(v_i b_i^v)\sum_h \exp(\sum_j h_j b_j^h + \sum_i\sum_j w_{ij}v_i h_j)) \\ &= -\sum_i v_i b_i^v - \log(\sum_h \exp(\sum_j h_j b_j^h + \sum_{i,j} v_i h_j w_{ij})) \\ &= -\sum_i v_i b_i^v - \log(\sum_h \prod_j \exp(h_j(b_j^h + \sum_i v_i w_{ij}))) \\ &= -\sum_i v_i b_i^v - \log(\prod_j \sum_{h_j} \exp(h_j(b_j^h + \sum_i v_i w_{ij}))) \\ &= -\sum_i v_i b_i^v - \sum_j \log(\sum_{h_j} \exp(h_j(b_j^h + \sum_i v_i w_{ij}))) \end{aligned}$$

Since the hidden units have binary states, the expression can be further simplified :

$$\mathcal{F}(v) = -\sum_i v_i b_i^v - \sum_j \log(1+\exp((b_j^h + \sum_i v_i w_{ij})))$$

We are now interested in the conditional probabilities $p(\mathbf{v}|\mathbf{h})$ and $p(\mathbf{h}|\mathbf{v})$ which define the update

rules of the network. By definition of the conditional probabilities and of the free energy[3]:

$$\begin{aligned}
p(\mathbf{h}=h|\mathbf{v}=v) &= \frac{p(\mathbf{h}=h,\mathbf{v}=v)}{p(\mathbf{v}=v)} \\
&= \frac{e^{-E(v,h)}}{\sum_{h'} e^{-E(v,h')}} \\
&= \frac{\exp(\sum_i v_i b_i^v + \sum_j h_j b_j^h + \sum_{i,j} v_i h_j w_{ij})}{\sum_{h'} \exp(\sum_i v_i b_i^v + \sum_j h'_j b_j^h + \sum_{i,j} v_i h'_j w_{ij})} \\
&= \frac{\exp(\sum_j h_j b_j^h + \sum_{i,j} v_i h_j w_{ij})}{\sum_{h'} \exp(\sum_j h'_j b_j^h + \sum_{i,j} v_i h'_j w_{ij})} \\
&= \frac{\prod_j \exp(h_j(b_j^h + \sum_i v_i w_{ij}))}{\prod_j \sum_{h'_j} \exp(h'_j(b_j^h + \sum_i v_i w_{ij}))} \\
&= \prod_j \frac{\exp(h_j(b_j^h + \sum_i v_i w_{ij}))}{\sum_{h'_j} \exp(h'_j(b_j^h + \sum_i v_i w_{ij}))} = \prod_j p(\mathbf{h}_j = h_j|\mathbf{v}=v)
\end{aligned}$$

By identifying the two last terms, the conditional probabilities of the hidden components then read :

$$p(\mathbf{h}_j = h_j|\mathbf{v}=v) = \frac{\exp(h_j(b_j^h + \sum_i v_i w_{ij}))}{\sum_{h'_j} \exp(h'_j(b_j^h + \sum_i v_i w_{ij}))}$$

Since the hidden units are binary, the update rules finally read :

$$\begin{aligned}
p(\mathbf{h}_j = 1|\mathbf{v}=v) &= \frac{\exp(b_j^h + \sum_i v_i w_{ij})}{1 + \exp(b_j^h + \sum_i v_i w_{ij})} \\
&= \sigma(b_j^h + \sum_i v_i w_{ij})
\end{aligned}$$

with σ the logistic function defined as $\sigma(x) = \frac{1}{1+\exp(-x)}$. By a similar derivation, since the network is symmetric, we find :

$$p(\mathbf{v}_i = 1|\mathbf{h}=h) = \sigma(b_i^v + \sum_j h_j w_{ij})$$

These are classical update rules of neural networks.

19.2.2 Training

We now wish to train the network so that higher probabilities are given to the training samples clamped on the visible units and a lower probability to all the other samples. This will in effect shape the energy landscape in favor of the training samples. We therefore wish to maximise the probability $\sum_d p(\mathbf{v}=\hat{v}_d)$ where $\hat{v}_d$ are the training data. Maximizing this sum which is called the *likelihood* is equivalent to maximizing the sum of the logarithm of the probabilities. This is a technical point simplifying the derivations. The sum of the logarithms of the probabilities is called the *log likelihood* and we define a cost function as :

$$\mathcal{L}(\theta) = -\sum_d \log(p(\mathbf{v}=\hat{v}_d|\theta))$$

with θ the parameter vectors that we now explicitly introduce in the notations to highlight the dependencies of the cost function on the parameters. To compute the gradient of the cost function,

[3]for the last equality, we use the fact that the components of $\mathbf{h}$ are pairwise independent because there is no connection within the hidden layer

we use the expression of the free energy :

$$\begin{aligned}
-\frac{\partial \log p(\mathbf{v}=v_d)}{d\theta} &= \frac{\partial \mathcal{F}(v_d)}{d\theta} + \frac{\partial \log(\sum_{v'} e^{-\mathcal{F}(v')})}{d\theta} \\
&= \frac{\partial \mathcal{F}(v_d)}{d\theta} + \frac{1}{\sum_{v'} e^{-\mathcal{F}(v')}} \sum_{v'} \frac{\partial e^{-\mathcal{F}(v')}}{d\theta} \\
&= \frac{\partial \mathcal{F}(v_d)}{d\theta} - \frac{1}{Z} \sum_{v'} \frac{\partial \mathcal{F}(v')}{d\theta} e^{-\mathcal{F}(v')} \\
&= \frac{\partial \mathcal{F}(v_d)}{d\theta} - \sum_{v'} \frac{\partial \mathcal{F}(v')}{d\theta} p(\mathbf{v}=v') \\
&= \frac{\partial \mathcal{F}(v_d)}{d\theta} - \mathbb{E}_p[\frac{\partial \mathcal{F}(\mathbf{v})}{d\theta}]
\end{aligned}$$

In this expression, we recognize a term depending on the free energy of a training example and a second term which is a sum over all the possible states and that depends on the model. In the vocabulary of the Boltzmann machines, the first term is called the **positive phase** while the second the **negative phase**. These notations do not merely depend on the sign of the expressions but more on the influence of these terms. The first term increases the probability of a training sample (by reducing its free energy), while the second decreases the probablity of the samples produced by the model. This is the combined effect of both terms that will shape the energy landscape so that the model generates our training data.

We now have to see how the two terms can be computed to update the parameters. We begin with the **positive** phase which can be analytically derived by using the expression of the free energy :

$$\begin{aligned}
\frac{\partial \mathcal{F}(v_d)}{d\theta} &= -\sum_i v_{d,i} \frac{\partial b_i^v}{d\theta} - \sum_j \frac{1}{1+\exp(b_j^h + \sum_i v_{d,i} w_{i,j})} \frac{\partial \exp(b_j^h + \sum_i v_{d,i} w_{i,j})}{d\theta} \\
&= -\sum_i v_{d,i} \frac{\partial b_i^v}{d\theta} - \sum_j \frac{\exp(b_j^h + \sum_i v_{d,i} w_{i,j})}{1+\exp(b_j^h + \sum_i v_{d,i} w_{i,j})} \left(\frac{\partial b_j^h}{d\theta} + \sum_i v_{d,i} \frac{\partial w_{ij}}{d\theta} \right) \\
&= -\sum_i v_{d,i} \frac{\partial b_i^v}{d\theta} - \sum_j \sigma(b_j^h + \sum_i v_{d,i} w_{i,j}) \left(\frac{\partial b_j^h}{d\theta} + \sum_i v_{d,i} \frac{\partial w_{ij}}{d\theta} \right) \\
&= -\sum_i v_{d,i} \frac{\partial b_i^v}{d\theta} - \sum_j \sigma(b_j^h + \sum_i v_{d,i} w_{i,j}) \frac{\partial b_j^h}{d\theta} - \sum_{i,j} \sigma(b_j^h + \sum_i v_{d,i} w_{i,j}) v_{d,i} \frac{\partial w_{ij}}{d\theta}
\end{aligned}$$

We can now derive specific expressions for the biases and weights :

$$\begin{aligned}
\forall i, \frac{\partial \mathcal{F}(v_d)}{db_i^v} &= -v_{d,i} \\
\forall j, \frac{\partial \mathcal{F}(v_d)}{db_j^h} &= -\sigma(b_j^h + \sum_i v_{d,i} w_{i,j}) \\
\forall i,j, \frac{\partial \mathcal{F}(v_d)}{dw_{ij}} &= -\sigma(b_j^h + \sum_i v_{d,i} w_{i,j}) v_{d,i}
\end{aligned}$$

We can recognize hebbian like learning rules; the weights are updated by the product of a pre- and post-synaptic term. For the **negative** phase, the main difficulty is that we cannot compute the marginal of the visible units $p(\mathbf{v})$. We can however notice that the mean can be approximated with Monte-Carlo. Suppose that even if we cannot compute the marginal, we can get samples from the model. We would then consider a set $\mathcal{N}$ of samples drawn from $p(\mathbf{v})$ and approximate the mean as :

$$\mathbb{E}_p[\frac{\partial \mathcal{F}(\mathbf{v})}{d\theta}] \approx \frac{1}{|\mathcal{N}|} \sum_{v \in \mathcal{N}} \frac{\partial \mathcal{F}(\mathbf{v})}{d\theta}$$

Since the sum can be computed, we need a method for sampling from $p(\mathbf{v})$. These methods are called *Markov Chain Monte Carlo* where we sample alternatively the visible and hidden units starting from a given sample. We would in principle need several iterations before reaching a so-called *thermal equilibrium*. The thermal equilibrium is a kind of steady state where, even if the states change (because of the stochastic update rules), the probablities from which the states are samples are fixed. While in principle we would need to reach *thermal equilibrium* (and this can actually take a certain unknown amount of iterations), an approximated method called *contrastive divergence* brings pretty good results. In the contrastive divergence, we initialize the visible units to a training sample, then update the hidden layer, the visible layer and the hidden layer again. This is called CD-1 while in general CD-k considers k such updates. The visible state that we sampled is called a *reconstruction*. The negative phase is computed only on the last visible and hidden states.

Part VI

Ensemble methods

Chapter 20

Introduction

In supervised learning, predictions are made based on an estimator built with a given learning algorithm. Ensemble methods aim at combining the predictions of several base estimators in order to improve generalization or robustness over a single estimator.

To get an informal idea of ensemble learning[1], consider Fig. 20.1. The top square corresponds to a classification problem with positive (denoted "+") and negative (denoted "−") examples. As estimators, we consider "decision stumps", that is linear separators that take decision based on a single dimension of the input space (here, they must be vertical or horizontal), as illustrated by the tree squares in the middle. Now, if one takes a weighted combination of such decisions stumps, the classification problem can be solved, as illustrated on the bottom square.

Strictly speaking, most of the ideas presented in the following chapters applies to arbitrary base estimators. However, they are often used with decision trees, so we start by presenting this learning paradigm, before providing an overview of the (non exhaustively, as usual) studied ensemble methods.

20.1 Decision trees

Next, we briefly present *decision trees* (Breiman et al., 1984). This section is largely inspired from[2] Hastie et al. (2009, Ch. 9) and focuses on the CART (Classification and Regression Tree) algorithm, of which `C4.5` (Quinlan, 1993) is a classic competitor.

20.1.1 Basic idea

The idea of decision trees is to partition the input space into a set of rectangles and to fit a simple (constant) model in each partition. As an illustration, consider Fig. 20.2, corresponding to a regression problem with continuous output y and two-dimensional input $x = (x_1, x_2)^\top \in (0,1)^2$. The top right panel shows a partition of the input space made by a recursive binary tree (the top left panel illustrates another partition, but that cannot be obtained with a recursive binary tree). The five regions can be obtained as follows:

- first, construct two regions, depending on $x_1 \leq t_1$ or $x_1 > t_1$;
- then, if $x_1 \leq t_1$, construct two regions, depending on $x_1 \leq t_2$ (region R_1) or $x_2 > t_2$ (region R_2);
- else, if $x_1 > t_1$, construct tow regions, depending on $x_1 \leq t_3$ (region R_3) or $x_1 > t_3$;
- lastly, if $x_1 > t_3$, construct two regions, depending on $x_2 \leq t_4$ (region R_4) or $x_2 > t_4$ (region R_5).

[1] This is more precisely an illustration of boosting.

[2] This book is an excellent general introduction to machine learning and is (legally and freely) available online: http://statweb.stanford.edu/~tibs/ElemStatLearn/.

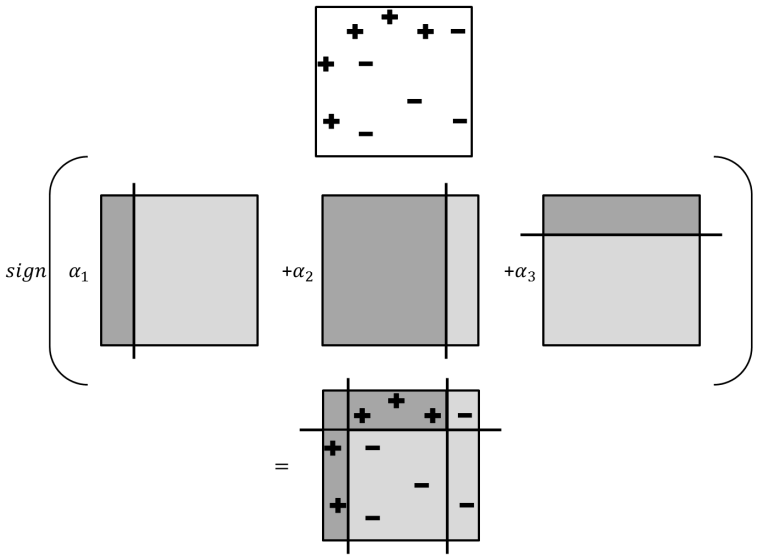


Figure 20.1: Combining weak learners to form a stronger learner. The figure is strongly inspired from Schapire and Freund (2012).

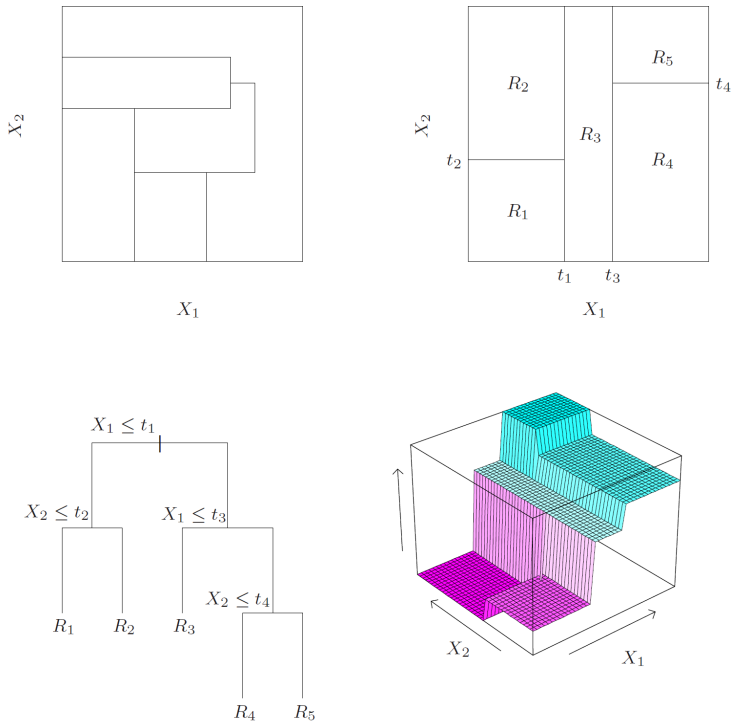


Figure 20.2: Top left: this partition cannot be obtained with a recursive binary tree. Top right: a partition corresponding to a binary tree. Bottom left: the corresponding tree. Bottom right: a function associated to this tree (each leaf—that is, partition of the input space—is associated to a constant value). The figure is taken from Hastie et al. (2009).

This model can be represented by the binary recursive tree shown on the bottom left panel. Inputs are fed to the root (top) of the tree, and they are assigned on the left or right branch, depending on the fact that the condition is satisfied or not, until reaching a leaf (terminal node). The leaves correspond to the regions $R_1, \dots, R_5$. The corresponding regression model predicts y with a constant c_m in region R_m:

$$f(x) = \sum_{m=1}^{5} c_m \mathbb{I}_{\{x \in R_m\}}.$$

The bottom right panel shows such a function, for the preceding tree and some constants c_m.

An advantage of trees is their interpretability (one can see easily from the drawn tree the stratification of data to reach the predicted value). There remains to show how such a tree can be built, depending on the problem at hand.

20.1.2 Building regression trees

Assume that we want to build a *regression tree* using the dataset $\mathcal{D} = \{(x_i, y_i)_{1 \le i \le n}\}$ with $x_i = (x_{i,1}, \dots, x_{i,d})^\top \in \mathbb{R}^d$. The algorithm should learn the topology (through splitting variables and split points) of the tree as well as values associated to each leaf, so as to minimize some learning criterion (such as the empirical risk based on the ℓ_2-loss).

First, assume that a partition in M regions $(R_1, \dots, R_M)$ is fixed, and the regression model associates a constant c_m in each region:

$$f(x) = \sum_{m=1}^{M} c_m \mathbb{I}_{\{x \in R_m\}}.$$

To minimize the risk based on the ℓ_2-loss, one should solve the following optimization problem:

$$\min_{c_1,\dots,c_M} \mathcal{R}_n(f) = \min_{c_1,\dots,c_M} \frac{1}{n} \sum_{i=1}^{n} (y_i - f(x_i))^2.$$

The solution is easily obtained by setting the gradient (respectively to each c_m) to zero:

$$\hat{c}_m = \text{ave}(y_i | x_i \in R_m) = \frac{\sum_{i=1}^n y_i \mathbb{I}_{\{x_i \in R_m\}}}{\sum_{i=1}^n \mathbb{I}_{\{x_i \in R_m\}}}.$$

This is simply the empirical expectation of outputs corresponding to inputs belonging to region R_m.

However, finding the best binary partition in terms of the risk $\mathcal{R}_n$ is much more difficult, and even computationally infeasible in general. Hence, the idea is to proceed with a greedy algorithm. We start with the whole dataset. Let j be a splitting variable (a component of the input) and s a split point, and define the pair of half planes

$$R_1(j,s) = \{x_i \in \mathcal{D} : x_{i,j} \le s\} \text{ and } R_2(j,s) = \{x_i \in \mathcal{D} : x_{i,j} > s\}.$$

Then, we search for the couple (j, s) that solves

$$\min_{j,s} \left(\min_{c_1} \sum_{x_i \in R_1(j,s)} (y_i - c_1)^2 + \min_{c_2} \sum_{x_i \in R_2(j,s)} (y_i - c_2)^2 \right).$$

For a given choice of j and s (notice that $1 \le j \le d$, as each input has d components, and that is is enough to consider $n-1$ split points, obtained by ordering the jth components of inputs in the dataset), the inner minimization problem is solved with

$$\hat{c}_1 = \text{ave}(y_i | x_i \in R_1(j,s)) \text{ and } \hat{c}_2 = \text{ave}(y_i | x_i \in R_2(j,s)).$$

Therefore, by scanning through each dimension of all the inputs, determination of the best pair (j, s) is feasible. Then, having found the best split, we partition the data into the two resulting

regions and repeat the splitting procedure for each of these regions. Then the process is repeated again on each resulting region, and so on. This is repeated until a stopping criterion is met, for example a maximum depth, a maximum number of leaves or a minimum number of samples per leaf.

The stopping criterion is not anodyne. Clearly, a very large tree will overfit the data (consider a tree with as many leaves as samples) while a too small tree might not capture the important structure. For example, a *decision stump* (mentioned at the beginning of this chapter) is a tree with two nodes. Consider the exemple of Fig. 20.1, a decision stump cannot capture the structure of the data, contrary to a slightly larger tree. A solution would be to prune the tree: one construct a big tree, then prune it by collapsing some of its internal nodes according to some criterion. See Hastie et al. (2009, Ch. 9) for more details. We do not study this further, as ensemble methods allow using such trees (big trees for bagging, small trees for boosting, see the next chapters).

20.1.3 Building classification trees

For building a *classification tree* ($y \in \{1, \dots, K\}$), the risk based on the ℓ_2-loss is not the best choice, we should consider another criteria to split the nodes.

For a node m representing a region R_m with $n_m = \sum_{i=1}^n \mathbb{I}_{\{x_i \in R_m\}}$ data points (writing $\mathcal{D}_m = \{(x_i, y_i) \in \mathcal{D} : x_i \in R_m\}$ the associated dataset), write

$$\hat{p}_{m,k} = \frac{1}{n_m} \sum_{x_i \in R_m} \mathbb{I}_{\{y_i = k\}} \tag{20.1}$$

the proportion of class k observations in node m. We classify the observations in node m to class $k(m) = \operatorname{argmax}_{1 \leq k \leq K} \hat{p}_{m,k}$ (majority class). Different measures $Q(\mathcal{D}_m)$ of node impurity[3] can be considered:

- misclassification error,

$$Q(\mathcal{D}_m) = \frac{1}{n_m} \sum_{x_i \in R_m} \mathbb{I}_{\{y_i \neq k(m)\}} = 1 - \hat{p}_{m,k(m)}; \tag{20.2}$$

- Gini index,

$$Q(\mathcal{D}_m) = \sum_{k \neq k'} \hat{p}_{m,k} \hat{p}_{m,k'} = \sum_{k=1}^K \hat{p}_{m,k}(1 - \hat{p}_{m,k});$$

- cross-entropy,

$$Q(\mathcal{D}_m) = -\sum_{k=1}^K \hat{p}_{m,k} \ln \hat{p}_{m,k}$$

These measures are illustrated in Fig. 20.3 in the case of binary classification.

The tree is growth as previously. For a region R_m, consider a couple (j, s) of splitting variable and split point and write $\mathcal{D}_{m,L}(j, s)$ the resulting dataset of the left node (of size n_{m_L}) and $\mathcal{D}_{m,R}(j, s)$ the dataset of the right node (of size n_{m_R}). The tree is growth by solving

$$\min_{j,s} \left(n_{m_L} Q(\mathcal{D}_{m,L}(j, s)) + n_{m_R} Q(\mathcal{D}_{m,R}(j, s)) \right)$$

for one of the preceding measures of impurity. As for the regression case, the tree can be pruned, but we do not study this aspect.

[3] In the regression case, the measure of impurity is

$$Q(\mathcal{D}_m) = \frac{1}{n_m} \sum_{x_i \in R_m} (y_i - \hat{c}_m)^2,$$

with $\hat{c}_m = \text{ave}(y_i | x_i \in R_m)$.

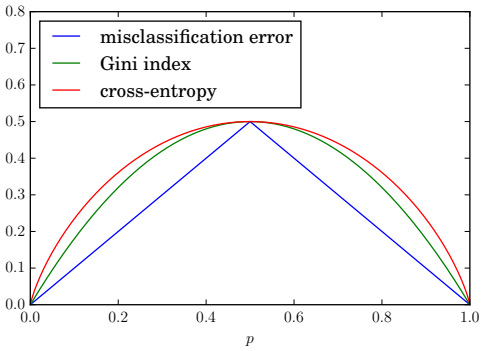


Figure 20.3: Measure of node impurity for binary classification, as a function of the proportion p in the second class.

20.1.4 More on trees

We have seen that decision trees are interpretable predictors that are quite easy to train. They have other advantages. For example, they can handle categorical values (when one of the components of the input takes a finite number of values), for example by binerazing them. They can be extended to cost-sensitive learning quite easily. They can also handle missing input values (if some components of some inputs are missing). See Hastie et al. (2009, Ch. 9) for more details on these subjects.

Unfortunately, trees are unstable, in the sense that they have a high variance. Often, a small change in the data can result in a very different series of split, thus on a different predictor. This instability is mainly caused by the hierarchical nature of the process: the effect of an error in the top split is propagated down to all of the splits below it. Bagging, to be presented in Ch. 21, is a way to reduce the related variance. The smaller the tree, the lesser this variance effect, but also the weaker the learner. Boosting, to be presented in Ch. 22, is a way to combine such weak learners to form a strong learner (without the instability of trees)

20.2 Overview

In this section, we provide a brief overview of the ensemble methods studied next. As explained before, such methods aim at combining the predictions of several base estimators (for example, the trees studied above) in order to improve generalization or robustness over a single estimator.

We will distinguish mainly tow families of ensemble methods:

- the **bagging** methods (or more generally averaging methods), to be presented in Ch. 21. The underlying idea is to build several estimators, (more or less) independently and in a randomized fashion, and to average their prediction. This leads to a reduction of variance for the combined estimator, that makes it applicable for example to large and unpruned trees;
- the **boosting** methods, to be presented in Ch. 22. The idea is to build sequentially (weak) base estimators in order to reduce the bias of the combined estimator (see for example Fig. 20.1 for an illustration), the motivation being to combine weak learners to form a strong learner.

also be linked to trees) and stacking (which somehow build a kind of meta-risk for combining arbitrary base estimators obtained from different learning algorithms).

There are other ensemble methods that we will not study here. We mention some of them for completeness:

- *Bayesian model averaging* (BMA) combine a set of candidate models (*e.g.*, for classification) using a Bayesian viewpoint. Informally, let $\mathcal{D}$ be the dataset, ξ a quantity of interest (*e.g.*, the prediction of the class for a given input) and write $\mathcal{M}_m$ each of the M candidate model.

BMA provides the posterior distribution on ξ conditioned on the examples in $\mathcal{D}$ by integrating over models:

$$\mathrm{P}(\xi \mid \mathcal{D}) = \sum_{m=1}^{M} \mathrm{P}(\xi \mid \mathcal{M}_m, \mathcal{D}) \mathrm{P}(\mathcal{M}_m \mid \mathcal{D}) \propto \sum_{m=1}^{M} \mathrm{P}(\xi \mid \mathcal{M}_m, \mathcal{D}) \mathrm{P}(\mathcal{D} \mid \mathcal{M}_n) \mathrm{P}(\mathcal{M}_n).$$

See Hoeting et al. (1999) for more on Bayesian model averaging and more generally Part **??** for an introduction to Bayesian Machine Learning (the above equations should become clear in light of this part of the course material);

- *mixture of experts* combine local predictors considering possibly heterogeneous sets of features. They can also be seen as a variation of decision trees[4], the difference being that the tree splits are not hard decisions but rather soft (fuzzy or probabilistic) ones, and that the model in each leaf might be more complex than a simple constant prediction. See Jacobs et al. (1991); Jordan and Jacobs (1994) or Hastie et al. (2009, Ch. 9) for more on this subject;
- *stacking* is a way of combining heterogeneous estimators for a given problem. The basic idea is to combine different estimators in a statistical way, that is by minimizing a given risk. For example, consider that a classification problem is solved by using many different classification algorithms, each one producing an estimator. Then, one can construct an hypothesis space being composed of, for example, all linear combinations of these estimators. Then, a classification risk can be minimized over this hypothesis space. Generally, it can be shown that the learned combination is at least as good as the better estimator. For more on this subject, see Wolpert (1992); Breiman (1996b); Smyth and Wolpert (1999); Ozay and Vural (2012), for example.

[4] As such, they might not be considered as an ensemble method, opinions vary on this subject.

Chapter 21

Bagging

Bagging stands for "bootstrap and aggregating". The underlying idea is to learn several estimators (more or less) independently (by introducing some kind of randomization) and to average their predictions. The averaged estimator is usually better than the single estimators because it reduces its variance. We motivate this informally. Assume that $X_1, \dots, X_B$ are B i.i.d. random variables of mean $\mu = \mathrm{E}[X_1]$ and of variance $\sigma^2 = \mathrm{var}(X_1) = \mathrm{E}\left[(X_1 - \mu)^2\right]$. Consider the empirical mean $\mu_B = \frac{1}{B}\sum_{b=1}^{B} X_b$. The expectation does not change, $\mathrm{E}[\mu_B] = \mu$, while the variance is reduced (thanks to decorrelation of the random variables), $\mathrm{var}(\mu_B) = \frac{1}{B}\sigma^2$.

In the supervised learning paradigm, the random quantity is the dataset $\mathcal{D} = \{(x_i, y_i)_{1 \le i \le n}\}$, where samples are drawn from a fixed but unknown distribution. From this, an estimate $f_{\mathcal{D}}$ is computed by minimizing the empirical risk of interest (see Ch. 5). This is a random quantity (through the dependency on the dataset) that admits an expectation. This estimator has also a variance, which somehow tells how different will be the predictions if the dataset is perturbated (this can also be linked to the Vapnik-Chervonenkis bound studied in Ch. 5).

Assume that we can sample datasets on demand. Then, let $\mathcal{D}_1, \dots, \mathcal{D}_B$ be datasets drawn independently, and write $f_b = f_{\mathcal{D}_b}$ the associated minimizer of the empirical risk. The averaged estimator is $f_{\mathrm{ave}} = \frac{1}{B}\sum_{b=1}^{B} f_b$. This does not change the expectation, $\mathrm{E}[f_{\mathrm{ave}}] = \mathrm{E}[f_1]$, but it reduces the variance, $\mathrm{var}(f_{\mathrm{ave}}) = \frac{1}{B}\mathrm{var}(f_1)$. This is of interest for example for decision trees: we have seen in Sec. 20 that large and unpruned trees have a small bias but a large variance.

Unfortunately, it is not possible to sample datasets on demand, as the underlying distribution is unknown. We have to do with the sole dataset we have. That is where bootstrapping is useful.

21.1 Bootstrap aggregating

Generally speaking, in statistics, bootstrapping refers to any method that relies on random sampling with replacement. Here, bootstrapping is applied to the dataset. The basic idea is to randomly draw datasets with replacement from the training data, each one having the same size as the original training set. This is done B times, producing B bootstrap datasets, denoted $\mathcal{D}_b$. Let $S(\mathcal{D})$ be any quantity that can be computed from the data set. From the bootstrap sampling one can compute the quantities $S(\mathcal{D}_b)$ and use them to compute any statistic of interest. This is illustrated in Fig. 21.1.

Bootstrap aggregation, or bagging, consists in bootstrapping the dataset to get B datasets $\mathcal{D}_b$, $1 \le b \le B$, in learning a predictor $f_{\mathcal{D}_b} = f_b$ for each of these datasets and then in averaging the predictors:

$$f_{\mathrm{bag}}(x) = \frac{1}{B}\sum_{b=1}^{B} f_b(x). \tag{21.1}$$

This can be seen as an approximation of the scheme described at the beginning of the chapter, the approximation coming from the fact that the empirical distribution is used instead of the unknown

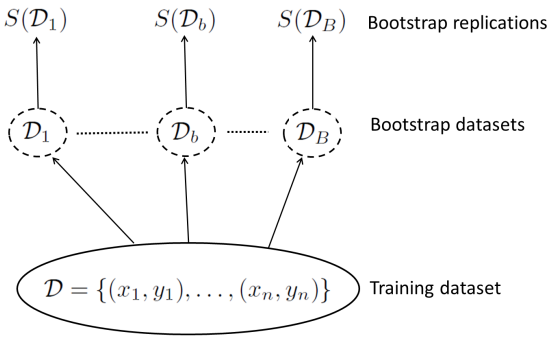


Figure 21.1: Illustration of the bootstrapping principle.

underlying distribution[1]. Due to this approximation, independence (or even decorrelation) cannot be assumed. Yet, this can improve the results empirically.

This idea can typically applied to decision trees. For regression trees, Eq. (21.1) can be applied directly. For classification trees, an average of predicted classes would not make sense. Recall that a classification tree will make a majority votes of the exemples belonging to the leaf where the input of interest ends up (and associates this way a class to each sample). A first possibility is to do a majority vote over trees:

$$f_{\text{bag}} = \underset{1 \leq i \leq K}{\text{argmax}} \left(\frac{1}{B} \sum_{b=1}^{B} \mathbb{I}_{\{f_b(x)=k\}} \right).$$

Another bagging strategy consists in considering the class proportion for the leave corresponding to the input of interest for each tree (see Eq. (20.1)), to average them over all trees and to output the class that maximizes this averaged class proportion.

There exists variations of the bagging approach, depending on how datasets are sampled from the original training set:

- samples can be drawn with replacement, which is the principle of the *bagging* approach, explained above (Breiman, 1996a);
- alternatively, random subsets of the dataset can be drawn as random subsets of the samples, which is known as *pasting* (Breiman, 1999);
- one can also selects randomly a subset of the components of the inputs (which is generally multi-dimensional) to learn models. When random subsets of the dataset are drawn as random subsets of the features, the method is known as *random subspaces* (Ho, 1998);
- is is possible to combine these ideas. When base estimatores are built on subsets of bots samples and features, it is known as *random patches* (Louppe and Geurts, 2012).

21.2 Random forests

Bagging average many noisy but approximately unbiased models to reduce the variance. However, there is necessarily some overlap between bootstrapped datasets, so the models corresponding to each of these datasets are correlated. We have justified informally the idea of averaging by the fact that if $X_1, \ldots, X_B$ are i.i.d. random variables, then $\text{var}(\mu_B) = \frac{\sigma^2}{B}$ (using the notations introduced at the beginning of the chapter). Now, assume that these random variables are identically distributed but not independent, with a positive pairwise correlation ρ:

$$\text{for } i \neq j, \quad \rho = \frac{\text{E}\left[(X_i - \mu)(X_j - \mu)\right]}{\sqrt{\text{var}(X_i)\,\text{var}(X_j)}}.$$

[1]This empirical distribution is a discrete distribution that associates a probability of $\frac{1}{n}$ to each sample (x_i, y_i). Sampling from the dataset with replacement is sampling according to this empirical distribution.

Then, one can easily show[2] that

$$\text{var}(\mu_B) = \rho\sigma^2 + \frac{1-\rho}{B}\sigma^2. \tag{21.2}$$

Therefore, the variance cannot be shrink below $\rho\sigma^2$.

Algorithm 19 Random Forest

Require: A dataset $\mathcal{D} = \{(x_i, y_i)_{1 \leq i \leq n}\}$, the size B of the ensemble, the number m of candidates for splitting.

for $b = 1$ **to** B **do**
  Draw a bootstrap dataset $\mathcal{D}_b$ of size n from the original training set $\mathcal{D}$.
  Grow a random tree using the bootstrapped dataset:
  repeat
    for all terminal node **do**
      Select m variables (attributes) among d, at random.
      Pick the best variable and split-point couple among the m.
      Split the node into two daughter nodes.
    end for
  until the stopping criterion is met (*e.g.*, minimum number of sample per node reached)
end for
return the ensemble of B trees.

When averaged models are trees, Breiman (2001) has proposed *random forests* to further reduce the variance. The underlying idea is to reduce the correlation between the trees by randomizing their constructions, without increasing the variance too much. This randomization is achieved in the tree-growing process thanks to random selection of input variables[3]: at each split, $m < d$ of the input variables (attributes) are selected at random as candidates for splitting, the choice of the best variable and split-point among these candidates being as explained in Ch. 20. See also Alg. 19.

Intuitively, reducing m will reduce the correlation between any pair of trees in the ensemble, and hence by Eq. (21.2) will reduce the variance of the average. However, The corresponding hypothesis space will be smaller, leading to an increased bias. A heuristic is to choose $m = \lfloor \sqrt{d} \rfloor$ and a minimum node size of 1 for classification, and $m = \lfloor \frac{d}{3} \rfloor$ and a minimum node size of 5 for regression. For further information about random forests, the reader can refer to (Hastie et al., 2009, Ch. 15) (that provides notably a bias-variance analysis).

21.3 Extremely randomized trees

Randomization can be pushed further with *extremely randomized forests*, introduced by Geurts et al. (2006). It is basically a random forest, with two important differences:

- split-points are also chosen randomly (in addition to splitting dimensions). More precisely, $m \leq d$ of the input variables (attributes) are chosen at random, and for each of these variables a split-point is selected at random;
- the full learning dataset is used to growth each tree (instead of a bootstrapped replica).

The rationale behind choosing also the split-point at random is to further reduce the correlation between trees (so as to reduce the variance of the average of the ensemble more strongly). The rationale for using the full learning set is to achieve a lower bias (at the price of an increased variance, that should be compensated by the randomization of split-points). The corresponding approach is described in Alg. 20.

[2] Indeed, we have that $\text{var}(\mu_B) = \text{E}\left[(\frac{1}{n}\sum_b (X_b - \mu))^2\right] = \frac{1}{n^2}\sum_{b,b'} \text{E}\left[(X_b - \mu)(X_{b'} - \mu)\right] = \frac{1}{n^2}(\sum_b \text{E}\left[(X_b - \mu)^2\right] + \sum_{b \neq b'} \text{E}\left[(X_b - \mu)(X_{b'} - \mu)\right]) = \frac{1}{n^2}(n(n-1)\rho\sigma^2 + n\sigma^2) = \rho\sigma^2 + \frac{1-\rho}{n}\sigma^2$.

[3] Notice that this is different from random subspaces: input variables are chosen randomly for each splitting, and not while bootstrapping the dataset.

Algorithm 20 Extremely Randomized Forest

Require: A dataset $\mathcal{D} = \{(x_i, y_i)_{1 \leq i \leq n}\}$, the size B of the ensemble, the number m of candidates for splitting.

for $b = 1$ **to** B **do**
  Grow a random tree using the original dataset:
  repeat
    for all terminal node **do**
      Select m variables among d, at random.
      for all sampled variables **do**
        Select a split at random
      end for
      Pick the best variable and split-point couple among the m candidates.
      Split the node into two daughter nodes.
    end for
  until the stopping criterion is met (*e.g.*, minimum number of sample per node reached)
end for
return the ensemble of B trees.

Empirically, it often provides better results than random forests. Another advantage of this approach is its lower computational complexity compared to random forests (instead of searching the best split-point among the m drawn dimensions, one chooses the split-point among the m randomly drawn split-points). See the original paper (Geurts et al., 2006) for more details.

Chapter 22

Boosting

We have seen in Ch. 21 that bagging consists in learning *in parallel* a set of models with *low bias* and *high variance* (learning being randomized, for example through bootstrapping), the prediction being made by averaging all these models. *Boosting* takes a different route. The underlying idea is to add *sequentially* models with *high bias* and *low variance* such as reducing the bias of the ensemble. This is illustrated on Fig. 20.1 page 210: combining decision stumps (binary trees with two leaf nodes) allows constructing a complex decision boundary.

The rest of this chapter is organized as follows. In Sec. 22.1, we present AdaBoost, a seminal and very effective boosting algorithm. In Sec. 22.2, we show how this algorithm can be derived and demonstrates why the ensemble achieves a lower error than each of its components (called weak learners) taken separately. In Sec. 22.3, we extend these ideas using an optimization perspective.

22.1 AdaBoost

The seminal *AdaBoost* algorithm of Freund and Schapire (1997) deals with cost-insensitive binary classification. The available dataset is of the form $\mathcal{D} = \{(x_i, y_i)_{1 \leq i \leq n}\}$ with $y_i \in \{-1, +1\}$. Before presenting AdaBoost, we discuss briefly weighted classification.

22.1.1 Weighted binary classification

So far, our estimates (or decision rules) have been obtained by minimizing an empirical risk or a convex surrogate (see Ch. 5.2). Typically, for binary classification the risk of interest is

$$\mathcal{R}_n(f) = \frac{1}{n} \sum_{i=1}^{n} \mathbb{I}_{\{y_i \neq f(x_i)\}} = \sum_{i=1}^{n} \frac{1}{n} \mathbb{I}_{\{y_i \neq f(x_i)\}}.$$

This way, all samples of the dataset have the same importance (each sample has a weight $\frac{1}{n}$). Now, assume that we want to associate a different weight w_i to each example (x_i, y_i) (this will be useful for boosting), such that $\sum_i w_i = 1$ and $w_i \geq 0$. The empirical risk to be considered is

$$\mathcal{R}_n(f) = \sum_{i=1}^{n} w_i \mathbb{I}_{\{y_i \neq f(x_i)\}}.$$

This is a generalization of the approaches considered before, in the sense that they correspond to the choice $w_i = \frac{1}{n}$, for all $1 \leq i \leq n$. Weighting the samples allow putting more emphasis on some of them on less on the others.

Minimizing the weighted empirical risk can be done (approximately) by sampling a bootstrap replicate according to the discrete distribution $(w_1, \ldots, w_n)$ (sampling with replacement according to this distribution). It can also be done more directly, in a problem-dependent manner. For example, consider the classification trees presented in Ch. 20. We have seen that a measure of node impurity is the misclassification error of Eq. (20.2) (see page 212). It can be replaced by a

weighted misclassification error:

$$Q(\mathcal{D}_m) = \sum_{x_i \in R_m} w_i \mathbb{I}_{\{y_i \neq k(m)\}}.$$

The tree is then growth by solving

$$\min_{j,s} \left(Q(\mathcal{D}_{m,L}(j,s)) + Q(\mathcal{D}_{m,R}(j,s)) \right).$$

Other measures of node impurity can be adapted in a similar way.

In the rest of this section, we assume that a *weak learner*[1] is available (typically, a decision stump), and that it is able to minimize the weighted risk using the dataset $\mathcal{D}$ and weights w_i, $1 \leq i \leq n$. Notice that we have presented weighted binary classification, but this idea is much more general (*e.g.*, weighted regression).

22.1.2 The AdaBoost algorithm

AdaBoost is presented in Alg. 21.

Algorithm 21 AdaBoost

Require: A dataset $\mathcal{D} = \{(x_i, y_i)_{1 \leq i \leq n}\}$, the size T of the ensemble.

1: Initialize the weights $w_i^1 = \frac{1}{n}$, $1 \leq i \leq n$

2: **for** $t = 1$ **to** T **do**

3: Fit a binary classifier $f_t(x)$ to the training data using weights w_i^t.

4: Compute the error ϵ_t made by this classifier:

$$\epsilon_t = \sum_{i=1}^{n} w_i^t \mathbb{I}_{\{y_i \neq f_t(x_i)\}}.$$

5: Compute the learning rate α_t:

$$\alpha_t = \frac{1}{2} \ln \left(\frac{1 - \epsilon_t}{\epsilon_t} \right).$$

6: Update the weights, for all $1 \leq i \leq n$:

$$w_i^{t+1} = \frac{w_i^t e^{-\alpha_t y_i f_t(x_i)}}{\sum_{j=1}^{n} w_j^t e^{-\alpha_t y_j f_t(x_j)}}.$$

7: **end for**

8: **return** the decision rule

$$H_T(x) = \operatorname{sgn}\left(F_T(x)\right) \text{ with } F_T(x) = \sum_{t=1}^{T} \alpha_t f_t(x).$$

Let us explain the rational behind this algorithm. At the beginning, all samples are equally weighted. Then, at each iteration t, one first trains a binary classifier $f_t(x)$ with the training set $\mathcal{D} = \{(x_i, y_i)_{1 \leq i \leq n}\}$ and weights w_i^t (that is, such as minimizing the weighted risk $\mathcal{R}_n(f) = \sum_{i=1}^{n} w_i^t \mathbb{I}_{\{y_i \neq f(x_i)\}}$). Write $\epsilon_t = \mathcal{R}_n(f_t)$ the error made by this classifier (see line 4 in Alg. 21). Notice that we have necessarily that $\epsilon_t < \frac{1}{2}$ (otherwise, the classifier does worse than random guessing, and $-f_t$ is a better classifier, with an empirical weighted risk below $\frac{1}{2}$). The closer to 0 is ϵ_t, the better the classifier is. However, with a weak learner such as a decision stump, the error will be more probably close to $\frac{1}{2}$. Then, one compute the learning rate α_t (see line 5 in Alg. 21). This rate is a decreasing function of the error ϵ_t: with $\epsilon_t = \frac{1}{2}$, $\alpha_t = 0$ (which means that if the classifier

[1] The learner is weak in the sense that it has a high bias.

does not better than random guessing, it is not added to the ensemble), and $\lim_{\epsilon_t \to 0} \alpha_t = +\infty$ (which suggests to stop adding models to the ensemble when the empirical risk is null). Then, the weights are updated (see line 6 in Alg. 21). This can be rewritten as (up to the normalization factor)

$$w_i^{t+1} \propto \begin{cases} w_i^t e^{-\alpha_t} & \text{if } f_t(x_i) = y_i \\ w_i^t e^{\alpha_t} & \text{if } f_t(x_i) \neq y_i \end{cases}.$$

This means that if the example (x_i, y_i) is correctly classified, its weight is decreased, while if it is incorrectly classified, its weight is increased. The final decision rule (the *strong learner*[2]) is the sign of the weighted combination of the learned classifiers:

$$H_T(x) = \operatorname{sgn}\left(F_T(x)\right) \text{ with } F_T(x) = \sum_{t=1}^{T} \alpha_t f_t(x).$$

To sum up, AdaBoost is a sequential algorithm. At each iteration, samples that where misclassified by the preceding classifier have their weight increased. Therefore, examples that are difficult to classify correctly receive ever-increasing influence as iterations proceed.

AdaBoost is a very efficient algorithm. For example, it is behind the face detection algorithm embedded in recent cameras and smartphones, see Viola and Jones (2001). If this chapter focuses on AdaBoost, boosting is a much larger field, the interested reader can refer to Schapire and Freund (2012) for a deeper introduction (see also Hastie et al. (2009, Ch. 10) for a different point of view).

22.2 Derivation and partial analysis

If the algorithm make sense, one can wonder how it is derived (that is, why this choice of reweighting and not another one), and what type of guarantee it can offers.

22.2.1 Forward stagewise additive modeling

Here, we show that AdaBoost fits an additive model that minimizes (sequentially) the empirical risk based on the exponential loss (a convex surrogate presented in Ch. 5.2). This way of deriving AdaBoost has first been proposed by Friedman et al. (2000), it is not how it has been derived originally (Freund and Schapire, 1997).

We have seen in Ch. 5.2 that a convex surrogate for the binary classification problem is the risk based on the exponential loss:

$$\mathcal{R}_n(F) = \frac{1}{n} \sum_{i=1}^{n} e^{-y_i F(x_i)}.$$

The loss is low if the class is correctly predicted, and high otherwise. Moreover, we are looking for an additive model of the form

$$F_T(x) = \sum_{t=1}^{T} \alpha_t f_t(x),$$

with f_t being a binary classifier (that is, $f_t \in \{-1, +1\}^{\mathcal{X}}$), called a basis function in the sequel. The corresponding optimization problem is therefore

$$\min_{(\alpha_t, f_t)_{1 \leq t \leq T}} \frac{1}{n} \sum_{i=1}^{n} e^{-y_i \sum_{t=1}^{T} \alpha_t f_t(x_i)}.$$

Yet, this optimization problem is too complicated. A simple alternative is to search for an approximate solution by sequentially adding basis functions and associate weights. Define $F_0 = 0$ and $F_t = F_{t-1} + \alpha_t f_t$. This consists in solving sequentially the following subproblems:

$$\min_{\alpha, f} \frac{1}{n} \sum_{i=1}^{n} e^{-y_i (F_{t-1}(x_i) + \alpha f(x_i))}.$$

[2]The learning is strong because it has a reduced bias, as will be shown later.

This is reminiscent of gradient descent (see also Sec. 22.3). This idea can also be straightforwardly abstracted to any loss function L (not necessarily corresponding to a binary classification problem):

$$\min_{\alpha,f} \frac{1}{n} \sum_{i=1}^{n} L(y_i, F_{t-1}(x_i) + \alpha f(x_i)).$$

Now, we compute the solution of this problem in the case of the exponential loss, with binary classifiers as basis functions.

At each iteration $t \geq 1$, we have to solve

$$(\alpha_t, f_t) = \operatorname*{argmin}_{\alpha,f} \frac{1}{n} \sum_{i=1}^{n} e^{-y_i(F_{t-1}(x_i)+\alpha f(x_i))}.$$

Define the weight w_i^t as

$$w_i^t = \frac{e^{-y_i F_{t-1}(x_i)}}{\sum_{j=1}^{n} e^{-y_j F_{t-1}(x_j)}}.$$

The optimization problem can thus be rewritten as

$$\begin{aligned}(\alpha_t, f_t) &= \operatorname*{argmin}_{\alpha,f} \frac{\sum_{j=1}^{n} e^{-y_j F_{t-1}(x_j)}}{n} \sum_{i=1}^{n} w_i^t e^{-\alpha y_i f(x_i)} \\ &= \operatorname*{argmin}_{\alpha,f} \sum_{i=1}^{n} w_i^t e^{-\alpha y_i f(x_i)}, \end{aligned} \tag{22.1}$$

the term $\frac{1}{n}\sum_{j=1}^{n} e^{-y_j F_{t-1}(x_j)}$ being a (positive) constant respectively to the optimization. Notice also that the terms w_i^t depend neither on α nor $f(x)$, so they can be seen as weights applied to each example. Notice also that $w_i^1 = \frac{1}{n}$ (as $F_0 = 0$ by assumption). Write $J_t(\alpha, f)$ the criterion of Eq. (22.1). It can be written equivalently as (by rearranging the sums):

$$\begin{aligned} J_t(\alpha, f) &= \sum_{i=1}^{n} w_i^t e^{-\alpha y_i f(x_i)} \qquad (22.2) \\ &= e^{-\alpha} \sum_{i:y_i = f(x_i)} w_i^t + e^{\alpha} \sum_{i:y_i \neq f(x_i)} w_i^t \\ &= \left(e^{\alpha} - e^{-\alpha}\right) \sum_{i=1}^{n} w_i^t \mathbb{I}_{\{y_i \neq f(x_i)\}} + e^{-\alpha} \sum_{i=1}^{n} w_i^t \\ &= \left(e^{\alpha} - e^{-\alpha}\right) \sum_{i=1}^{n} w_i^t \mathbb{I}_{\{y_i \neq f(x_i)\}} + e^{-\alpha}, \qquad (22.3) \end{aligned}$$

using the fact that $\sum_{i=1}^{n} w_i^t = 1$ (from the definition of the weights).

We can see on Eq. (22.3) that the solution can be obtained in two step (optimization over α and f having been separated). First, for any $\alpha > 0$, the solution to Eq. (22.3) for $f(x)$ is

$$f_t = \operatorname*{argmin}_{f \in \mathcal{H}} \sum_{i=1}^{n} w_i^t \mathbb{I}_{\{y_i \neq f(x_i)\}}. \tag{22.4}$$

In other words, f_t solves the weighted empirical risk corresponding to line 3 of Alg. 21 (we will see later that the weights are indeed the same). Here, $\mathcal{H}$ is the hypothesis space of considered weak learners (or basis functions). For example, it can be the space of decision stumps. Write ϵ_t the corresponding error

$$\epsilon_t = \sum_{i=1}^{n} w_i^t \mathbb{I}_{\{y_i \neq f_t(x_i)\}}.$$

We therefore have that

$$J_t(\alpha, f_t) = (e^{\alpha} - e^{-\alpha})\epsilon_t + e^{-\alpha}. \tag{22.5}$$

Solving for α, we get

$$\nabla_\alpha J_t(\alpha, f_t) = 0 \Leftrightarrow \alpha_t = \frac{1}{2}\ln\left(\frac{1-\epsilon_t}{\epsilon_t}\right). \tag{22.6}$$

We retrieve the learning rate of line 5 in Alg. 21. Thus, we have solved problem (22.1), its solution being given by Eqs. (22.4) and (22.6). Regarding the weights update, we have that (up to the normalizing constant):

$$\begin{aligned} w^{t+1} &\propto e^{-y_i F_t(x_i)} \\ &= e^{-y_i(F_{t-1}(x_i)+\alpha_t f_t(x_i))} = e^{-y_i F_{t-1}(x_i)} e^{-\alpha_t y_i f_t(x_i)} \\ &\propto w_i^t e^{-\alpha_t y_i f_t(x_i)}, \end{aligned}$$

which is the update rule in line 6 of Alg. 21.

Therefore, we have shown that AdaBoost can be derived as an additive model minimizing sequentially the empirical risk based on the empirical loss (this is generally called *forward stagewise additive modeling*). Now, one can wonder to what extent this empirical risk is indeed minimized, notably based on the quality (or bias) of each basis function (each function f_t—for example decision stumps—minimizing intermediate weighted risks).

22.2.2 Bounding the empirical risk

In this section, we provide (and prove) a bound on the empirical risk (based on the binary loss) of the solution H_T computed by AdaBoost. For this, we define the edge $\gamma_t = \frac{1}{2} - \epsilon_t$, which measures how much better than random guessing (error rate of $\frac{1}{2}$) is the error rate of the tth learned classifier f_t. As explained before, this weak learners will typically have low bias but high variance (such as decision stumps), so the edge γ_t would be small (the weak learners do only slightly better than random guessing). The next result shows that the ensemble $H_T(x) = \text{sgn}(F_T(x))$ does much better than a single weak learner, given that it is big enough.

Theorem 22.1 (Freund and Schapire 1997)**.** *Write $\gamma_t = \frac{1}{2} - \epsilon_t$ the edge of the tth classifier. The empirical risk of the combined classifier H_T produced by AdaBoost (Alg. 21) satisfies*

$$\frac{1}{n}\sum_{i=1}^{n} \mathbb{I}_{\{y_i \neq H_T(x_i)\}} \leq \prod_{t=1}^{T} \sqrt{1-4\gamma_t^2} \leq e^{-2\sum_{t=1}^{T}\gamma_t^2}.$$

In other words, the training error drops exponentially fast as a function of the number of combined weak learners. For example, if each weak learner has a 40% misclassification rate, then $\gamma_t = 0.1$ and the empirical risk is bounded by $(\sqrt{1-4(0.1)^2})^T \leq 0.98^T$, which can be arbitrarily close to zero, given a large enough T. Now, we prove this result.

Proof of Th. 22.1. Recall that $F_T(x) = \sum_{t=1}^{T} \alpha_t f_t(x)$. Write Z_t the normalizing factor of the weights at round t:

$$Z_t = \sum_{i=1}^{n} w_i^t e^{-\alpha_t y_i f_t(x_i)}. \tag{22.7}$$

Unraveling the recurrence of AdaBoost that defines the weights, we have

$$\begin{aligned} w_i^{T+1} &= w_i^T \frac{e^{-\alpha_T y_i f_T(x_i)}}{Z_T} \\ &= w_i^{T-1} \frac{e^{-\alpha_{T-1} y_i f_{T-1}(x_i)}}{Z_{T-1}} \frac{e^{-\alpha_T y_i f_T(x_i)}}{Z_T} \\ &= w_i^1 \frac{e^{-\alpha_1 y_i f_1(x_i)}}{Z_1} \cdots \frac{e^{-\alpha_T y_i f_T(x_i)}}{Z_T} \\ &= \frac{w_i^1 e^{-y_i \sum_{t=1}^{T} \alpha_t h_t(x_i)}}{\prod_{t=1}^{T} Z_t} \\ &= \frac{w_i^1 e^{-y_i F_T(x_i)}}{\prod_{t=1}^{T} Z_t}. \end{aligned} \tag{22.8}$$

Recall that $H_T(x) = \operatorname{sgn}(F_T(x))$. We would like to bound the binary loss by the exponential one, its convex surrogate. If $H_T(x) \neq y$, then $yF_T(x) < 0$, thus $e^{-yF_T(x)} \geq 1$. Therefore, we always have that $\mathbb{I}_{\{y \neq H_T(x)\}} \leq e^{-yF(x)}$. Therefore, the empirical risk of interest can be bounded as follows:

$$\begin{aligned}
\frac{1}{n}\sum_{i=1}^{n} \mathbb{I}_{\{y_i \neq H_T(x_i)\}} &\leq \frac{1}{n}\sum_{i=1}^{n} e^{-y_i F_T(x_i)} \\
&= \sum_{i=1}^{n} w_i^1 e^{-y_i F_T(x_i)} && \text{(by def. of } w_i^1\text{)} \\
&= \sum_{i=1} w_i^{T+1} \prod_{t=1}^{T} Z_t && \text{(by Eq. (22.8))} \\
&= \prod_{t=1}^{T} Z_t && \text{(as } w_i^{T+1} \text{ is a discrete distribution)} \quad (22.9)
\end{aligned}$$

Recall the definition of $J_t(\alpha, f)$ in Eq. (22.2) and the definition of Z_t in Eq. (22.7). It is clear that $Z_t = J_t(\alpha_t, f_t)$. Therefore, from Eq. (22.5), we have that

$$\begin{aligned}
Z_t &= (e^{\alpha_t} - e^{-\alpha_t})\epsilon_t + e^{-\alpha_t} && (22.10) \\
&= e^{-\alpha_t}(1 - \epsilon_t) + e^{\alpha_t}\epsilon_t \\
&= e^{-\alpha_t}\left(\frac{1}{2} + \gamma_t\right) + e^{\alpha_t}\left(\frac{1}{2} - \gamma_t\right) && \text{(by def., } \epsilon_t = \frac{1}{2} - \gamma_t\text{)} \\
&= \sqrt{\frac{\frac{1}{2} - \gamma_t}{\frac{1}{2} + \gamma_t}}\left(\frac{1}{2} + \gamma_t\right) + \sqrt{\frac{\frac{1}{2} + \gamma_t}{\frac{1}{2} - \gamma_t}}\left(\frac{1}{2} - \gamma_t\right) && \text{(by defs. of } \alpha_t \text{—line 5 in Alg. 21—and } \gamma_t\text{)} \\
&= \sqrt{1 - 4\gamma_t^2} && \text{(by direct calculus).}
\end{aligned}$$

Plugging this result into Eq. (22.9) provides the first bound of the theorem. Using the fact that for all $x \in \mathbb{R}$ we have $1 + x \leq e^x$ provides the second bound and concludes the proof. □

Notice that optimizing the bound in Eq. (22.9) (by minimizing $\prod_{t=1}^{T} Z_t$ over α_t and f_t, $1 \leq t \leq T$, using the relation (22.10)) allows deriving the AdaBoost algorithm. It is how it has been done originally (Freund and Schapire, 1997).

If this result shows that combining enough weak learners, whatever their quality, allows having an arbitrary small empirical risk, it tells noting about the generalization error (that is how the risk, $\mathcal{R}(F_T) = \mathrm{E}\left[\mathbb{I}_{\{Y \neq F_T(X)\}}\right]$, can be controlled). Direct bound on this risk can be obtained by using rather directly the Vapnik-Chervonenkis theory presented in Ch. 5.1. Yet, this analysis would show that AdaBoost suffers from overfitting: to obtain a low empirical risk, one has to add many basis functions (or weak learners), leading to a large Vapnik-Chervonenkis dimension, and thus a large variance. However, AdaBoost does not suffer from this in general. Another (and better) line of analysis is based on the concept of *margin* (which is also central for support vector machines). Here, for the classifier $H_T(x) = \operatorname{sgn}(F_T(x))$, the margin of an example (x, y) is the quantity $yF(x)$. The larger it is (in absolute value), the more confident we are about the prediction (to be correct or not, depending on the sign). One can provide bounds on the risk based on this notion of margin: very roughly, the larger the margin, the sharper the bound. On the other hand, it is possible to show that AdaBoost tends to enlarge the margins of the computed classifier, as the number of iterations increases. A deeper discussion of this is beyond the scope of this manuscript, but the interested reader can refer to Schapire and Freund (2012, Ch. 4 and 5) for more details.

22.3 Restricted functional gradient descent

We have seen that AdaBoost can be derived as a stagewise additive modeling approach for minimizing the exponential loss. Here, we somehow generalize this idea from a convex optimization perspective.

Consider the binary classification problem with a convex surrogate (see Ch. 5.2). We're looking for a classifier $H(x) = \text{sgn}(F(x))$ with $F \in \mathbb{R}^{\mathcal{X}}$. Let $L(y, F(x))$ be a convex surrogate to the binary loss (for example, the exponential loss, $L(y, F(x)) = e^{-yF(x)}$). We would like to minimize the empirical risk:

$$\min_{F \in \mathbb{R}^{\mathcal{X}}} \mathcal{R}_n(F) \text{ with } \mathcal{R}_n(F) = \frac{1}{n} \sum_{i=1}^{n} L(y_i, F(x_i)).$$

As a sum of convex function, $\mathcal{R}_n$ is convex in F. A standard approach for minimizing a convex function is to perform a gradient descent. This would give an update rule of the kind:

$$F_{t+1} = F_t - \alpha_t \nabla_F \mathcal{R}_n(F_t),$$

with α_t the learning rate. The problem here is that F is not a variable, it is a function, so we need to introduce the concept of functional gradient. To do so, we need to introduce a relevant Hilbert space.

Assume that the input space $\mathcal{X}$ is measurable and let μ be a probability measure. The function space $L^2(\mathcal{X}, \mathbb{R}, \mu)$ is the set of all equivalence classes of functions $F \in \mathbb{R}^{\mathcal{X}}$ such that the Lebesgue integral $\int_{\mathcal{X}} F(x)^2 d\mu(x)$ is finite. This Hilbert space has a natural inner product: $\langle F, G \rangle_\mu = \int_{\mathcal{X}} F(x)G(x)d\mu(x)$. A functional is an operator that associates a scalar to a function of this Hilbert space. Let $J : L^2(\mathcal{X}, \mathbb{R}, \mu) \to \mathbb{R}$ be such a functional, its Fréchet derivative is the linear operator $\nabla J(F)$ satisfying

$$\lim_{G \to 0} \frac{J(F+G) - J(F) - \langle \nabla J(F), G \rangle_\mu}{\|G\|_\mu} = 0.$$

It can also be implicitly defined as $J(F + G) = J(F) + \langle \nabla J(F), G \rangle_\mu + o(\|G\|_\mu)$.

The probability measure we're interested in here is the discret measure that associates a probability $\frac{1}{n}$ to each input x_i of the dataset (and a probability of 0 for any x that does not belong to the dataset). Write $\hat{\rho}$ this measure, the associated inner product is

$$\langle F, G \rangle_n = \langle F, G \rangle_{\hat{\rho}} = \frac{1}{n} \sum_{i=1}^{n} F(x_i)G(x_i).$$

The functional we are interested in is the empirical risk $\mathcal{R}_n$, and one can compute[3] its Fréchet derivative $\nabla_F \mathcal{R}_n(F)$. So, we can write a gradient descent:

$$F_{t+1} = F_t - \alpha_t \nabla_F \mathcal{R}_n(F_t).$$

However, the Fréchet derivative is a function (indeed, a set of functions), known only in the datapoints x_i. It does not allows generalizing and it is not a practical object for computing. The idea is therefore to "restrict" this gradient to the hypothesis space $\mathcal{H}$ of interest. By "restricting" the gradient, we mean here looking for the function of $\mathcal{H}$ being the more collinear to the gradient (with a comparable norm). This way, we follow approximately the direction of the gradient, so we reduce the empirical risk. Searching for this collinear function amounts to solve the following optimization problem:

$$f_t \in \underset{f \in \mathcal{H}}{\text{argmax}} \frac{\langle \nabla_F \mathcal{R}_n(F_t), f \rangle_n}{\|f\|_n}.$$

Then, we apply the gradient update, but with the functional gradient being replaced by its approximation f_t:

$$F_t = F_{t-1} - \alpha_t f_t.$$

Thus, we compute an additive model, as before. Yet, it is here obtained as a *restricted functional gradient descent*.

[3] Generally speaking, all rules of gradient computation, such as the composition rule, apply. From a practical point of view, as only the datapoints x_i do matter (given the considered discrete measure), one can see the function F as a vector $(F(x_1), \ldots, F(x_n))^\top$ and take the derivative of the loss respectively to each component seen as a variable. We do this later for the exponential loss.

We apply now this idea to the exponential loss. We recall that the associated empirical risk is

$$\mathcal{R}_n(F) = \frac{1}{n}\sum_{i=1}^{n} e^{-y_i F(x_i)}.$$

For a function F, the functional gradient is the set

$$\nabla\mathcal{R}_n(F) = \left\{ G \in \mathbb{R}^{\mathcal{X}} : G(x_i) = -y_i e^{-y_i F(x_i)} \right\}. \tag{22.11}$$

To get this result, we can apply the informal method explained in footnote 3 page 225:

$$G(x_i) = \frac{\partial L(y_i, F(x_i))}{\partial F(x_i)} = \frac{\partial e^{-y_i F(x_i)}}{\partial F(x_i)} = -y_i e^{-y_i F(x_i)}.$$

There remains to project this gradient on the hypothesis space. Here we consider $\mathcal{H} \subset \{-1,+1\}^{\mathcal{X}}$ (for example, $\mathcal{H}$ can be the set of decision stumps, as usual), thus for $f \in \mathcal{H}$ we have $\|f\|_n^2 = \frac{1}{n}\sum_{i=1}^{n} f^2(x_i) = 1$. The update rule is thus

$$F_{t+1} = F_t - \alpha_t f_t \text{ with } f_t = \operatorname*{argmax}_{f\in\mathcal{H}} \frac{\langle \nabla\mathcal{R}_n(F_t), f\rangle_n}{\|f\|_n} = \operatorname*{argmax}_{f\in\mathcal{H}} \langle \nabla\mathcal{R}_n(F_t), f\rangle_n.$$

Using the result of Eq. (22.11), we have

$$\begin{aligned}\langle \nabla\mathcal{R}_n(F_t), f\rangle_n &= \frac{1}{n}\sum_{i=1}^{n} G(x_i) f(x_i) \\ &= \frac{1}{n}\sum_{i=1}^{n} e^{-y_i F_t(x_i)}(-y_i f(x_i)).\end{aligned}$$

On the other hand, notice that

$$-y_i f(x_i) = \begin{cases} 1 & \text{if } y_i \neq f(x_i) \\ -1 & \text{if } y_i = f(x_i) \end{cases} = 2\mathbb{I}_{\{y_i \neq f(x_i)\}} - 1.$$

Therefore, we have (as $e^{-y_i F_t(x_i)}$ does not depend on f)

$$\operatorname*{argmax}_{f\in\mathcal{H}} \langle \nabla\mathcal{R}_n(F_t), f\rangle_n = \operatorname*{argmax}_{f\in\mathcal{H}} \frac{1}{n}\sum_{i=1}^{n} e^{-y_i F_t(x_i)} \mathbb{I}_{\{y_i \neq f(x_i)\}}$$

The update rule is therefore

$$\begin{aligned} F_{t+1} &= F_t - \alpha_t f_t \text{ with } f_t = \operatorname*{argmax}_{f\in\mathcal{H}} \frac{1}{n}\sum_{i=1}^{n} e^{-y_i F_t(x_i)} \mathbb{I}_{\{y_i \neq f(x_i)\}} \\ &= F_t + \alpha_t f_t \text{ with } f_t = \operatorname*{argmin}_{f\in\mathcal{H}} \frac{1}{n}\sum_{i=1}^{n} e^{-y_i F_t(x_i)} \mathbb{I}_{\{y_i \neq f(x_i)\}}, \end{aligned} \tag{22.12}$$

the last line being obtained by injecting the negative sign in the optimization problem (recall that f takes only the values ± 1). As expressed in Eq. (22.12), the (negative of the) restricted gradient is exactly the classifier computed in line 3 of Alg. 21, aiming at minimizing the error of line 4 of the same algorithm.

There remains to choose the learning rate. The convex optimization theory offers a bunch of choices for this. For example, a classic (but not necessarily wise) choice consists in setting α_t such that $\sum_{t\geq 1} \alpha_t = +\infty$ and $\sum_{t\geq 1} \alpha_t^2 < \infty$ (typically, $\alpha_t \propto \frac{1}{t}$). Here we can perform a *line search*, that is we can look for the learning rate that will imply the maximum decrease of the empirical risk. Formally, this is donne by solving the following optimization problem:

$$\alpha_t = \operatorname*{argmin}_{\alpha>0} \mathcal{R}_n(F_t + \alpha f_t).$$

Using the same techniques as before, we have

$$\begin{aligned} n\mathcal{R}_n(F_t + \alpha f_t) &= \sum_{i=1}^{n} e^{-y_i(F_t(x_i)+\alpha f_t(x_i))} \\ &= e^{-\alpha} \sum_{i:y_i=f(x_i)} e^{-y_i F_t(x_i)} + e^{\alpha} \sum_{i:y_i \neq f(x_i)} e^{-y_i F_t(x_i)} \\ &= (e^{\alpha} - e^{-\alpha}) \sum_{i=1}^{n} e^{-y_i F_t(x_i)} \mathbb{I}_{\{y_i \neq f_t(x_i)\}} + e^{\alpha} \sum_{i=1}^{n} e^{-y_i F_t(x_i)}. \end{aligned}$$

Solving for α, we get

$$\nabla_\alpha \mathcal{R}_n(F_t + \alpha f_t) = 0 \Leftrightarrow \alpha_t = \frac{1}{2} \ln \left(\frac{1-\epsilon_t}{\epsilon_t} \right) \text{ with } \epsilon_t = \frac{\sum_{i=1}^{n} e^{-y_i F_t(x_i)} \mathbb{I}_{\{y_i \neq f_t(x_i)\}}}{\sum_{i=1}^{n} e^{-y_i F_t(x_i)}}.$$

This is exactly the learning rate of AdaBoost (see line 5 of Alg. 21).

Therefore, we have derived AdaBoost in a third way, from an optimization perspective. It is of high interest, as it allows relying on the whole optimization field. The interested reader can refer to Mason et al. (1999); Friedman (2001); Grubb and Bagnell (2011); Geist (2015b) for more about this kind of approach.

Part VII

Sequential Decision Making

Chapter 23

Bandits

A (multi-armed) *bandit* problem is the most basic example of a sequential decision problem with a trade-off between exploration and exploitation. A gambler (or player, or forecaster) is facing a number of options (or actions). At each time step, the player chooses an option and receives a reward (or a payoff). The goal is to maximize the total sum of rewards obtained in a sequence of allocations. A tradeoff between exploration and exploitation arises: the player must balance the exploitation of actions that did well in the past and the exploration of actions that could give higher reward in the future. The name "bandit" comes from the American slang "one-armed bandit" that refers to a slot machine: the gambler is facing many slot machines at once in a casino and must repeatedly choose where to insert the next coin.

Bandits have numerous applications. They have first been introduced by Thompson (1933) for studying clinical trials (different treatments are available for a given disease, one must choose which treatment to use on the next patient). Nowadays, they are widely used in online services (for adapting the service to the user's individual sequence of requests). For example, they can be used for ad placement (determining which advertisement to display on a web page, see for example Chapelle and Li (2011)). They can also be used in cognitive radio for opportunistic spectrum access (Jouini et al., 2012). They can also be used less directly. For example, they are at the core of the MoGo program (Gelly et al., 2006) that plays Go at world-class level (see also Munos (2014)).

The rest of this chapter is organized as follows. In Sec. 23.1, we formalize the stochastic bandit problem. In Sec. 23.2, we explain the idea of "optimism in the face of uncertainty" and introduce concentration inequalities. In Sec. 23.3 we present the classical UCB (Upper Confidence Bound) strategy and prove its effectiveness. In Sec. 23.4, we discuss briefly other kinds of bandits and problems. The material presented in this chapter is largely inspired from the monograph of Bubeck and Cesa-Bianchi (2012).

23.1 The stochastic bandit problem

The basic stochastic multi-armed bandit problem is formalized as follows. Each arm (or option, or action) $i \in \{1, \dots, K\}$ is associated to an *unknown* probability measure ν_i. In the sequel, we assume that the rewards are bounded (in $[0, 1]$, without loss of generality, that is $\nu_i([0, 1]) = 1$). At each time step $t = 1, 2, \dots$, the player chooses and arm $I_t \in \{1, \dots, K\}$ (based on past choices and observations) and receives a reward $X_{I_t,t}$ drawn from ν_{I_t}, independently from the past. We write $\mu_i = \mathrm{E}[X_{i,t}] = \int x d\nu_i(x)$ the expectation of the ith arm and define

$$\mu_* = \max_{1 \le i \le K} \mu_i \text{ and } i_* \in \operatorname*{argmax}_{1 \le i \le K} \mu_i$$

the highest expectation and the corresponding arm (which is not necessarily unique).

The ideal (but unreachable) strategy would consist in choosing systematically $I_t = i_*$. Therefore, the quality of a strategy can be measured with the *regret*, defined as the cumulative difference

(in expectation) between the optimal arm and chosen arms[1] after n rounds:

$$\mathrm{R}_n = n\mu_* - \mathrm{E}\left[\sum_{t=1}^{n} \mu_{I_t}\right]. \tag{23.1}$$

The better the strategy, the lower the regret. So, we should design the sequential decisions such as minimizing this quantity.

Next, we formulate this regret differently. Write

$$T_i(s) = \sum_{t=1}^{s} \mathbb{I}_{\{I_t=i\}}$$

the number of times the player selected arm i during the first s rounds and

$$\Delta_i = \mu_* - \mu_i$$

the suboptimality of arm i. Obviously, we have that

$$\sum_{i=1}^{K} T_i(n) = n.$$

On the other hand, one can easily check that

$$\sum_{t=1}^{n} \mu_{I_t} = \sum_{i=1}^{K} \mu_i T_i(n).$$

Therefore, the regret can be rewritten as follows:

$$\begin{aligned}
\mathrm{R}_n &= n\mu_* - \mathrm{E}\left[\sum_{t=1}^{n} \mu_{I_t}\right] \\
&= \left(\mathrm{E}\left[\sum_{i=1}^{K} T_i(n)\right]\right)\mu_* - \mathrm{E}\left[\sum_{i=1}^{K} \mu_i T_i(n)\right] \\
&= \sum_{i=1}^{K} \Delta_i \mathrm{E}\left[T_i(n)\right].
\end{aligned} \tag{23.2}$$

Therefore, a good strategy should control $\mathrm{E}\left[T_i(n)\right]$ for $i \neq i_*$, the (expected) number of times a suboptimal arm is played.

23.2 Optimism in the face of uncertainty

At time step t, the player has gathered a number of observations (a reward each time an arm has been pulled). From this, he can estimate the expectation for each arm (by computing the empirical mean). A strategy could be to act greedily respectively to these estimated means (select the arm with the highest empirical mean). However, this would be a bad strategy. Assume for examples that rewards are drawn according to Bernoulli distributions (that is the reward is either 0 or 1). Then this (pure exploitation) strategy would lead to always select the first arm that provided a reward (which is not necessarily the optimal arm, obviously). One should add some exploration to this strategy. For example, let $0 < \epsilon < 1$ be a user-defined value. A possible strategy (called ϵ-greedy strategy) is to act greedily respectively to the estimated expectation with probability $1-\epsilon$, and to choose an arm at random with probability ϵ.

Now, assume that we are able to construct a high probability confidence interval for each arm (with high probability, the true expectation μ_i of arm i is in a given interval), as illustrated in Fig. 23.1. In this exemple, we can say (still with high probability) that arm 2 is better than arm

[1] Notice that I_t is a random quantity, as it depends on past observed rewards.

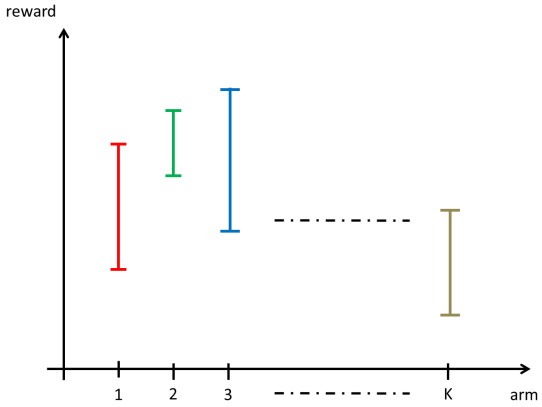


Figure 23.1: Optimism in the face of uncertainty.

K (as the lower bound of the interval of arm 2 is higher than the higher bound of arm K), but it is more difficult to tell which of arms 2 and 3 is the better. Optimism in the face of uncertainty consists in acting greedily respectively to the most "favorable" case, here to act greedily respectively to the higher upper bound of the arm's confidence interval. In Fig. 23.1, *optimism in the face of uncertainty* consists in choosing arm 3. Computing these confidence intervals will be done next through the use of a *concentration inequality*. The related strategy and the analysis of its regret are provided in Sec. 23.3.

Let $X_1, \dots, X_n$ be i.i.d. (independent and identically distributed) random variables. Write $\mu = \mathrm{E}[X_1]$ their common expectation and μ_n the related empirical mean:

$$\mu_n = \frac{1}{n} \sum_{i=1}^{n} X_i.$$

Typically, these random variables are the rewards obtained for pulling a given arm n times. The question we would like to answer is: how close is μ_n to μ (with some probability)? We first give a general answer to this question before instantiating it to the case of bounded random variables.

Theorem 23.1 (*Hoeffding's inequality* (Hoeffding, 1963; Bubeck and Cesa-Bianchi, 2012))**.** *Assume that there exists a convex function $\psi : \mathbb{R}_+ \to \mathbb{R}_+$ such that*

$$\forall \lambda \geq 0, \quad \ln \mathrm{E}\left[e^{\lambda(X_1 - \mu)}\right] \leq \psi(\lambda) \text{ and } \ln \mathrm{E}\left[e^{\lambda(\mu - X_1)}\right] \leq \psi(\lambda).$$

Define the Legendre-Fenchel transform of ψ as

$$\psi_*(\epsilon) = \sup_{\lambda \geq 0} \left(\lambda \epsilon - \psi(\lambda)\right).$$

Then:

$$\mathrm{P}\left(\mu_n - \mu \geq \epsilon\right) \leq e^{-n\psi_*(\epsilon)} \text{ and } \mathrm{P}\left(\mu - \mu_n \geq \epsilon\right) \leq e^{-n\psi_*(\epsilon)}.$$

Before proving this result (called a *concentration inequality*, as it states how the empirical mean concentrates around the expectation), we give some intuitions about its meaning. The moment condition (the assumption about the existence of the ψ function) provides information about the tail of the distribution, notably how it concentrates around its mean. For example, if $X \sim \mathcal{N}(\mu, \sigma^2)$ (X is Gaussian of mean μ and variance σ^2), it is a standard result of the probability theory that for any $\lambda \in \mathbb{R}$, we have $\ln \mathrm{E}\left[\exp \lambda(X - \mu)\right] = \frac{\sigma^2}{2}$. We will see later that we have a similar result for bounded random variables. The Legendre-Fenchel transform is a standard tool in convex optimization (how it is introduced will be clear in the proof). For example, for $\psi(\lambda) \propto \lambda^2$, we have $\psi_*(\epsilon) \propto \epsilon^2$. Next, we explain why this result is indeed a confidence interval. A direct corollary of

this theorem is

$$\begin{aligned}
\mathrm{P}\left(|\mu_n - \mu| \geq \epsilon\right) &= \mathrm{P}\left(\{\mu_n - \mu \geq \epsilon\} \cup \{\mu - \mu_n \geq \epsilon\}\right) \\
&\leq \mathrm{P}\left(\mu_n - \mu \geq \epsilon\right) + \mathrm{P}\left(\mu - \mu_n \geq \epsilon\right) && \text{(union bound)} \\
&\leq 2e^{-n\psi_*(\epsilon)} && \text{(by Th. 23.1).} \qquad (23.3)
\end{aligned}$$

Write δ this upper-bound on the probability

$$\delta = 2e^{-n\psi_*(\epsilon)} \Leftrightarrow \epsilon = \psi_*^{-1}\left(\frac{1}{n}\ln\frac{2}{\delta}\right).$$

Eq. (23.3) can thus equivalently be rewritten as

$$\mathrm{P}\left(|\mu_n - \mu| \geq \psi_*^{-1}\left(\frac{1}{n}\ln\frac{2}{\delta}\right)\right) \leq \delta \Leftrightarrow \mathrm{P}\left(|\mu_n - \mu| \leq \psi_*^{-1}\left(\frac{1}{n}\ln\frac{2}{\delta}\right)\right) \geq 1 - \delta.$$

Alternatively, we can say that with probability at least $1 - \delta$, we have

$$|\mu_n - \mu| \leq \psi_*^{-1}\left(\frac{1}{n}\ln\frac{2}{\delta}\right)$$

This is called a *PAC (Probably Approximately Correct)* result. This is is indeed a confidence interval, as it says that with probability at least $1 - \delta$ we have

$$\mu \in \left[\mu_n - \psi_*^{-1}\left(\frac{1}{n}\ln\frac{2}{\delta}\right), \mu_n + \psi_*^{-1}\left(\frac{1}{n}\ln\frac{2}{\delta}\right)\right].$$

This is exactly the kind of result we were looking for. Next, we prove the theorem.

Proof of Th. 23.1. The proof is based on what is called a *Chernoff argument*. Let $\lambda > 0$, we have

$$\begin{aligned}
\mathrm{P}\left(\mu_n - \mu \geq \epsilon\right) &= \mathrm{P}\left(\sum_{i=1}^{n}(X_i - \mu) \geq n\epsilon\right) \\
&= \mathrm{P}\left(e^{\lambda\sum_{i=1}^{n}(X_i-\mu)} \geq e^{\lambda n\epsilon}\right).
\end{aligned}$$

Recall that the Markov's inequality states[2] that if Y is a positive random variable and c a positive constant we have

$$\mathrm{P}\left(Y \geq c\right) \leq \frac{\mathrm{E}\left[Y\right]}{c}.$$

Therefore, we have

$$\begin{aligned}
\mathrm{P}\left(e^{\lambda\sum_{i=1}^{n}(X_i-\mu)} \geq e^{\lambda n\epsilon}\right) &\leq e^{-n\lambda\epsilon}\mathrm{E}\left[e^{\lambda\sum_{i=1}^{n}(X_i-\mu)}\right] && \text{(by Markov)} \\
&= e^{-\lambda n\epsilon}\prod_{i=1}^{n}\mathrm{E}\left[e^{\lambda(X_i-\mu)}\right] && \text{(by independence)} \\
&= e^{-\lambda n\epsilon}\left(\mathrm{E}\left[e^{\lambda(X_1-\mu)}\right]\right)^n && \text{(r.v. are i.d.)} \\
&= e^{-n(\lambda\epsilon - \ln \mathrm{E}[\exp(\lambda(X_1-\mu))])} \\
&\leq e^{-n(\lambda\epsilon-\psi(\lambda))}.
\end{aligned}$$

[2]Refer to any basic course on probabilities. We give the proof for completeness. We have

$$\mathrm{E}\left[Y\right] = \int Y\mathrm{dP} = \int_{Y<c} Y\mathrm{dP} + \int_{Y\geq c} Y\mathrm{dP} \geq \int_{Y\geq c} Y\mathrm{dP} \geq c\int_{Y\geq c}\mathrm{dP} = c\mathrm{P}\left(Y \geq c\right),$$

from which the Markov's inequality follows.

This being true for any $\lambda > 0$, it is true for the minimizer, thus

$$\begin{aligned}\mathrm{P}\left(\mu_n - \mu \geq \epsilon\right) &\leq \inf_{\lambda>0} e^{-n(\lambda\epsilon - \psi(\lambda))} \\ &= e^{-n\sup_{\lambda>0}(\lambda\epsilon-\psi(\lambda))} \\ &= e^{-n\psi_*(\epsilon)}.\end{aligned}$$

This shows the first inequality, the proof for the second one being the same. □

When introducing the stochastic bandit problem in Sec. 23.1, we have assumed that the rewards are bounded (this is not mandatory, but usual). In this case, the bound can be instantiated, thanks to the following Lemma due to Hoeffding (1963), that specify ψ in this case.

Lemma 23.1 (Hoeffding (1963))**.** *Let Y be a random variable such that $\mathrm{E}[Y] = 0$ and $c \leq Y \leq d$ almost surely*[3]*. Then, for any $s \geq 0$, we have*

$$\mathrm{E}\left[e^{sY}\right] \leq e^{s^2\frac{(d-c)^2}{8}}.$$

Proof. Let $s > 0$. The function $x \to e^{sx}$ is convex, thus $e^{s(tx+(1-t)y)} \leq te^{sx} + (1-t)e^{sy}$. Notice also that for any x

$$x = \frac{d-x}{d-c}c + \frac{x-c}{d-c}d.$$

Therefore, we have for the r.v. Y:

$$e^{sY} \leq \frac{d-Y}{d-c}e^{sc} + \frac{Y-c}{d-c}e^{sd}.$$

Taking the expectation (recall that $\mathrm{E}[Y] = 0$):

$$\begin{aligned}\mathrm{E}\left[e^{sY}\right] &\leq \frac{d}{d-c}e^{sc} + \frac{-c}{d-c}e^{sd} \\ &= e^{sc}\left(\frac{d}{d-c} + \frac{-c}{d-c}e^{s(d-c)}\right).\end{aligned}$$

Define $p = \frac{-c}{d-c} > 0$ (which implies that $\frac{d}{d-c} = 1-p$) and $u = s(d-c)$. Therefore, $sc = -pu$ and we can write

$$\begin{aligned}\mathrm{E}\left[e^{sY}\right] &\leq e^{-pu}\left(1-p+pe^u\right) = e^{\varphi(u)} \\ \text{with } \varphi(u) &= -pu + \ln\left(1-p+pe^u\right).\end{aligned}$$

We will bound $\varphi(u)$. We have that $\varphi(0) = 0$ and $\varphi'(0) = 0$. The second derivative is

$$\varphi''(u) = \frac{pe^u(1-p)}{(1-p+pe^u)^2} \leq \frac{1}{4}.$$

For this last statement, note that by writing $t = \frac{pe^u}{1-p+pe^u} > 0$ we have $\varphi''(u) = t(1-t)$ which is obviously bounded by $\frac{1}{4}$. From the Taylor-Lagrange formula, there exists a ξ such that

$$\varphi(u) = \varphi(0) + \varphi'(0)u + \varphi''(\xi)\frac{u^2}{2} \leq \frac{u^2}{8} = \frac{s^2(d-c)^2}{8},$$

which proves the result. □

From this, we have a direct corollary of Th. 23.1.

Corollary 23.1 (Hoeffding (1963))**.** *Assume that $0 \leq X_1 \leq 1$ almost surely. Then we have*

$$\mathrm{P}\left(\mu_n - \mu \geq \epsilon\right) \leq e^{-2n\epsilon^2} \text{ and } \mathrm{P}\left(\mu - \mu_n \geq \epsilon\right) \leq e^{-2n\epsilon^2}.$$

[3] Obviously, $c \leq 0 \leq d$.

Proof. We have that

$$0 \leq X_1 \leq 1 \Leftrightarrow -\mu \leq X_1 - \mu \leq 1 - \mu.$$

Applying Lemma 23.1 with $Y = X_1 - \mu$, $c = -\mu$ and $d = 1 - \mu$ we obtain $\psi(\lambda) = \frac{\lambda^2}{8}$ (the same hold for $\mu - X_1$). To obtain the Legendre-Fenchel transform, we set the gradient of $\lambda\epsilon - \frac{\lambda^2}{8}$ to zero, which gives $\lambda = 4\epsilon$, thus $\psi_*(\epsilon) = 2\epsilon^2$. Applying Th. 23.1 allows concluding. □

We will next apply these results to derive a strategy for the bandit problem.

23.3 The UCB strategy

Recall that the optimism in the face of uncertainty consists in acting greedily respectively to the upper bound of a confidence interval. The *UCB (Upper Confidence Bound) strategy* applies this principle. Write $\mu_{i,s}$ the sample mean of rewards obtained by pulling arm i for s times:

$$\mu_{i,s} = \frac{1}{s} \sum_{t=1}^{s} X_{i,t}.$$

As we are looking for an upper bound on μ_i, we have from Th. 23.1 that with probability at least $1 - \delta$,

$$\mu_i < \mu_{i,s} + \psi_*^{-1}\left(\frac{1}{n} \ln \frac{1}{\delta}\right).$$

With the choice of $\delta = \frac{1}{t^\alpha}$ where $\alpha > 0$ is an input parameter (and with $s = T_i(t-1)$, the number of times arm i has been played before round t), we obtain the so-called (α, ψ)-UCB strategy of Bubeck and Cesa-Bianchi (2012):

$$I_t \in \operatorname*{argmax}_{1 \leq i \leq K} \left(\mu_{i,T_i(t-1)} + \psi_*^{-1}\left(\frac{\alpha \ln t}{T_i(t-1)}\right)\right).$$

If the rewards are bounded in $[0, 1]$, we obtain the original UCB (Upper Confidence Bound) strategy of Auer et al. (2002) (using the results in the proof of Cor. 23.1, that is $\psi_*(\epsilon) = 2\epsilon^2 \Leftrightarrow \psi_*^{-1}(u) = \sqrt{\frac{u}{2}}$):

$$I_t \in \operatorname*{argmax}_{1 \leq i \leq K} \left(\mu_{i,T_i(t-1)} + \sqrt{\frac{\alpha \ln t}{2T_i(t-1)}}\right).$$

Therefore, we end up with a simple strategy that only requires updating empirical means as arms as pulled, and to act greedily respectively to the above quantity (which is the empirical mean plus a kind of bonus). An important question is to know what regret is suffered by these strategies. The answer is given by the next theorem.

Theorem 23.2 (Auer et al. (2002); Bubeck and Cesa-Bianchi (2012))**.** *Assume that the rewards distributions satisfy the assumption of Th. 23.1. Then the (α, ψ)-UCB strategy with $\alpha > 2$ satisfies*

$$\mathrm{R}_n \leq \sum_{i:\Delta_i > 0} \left(\frac{\alpha \Delta_i}{\psi_*\left(\frac{\Delta_i}{2}\right)} \ln n + \frac{\alpha}{\alpha - 2}\right).$$

If rewards are bounded in $[0, 1]$, the bound on the regret simplifies as (using the fact that $\psi_*(\epsilon) = 2\epsilon^2$):

$$\mathrm{R}_n \leq \sum_{i:\Delta_i > 0} \left(\frac{2\alpha}{\Delta_i} \ln n + \frac{\alpha}{\alpha - 2}\right).$$

This tells that each suboptimal arm is chosen no more that a logarithmic number of times ($\ln n$), and that arms close to the optimal one are chosen more often ($\frac{1}{\Delta_i}$). We prove the result now.

Proof of Th. 23.2. Assume without loss of generality that $I_t = i \neq i*$. Then, at least on of the tree following equations must be true:

$$\mu_{i_*,T_{i_*}(t-1)} + \psi_*^{-1}\left(\frac{\alpha \ln t}{T_{i_*}(t-1)}\right) \leq \mu_* \tag{23.4}$$

$$\mu_{i,T_i(t-1)} > \mu_i + \psi_*^{-1}\left(\frac{\alpha \ln t}{T_i(t-1)}\right) \tag{23.5}$$

$$T_i(t-1) < \frac{\alpha \ln n}{\psi_*\left(\frac{\Delta_i}{2}\right)} \tag{23.6}$$

Eq. (23.4) states that the upper-bound for the optimal arm is below the associated mean, Eq. (23.5) states that the lower-bound for the considered arm is above the associated mean and Eq. (23.6) means that arm i has not been pulled enough. If the three equations were false, we would have

$$\begin{aligned}
\mu_{i_*,T_{i_*}(t-1)} + \psi_*^{-1}\left(\frac{\alpha \ln t}{T_{i_*}(t-1)}\right) &> \mu_* && \text{by (23.4) false} \\
&= \mu_i + \Delta_i && \text{by def. of } \Delta_i \\
&\geq \mu_i + 2\psi_*^{-1}\left(\frac{\alpha \ln t}{T_i(t-1)}\right) && \text{by (23.6) false} \\
&\geq \mu_{i,T_i(t-1)} + \psi_*^{-1}\left(\frac{\alpha \ln t}{T_i(t-1)}\right) && \text{by (23.5) false,}
\end{aligned}$$

which implies that $I_t \neq i$, which is a contradiction.

Recall that for controlling the regret it is enough to control the (expected) number of times each arm has been played (Eq. (23.1) being equivalent to Eq. (23.2)). Define u as

$$u = \left\lceil \frac{\alpha \ln n}{\psi_*\left(\frac{\Delta_i}{2}\right)} \right\rceil.$$

We have that

$$\begin{aligned}
\mathrm{E}\left[T_i(n)\right] &= \mathrm{E}\left[\sum_{t=1}^{n} \mathbb{I}_{\{I_t=i\}}\right] \\
&\leq u + \mathrm{E}\left[\sum_{t=u+1}^{n} \mathbb{I}_{\{I_t=i \text{ and (23.6) is false}\}}\right] \\
&\leq u + \mathrm{E}\left[\sum_{t=u+1}^{n} \mathbb{I}_{\{\text{(23.4) or (23.5) is true}\}}\right] \\
&\leq u + \sum_{t=u+1}^{n} \left(\mathrm{P}\left(\text{(23.4) is true}\right) + \mathrm{P}\left(\text{(23.5) is true}\right)\right).
\end{aligned}$$

We can bound the probability of event (23.4) as follows:

$$\begin{aligned}
\mathrm{P}\left(\text{(23.4) is true}\right) &\leq \mathrm{P}\left(\exists s \in \{1, \dots t\} : \mu_{i_*,s} + \psi_*^{-1}\left(\frac{\alpha \ln t}{s}\right) \leq \mu_*\right) \\
&\leq \sum_{s=1}^{t} \mathrm{P}\left(\mu_{i_*,s} + \psi_*^{-1}\left(\frac{\alpha \ln t}{s}\right) \leq \mu_*\right) \\
&\leq \sum_{s=1}^{t} \frac{1}{t^\alpha} = \frac{1}{t^{\alpha-1}},
\end{aligned}$$

where we used a union bound and Hoeffding with $\delta = t^{-\alpha}$. The same bound holds for the probability of event (23.5):

$$\mathrm{P}\left(\text{(23.5) is true}\right) \leq \frac{1}{t^{\alpha-1}}.$$

Combining these results, we have

$$\begin{aligned}\mathrm{E}\left[T_i(n)\right] &\leq u + 2\sum_{t=u+1}^{n} \frac{1}{t^{\alpha-1}} \\ &\leq \frac{\alpha \ln n}{\psi_*\left(\frac{\Delta_i}{2}\right)} + 1 + 2\int_1^{\infty} \frac{1}{t^{\alpha-1}} dt \\ &= \frac{\alpha \ln n}{\psi_*\left(\frac{\Delta_i}{2}\right)} + \frac{\alpha}{\alpha - 2}.\end{aligned}$$

Injecting this result in Eq. (23.2) concludes the proof. □

Some strategies allow achieving a lower regret (and even optimal ones, in the sense that lower bounds on the regret—for any possible strategy—can be proven), at the price of more complicated analysis. Yet, UCB is a popular strategy, simple to apply and quite effective in practice. See Bubeck and Cesa-Bianchi (2012, Ch. 2) for more details.

23.4 More on bandits

There are other kinds of problems for stochastic bandit. For example, the best arm identification problem consists in identifying the best arm given a fixed budget or a fixed confidence (*e.g.*, see Gabillon et al. (2012)), not paying much attention to the sum of rewards obtained when pulling arm (for example, a company may want to identify the best product among K variants before actually placing it on the market). The stochastic bandit can also be addressed from a Bayesian viewpoint (Thompson, 1933; Korda et al., 2013).

There are also other kinds of bandit. In adverserial bandit, the reward does not follows a distribution, it is given by an opponent (or adversary). In contextual bandits, a side information is associated to each arm. In linear bandit, there is some structure in the reward function (that allows handling a possibly infinite number of arms). In Markovian bandits, each arm is associated to a Markov process and pulling an arm causes a stochastic transition of the underlying chain. There are (many) other kinds of bandits. See Bubeck and Cesa-Bianchi (2012) and references therein for a deeper introduction to the subject.

Chapter 24

Reinforcement learning

Reinforcement learning (RL) can be broadly seen as the machine learning answer to the control problem. In this paradigm, an agent is interacting with the world by tacking actions and observing the resulting configuration of the world (in a sequential manner). This agent receives (numerical) rewards (given by some oracle) that are a local information about the quality of the control. The aim of this agent is to learn the sequence of actions such as maximizing some notion of *cumulative* reward. This chapter provides an introduction to the field of reinforcement learning, more can be found on reference textbooks (Sutton and Barto, 1998; Bertsekas and Tsitsiklis, 1996; Szepesvári, 2010; Sigaud and Buffet, 2013). This field is inspired by behaviorial psychology (this explains part of the vocabulary, such as the notion of reward) and has connections to computational neuroscience, yet this chapter focuses on the mathematical and learning aspects.

24.1 Introduction

In *reinforcement learning*, an agent is interacting with a system (sometime called the environment, or the world), as exemplified in Fig. 24.1. At each discrete time step, the system is in a given state (or configuration) that can be observed by the agent. Based on this state, the agent apply some action. Following this action, the system transits in a new state and the agent receives a numerical reward from an oracle, this reward being a local clue of the quality of the control. The goal of the agent is to take sequential decisions such as maximizing some notion of *cumulative* reward, typically the sum of rewards gathered along the followed path. An important thing to understand is that the rewards quantify the goal of the control, and not how this goal should be reached (this is what the agent has to learn).

To clarify this paradigm, consider the simple example of a robot in a maze. The goal of the

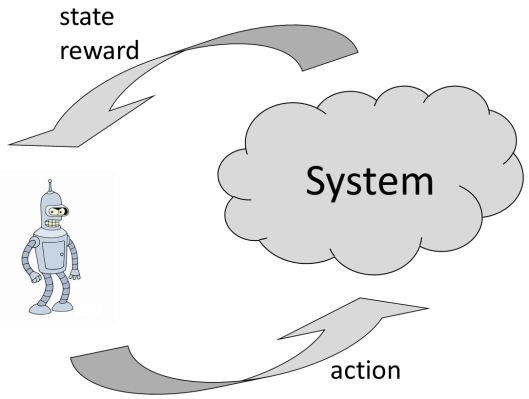


Figure 24.1: The perception-action cycle in reinforcement learning.

robot is to find the shortest path to the exit. The state of the system is the current position of the robot in the maze. Four actions are available, one for each direction. Choosing such an action amount to moving in the required direction. In this problem, the reward can be -1 for any move, except for the move that leads to the exit which is rewarded by 0. Notice that the only informative reward is given for going through the exit. Here, for any path leading to the exit, the sum of rewards is the negative of the length of the path. Therefore, maximizing the sum of rewards is equivalent to finding the shortest path to the exit.

A first issue is to formalize mathematically such a control problem. In reinforcement learning, this is widely done thanks to Markov Decision Processes (MDPs), to be presented in Sec. 24.2. A second issue is to compute the best possible control when the model (the MDP) is known, which is addressed by Dynamic Programming (DP), to be presented in Sec. 24.3. Consider again the maze problem. A smart strategy consists in starting from the exit, and then retro-propagating the possible paths until reaching the starting point. This is roughly what DP does. Puterman (1994) and Bertsekas (1995) provide reference textbooks on MDPs and DP. A third problem is to estimate this optimal strategy from data (interaction data between the agent and the system), when the model is unknown (this is reinforcement learning). This is addressed in Sec. 24.4-24.6.

24.2 Formalism

In the sequel, we write $\Delta_{\mathcal{X}}$ the set of probability measures over a discrete set $\mathcal{X}$ and $\mathcal{Y}^{\mathcal{X}}$ the set of applications from $\mathcal{X}$ to $\mathcal{Y}$.

24.2.1 Markov Decision Process

A *Markov Decision Process (MDP)* is a tuple $\{\mathcal{S}, \mathcal{A}, P, r, \gamma\}$ where:

- $\mathcal{S}$ is the (finite) state space[1];
- $\mathcal{A}$ is the (finite) action space[2];
- $P \in \Delta_{\mathcal{S}}^{\mathcal{S}\times\mathcal{A}}$ is the Markovian transition kernel. The term $P(s'|s,a)$ denotes the probability of transiting in state s' given that action a was chosen in state s. The transition kernel is Markovian because the probability to go to s' depends on the fact that action a was chosen in state s, but it does not depend on the path followed to reach this state s. This assumption is at the core everything presented here[3];
- $r \in \mathbb{R}^{\mathcal{S}\times\mathcal{A}}$ is the reward function[4], it associates the reward $r(s,a)$ for taking action a in state s. The reward function is assumed to be uniformly bounded;
- $\gamma \in (0,1)$ is a discount factor that favors shorter term rewards (see the definition of the value function, later). The closer is γ to 1, the more importance we give to far (in time) rewards. Usually, this parameter is set to a value close to 1.

So, the system is in state $s \in \mathcal{S}$, the agent chooses an action $a \in \mathcal{A}$ and get the reward $r(s,a)$, then the system transits stochastically to a new state s', this new state being drawn from the conditional probability $P(.|s,a)$.

[1] We will assume larger (countable or even infinite compact) state spaces later.

[2] It is quite difficult to consider large action spaces, but see Sec. 24.6

[3] Consider the maze exemple of Sec. 24.1. If the robot knows its position, the dynamics are indeed Markovian. If the agent has only a partial observation (for example, it knows if there are walls among him, but no more), the dynamics are no longer Markovian. This is known as partially observable MDPs, see for example Kaelbling et al. (1998). This topic will not be covered in this chapter, but note that the general strategy consists in computing something which is Markovian.

[4] One can define more generally the reward function as $r \in \mathbb{R}^{\mathcal{S}\times\mathcal{A}\times\mathcal{S}}$, that is giving a reward $r(s,a,s')$ for each transition from s to s' under action a. However, one can define an expected reward function as $\bar{r}(s,a) = \sum_{s'\in\mathcal{S}} P(s'|s,a)r(s,a,s')$, the mean reward for choosing action a in state s. As only this mean reward will be of importance in the following, we keep the notations simple.

24.2.2 Policy and value function

The strategy of the agent (the way it chooses actions) is called a *policy*[5] and is noted $\pi \in \mathcal{A}^{\mathcal{S}}$: in state s, an agent applying policy π chooses the action $\pi(s)$. The problem is to find the best policy, but this requires quantifying the quality of a policy. This is done thanks to the associated *value function* $v_\pi \in \mathbb{R}^{\mathcal{S}}$, that associates to each state the expected (transition being stochastic) and discounted (by γ) cumulative reward that is obtained by following policy π from state s:

$$v_\pi(s) = \mathbb{E}[\sum_{t=0}^{\infty} \gamma^t r(S_t, \pi(S_t))|S_0 = s, S_{t+1} \sim P(.|S_t, \pi(S_t))]. \tag{24.1}$$

In other words, if the agent starts in state s and keeps following the policy π (that is taking the action given by π whenever a decision is required), it will receive a sequence of rewards. The discounted cumulative reward of this state is the sum of gathered rewards along the (infinite) trajectory, the reward being received at the tth time step being discounted by γ^t (this favors closer rewards and allows for the sum to be finite). This quantity being random (due to the transitions being random), the value of the state is defined as the expectation of the discounted cumulative reward. There exist other criteria for quantifying the quality of a policy, but they will not be considered here[6].

A value function allows quantifying the quality of a policy, and it allows comparing policies as follows:

$$\pi_1 \geq \pi_2 \Leftrightarrow \forall s \in \mathcal{S}, \quad v_{\pi_1}(s) \geq v_{\pi_2}(s).$$

Notice that this is a partial ordering, thus two policies might not be comparable. Solving an MDP means computing the optimal policy π_* satisfying $v_{\pi_*} \geq v_\pi$, for all $\pi \in \mathcal{A}^{\mathcal{S}}$. In other words, the optimal policy satisfies

$$\pi_* \in \operatorname*{argmax}_{\pi \in \mathcal{A}^{\mathcal{S}}} v_\pi.$$

It is possible to show that such a policy exists (the result is admitted). Before showing how such an optimal policy can be computed (see Sec. 24.3), we develop the notion of value function.

24.2.3 Bellman operators

The value function, as defined in Eq. (24.1), is not practical. However, it can be rewritten in a recursive way:

$$\begin{aligned}
v_\pi(s) &= \mathbb{E}[\sum_{t=0}^{\infty} \gamma^t r(S_t, \pi(S_t))|S_0 = s, S_{t+1} \sim P(.|S_t, \pi(S_t))] \\
&= r(s, \pi(s)) + \mathbb{E}[\sum_{t=1}^{\infty} \gamma^t r(S_t, \pi(S_t))|S_0 = s, S_{t+1} \sim P(.|S_t, \pi(S_t))] \\
&= r(s, \pi(s)) + \gamma \mathbb{E}[\sum_{t=0}^{\infty} \gamma^t r(S_{t+1}, \pi(S_{t+1}))|S_0 = s, S_{t+1} \sim P(.|S_t, \pi(S_t))] \\
\Leftrightarrow v_\pi(s) &= r(s, \pi(s)) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s, \pi(s)) v_\pi(s').
\end{aligned} \tag{24.2}$$

[5] More precisely, it is a deterministic policy. One can consider more generally a stochastic policy, that is $\pi \in \Delta_{\mathcal{A}}^{\mathcal{S}}$: for each state s, $\pi(.|s)$ is a distribution over actions. This will be useful in Sec. 24.6, but for now deterministic policies are enough.

[6] Still, we can mention the finite horizon criteria, defined for a given horizon H, the associated value function being

$$v_\pi(s) = \mathbb{E}[\sum_{t=0}^{H} r(S_t, \pi(S_t))|S_0 = s, S_{t+1} \sim P(.|S_t, \pi(S_t))],$$

or the average criteria, the corresponding value being

$$v_\pi(s) = \lim_{H \to \infty} \mathbb{E}[\frac{1}{H} \sum_{t=0}^{H} r(S_t, \pi(S_t))|S_0 = s, S_{t+1} \sim P(.|S_t, \pi(S_t))].$$

In other words, computing the value of state s can be done by knowing the value of possible next states (and the probability to reach these states). Notice that this is a linear system. To see this more clearly, first notice that a function from $\mathbb{R}^{\mathcal{S}}$ can be seen as a vector, and vice-versa (the state space being finite). We introduce $P_\pi \in \mathbb{R}^{\mathcal{S}\times\mathcal{S}}$ and $r_\pi \in \mathbb{R}^{\mathcal{S}}$, defined as

$$P_\pi = (P(s'|s,\pi(s)))_{s,s'\in\mathcal{S}} \text{ and } r_\pi = (r(s,\pi(s)))_{s\in\mathcal{S}}\,.$$

The term P_π is a stochastic matrix (each row sums to one) and r_π is a vector. Using this notation, Eq. (24.2) can be written as

$$v_\pi = r_\pi + \gamma P_\pi v_\pi \Leftrightarrow v_\pi = (I-\gamma P_\pi)^{-1} r_\pi, \tag{24.3}$$

where I is the identity matrix. Notice that P_π being a stochastic matrix, its spectrum (largest eigenvalue) is bounded by 1, and as $\gamma < 1$, the matrix $(I-\gamma P_\pi)$ is indeed invertible.

We can now introduce the *Bellman evaluation operator*[7] $T_\pi : \mathbb{R}^{\mathcal{S}} \to \mathbb{R}^{\mathcal{S}}$, defined as

$$\forall v \in \mathbb{R}^{\mathcal{S}}, \quad T_\pi v = r_\pi + \gamma P_\pi v,$$

which means equivalently that componentwise we have

$$\forall s \in \mathcal{S}, \quad [T_\pi v](s) = r(s,\pi(s)) + \gamma \sum_{s'\in\mathcal{S}} P(s'|s,\pi(s))v(s').$$

This affine operator applies to any function $v \in \mathbb{R}^{\mathcal{S}}$ (not necessarily a value function corresponding to a policy) and Eq. (24.3) shows that v_π is the unique fixed point of this operator[8]:

$$v_\pi = T_\pi v_\pi.$$

Therefore, we have a tool to compute the value function of any policy, which provides useful for quantifying its quality. However, we also would like to characterize directly the optimal policy π_* (more precisely, the related value function).

Write $v_* = v_{\pi_*}$ the value function associated to an optimal policy (called the optimal value function). Assume that the optimal value function v_* is known, but not the optimal policy π_*. For any state, this policy should take the action that leads to the higher possible value, that is

$$\pi_*(s) \in \operatorname*{argmax}_{a\in\mathcal{A}} \left(r(s,a) + \gamma \sum_{s'\in\mathcal{S}} P(s'|s,a)v_*(s') \right). \tag{24.4}$$

We say that π_* is *greedy* respectively to v_*. Therefore, knowing v_*, one can compute π_*. The remaining problem is to characterize v_*. We have seen in the evaluation problem that knowing the value in the next state is sufficient to compute the value in the current state (see Eq. (24.2)). The same principle can be applied to compute the optimal value. Knowing the optimal value of the next state, the optimal value of the current state is the one that *maximizes* the sum of the immediate reward and of the discounted optimal value of the next state:

$$\forall s \in \mathcal{S}, \quad v_*(s) = \max_{a\in\mathcal{A}} \left(r(s,a) + \gamma \sum_{s'\in\mathcal{S}} P(s'|s,a)v_*(s') \right).$$

To see if this problem admits a solution, we introduce the *Bellman optimality operator* $T_* : \mathbb{R}^{\mathcal{S}} \to \mathbb{R}^{\mathcal{S}}$, defined as

$$\forall v \in \mathbb{R}^{\mathcal{S}}, \quad T_* v = \max_{\pi\in\mathcal{A}^{\mathcal{S}}} \left(r_\pi + \gamma P_\pi v\right), \tag{24.5}$$

[7] Named after Bellman (1957).

[8] Indeed, one can easily show that it is a contraction for the supremum norm defined as $\|v\|_\infty = \max_{s\in\mathcal{S}} |v(s)|$: for any $u, v \in \mathbb{R}^{\mathcal{S}}$, we have

$$\|T_\pi v - T_\pi u\|_\infty = \|\gamma P_\pi (v-u)\|_\infty \leq \gamma \|v-u\|_\infty.$$

This also shows unicity of the fixed point and a way to compute it, thanks to the Banach theorem, see later.

which means equivalently that componentwise we have

$$\forall s \in \mathcal{S}, \quad [T_* v](s) = \max_{a \in \mathcal{A}} \left(r(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a) v(s') \right). \tag{24.6}$$

This operator is a contraction in supremum norm[9]. Therefore, thanks to the Banach theorem[10], T_* admits v_* as its unique fixed point:

$$v_* = T_* v_*.$$

Next, we show how the optimal policy (or equivalently the optimal value function, as shown in Eq. (24.4)) can be computed.

24.3 Dynamic Programming

Dynamic Programming (DP) refers to the set of methods that allow solving an MDP when the model is known (that is, when the transition kernel and the reward function are analytically known). We present the more classic here, that is linear programming, value iteration and policy iteration. In practice, the model is often unknown and one has to rely on data (see Sec. 24.4 and next) and thus on *learning*, but methods for this case are often based on the classic DP paradigm.

24.3.1 Linear programming

The optimal value function v_* can be computed by solving the following linear program (writing $\mathbf{1} \in \mathbb{R}^{\mathcal{S}}$ the vector with all components equal to 1):

$$\begin{aligned} &\min_{v \in \mathbb{R}^{\mathcal{S}}} \mathbf{1}^\top v \\ &\text{subject to } v \geq T_* v. \end{aligned} \tag{24.7}$$

We start by explaining why v_* is indeed the solution of this linear program, then we express it in a less compact form.

Recall that the operator T_* applies to any $v \in \mathbb{R}^{\mathcal{S}}$ and that it is defined as $T_* v = \max_{\pi \in \mathcal{A}^{\mathcal{S}}} (r_\pi + \gamma P_\pi v)$ (see Eq. (24.5)). Let π be any policy and v be such $v \geq T_* v$. By the definition of T_*, we have that $v \geq r_\pi + \gamma P_\pi v$ (the inequality is true for any policy, in particular for π). We can repeatedly apply this inequality:

$$\begin{aligned} v &\geq r_\pi + \gamma P_\pi v && (24.8) \\ &\geq r_\pi + \gamma P_\pi \left(r_\pi + \gamma P_\pi v_\pi \right) \\ &\geq \sum_{t=0}^{\infty} \gamma^t P_\pi^t r_\pi && (24.9) \\ &= (I - \gamma P_\pi)^{-1} r_\pi = v_\pi, && (24.10) \end{aligned}$$

[9] Let $u, v \in \mathbb{R}^{\mathcal{S}}$ and write s any state. Assume without loss of generality that $[T_* v](s) \geq [T_* u](s)$. Write also $a_*^s \in \operatorname{argmax}_{a \in \mathcal{A}} (r(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a) v(s'))$. We have

$$\begin{aligned} 0 &\leq |[T_* v](s) - [T_* u](s)| = [T_* v](s) - [T_* u](s) \\ &= \max_{a \in \mathcal{A}} \left(r(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a) v(s') \right) - \max_{a \in \mathcal{A}} \left(r(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a) u(s') \right) \\ &\leq r(s,a_*^s) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a_*^s) v(s') - \left(r(s,a_*^s) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a_*^s) u(s') \right) \leq \gamma \|v - u\|_\infty. \end{aligned}$$

This being true for any s, we have $\|T_* v - T_* u\|_\infty \leq \gamma \|v - u\|_\infty$.

[10] If an operator T is a contraction, it admits a unique fixed point, which can be constructed as the limit of the sequence $v_{k+1} = T v_k$ for any initialization v_0.

where Eq. (24.9) is obtained by repeatedly applying inequality (24.8) and Eq. (24.10) uses the fact[11] that $(I-\gamma P_\pi)^{-1} = \sum_{t=0}^{\infty} \gamma^t P_\pi^t$ and that $v_\pi = (I-\gamma P_\pi)^{-1} r_\pi$ (recall Eq. (24.3)). This being true for any policy, it is true for the optimal one, and we have just shown that

$$v \geq T_* v \Rightarrow v \geq v_*.$$

Moreover, $v \geq v_*$ implies that $\mathbf{1}^\top v \geq \mathbf{1}^\top v_*$. Consequently, minimizing $\mathbf{1}^\top v$ under the constraint that $v \geq T_* v$ provides the optimal value function.

Algorithm 22 Linear programming

1: Solve

$$\min_{v \in \mathbb{R}^{\mathcal{S}}} \sum_{s \in \mathcal{S}} v(s)$$
$$\text{subject to } v(s) \geq r(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a) v(s'), \quad \forall s \in \mathcal{S}, \forall a \in \mathcal{A}$$

and get v_*.

2: **return** the policy π_* defined as

$$\pi_*(s) \in \operatorname*{argmax}_{a \in \mathcal{A}} \left(r(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a) v_*(s') \right).$$

The optimal policy can be computed from v_* as the greedy policy (recall Eq. (24.4)). The linear programming approach to solving MDP is summarized in Alg. 22 (observe that this formulation is indeed equivalent to Eq. (24.7)). This program has $|\mathcal{S}|$ variables and $|\mathcal{S} \times \mathcal{A}|$ constraints.

24.3.2 Value iteration

We have seen in Sec. 24.2 that the operator T_* is a contraction:

$$\forall u, v \in \mathbb{R}^{\mathcal{S}}, \quad \|T_* u - T_* v\|_\infty \leq \|u - v\|_\infty.$$

Therefore, the Banach fixed-point theorem states that for any initialization v_0, the sequence defined as

$$v_{k+1} = T_* v_k$$

will converge to v_*: $\lim_{k \to \infty} v_k = v_*$. This provide a simple algorithm for computing v_*. However, the convergence is asymptotic, and one should stop the iteration before. A natural stopping criterion is to check if two subsequent fonctions are closed enough (in supremum norm), that is $\|v_{k+1} - v_k\|_\infty \leq \epsilon$, for a user-defined value of ϵ.

Doing so, we obtain a function $v_k \in \mathbb{R}^{\mathcal{S}}$, which is not necessary a value function (as there is no reason that it corresponds to a policy). Yet, what we are interested in is finding a control policy. The notion of *greedy policy* can be extended to any function $v \in \mathbb{R}^{\mathcal{S}}$. We define a policy π to be greedy respectively to a function v, noted $\pi \in \mathcal{G}(v)$, as follows:

$$\pi \in \mathcal{G}(v) \Leftrightarrow T_\pi v = T_* v \Leftrightarrow \forall s \in \mathcal{S}, \pi(s) \in \operatorname*{argmax}_{a \in \mathcal{A}} \left(r(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a) v(s') \right).$$

The output of the algorithm is thus simply $\pi_k \in \mathcal{G}(v_k)$. This method is called *value iteration* and is summarized in Alg. 23.

The stopping criterion makes sense, the iterations are stopped when they do not involve much change. Yet, we can wonder how close is the computed v_k to the optimal value function v_*. We

[11] This is a Taylor expansion, valid due to P_π being a stochastic matrix and γ being strictly bounded by 1. It can also be checked algebraically.

Algorithm 23 Value iteration

Require: An initial $v_0 \in \mathbb{R}^{\mathcal{S}}$, a stopping criterion ϵ

$k = 0$

repeat

  for all $s \in \mathcal{S}$ **do**

$$v_{k+1}(s) = \max_{a \in \mathcal{A}} \left(r(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a) v_k(s') \right)$$

  end for

  $k \leftarrow k + 1$

until $\|v_{k+1} - v_k\|_\infty \leq \epsilon$

return a policy $\pi_k \in \mathcal{G}(v_k)$:

$$\pi_k(s) \in \operatorname*{argmax}_{a \in \mathcal{A}} \left(r(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a) v_k(s') \right).$$

have the following simple bound:

$$\begin{aligned}
\|v_* - v_k\|_\infty &= \|T_* v_* - T_* v_k + T_* v_k - v_k\|_\infty && \text{(recall that } v_* = T_* v_*) \\
&\leq \|T_* v_* - T_* v_k\|_\infty + \|T_* v_k - v_k\|_\infty && \text{(triangle ineq.)} \\
&\leq \gamma \|v_* - v_k\|_\infty + \|T_* v_k - v_k\|_\infty && (T_* \text{ is a } \gamma\text{-contraction}) \\
&\leq \gamma \|v_* - v_k\|_\infty + \epsilon && (T_* v_k = v_{k+1} \text{ and } \|v_{k+1} - v_k\|_\infty \leq \epsilon) \\
\Leftrightarrow \|v_* - v_k\|_\infty &\leq \frac{1}{1-\gamma} \epsilon.
\end{aligned}$$

This result tells us how close is v_k to v_* given the considered stopping criterion (there is an expansion of $\frac{1}{1-\gamma}$ of the error ϵ). Yet, it does not tell how good is the policy $\pi_k \in \mathcal{G}(v_k)$, the output of the algorithm that will be applied to the system, and this what we are really interested in. The quality of a policy is quantified by the associated value function, thus the closeness to optimality of π_k can be quantified by $\|v_* - v_{\pi_k}\|_\infty$. We have

$$\begin{aligned}
\|v_* - v_{\pi_k}\|_\infty &= \|T_* v_* - T_{\pi_k} v_k + T_{\pi_k} v_k - T_{\pi_k} v_{\pi_k}\|_\infty \\
&\leq \|T_* v_* - T_{\pi_k} v_k\|_\infty + \|T_{\pi_k} v_k - T_{\pi_k} v_{\pi_k}\|_\infty \\
&\leq \|T_* v_* - T_* v_k\|_\infty + \gamma \|v_k - v_{\pi_k}\|_\infty \\
&\leq \gamma \|v_* - v_k\|_\infty + \gamma \|v_k - v_* + v_* - v_{\pi_k}\|_\infty \\
&\leq 2\gamma \|v_* - v_k\|_\infty + \gamma \|v_* - v_{\pi_k}\|_\infty \\
\Leftrightarrow \|v_* - v_{\pi_k}\|_\infty &\leq \frac{2\gamma}{1-\gamma} \|v_* - v_k\|_\infty,
\end{aligned}$$

where we used the fact that v_* and v_{π_k} are respectively fixed points of T_* and T_{π_k}, that $T_{\pi_k} v_k = T_* v_k$ (as $\pi_k \in \mathcal{G}(v_k)$), that T_* and T_{π_k} are γ-contractions, as well as the triangle inequality. Combining this result with the preceding one, we have

$$\|v_* - v_{\pi_k}\|_\infty \leq \frac{2\gamma}{(1-\gamma)^2} \epsilon.$$

So, we can set ϵ such as having the desired control quality (note that the bound might not be tight).

24.3.3 Policy iteration

Let π be any policy, its value function v_π can be computed as the solution of the linear system in Eq. (24.3). How could we propose a better policy? A natural idea is to consider a policy being

greedy respectively to v_π, $\pi' \in \mathcal{G}(v_\pi)$. Indeed, such a policy satisfies

$$\forall s \in \mathcal{S}, \quad \pi'(s) \in \operatorname*{argmax}_{a \in \mathcal{A}} \left(r(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a) v_\pi(s') \right),$$

which implies that

$$\forall s \in \mathcal{S}, \quad r(s,\pi'(s)) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,\pi'(s)) v_\pi(s') \geq r(s,\pi(s)) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,\pi(s)) v_\pi(s') = v_\pi(s),$$

or more compactly that $T_{\pi'} v_\pi = T_* v_\pi \geq T_\pi v_\pi = v_\pi$. Yet, this is not enough to tell that π' is *better* than π (that is, that $v_{\pi'} \geq v_\pi$).

This result is indeed true, it can be shown as follows:

$$\begin{aligned}
& T_{\pi'} v_\pi = T_* v_\pi \geq T_\pi v_\pi = v_\pi \\
\Leftrightarrow & r_{\pi'} + \gamma P_{\pi'} v_\pi \geq v_\pi \\
\Leftrightarrow & r_{\pi'} + \gamma P_{\pi'} v_{\pi'} + \gamma P_{\pi'} (v_\pi - v_{\pi'}) \geq v_\pi \\
\Leftrightarrow & (I - \gamma P_{\pi'}) v_{\pi'} \geq (I - \gamma P_{\pi'}) v_\pi \\
\Rightarrow & v_{\pi'} \geq v_\pi.
\end{aligned}$$

We used notably the fact that $r_{\pi'} + \gamma P_{\pi'} v_{\pi'} = T_{\pi'} v_{\pi'} = v_{\pi'}$ and (for the last line) the fact that pre-multiplying a componentwise inequality by a positive matrix does not change the inequality[12].

This suggests the following algorithm. Choose an initial policy π_0 and then iterates as follows:

1. solve $T_{\pi_k} v_{\pi_k} = v_{\pi_k}$ (this is called the *policy evaluation* step);
2. compute $\pi_{k+1} \in \mathcal{G}(v_{\pi_k})$ (this is called the *policy improvement* step).

We have shown that $v_{\pi_{k+1}} \geq v_{\pi_k}$, so either $v_{\pi_{k+1}} > v_{\pi_k}$ or $v_{\pi_{k+1}} = v_{\pi_k}$. In this equality case, we have

$$T_* v_{\pi_k} = T_{\pi_{k+1}} v_{\pi_k} = T_{\pi_{k+1}} v_{\pi_{k+1}} = v_{\pi_{k+1}} = v_{\pi_k}.$$

This means that v_{π_k} is the fixed point of T_*, and thus that $v_{\pi_k} = v_*$, that is $\pi_{k+1} = \pi_*$ is the optimal policy. This suggests to stop iterations when $v_{\pi_{k+1}} = v_{\pi_k}$. Moreover, the number of policies being finite (it is $|\mathcal{A}|^{|\mathcal{S}|}$), the number of iterations is finite (bounded by the number of policies). This method is called *policy iteration* and is summarized in Alg. 24.

Each iteration of policy iteration has a higher computational cost than one of value iteration (as a linear system has to be solved), but it will converges in finite time. Moreover, it converges empirically often very quickly (only few iterations are required).

24.4 Approximate Dynamic Programming

In Sec. 24.3, we have studied methods for solving MDPs. They can be used if the state and action spaces are small enough for a value function to be explicitly represented and they require the model (that is, the transition kernel and the reward function) to be known. Unfortunately, it is far from being the usual case:

- the state space can be too large[13] (even continuous[14]) for the value function to be represented exactly. In this case, one can adopt for example a linear parametrization for the value function

[12] For two vectors u and v satisfying $u \geq v$, and for a matrix A having positive elements, we have $Au \geq Av$ (this would be false for an arbitrary matrix). Here, the considered matrix is $(I - \gamma P_{\pi'})^{-1} = \sum_{t \geq 0} \gamma^t P_{\pi'}^t$, all its elements are obviously positive.

[13] Consider the Tetris game, the state being the current board, actions correspond to placing falling tetraminos, the reward is $+1$ for removing a line, the size of the state space is 2^{200} for a 10×20 board, which is too huge to be handled by a machine.

[14] We have considered finite state spaces so far. Yet, all involved sums are expectations. For example, the Bellman evaluation operator is $[T_\pi v](s) = r(s,\pi(s)) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,\pi(s)) v_\pi(s')$. With a continuous state space, it can be simply written as $[T_\pi v](s) = r(s,\pi(s)) + \gamma \int_{\mathcal{S}} v_\pi(s') P(ds'|s,\pi(s))$. More abstractly, it can be written as $[T_\pi v](s) = r(s,\pi(s)) + \gamma \mathbb{E}_{S' \sim P(.|s,\pi(s))}[v(S')]$, which encompasses both cases. Up to some technicalities, what we have presented so far remains true for the continuous case.

Algorithm 24 Policy iteration

Require: An initial $\pi_0 \in \mathcal{A}^{\mathcal{S}}$
1: $k = 0$
2: **repeat**
3: solve (policy evaluation)

$$v_k(s) = r(s, \pi_k(s)) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s, \pi_k(s)) v_k(s'), \quad \forall s \in \mathcal{S}.$$

4: Compute (policy improvement)

$$\pi_{k+1}(s) \in \operatorname*{argmax}_{a \in \mathcal{A}} \left(r(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a) v_k(s') \right).$$

5: $k \leftarrow k + 1$
6: **until** $v_{k+1} = v_k$
7: **return** the policy $\pi_{k+1} = \pi_*$

(see also the discussion on hypothesis spaces in Ch. 5),

$$v_\theta(s) = \theta^\top \phi(s) = \sum_{i=1}^{d} \theta_i \phi_i(s),$$

where θ are the parameters to be learnt and $\phi : \mathcal{S} \to \mathbb{R}^d$ is a predefined feature vector (the vector which components are the d user-defined basis functions $\phi_i(s)$);

- the model might be unknown and one has to rely on a dataset of the type

$$\mathcal{D} = \{(s_i, a_i, r_i, s'_i)_{1 \leq i \leq n}\}, \tag{24.11}$$

where action a_i is taken in state s_i (according to a given policy, to a random policy, or something else), the reward satisfies $r_i = r(s_i, a_i)$ and the next state is sampled according to the dynamics, $s'_i \sim P(.|s_i, a_i)$. There are multiple ways this dataset can be obtained. For example, it can be given beforehand (batch learning, the main case we consider in this section). It can also be gathered in an online fashion by the agent that tries to learn the optimal control (a case we address in Sec. 24.5). It can also be obtained in a somehow controlled manner if one has access to a simulator (which does not mean that the model is known). Here is an example of how this data can be used: first, notice that almost everything turns around Bellman operators, and that without the model, they cannot be computed. However, they can still be approximated from data. Assume that $a_i = \pi(s_i)$ for a policy of interest π. One can consider the *sampled* operator

$$[\hat{T}_\pi v](s_i) = r_i + \gamma v(s'_i).$$

This is an unbiased estimate of the evaluation operator, as $\mathbb{E}[[\hat{T}_\pi v](s_i)|s_i] = \mathbb{E}_{S' \sim P(.|s_i,a_i)}[r_i + \gamma v(S')] = [T_\pi v](s_i)$.

In this section, we will study approximate value and policy iteration algorithms that handle these problems. There exist also approximate linear programming approaches, extending the one presented in Sec. 24.3.1, but we will not discuss them here (as they are restrictive, in some sense). The interested reader can nevertheless refer (notably) to de Farias and Van Roy (2003, 2004) for more about this.

24.4.1 State-action value function

Before introducing *approximate dynamic programming (ADP)*, we need to extend the notion of value function. Indeed, we have seen that the notion of greedy policy is central in dynamic

programming (to compute the optimal policy from the optimal value function or to improve a policy in the policy iteration scheme). However, computing a greedy policy requires knowing the model, as

$$\pi \in \mathcal{G}(v) \Leftrightarrow \forall s \in \mathcal{S}, \quad \pi(s) \in \operatorname*{argmax}_{a \in \mathcal{A}} \left(r(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a) v(s') \right).$$

Assume that we can estimate the optimal value function from the data (thus, without knowing the model). This is a first step, but what we are interested in is a good control policy. Deducing a greedy policy from this estimated policy would not (or at least hardly) be possible from data only.

Another problem with value functions is that the Bellman optimality operator cannot be sampled as easily as the Bellman evaluation operator. We have seen just before that $[\hat{T}_\pi v](s_i) = r_i + \gamma v(s'_i)$ is an unbiased estimate of the evaluation operator. Recall the definition of the optimality operator (see Eq. (24.6)):

$$[T_* v](s) = \max_{a \in \mathcal{A}} \mathbb{E}_{S' \sim P(.|s,a)} [r(s,a) + \gamma v(S')].$$

To define a sampled operator $\hat{T}_*$, one would need to sample all actions (and related next states) for all states s_i in the dataset[15] (in order to compute the max). Write $s'_{i,a}$ a next state sampled according to $P(.|s_i, a)$. One could consider the following sampled operator

$$[\hat{T}_* v](s_i) = \max_{a \in \mathcal{A}} \left(r(s_i,a) + \gamma v(s'_{i,a}) \right).$$

Anyway, this estimator would be biased (as the expectation of a max is not the max of an expectation, $\mathbb{E}[[\hat{T}_* v](s_i)|s_i] \neq T_*(s_i)$).

There is a simple solution to alleviate these problems, namely the *state-action value function*, also called Q-function or quality function. For a given policy π, the state-action value function $Q_\pi(s,a) \in \mathbb{R}^{\mathcal{S} \times \mathcal{A}}$ associate to each state-action pair the expected discounted cumulative reward for starting in this state, taking this action (that might be different from the action advised by the policy) and following the policy π afterward:

$$\forall (s,a) \in \mathcal{S} \times \mathcal{A}, \quad Q_\pi(s,a) = \mathbb{E}[\sum_{t=0}^{\infty} \gamma^t r(S_t, A_t) | S_0 = s, A_0 = a, S_{t+1} \sim P(.|S_t, A_t), A_{t+1} = \pi(S_{t+1})].$$

Roughly speaking, this adds a degree of freedom to the definition of the value function by letting free the choice of the first action. Notably, it is clear from this definition that value and Q-functions are related as follows:

$$v_\pi(s) = Q_\pi(s, \pi(s)).$$

A *Bellman evaluation operator* can easily be defined (we used the same notation, which is slightly abusive as it operates on a different object) as $T_\pi : \mathbb{R}^{\mathcal{S} \times \mathcal{A}} \to \mathbb{R}^{\mathcal{S} \times \mathcal{A}}$ such that for $Q \in \mathbb{R}^{\mathcal{S} \times \mathcal{A}}$ we have componentwise:

$$\forall (s,a) \in \mathcal{S} \times \mathcal{A}, \quad [T_\pi Q](s,a) = r(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a) Q(s', \pi(s')).$$

The state-action value function Q_π is the unique fixed point of the operator T_π (this operator being a γ-contraction):

$$Q_\pi = T_\pi Q_\pi.$$

Therefore, computing the Q-function of a policy π also amounts to solving a linear system. The optimal value function $Q_* = Q_{\pi_*}$ also satisfies a fixed-point equation. Let define the *Bellman optimality operator* $T_* : \mathbb{R}^{\mathcal{S} \times \mathcal{A}} \to \mathbb{R}^{\mathcal{S} \times \mathcal{A}}$ such that for $Q \in \mathbb{R}^{\mathcal{S} \times \mathcal{A}}$ we have componentwise:

$$\forall (s,a) \in \mathcal{S} \times \mathcal{A}, \quad [T_* Q](s,a) = r(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a) \max_{a' \in \mathcal{A}} Q(s', a').$$

[15] To do so, we should have access to a simulator.

The optimal state-action value function Q_* is the unique fixed point of the operator T_* (this operator being a γ-contraction):

$$Q_* = T_* Q_*.$$

Notice that the optimal value and quality functions are related as follows:

$$\forall s \in \mathcal{S}, \quad v_*(s) = \max_{a \in \mathcal{A}} Q_*(s,a).$$

A first advantage of the quality function is that it allows computing a *greedy policy* without knowing the model. Indeed, for a policy improvement step (to compute a greedy policy respectively to v_π, recalling that it satisfies $v_\pi(s) = Q_\pi(s, \pi(s))$), we have

$$\begin{aligned} \pi' \in \mathcal{G}(v_\pi) &\Leftrightarrow \forall s \in \mathcal{S}, \quad \pi(s) \in \operatorname*{argmax}_{a \in \mathcal{A}} \left(r(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a) v_\pi(s') \right) \\ &\Leftrightarrow \forall s \in \mathcal{S}, \quad \pi'(s) \in \operatorname*{argmax}_{a \in \mathcal{A}} Q_\pi(s,a). \end{aligned}$$

If one is able to compute the optimal quality function , it is possible to compute the optimal policy as being greedy respectively to it:

$$\pi_*(s) \in \operatorname*{argmax}_{a \in \mathcal{A}} Q_*(s,a).$$

In all cases, we define a policy π as being greedy respectively to a function $Q \in \mathbb{R}^{\mathcal{S} \times \mathcal{A}}$ (which is not necessarily the state-action value function of a policy) as

$$\forall Q \in \mathbb{R}^{\mathcal{S} \times \mathcal{A}}, \quad \pi \in \mathcal{G}(Q) \Leftrightarrow \forall s \in \mathcal{S}, \quad \pi(s) \in \operatorname*{argmax}_{a \in \mathcal{A}} Q(s,a).$$

When working with data (with the dataset given in Eq. (24.11)), both operators can be sampled. The Bellman evaluation operator can be sampled as

$$[\hat{T}_\pi Q](s_i, a_i) = r_i + \gamma Q(s'_i, \pi(s'_i)).$$

The Bellman optimality operator can also be sampled:

$$[\hat{T}_* Q](s_i, a_i) = r_i + \gamma \max_{a' \in \mathcal{A}} Q(s'_i, a').$$

Now, both sampled operators are unbiased (contrary to sampled optimality operator applying on value functions).

The policy and value iteration algorithms can be easily rewritten with state-action value functions replacing value functions. For all the reasons given so far, it is customary to work with quality functions when the model is unknown. We have seen that when the state space is too large, value functions should be searched for in some hypothesis space. For example, with a linear parameterization, the quality functions would be of the form $Q_\theta(s,a) = \theta^\top \phi(s,a)$. Yet, the states are usually continuous while the actions are discrete (a less frequent case in supervised learning). A standard approach consists in defining a feature vector $\phi(s)$ for the state space, and to extend it to the state-action space as follows (δ being the Dirac function):

$$\phi(s,a) = \begin{bmatrix} \delta_{a=a_1} \phi(s)^\top & \dots & \delta_{a=a_{|\mathcal{A}|}} \phi(s)^\top \end{bmatrix}^\top.$$

Notice that this is reminiscent of the concept of score function for cost-sensitive multiclass clasification seen in Sec. 5.2.

24.4.2 Approximate value iteration

In this section, we work with a dataset as in Eq. (24.11), that is

$$\mathcal{D} = \{(s_i, a_i, r_i, s'_i)_{1 \leq i \leq n}\},$$

and the aim is to estimate from this set of transitions the optimal Q-function Q_* (from which we can estimate an optimal policy by being greedy). The Bellman optimality operator applying on $\mathbb{R}^{\mathcal{S}\times\mathcal{A}}$ is a γ-contraction (the proof is very similar to the case of value functions). Therefore, thanks to the Banach theorem, the iteration

$$Q_{k+1} = T_* Q_k$$

will converge to Q_*. Yet, there are two problems:

- the operator T_* cannot be applied to Q_k, the model being unknown;
- as we are working with a too large state space, the Q-functions should belong to some hypothesis space $\mathcal{H}$, and there is not reason that $T_* Q_k \in \mathcal{H}$ holds.

The first problem can be avoided by using the sampled operator presented before instead, the second one indeed corresponds to a regression problem, as shown below.

Now, we construct an approximate variation of value iteration applied to state-action value functions. Assume that we adopt a linear parametrization for the Q-function, that is we consider the following hypothesis space

$$\mathcal{H} = \{Q_\theta(s,a) = \theta^\top \phi(s,a), \theta \in \mathbb{R}^d\},$$

with $\phi(s,a)$ a predefined feature vector. Let θ_0 be some initial parameter vector and let $Q_0 = Q_{\theta_0}$ be the associated quality function. At iteration k we have $Q_k = Q_{\theta_k}$. We can sample the optimality operator for state-action couples available in the dataset $\mathcal{D}$:

$$\forall 1 \leq i \leq n, \quad [\hat{T}_* Q_k](s_i,a_i) = r_i + \gamma \max_{a' \in \mathcal{A}} Q_k(s_i', a').$$

So, we have n target values for the next function Q_{k+1} (corresponding to inputs (s_i,a_i)), and this function must belong to $\mathcal{H}$. Finding the function Q_{k+1} is therefore a regression problem that can be solved by minimizing the risk based on the ℓ_2-loss, for example. Therefore, Q_{k+1} can be computed as follows:

$$Q_{k+1} \in \operatorname*{argmin}_{Q_\theta \in \mathcal{H}} \frac{1}{n} \sum_{i=1}^n \Big(Q_\theta(s_i,a_i) - [\hat{T}_* Q_k](s_i,a_i) \Big)^2.$$

Given the chosen hypothesis space, this is simply a linear least-squares problem with inputs (s_i,a_i) and outputs $r_i + \gamma \max_{a' \in \mathcal{A}} Q_k(s_i',a')$, and simple calculus[16] gives the solution:

$$Q_{k+1} = Q_{\theta_{k+1}} \text{ with } \theta_{k+1} = \left(\sum_{i=1}^n \phi(s_i,a_i)\phi(s_i,a_i)^\top \right)^{-1} \left(\sum_{i=1}^n \phi(s_i,a_i) \left(r_i + \gamma \max_{a' \in \mathcal{A}} Q_{\theta_k}(s_i',a') \right) \right).$$

Alternatively, we can see this as projecting the sampled operator applied to the previous function onto the hypothesis space, which can written more compactly as $Q_{k+1} = \Pi \hat{T}_* Q_k$, writing Π the projection.

For the regression step, we have considered a quadratic risk with a linear parametrization (that is, a linear least-squares), but other regression schemes can be envisioned. Write abstractly $\mathfrak{A}$ the operator that gives a function from observations (such as the result of minimizing a risk, or a random forest, or something else), *approximate value iteration* can be written generically as (for some initialization Q_0)

$$Q_{k+1} = \mathfrak{A} \hat{T}_* Q_k.$$

Yet, if the Bellman operator T_* is a contraction, there is not reason for the composed operator $\mathfrak{A}\hat{T}_*$ to be a contraction. Indeed, in the example developed before (with the linear least-squares), the operator $\Pi\hat{T}_*$ is not a contraction, and the iterations may diverges. A sufficient condition for the operator $\mathfrak{A}\hat{T}_*$ to be a contraction is to use *averagers* as function approximators in the regression step. We do not explain here what an averager is, but random forests and ensemble of extremely randomized trees (see Ch. 21) belong to this category. Using this kind of function approximator in the regression step therefore ensure convergence.

[16] Generically, the problem is to solve $\min_\theta \frac{1}{n}\sum_{i=1}^n (y_i - \theta^\top x_i)^2$. Computing the gradient (resp. to θ) and setting it to zero provides the solution $\sum x_i x_i^\top \theta = \sum x_i y_i$.

Algorithm 25 Approximate value iteration

Require: A dataset $\mathcal{D} = \{(s_i, a_i, r_i, s'_i)_{1 \leq i \leq n}\}$, the number K of iterations, a function approximator, an initial state-action value function Q_0.

1: **for** $k = 0$ **to** K **do**

2: Apply the sampled Bellman operator to function Q_k:

$$[\hat{T}_* Q_k](s_i, a_i) = r_i + \gamma \max_{a' \in \mathcal{A}} Q_k(s'_i, a').$$

3: Solve the regression problem with inputs (s_i, a_i) and outputs $[\hat{T}_* Q_k](s_i, a_i)$ to get the Q-function Q_{k+1}

4: **end for**

5: **return** The greedy policy $\pi_{K+1} \in \mathcal{G}(Q_{K+1})$:

$$\forall s \in \mathcal{S}, \quad \pi_{K+1} \in \operatorname*{argmax}_{a \in \mathcal{A}} Q_{K+1}(s, a).$$

The generic approximate value iteration is provided in Alg. 25. An important thing to notice is that this algorithm reduces the learning of an optimal control to a sequence of supervised learning problems. For the definition of an averager and a discussion on necessary conditions for $\mathfrak{A}\hat{T}_*$ to be a contraction, see Gordon (1995). When the function approximator is an ensemble of extremely randomized trees, the algorithm is known as fitted-Q (Ernst et al., 2005) and is quite efficient empirically. Approximate value iteration has been experimented with other function approximators, such as neural networks (Riedmiller, 2005) or Nadaraya-Watson estimators (Ormoneit and Sen, 2002).

So far, we have assumed that the dataset is given beforehand. The quality of this dataset is very important for good empirical results. For example, if states s_i in $\mathcal{D}$ are sampled in a too small part of the state space, no algorithm will be able to recover a good controller. If one has access to a simulator (or to the real system), it is possible to choose how data are sampled (how states s_i are sampled, according to what policy actions a_i are sampled, the next states s'_i being imposed by the dynamics). In this case, one can wonder what is the best way to sample states. A sensible approach would consist in following the current policy $\pi_{k+1} \in \mathcal{G}(Q_{k+1})$ slightly randomized (this is linked to what is known as the exploration-exploitation dilemma, to be discussed in Sec. 24.5). Choosing the right distribution is a difficult problem, and we will not discuss it much further here. However, there is an important remark: in supervised learning, the distribution is fixed beforehand (given by the problem at hand), while in approximate dynamic programming (that is, in reinforcement learning) only the dynamics is fixed, which can involve many different distributions on transitions. Therefore, things are much more difficult to analyse in this setting.

We can also have a word about model evaluation. An important question is: how good is the policy π_{K+1} returned by approximate value iteration? When estimating a function in supervised learning, its quality can be assessed by using cross-validation, for example. In reinforcement learning, this is much more difficult, cross-validation cannot be applied. The best way to assess the quality of the policy π_{K+1} is to apply it to the control problem (and if it is a real system, and not a simulated one, it can be dangerous). There are few works on model evaluation for reinforcement learning (Farahmand and Szepesvári, 2011; Thomas et al., 2015), but notice that there is no answer as easy as in supervised learning.

24.4.3 Approximate policy iteration

The policy iteration algorithm can also be approximated. Recall the steps involved in an iteration of policy iteration:

1. policy evaluation: solve the fixed-point equation $Q_{\pi_k} = T_{\pi_k} Q_{\pi_k}$;
2. policy improvement: compute the greedy policy $\pi_{k+1} = \mathcal{G}(Q_{\pi_k})$.

Given any function $Q \in \mathbb{R}^{\mathcal{S}\times\mathcal{A}}$, computing an associated greedy policy is easy (that is partly why the state-action value function has been introduced). Therefore, the step to be approximated is the policy evaluation step. In other words, the problem consists in estimating the quality function of a given policy, from data. An iteration of *approximate policy iteration* can be (informally) summarized as follows:

1. approximate policy evaluation: find a function $Q_k \in \mathcal{H}$ such that $Q_k \approx T_{\pi_k} Q_k$;
2. policy improvement: compute the greedy policy $\pi_{k+1} = \mathcal{G}(Q_{\pi_k})$.

The whole procedure is presented in Alg. 26.

Algorithm 26 Approximate policy iteration

Require: An initial $\pi_0 \in \mathcal{A}^{\mathcal{S}}$ (possibly an initial Q_0 and $\pi_0 \in \mathcal{G}(Q_0)$), number of iterations K

1: **for** $k = 0$ **to** K **do**

2: approximate policy evaluation:

$$\text{find } Q_k \in \mathcal{H} \text{ such that } Q_k \approx T_{\pi_k} Q_k.$$

3: policy improvement:

$$\pi_{k+1} \in \mathcal{G}(Q_k).$$

4: **end for**

5: **return** the policy π_{K+1}

So, the core question is: how to estimate the quality function of a given policy from a given dataset? As before, the model is unknown and the Bellman operator can only be sampled. Moreover, the state space being too large, we're looking for a Q-function belonging to some predefined hypothesis space. Here, we will assume a linear parametrization, that is $\mathcal{H} = \{Q_\theta(s,a) = \theta^\top \phi(s,a), \theta \in \mathbb{R}^d\}$. Let π be any policy, we discuss now how to find an approximate fixed point of T_π, that is a function $Q_\theta \in \mathcal{H}$ such that $Q_\theta \approx T_\pi Q_\theta$.

Monte Carlo Rollouts

We know that Q_π is the fixed point of T_π. Therefore, solving the approximate fixed point equation amounts to approximating Q_π. Assume that this function is known for the state-action couples (s_i, a_i) in the dataset $\mathcal{D}$. Then, this is simply a regression problem. For example, with an ℓ_2-loss, the problem to be solved is the following linear least-squares:

$$\min_{\theta\in\mathbb{R}^d} \frac{1}{n} \sum_{i=1}^{n} \left(Q_\pi(s_i,a_i) - Q_\theta(s_i,a_i)\right)^2.$$

Unfortunately, the Q-function Q_π is obviously unknown (it is what we would like to estimate). However, if a simulator is available, it can be estimated for any given state-action couple (s_i, a_i). To do so, the idea is to sample a full trajectory starting in s_i where action a_i is chosen first, all subsequent states being sampled according to the system dynamics and all subsequent actions being chosen according to the policy π. Write q_i the associated discounted cumulative reward (the sum of discounted rewards gathered along the sampled trajectory), it is an unbiased estimate of the state-action value function: $\mathbb{E}[q_i|s_i,a_i] = Q_\pi(s_i,a_i)$. This is called a *Monte Carlo rollout*. This fits the regression setting and one can solve

$$\min_{\theta\in\mathbb{R}^d} \frac{1}{n} \sum_{i=1}^{n} \left(q_i - Q_\theta(s_i,a_i)\right)^2.$$

to generalize the simulated state-action values. The solution is here the classical linear-least squares estimator, the vector parameter minimizing this empirical risk being

$$\theta_n = \left(\sum_{i=1}^{n} \phi(s_i,a_i)\phi(s_i,a_i)^\top\right)^{-1} \sum_{i=1}^{n} \phi(s_i,a_i) q_i.$$

Notice that other losses and other function approximators could be used (as it is a standard regression problem). The disadvantage of this approach is that it requires a simulator (which is not always available, and which can be costly to use) and that the rollouts can be quite noisy (due to the variance induced by the stochasticity of the system). Also, if it formally requires to sample infinite trajectories, practically finite trajectories are sampled[17].

Residual approach

As we are looking for an approximate fixe point of T_π, a natural idea consists in minimizing $\|Q_\theta - T_\pi Q_\theta\|$ for some norm. If we can set this quantity to zero, then we have found the fixed point (and the the state-action value function Q_π). This is called a *residual approach* as we try to minimize the residual between Q_θ and its image $T_\pi Q_\theta$ under the Bellman evaluation operator. We work with data and the operator can only be sampled, so a natural optimization problem to consider is

$$\min_{\theta\in\mathbb{R}^d} \frac{1}{n}\sum_{i=1}^n \left([\hat{T}_\pi Q_\theta](s_i,a_i) - Q_\theta(s_i,a_i)\right)^2 = \min_{\theta\in\mathbb{R}^d} \frac{1}{n}\sum_{i=1}^n \left(r_i + \gamma Q_\theta(s_i',\pi(s_i')) - Q_\theta(s_i,a_i)\right)^2.$$

With a linear parametrization $Q_\theta(s,a) = \theta^\top \phi(s,a)$, this can be solved analytically by zeroing the gradient, the minimizer is then given by

$$\theta_n = \left(\frac{1}{n}\sum_{i=1}^n \Delta\phi_i \Delta\phi_i^\top\right)^{-1} \left(\frac{1}{n}\sum_{i=1}^n \Delta\phi_i r_i\right) \text{ with } \Delta\phi_i = \phi(s_i,a_i) - \gamma\phi(s_i',\pi(s_i')).$$

The problem with this approach is that it leads to minimizing a biased surrogates to the residual. Indeed, if $\hat{T}_\pi$ is an unbiased estimator of the Bellman operator, it is no longer true with its square (the square of the mean is not the mean of the square):

$$\begin{aligned}\mathbb{E}[([\hat{T}_\pi Q_\theta](s_i,a_i) - Q_\theta(s_i,a_i))^2 | s_i,a_i] &= ([T_\pi Q_\theta](s_i,a_i) - Q_\theta(s_i,a_i))^2 + \text{var}([\hat{T}_\pi Q_\theta](s_i,a_i)|s_i,a_i) \\ &\neq ([T_\pi Q_\theta](s_i,a_i) - Q_\theta(s_i,a_i))^2.\end{aligned}$$

If the dynamics is deterministic, this approach is fine (the variance term is null). However, with stochastic transitions the estimate will be biased. The variance term acts as a regularizing factor, which is good in general, but not here, as it cannot be controlled (there is no factor for trading off the risk and the regularization). For more about this, see Antos et al. (2008).

Least-Squares Temporal Differences

Asymptotically, a linear least-squares projects the function of interest onto the hypothesis space. Writing Π this projection, the idea here is to find a fixed point of the composed operator ΠT_π, that is to solve $Q_\theta = \Pi T_\pi Q_\theta$. This approach is known as LSTD (Least-Squares Temporal Differences) and has been originally proposed from a different perspective (more precisely, an error-in-variable model) by Bradtke and Barto (1996).

Consider Fig. 24.2. On the left side, we show what the Monte Carlo does asymptotically: it projects the Q-function of interest onto the hypothesis space $\mathcal{H}$. We have seen that this requires a simulator. Consider now the figure on the right. Let Q_θ be a function of $\mathcal{H}$. One can apply the Bellman evaluation operator to this function, to get $T_\pi Q_\theta$. Notice that there is not reason for $T_\pi Q_\theta$ to belong to the hypothesis space $\mathcal{H}$. The residual approach tries to directly minimize the distance between these functions (but we have seen that it is biased). Now, take $T_\pi Q_\theta$ and project it back onto the hypothesis space, which gives $\Pi T_\pi Q_\theta$. LSTD search for the parameter vector that minimizes the distance between Q_θ and $\Pi T_\pi Q_\theta$, the dashed line in the figure (and one can show that this distance is zero).

[17] A first solution is to truncate the trajectories to an horizon H, the error will be bounded by $\frac{\gamma^H \|r\|_\infty}{1-\gamma}$. Another solution is to sample the horizon according to a geometric law of parameter $\frac{1}{1-\gamma}$ and to take q_i as the *undiscounted* cumulative reward. One can check that it is an unbiased estimate, but the random horizon can be arbitrary long (a geometric law is unbounded).

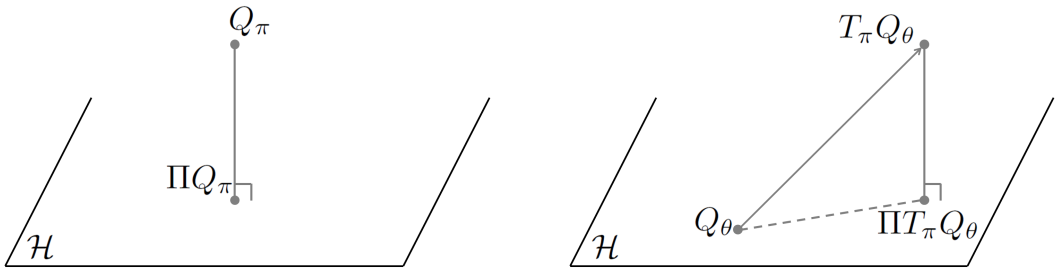


Figure 24.2: Illustration for the Monte Carlo Rollouts (left) and LSTD (right). See the text for details.

As usual, we work with data. LSTD can be expressed as solving the following nested optimization problem:

$$\begin{cases} w_\theta = \operatorname{argmin}_{w\in\mathbb{R}^d} \frac{1}{n}\sum_{i=1}^n (r_i + \gamma Q_\theta(s_i', \pi(s_i')) - Q_w(s_i, a_i))^2 \\ \theta_n = \operatorname{argmin}_{\theta\in\mathbb{R}^d} \frac{1}{n}\sum_{i=1}^n (Q_\theta(s_i, a_i) - Q_{w_\theta}(s_i, a_i))^2 \end{cases}.$$

The first equation correspond to the projection of $\hat{T}_\pi Q_\theta$ onto $\mathcal{H}$ and the second equation to the minimization of the distance between Q_θ and the projection of $\hat{T}_\pi Q_\theta$. This is a nested optimization problem as both solutions are interleaved (w_θ depends en θ and vice-versa). Given the chosen linear parametrization, this can be solved analytically. The first equation is a simple linear least-squares problem in w, which solution is given by

$$w_\theta = \left(\sum_{i=1}^n \phi(s_i, a_i)\phi(s_i, a_i)^\top\right)^{-1} \sum_{i=1}^n \phi(s_i, a_i)(r_i + \gamma\theta^\top \phi(s_i', \pi(s_i'))).$$

The second equation is minimized with $\theta = w_\theta$ (and the distance is zero). Therefore, the solution is

$$\begin{aligned} \theta_n = w_{\theta_n} \Leftrightarrow \theta_n &= \left(\sum_{i=1}^n \phi(s_i, a_i)\phi(s_i, a_i)^\top\right)^{-1} \sum_{i=1}^n \phi(s_i, a_i)(r_i + \gamma\theta_n^\top \phi(s_i', \pi(s_i'))) \\ \Leftrightarrow \theta_n &= \left(\sum_{i=1}^n \phi(s_i, a_i)\left(\phi(s_i, a_i) - \gamma\phi(s_i', \pi(s_i'))\right)^\top\right)^{-1} \sum_{i=1}^n \phi(s_i, a_i) r_i. \end{aligned}$$

When LSTD is the policy evaluation step of approximate policy iteration, the resulting algorithm is called LSPI (Lagoudakis and Parr, 2003) for least-squares policy iteration, and is summarized in Alg. 27.

We have see that a central question here is how to approximate the quality function from data. We have shown the main methods, but many other exist. For an overview of the subject, the interested reader can refer to Geist and Pietquin (2013); Geist and Scherrer (2014) (some other will be briefly discussed in Sec. 24.5). For a discussion about the link between the residual approach and LSTD, see Scherrer (2010).

Approximating the policy

So far, we have tried to estimate a state-action value function, the policy being deduced from it. However, in some cases, it might easer to approximate directly the policy (for example, because it has a simpler form). In an approximate policy iteration scheme, this would mean that we approximate the policy improvement step, instead of the policy evaluation step.

At iteration k, assume that $Q_{\pi_k}(s_i, a)$ is known for all $1 \leq i \leq n$ and $a \in \mathcal{A}$. Here, let $\mathcal{F} \subset \mathcal{A}^\mathcal{S}$ an hypothesis space of policies. Consider the following optimization problem:

$$\pi_{k+1} \in \operatorname*{argmin}_{\pi\in\mathcal{F}} \frac{1}{n}\sum_{i=1}^n \left(\max_{a\in\mathcal{A}} Q_{\pi_k}(s_i, a) - Q_{\pi_k}(s_i, \pi(s_i))\right). \tag{24.12}$$

Algorithm 27 Least-squares policy iteration

Require: An initial $\pi_0 \in \mathcal{A}^{\mathcal{S}}$ (possibly an initial Q_0 and $\pi_0 \in \mathcal{G}(Q_0)$), number of iterations K

1: **for** $k = 0$ **to** K **do**

2: approximate policy evaluation:

$$\theta_k = \left(\sum_{i=1}^{n} \phi(s_i, a_i) \left(\phi(s_i, a_i) - \gamma \phi(s_i', \pi_k(s_i')) \right)^\top \right)^{-1} \sum_{i=1}^{n} \phi(s_i, a_i) r_i.$$

3: policy improvement:

$$\pi_{k+1} \in \mathcal{G}(Q_{\theta_k}).$$

4: **end for**

5: **return** the policy π_{K+1}

This is a cost-sensitive multiclass classification problem (see also Sec. 5.2). Notice that a policy can be seen as a decision rule (as it associates a label—an action—to an input—a state). Take a policy π, it will suffer a cost of $\max_{a \in \mathcal{A}} Q_{\pi_k}(s_i, a) - Q_{\pi_k}(s_i, \pi(s_i))$ for choosing the action $\pi(s_i)$ in state s_i instead of the greedy action $\operatorname{argmax}_{a \in \mathcal{A}} Q_{\pi_k}(s_i, a)$. So, solving the above optimization problem, with an infinite amount of data and a rich enough hypothesis space, gives the greedy policy $\mathcal{G}(\pi_{k+1})$. As we work with a finite amount of data and as the hypothesis space may not contain the greedy policy, in practice we obtain an approximate greedy policy.

Obviously, the state-action value function is unknown, while it is required to express problem (24.12). Yet, we only need to know (possibly approximately) the state action value function for the state-action couples $\{(s_i, a)_{1 \leq i \leq n, a \in \mathcal{A}}\}$. We can estimate these values by performing Monte Carlo rollouts. Notice that we only need to estimate pointwise the function, we do not need to generalize it to the whole state-action space (generalization is done by the policy). The interesting aspect here is that we have reduced the optimal control problem to a sequence of classification problems.

This approach is called DPI (Lazaric et al., 2010) for direct policy iteration. Notice that all the approximate dynamic algorithms we have presented so far are special cases of the generic *approximate modified policy iteration* approach. The interested reader can refer to Scherrer et al. (2015) for more about this.

24.5 Online learning

In Sec. 24.4, we have considered that a set of data was provided, and we tried to learn a good controller from it (with possible call to a simulator from time to time). Another setting of interest is online learning. Here, the agent is interacting with the system, and tries to learn the optimal control while interacting with the system. This call for learning in an online fashion (while methods considered so far are batch algorithms). Here, we will focus on learning state-action value functions (the policy being derived from it). As the agent can choose the action that will be applied to the system, he can somehow control the kind of data he will observe. At each time step, he will have to choose between two alternatives: he can either choose an action that he think to be optimal, according to the currently estimated quality function, or he can choose a supposedly suboptimal action, but that can improve its knowledge of the world (its estimate of the Q-function) and therefore lead to highest cumulative rewards in the future. This is called the *exploration-exploitation dilemma*.

24.5.1 SARSA and Q-learning

We start by presenting methods for estimating quality functions in an online manner.

SARSA

Let π any policy, we present now an algorithm for estimating the Q-function of this policy in an online fashion. Let $\mathcal{H} = \{Q_\theta(s,a) = \theta^\top \phi(s,a), \theta \in \mathbb{R}^d\}$ be an hypothesis space of linearly parameterized functions. Assume that Q_π is known, we would like to solve

$$\min_{\theta \in \mathbb{R}^d} \frac{1}{n} \sum_{i=1}^{n} \left(Q_\pi(s_i,a_i) - Q_\theta(s_i,a_i)\right)^2 .$$

If we minimize this risk using a stochastic gradient descent, we get classically the following update rule,

$$\begin{aligned}\theta_{i+1} &= \theta_i - \frac{\alpha_i}{2} \nabla \left(Q_\pi(s_i,a_i) - Q_\theta(s_i,a_i)\right)^2 \\ &= \theta_i + \alpha_i \phi(s_i,a_i) \left(Q_\pi(s_i,a_i) - \theta_i^\top \phi(s_i,a_i)\right),\end{aligned}$$

where α_i is a learning rate. This is a standard Widrow-Hoff update. At time-step i, the state-action couple (s_i,a_i) is observed, and the parameter vector is updated according to the gain $\alpha_i \phi(s_i,a_i)$ and to the prediction error $Q_\pi(s_i,a_i) - \theta_i^\top \phi(s_i,a_i)$.

Unfortunately, and as usual, the state-action value function is unknown (as it is what we would like to estimate). The idea is to apply a bootstrapping principle: the unobserved value $Q_\pi(s_i,a_i)$ is replaced by an estimate computed by applying the sampled Bellman evaluation operator to the current estimate $Q_{\theta_i}(s_i,a_i)$, that is $[\hat{T}_\pi Q_{\theta_i}](s_i,a_i) = r_i + \gamma Q_{\theta_i}(s_{i+1}, \pi(s_{i+1}))$, where $s_{i+1} \sim P(.|s_i,a_i)$ (s_{i+1} is obtained by applying action a_i to the system). To sum up, let $(s_i,a_i,r_i,s_{i+1},a_{i+1})$ be the current transition, with $r_i = r(s_i,a_i)$, $s_{i+1} \sim P(.|s_i,a_i)$ and $a_{i+1} = \pi(s_{i+1})$, the update rule is

$$\begin{aligned}\theta_{i+1} &= \theta_i + \alpha_i \phi(s_i,a_i) \left(r_i + \gamma Q_{\theta_i}(s_{i+1}, \pi(s_{i+1})) - Q_{\theta_i}(s_i,a_i)\right) \\ &= \theta_i + \alpha_i \phi(s_i,a_i) \left(r_i + \gamma \theta_i^\top \phi(s_{i+1},a_{i+1}) - \theta_i^\top \phi(s_i,a_i)\right).\end{aligned}$$

This is called a temporal difference algorithm, due to the prediction error $r_i + \gamma Q_{\theta_i}(s_{i+1}, \pi(s_{i+1})) - Q_{\theta_i}(s_i,a_i)$ being a temporal difference.

Algorithm 28 SARSA

Require: An initial parameter vector θ_0, the initial state s_0, an initial action a_0, the learning rates $(\alpha_i)_{i \geq 0}$

1: i = 0
2: **while true do**
3: Apply action a_i in state s_i
4: Get the reward r_i and observe the new state s_{i+1}
5: **Choose the action a_{i+1} to be applied in state s_{i+1}**
6: Update the parameter vector of the Q-function according to the transition $(s_i,a_i,r_i,s_{i+1},a_{i+1})$;

$$\theta_{i+1} = \theta_i + \alpha_i \phi(s_i,a_i) \left(r_i + \gamma \theta_i^\top \phi(s_{i+1},a_{i+1}) - \theta_i^\top \phi(s_i,a_i)\right)$$

7: $i \leftarrow i+1$
8: **end while**

The resulting algorithm is called SARSA (due to the transition $(s_i,a_i,r_i,s_{i+1},a_{i+1})$) and is summarized in Alg. 28. Notice that we remain vague about how action a_{i+1} is chosen (see line 5 in Alg. 28). We have said just before that action a_{i+1} is chosen according to π, the policy to be evaluated. If we consider a policy evaluation problem, that's fine. However, we would like to learn the optimal control. A possibility would be to take a_{i+1} as the greedy action respectively to the current estimate Q_{θ_i}. This would correspond to an optimistic[18] approximate policy iteration

[18]The optimism lies in the fact that the evaluation step occurs for only one transition before taking the greedy step.

scheme. Yet, this would be too conservative (if the Q-function is badly estimated, the agent can get stuck) and it should be balanced with some exploration. Before expending on this, we present an alternative algorithm.

Q-learning

We can do the same job as for SARSA with the optimal Q-function Q_* directly, instead of Q_π. The update rule (assuming Q_* known) is:

$$\theta_{i+1} = \theta_i + \alpha_i \phi(s_i, a_i) \left(Q_*(s_i, a_i) - \theta_i^\top \phi(s_i, a_i) \right).$$

The function Q_* is unknown, but it can be bootstrapped by replacing it by the estimate obtained by applying the sampled Bellman *optimality* operator to the current estimate, $[\hat{T}_* Q_{\theta_i}](s_i, a_i) = r_i + \gamma \max_{a \in \mathcal{A}} \hat{Q}_{\theta_i}(s_i', a)$, giving the following update rule:

$$\begin{aligned}\theta_{i+1} &= \theta_i + \alpha_i \phi(s_i, a_i) \left(r_i + \gamma \max_{a \in \mathcal{A}} Q_{\theta_i}(s_{i+1}, a) - Q_{\theta_i}(s_i, a_i) \right) \\ &= \theta_i + \alpha_i \phi(s_i, a_i) \left(r_i + \gamma \max_{a \in \mathcal{A}} (\theta_i^\top \phi(s_{i+1}, a)) - \theta_i^\top \phi(s_i, a_i) \right).\end{aligned}$$

Notice that whatever the behavior policy (the way actions a_i are chosen), we will estimate directly the optimal Q-function (given some assumptions, notably the behavior policy should visit often enough all state-action pairs). This is called an *off-policy* algorithm (the estimated policy—here the optimal one—is different from the behavior policy). This is different from SARSA (used for policy evaluation), where the followed policy and the evaluated policy are the same. This is called *on-policy* learning.

Algorithm 29 Q-learning

Require: An initial parameter vector θ_0, the initial state s_0, the learning rates $(\alpha_i)_{i \geq 0}$

1: i = 0
2: **while true do**
3: **Choose the action a_i to be applied in state s_i**
4: Apply action a_i in state s_i
5: Get the reward r_i and observe the new state s_{i+1}
6: Update the parameter vector of the Q-function according to the transition (s_i, a_i, r_i, s_{i+1});

$$\theta_{i+1} = \theta_i + \alpha_i \phi(s_i, a_i) \left(r_i + \gamma \max_{a \in \mathcal{A}} (\theta_i^\top \phi(s_{i+1}, a)) - \theta_i^\top \phi(s_i, a_i) \right)$$

7: $i \leftarrow i + 1$
8: **end while**

The resulting algorithm is called Q-learning and is summarized in Alg. 29. Again, we remain vague about how action a_i is chosen (see line 3 in Alg. 29). If the Q-function is perfectly estimated, the wisest choice would consist in taking the greedy action. However, if some action has never been tried (or not enough, that is the Q-function estimation has not converged), one cannot know if it is not indeed a better action than the greedy one. Therefore, exploitation (taking the greedy action) should be combined with exploration (taking another action).

Notice that there exist many other online learning algorithms. For example, LSTD can be made online by using Sherman-Morrison (much like how recursive linear least-squares are derived from linear least-squares). For more about this, see again Geist and Pietquin (2013); Geist and Scherrer (2014).

24.5.2 The exploration-exploitation dilemma

Therefore, for both SARSA and Q-learning, there is a dilemma between exploitation (taking the greedy action) and exploration (taking a non-greedy action, leading possibly to higher discounted

cumulative rewards). We present two simple solutions, that is policies balancing exploration and exploitation, that can be used in line 5 of Alg. 28 and line 3 of Alg. 29.

Let ϵ be in $(0, 1)$. An *ϵ-greedy policy* chooses the greedy action with probability $1 - \epsilon$, and a random action with probability ϵ:

$$\pi_\epsilon(s) = \begin{cases} \text{argmax}_{a \in \mathcal{A}} Q_\theta(s, a) & \text{with probability } 1 - \epsilon \\ \text{a random action} & \text{with probability } \epsilon \end{cases}.$$

Assume that ϵ is small. Most of time, the agent will act greedily respectively to the currently estimated quality function. However, from time to time, it will try a different action, to see if it cannot gather higher values. Practically, it is customary to set a high value of ϵ at the beginning of learning, such as favoring exploration, and then to decrease ϵ as the estimation of the Q-function improves, so as to act more and more greedily.

A different kind policy is the *softmax policy*. Let Q_θ be the currently estimated quality function and let $\tau > 0$ be a temperature parameter, the softmax policy is a stochastic policy defined as

$$\pi_\tau(a|s) = \frac{e^{\frac{1}{\tau} Q_\theta(s,a)}}{\sum_{a' \in \mathcal{A}} e^{\frac{1}{\tau} Q_\theta(s,a')}}.$$

It is easy to check that this indeed defines a conditional probability. It is clear from this definition that action with high Q-values will be sampled more often. The parameter τ allows going from a purely uniform random policy ($\tau \to \infty$) to the greedy policy ($\tau \to 0$).

These policies make sense for balancing exploration and exploitation. However, one can wonder how effective they are (empirically, which can be tried, but also theoretically). Moreover, one can wonder what is the best exploration strategy. This is a very difficult question. Indeed, consider the bandit problem studied in Ch. 23. A bandit is indeed an MDP with a single state (there is no transition kernel and $\gamma = 0$). We have studied the UCB strategy, that adresse the exploration-exploitation dilemma. Things are much more complicated in the general MDP setting. For a few possible strategies, the interested read can refer to Bayesian reinforcement learning (Poupart et al., 2006; Vlassis et al., 2012) or R-max (Brafman and Tennenholtz, 2003), among many others.

24.6 Policy search and actor-critic methods

So far, we have only considered discrete actions. Indeed, all approaches studied before involve computing some max or argmax over actions, which becomes a possibly difficult optimization problem when actions are continuous. In this section, we discuss an alternative way to estimate an optimal policy, by searching directly in the policy space. The idea is to parameterize the policy and to look for the set of parameters optimizing some objective function.

To do so, we need to work with stochastic policies, such that no action has a null probability (for a reason that will be clear later). A stochastic policy $\pi \in \Delta_{\mathcal{A}}^{\mathcal{S}}$ associates to each state s a conditional probability over actions $\pi(.|s)$. All the things we have defined for deterministic policies extend naturally to stochastic policies. For example, the Bellman evaluation equation is

$$v_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \left(r(s, a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s, a) v_\pi(s) \right).$$

The value and quality function are linked as follows:

$$v_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) Q_\pi(s, a).$$

Therefore, we can notably write

$$Q_\pi(s, a) = r(s, a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s, a) v_\pi(s),$$

as in the deterministic case.

Here, we look for parameterized policies belonging to some hypothesis space $\mathcal{F} = \{\pi_\theta, \theta \in \mathbb{R}^d\}$. For example, for discrete actions, a common choice is to parameterize the policy with a softmax distribution:

$$\pi_\theta(a|s) = \frac{e^{\theta^\top \phi(s,a)}}{\sum_{a' \in \mathcal{A}} e^{\theta^\top \phi(s,a')}}, \tag{24.13}$$

with $\phi(s,a)$ being a predefined feature vector. For continuous actions, a common approach consists in embedding a parameterized deterministic policy into a Gaussian distribution. For example, if $\mathcal{A} = \mathbb{R}$, we can consider

$$\pi_\theta(a|s) \propto e^{-\frac{1}{2}\left(\frac{a-\theta^\top \phi(s)}{\sigma}\right)^2},$$

with $\phi(s)$ being a predefined feature vector.

Let $\nu \in \Delta_{\mathcal{S}}$ be a distribution over states, defined by the user (it weights states where we would like to have good estimates). The *policy search* methods aim at solving the following optimization problem:

$$\max_{\theta \in \mathbb{R}^d} J(\theta) \text{ with } J(\theta) = \sum_{s \in \mathcal{S}} \nu(s) v_{\pi_\theta}(s) = \mathbb{E}_{S \sim \nu}[v_{\pi_\theta}(S)].$$

In dynamic programming, the aim is to find the policy that maximizes the value for *every* state. In the current setting, there are too many states, so we instead try to find the policy that maximizes the associated value function averaged over a predefined distribution over states.

24.6.1 The policy gradient theorem

There remains to know how this optimization problem can be solved. A natural idea is to perform a gradient ascent. To do so, we need to compute the gradient of J_θ:

$$\nabla_\theta J(\theta) = \sum_{s \in \mathcal{S}} \nu(s) \nabla_\theta v_{\pi_\theta}(s).$$

If we look at the gradient of the value function:

$$\begin{aligned}
\nabla_\theta v_{\pi_\theta}(s) &= \nabla_\theta \sum_{a \in \mathcal{A}} \pi_\theta(a|s) Q_{\pi_\theta}(s,a) \\
&= \sum_{a \in \mathcal{A}} \left(\nabla_\theta(\pi_\theta(a|s)) Q_{\pi_\theta}(s,a) + \pi_\theta(a|s) \nabla_\theta(Q_{\pi_\theta}(s,a))\right) \\
&= \sum_{a \in \mathcal{A}} \left(\nabla_\theta(\pi_\theta(a|s)) Q_{\pi_\theta}(s,a) + \pi_\theta(a|s) \nabla_\theta(r(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a) v_{\pi_\theta}(s'))\right) \\
&= \sum_{a \in \mathcal{A}} \pi_\theta(a|s) \left(Q_{\pi_\theta}(s,a) \nabla_\theta \ln \pi_\theta(a|s) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a) \nabla_\theta v_{\pi_\theta}(s')\right),
\end{aligned}$$

where we used a classic log-trick for the last line[19] We can see that componentwise, this is a Bellman evaluation equation for the policy π_θ and the reward $Q_{\pi_\theta}(s,a) \nabla_{\theta_j} \ln \pi_\theta(a|s)$. Let $1 \leq j \leq d$ and θ_j be the jth component of the vector θ, write also $\mathfrak{R}(s) = \sum_{a \in \mathcal{A}} \pi_\theta(a|s) Q_{\pi_\theta}(s,a) \nabla_{\theta_j} \ln \pi_\theta(a|s)$, we have equivalently that

$$\forall 1 \leq j \leq d, \quad \nabla_{\theta_j} v_{\pi_\theta} = (I - \gamma P_\pi)^{-1} \mathfrak{R}.$$

Notice that the componentwise gradient of the objective can be written as $\nabla_{\theta_j} J(\theta) = \nu^\top \nabla_{\theta_j} v_{\pi_\theta}$, we therefore have

$$\nabla_{\theta_j} J(\theta) = \nu^\top (I - \gamma P_\pi)^{-1} \mathfrak{R}.$$

The quantity defined as

$$d_{\nu,\pi} = (1-\gamma) \nu^\top (I - \gamma P_\pi)^{-1}$$

[19] This log-trick is the fact that $\nabla \pi(a|s) = \pi(a|s) \nabla \ln \pi(a|s)$. The log is the reason why we consider stochastic policies (no action can have probability zero, or the log is ill defined).

is a distribution. It correspond to the state occupancy obtained by sampling an initial state according to ν and then following the optimal policy for a random time drawn from a geometric distribution of parameter $\frac{1}{1-\gamma}$. From the two previous equations, we can finally write

$$\begin{aligned}\nabla_\theta J(\theta) &= \frac{1}{1-\gamma}\sum_{s\in\mathcal{S}} d_{\nu,\pi(s)} \sum_{a\in\mathcal{A}} \pi_\theta(a|s) Q_{\pi_\theta}(s,a) \nabla_\theta \ln \pi_\theta(a|s) \\ &= \frac{1}{1-\gamma}\mathbb{E}_{S\sim d_{\nu,\pi}, A\sim\pi_\theta(.|S)}[Q_{\pi_\theta}(S,A)\nabla_\theta \ln \pi_\theta(A|S)]. \end{aligned} \tag{24.14}$$

This result is called the policy gradient theorem, see Sutton et al. (1999) for an alternative derivation (who first provided this result).

A local maximum can thus be searched for by doing a gradient ascent,

$$\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta),$$

with α being a learning rate. The gradient can be estimated using Monte Carlo rollouts, see Baxter and Bartlett (2001).

24.6.2 Actor-critic methods

Most of the methods we are considered so far are called critic methods (as quality functions are estimated, not the policies, and the value is a critic of the policy). Policy search (and DPI, the approximate policy iteration algorithm that reduces to a sequence of classification problems) is called an actor method, as only the policy is learnt, and it is the object which acts with the system.

Sometime, it is possible to learn the policy and the quality function. Such methods are called *actor-critic* methods. Consider for example the gradient in Eq. (24.14). It involves a Q-function, which can be estimated pointwise using rollouts. Now, we have studied methods for approximating such a function. The question we study now is: can we replace Q_π in Eq. 24.14 by an approximation $Q_w \in \mathcal{H}$, without changing the gradient?

Policy gradient with a critic

Let $Q_w \in \mathcal{H}$ be a parameterized function of $\mathbb{R}^{\mathcal{S}\times\mathcal{A}}$ (the parametrization will be made explicit later). We would like to replace Q_π by Q_w in Eq. 24.14. To do so, we must have (to shorten the notations, we write "$S\sim d_{\nu,\pi}, A\sim\pi_\theta(.|S)$" as $d_{\nu,\pi}$):

$$\begin{aligned}&\mathbb{E}_{d_{\nu,\pi}}[Q_{\pi_\theta}(S,A)\nabla_\theta \ln \pi_\theta(A|S)] = \mathbb{E}_{d_{\nu,\pi}}[Q_w(S,A)\nabla_\theta \ln \pi_\theta(A|S)] \\ \Leftrightarrow &\mathbb{E}_{d_{\nu,\pi}}[(Q_{\pi_\theta}(S,A) - Q_w(S,A))\nabla_\theta \ln \pi_\theta(A|S)] = 0. \end{aligned} \tag{24.15}$$

Now, assume that we have

$$\nabla_\theta \ln \pi_\theta(a|s) = \nabla_w Q_w(s,a), \quad \forall (s,a)\in\mathcal{S}\times\mathcal{A}.$$

Injecting this into Eq. (24.15), one can recognize a gradient:

$$\mathbb{E}_{d_{\nu,\pi}}[(Q_{\pi_\theta}(S,A) - Q_w(S,A))\nabla_w Q_w(S,A)] = 0 \Leftrightarrow \nabla_w \mathbb{E}_{d_{\nu,\pi}}[(Q_{\pi_\theta}(S,A) - Q_w(S,A))^2] = 0.$$

In other words, Q_w must be a local optimum of the risk based on the ℓ_2-loss, with the state-action distribution given by $d_{\nu,\pi}$, and with the target function being Q_π.

We have just shown that if the parametrization of the state-action value function is *compatible*, in the sense that

$$\forall (s,a)\in\mathcal{S}\times\mathcal{A}, \quad \nabla_\theta \ln \pi_\theta(a|s) = \nabla_w Q_w(s,a),$$

and if Q_w is a local optimum of the risk based on the ℓ_2-loss, with state-action distribution given by $d_{\nu,\pi}$, and with the target function being Q_π, that is

$$\nabla_w \mathbb{E}_{d_{\nu,\pi}}[(Q_{\pi_\theta}(S,A) - Q_w(S,A))^2] = 0,$$

then the gradient satisfies

$$\nabla_\theta J(\theta) = \mathbb{E}_{d_{\nu,\pi}}[Q_w(S,A)\nabla_\theta \ln \pi_\theta(A|S)].$$

So, the state-action value function appearing in the gradient can be replaced by its projection onto the hypothesis space of compatible functions. This result has first been given by Sutton et al. (1999). Notice that if formally the problem should be resolved using Monte Carlo rollouts, in practice temporal difference algorithms are often used (and they do not compute the projection, in general).

Let see what this compatibility condition gives with the softmax parametrization of Eq. (24.13). We have

$$\begin{aligned}\nabla_\theta \ln \pi_\theta(a|s) &= \nabla_\theta \ln \frac{e^{\theta^\top \phi(s,a)}}{\sum_{a' \in \mathcal{A}} e^{\theta^\top \phi(s,a')}} \\ &= \phi(s,a) - \frac{\sum_{a' \in \mathcal{A}} \phi(s,a') e^{\theta^\top \phi(s,a')}}{\sum_{a' \in \mathcal{A}} e^{\theta^\top \phi(s,a')}} \\ &= \phi(s,a) - \sum_{a' \in \mathcal{A}} \pi_\theta(a'|s)\phi(s,a').\end{aligned}$$

So, a compatible function approximation would be

$$Q_w(s,a) = w^\top(\phi(s,a) - \sum_{a' \in \mathcal{A}} \pi_\theta(a'|s)\phi(s,a')).$$

Yet, notice that we would have $\sum_{a \in \mathcal{A}} \pi_\theta(a|s) Q_w(s,a) = 0$, for any $s \in \mathcal{S}$. This has no reason to be true for a quality function. Yet, this is true for the *advantage function*, defined as

$$A_\pi(s,a) = Q_\pi(s,a) - v_\pi(s).$$

Indeed, this would be true for any compatible approximator, as

$$\sum_{a \in \mathcal{A}} \pi(a|s)\nabla_\theta \ln \pi_\theta(a|s) = \sum_{a \in \mathcal{A}} \nabla_\theta \pi_\theta(a|s) = \nabla_\theta \sum_{a \in \mathcal{A}} \pi_\theta(a|s) = \nabla_\theta 1 = 0. \tag{24.16}$$

However, this is not really a problem. Indeed, let $v \in \mathbb{R}^{\mathcal{S}}$ be any function depending only on states. One can easily show that $\mathbb{E}_{d_{\nu,\pi}}[v(s)\nabla_\theta \ln \pi_\theta(a|s)] = 0$, using the same trick as in Eq. (24.16). Therefore, we have

$$\forall v \in \mathbb{R}^{\mathcal{S}}, \quad J(\theta) = \mathbb{E}_{d_{\nu,\pi}}[(Q_\pi(S,A) + v(s))\nabla_\theta \ln \pi_\theta(A|S)].$$

This means that we can take simply

$$Q_w(s,a) = \theta^\top \phi(s,a)$$

as a compatible function representation for the estimated quality function.

Natural policy gradient

An alternative to gradient ascent is *natural gradient* ascent. The natural gradient is the gradient premultiplied by the inverse of the Fisher information matrix (instead of following the steepest direction in the parameter space, it follows the steepest direction with respect to Fisher metric, which tends to be much more efficient empirically, see Amari (1998b) for more about natural gradients).

Generally, the natural gradient $\tilde{\nabla}$ is defined as

$$\tilde{\nabla}_\theta J(\theta) = F(\theta)^{-1} \nabla_\theta J(\theta),$$

with $F(\theta)$ the Fisher information matrix. In our case, the matrix is defined as (Peters and Schaal, 2008)

$$F(\theta) = \mathbb{E}_{d_{\nu,\pi}}[\nabla_\theta \ln \pi_\theta(A|S)(\nabla_\theta \ln \pi_\theta(A|S))^\top].$$

Let Q_w be a linearly parameterized function satisfying the required conditions, that is $Q_w(s,a) = w^\top \nabla_\theta \ln \pi_\theta(a|s)$ and $\nabla_w \mathbb{E}_{d_{\nu,\pi}}[(Q_{\pi_\theta}(S,A) - Q_w(S,A))^2] = 0$, the we have

$$\begin{aligned}
\tilde{\nabla}_\theta J(\theta) &= F(\theta)^{-1} \nabla_\theta J(\theta) \\
&= (\mathbb{E}_{d_{\nu,\pi}}[\nabla_\theta \ln \pi_\theta(A|S)(\nabla_\theta \ln \pi_\theta(A|S))^\top])^{-1} \mathbb{E}_{d_{\nu,\pi}}[\nabla_\theta \ln \pi_\theta(A|S)^\top w \nabla_\theta \ln \pi_\theta(A|S)] \\
&= (\mathbb{E}_{d_{\nu,\pi}}[\nabla_\theta \ln \pi_\theta(A|S)(\nabla_\theta \ln \pi_\theta(A|S))^\top])^{-1} \mathbb{E}_{d_{\nu,\pi}}[\nabla_\theta \ln \pi_\theta(A|S)(\nabla_\theta \ln \pi_\theta(A|S))^\top] w \\
&= w
\end{aligned}$$

Therefore, the policy parameters are simply updated using the parameters computed for the quality function. The related algorithms are called natural actor-critics and have been introduced by Peters and Schaal (2008). For more about policy search and actor-critics, the interested reader can refer to Grondman et al. (2012); Deisenroth et al. (2013)

Bibliography

Amari, S. (1998a). Natural Gradient Works Efficiently in Learning. *Neural Computation*, 10(2):251–276.

Amari, S.-I. (1998b). Natural gradient works efficiently in learning. *Neural Computation*, 10(2):251–276.

Andrieu, C., de Freitas, N., Doucet, A., and Jordan, M. I. (2003). An Introduction to MCMC for Machine Learning. *Machine Learning*, 50(1-2):5–43.

Antos, A., Szepesvári, C., and Munos, R. (2008). Learning near-optimal policies with Bellman-residual minimization based fitted policy iteration and a single sample path. *Machine Learning*, 71(1):89–129.

Auer, P., Cesa-Bianchi, N., and Fischer, P. (2002). Finite-time analysis of the multiarmed bandit problem. *Machine learning*, 47(2-3):235–256.

Ávila Pires, B., Szepesvari, C., and Ghavamzadeh, M. (2013). Cost-sensitive multiclass classification risk bounds. In *International Conference on Machine Learning (ICML)*, pages 1391–1399.

Barron, A. R. (1988). Complexity Regularization with Application to Artificial Neural Networks. In *Nonparametric Functional Estimation and Related Topics*, pages 561–576. Kluwer Academic Publishers.

Bartlett, P. L., Boucheron, S., and Lugosi, G. (2002). Model Selection and Error Estimation. *Machine Learning*, 48(1-3):85–113.

Bartlett, P. L., Jordan, M. I., and McAuliffe, J. D. (2006). Convexity, classification, and risk bounds. *Journal of the American Statistical Association*, 101(473):138–156.

Baxter, J. and Bartlett, P. L. (2001). Infinite-horizon policy-gradient estimation. *Journal of Artificial Intelligence Research*, pages 319–350.

Beijbom, O., Saberian, M., Kriegman, D., and Vasconcelos, N. (2014). Guess-averse loss functions for cost-sensitive multiclass boosting. In *International Conference on Machine Learning (ICML)*, pages 586–594.

Bellman, R. (1957). A Markovian Decision Process. *Journal of Mathematics and Mechanics*, 6(5):679–684.

Bengio, Y., Courville, A. C., and Vincent, P. (2013). Representation Learning: A Review and New Perspectives. *IEEE Trans. Pattern Anal. Mach. Intell.*, 35(8):1798–1828.

Bengio, Y., Goodfellow, I. J., and Courville, A. (2015). Deep Learning. Book in preparation for MIT Press.

Bengio, Y., Lamblin, P., Popovici, D., and Larochelle, H. (2007). Greedy layer-wise training of deep networks. In Schölkopf, B., Platt, J., and Hoffman, T., editors, *Advances in Neural Information Processing Systems 19*, pages 153–160. MIT Press, Cambridge, MA.

Bengio, Y., Simard, P. Y., and Frasconi, P. (1994). Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, 5(2):157–166.

Bertsekas, D. P. (1995). *Dynamic Programming and Optimal Control*. Athena Scientific.

Bertsekas, D. P. and Tsitsiklis, J. N. (1996). *Neuro-Dynamic Programming*. Athena Scientific.

Bishop, C. M. (1995). *Neural Networks for Pattern Recognition*. Oxford University Press, Oxford, UK.

Boucheron, S., Lugosi, G., and Massart, P. (2013). Concentration inequalities: a non asymptotic theory of independence.

Bousquet, O., Boucheron, S., and Lugosi, G. (2004). Introduction to Statistical Learning Theory. In *Advanced Lectures on Machine Learning*, pages 169–207.

Bradtke, S. J. and Barto, A. G. (1996). Linear Least-Squares algorithms for temporal difference learning. *Machine Learning*, 22(1-3):33–57.

Brafman, R. I. and Tennenholtz, M. (2003). R-max:a general polynomial time algorithm for near-optimal reinforcement learning. *The Journal of Machine Learning Research*, 3:213–231.

Breiman, L. (1996a). Bagging predictors. *Machine learning*, 24(2):123–140.

Breiman, L. (1996b). Stacked regressions. *Machine learning*, 24(1):49–64.

Breiman, L. (1999). Pasting small votes for classification in large databases and on-line. *Machine Learning*, 36(1-2):85–103.

Breiman, L. (2001). Random forests. *Machine learning*, 45(1):5–32.

Breiman, L., Friedman, J., Olshen, R., and Stone, C. (1983). *CART: Classification and Regression Trees*. Wadsworth: Belmont, CA.

Breiman, L., Friedman, J., Stone, C. J., and Olshen, R. A. (1984). *Classification and regression trees*. CRC press.

Brent, R. (1973). *Algorithms for minimization without derivatives*, chapter 3. Prentice-Hall.

Broomhead, D. and Lowe, D. (1988). Multivariable Functional Interpolation and Adaptive Networks. *Complex Systems*, 2:321–355.

Bubeck, S. and Cesa-Bianchi, N. (2012). Regret Analysis of Stochastic and Nonstochastic Multi-armed Bandit Problems. *Foundations and Trends in Machine Learning*, 5(1):1–122.

Bunea, F., Tsybakov, A., Wegkamp, M., et al. (2007). Sparsity oracle inequalities for the lasso. *Electronic Journal of Statistics*, 1:169–194.

Carreira-Perpiñán, M. Á. and Hinton, G. (2005). On Contrastive Divergence Learning. In Cowell, R. G. and Ghahramani, Z., editors, *AISTATS*. Society for Artificial Intelligence and Statistics.

Chapelle, O. and Li, L. (2011). An empirical evaluation of thompson sampling. In *Advances in neural information processing systems*, pages 2249–2257.

Cireşan, D. C., Meier, U., Gambardella, L. M., and Schmidhuber, J. (2010). Deep, Big, Simple Neural Nets for Handwritten Digit Recognition. *Neural Computation*, 22(12):3207–3220.

Ciresan, D. C., Meier, U., Gambardella, L. M., and Schmidhuber, J. (2011a). Convolutional Neural Network Committees for Handwritten Character Classification. In *ICDAR*, pages 1135–1139. IEEE Computer Society.

Ciresan, D. C., Meier, U., Masci, J., Gambardella, L. M., and Schmidhuber, J. (2011b). Flexible, High Performance Convolutional Neural Networks for Image Classification. In Walsh, T., editor, *IJCAI*, pages 1237–1242. IJCAI/AAAI.

Ciressan, D., Meier, U., and Schmidhuber, J. (2012). Multi-column Deep Neural Networks for Image Classification. Technical report, IDSIA.

Cortes, C. and Vapnik, V. (1995). Support-vector networks. *Machine learning*, 20(3):273–297.

Cottrell, M., Fort, J., and Pagès, G. (1998). Theoretical aspects of the SOM algorithm. *Neurocomputing*, 21(1–3):119–138.

Cover, T. M. (1965). Geometrical and Statistical Properties of Systems of Linear Inequalities with Applications in Pattern Recognition. *Electronic Computers, IEEE Transactions on*, EC-14(3):326–334.

Cox, T. F. and Cox, M. (2000). *Multidimensional Scaling, Second Edition*. Chapman and Hall/CRC, 2 edition.

Cristanini, N. and Shawe-Taylor, J. (2000). *An Introduction to Support Vector Machines and other kernel-based learning methods*. Cambridge University Press.

Cucker, F. and Smale, S. (2001). On the mathematical foundations of learning. *Bulletin of the american mathematical society*, 39(1):1–49.

Cybenko, G. (1989). Approximation by Superpositions of a Sigmoidal Function. *Mathematics of Control, Signals, and Systems*, 2(4):303–314.

Dahl, G. E., Jaitly, N., and Salakhutdinov, R. (2014). Multi-task Neural Networks for QSAR Predictions. *CoRR*, abs/1406.1231.

Dauphin, Y. N., Pascanu, R., Gülçehre, c., Cho, K., Ganguli, S., and Bengio, Y. (2014). Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. In Ghahramani, Z., Welling, M., Cortes, C., Lawrence, N. D., and Weinberger, K. Q., editors, *NIPS*, pages 2933–2941.

de Farias, D. P. and Van Roy, B. (2003). The linear programming approach to approximate dynamic programming. *Operations Research*, 51(6):850–865.

de Farias, D. P. and Van Roy, B. (2004). On constraint sampling in the linear programming approach to approximate dynamic programming. *Mathematics of operations research*, 29(3):462–478.

Deisenroth, M. P., Neumann, G., Peters, J., et al. (2013). A survey on policy search for robotics. *Foundations and Trends in Robotics*, 2(1-2):1–142.

Delalleau, O. and Bengio, Y. (2011). Shallow vs. Deep Sum-Product Networks. In Shawe-Taylor, J., Zemel, R. S., Bartlett, P. L., Pereira, F. C. N., and Weinberger, K. Q., editors, *NIPS*, pages 666–674.

Deng, L., Hinton, G. E., and Kingsbury, B. (2013). New types of deep neural network learning for speech recognition and related applications: an overview. In *ICASSP*, pages 8599–8603. IEEE.

Deng, L. and Platt, J. C. (2014). Ensemble deep learning for speech recognition. In Li, H., Meng, H. M., Ma, B., Chng, E., and Xie, L., editors, *INTERSPEECH*, pages 1915–1919. ISCA.

Eberhart, R. C. and Kennedy, J. (1995). Particle swarm optimization. In *Proceedings*, volume 4, pages 1942–1948.

Efron, B., Hastie, T., Johnstone, I., and Tibshirani, R. (2004a). Least Angle Regression. *The Annals of Statistics*, 32(2):407–451.

Efron, B., Hastie, T., Johnstone, I., Tibshirani, R., et al. (2004b). Least angle regression. *The Annals of statistics*, 32(2):407–499.

Engelbrecht, A. P. (2007). *Fundamentals of Computational Swarm Intelligence*. Wiley.

Ernst, D., Geurts, P., and Wehenkel, L. (2005). Tree-Based Batch Mode Reinforcement Learning. *Journal of Machine Learning Research*, 6:503–556.

Evgeniou, T., Pontil, M., and Poggio, T. (2000). Regularization Networks and Support Vector Machines. *Advances in Computational Mathematics*, 13(1):1–50.

Farahmand, A.-m. and Szepesvári, C. (2011). Model selection in reinforcement learning. *Machine learning*, 85(3):299–332.

Freund, Y. and Schapire, R. E. (1997). A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of computer and system sciences*, 55(1):119–139.

Freund, Y. and Schapire, R. E. (1999). Large Margin Classification Using the Perceptron Algorithm. *Machine Learning*, 37(3):277–296.

Frezza-Buet, H. (2014). Online Computing of Non-Stationary Distributions Velocity Fields by an Accuracy Controlled Growing Neural Gas. *Neural Networks*, 60:203–221.

Friedman, J., Hastie, T., and Tibshirani, R. (2000). Additive logistic regression: a statistical view of boosting. *The annals of statistics*, 28(2):337–407.

Friedman, J. H. (2001). Greedy function approximation: a gradient boosting machine. *Annals of statistics*, pages 1189–1232.

Fritzke, B. (1995a). A growing neural gas network learns topologies. In Tesauro, G., Touretzky, D. S., and Leen, T. K., editors, *Advances in Neural Information Processing Systems 7*, pages 625–632. MIT Press, Cambridge MA.

Fritzke, B. (1995b). Growing Grid - a self-organizing network with constant neighborhood range and adaptation strength. *Neural Processing Letters*, 2:9–13.

Fritzke, B. (1997). *Some Competitive Learning Methods*. http://www.ki.inf.tu-dresden.de/~fritzke/JavaPaper/.

Fukushima, K. (1980). Neocognitron: A Self-Organizing Neural Network Model for a Mechanism of Pattern Recognition Unaffected by Shift in Position. *Biological Cybernetics*, 36:193–202.

Gabillon, V., Ghavamzadeh, M., and Lazaric, A. (2012). Best arm identification: A unified approach to fixed budget and fixed confidence. In *Advances in Neural Information Processing Systems*, pages 3212–3220.

Gallant, S. I. (1990). Perceptron-based learning algorithms. *IEEE Transactions on Neural Networks*, 1(2):179–191.

Geist, M. (2013-2014). Abrégé non exhaustif sur l'évaluation et la sélection de modèles et la sélection de variables. Technical report, CentraleSupélec.

Geist, M. (2015a). Précis introductif à l'apprentissage statistique. Support de cours, CentraleSupélec. http://www.metz.supelec.fr/metz/personnel/geist_mat/pdfs/poly_as_v2.pdf.

Geist, M. (2015b). Soft-max boosting. *Machine Learning*.

Geist, M. and Pietquin, O. (2013). An Algorithmic Survey of Parametric Value Function Approximation. *IEEE Transactions on Neural Networks and Learning Systems*, 24(6):845–867.

Geist, M. and Scherrer, B. (2014). Off-policy Learning with Eligibility Traces: A Survey. *Journal of Machine Learning Research (JMLR)*, 15:289–333.

Gelly, S., Wang, Y., Munos, R., and Teytaud, O. (2006). Modification of UCT with patterns in Monte-Carlo go. *Technical Report RR-6062*, 32:30–56.

Gers, F. A., Schmidhuber, J., and Cummins, F. A. (2000). Learning to Forget: Continual Prediction with LSTM. *Neural Computation*, 12(10):2451–2471.

Geurts, P., Ernst, D., and Wehenkel, L. (2006). Extremely randomized trees. *Machine learning*, 63(1):3–42.

Glorot, X. and Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In Teh, Y. W. and Titterington, D. M., editors, *AISTATS*, volume 9 of *JMLR Proceedings*, pages 249–256. JMLR.org.

Gordon, G. (1995). Stable Function Approximation in Dynamic Programming. In *International Conference on Machine Learning (IMCL)*.

Graves, A. (2013). Generating Sequences With Recurrent Neural Networks. *CoRR*, abs/1308.0850.

Graves, A., Fernández, S., Gomez, F. J., and Schmidhuber, J. (2006). Connectionist temporal classification: labelling unsegmented sequence data with recurrent neural networks. In Cohen, W. W. and Moore, A., editors, *ICML*, volume 148 of *ACM International Conference Proceeding Series*, pages 369–376. ACM.

Graves, A., rahman Mohamed, A., and Hinton, G. E. (2013). Speech recognition with deep recurrent neural networks. In *ICASSP*, pages 6645–6649. IEEE.

Graves, A. and Schmidhuber, J. (2005). Framewise phoneme classification with bidirectional LSTM and other neural network architectures. *Neural Networks*, 18(5-6):602–610.

Grondman, I., Buşoniu, L., Lopes, G. A., and Babuška, R. (2012). A survey of actor-critic reinforcement learning: Standard and natural policy gradients. *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, 42(6):1291–1307.

Grubb, A. and Bagnell, D. (2011). Generalized boosting algorithms for convex optimization. In *International Conference on Machine Learning*, pages 1209–1216.

Guenther, W. C. (1969). Shortest Confidence Intervals. *The American Statistician*, 23(1):22–25.

Guermeur, Y. (2007). Vc theory of large margin multi-category classifiers. *The Journal of Machine Learning Research*, 8:2551–2594.

Guyon, I. and Elisseeff, A. (2003). An Introduction to Variable and Feature Selection. *J. Mach. Learn. Res.*, 3:1157–1182.

Györfi, L., Kohler, M., Krzyzak, A., and Walk, H. (2006). *A distribution-free theory of nonparametric regression*. Springer.

Hall, M. A. (1999). *Correlation-based Feature Selection for Machine Learning*. PhD thesis.

Han, H.-G. and Qiao, J.-F. (2012). Adaptive Computation Algorithm for RBF Neural Network. *IEEE Trans. Neural Netw. Learning Syst.*, 23(2):342–347.

Hansen, N. (2006). The CMA evolution strategy: a comparing review. In Lozano, J., Larranaga, P., Inza, I., and Bengoetxea, E., editors, *Towards a new evolutionary computation. Advances on estimation of distribution algorithms*, pages 75–102. Springer.

Hartman Eric J., Keeler James D., and Kowalski Jacek M. (1990). Layered Neural Networks with Gaussian Hidden Units as Universal Approximations. *Neural Computation*, 2(2):210–215. doi: 10.1162/neco.1990.2.2.210.

Håstad, J. (1986). Almost Optimal Lower Bounds for Small Depth Circuits. In Hartmanis, J., editor, *STOC*, pages 6–20. ACM.

Håstad, J. and Goldmann, M. (1991). On the Power of Small-Depth Threshold Circuits. *Computational Complexity*, 1:113–129.

Hastie, T., Tibshirani, R., and Friedman, J. (2009). *The Elements of Statsitcal Learning*. Springer, 2nd edition.

Hertz, J., Krogh, A., and Palmer, R. G. (1991). *Introduction to the Theory of Neural Computation*. Addison-Wesley, Redwood City, CA.

Hinton, G. and Roweis, S. (2002). Stochastic Neighbor Embedding. In *Advances in Neural Information Processing Systems 15*, pages 833–840. MIT Press.

Hinton, G. E. (2009). Deep belief networks. *Scholarpedia*, 4(5):5947.

Hinton, G. E. (2012). A Practical Guide to Training Restricted Boltzmann Machines. In Montavon, G., Orr, G. B., and Müller, K.-R., editors, *Neural Networks: Tricks of the Trade (2nd ed.)*, volume 7700 of *Lecture Notes in Computer Science*, pages 599–619. Springer.

Hinton, G. E., Osindero, S., and Teh, Y.-W. (2006). A Fast Learning Algorithm for Deep Belief Nets. *Neural Comput.*, 18(7):1527–1554.

Hinton, G. E. and Sejnowski, T. (1986). Learning and relearning in Boltzmann machines. In *Parallel distributed processing: Explorations in the microstructure of cognition*, pages 282–317–. MIT Press, Cambridge, MA.

Ho, T. K. (1998). The random subspace method for constructing decision forests. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 20(8):832–844.

Hochreiter, S. (1991). Untersuchungen zu dynamischen neuronalen Netzen. Diploma thesis, Institut für Informatik, Lehrstuhl Prof. Brauer, Technische Universität München.

Hochreiter, S. and Schmidhuber, J. (1997). Long Short-Term Memory. *Neural Computation*, 9:1735–1780.

Hoeffding, W. (1963). Probability inequalities for sums of bounded random variables. *Journal of the American statistical association*, 58(301):13–30.

Hoeting, J. A., Madigan, D., Raftery, A. E., and Volinsky, C. T. (1999). Bayesian model averaging: a tutorial. *Statistical science*, pages 382–401.

Hornik, K., Stinchcombe, M., and White, H. (1989). Multilayer feedforward networks are universal approximators. *Neural Networks*, 2:356–366.

Hornik, K., Stinchcombe, M., and White, H. (1990). Universal Approximation of an Unknown Mapping and Its Derivatives Using Multilayer Feedforward Networks. *Neural Networks*, 3:551–560.

Hsu, D., Kakade, S. M., and Zhang, T. (2014). Random design analysis of ridge regression. *Foundations of Computational Mathematics*, 14(3):569–600.

Jacobs, R. A., Jordan, M. I., Nowlan, S. J., and Hinton, G. E. (1991). Adaptive mixtures of local experts. *Neural computation*, 3(1):79–87.

Jaeger, H. (2002). A tutorial on training recurrent neural networks, covering BPPT, RTRL, EKF and the ëcho state networkäpproach . Technical report, Fraunhofer Institute for Autonomous Intelligent Systems (AIS). http://minds.jacobs-university.de/sites/default/files/uploads/papers/ESNTutorialRev.pdf.

Jaeger, H. (2004). Harnessing Nonlinearity: Predicting Chaotic Systems and Saving Energy in Wireless Communication. *Science*, 304:78–80.

Japkowicz, N., Hanson, S. J., and Gluck, M. A. (2000). Nonlinear Autoassociation Is Not Equivalent to PCA. *Neural Computation*, 12(3):531–545.

Johansson, E. M., Dowla, F. U., and Goodman, D. M. (1991). Backpropagation Learning for Multilayer Feed-Forward Neural Networks Using the Conjugate Gradient Method. *Int. J. Neural Syst.*, 2(4):291–301.

Jordan, M. I. and Jacobs, R. A. (1994). Hierarchical mixtures of experts and the EM algorithm. *Neural computation*, 6(2):181–214.

Jouini, W., Moy, C., and Palicot, J. (2012). Decision making for cognitive radio equipment: analysis of the first 10 years of exploration. *EURASIP J. Wireless Comm. and Networking*, 2012:26.

Kaelbling, L. P., Littman, M. L., and Cassandra, A. R. (1998). Planning and acting in partially observable stochastic domains. *Artificial Intelligence*, 101(1-2):99–134.

Kanungo, T., Mount, D. M., Netanyahu, N., Piatko, C., Silverman, R., and Wu, A. Y. (2002). An efficient k-means clustering algorithm: Analysis and implementation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 24:881–892.

Karpathy, A. and Li, F.-F. (2014). Deep Visual-Semantic Alignments for Generating Image Descriptions. *CoRR*, abs/1412.2306.

Keerthi, S. S., Shevade, S. K., Bhattacharyya, C., and Murthy, K. R. K. (1999). Improvements to Platt's SMO Algorithm for SVM Classifier Design. Technical Report CD-99-14, National University of Singapore.

Kohonen, T. (1989). *Self-Organization and Associative Memory*, volume 8 of *Springer Series in Information Sciences*. Springer-Verlag.

Kohonen, T. (2013). Essentials of the self-organizing maps. *Neural Networks*, (37):52–65.

Korda, N., Kaufmann, E., and Munos, R. (2013). Thompson sampling for 1-dimensional exponential family bandits. In *Advances in Neural Information Processing Systems*, pages 1448–1456.

Krizhevsky, A. (2014). One weird trick for parallelizing convolutional neural networks. *CoRR*, abs/1404.5997.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). ImageNet Classification with Deep Convolutional Neural Networks. In Bartlett, P. L., Pereira, F. C. N., Burges, C. J. C., Bottou, L., and Weinberger, K. Q., editors, *NIPS*, pages 1106–1114.

Lagoudakis, M. G. and Parr, R. (2003). Least-squares policy iteration. *Journal of Machine Learning Research (JMLR)*, 4:1107–1149.

Lawson, C. and Hanson, R. (1974). *Solving least squares problems*. Prentice-Hall series in automatic computation. Prentice-Hall, Englewood Cliffs, NJ.

Lazaric, A., Ghavamzadeh, M., and Munos, R. (2010). Analysis of a classification-based policy iteration algorithm. In *International Conference on Machine Learning (ICML)*, pages 607–614.

Lecun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324.

LeCun, Y., Bottou, L., Orr, G., and Müller, K.-R. (1998). Efficient BackProp. In Orr, G. and Müller, K.-R., editors, *Neural Networks: Tricks of the Trade*, volume 1524 of *Lecture Notes in Computer Science*, pages 9–50. Springer Berlin Heidelberg.

Lee, J. A. and Verleysen, M. (2007). *Nonlinear dimensionality reduction*. Springer.

Lee, Y., Lin, Y., and Wahba, G. (2004). Multicategory support vector machines: Theory and application to the classification of microarray data and satellite radiance data. *Journal of the American Statistical Association*, 99(465):67–81.

Linde, Y., Buzo, A., and Gray, R. M. (1980). Algorithm for Vector Quantization Design. *IEEE transactions on communications systems*, 28(1):84–95.

Lloyd, S. P. (1982). Least Squares Quantization in PCM. *IEEE Transactions on Information Theory*, 28(2):129–137.

Louppe, G. and Geurts, P. (2012). Ensembles on random patches. In *Machine Learning and Knowledge Discovery in Databases*, pages 346–361. Springer.

Lugosi, G. and Wegkamp, M. (2004). Complexity regularization via localized random penalties. *The Annals of Statistic*, 32(4):1679–1697.

Lukosevicius, M. (2012). A Practical Guide to Applying Echo State Networks. In Montavon, G., Orr, G. B., and Müller, K.-R., editors, *Neural Networks: Tricks of the Trade (2nd ed.)*, volume 7700 of *Lecture Notes in Computer Science*, pages 659–686. Springer.

MacQueen, J. B. (1967). Some Methods for Classification and Analysis of MultiVariate Observations. In Cam, L. M. L. and Neyman, J., editors, *Proc. of the fifth Berkeley Symposium on Mathematical Statistics and Probability*, volume 1, pages 281–297. University of California Press.

Martens, J. (2010). Deep learning via Hessian-free optimization. In Fürnkranz, J. and Joachims, T., editors, *Proc. of the International Conference on Machine Learning (ICML) 2010*, pages 735–742. Omnipress.

Martinez, T. M. and Schulten, K. J. (1994). Topology Representing Networks. *Neural Networks*, 7(3):507–522.

Mason, L., Baxter, J., Bartlett, P., and Frean, M. (1999). Boosting algorithms as gradient descent in function space. In *Neural Information Processing Systems (NIPS)*.

McCulloch, W. S. and Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5:115–133.

Mhaskar, H. and Micchelli, C. (1995). Degree of Approximation by Neural and Translation Networks with a Single Hidden Layer. *Advances in Applied Mathematics*, 16(2):151–183.

Munos, R. (2014). From bandits to Monte-Carlo Tree Search: The optimistic principle applied to optimization and planning. *Foundations and Trends in Machine Learning*.

Muselli, M. (1997). On convergence properties of pocket algorithm. *IEEE Transactions on Neural Networks*, 8(3):623–629.

Nair, V. and Hinton, G. E. (2010). Rectified Linear Units Improve Restricted Boltzmann Machines. In Fürnkranz, J. and Joachims, T., editors, *ICML*, pages 807–814. Omnipress.

Ormoneit, D. and Sen, S. (2002). Kernel-Based Reinforcement Learning. *Machine Learning*, 49:161–178.

Ozay, M. and Vural, F. T. Y. (2012). A new fuzzy stacked generalization technique and analysis of its performance. *arXiv preprint arXiv:1204.0171*.

Park, J. and Sandberg, I. W. (1991). Universal Approximation Using Radial-Basis-Function Networks. *Neural Computation*, 3:246–257.

Pascanu, R., Montufar, G., and Bengio, Y. (2013). On the number of inference regions of deep feed forward networks with piece-wise linear activations. *CoRR*, abs/1312.6098.

Pearson, K. (1901). LIII. On lines and planes of closest fit to systems of points in space. *Philosophical Magazine Series 6*, 2(11):559–572.

Peng, J. X., Li, K., and Irwin, G. W. (2007). A Novel Continuous Forward Algorithm for RBF Neural Modelling. *IEEE Trans. Automat. Contr.*, 52(1):117–122.

Peters, J. and Schaal, S. (2008). Natural Actor-Critic. *Neurocomputing*, 71:1180–1190.

Platt, J. (1998). Fast training of support vector machines using sequential minimal optimization. In schölkopf, B., Burges, C., and Smola, A., editors, *Advances in Kernel Methods: Support Vector Machines*. MIT Press, Cambridge, MA.

Poupart, P., Vlassis, N., Hoey, J., and Regan, K. (2006). An analytic solution to discrete bayesian reinforcement learning. In *International Conference on Machine Learning (ICML)*, pages 697–704.

Prechelt, L. (1996). Early Stopping-But When? In Orr, G. B. and Müller, K.-R., editors, *Neural Networks: Tricks of the Trade*, volume 1524 of *Lecture Notes in Computer Science*, pages 55–69. Springer.

Pudil, P., Novovičová, and Kittler, J. (1993). Floating search methods in feature selection. *Pattern Recognition Letters*, 15:1119–1125.

Puterman, M. L. (1994). *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley-Interscience.

Quinlan, J. R. (1993). C4. 5: Programs for machine learning.

Ranzato, M. A., Poultney, C. S., Chopra, S., and LeCun, Y. (2006). Efficient Learning of Sparse Representations with an Energy-Based Model. In Schölkopf, B., Platt, J. C., and Hoffman, T., editors, *NIPS 2006*, pages 1137–1144. MIT Press.

Riedmiller, M. (2005). Neural fitted q iteration–first experiences with a data efficient neural reinforcement learning method. In *European Conference on Machine Learning (ECML)*, pages 317–328. Springer.

Rodan, A. and Tiňo, P. (2011). Minimum Complexity Echo State Network. *IEEE Transactions on Neural Networks*, 22(1):131–144.

Rosenblatt, F. (1962). *Principles of neurodynamics; perceptrons and the theory of brain mechanisms*. Washington, Spartan Books.

Roweis, S. T. and Saul, L. K. (2000). Nonlinear dimensionality reduction by locally linear embedding. *Science*, 290:2323–2326.

Rumelhart, D., Hinton, G., and Williams, R. (1986). *Learning Internal Representations by Error Propagation*, pages 318–362. MIT Press.

Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M. S., Berg, A. C., and Li, F.-F. (2014). ImageNet Large Scale Visual Recognition Challenge. *CoRR*, abs/1409.0575.

Salakhutdinov, R. and Hinton, G. E. (2009). Deep Boltzmann Machines. *Journal of Machine Learning Research - Proceedings Track*, 5:448–455.

Sammon, J. W. (1969). A Nonlinear Mapping for Data Structure Analysis. *IEEE Trans. Comput.*, 18(5):401–409.

Schapire, R. E. and Freund, Y. (2012). *Boosting: Foundations and algorithms*. MIT press.

Scherer, D., Müller, A. C., and Behnke, S. (2010). Evaluation of Pooling Operations in Convolutional Architectures for Object Recognition. In Diamantaras, K. I., Duch, W., and Iliadis, L. S., editors, *ICANN (3)*, volume 6354 of *Lecture Notes in Computer Science*, pages 92–101. Springer.

Scherrer, B. (2010). Should one compute the Temporal Difference fix point or minimize the Bellman Residual? The unified oblique projection view. In *International Conference on Machine Learning (ICML)*, pages 959–966.

Scherrer, B., Ghavamzadeh, M., Gabillon, V., Lesner, B., and Geist, M. (2015). Approximate Modified Policy Iteration and its Application to the Game of Tetris. *Journal of Machine Learning Research*.

Schmidhuber, J. (1992). Learning Complex, Extended Sequences Using the Principle of History Compression. *Neural Computation*, 4(2):234–242.

Schmidhuber, J. (2015). Deep learning in neural networks: An overview. *Neural Networks*, 61:85–117.

Schölkopf, B., Platt, J., Shawe-Taylor, J., Smola, A. J., and Williamson, R. C. (2001). Estimating the support of a high-dimensional distribution. *Neural Computation*, 13:1443–1471.

Scholkopf, B., Smola, A., and Müller, K.-R. (1999). Kernel principal component analysis. In *Advances in kernel methods - support vector learning*, pages 327–352. MIT Press.

Schölkopf, B., Smola, A. J., Williamson, R. C., and Bartlett, P. L. (2000). New support vector algorithms. *Neural Computation*, 12:1207–1245.

Schwenker, F., Kestler, H. A., and Palm, G. (2001). Three learning phases for radial-basis-function networks. *Neural Networks*, 14(4-5):439–458.

Shawe-Taylor, J. and Cristanini, N. (2004). *Kernel Methods for Pattern Analysis*. Cambridge University Press.

Sigaud, O. and Buffet, O. (2013). *Markov decision processes in artificial intelligence*. John Wiley & Sons.

Simard, P. Y., Steinkraus, D., and Platt, J. C. (2003). Best Practices for Convolutional Neural Networks Applied to Visual Document Analysis. In *ICDAR*, pages 958–962. IEEE Computer Society.

Smolensky, P. (1986). Information processing in dynamical systems: Foundations of harmony theory. In *Parallel distributed processing: Explorations in the microstructure of cognition*, pages 194–281–. MIT Press, Cambridge, MA.

Smyth, P. and Wolpert, D. (1999). Linearly combining density estimators via stacking. *Machine Learning*, 36(1-2):59–83.

Soheili, N. (2014). *Elementary Algorithms for Solving Convex Optimization Problems*. PhD thesis, Carnegie Mellon University.

Soheili, N. and Pena, J. (2013). A primal-dual smooth perceptron-von neumann algorithm. *Discrete Geometry and Optimization*, 69:303–320.

Somol, P., Novovicova, J., and Pudil, P. (2010). *Pattern recognition recent advances*, chapter Efficient Feature Subset Selection and Subset Size Optimization. InTech.

Somol, P., Pudil, P., Novovicová, J., and Paclík, P. (1999). Adaptive floating search methods in feature selection. *Pattern Recognition Letters*, 20(11-13):1157–1163.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research*, 15:1929–1958.

Steinwart, I. (2007). How to compare different loss functions and their risks. *Constructive Approximation*, 26(2):225–287.

Sutskever, I. (2013). *Training recurrent neural networks*. PhD thesis, University of Toronto.

Sutskever, I., Martens, J., and Hinton, G. E. (2011). Generating Text with Recurrent Neural Networks. In Getoor, L. and Scheffer, T., editors, *ICML*, pages 1017–1024. Omnipress.

Sutskever, I., Vinyals, O., and Le, Q. V. (2014). Sequence to Sequence Learning with Neural Networks. In Ghahramani, Z., Welling, M., Cortes, C., Lawrence, N. D., and Weinberger, K. Q., editors, *NIPS*, pages 3104–3112.

Sutton, R. S. and Barto, A. G. (1998). *Reinforcement Learning: An Introduction*. The MIT Press.

Sutton, R. S., McAllester, D. A., Singh, S. P., and Mansour, Y. (1999). Policy Gradient Methods for Reinforcement Learning with Function Approximation. In *Neural Information Processing Systems (NIPS)*, pages 1057–1063.

Szepesvári, C. (2010). Algorithms for reinforcement learning. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 4(1):1–103.

Tenenbaum, J. B., de Silva, V., and Langford, J. C. (2000). A Global Geometric Framework for Nonlinear Dimensionality Reduction. *Science*, 290(5500):2319.

Thomas, P., Theocharous, G., and Ghavamzadeh, M. (2015). High confidence off-policy evaluation. In *Twenty-Ninth AAAI Conference on Artificial Intelligence*.

Thompson, W. R. (1933). On the likelihood that one unknown probability exceeds another in view of the evidence of two samples. *Biometrika*, pages 285–294.

Tibshirani, R. (1996a). Regression shrinkage and selection via the LASSO. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 267–288.

Tibshirani, R. (1996b). Regression shrinkage and selection via the Lasso. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 267–288.

Tikhonov, A. (1963). Solution of incorrectly formulated problems and the regularization method. In *Soviet Mathematics*, volume 5, pages 1035–1038.

Tipping, M. E. (2001). Sparse Kernel Principal Component Analysis. In Leen, T., Dietterich, T., and Tresp, V., editors, *Advances in Neural Information Processing Systems 13*, pages 633–639. MIT Press.

Triefenbach, F., Jalalvand, A., Schrauwen, B., and Martens, J.-P. (2010). Phoneme Recognition with Large Hierarchical Reservoirs. In Lafferty, J. D., Williams, C. K. I., Shawe-Taylor, J., Zemel, R. S., and Culotta, A., editors, *NIPS*, pages 2307–2315. Curran Associates, Inc.

Valiant, L. G. (1984). A theory of the learnable. *Communications of the ACM*, 27(11):1134–1142.

van der Maaten, L. (2014). Accelerating t-SNE using Tree-Based algorithms. *Journal of Machine Learning Research*, 15:3221–3245.

van der Maaten, L. and Hinton, G. (2008). Visualizaing Data using t-SNE. *Journal of Machine Learning Research*, 9:2579–2605.

Vapnik, V. N. (1998). *Statistical Learning Theory*. Wiley-Interscience.

Vapnik, V. N. (1999). An overview of statistical learning theory. *Neural Networks, IEEE Transactions on*, 10(5):988–999.

Vapnik, V. N. (2000). *The Nature of Statistical Learning Theory*. Statistics for Engineering and Information Science. Springer.

Vincent, P., Larochelle, H., Bengio, Y., and Manzagol, P.-A. (2008). Extracting and composing robust features with denoising autoencoders. In Cohen, W. W., McCallum, A., and Roweis, S. T., editors, *ICML*, volume 307 of *ACM International Conference Proceeding Series*, pages 1096–1103. ACM.

Viola, P. and Jones, M. (2001). Rapid object detection using a boosted cascade of simple features. In *Computer Vision and Pattern Recognition (CVPR)*. IEEE.

Vlassis, N., Ghavamzadeh, M., Mannor, S., and Poupart, P. (2012). Bayesian reinforcement learning. In *Reinforcement Learning*, pages 359–386. Springer.

Waibel, A. H., Hanazawa, T., Hinton, G. E., Shikano, K., and Lang, K. J. (1989). Phoneme recognition using time-delay neural networks. *IEEE Trans. Acoustics, Speech, and Signal Processing*, 37(3):328–339.

Wang, C., Venkatesh, S. S., and Judd, J. S. (1993). Optimal Stopping and Effective Machine Complexity in Learning. In Cowan, J. D., Tesauro, G., and Alspector, J., editors, *NIPS*, pages 303–310. Morgan Kaufmann.

Wang, J. (2013). Boosting the generalized margin in cost-sensitive multiclass classification. *Journal of Computational and Graphical Statistics*, 22(1):178–192.

Wegkamp, M. (2003). Model selection in nonparametric regression. *Ann. Statist.*, 31(1):252–273.

Werbos, P. (1981). Application of advances in nonlinear sensitivity analysis. In *Proc. of the 10th IFIP conference*, pages 762–770.

Werbos, P. J. (1988). Generalization of backpropagation with application to a recurrent gas market model. *Neural Networks*, 1(4):339–356.

Widrow, B. and Hoff, M. (1962). Associative Storage and Retrieval of Digital Information in Networks of Adaptive Neurons. *Biological Prototypes and Synthetic Systems*, 1.

Williams, R. J. and Zipser, D. (1989). A Learning Algorithm for Continually Running Fully Recurrent Neural Networks. *Neural Computation*, 1(2):270–280.

Wolpert, D. H. (1992). Stacked generalization. *Neural networks*, 5(2):241–259.

Yu, D. and Deng, L. (2015). *Automatic Speech Recognition A Deep Learning Approach*. Springer-Verlag.

Zeiler, M., Ranzato, M., Monga, R., Mao, M., Yang, K., Le, Q., Nguyen, P., Senior, A., Vanhoucke, V., Dean, J., and Hinton, G. (2013). On Rectified Linear Units For Speech Processing. In *38th International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 3517–3521, Vancouver. IEEE.

Zhao, Z., Morstatter, F., Sharma, S., Alelyani, S., Anand, A., and Liu, H. (2008). Advancing Feature Selection Research - ASU Feature Selection Repository. Technical report, Arizona State University.

Zou, H. and Hastie, T. (2003). Regularization and Variable Selection via the Elastic Net. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 67(2):301–320.

Index

ϵ-SVR, 128
ν-SVC, 127
ν-SVR, 128

active learning, 21
active sampling, 14
actor-critic, 260
AdaBoost, 219
ADALINE, 166
ambient space, 28
approximate dynamic programming (ADP), 247
approximate policy iteration, 252
approximate value iteration, 250

bag-of-words, 68
Bagging, 215
bagging, 32
bags of words, 117
bandit, 231
batch sampling, 14
Bayes' rule, 36
Bayesian inference, 36, 40
Bayesian learning, 40
Bayesian model averaging, 213
Bellman evaluation operator, 242, 248
Bellman optimality operator, 242, 248
bias-variance decomposition, 53
bias-variance trade-off, 32
binary loss, 29
Boosting, 219
boosting, 33
bootstrapping, 32

C-SVC, 127
calibration, 63
chain rule, 175
classification, 19
classification tree, 212
competition, 147
concentration inequality, 233
confusion matrix, 44
conjugate prior, 41
consistency of the ERM principle, 55
Correlation-based feature selection, 72
cost sensitive, 46
curse of dimensionality, 29, 69

data augmentation, 183
dataset, 12
dead units, 150
decision stump, 212
decision trees, 209
Delaunay triangulation, 140
delta rule, 168
denoising autoencoders, 184
density function, 135
distortion, 133
double centering, 82
dual problem, 103
Dynamic Programming (DP), 243

Echo state networks, 193
embedded, 70
empirical risk, 30, 53
empirical risk minimization, 25, 31, 53
ensemble learning, 32
ERM, 31
Euclidean distance matrix, 81
exploration-exploitation dilemma, 255
extremely randomized forests, 217

f-score, 47
f1-score, 47
false negatives, 46
false positives, 46
feature extraction, 70
feature selection, 69
feature space, 28, 84, 165
feedforward neural network, 157
filters, 70
forward stagewise additive modeling, 223
frequentist approach, 27

generalization, 25
gradient descent, 167
Gram matrix, 81
greedy policy, 244, 249
growing grid, 147
growing neural gas, 147

hinge loss, 61, 101
Hoeffding's inequality, 57, 233
hypothesis space, 27, 51

i.i.d, 12
independent and identically distributed, 12
induction principle, 25, 31
inductive bias, 32
inductive learning, 25

joint variable, 35

k-fold cross-validation, 43
k-means, 146
k-nearest neighbours, 25
kernel, 85, 111
kernel trick, 85, 112, 115

Lagrange multipliers, 103
Lagrangian, 103
learning by heart, 31
Least Absolute Shrinkage and Selection Operator, 70
leave-one-out cross-validation, 43
line search, 226
linear functions, 27
linear separator, 95
linearly separable, 28, 95, 162
Lloyd iteration, 145
loss function, 29, 52

manifold learning, 87
margin, 96, 224
marginalization, 35
Markov Decision Process (MDP), 240
Markovian Decision Process, 23
masked Delaunay triangulation, 140
Mercer's theorem, 85, 114
mini-batch, 14
minimal enclosing sphere, 128
minimal norm solution, 167
mixture of experts, 214
Moore-Penrose pseudo-inverse, 167
multi-class classification, 20

n-grams, 68
natural gradient, 261
neural network, 155
nonparametric hypothesis space, 27
normal equations, 167

off-policy, 257
on-policy, 257
one-class SVM, 129
one-versus-all, 20
one-versus-one, 20
one-versus-rest, 20
online sampling, 14
optimal control, 21
optimal policy, 23
optimism in the face of uncertainty, 233
optimization problem, 102
optimization theory, 102
oracle, 12, 51
overfitting, 31, 43

p-spectrum kernel, 117
PAC (Probably Approximately Correct), 57, 234
parametric functions, 27
parametric hypothesis space, 27
perceptron convergence theorem, 163
perceptron learning rule, 160
policy, 23, 241
policy iteration, 246
policy search, 259
precision, 47
Principal Component Analysis, 73
prior, 37
prototypes, 133

quadratic loss, 30

random forests, 217
real risk, 30
recall, 47
recurrent neural network, 157
recurrent neural networks, 190
recursive neural network, 157
regression, 20
regression tree, 211
regret, 231
reinforcement learning, 21, 239
restricted functional gradient descent, 225
risk, 29, 53
ROC space, 47

sample covariance matrix, 75, 81
scalar linear functions, 27
score, 20, 63
second order Voronoï tessellation, 143
self-organizing map, 149
semi-supervised learning, 21
sensitivity, 47
Sequential Backward Search, 71
Sequential Floating Backward Search, 71
Sequential Floating Forward Search, 71
Sequential Forward Search, 71
shortest confidence interval, 138
Singular Value Decomposition, 167
slack variables, 101
SMO, 119
specificity, 47
stacked denoising autoencoder, 184

stacking, 214
standardization, 15
state-action value function, 248
statistical learning theory, 51
steepest descent, 167
Stochastic Gradient Descent, 167
strong learner, 221
Supervised learning, 19
support vectors, 99
surrogate, 60
SVC, 127
SVR, 127

target, 138
targeted vector quantization process, 138
test set, 43
time-delay neural network, 189
training set, 43
transductive learning, 25
true negatives, 46
true positives, 46
two-class classification, 19

UCB (Upper Confidence Bound) strategy, 236
uncertainty, 40
Unsupervised learning, 15

validation set, 44
value function, 241
value iteration, 244
vanishing gradient, 184
Vapnik-Chervonenkis dimension, 58
variable selection, 70
vector quantization, 133
Voronoï distortion, 135
Voronoï subsets, 135
Voronoï tessellation, 138, 140

weak learner, 220
weight sharing, 182
winner-take-all, 147
winner-take-most, 149
wrappers, 70