Mastering Machine Learning Algorithms

Expert techniques to implement popular machine learning algorithms and fine-tune your models

By Giuseppe Bonaccorso

Packt
www.packt.com

Mastering Machine Learning Algorithms

Expert techniques to implement popular machine learning algorithms and fine-tune your models

Giuseppe Bonaccorso

Packt

BIRMINGHAM - MUMBAI

Mastering Machine Learning Algorithms

Commissioning Editor: Pravin Dhandre
Acquisition Editor: Divya Poojari
Content Development Editor: Eisha Dsouza
Technical Editors: Jovita Alva, Ishita Vora
Copy Editor: Safis Editing
Project Coordinator: Shweta H Birwatkar
Proofreader: Safis Editing
Indexer: Priyanka Dhadke
Graphics: Jisha Chirayil
Production Coordinator: Aparna Bhagat

First published: May 2018

Production reference: 1240518

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

ISBN 978-1-78862-111-3

www.packtpub.com

To my parents, who always supported me in the journey of life!

– Giuseppe Bonaccorso

mapt.io

Mapt is an online digital library that gives you full access to over 5,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Mapt is fully searchable
- Copy and paste, print, and bookmark content

PacktPub.com

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Contributors

About the author

Giuseppe Bonaccorso is an experienced team leader/manager in AI, machine/deep learning solution design, management, and delivery. He got his M.Sc.Eng. in Electronics in 2005 from University of Catania, Italy, and continued his studies at University of Rome Tor Vergata and University of Essex, UK. His main interests include machine/deep learning, reinforcement learning, big data, bio-inspired adaptive systems, cryptocurrencies, and NLP.

I want to thank the people who have been close to me and have supported me, especially my parents, who never stopped encouraging me.

About the reviewer

Francesco Azzola is an electronics engineer with over 15 years of experience in computer programming. He is the author of *Android Things Projects* by Packt. He loves creating IoT projects using Arduino, Raspberry Pi, Android, and other IoT platforms. He is interested in convergence of IoT and mobile applications. He is certified in SCEA, SCWCD, and SCJP.

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit `authors.packtpub.com` and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Table of Contents

Preface

In the last few years, machine learning has become a more and more important field in the majority of industries. Many tasks once considered impossible to automate are now completely managed by computers, allowing human beings to focus on more creative tasks. This revolution has been made possible by the dramatic improvement of standard algorithms, together with a continuous reduction in hardware prices. The complexity that was a huge obstacle only a decade ago is now a problem than even a personal computer can solve. The general availability of high-level open source frameworks has allowed everybody to design and train extremely powerful models.

The main goal of this book is to introduce the reader to complex techniques (such as semi-supervised and manifold learning, probabilistic models, and neural networks), balancing mathematical theory with practical examples written in Python. I wanted to keep a pragmatic approach, focusing on the applications but not neglecting the necessary theoretical foundation. In my opinion, a good knowledge of this field can be acquired only by understanding the underlying logic, which is always expressed using mathematical concepts. This extra effort is rewarded with a more solid awareness of every specific choice and helps the reader understand how to apply, modify, and improve all the algorithms in specific business contexts.

Machine learning is an extremely wide field and it's impossible to cover all the topics in a book. In this case, I've done my best to cover a selection of algorithms belonging to supervised, semi-supervised, unsupervised, and Reinforcement Learning, providing all the references necessary to further explore each of them. The examples have been designed to be easy to understand without any deep insight into the code; in fact, I believe it's more important to show the general cases and let the reader improve and adapt them to cope with particular scenarios. I apologize for mistakes: even if many revisions have been made, it's possible that some details (both in the formulas and in the code) got away. I hope this book will be the starting point for many professionals struggling to enter this fascinating world with a pragmatic and business-oriented viewpoint!

Who this book is for

The ideal audience for this book is computer science students and professionals who want to acquire detailed knowledge of complex machine learning algorithms and applications. The approach is always pragmatic; however, the theoretical part requires some advanced mathematical skills that all graduates (in computer science, engineering, mathematics, or science) should have acquired. The book can be also utilized by more business-oriented professionals (such as CPOs and product managers) to understand how machine learning can be employed to improve existing products and businesses.

What this book covers

`Chapter 1`, *Machine Learning Model Fundamentals*, explains the most important theoretical concepts regarding machine learning models, including bias, variance, overfitting, underfitting, data normalization, and cost functions. It can be skipped by those readers with a strong knowledge of these concepts.

`Chapter 2`, *Introduction to Semi-Supervised Learning*, introduces the reader to the main elements of semi-supervised learning, focusing on inductive and transductive learning algorithms.

`Chapter 3`, *Graph-Based Semi-Supervised Learning*, continues the exploration of semi-supervised learning algorithms belonging to the families of graph-based and manifold learning models. Label propagation and non-linear dimensionality reduction are analyzed in different contexts, providing some effective solutions that can be immediately exploited using Scikit-Learn functionalities.

`Chapter 4`, *Bayesian Networks and Hidden Markov Models*, introduces the concepts of probabilistic modeling using direct acyclic graphs, Markov chains, and sequential processes.

`Chapter 5`, *EM Algorithm and Applications*, explains the generic structure of the Expectation-Maximization (EM) algorithm. We discuss some common applications, such as Gaussian mixture, Principal Component Analysis, Factor Analysis, and Independent Component Analysis. This chapter requires deep mathematical knowledge; however, the reader can skip the proofs and concentrate on the final results.

`Chapter 6`, *Hebbian Learning and Self-Organizing Maps*, introduces Hebb's rule, which is one of the oldest neuro-scientific concepts and whose applications are incredibly powerful. The chapter explains how a single neuron works and presents two complex models (Sanger network and Rubner-Tavan network) that can perform a Principal Component Analysis without the input covariance matrix.

`Chapter 7`, *Clustering Algorithms*, introduces some common and important unsupervised algorithms, such as k-Nearest Neighbors (based on KD Trees and Ball Trees), K-means (with K-means++ initialization), fuzzy C-means, and spectral clustering. Some important metrics (such as Silhouette score/plots) are also analyzed.

`Chapter 8`, *Ensemble Learning*, explains the main concepts of ensemble learning (bagging, boosting, and stacking), focusing on Random Forests, AdaBoost (with its variants), Gradient Boosting, and Voting Classifiers.

`Chapter 9`, *Neural Networks for Machine Learning*, introduces the concepts of neural computation, starting with the behavior of a perceptron and continuing the analysis of multi-layer perceptron, activation functions, back-propagation, stochastic gradient descent (and the most important optimization algorithm), regularization, dropout, and batch normalization.

`Chapter 10`, *Advanced Neural Models*, continues the explanation of the most important deep learning methods focusing on convolutional networks, recurrent networks, LSTM, and GRU.

`Chapter 11`, *Autoencoders*, explains the main concepts of an autoencoder, discussing its application in dimensionality reduction, denoising, and data generation (variational autoencoders).

`Chapter 12`, *Generative Adversarial Networks*, explains the concept of adversarial training. We focus on Deep Convolutional GANs and Wasserstein GANs. Both techniques are extremely powerful generative models that can learn the structure of an input data distribution and generate brand new samples without any additional information.

`Chapter 13`, *Deep Belief Networks*, introduces the concepts of Markov random fields, Restricted Boltzmann Machines, and Deep Belief Networks. These models can be employed both in supervised and unsupervised scenarios with excellent performance.

`Chapter 14`, *Introduction to Reinforcement Learning*, explains the main concepts of Reinforcement Learning (agent, policy, environment, reward, and value) and applies them to introduce policy and value iteration algorithms and Temporal-Difference Learning (TD(0)). The examples are based on a custom checkerboard environment.

`Chapter 15`, *Advanced Policy Estimation Algorithms*, extends the concepts defined in the previous chapter, discussing the TD(λ) algorithm, TD(0) Actor-Critic, SARSA, and Q-Learning. A basic example of Deep Q-Learning is also presented to allow the reader to immediately apply these concepts to more complex environments.

To get the most out of this book

There are no strict prerequisites for this book; however, it's important to have basic-intermediate Python knowledge with a specific focus on NumPy. Whenever necessary, I will provide instructions/references to install specific packages and exploit more advanced functionalities. As Python is based on a semantic indentation, the published version can contain incorrect newlines that raise exceptions when executing the code. For this reason, I invite all readers without deep knowledge of this language to refer to the original source code provided with the book.

All the examples are based on Python 3.5+. I suggest using the Anaconda distribution (`https://www.anaconda.com/download/`), which is probably the most complete and powerful one for scientific projects. The majority of the required packages are already built in and it's very easy to install the new ones (sometimes with optimized versions). However, any other Python distribution can be used. Moreover, I invite readers to test the examples using Jupyter (formerly known as IPython) notebooks so as to avoid rerunning the whole example when a change is made. If instead an IDE is preferred, I suggest PyCharm, which offers many built-in functionalities that are very helpful in data-oriented and scientific projects (such as the internal Matplotlib viewer).

A good mathematics background is necessary to fully understand the theoretical part. In particular, basic skills in probability theory, calculus, and linear algebra are required. However, I advise you not to give up when a concept seems too difficult. The reference sections contain many useful books, and the majority of concepts are explained quite well on Wikipedia too. When something unknown is encountered, I suggest reading the specific documentation before continuing. In many cases, it's not necessary to have complete knowledge and even an introductory paragraph can be enough to understand their rationale.

Download the example code files

You can download the example code files for this book from your account at `www.packtpub.com`. If you purchased this book elsewhere, you can visit `www.packtpub.com/support` and register to have the files emailed directly to you.

You can download the code files by following these steps:

1. Log in or register at `www.packtpub.com`.
2. Select the **SUPPORT** tab.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box and follow the onscreen instructions.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR/7-Zip for Windows
- Zipeg/iZip/UnRarX for Mac
- 7-Zip/PeaZip for Linux

The code bundle for the book is also hosted on GitHub at `https://github.com/PacktPublishing/Mastering-Machine-Learning-Algorithms`. In case there's an update to the code, it will be updated on the existing GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: `http://www.packtpub.com/sites/default/files/downloads/MasteringMachineLearningAlgorithms_ColorImages.pdf`.

Conventions used

There are a number of text conventions used throughout this book.

`CodeInText`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "In Scikit-Learn, it's possible to split the original dataset using the `train_test_split()` function."

A block of code is set as follows:

```
from sklearn.model_selection import train_test_split

X_train, X_test, Y_train, Y_test = train_test_split(X, Y, train_size=0.7,
random_state=1)
```

Bold: Indicates a new term, an important word, or words that you see onscreen.

Warnings or important notes appear like this.

Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: Email `feedback@packtpub.com` and mention the book title in the subject of your message. If you have questions about any aspect of this book, please email us at `questions@packtpub.com`.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit `www.packtpub.com/submit-errata`, selecting your book, clicking on the Errata Submission Form link, and entering the details.

Piracy: If you come across any illegal copies of our works in any form on the Internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packtpub.com` with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit `authors.packtpub.com`.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit `packtpub.com`.

1
Machine Learning Model Fundamentals

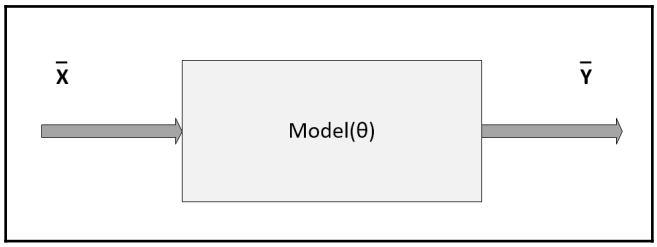
Machine learning models are mathematical systems that share many common features. Even if, sometimes, they have been defined only from a theoretical viewpoint, research advancement allows us to apply several concepts to better understand the behavior of complex systems such as deep neural networks. In this chapter, we're going to introduce and discuss some fundamental elements that some skilled readers may already know, but that, at the same time, offer several possible interpretations and applications.

In particular, in this chapter we're discussing the main elements of:

- Data-generating processes
- Finite datasets
- Training and test split strategies
- Cross-validation
- Capacity, bias, and variance of a model
- Vapnik-Chervonenkis theory
- Cramér-Rao bound
- Underfitting and overfitting
- Loss and cost functions
- Regularization

Models and data

Machine learning algorithms work with data. They create associations, find out relationships, discover patterns, generate new samples, and more, working with well-defined datasets. Unfortunately, sometimes the assumptions or the conditions imposed on them are not clear, and a lengthy training process can result in a complete validation failure. Even if this condition is stronger in deep learning contexts, we can think of a model as a gray box (some transparency is guaranteed by the simplicity of many common algorithms), where a vectorial input $\bar{X}$ is transformed into a vectorial output $\bar{Y}$:



Schema of a generic model parameterized with the vector θ

In the previous diagram, the model has been represented by a pseudo-function that depends on a set of parameters defined by the vector θ. In this section, we are only considering **parametric** models, although there's a family of algorithms that are called **non-parametric**, because they are based only on the structure of the data. We're going to discuss some of them in upcoming chapters.

The task of a parametric learning process is therefore to find the best parameter set that maximizes a target function whose value is proportional to the accuracy (or the error, if we are trying to minimize them) of the model given a specific input X and output Y. This definition is not very rigorous, and it will be improved in the following sections; however, it's useful as a way to understand the context we're working in.

Then, the first question to ask is: What is the nature of X? A machine learning problem is focused on learning abstract relationships that allow a consistent generalization when new samples are provided. More specifically, we can define a stochastic **data generating process** with an associated joint probability distribution:

$$p_{data}(\bar{x}, \bar{y}) = p(\bar{y}|\bar{x})p(\bar{x})$$

Sometimes, it's useful to express the joint probability *p(x, y)* as a product of the conditional *p(y|x)*, which expresses the probability of a label given a sample, and the marginal probability of the samples *p(x)*. This expression is particularly useful when the prior probability *p(x)* is known in semi-supervised contexts, or when we are interested in solving problems using the **Expectation Maximization** (**EM**) algorithm. We're going to discuss this approach in upcoming chapters.

In many cases, we are not able to derive a precise distribution; however, when considering a dataset, we always assume that it's drawn from the original data-generating distribution. This condition isn't a purely theoretical assumption, because, as we're going to see, whenever our data points are drawn from different distributions, the accuracy of the model can dramatically decrease.

If we sample N **independent and identically distributed** (**i.i.d.**) values from p_{data}, we can create a finite dataset *X* made up of *k*-dimensional real vectors:

$$X = \{\bar{x}_0, \bar{x}_1, \ldots, \bar{x}_N\} \ where \ \bar{x}_i \in \mathbb{R}^k$$

In a supervised scenario, we also need the corresponding labels (with *t* output values):

$$Y = \{\bar{y}_0, \bar{y}_1, \ldots, \bar{y}_N\} \ where \ \bar{y}_i \in \mathbb{R}^t$$

When the output has more than two classes, there are different possible strategies to manage the problem. In classical machine learning, one of the most common approaches is **One-vs-All**, which is based on training *N* different binary classifiers where each label is evaluated against all the remaining ones. In this way, *N-1* is performed to determine the right class. With shallow and deep neural networks, instead, it's preferable to use a **softmax** function to represent the output probability distribution for all classes:

$$\tilde{y}_i = \left(\frac{e^{z_0}}{\sum e^z}, \frac{e^{z_1}}{\sum e^z}, \ldots, \frac{e^{z_N}}{\sum e^z} \right)$$

This kind of output (z_i represents the intermediate values, and the sum of the terms is normalized to *1*) can be easily managed using the cross-entropy cost function (see the corresponding paragraph in the *Loss and cost functions* section).

Zero-centering and whitening

Many algorithms show better performances (above all, in terms of training speed) when the dataset is symmetric (with a zero-mean). Therefore, one of the most important preprocessing steps is so-called **zero-centering**, which consists in subtracting the feature-wise mean $E_x[X]$ from all samples:

$$\hat{x}_i = \bar{x}_i - E_x[X]$$

This operation, if necessary, is normally reversible, and doesn't alter relationships both among samples and among components of the same sample. In deep learning scenarios, a zero-centered dataset allows exploiting the symmetry of some activation function, driving to a faster convergence (we're going to discuss these details in the next chapters).

Another very important preprocessing step is called **whitening**, which is the operation of imposing an identity covariance matrix to a zero-centered dataset:

$$E_x[X^T X] = I$$

As the covariance matrix $E_x[X^T X]$ is real and symmetric, it's possible to eigendecompose it without the need to invert the eigenvector matrix:

$$E_x[X^T X] = V \Omega V^T$$

The matrix V contains the eigenvectors (as columns), and the diagonal matrix Ω contains the eigenvalues. To solve the problem, we need to find a matrix A, such that:

$$\hat{x}_i = A\bar{x}_i \;\; and \;\; E_x[\hat{X}^T \hat{X}] = I$$

Using the eigendecomposition previously computed, we get:

$$E_x[\hat{X}^T \hat{X}] = E_x[AX^T X A^T] = AE_x[X^T X]A^T = AV\Omega V^T A^T = I$$

Hence, the matrix A is:

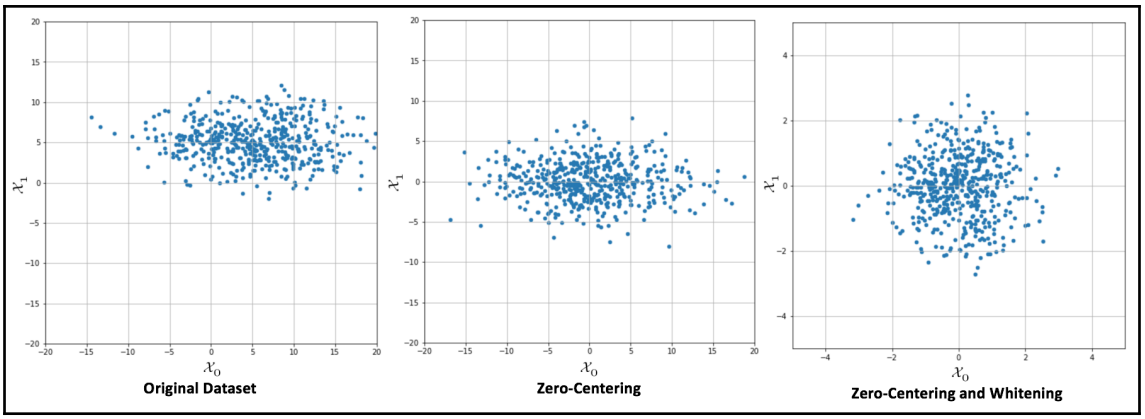
$$AA^T = V\Omega^{-1}V^T \;\Rightarrow\; A = V\Omega^{-\frac{1}{2}}$$

One of the main advantages of whitening is the decorrelation of the dataset, which allows an easier separation of the components. Furthermore, if X is whitened, any orthogonal transformation induced by the matrix P is also whitened:

$$Y = PX \;\Rightarrow\; E[Y^T Y] = PE[X^T X]P^T = PP^T = I$$

Moreover, many algorithms that need to estimate parameters that are strictly related to the input covariance matrix can benefit from this condition, because it reduces the actual number of independent variables (in general, these algorithms work with matrices that become symmetric after applying the whitening). Another important advantage in the field of deep learning is that the gradients are often higher around the origin, and decrease in those areas where the activation functions (for example, the hyperbolic tangent or the sigmoid) saturate ($|x| \rightarrow \infty$). That's why the convergence is generally faster for whitened (and zero-centered) datasets.

In the following graph, it's possible to compare an **original dataset**, **zero-centering**, and **whitening**:



Original dataset (left), centered version (center), whitened version (right)

When a whitening is needed, it's important to consider some important details. The first one is that there's a scale difference between the real sample covariance and the estimation $X^T X$, often adopted with the **singular value decomposition** (**SVD**). The second one concerns some common classes implemented by many frameworks, like Scikit-Learn's `StandardScaler`. In fact, while zero-centering is a feature-wise operation, a whitening filter needs to be computed considering the whole covariance matrix (`StandardScaler` implements only unit variance, feature-wise scaling).

Luckily, all Scikit-Learn algorithms that benefit from or need a whitening preprocessing step provide a built-in feature, so no further actions are normally required; however, for all readers who want to implement some algorithms directly, I've written two Python functions that can be used both for zero-centering and whitening. They assume a matrix X with a shape ($N_{Samples} \times n$). Moreover, the `whiten()` function accepts the parameter `correct`, which allows us to apply the scaling correction (the default value is `True`):

```
import numpy as np

def zero_center(X):
    return X - np.mean(X, axis=0)

def whiten(X, correct=True):
    Xc = zero_center(X)
    _, L, V = np.linalg.svd(Xc)
    W = np.dot(V.T, np.diag(1.0 / L))
    return np.dot(Xc, W) * np.sqrt(X.shape[0]) if correct else 1.0
```

Training and validation sets

In real problems, the number of samples is limited, and it's usually necessary to split the initial set X (together with Y) into two subsets as follows:

- **Training set** used to train the model
- **Validation set** used to assess the score of the model without any bias, with samples never seen before

According to the nature of the problem, it's possible to choose a split percentage ratio of 70% – 30% (a good practice in machine learning, where the datasets are relatively small), or a higher training percentage (80%, 90%, up to 99%) for deep learning tasks where the number of samples is very high. In both cases, we are assuming that the training set contains all the information required for a consistent generalization. In many simple cases, this is true and can be easily verified; but with more complex datasets, the problem becomes harder. Even if we think to draw all the samples from the same distribution, it can happen that a randomly selected test set contains features that are not present in other training samples. Such a condition can have a very negative impact on global accuracy and, without other methods, it can also be very difficult to identify. This is one of the reasons why, in deep learning, training sets are huge: considering the complexity of the features and structure of the data generating distributions, choosing large test sets can limit the possibility of learning particular associations.

In Scikit-Learn, it's possible to split the original dataset using the `train_test_split()` function, which allows specifying the train/test size, and if we expect to have randomly shuffled sets (default). For example, if we want to split `X` and `Y`, with 70% training and 30% test, we can use:

```
from sklearn.model_selection import train_test_split

X_train, X_test, Y_train, Y_test = train_test_split(X, Y, train_size=0.7,
random_state=1)
```

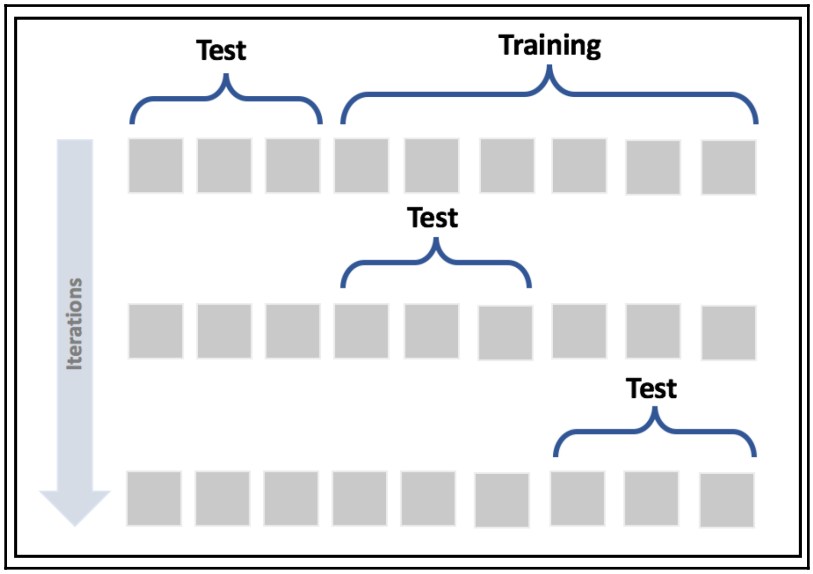
Shuffling the sets is always a good practice, in order to reduce the correlation between samples. In fact, we have assumed that `X` is made up of i.i.d samples, but several times two subsequent samples have a strong correlation, reducing the training performance. In some cases, it's also useful to re-shuffle the training set after each training epoch; however, in the majority of our examples, we are going to work with the same shuffled dataset throughout the whole process. Shuffling has to be avoided when working with sequences and models with memory: in all those cases, we need to exploit the existing correlation to determine how the future samples are distributed.

When working with NumPy and Scikit-Learn, it's always a good practice to set the random seed to a constant value, so as to allow other people to reproduce the experiment with the same initial conditions. This can be achieved by calling `np.random.seed(...)` and using the `random-state` parameter present in many Scikit-Learn methods.

Cross-validation

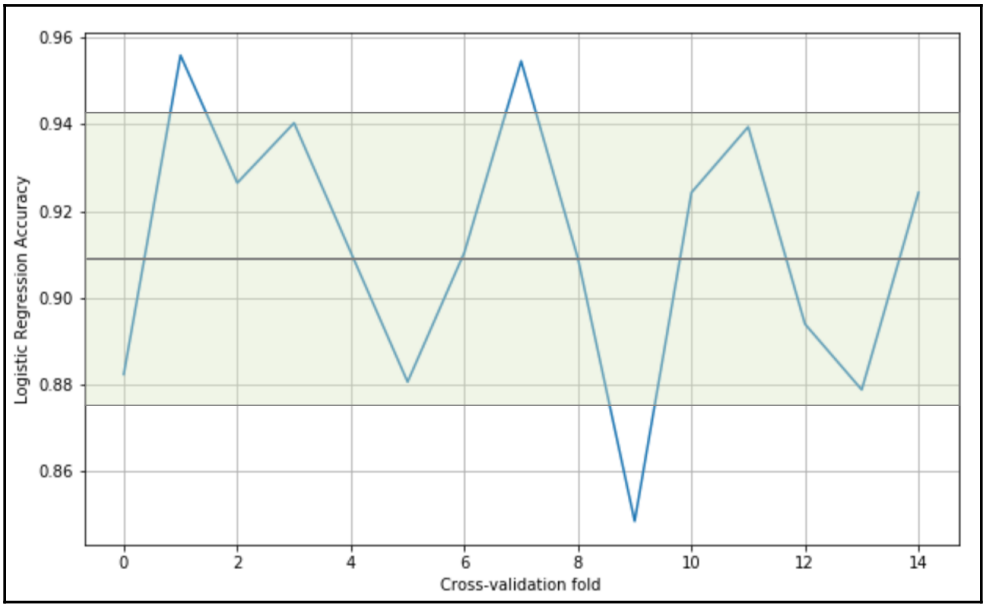
A valid method to detect the problem of wrongly selected test sets is provided by the **cross-validation** technique. In particular, we're going to use the **K-Fold** cross-validation approach. The idea is to split the whole dataset *X* into a moving test set and a training set (the remaining part). The size of the test set is determined by the number of folds so that, during *k* iterations, the test set covers the whole original dataset.

In the following diagram, we see a schematic representation of the process:



K-Fold cross-validation schema

In this way, it's possible to assess the accuracy of the model using different sampling splits, and the training process can be performed on larger datasets; in particular, on *(k-1)*N* samples. In an ideal scenario, the accuracy should be very similar in all iterations; but in most real cases, the accuracy is quite below average. This means that the training set has been built excluding samples that contain features necessary to let the model fit the separating hypersurface considering the real p_{data}. We're going to discuss these problems later in this chapter; however, if the standard deviation of the accuracies is too high (a threshold must be set according to the nature of the problem/model), that probably means that X hasn't been drawn uniformly from p_{data}, and it's useful to evaluate the impact of the outliers in a preprocessing stage. In the following graph, we see the plot of a 15-fold cross-validation performed on a logistic regression:



Cross-validation accuracies

The values oscillate from 0.84 to 0.95, with an average (solid horizontal line) of 0.91. In this particular case, considering the initial purpose was to use a linear classifier, we can say that all folds yield high accuracies, confirming that the dataset is linearly separable; however, there are some samples (excluded in the ninth fold) that are necessary to achieve a minimum accuracy of about 0.88.

K-Fold cross-validation has different variants that can be employed to solve specific problems:

- **Stratified K-Fold**: A **Standard K-Fold** approach splits the dataset without considering the probability distribution $p(y|x)$, therefore some folds may theoretically contain only a limited number of labels. Stratified K-Fold, instead, tries to split X so that all the labels are equally represented.

- **Leave-one-out** (**LOO**): This approach is the most drastic because it creates *N* folds, each of them containing *N-1* training samples and only 1 test sample. In this way, the maximum possible number of samples is used for training, and it's quite easy to detect whether the algorithm is able to learn with sufficient accuracy, or if it's better to adopt another strategy. The main drawback of this method is that *N* models must be trained, and when *N* is very large this can cause a performance issue. Moreover, with a large number of samples, the probability that two random values are similar increases, therefore many folds will yield almost identical results. At the same time, LOO limits the possibilities for assessing the generalization ability, because a single test sample is not enough for a reasonable estimation.
- **Leave-P-out** (**LPO**): In this case, the number of test samples is set to *p* (non-disjoint sets), so the number of folds is equal to the binomial coefficient of *n* over *p*. This approach mitigates LOO's drawbacks, and it's a trade-off between K-Fold and LOO. The number of folds can be very high, but it's possible to control it by adjusting the number *p* of test samples; however, if *p* isn't small or big enough, the binomial coefficient can *explode*. In fact, when *p* has about *n/2* samples, the number of folds is maximal:

$$\binom{n}{p} = \frac{n!}{p!(n-p)!} \approx \prod_{t=1}^{p} \frac{n-t}{t} \;\; if \;\; p \approx \frac{n}{2} \;\; and \;\; n \gg 1$$

Scikit-Learn implements all those methods (with some other variations), but I suggest always using the `cross_val_score()` function, which is a helper that allows applying the different methods to a specific problem. In the following snippet based on a polynomial **Support Vector Machine** (**SVM**) and the MNIST digits dataset, the function is applied specifying the number of folds (parameter `cv`). In this way, Scikit-Learn will automatically use Stratified K-Fold for categorical classifications, and **Standard K-Fold** for all other cases:

```
from sklearn.datasets import load_digits
from sklearn.model_selection import cross_val_score
from sklearn.svm import SVC

data = load_digits()
svm = SVC(kernel='poly')

skf_scores = cross_val_score(svm, data['data'], data['target'], cv=10)

print(skf_scores)
[ 0.96216216  1.          0.93922652  0.99444444  0.98882682  0.98882682
  0.99441341  0.99438202  0.96045198  0.96590909]
```

```
print(skf_scores.mean())
0.978864325583
```

The accuracy is very high (> 0.9) in every fold, therefore we expect to have even higher accuracy using the LOO method. As we have 1,797 samples, we expect the same number of accuracies:

```
from sklearn.model_selection import cross_val_score, LeaveOneOut

loo_scores = cross_val_score(svm, data['data'], data['target'],
cv=LeaveOneOut())

print(loo_scores[0:100])
[ 1.  1.  1.  1.  1.  0.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.
  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.
  1.  0.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.
  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  0.  1.
  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.
  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.]

print(loo_scores.mean())
0.988870339455
```

As expected, the average score is very high, but there are still samples that are misclassified. As we're going to discuss, this situation could be a potential candidate for overfitting, meaning that the model is learning perfectly how to map the training set, but it's losing its ability to generalize; however, LOO is not a good method to measure this model ability, due to the size of the validation set.

We can now evaluate our algorithm with the LPO technique. Considering what was explained before, we have selected the smaller Iris dataset and a classification based on a logistic regression. As there are *N=150* samples, choosing `p = 3`, we get 551,300 folds:

```
from sklearn.datasets import load_iris
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import cross_val_score, LeavePOut

data = load_iris()

p = 3
lr = LogisticRegression()

lpo_scores = cross_val_score(lr, data['data'], data['target'],
cv=LeavePOut(p))

print(lpo_scores[0:100])
```

```
[ 1.          1.          1.          1.          1.          1.
1.
  1.          1.          1.          1.          1.          1.
1.
  1.          1.          1.          1.          1.          1.
1.
  1.          1.          1.          1.          1.          1.
1.
  1.          1.          1.          1.          1.          1.
1.
  1.          1.          1.          1.          1.          1.
1.
  1.          1.          1.          1.          1.          1.
1.
  1.          1.          1.          1.          1.          1.
1.
  1.          0.66666667  ...

print(lpo_scores.mean())
0.955668420098
```

As in the previous example, we have printed only the first 100 accuracies; however, the global trend can be immediately understood with only a few values.

The cross-validation technique is a powerful tool that is particularly useful when the performance cost is not too high. Unfortunately, it's not the best choice for deep learning models, where the datasets are very large and the training processes can take even days to complete. However, as we're going to discuss, in those cases the right choice (the split percentage), together with an accurate analysis of the datasets and the employment of techniques such as normalization and regularization, allows fitting models that show an excellent generalization ability.

Features of a machine learning model

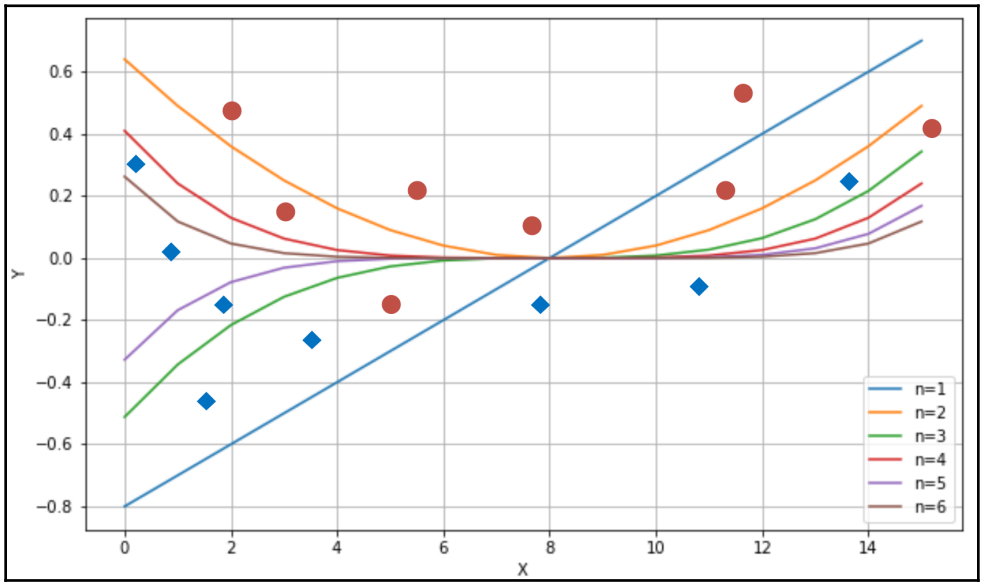
In this section, we're going to consider supervised models, and try to determine how it's possible to measure their theoretical potential accuracy and their ability to generalize correctly over all possible samples drawn from p_{data}. The majority of these concepts were developed before the *deep learning age*, but continue to have an enormous influence on research projects. The idea of *capacity*, for example, is an open-ended question that neuroscientists keep on asking themselves about the human brain. Modern deep learning models with dozens of layers and millions of parameters reopened the theoretical question from a mathematical viewpoint. Together with this, other elements, like the limits for the variance of an estimator, again attracted the limelight because the algorithms are becoming more and more powerful, and performances that once were considered far from any feasible solution are now a reality. Being able to train a model, so as to exploit its full capacity, maximize its generalization ability, and increase the accuracy, overcoming even human performances, is what a deep learning engineer nowadays has to expect from his work.

Capacity of a model

If we consider a supervised model as a set of parameterized functions, we can define **representational capacity** as the intrinsic ability of a certain generic function to map a relatively large number of data distributions. To understand this concept, let's consider a function *f(x)* that admits infinite derivatives, and rewrite it as a Taylor expansion:

$$f(x) = f(x_0) + \frac{f'(x_0)}{1!}(x - x_0) + \frac{f''(x_0)}{2!}(x - x_0)^2 + \ldots = \sum_{n=0}^{\infty} \frac{f^{(n)}(x_0)}{n!}(x - x_0)^n$$

We can decide to take only the first *n* terms, so to have an *n*-degree polynomial function. Consider a simple bi-dimensional scenario with six functions (starting from a linear one); we can observe the different behavior with a small set of data points:



Different behavior produced by six polynomial separating curves

The ability to rapidly change the curvature is proportional to the degree. If we choose a linear classifier, we can only modify its slope (the example is always in a bi-dimensional space) and the intercept. Instead, if we pick a higher-degree function, we have more possibilities to *bend* the curvature when it's necessary. If we consider **n=1** and **n=2** in the plot (on the top-right, they are the first and the second functions), with **n=1**, we can include the dot corresponding to *x=11*, but this choice has a negative impact on the dot at *x=5*.

Only a parameterized non-linear function can solve this problem efficiently, because this simple problem requires a representational capacity higher than the one provided by linear classifiers. Another classical example is the XOR function. For a long time, several researchers opposed perceptrons (linear neural networks), because they weren't able to classify a dataset generated by the XOR function. Fortunately, the introduction of multilayer perceptrons, with non-linear functions, allowed us to overcome this problem, and many whose complexity is beyond the possibilities of any classic machine learning model.

Vapnik-Chervonenkis capacity

A mathematical formalization of the capacity of a classifier is provided by the **Vapnik-Chervonenkis theory**. To introduce the definition, it's first necessary to define the concept of **shattering**. If we have a class of sets *C* and a set *M*, we say that *C* shatters *M* if:

$$\forall m_i \subseteq M \; \exists \; c_j \in C \Rightarrow m_j = c_i \cap M$$

In other words, given any subset of *M*, it can be obtained as the intersection of a particular instance of *C* (c_j) and *M* itself. If we now consider a model as a parameterized function:

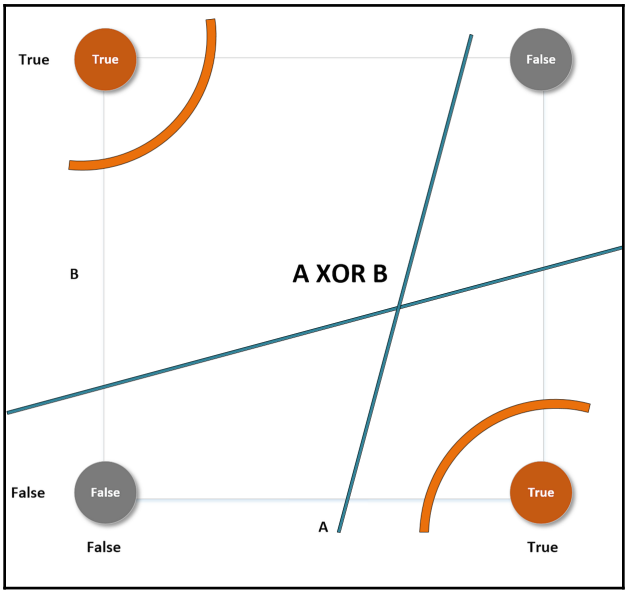
$$C = f(\bar{\theta}) \; where \; \bar{\theta} \in \mathbb{R}^p$$

We want to determine its capacity in relation to a finite dataset *X*:

$$X = \{\bar{x}_0, \bar{x}_1, \ldots, \bar{x}_N\} \; where \; \bar{x}_i \in \mathbb{R}^k$$

According to the Vapnik-Chervonenkis theory, we can say that the model *f* shatters *X* if there are no classification errors for every possible label assignment. Therefore, we can define the **Vapnik-Chervonenkis-capacity** or **VC-capacity** (sometimes called **VC-dimension**) as the maximum cardinality of a subset of *X* so that *f* can shatter it.

For example, if we consider a linear classifier in a bi-dimensional space, the VC-capacity is equal to 3, because it's always possible to label three samples so that *f* shatters them; however, it's impossible to do it in all situations where $N > 3$. The XOR problem is an example that needs a VC-capacity higher than 3. Let's explore the following plot:



XOR problem with different separating curves

This particular label choice makes the set non-linearly separable. The only way to overcome this problem is to use higher-order functions (or non-linear ones). The curve lines (belonging to a classifier whose VC-capacity is greater than 3) can separate both the upper-left and the lower-right regions from the remaining space, but no straight line can do the same (while it can always separate one point from the other three).

Bias of an estimator

Let's now consider a parameterized model with a single vectorial parameter (this isn't a limitation, but only a didactic choice):

$$p(X;\bar{\theta})$$

The goal of a learning process is to estimate the parameter θ so as, for example, to maximize the accuracy of a classification. We define the **bias of an estimator** (in relation to a parameter θ):

$$Bias\left[\tilde{\theta}\right] = E_{x|\bar{\theta}}\left[\tilde{\theta}\right] - \bar{\theta} = \left(\sum_{x} \tilde{\theta} p(x|\bar{\theta})\right) - \bar{\theta}$$

In other words, the bias is the difference between the expected value of the estimation and the real parameter value. Remember that the estimation is a function of *X*, and cannot be considered a constant in the sum.

An estimator is said to be **unbiased** if:

$$Bias\left[\tilde{\theta}\right] = 0 \Rightarrow E\left[\tilde{\theta}\right] = \bar{\theta}$$

Moreover, the estimator is defined as **consistent** if the sequence of estimations converges (at least with probability 1) to the real value when $k \rightarrow \infty$:

$$\forall \epsilon > 0 \quad P(|\bar{\theta} - \tilde{\theta}_k| < \epsilon) \rightarrow 1 \;\; when \;\; k \rightarrow \infty$$

Given a dataset *X* whose samples are drawn from p_{data}, the accuracy of an estimator is inversely proportional to its bias. Low-bias (or unbiased) estimators are able to map the dataset *X* with high-precision levels, while high-bias estimators are very likely to have a capacity that isn't high enough for the problem to solve, and therefore their ability to detect the whole dynamic is poor.

Let's now compute the derivative of the bias with respect to the vector θ (it will be useful later):

$$\frac{\partial Bias\left[\tilde{\theta}\right]}{\partial \bar{\theta}} = \frac{\partial}{\partial \bar{\theta}}\left(\left(\sum_{x} \tilde{\theta} p(x|\bar{\theta})\right) - \bar{\theta}\right) = \left(\sum_{x} \tilde{\theta} \frac{\partial p(x|\bar{\theta})}{\partial \bar{\theta}}\right) - 1 =$$

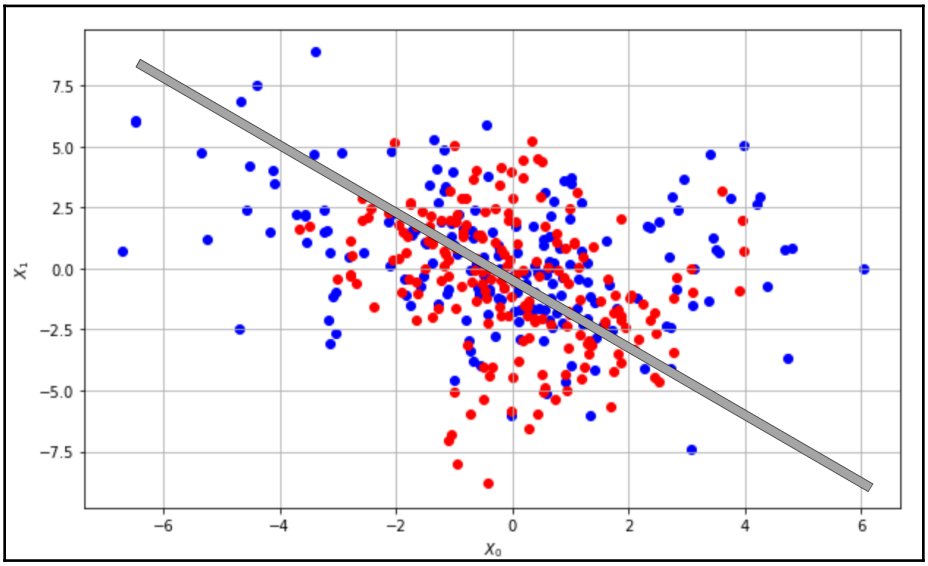
$$= \left(\sum_{x} \tilde{\theta} p(x|\bar{\theta}) \frac{\partial \, log \, p(x|\bar{\theta})}{\partial \bar{\theta}}\right) - 1 = E_{x|\bar{\theta}}\left[\tilde{\theta} \frac{\partial \, log \, p(x|\bar{\theta})}{\partial \bar{\theta}}\right] - 1$$

Consider that the last equation, thanks to the linearity of *E[•]*, holds also if we add a term that doesn't depend on x to the estimation of θ. In fact, in line with the laws of probability, it's easy to verify that:

$$\sum_{x}(\tilde{\theta}+a)\,p(x|\bar{\theta}) = \sum_{x}\tilde{\theta}p(x|\bar{\theta}) + a\sum_{x}p(x|\bar{\theta}) = \sum_{x}\tilde{\theta}p(x|\bar{\theta})$$

Underfitting

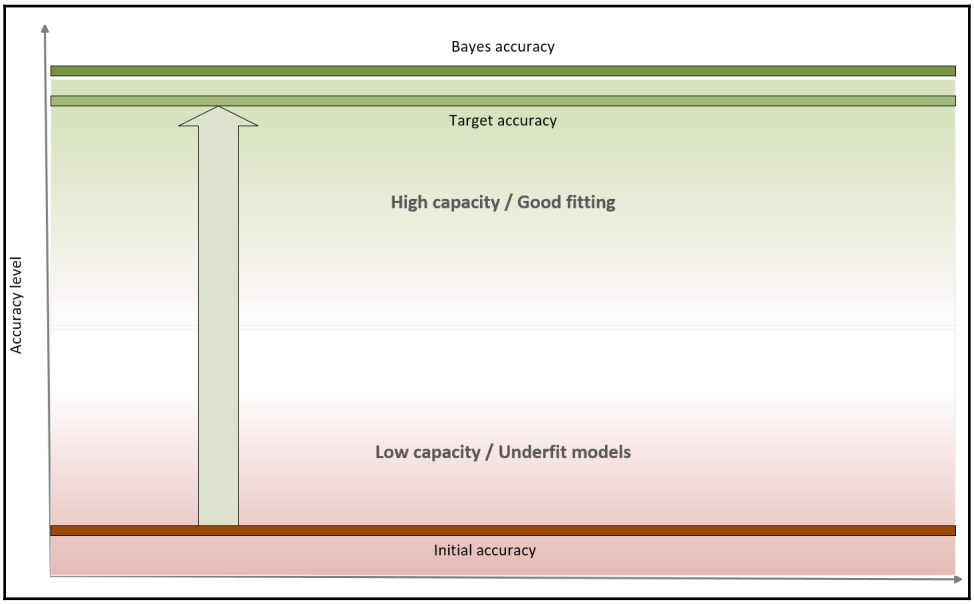
A model with a high bias is likely to underfit the training set. Let's consider the scenario shown in the following graph:



Underfitted classifier: The curve cannot separate correctly the two classes

Even if the problem is very hard, we could try to adopt a linear model and, at the end of the training process, the slope and the intercept of the separating line are about -1 and 0 (as shown in the plot); however, if we measure the accuracy, we discover that it's close to 0! Independently from the number of iterations, this model will never be able to learn the association between X and Y. This condition is called **underfitting**, and its major indicator is a very low training accuracy. Even if some data preprocessing steps can improve the accuracy, when a model is underfitted, the only valid solution is to adopt a higher-capacity model.

In a machine learning task, our goal is to achieve the maximum accuracy, starting from the training set and then moving on to the validation set. More formally, we can say that we want to improve our models so to get as close as possible to **Bayes accuracy**. This is not a well-defined value, but a theoretical upper limit that is possible to achieve using an estimator. In the following diagram, we see a representation of this process:



Accuracy level diagram

Bayes accuracy is often a purely theoretical limit and, for many tasks, it's almost impossible to achieve using even biological systems; however, advancements in the field of deep learning allow to create models that have a target accuracy slightly below the Bayes one. In general, there's no closed form for determining the Bayes accuracy, therefore human abilities are considered as a benchmark. In the previous classification example, a human being is immediately able to distinguish among different dot classes, but the problem can be very hard for a limited-capacity classifier. Some of the models we're going to discuss can solve this problem with a very high target accuracy, but at this point, we run another risk that can be understood after defining the concept of variance of an estimator.

Variance of an estimator

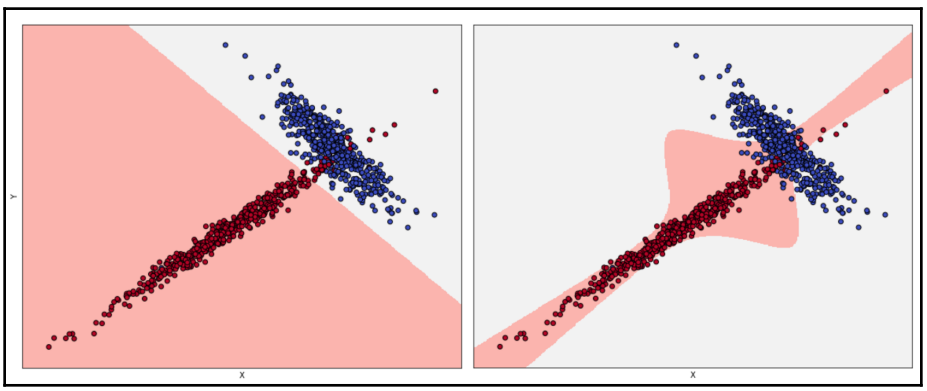
At the beginning of this chapter, we have defined the data generating process p_{data}, and we have assumed that our dataset X has been drawn from this distribution; however, we don't want to learn existing relationships limited to X, but we expect our model to be able to generalize correctly to any other subset drawn from p_{data}. A good measure of this ability is provided by the **variance of the estimator**:

$$Var\left[\tilde{\theta}\right] = StdErr\left[\tilde{\theta}\right]^2 = E[\left(\tilde{\theta} - E[\tilde{\theta}]\right)^2]$$

The variance can be also defined as the square of the standard error (analogously to the standard deviation). A high variance implies dramatic changes in the accuracy when new subsets are selected, because the model has probably reached a very high training accuracy through an over-learning of a limited set of relationships, and it has almost completely lost its ability to generalize.

Overfitting

If underfitting was the consequence of a low capacity and a high bias, **overfitting** is a phenomenon that a high variance can detect. In general, we can observe a very high training accuracy (even close to the Bayes level), but not a poor validation accuracy. This means that the capacity of the model is high enough or even excessive for the task (the higher the capacity, the higher the probability of large variances), and that the training set isn't a good representation of p_{data}. To understand the problem, consider the following classification scenarios:



Acceptable fitting (left), overfitted classifier (right)

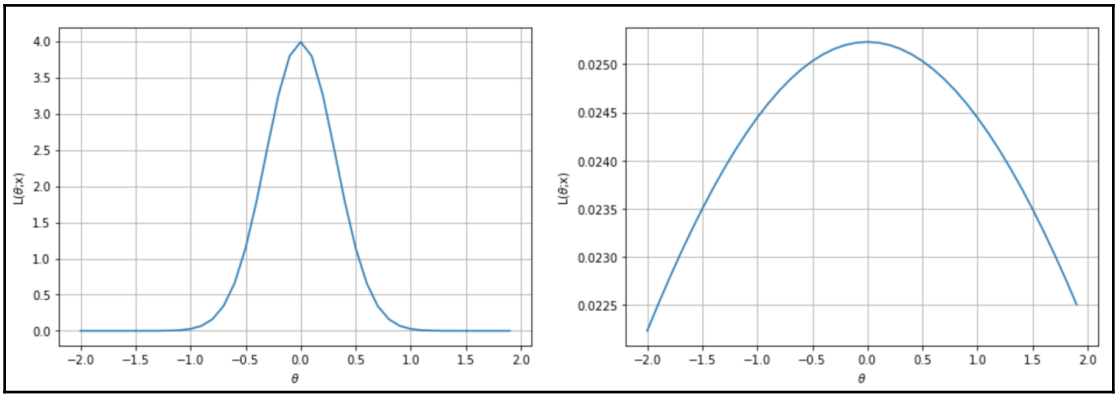
The left plot has been obtained using logistic regression, while, for the right one, the algorithm is SVM with a sixth-degree polynomial kernel. If we consider the second model, the decision boundaries seem much more precise, with some samples just over them. Considering the shapes of the two subsets, it would be possible to say that a non-linear SVM can better capture the dynamics; however, if we sample another dataset from p_{data} and the diagonal *tail* becomes wider, logistic regression continues to classify the points correctly, while the SVM accuracy decreases dramatically. The second model is very likely to be overfitted, and some corrections are necessary. When the validation accuracy is much lower than the training one, a good strategy is to increase the number of training samples to consider the real p_{data}. In fact, it can happen that a training set is built starting from a hypothetical distribution that doesn't reflect the real one; or the number of samples used for the validation is too high, reducing the amount of information carried by the remaining samples. Cross-validation is a good way to assess the quality of datasets, but it can always happen that we find completely new subsets (for example, generated when the application is deployed in a production environment) that are misclassified, even if they were supposed to belong to p_{data}. If it's not possible to enlarge the training set, data augmentation could be a valid solution, because it allows creating artificial samples (for images, it's possible to mirror, rotate, or blur them) starting from the information stored in the known ones. Other strategies to prevent overfitting are based on a technique called **regularization**, which we're going to discuss in the last part of this chapter. For now, we can say that the effect of regularization is similar to a partial linearization, which implies a capacity reduction with a consequent variance decrease.

The Cramér-Rao bound

If it's theoretically possible to create an unbiased model (even asymptotically), this is not true for variance. To understand this concept, it's necessary to introduce an important definition: the **Fisher information**. If we have a parameterized model and a data-generating process p_{data}, we can define the likelihood function by considering the following parameters:

$$L(\bar{\theta}|X) = p(X|\bar{\theta})$$

This function allows measuring how well the model describes the original data generating process. The shape of the likelihood can vary substantially, from well-defined, peaked curves, to almost flat surfaces. Let's consider the following graph, showing two examples based on a single parameter:



Very peaked likelihood (left), flatter likelihood (right)

We can immediately understand that, in the first case, the maximum likelihood can be easily reached by gradient ascent, because the surface is very peaked. In the second case, instead, the gradient magnitude is smaller, and it's rather easy to stop before reaching the actual maximum because of numerical imprecisions or tolerances. In worst cases, the surface can be almost flat in very large regions, with a corresponding gradient close to zero. Of course, we'd like to always work with very sharp and peaked likelihood functions, because they carry more information about their maximum. More formally, the Fisher information quantifies this value. For a single parameter, it is defined as follows:

$$I(\theta) = E_{\bar{x}|\theta}\left[\left(\frac{\partial}{\partial\theta} log\, p(\bar{x}|\theta)\right)^2\right]$$

The Fisher information is an unbounded non-negative number that is proportional to the amount of information carried by the log-likelihood; the use of logarithm has no impact on the gradient ascent, but it simplifies complex expressions by turning products into sums. This value can be interpreted as the *speed* of the gradient when the function is reaching the maximum; therefore, higher values imply better approximations, while a hypothetical value of zero means that the probability to determine the right parameter estimation is also null.

When working with a set of *K* parameters, the Fisher information becomes a positive semidefinite matrix:

$$I(\bar{\theta}) = \begin{pmatrix} E_{\bar{x}|\bar{\theta}}\left[\left(\frac{\partial}{\partial\theta_0} log\, p(\bar{x}|\bar{\theta})\right)\left(\frac{\partial}{\partial\theta_0} log\, p(\bar{x}|\bar{\theta})\right)\right] & \cdots & E_{\bar{x}|\bar{\theta}}\left[\left(\frac{\partial}{\partial\theta_0} log\, p(\bar{x}|\bar{\theta})\right)\left(\frac{\partial}{\partial\theta_K} log\, p(\bar{x}|\bar{\theta})\right)\right] \\ \vdots & \ddots & \vdots \\ E_{\bar{x}|\bar{\theta}}\left[\left(\frac{\partial}{\partial\theta_K} log\, p(\bar{x}|\bar{\theta})\right)\left(\frac{\partial}{\partial\theta_0} log\, p(\bar{x}|\bar{\theta})\right)\right] & \cdots & E_{\bar{x}|\bar{\theta}}\left[\left(\frac{\partial}{\partial\theta_K} log\, p(\bar{x}|\bar{\theta})\right)\left(\frac{\partial}{\partial\theta_K} log\, p(\bar{x}|\bar{\theta})\right)\right] \end{pmatrix}$$

This matrix is symmetric, and also has another important property: when a value is zero, it means that the corresponding couple of parameters are orthogonal for the purpose of the maximum likelihood estimation, and they can be considered separately. In many real cases, if a value is close to zero, it determines a very low correlation between parameters and, even if it's not mathematically rigorous, it's possible to decouple them anyway.

At this point, it's possible to introduce the **Cramér-Rao bound**, which states that for every unbiased estimator that adopts *x* (with probability distribution *p(x; θ)*) as a measure set, the variance of any estimator of θ is always lower-bounded according to the following inequality:

$$Var\left[\tilde{\theta}\right] \geqslant \frac{1}{I\left(\theta\right)}$$

In fact, considering initially a generic estimator and exploiting Cauchy-Schwarz inequality with the variance and the Fisher information (which are both expressed as expected values), we obtain:

$$E_{\bar{x}|\theta}\left[(\tilde{\theta} - E_{\bar{x}|\theta}[\tilde{\theta}])^2\right] E_{\bar{x}|\theta}\left[\left(\frac{\partial\, log\, p(\bar{x}|\theta)}{\partial\theta}\right)^2\right] \geqslant E_{\bar{x}|\theta}\left[(\tilde{\theta} - E_{\bar{x}|\theta}[\tilde{\theta}])\frac{\partial\, log\, p(\bar{x}|\theta)}{\partial\theta}\right]^2$$

Now, if we use the expression for derivatives of the bias with respect to θ, considering that the expected value of the estimation of θ doesn't depend on *x*, we can rewrite the right side of the inequality as:

$$E_{x|\theta}\left[(\tilde{\theta} - E_{x|\theta}[\tilde{\theta}])\frac{\partial\, log\, p(\bar{x}|\theta)}{\partial\theta}\right]^2 = \left(\frac{\partial Bias\left[\tilde{\theta}\right]}{\partial\theta} + 1\right)^2$$

If the estimator is unbiased, the derivative on the right side is equal to zero, therefore, we get:

$$Var\left[\tilde{\theta}\right] \cdot I(\theta) \geqslant 1$$

In other words, we can try to reduce the variance, but it will be always lower-bounded by the inverse Fisher information. Therefore, given a dataset and a model, there's always a limit to the ability to generalize. In some cases, this measure is easy to determine; however, its real value is theoretical, because it provides the likelihood function with another fundamental property: it carries all the information needed to estimate the worst case for variance. This is not surprising: when we discussed the capacity of a model, we saw how different functions could drive to higher or lower accuracies. If the training accuracy is high enough, this means that the capacity is appropriate or even excessive for the problem; however, we haven't considered the role of the likelihood $p(X|\ \theta)$.

High-capacity models, in particular, with small or low-informative datasets, can drive to flat likelihood surfaces with a higher probability than lower-capacity models. Therefore, the Fisher information tends to become smaller, because there are more and more parameter sets that yield similar probabilities, and this, at the end of the day, drives to higher variances and an increased risk of overfitting. To conclude this section, it's useful to consider a general empirical rule derived from the **Occam's razor** principle: whenever a simpler model can explain a phenomenon with enough accuracy, it doesn't make sense to increase its capacity. A simpler model is always preferable (when the performance is good and it represents accurately the specific problem), because it's normally faster both in the training and in the inference phases, and more efficient. When talking about deep neural networks, this principle can be applied in a more precise way, because it's easier to increase or decrease the number of layers and neurons until the desired accuracy has been achieved.

Loss and cost functions

At the beginning of this chapter, we discussed the concept of generic target function so as to optimize in order to solve a machine learning problem. More formally, in a supervised scenario, where we have finite datasets X and Y:

$$X = \{\bar{x}_0, \bar{x}_1, \ldots, \bar{x}_N\}\ \ where\ \bar{x}_i \in \mathbb{R}^k$$

$$Y = \{\bar{y}_0, \bar{y}_1, \ldots, \bar{y}_N\}\ \ where\ \bar{y}_i \in \mathbb{R}^t$$

We can define the generic **loss function** for a single sample as:

$$J\left(\bar{x}_i, \bar{y}_i; \bar{\theta}\right) = J\left(f\left(\bar{x}_i, \bar{\theta}\right), \bar{y}_i\right) = J\left(\bar{y}_i, \bar{y}_i\right)$$

J is a function of the whole parameter set, and must be proportional to the error between the true label and the predicted. Another important property is convexity. In many real cases, this is an almost impossible condition; however, it's always useful to look for convex loss functions, because they can be easily optimized through the gradient descent method. We're going to discuss this topic in `Chapter 9`, *Neural Networks for Machine Learning*. However, for now, it's useful to consider a loss function as an intermediate between our training process and a pure mathematical optimization. The missing link is the complete data. As already discussed, *X* is drawn from p_{data}, so it should represent the true distribution. Therefore, when minimizing the loss function, we're considering a potential subset of points, and never the whole real dataset. In many cases, this isn't a limitation, because, if the bias is null and the variance is small enough, the resulting model will show a good generalization ability (high training and validation accuracy); however, considering the data generating process, it's useful to introduce another measure called **expected risk**:

$$E_{Risk}\left[f\right] = \int J\left(f\left(\bar{x}, \bar{\theta}\right), \bar{y}\right) p_{data}\left(\bar{x}, \bar{y}\right) d\bar{x} d\bar{y}$$

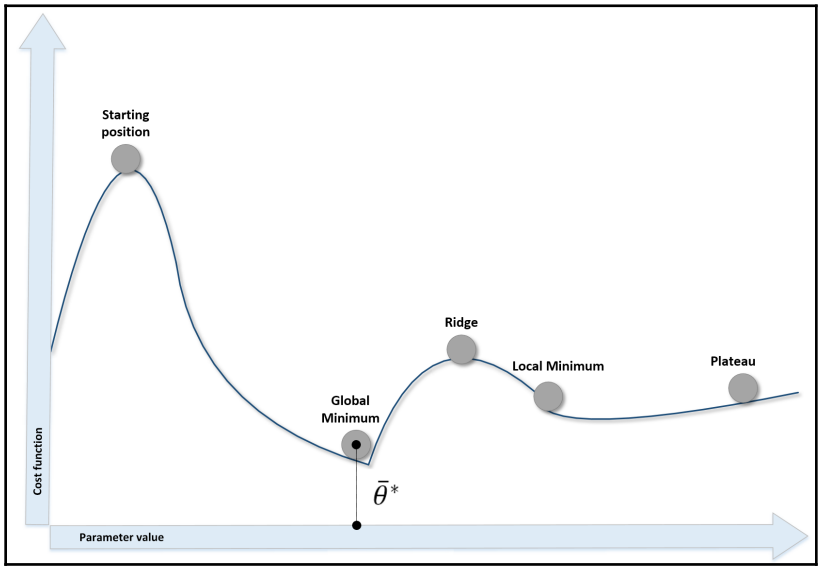
This value can be interpreted as an average of the loss function over all possible samples drawn from $p_{data.}$ Minimizing the expected risk implies the maximization of the global accuracy. When working with a finite number of training samples, instead, it's common to define a **cost function** (often called a loss function as well, and not to be confused with the log-likelihood):

$$L\left(X, Y; \bar{\theta}\right) = \sum_{i=0}^{N} J\left(\bar{x}_i, \bar{y}_i; \bar{\theta}\right)$$

This is the actual function that we're going to minimize and, divided by the number of samples (a factor that doesn't have any impact), it's also called **empirical risk**, because it's an approximation (based on real data) of the expected risk. In other words, we want to find a set of parameters so that:

$$\bar{\theta}^* = argmax_{\bar{\theta}} L\left(X, Y; \bar{\theta}\right)$$

When the cost function has more than two parameters, it's very difficult and perhaps even impossible to understand its internal structure; however, we can analyze some potential conditions using a bidimensional diagram:



Different kinds of points in a bidimensional scenario

The different situations we can observe are:

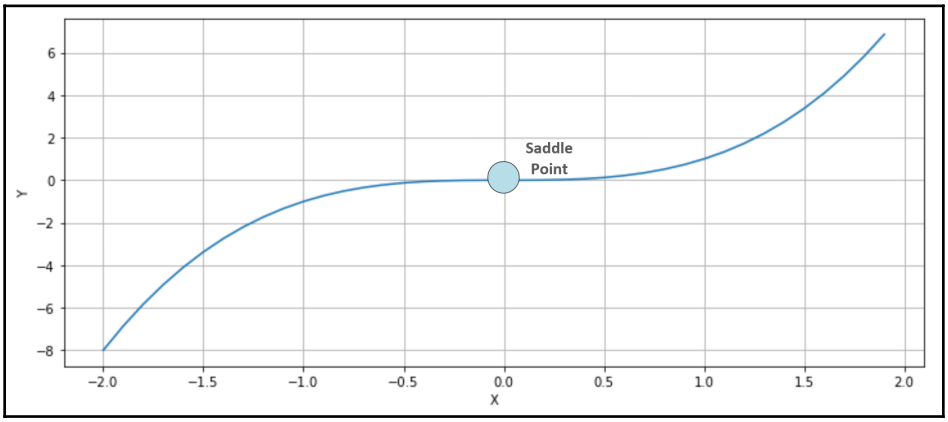
- The **starting point**, where the cost function is usually very high due to the error.
- **Local minima**, where the gradient is null (and the second derivative is positive). They are candidates for the optimal parameter set, but unfortunately, if the concavity isn't too deep, an inertial movement or some noise can easily move the point away.
- **Ridges** (or **local maxima**), where the gradient is null, and the second derivative is negative. They are unstable points, because a minimum perturbation allows escaping, reaching lower-cost areas.

- **Plateaus**, or the region where the surface is almost flat and the gradient is close to zero. The only way to escape a plateau is to keep a residual kinetic energy—we're going to discuss this concept when talking about neural optimization algorithms (`Chapter 9`, *Neural Networks for Machine Learning*).
- **Global minimum**, the point we want to reach to optimize the cost function.

Even if local minima are likely when the number of parameters is small, they become very unlikely when the model has a large number of parameters. In fact, an n-dimensional point θ^* is a local minimum for a convex function (and here, we're assuming L to be convex) only if:

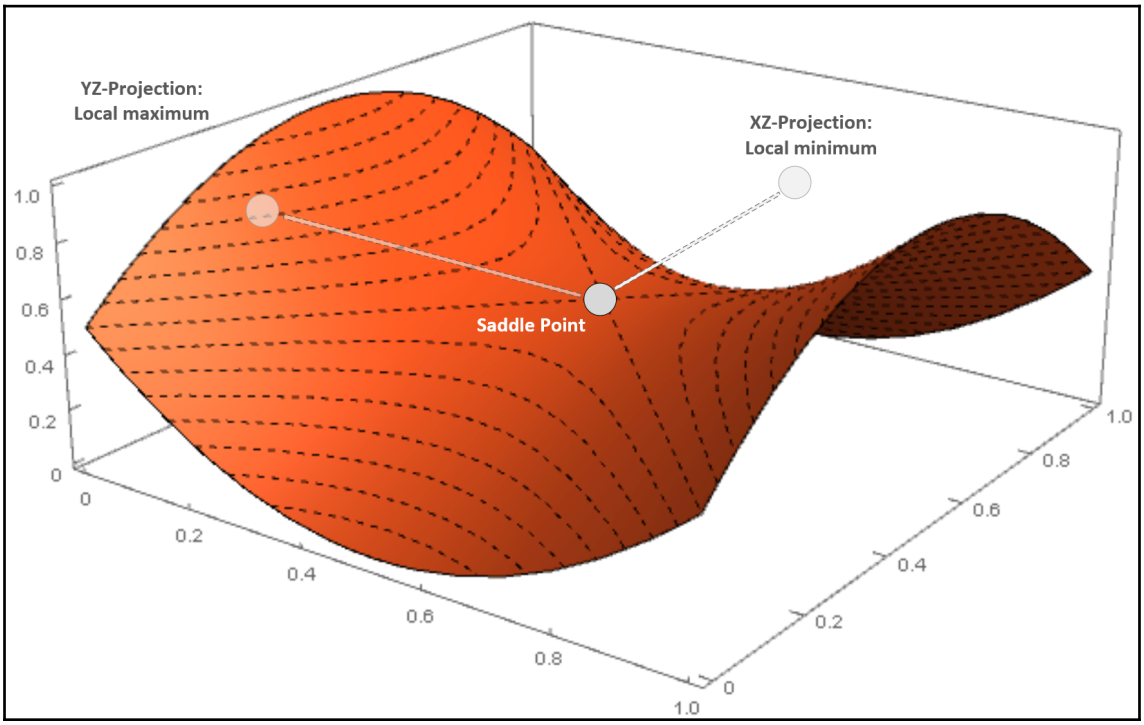
$$\begin{cases} \nabla_\theta L(x^*) = 0 \\ H_\theta L(\theta^*) \; positive \; semidef. \end{cases}$$

The second condition imposes a positive semi-definite Hessian matrix (equivalently, all principal minors H_n made with the first n rows and n columns must be non-negative), therefore all its eigenvalues $\lambda_0, \lambda_1, ..., \lambda_N$ must be non-negative. This probability decreases with the number of parameters (H is a $n \times n$ square matrix and has n eigenvalues), and becomes close to zero in deep learning models where the number of weights can be in the order of 10,000,000 (or even more). The reader interested in a complete mathematical proof can read *High Dimensional Spaces, Deep Learning and Adversarial Examples, Dube S., arXiv:1801.00634 [cs.CV]*. As a consequence, a more common condition to consider is instead the presence of **saddle points**, where the eigenvalues have different signs and the orthogonal directional derivatives are null, even if the points are neither local maxima nor minima. Consider, for example, the following plot:



Saddle point in a bidimensional scenario

The function is $y=x3$ whose first and second derivatives are $y'=3x2$ and $y''=6x$. Therefore, $y'(0)=y''(0)=0$. In this case (single-valued function), this point is also called a **point of inflection**, because at $x=0$, the function shows a change in the concavity. In three dimensions, it's easier to understand why a saddle point has been called in this way. Consider, for example, the following plot:



Saddle point in a three-dimensional scenario

The surface is very similar to a horse saddle, and if we project the point on an orthogonal plane, XZ is a minimum, while on another plane (YZ) it is a maximum. Saddle points are quite dangerous, because many simpler optimization algorithms can slow down and even stop, losing the ability to find the right direction. In `Chapter 9`, *Neural Networks for Machine Learning*, we're going to discuss some methods that are able to mitigate this kind of problem, allowing deep models to converge.

Examples of cost functions

In this section, we expose some common **cost functions** that are employed in both classification and regression tasks. Some of them will be extensively adopted in our examples in the next chapters, particularly when discussing training processes in shallow and deep neural networks.

Mean squared error

Mean squared error is one of the most common regression cost functions. Its generic expression is:

$$L\left(X,Y;\bar{\theta}\right)=\frac{1}{N}\sum_{i=0}^{N-1}\left(f\left(\bar{x}_i,\bar{\theta}\right)-y_i\right)^2$$

This function is differentiable at every point of its domain and it's convex, so it can be optimized using the **stochastic gradient descent** (**SGD**) algorithm; however, there's a drawback when employed in regressions where there are outliers. As its value is always quadratic when the distance between the prediction and the actual value (corresponding to an outlier) is large, the relative error is high, and this can lead to an unacceptable correction.

Huber cost function

As explained, mean squared error isn't robust to outliers, because it's always quadratic independently of the distance between actual value and prediction. To overcome this problem, it's possible to employ the **Huber cost function**, which is based on threshold t_H, so that for distances less than t_H, its behavior is quadratic, while for a distance greater than t_H, it becomes linear, reducing the entity of the error and, therefore, the relative importance of the outliers.

The analytical expression is:

$$L\left(X,Y;\bar{\theta},t_H\right)=\begin{cases}\frac{1}{2}\sum_{i=0}^{N-1}\left(f\left(\bar{x}_i,\bar{\theta}\right)-y_i\right)^2 & if\left|f\left(\bar{x}_i,\bar{\theta}\right)-y_i\right|\leqslant t_H\\ t_H\sum_{i=0}^{N-1}\left|f\left(\bar{x}_i,\bar{\theta}\right)-y_i\right|-\frac{t_H}{2} & if\left|f\left(\bar{x}_i,\bar{\theta}\right)-y_i\right|>t_H\end{cases}$$

Hinge cost function

This cost function is adopted by SVM, where the goal is to maximize the distance between the separation boundaries (where the support vector lies). It's analytic expression is:

$$L(X, Y; \bar{\theta}) = \sum_{i=0}^{N-1} max\left(0, 1 - f\left(\bar{x}_i, \bar{\theta}\right) \cdot y_i\right)$$

Contrary to the other examples, this cost function is not optimized using classic stochastic gradient descent methods, because it's not differentiable at all points where:

$$f\left(\bar{x}_i, \bar{\theta}\right) \cdot y_i = 1 \Rightarrow max(0, 0)$$

For this reason, SVM algorithms are optimized using quadratic programming techniques.

Categorical cross-entropy

Categorical cross-entropy is the most diffused classification cost function, adopted by logistic regression and the majority of neural architectures. The generic analytical expression is:

$$L(X, Y; \bar{\theta}) = -\sum_{i=0}^{N-1} y_i \; log \; f\left(\bar{x}_i, \bar{\theta}\right)$$

This cost function is convex and can be easily optimized using stochastic gradient descent techniques; moreover, it has another important interpretation. If we are training a classifier, our goal is to create a model whose distribution is as similar as possible to *pdata*. This condition can be achieved by minimizing the Kullback-Leibler divergence between the two distributions:

$$D_{KL}\left(p_{data} || \tilde{p}_M\right) = \sum_{i=0}^{N-1} p_{data}\left(\bar{x}_i, y_i\right) log \; \frac{p_{data}\left(\bar{x}_i, y_i\right)}{\tilde{p}_M\left(\bar{x}_i, y_i; \bar{\theta}\right)}$$

In the previous expression, p_M is the distribution generated by the model. Now, if we rewrite the divergence, we get:

$$D_{KL}\left(p_{data}||\tilde{p}_M\right) = \sum_{i=0}^{N-1} p_{data}\left(\bar{x}_i, y_i\right) log\, p_{data}\left(\bar{x}_i, y_i\right) - \sum_{i=0}^{N-1} p_{data}\left(\bar{x}_i, y_i\right) log\, \tilde{p}_M\left(\bar{x}_i, y_i; \bar{\theta}\right) =$$
$$= H\left(p_{data}\left(\bar{x}_i, y_i\right)\right) + H\left(p_{data}\left(\bar{x}_i, y_i\right), \tilde{p}_M\left(\bar{x}_i, y_i; \bar{\theta}\right)\right)$$

The first term is the entropy of the data-generating distribution, and it doesn't depend on the model parameters, while the second one is the cross-entropy. Therefore, if we minimize the cross-entropy, we also minimize the Kullback-Leibler divergence, forcing the model to reproduce a distribution that is very similar to p_{data}. This is a very elegant explanation as to why the cross-entropy cost function is an excellent choice for classification problems.

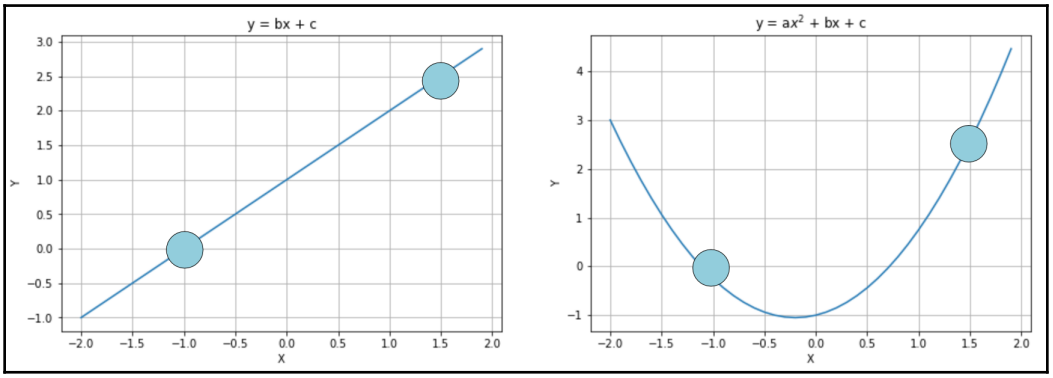
Regularization

When a model is ill-conditioned or prone to overfitting, **regularization** offers some valid tools to mitigate the problems. From a mathematical viewpoint, a regularizer is a penalty added to the cost function, so to impose an extra-condition on the evolution of the parameters:

$$L_R\left(X, Y; \bar{\theta}\right) = L\left(X, Y; \bar{\theta}\right) + \lambda g\left(\bar{\theta}\right)$$

The parameter λ controls the strength of the regularization, which is expressed through the function $g(\theta)$. A fundamental condition on $g(\theta)$ is that it must be differentiable so that the new composite cost function can still be optimized using SGD algorithms. In general, any regular function can be employed; however, we normally need a function that can contrast the indefinite growth of the parameters.

To understand the principle, let's consider the following diagram:



Interpolation with a linear curve (left) and a parabolic one (right)

In the first diagram, the model is linear and has two parameters, while in the second one, it is quadratic and has three parameters. We already know that the second option is more prone to overfitting, but if we apply a regularization term, it's possible to avoid the growth of a (first quadratic parameter), transforming the model into a linearized version. Of course, there's a difference between choosing a lower-capacity model and applying a regularization constraint. In fact, in the first case, we are renouncing the possibility offered by the extra capacity, running the risk of increasing the bias, while with regularization we keep the same model but optimize it so to reduce the variance. Let's now explore the most common regularization techniques.

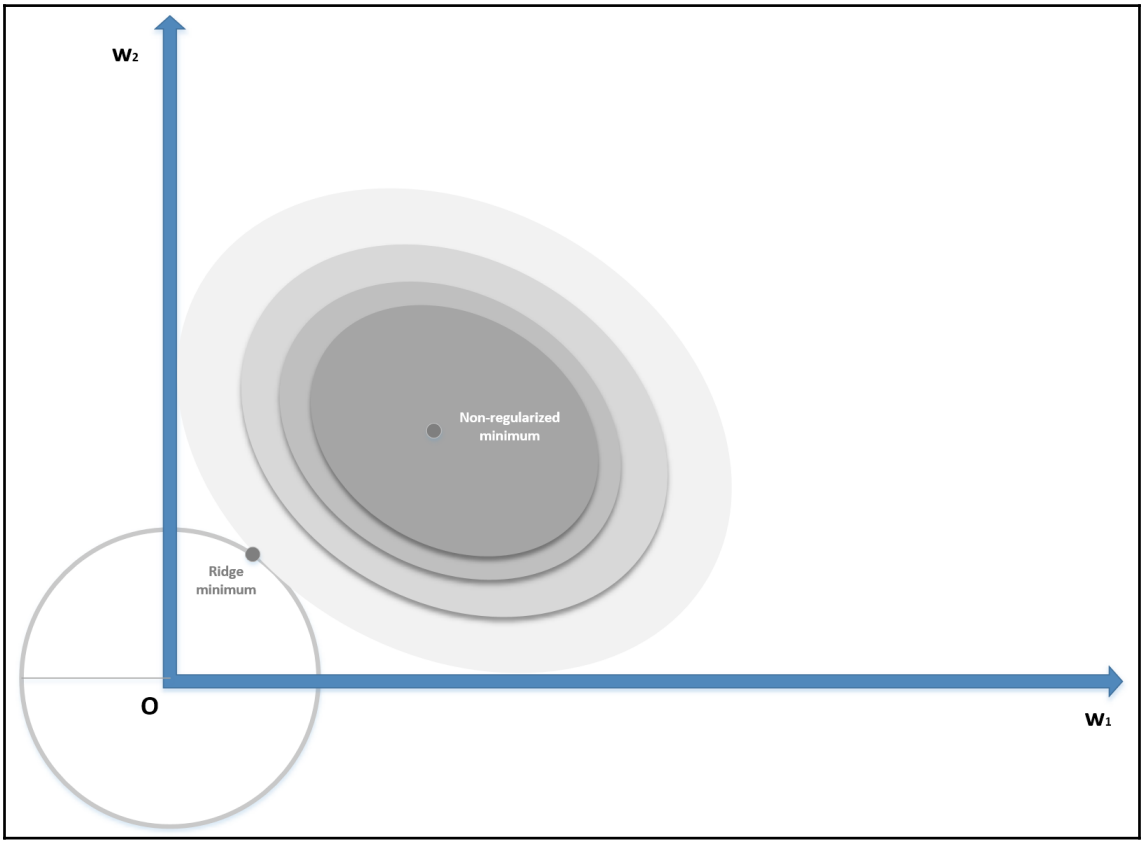
Ridge

Ridge regularization (also known as **Tikhonov regularization**) is based on the squared L2-norm of the parameter vector:

$$L_R\left(X, Y; \bar{\theta}\right) = L\left(X, Y; \bar{\theta}\right) + \lambda \left\| \bar{\theta} \right\|_2^2$$

This penalty avoids an infinite growth of the parameters (for this reason, it's also known as **weight shrinkage**), and it's particularly useful when the model is ill-conditioned, or there is multicollinearity, due to the fact that the samples are completely independent (a relatively common condition).

In the following diagram, we see a schematic representation of the Ridge regularization in a bidimensional scenario:



Ridge (L2) regularization

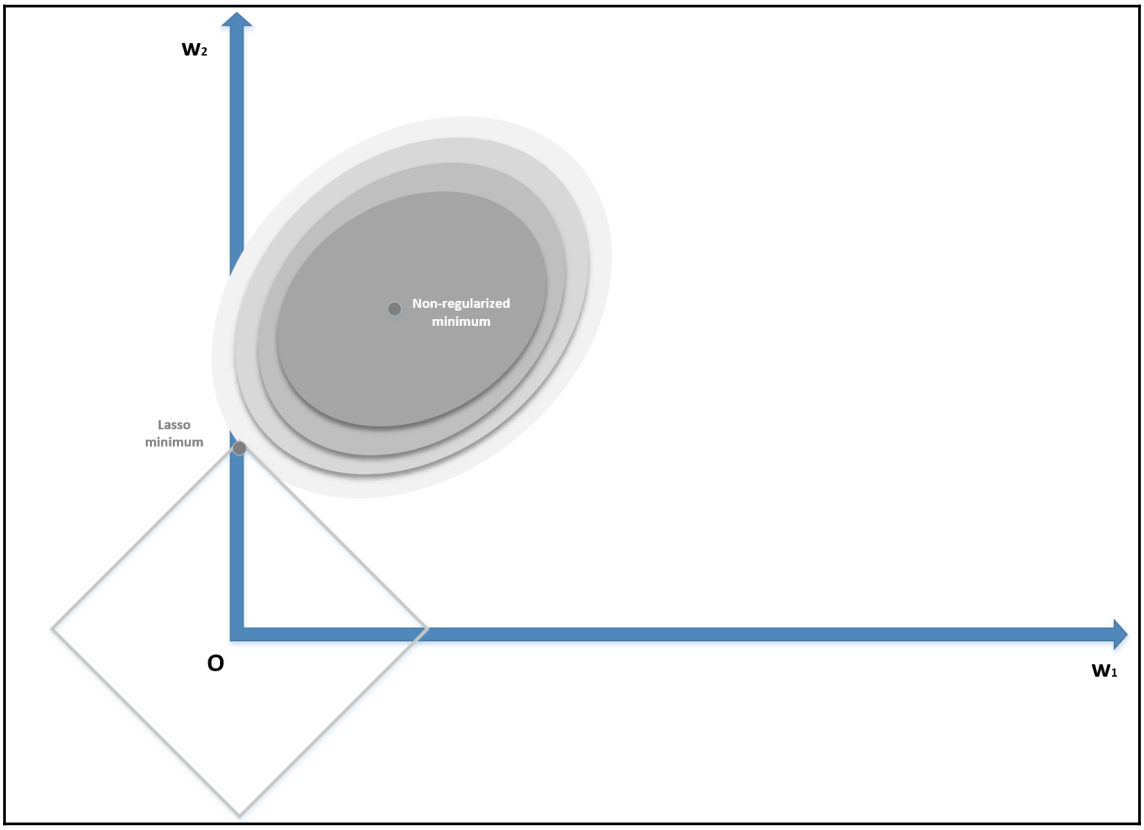
The zero-centered circle represents the Ridge boundary, while the shaded surface is the original cost function. Without regularization, the minimum ($\mathbf{w}_1$, $\mathbf{w}_2$) has a magnitude (for example, the distance from the origin) which is about double the one obtained by applying a Ridge constraint, confirming the expected shrinkage. When applied to regressions solved with the **Ordinary Least Squares** (**OLS**) algorithm, it's possible to prove that there always exists a Ridge coefficient, so that the weights are shrunk with respect the OLS ones. The same result, with some restrictions, can be extended to other cost functions.

Lasso

Lasso regularization is based on the *L1*-norm of the parameter vector:

$$L_R\left(X,Y;\bar{\theta}\right) = L\left(X,Y;\bar{\theta}\right) + \lambda\left\|\bar{\theta}\right\|_1$$

Contrary to Ridge, which shrinks all the weights, Lasso can shift the smallest one to zero, creating a sparse parameter vector. The mathematical proof is beyond the scope of this book; however, it's possible to understand it intuitively by considering the following diagram (bidimensional):



Lasso (L1) regularization

The zero-centered square represents the Lasso boundaries. If we consider a generic line, the probability of being tangential to the square is higher at the corners, where at least one (exactly one in a bidimensional scenario) parameter is null. In general, if we have a vectorial convex function *f(x)* (we provide a definition of convexity in `Chapter 5`, *EM Algorithm and Applications*), we can define:

$$g(\bar{x}) = f(\bar{x}) + \|\bar{x}\|_p$$

As any L_p-norm is convex, as well as the sum of convex functions, *g(x)* is also convex. The regularization term is always non-negative, therefore the minimum corresponds to the norm of the null vector. When minimizing *g(x)*, we need to also consider the contribution of the gradient of the norm in the ball centered in the origin where, however, the partial derivatives don't exist. Increasing the value of *p*, the norm becomes smoothed around the origin, and the partial derivatives approach zero for $|x_i| \rightarrow 0$.

On the other side, with *p=1* (excluding the L_0-norm and all the norms with $p \in]0, 1[$ that allow an even stronger sparsity, but are non-convex), the partial derivatives are always +1 or -1, according to the sign of x_i ($x_i \neq 0$). Therefore, it's *easier* for the L_1-norm to push the smallest components to zero, because the contribution to the minimization (for example, with a gradient descent) is independent of x_i, while an L_2-norm decreases its *speed* when approaching the origin. This is a non-rigorous explanation of the sparsity achieved using the L_1-norm. In fact, we also need to consider the term *f(x)*, which bounds the value of the global minimum; however, it may help the reader to develop an intuitive understanding of the concept. It's possible to find further and mathematically rigorous details in *Optimization for Machine Learning, (e*dited by) *Sra S., Nowozin S., Wright S. J., The MIT Press*.

Lasso regularization is particularly useful whenever a sparse representation of a dataset is needed. For example, we could be interested in finding the feature vectors corresponding to a group of images. As we expect to have many features but only a subset present in each image, applying the Lasso regularization allows forcing all the smallest coefficients to become null, suppressing the presence of the secondary features. Another potential application is latent semantic analysis, where our goal is to describe the documents belonging to a corpus in terms of a limited number of topics. All these methods can be summarized in a technique called **sparse coding**, where the objective is to reduce the dimensionality of a dataset (also in non-linear scenarios) by extracting the most representative atoms, using different approaches to achieve sparsity.

ElasticNet

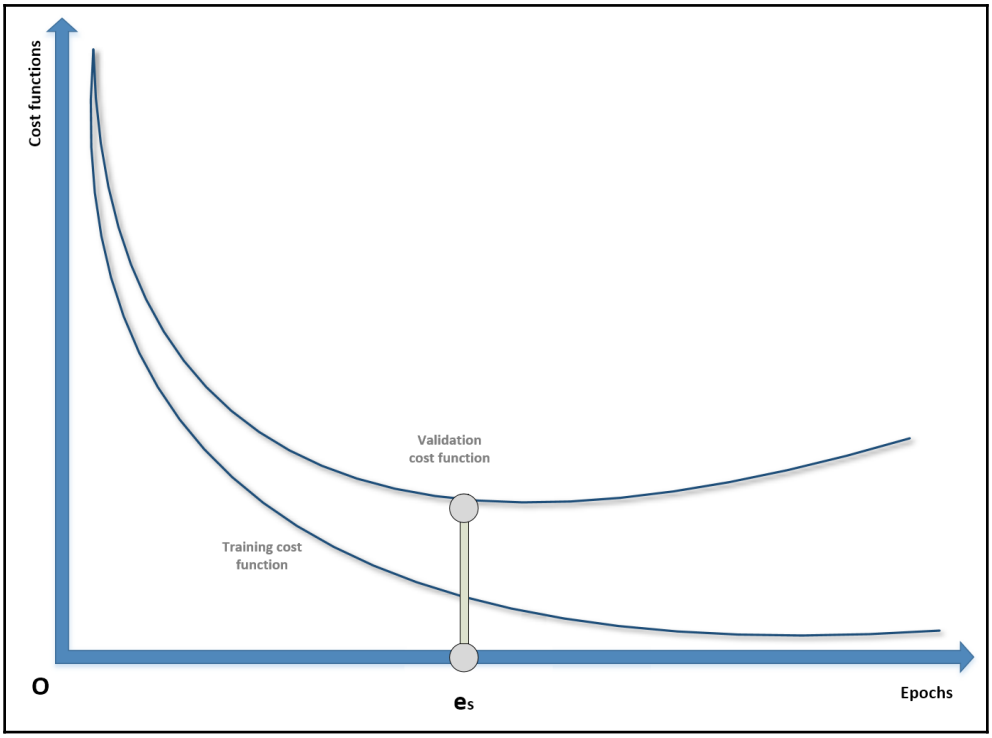
In many real cases, it's useful to apply both Ridge and Lasso regularization in order to force weight shrinkage and a global sparsity. It is possible by employing the **ElasticNet** regularization, defined as:

$$L_R\left(X, Y; \bar{\theta}\right) = L\left(X, Y; \bar{\theta}\right) + \lambda_1 \left\|\bar{\theta}\right\|_2^2 + \lambda_2 \left\|\bar{\theta}\right\|_1$$

The strength of each regularization is controlled by the parameters λ_1 and λ_2. ElasticNet can yield excellent results whenever it's necessary to mitigate overfitting effects while encouraging sparsity. We are going to apply all the regularization techniques when discussing some deep learning architectures.

Early stopping

Even though it's a pure regularization technique, **early stopping** is often considered as a *last resort* when all other approaches to prevent overfitting and maximize validation accuracy fail. In many cases (above all, in deep learning scenarios), it's possible to observe a typical behavior of the training process considering both training and the validation cost functions:



Example of early stopping before the beginning of ascending phase of U-curve

During the first epochs, both costs decrease, but it can happen that after a *threshold* epoch e_s, the validation cost starts increasing. If we continue with the training process, this results in overfitting the training set and increasing the variance. For this reason, when there are no other options, it's possible to prematurely stop the training process. In order to do so, it's necessary to store the last parameter vector before the beginning of a new iteration and, in the case of no improvements or the accuracy worsening, to stop the process and recover the last parameters. As explained, this procedure must never be considered as the best choice, because a better model or an improved dataset could yield higher performances. With early stopping, there's no way to verify alternatives, therefore it must be adopted only at the last stage of the process and never at the beginning. Many deep learning frameworks such as Keras include helpers to implement an early stopping callback; however, it's important to check whether the last parameter vector is the one stored before the last epoch or the one corresponding to e_s. In this case, it could be useful to repeat the training process, stopping it at the epoch previous to e_s (where the minimum validation cost has been achieved).

Summary

In this chapter, we discussed fundamental concepts shared by almost any machine learning model. In the first part, we have introduced the data generating process, as a generalization of a finite dataset. We explained which are the most common strategies to split a finite dataset into a training block and a validation set, and we introduced cross-validation, with some of the most important variants, as one of the best approaches to avoid the limitations of a static split.

In the second part, we discussed the main properties of an estimator: capacity, bias, and variance. We also introduced the Vapnik-Chervonenkis theory, which is a mathematical formalization of the concept of representational capacity, and we analyzed the effects of high biases and high variances. In particular, we discussed effects called underfitting and overfitting, defining the relationship with high bias and high variance.

In the third part, we introduced the loss and cost functions, first as proxies of the expected risk, and then we detailed some common situations that can be experienced during an optimization problem. We also exposed some common cost functions, together with their main features. In the last part, we discussed regularization, explaining how it can mitigate the effects of overfitting.

In the next chapter, `Chapter 2`, *Introduction to Semi-Supervised Learning*, we're going to introduce semi-supervised learning, focusing our attention on the concepts of transductive and inductive learning.

2
Introduction to Semi-Supervised Learning

Semi-supervised learning is a machine learning branch that tries to solve problems with both labeled and unlabeled data with an approach that employs concepts belonging to clustering and classification methods. The high availability of unlabeled samples, in contrast with the difficulty of labeling huge datasets correctly, drove many researchers to investigate the best approaches that allow extending the knowledge provided by the labeled samples to a larger unlabeled population without loss of accuracy. In this chapter, we're going to introduce this branch and, in particular, we will discuss:

- The semi-supervised scenario
- The assumptions needed to efficiently operate in such a scenario
- The different approaches to semi-supervised learning
- Generative Gaussian mixtures algorithm
- Contrastive pessimistic likelihood estimation approach
- **Semi-supervised Support Vector Machines** (**S^3VM**)
- **Transductive Support Vector Machines** (**TSVM**)

Semi-supervised scenario

A typical semi-supervised scenario is not very different from a supervised one. Let's suppose we have a data generating process, p_{data}:

$$p_{data}(\bar{x}, \bar{y}) = p(\bar{y}|\bar{x})p(\bar{x})$$

However, contrary to a supervised approach, we have only a limited number *N* of samples drawn from p_{data} and provided with a label, as follows:

$$\begin{cases} X_l = \{\bar{x}_0^l, \bar{x}_1^l, \ldots, \bar{x}_N^l\} \;\; where \;\; \bar{x}_i^l \in \mathbb{R}^p \\ Y_l = \{\bar{y}_0^l, \bar{y}_1^l, \ldots, \bar{y}_N^l\} \;\; where \;\; \bar{y}_i^l \in \mathbb{R}^q \end{cases}$$

Instead, we have a larger amount (*M*) of unlabeled samples drawn from the marginal distribution *p(x)*:

$$X_u = \{\bar{x}_0^u, \bar{x}_1^u, \ldots, \bar{x}_M^u\} \;\; where \;\; \bar{x}_i^u \in \mathbb{R}^p$$

In general, there are no restrictions on the values of *N* and *M;* however, a semi-supervised problem arises when the number of unlabeled samples is much higher than the number of complete samples. If we can draw $N >> M$ labeled samples from p_{data}, it's probably useless to keep on working with semi-supervised approaches and preferring classical supervised methods is likely to be the best choice. The extra complexity we need is justified by $M >> N$, which is a common condition in all those situations where the amount of available unlabeled data is large, while the number of correctly labeled samples is quite a lot lower. For example, we can easily access millions of free images but detailed labeled datasets are expensive and include only a limited subset of possibilities. However, is it always possible to apply semi-supervised learning to improve our models? The answer to this question is almost obvious: unfortunately no. As a rule of thumb, we can say that if the knowledge of X_u increases our knowledge about the prior distribution *p(x)*, a semi-supervised algorithm is likely to perform better than a purely supervised (and thus limited to X_l) counterpart. On the other hand, if the unlabeled samples are drawn from different distributions, the final result can be quite a lot worse. In real cases, it's not so immediately necessary to decide whether a semi-supervised algorithm is the best choice; therefore, cross-validation and comparisons are the best practices to employ when evaluating a scenario.

Transductive learning

When a semi-supervised model is aimed at finding the labels for the unlabeled samples, the approach is called transductive learning. In this case, we are not interested in modeling the whole distribution $p(x|y)$, which implies determining the density of both datasets, but rather in finding $p(y|x)$ only for the unlabeled points. In many cases, this strategy can be time-saving and it's always preferable when our goal is more oriented at improving our knowledge about the unlabeled dataset.

Inductive learning

Contrary to transductive learning, inductive learning considers all the X samples and tries to determine a complete $p(x|y)$ or a function $y=f(x)$ that can map both labeled and unlabeled points to their corresponding labels. In general, this method is more complex and requires more computational time; therefore, according to *Vapnik's principle*, if not required or necessary, it's always better to pick the most pragmatic solution and, possibly, expand it if the problem requires further details.

Semi-supervised assumptions

As explained in the previous section, semi-supervised learning is not guaranteed to improve a supervised model. A wrong choice could lead to a dramatic worsening in performance; however, it's possible to state some fundamental assumptions which are required for semi-supervised learning to work properly. They are not always mathematically proven theorems, but rather empirical observations that justify the choice of an approach otherwise completely arbitrary.

Smoothness assumption

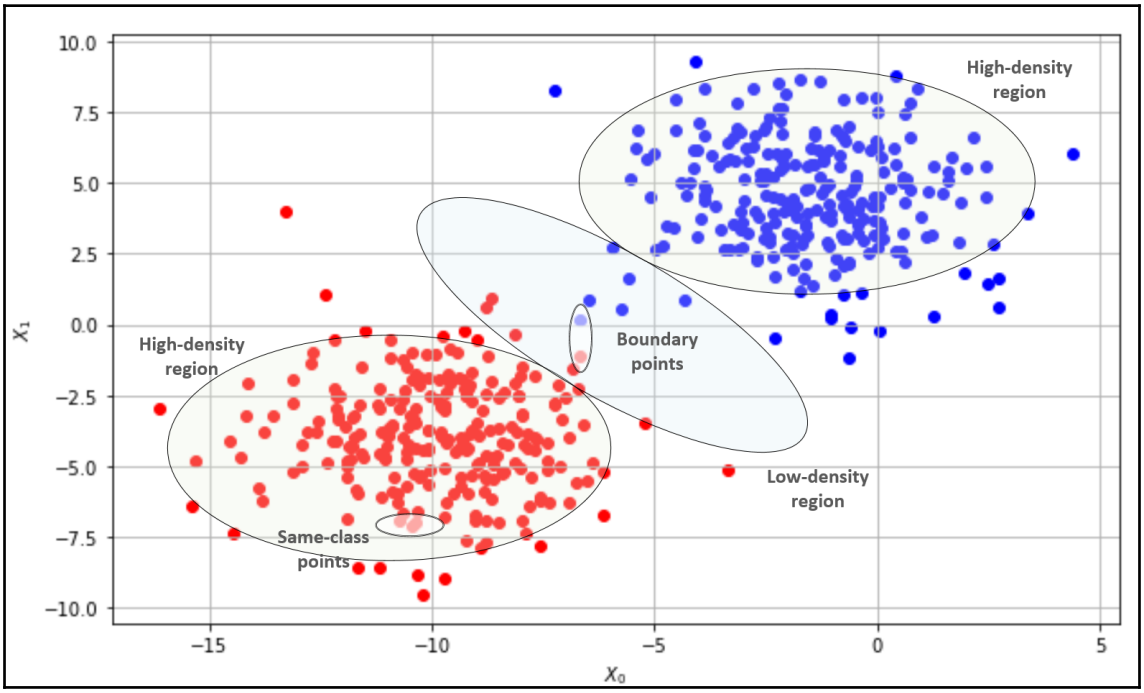
Let's consider a real-valued function $f(x)$ and the corresponding metric spaces X and Y. Such a function is said to be Lipschitz-continuous if:

$$\exists\, K : \forall\, x_1 \; and \; x_2 \in X \Rightarrow d_Y\left(f(x_1), f(x_2)\right) \leqslant K d_X(x_1, x_2)$$

In other words, if two points x_1 and x_2 are near, the corresponding output values y_1 and y_2 cannot be arbitrarily far from each other. This condition is fundamental in regression problems where a generalization is often required for points that are between training samples. For example, if we need to predict the output for a point $x_t : x_1 < x_t < x_2$ and the regressor is Lipschitz-continuous, we can be sure that y_t will be correctly bounded by y_1 and y_2. This condition is often called general smoothness, but in semi-supervised it's useful to add a restriction (correlated with the cluster assumption): if two points are in a high density region (cluster) and they are close, then the corresponding outputs must be close too. This extra condition is very important because, if two samples are in a low density region they can belong to different clusters and their labels can be very different. This is not always true, but it's useful to include this constraint to allow some further assumptions in many definitions of semi-supervised models.

Cluster assumption

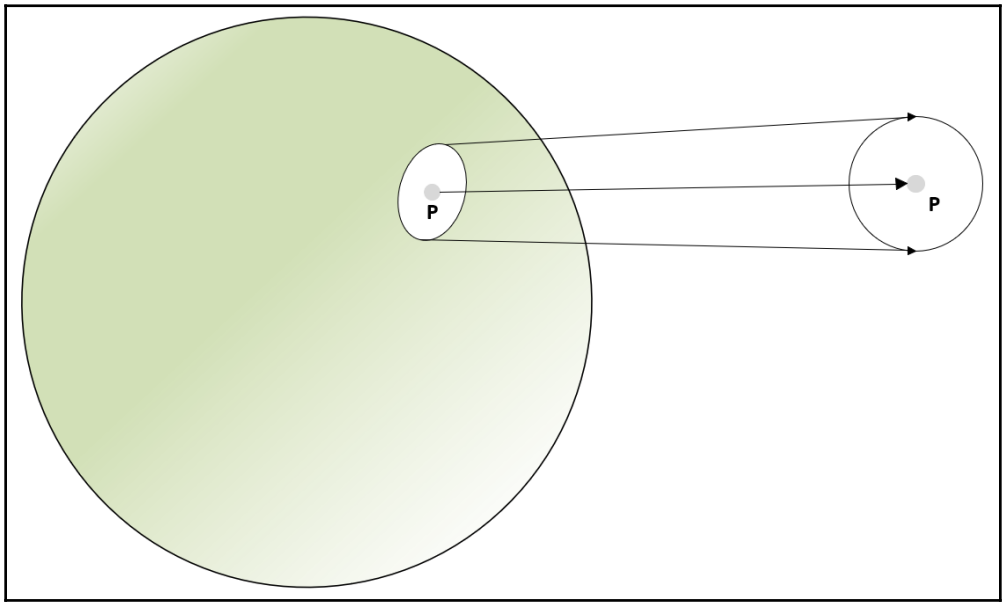
This assumption is strictly linked to the previous one and it's probably easier to accept. It can be expressed with a chain of interdependent conditions. Clusters are high density regions; therefore, if two points are close, they are likely to belong to the same cluster and their labels must be the same. Low density regions are separation spaces; therefore, samples belonging to a low density region are likely to be boundary points and their classes can be different. To better understand this concept, it's useful to think about supervised SVM: only the support vectors should be in low density regions. Let's consider the following bidimensional example:



In a semi-supervised scenario, we couldn't know the label of a point belonging to a high density region; however, if it is close enough to a labeled point that it's possible to build a ball where all the points have the same average density, we are allowed to predict the label of our test sample. Instead, if we move to a low-density region, the process becomes harder, because two points can be very close but with different labels. We are going to discuss the semi-supervised, low-density separation problem at the end of this chapter.

Manifold assumption

This is the less intuitive assumption, but it can be extremely useful to reduce the complexity of many problems. First of all, we can provide a non-rigorous definition of a manifold. An *n*-manifold is a topological space that is globally curved, but locally homeomorphic to an *n*-dimensional Euclidean space. In the following diagram, there's an example of a manifold: the surface of a sphere in $\Re^3$:



2D manifold obtained from a spherical surface

The small patch around *P* (for $\varepsilon \to 0$) can be mapped to a flat circular surface. Therefore, the properties of a manifold are locally based on the Euclidean geometry, while, globally, they need a proper mathematical extension which is beyond the scope of this book (further information can be found in *Semi-supervised learning on Riemannian manifolds*, *Belkin M., Niyogi P., Machine Learning 56, 2004*).

The manifold assumption states that p-dimensional samples (where $p >> 1$) approximately lie on a q-dimensional manifold with $p << q$. Without excessive mathematical rigor, we can say that, for example, if we have N *1000*-dimensional bounded vectors, they are enclosed into a *1000*-dimensional hypercube with edge-length equal to r. The corresponding n-volume is $r^p = r^{1000}$, therefore, the probability of filling the entire space is very small (and decreases with p). What we observe, instead, is a high density on a lower dimensional manifold. For example, if we look at the Earth from space, we might think that its inhabitants are uniformly distributed over the whole volume. We know that this is false and, in fact, we can create maps and atlases which are represented on two-dimensional manifolds. It doesn't make sense to use three-dimensional vectors to map the position of a human being. It's easier to use a projection and work with latitude and longitude.

This assumption authorizes us to apply dimensionality reduction methods in order to avoid the *Curse of Dimensionality*, theorized by Bellman (in *Dynamic Programming and Markov Process, Ronald A. Howard, The MIT Press*). In the scope of machine learning, the main consequence of such an effect is that when the dimensionality of the samples increases, in order to achieve a high accuracy, it's necessary to use more and more samples. Moreover, Hughes observed (the phenomenon has been named after him and it's presented in the paper *Hughes G. F., On the mean accuracy of statistical pattern recognizers, IEEE Transactions on Information Theory, 1968, 14/1*) that the accuracy of statistical classifiers is inversely proportional to the dimensionality of the samples. This means that whenever it's possible to work on lower dimensional manifolds (in particular in semi-supervised scenarios), two advantages are achieved:

- Less computational time and memory consumption
- Higher classification accuracy

Generative Gaussian mixtures

Generative Gaussian mixtures is an inductive algorithm for semi-supervised clustering. Let's suppose we have a labeled dataset (X_l, Y_l) containing N samples (drawn from p_{data}) and an unlabeled dataset X_u containing $M >> N$ samples (drawn from the marginal distribution $p(x)$). It's not necessary that $M >> N$, but we want to create a real semi-supervised scenario, with only a few labeled samples. Moreover, we are assuming that all unlabeled samples are consistent with p_{data}. This can seem like a vicious cycle, but without this assumption, the procedure does not have a strong mathematical foundation. Our goal is to determine a complete $p(x|y)$ distribution using a generative model. In general, it's possible to use different priors, but we are now employing multivariate Gaussians to model our data:

$$f(\bar{x};\bar{\mu};\Sigma) = \frac{1}{\sqrt{det(2\pi\Sigma)}} e^{-\frac{(\bar{x}-\bar{\mu})^T\Sigma^{-1}(\bar{x}-\bar{\mu})}{2}}$$

Thus, our model parameters are means and covariance matrices for all Gaussians. In other contexts, it's possible to use binomial or multinomial distributions. However, the procedure doesn't change; therefore, let's assume that it's possible to approximate $p(x|y)$ with a parametrized distribution $p(x|y, \theta)$. We can achieve this goal by minimizing the Kullback-Leibler divergence between the two distributions:

$$argmin_{\bar{\theta}} D_{KL}(p(\bar{x}|y)||p(\bar{x}|y,\bar{\theta})) = \sum_i p(\bar{x}_i|y_i) log \frac{p(\bar{x}_i|y_i)}{p(\bar{x}_i|y_i,\bar{\theta})}$$

In `Chapter 5`, *EM Algorithm and Applications* we are going to show that this is equivalent to maximizing the likelihood of the dataset. To obtain the likelihood, it's necessary to define the number of expected Gaussians (which is known from the labeled samples) and a weight-vector that represents the marginal probability of a specific Gaussian:

$$\bar{w} = (p(y=1), p(y=2), \ldots, p(y=M))$$

Using the Bayes' theorem, we get:

$$p(y_i|\bar{x}_j,\bar{\theta},\bar{w}) \propto w_i p(\bar{x}_j|y_i,\bar{\theta})$$

As we are working with both labeled and unlabeled samples, the previous expression has a double interpretation:

- For unlabeled samples, it is computed by multiplying the i^{th} Gaussian weight times the probability $p(x_j)$ relative to the i^{th} Gaussian distribution.
- For labeled samples, it can be represented by a vector p = [0, 0, ... 1, ... 0, 0] where 1 is the i^{th} element. In this way, we force our model to trust the labeled samples in order to find the best parameter values that maximize the likelihood on the whole dataset.

With this distinction, we can consider a single log-likelihood function where the term $f_w(y_i|x_j)$ has been substituted by a per sample weight:

$$L(\bar{\theta};\bar{w}) = \sum_j log \sum_i f_w(y_i|\bar{x}_j) p(\bar{x}_j|y_i,\bar{\theta}) = \sum_j log \sum_i w_i p(\bar{x}_j|y_i,\bar{\theta})$$

It's possible to maximize the log-likelihood using the EM algorithm (see `Chapter 5`, *EM Algorithm and Applications*). In this context, we provide the steps directly:

- $p(y_i|x_j,\theta,w)$ is computed according to the previously explained method
- The parameters of the Gaussians are updated using these rules:

$$w_i = \frac{\sum_j p(y_i|\bar{x}_j,\bar{\theta},\bar{w})}{N}$$

$$\bar{\mu}_i = \frac{\sum_j \left(p(y_i|\bar{x}_j,\bar{\theta},\bar{w})\bar{x}_j\right)}{\sum_j p(y_i|\bar{x}_j,\bar{\theta},\bar{w})}$$

$$\Sigma_i = \frac{\sum_j \left(p(y_i|\bar{x}_j,\bar{\theta},\bar{w})(\bar{x}_j-\bar{\mu}_i)(\bar{x}_j-\bar{\mu}_i)^T\right)}{\sum_j p(y_i|\bar{x}_j,\bar{\theta},\bar{w})}$$

N is the total number of samples. The procedure must be iterated until the parameters stop modifying or the modifications are lower than a fixed threshold.

Example of a generative Gaussian mixture

We can now implement this model in Python using a simple bidimensional dataset, created using the `make_blobs()` function provided by Scikit-Learn:

```
from sklearn.datasets import make_blobs

nb_samples = 1000
nb_unlabeled = 750

X, Y = make_blobs(n_samples=nb_samples, n_features=2, centers=2,
cluster_std=2.5, random_state=100)

unlabeled_idx = np.random.choice(np.arange(0, nb_samples, 1),
replace=False, size=nb_unlabeled)
Y[unlabeled_idx] = -1
```

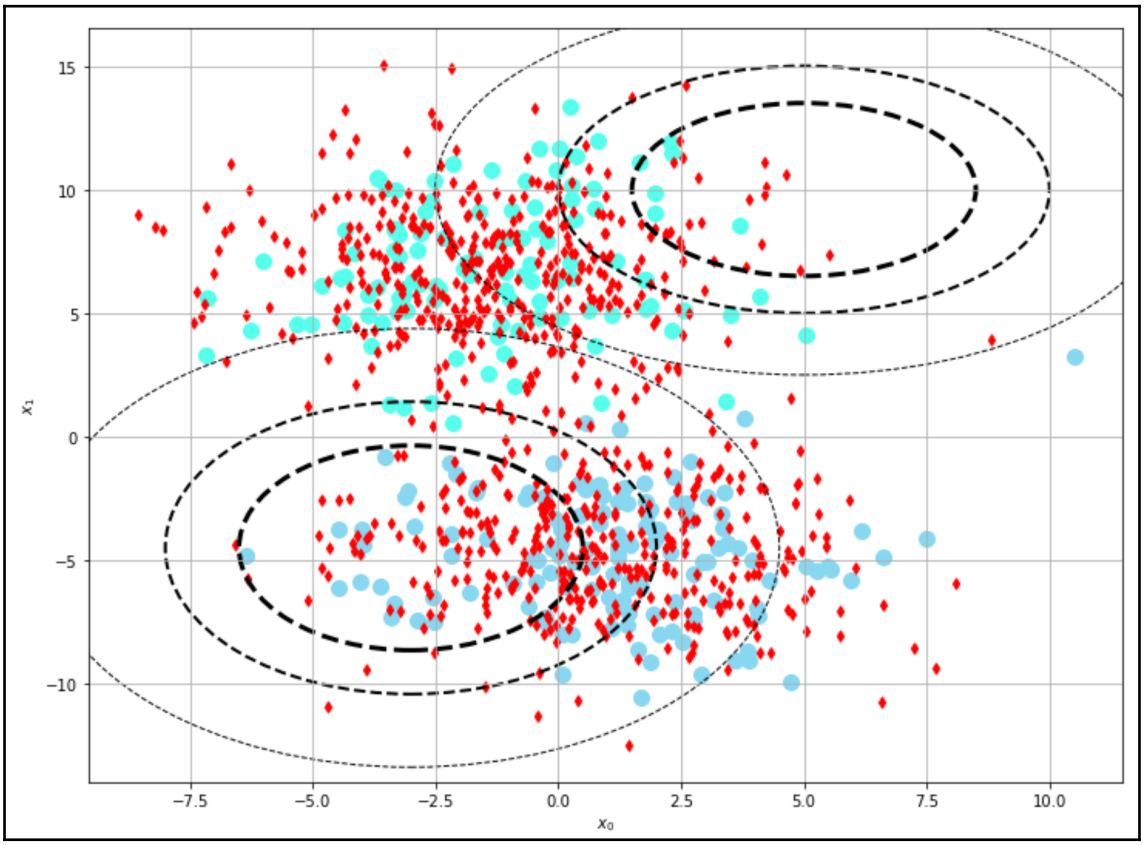
We have created 1,000 samples belonging to 2 classes. 750 points have then been randomly selected to become our unlabeled dataset (the corresponding class has been set to -1). We can now initialize two Gaussian distributions by defining their mean, covariance, and weight. One possibility is to use random values:

```
import numpy as np

# First Gaussian
m1 = np.random.uniform(-7.5, 10.0, size=2)
c1 = np.random.uniform(5.0, 15.0, size=(2, 2))
c1 = np.dot(c1, c1.T)
q1 = 0.5

# Second Gaussian
m2 = np.random.uniform(-7.5, 10.0, size=2)
c2 = np.random.uniform(5.0, 15.0, size=(2, 2))
c2 = np.dot(c2, c2.T)
q2 = 0.5
```

However, as the covariance matrices must be positive semi definite, it's useful to alter the random values (by multiplying each matrix by the corresponding transpose) or to set hard-coded initial parameters. In this case, we could pick the following example:

```
import numpy as np

# First Gaussian
m1 = np.array([-3.0, -4.5])
c1 = np.array([[25.0, 5.0],
               [5.0, 35.0]])
q1 = 0.5

# Second Gaussian
m2 = np.array([5.0, 10.0])
c2 = np.array([[25.0, -10.0],
               [-10.0, 25.0]])
q2 = 0.5
```

The resulting plot is shown in the following graph, where the small diamonds represent the unlabeled points and the bigger dots, the samples belonging to the known classes:

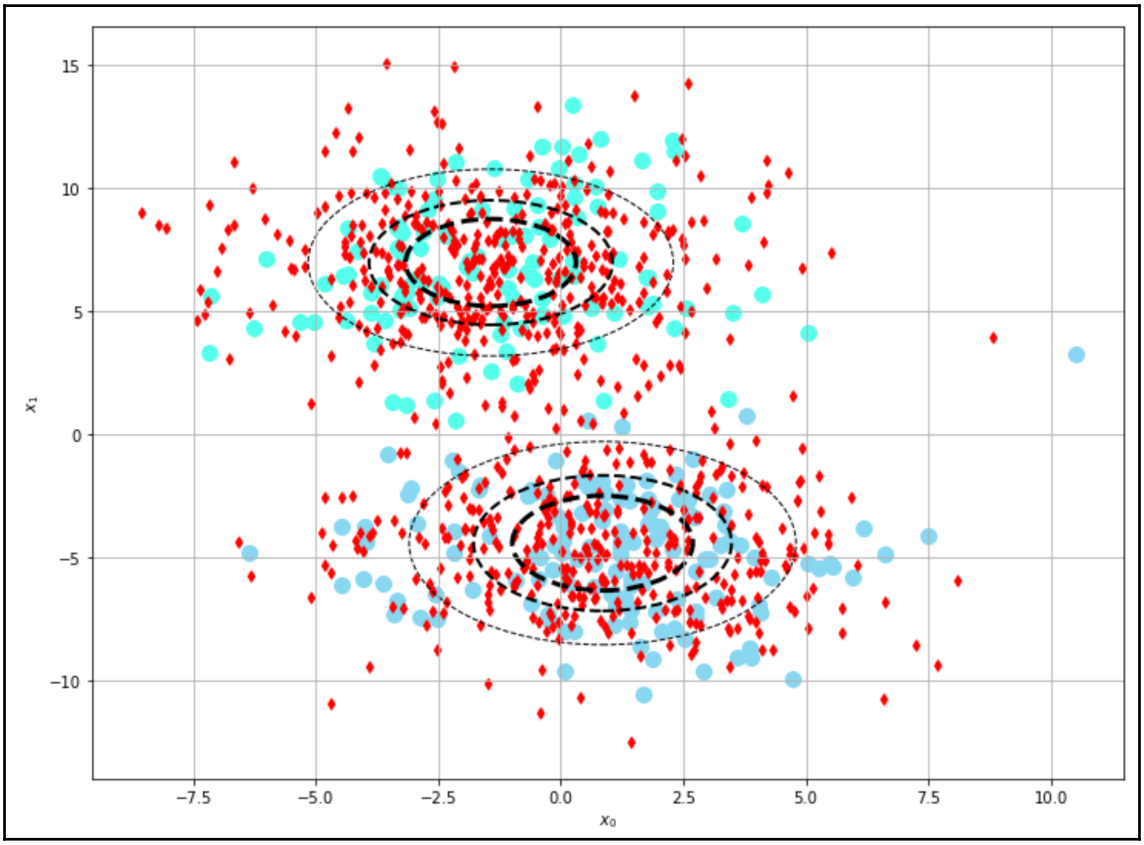


Initial configuration of the Gaussian mixture

The two Gaussians are represented by the concentric ellipses. We can now execute the training procedure. For simplicity, we repeat the update for a fixed number of iterations. The reader can easily modify the code in order to introduce a threshold:

```
from scipy.stats import multivariate_normal

nb_iterations = 5

for i in range(nb_iterations):
    Pij = np.zeros((nb_samples, 2))
    for i in range(nb_samples):
        if Y[i] == -1:
            p1 = multivariate_normal.pdf(X[i], m1, c1, allow_singular=True)
* q1
            p2 = multivariate_normal.pdf(X[i], m2, c2, allow_singular=True)
* q2
            Pij[i] = [p1, p2] / (p1 + p2)
        else:
            Pij[i, :] = [1.0, 0.0] if Y[i] == 0 else [0.0, 1.0]
    n = np.sum(Pij, axis=0)
    m = np.sum(np.dot(Pij.T, X), axis=0)
    m1 = np.dot(Pij[:, 0], X) / n[0]
    m2 = np.dot(Pij[:, 1], X) / n[1]
    q1 = n[0] / float(nb_samples)
    q2 = n[1] / float(nb_samples)
    c1 = np.zeros((2, 2))
    c2 = np.zeros((2, 2))

    for t in range(nb_samples):
        c1 += Pij[t, 0] * np.outer(X[t] - m1, X[t] - m1)
        c2 += Pij[t, 1] * np.outer(X[t] - m2, X[t] - m2)
    c1 /= n[0]
    c2 /= n[1]
```

The first thing at the beginning of each cycle is to initialize the `Pij` matrix that will be used to store the $p(y_i|x_j,\theta,w)$ values. Then, for each sample, we can compute $p(y_i|x_j,\theta,w)$ considering whether it's labeled or not. The Gaussian probability is computed using the SciPy function `multivariate_normal.pdf()`. When the whole P_{ij} matrix has been populated, we can update the parameters (means and covariance matrix) of both Gaussians and the relative weights. The algorithm is very fast; after five iterations, we get the stable state represented in the following graph:



The two Gaussians have perfectly mapped the space by setting their parameters so as to cover the high-density regions. We can check for some unlabeled points, as follows:

```
print(np.round(X[Y==-1][0:10], 3))

[[  1.67    7.204]
 [ -1.347  -5.672]
 [ -2.395  10.952]
 [ -0.261   6.526]
 [  1.053   8.961]
 [ -0.579  -7.431]
 [  0.956   9.739]
 [ -5.889   5.227]
 [ -2.761   8.615]
 [ -1.777   4.717]]
```

It's easy to locate them in the previous plot. The corresponding classes can be obtained through the last P_{ij} matrix:

```
print(np.round(Pij[Y==-1][0:10], 3))

[[ 0.002  0.998]
 [ 1.     0.   ]
 [ 0.     1.   ]
 [ 0.003  0.997]
 [ 0.     1.   ]
 [ 1.     0.   ]
 [ 0.     1.   ]
 [ 0.007  0.993]
 [ 0.     1.   ]
 [ 0.02   0.98 ]]
```

This immediately verifies that they have been correctly labeled and assigned to the right cluster. This algorithm is very fast and produces excellent results in terms of density estimation. In `Chapter 5`, *EM Algorithm and Applications*, we are going to discuss a general version of this algorithm, explaining the complete training procedure based on the EM algorithm.

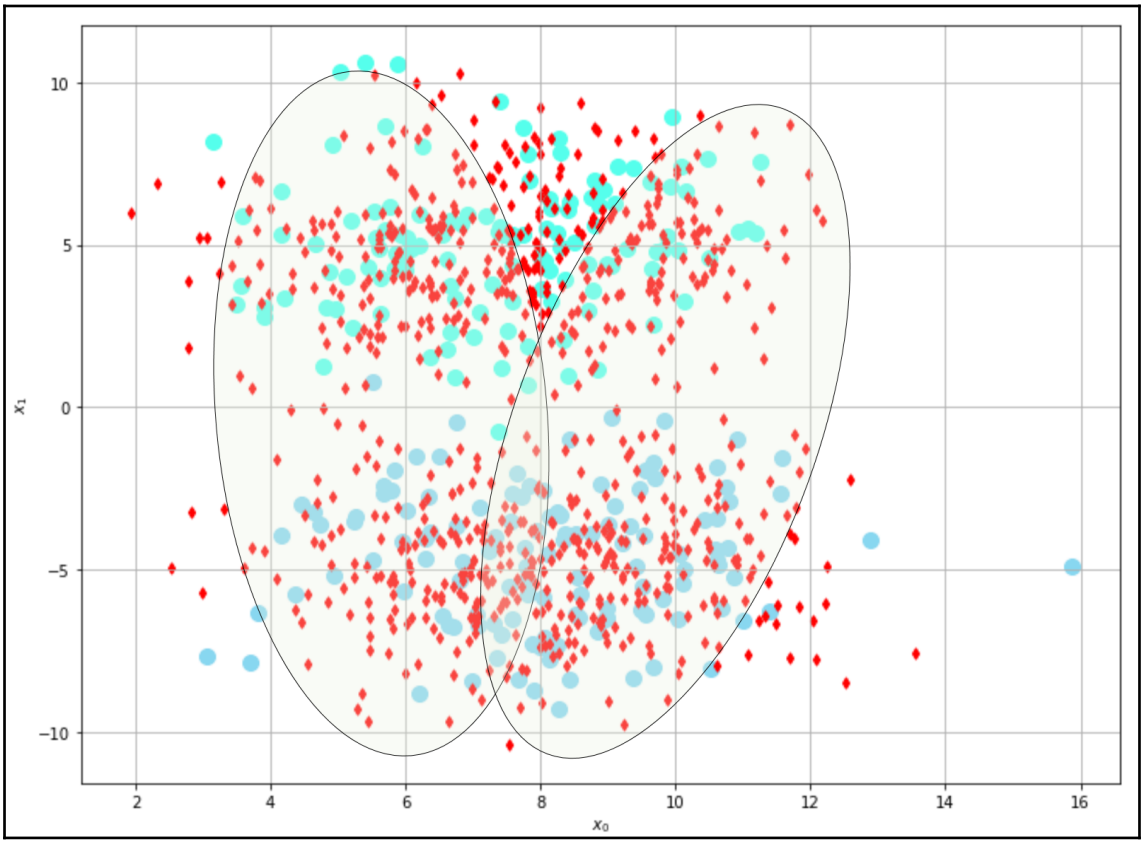
In all the examples that involve random numbers, the seed is set equal to 1,000 (`np.random.seed(1000)`). Other values or subsequent experiments without resetting it can yield slightly different results.

Weighted log-likelihood

In the previous example, we have considered a single log-likelihood for both labeled and unlabeled samples:

$$L(\bar{\theta}; \bar{w}) = \sum_j log \sum_i f_w(y_i|\bar{x}_j)p(\bar{x}_j|y_i, \bar{\theta}) = \sum_j log \sum_i w_i p(\bar{x}_j|y_i, \bar{\theta})$$

This is equivalent to saying that we trust the unlabeled points just like the labeled ones. However, in some contexts, this assumption can lead to completely wrong estimations, as shown in the following graph:



Biased final Gaussian mixture configuration

In this case, the means and covariance matrices of both Gaussian distributions have been biased by the unlabeled points and the resulting density estimation is clearly wrong. When this phenomenon happens, the best thing to do is to consider a double weighted log-likelihood. If the first *N* samples are labeled and the following *M* are unlabeled, the log-likelihood can be expressed as follows:

$$L(\bar{\theta};\bar{w}) = \sum_{j=1}^{N} log \sum_{i} p(y_i|\bar{\theta}) p(y_i|\bar{x}_j,\bar{\theta}) + \lambda \sum_{j=N+1}^{N+M} log \sum_{i} w_i p(\bar{x}_j|y_i,\bar{\theta})$$

In the previous formula, the term λ, if less than 1, can underweight the unlabeled terms, giving more importance to the labeled dataset. The modifications to the algorithm are trivial because each unlabeled weight has to be scaled according to λ, reducing its estimated probability. In *Semi-Supervised Learning*, *Chapelle O.*, *Schölkopf B.*, *Zien A.*, (*edited by*), *The MIT Press*, the reader can find a very detailed discussion about the choice of λ. There are no golden rules; however, a possible strategy could be based on the cross-validation performed on the labeled dataset. Another (more complex) approach is to consider different increasing values of λ and pick the first one where the log-likelihood is maximum. I recommend the aforementioned book for further details and strategies.

Contrastive pessimistic likelihood estimation

As explained at the beginning of this chapter, in many real life problems, it's cheaper to retrieve unlabeled samples, rather than correctly labeled ones. For this reason, many researchers worked to find out the best strategies to carry out a semi-supervised classification that could outperform the supervised counterpart. The idea is to train a classifier with a few labeled samples and then improve its accuracy after adding weighted unlabeled samples. One of the best results is the **Contrastive Pessimistic Likelihood Estimation** (**CPLE**) algorithm, proposed by M. Loog (in *Loog M.*, *Contrastive Pessimistic Likelihood Estimation for Semi-Supervised Classification*, *arXiv:1503.00269*).

Before explaining this algorithm, an introduction is necessary. If we have a labeled dataset (*X*, *Y*) containing *N* samples, it's possible to define the log-likelihood cost function of a generic estimator, as follows:

$$L(\bar{\theta};\bar{x},\bar{y}) = \sum_{i} log\, p(x_i,y_i|\bar{\theta})$$

After training the model, it should be possible to determine $p(y_i|x_i, \theta)$, which is the probability of a label given a sample x_i. However, some classifiers are not based on this approach (like SVM) and evaluate the right class, for example, by checking the sign of a parametrized function $f(x_i, \theta)$. As CPLE is a generic framework that can be used with any classification algorithm when the probabilities are not available, it's useful to implement a technique called Platt scaling, which allows transforming the decision function into a probability through a parametrized sigmoid. For a binary classifier, it can be expressed as follows:

$$p(y_i = +1|x_i, \bar{\theta}) = \frac{1}{1 + e^{\alpha f(x_i;\bar{\theta})+\beta}}$$

α and β are parameters that must be learned in order to maximize the likelihood. Luckily Scikit-Learn provides the method `predict_proba()`, which returns the probabilities for all classes. Platt scaling is performed automatically or on demand; for example, the SCV classifier needs to have the parameter `probability=True` in order to compute the probability mapping. I always recommend checking the documentation before implementing a custom solution.

We can consider a full dataset, made up of labeled and unlabeled samples. For simplicity, we can reorganize the original dataset, so that the first N samples are labeled, while the next M are unlabeled:

$$X_t = \left\{(\bar{x}_1, \bar{y}_1), (\bar{x}_2, \bar{y}_2), \ldots, (\bar{x}_N, \bar{y}_N), \bar{x}^u_{N+1}, \ldots, \bar{x}^u_{N+M})\right\}$$

As we don't know the labels for all x^u samples, we can decide to use M k-dimensional (k is the number of classes) soft-labels q_i that can be optimized during the training process:

$$Q = \{\bar{q}_1, \bar{q}_2, \ldots, \bar{q}_M\} \quad where \;\; \bar{q}_i \in \mathbb{R}^k \;\; and \;\; \sum_k q_i^{(k)} = 1$$

The second condition in the previous formula is necessary to guarantee that each q_i represents a discrete probability (all the elements must sum up to 1.0). The complete log-likelihood cost function can, therefore, be expressed as follows:

$$L(\bar{\theta}; X_t, Q) = L(\bar{\theta}; \bar{x}, \bar{y}) + \sum_{i=N+1}^{N+M} \sum_k q_i^{(k)} log\, p(\bar{x}_i^u, y_i^u = k|\bar{\theta})$$

The first term represents the log-likelihood for the supervised part, while the second one is responsible for the unlabeled points. If we train a classifier with only the labeled samples, excluding the second addend, we get a parameter set θ_{sup}. CPLE defines a contrastive condition (as a log-likelihood too), by defining the improvement in the total cost function given by the semi-supervised approach, compared to the supervised solution:

$$CL(\bar{\theta}, \bar{\theta}_{sup}, X_t, Q) = L(\bar{\theta}; X_t, Q) - L(\bar{\theta}_{sup}; X_t, Q)$$

This condition allows imposing that the semi-supervised solution must outperform the supervised one, in fact, maximizing it; we both increase the first term and reduce the second one, obtaining a proportional increase of CL (the term *contrastive* is very common in machine learning and it normally indicates a condition which is achieved as the difference between two opposite constraints). If CL doesn't increase, it probably means that the unlabeled samples have not been drawn from the marginal distribution *p(x)* extracted from p_{data}.

Moreover, in the previous expression, we have implicitly used soft-labels, but as they are initially randomly chosen and there's no ground truth to support their values, it's a good idea not to trust them by imposing a pessimistic condition (as another log-likelihood):

$$CPL(\bar{\theta}, \bar{\theta}_{sup}, X_t, Q) = min_q\ CL(\bar{\theta}, \bar{\theta}_{sup}, X_t, Q)$$

By imposing this constraint, we try to find the soft-labels that minimize the contrastive log-likelihood; that's why this is defined as a pessimistic approach. It can seem a contradiction; however, trusting soft-labels can be dangerous, because the semi-supervised log-likelihood could be increased even with a large percentage of misclassification. Our goal is to find the best parameter set that is able to guarantee the highest accuracy starting from the supervised baseline (which has been obtained using the labeled samples) and improving it, without forgetting the structural features provided by the labeled samples.

Therefore, our final goal can be expressed as follows:

$$\bar{\theta}_{semi} = max_{\bar{\theta}}\ CPL(\bar{\theta}, \bar{\theta}_{sup}, X_t, Q)$$

Example of contrastive pessimistic likelihood estimation

We are going to implement the CPLE algorithm in Python using a subset extracted from the MNIST dataset. For simplicity, we are going to use only the samples representing the digits 0 and 1:

```
from sklearn.datasets import load_digits

import numpy as np

X_a, Y_a = load_digits(return_X_y=True)

X = np.vstack((X_a[Y_a == 0], X_a[Y_a == 1]))
Y = np.vstack((np.expand_dims(Y_a, axis=1)[Y_a==0], np.expand_dims(Y_a,
axis=1)[Y_a==1]))

nb_samples = X.shape[0]
nb_dimensions = X.shape[1]
nb_unlabeled = 150
Y_true = np.zeros((nb_unlabeled,))

unlabeled_idx = np.random.choice(np.arange(0, nb_samples, 1),
replace=False, size=nb_unlabeled)
Y_true = Y[unlabeled_idx].copy()
Y[unlabeled_idx] = -1
```

After creating the restricted dataset (X, Y) which contain 360 samples, we randomly select 150 samples (about 42%) to become unlabeled (the corresponding y is -1). At this point, we can measure the performance of logistic regression trained only on the labeled dataset:

```
from sklearn.linear_model import LogisticRegression

lr_test = LogisticRegression()
lr_test.fit(X[Y.squeeze() != -1], Y[Y.squeeze() != -1].squeeze())
unlabeled_score = lr_test.score(X[Y.squeeze() == -1], Y_true)

print(unlabeled_score)
0.573333333333
```

So, the logistic regression shows 57% accuracy for the classification of the unlabeled samples. We can also evaluate the cross-validation score on the whole dataset (before removing some random labels):

```
from sklearn.model_selection import cross_val_score
```

```
total_cv_scores = cross_val_score(LogisticRegression(), X, Y.squeeze(),
cv=10)

print(total_cv_scores)
[ 0.48648649  0.51351351  0.5         0.38888889  0.52777778  0.36111111
  0.58333333  0.47222222  0.54285714  0.45714286]
```

Thus, the classifier achieves an average 48% accuracy when using 10 folds (each test set contains 36 samples) if all the labels are known.

We can now implement a CPLE algorithm. The first thing is to initialize a `LogisticRegression` instance and the soft-labels:

```
lr = LogisticRegression()
q0 = np.random.uniform(0, 1, size=nb_unlabeled)
```

q0 is a random array of values bounded in the half-open interval [0, 1]; therefore, we also need a converter to transform q_i into an actual binary label:

$$y(q) = \begin{cases} 0 \ \ if \ \ q < 0.5 \\ 1 \ \ otherwise \end{cases}$$

We can achieve this using the NumPy function `np.vectorize()`, which allows us to apply a transformation to all the elements of a vector:

```
trh = np.vectorize(lambda x: 0.0 if x < 0.5 else 1.0)
```

In order to compute the log-likelihood, we need also a weighted log-loss (similar to the Scikit-Learn function `log_loss()`, which, however, computes the negative log-likelihood but doesn't support weights):

```
def weighted_log_loss(yt, p, w=None, eps=1e-15):
    if w is None:
        w_t = np.ones((yt.shape[0], 2))
    else:
        w_t = np.vstack((w, 1.0 - w)).T
    Y_t = np.vstack((1.0 - yt.squeeze(), yt.squeeze())).T
    L_t = np.sum(w_t * Y_t * np.log(np.clip(p, eps, 1.0 - eps)), axis=1)
    return np.mean(L_t)
```

This function computes the following expression:

$$L(\bar{y}_i, \bar{p}, \bar{w}) = \frac{1}{N} \sum_i (y_{t_i} \ log(p_i) + (1 - y_{t_i}) log(1 - p_i))$$

We need also a function to build the dataset with variable soft-labels q_i:

```
def build_dataset(q):
    Y_unlabeled = trh(q)
    X_n = np.zeros((nb_samples, nb_dimensions))
    X_n[0:nb_samples - nb_unlabeled] = X[Y.squeeze()!=-1]
    X_n[nb_samples - nb_unlabeled:] = X[Y.squeeze()==-1]
    Y_n = np.zeros((nb_samples, 1))
    Y_n[0:nb_samples - nb_unlabeled] = Y[Y.squeeze()!=-1]
    Y_n[nb_samples - nb_unlabeled:] = np.expand_dims(Y_unlabeled, axis=1)
    return X_n, Y_n
```

At this point, we can define our contrastive log-likelihood:

```
def log_likelihood(q):
    X_n, Y_n = build_dataset(q)
    Y_soft = trh(q)
    lr.fit(X_n, Y_n.squeeze())
    p_sup = lr.predict_proba(X[Y.squeeze() != -1])
    p_semi = lr.predict_proba(X[Y.squeeze() == -1])
    l_sup = weighted_log_loss(Y[Y.squeeze() != -1], p_sup)
    l_semi = weighted_log_loss(Y_soft, p_semi, q)
    return l_semi - l_sup
```

This method will be called by the optimizer, passing a different q vector each time. The first step is building the new dataset and computing `Y_soft`, which are the labels corresponding to q. Then the logistic regression classifier is trained with with the dataset (as `Y_n` is a (k, 1) array, it's necessary to squeeze it to avoid a warning. The same thing is done when using Y as a boolean indicator). At this point, it's possible to compute both p_{sup} and p_{semi} using the method `predict_proba()` and, finally, we can compute the semi-supervised and supervised log-loss, which is the term, a function of q_i, that we want to minimize, while the maximization of θ is done implicitly when training the logistic regression.

The optimization is carried out using the BFGS algorithm implemented in SciPy:

```
from scipy.optimize import fmin_bfgs

q_end = fmin_bfgs(f=log_likelihood, x0=q0, maxiter=5000, disp=False)
```

This is a very fast algorithm, but the user is encouraged to experiment with methods or libraries. The two parameters we need in this case are `f`, which is the function to minimize, and `x0`, which is the initial condition for the independent variables. `maxiter` is useful for avoiding an excessive number of iterations when no improvements are achieved. Once the optimization is complete, `q_end` contains the optimal soft-labels. We can, therefore, rebuild our dataset:

```
X_n, Y_n = build_dataset(q_end)
```

With this final configuration, we can retrain the logistic regression and check the cross-validation accuracy:

```
final_semi_cv_scores = cross_val_score(LogisticRegression(), X_n,
Y_n.squeeze(), cv=10)

print(final_semi_cv_scores)
[ 1.          1.          0.89189189  0.77777778  0.97222222  0.88888889
  0.61111111  0.88571429  0.94285714  0.48571429]
```

The semi-supervised solution based on the CPLE algorithms achieves an average 84% accuracy, outperforming, as expected, the supervised approach. The reader can try other examples using different classifiers, such SVM or Decision Trees, and verify when CPLE allows obtaining higher accuracy than other supervised algorithms.

Semi-supervised Support Vector Machines (S^3VM)

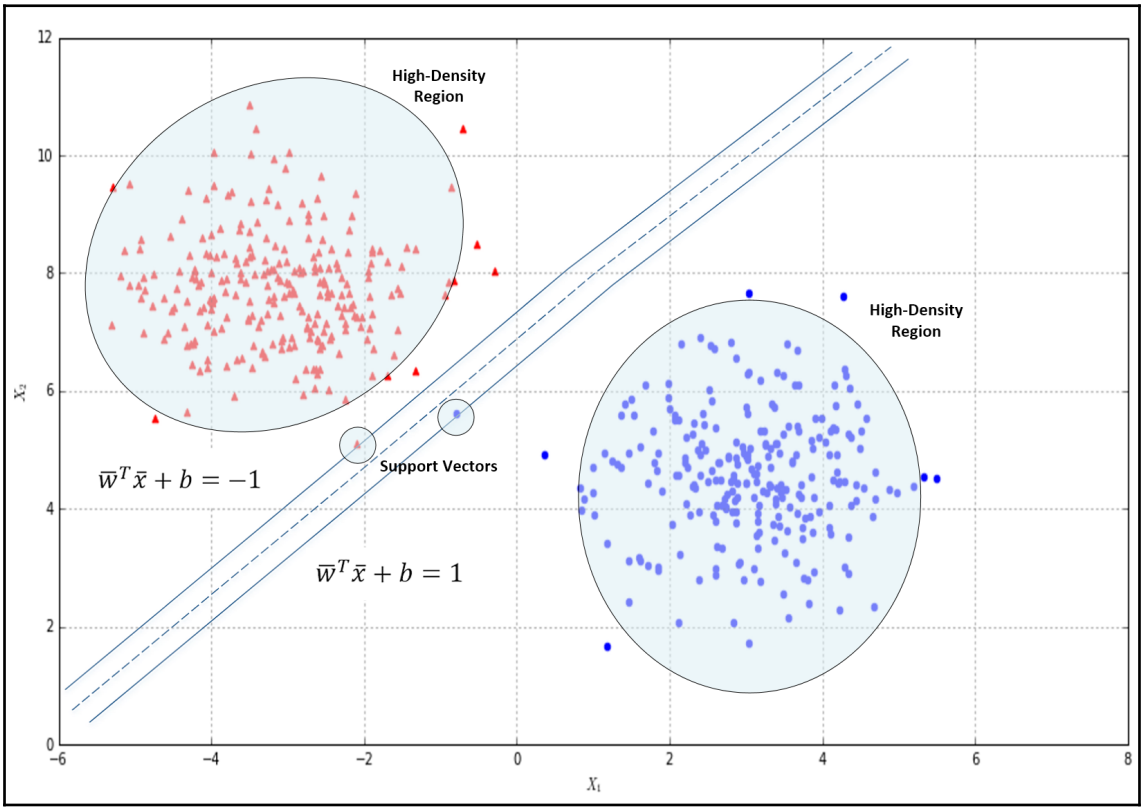
When we discussed the cluster assumption, we also defined the low-density regions as boundaries and the corresponding problem as low-density separation. A common supervised classifier which is based on this concept is a **Support Vector Machine** (**SVM**), the objective of which is to maximize the distance between the dense regions where the samples must be. For a complete description of linear and kernel-based SVMs, please refer to *Bonaccorso G.*, *Machine Learning Algorithms*, *Packt Publishing*; however, it's useful to remind yourself of the basic model for a linear SVM with slack variables ξ_i:

$$\begin{cases} min_{\bar{w},b,\xi}\ \frac{1}{2}\bar{w}^T\bar{w} + C\sum_i \xi_i \\ y_i(\bar{w}^T\bar{x}_i + b) \geqslant 1 - \xi_i\ \ and\ \ \xi_i \geqslant 0\ \ \forall\ \ i = 1..N \end{cases}$$

This model is based on the assumptions that y_i can be either -1 or 1. The slack variables ξ_i or soft-margins are variables, one for each sample, introduced to reduce the *strength* imposed by the original condition ($min\ ||w||$), which is based on a hard margin that misclassifies all the samples that are on the wrong side. They are defined by the Hinge loss, as follows:

$$\xi_i = max\left(0, 1 - y_i(\bar{w}^T \cdot \bar{x}_i + b)\right)$$

With those variables, we allow some points to overcome the limit without being misclassified if they remain within a distance controlled by the corresponding slack variable (which is also minimized during the training phase, so as to avoid uncontrollable growth). In the following diagram, there's a schematic representation of this process:



SVM generic scenario

The last elements of each high-density regions are the support vectors. Between them, there's a low-density region (it can also be zero-density in some cases) where our separating hyperplane lies. In `Chapter 1`, *Machine Learning Model Fundamentals*, we defined the concept of *empirical risk* as a proxy for expected risk; therefore, we can turn the SVM problem into the minimization of empirical risk under the Hinge cost function (with or without Ridge Regularization on w):

$$L(X, Y; \bar{w}, b) = \frac{1}{N} \sum_{i} max \left(0, 1 - y_i(\bar{w}^T \cdot \bar{x}_i + b)\right)$$

Theoretically, every function which is always bounded by two hyperplanes containing the support vectors is a good classifier, but we need to minimize the empirical risk (and, so, the expected risk); therefore we look for the maximum margin between high-density regions. This model is able to separate two dense regions with irregular boundaries and, by adopting a kernel function, also in non-linear scenarios. The natural question, at this point, is about the best strategy to integrate labeled and unlabeled samples when we need to solve this kind of problem in a semi-supervised scenario.

The first element to consider is the ratio: if we have a low percentage of unlabeled points, the problem is mainly supervised and the generalization ability learned using the training set should be enough to correctly classify all the unlabeled points. On the other hand, if the number of unlabeled samples is much larger, we return to an almost pure clustering scenario (like the one discussed in the paragraph about the Generative Gaussian mixtures). In order to exploit the strength of semi-supervised methods in low-density separation problems, therefore, we should consider situations where the ratio labeled/unlabeled is about 1.0. However, even if we have the predominance of a class (for example, if we have a huge unlabeled dataset and only a few labeled samples), it's always possible to use the algorithms we're going to discuss, even if, sometimes, their performance could be equal to or lower than a pure supervised/clustering solution. Transductive SMVs, for example, showed better accuracies when the labeled/unlabeled ratio is very small, while other methods can behave in a completely different way. However, when working with semi-supervised learning (and its assumptions), it is always important to bear in mind that each problem is supervised and unsupervised at the same time and the best solution must be evaluated in every different context.

A solution for this problem is offered by the *Semi-Supervised SVM* (also known as S^3VM) algorithm. If we have *N* labeled samples and *M* unlabeled samples, the objective function becomes as follows:

$$min_{\bar{w},b,\bar{\eta},\bar{\xi},\bar{z}} \left[\|\bar{w}\| + C \left(\sum_{i=1}^{N} \eta_i + \sum_{j=N+1}^{N+M} min(\xi_j, z_j) \right) \right]$$

The first term imposes the standard SVM condition about the maximum separation distance, while the second block is divided into two parts:

- We need to add *N* slack variables η_i to guarantee a soft-margin for the labeled samples.
- At the same time, we have to consider the unlabeled points, which could be classified as +1 or -1. Therefore, we have two corresponding slack-variable sets ξ_i and z_i. However, we want to find the smallest variable for each possible pair, so as to be sure that the unlabeled sample is placed on the sub-space where the maximum accuracy is achieved.

The constraints necessary to solve the problems become as follows:

$$\begin{cases} y_i(\bar{w}^T \cdot x_i + b) \geqslant 1 - \eta_i \;\; and \;\; \eta_i \geqslant 0 \;\; \forall \;\; i = 1..N \\ (\bar{w}^T \cdot \bar{x}_j - b) \geqslant 1 - \xi_j \;\; and \;\; \xi_j \geqslant 0 \;\; \forall \;\; j = N+1 \ldots N+M \\ -(\bar{w}^T \cdot \bar{x}_j - b) \geqslant 1 - z_j \;\; and \;\; z_j \geqslant 0 \;\; \forall \;\; j = N+1 \ldots N+M \end{cases}$$

The first constraint is limited to the labeled points and it's the same as a supervised SVM. The following two, instead, take into account the possibility that an unlabeled sample could be classified as +1 or -1. Let's suppose, for example, that the label y_j for the sample x_j should be +1 and the first member of the second inequality is a positive number *K* (so the corresponding term of the third equation is *-K*). It's easy to verify that the first slack variable is $\xi_i \geq 1 - K$, while the second one is $z_j \geq 1 + K$.

Therefore, in the objective, ξ_i is chosen to be minimized. This method is inductive and yields good (if not excellent) performances; however, it has a very high computational cost and should be solved using optimized (native) libraries. Unfortunately, it is a non-convex problem and there are no standard methods to solve it so it always reaches the optimal configuration.

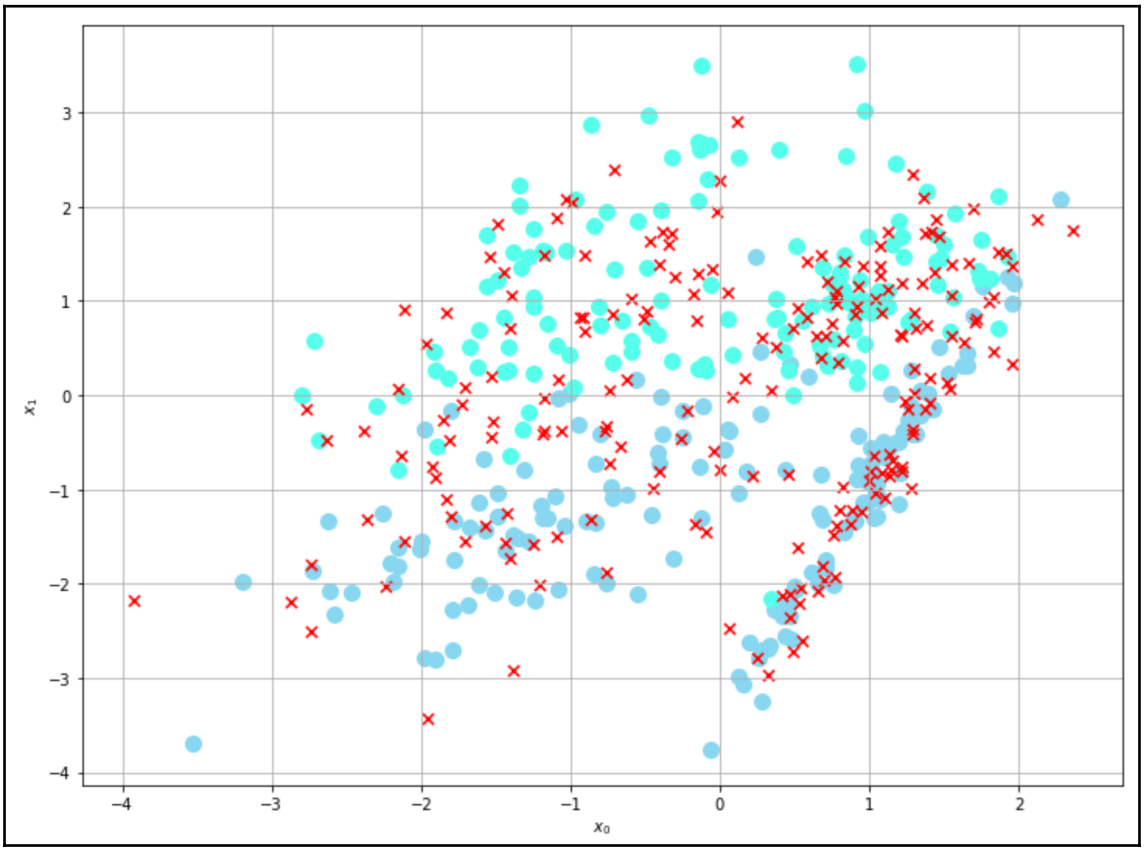
Example of S^3VM

We now implement an S^3VM in Python using the SciPy optimization methods, which are mainly based on C and FORTRAN implementations. The reader can try it with other libraries such as NLOpt and LIBSVM and compare the results. A possibility suggested by Bennet and Demiriz is to use the L1-norm for w, so as to linearize the objective function; however, this choice seems to produce good results only for small datasets. We are going to keep the original formulation based on the L2-norm, using an **Sequential Least Squares Programming** (**SLSQP**) algorithm to optimize the objective.

Let's start by creating a bidimensional dataset with both labeled and unlabeled samples:

```
from sklearn.datasets import make_classification

nb_samples = 500
nb_unlabeled = 200

X, Y = make_classification(n_samples=nb_samples, n_features=2,
n_redundant=0, random_state=1000)
Y[Y==0] = -1
Y[nb_samples - nb_unlabeled:nb_samples] = 0
```

For simplicity (and without any impact, because the samples are shuffled), we set last 200 samples as unlabeled ($y = 0$). The corresponding plot is shown in the following graph:



Original labeled and unlabeled dataset

The crosses represent unlabeled points, which are spread throughout the entire dataset. At this point we need to initialize all variables required for the optimization problem:

```
import numpy as np

w = np.random.uniform(-0.1, 0.1, size=X.shape[1])
eta = np.random.uniform(0.0, 0.1, size=nb_samples - nb_unlabeled)
xi = np.random.uniform(0.0, 0.1, size=nb_unlabeled)
zi = np.random.uniform(0.0, 0.1, size=nb_unlabeled)
b = np.random.uniform(-0.1, 0.1, size=1)
C = 1.0

theta0 = np.hstack((w, eta, xi, zi, b))
```

As the optimization algorithm requires a single array, we have stacked all vectors into a horizontal array `theta0` using the `np.hstack()` function. We also need to vectorize the `min()` function in order to apply it to arrays:

```
vmin = np.vectorize(lambda x1, x2: x1 if x1 <= x2 else x2)
```

Now, we can define the objective function:

```
def svm_target(theta, Xd, Yd):
    wt = theta[0:2].reshape((Xd.shape[1], 1))
    s_eta = np.sum(theta[2:2 + nb_samples - nb_unlabeled])
    s_min_xi_zi = np.sum(vmin(theta[2 + nb_samples - nb_unlabeled:2 +
nb_samples],
                              theta[2 + nb_samples:2 + nb_samples +
nb_unlabeled]))
    return C * (s_eta + s_min_xi_zi) + 0.5 * np.dot(wt.T, wt)
```

The arguments are the current `theta` vector and the complete datasets `Xd` and `Yd`. The dot product of w has been multiplied by 0.5 to keep the conventional notation used for supervised SVMs. The constant can be omitted without any impact. At this point, we need to define all the constraints, as they are based on the slack variables; each function (which shares the same parameters of the objectives) is parametrized with an index, `idx`. The labeled constraint is as follows:

```
def labeled_constraint(theta, Xd, Yd, idx):
    wt = theta[0:2].reshape((Xd.shape[1], 1))
    c = Yd[idx] * (np.dot(Xd[idx], wt) + theta[-1]) + \
    theta[2:2 + nb_samples - nb_unlabeled][idx] - 1.0
    return (c >= 0)[0]
```

The unlabeled constraints, instead, are as follows:

```
def unlabeled_constraint_1(theta, Xd, idx):
    wt = theta[0:2].reshape((Xd.shape[1], 1))
    c = np.dot(Xd[idx], wt) - theta[-1] + \
        theta[2 + nb_samples - nb_unlabeled:2 + nb_samples][idx -
nb_samples + nb_unlabeled] - 1.0
    return (c >= 0)[0]

def unlabeled_constraint_2(theta, Xd, idx):
    wt = theta[0:2].reshape((Xd.shape[1], 1))
    c = -(np.dot(Xd[idx], wt) - theta[-1]) + \
        theta[2 + nb_samples:2 + nb_samples + nb_unlabeled ][idx -
nb_samples + nb_unlabeled] - 1.0
    return (c >= 0)[0]
```

They are parametrized with the current `theta` vector, the `Xd` dataset, and an `idx` index. We need also to include the constraints for each slack variable (≥ 0):

```
def eta_constraint(theta, idx):
    return theta[2:2 + nb_samples - nb_unlabeled][idx] >= 0

def xi_constraint(theta, idx):
    return theta[2 + nb_samples - nb_unlabeled:2 + nb_samples][idx -
nb_samples + nb_unlabeled] >= 0

def zi_constraint(theta, idx):
    return theta[2 + nb_samples:2 + nb_samples+nb_unlabeled ][idx -
nb_samples + nb_unlabeled] >= 0
```

We can now set up the problem using the SciPy convention:

```
svm_constraints = []

for i in range(nb_samples - nb_unlabeled):
    svm_constraints.append({
            'type': 'ineq',
            'fun': labeled_constraint,
            'args': (X, Y, i)
        })
    svm_constraints.append({
            'type': 'ineq',
            'fun': eta_constraint,
            'args': (i,)
        })
for i in range(nb_samples - nb_unlabeled, nb_samples):
    svm_constraints.append({
            'type': 'ineq',
            'fun': unlabeled_constraint_1,
            'args': (X, i)
        })
    svm_constraints.append({
            'type': 'ineq',
            'fun': unlabeled_constraint_2,
            'args': (X, i)
        })
    svm_constraints.append({
            'type': 'ineq',
            'fun': xi_constraint,
            'args': (i,)
        })
    svm_constraints.append({
            'type': 'ineq',
            'fun': zi_constraint,
            'args': (i,)
        })
```

Each constraint is represented with a dictionary, where `type` is set to `ineq` to indicate that it is an inequality, `fun` points to the callable object and `args` contains all extra arguments (`theta` is the main x variable and it's automatically added). Using SciPy, it's possible to minimize the objective using the **Sequential Least Squares Programming** (**SLSQP**) or **Constraint Optimization by Linear Approximation** (**COBYLA**) algorithms. We preferred the former, because it works more rapidly and is more stable:

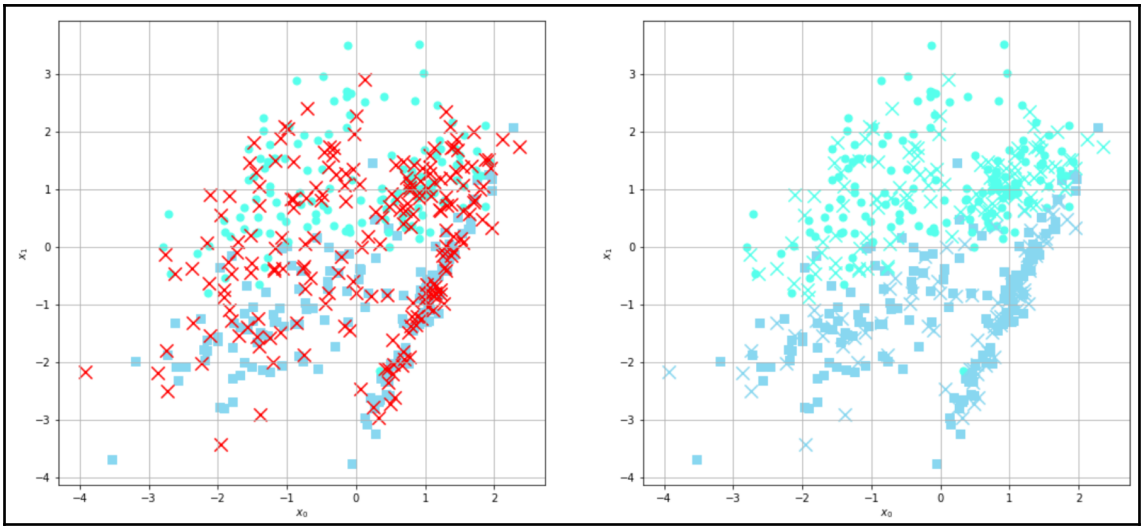
```
from scipy.optimize import minimize

result = minimize(fun=svm_target,
```

```
                x0=theta0,
                constraints=svm_constraints,
                args=(X, Y),
                method='SLSQP',
                tol=0.0001,
                options={'maxiter': 1000})
```

After the training process is complete, we can compute the labels for the unlabeled points:

```
theta_end = result['x']
w = theta_end[0:2]
b = theta_end[-1]

Xu= X[nb_samples - nb_unlabeled:nb_samples]
yu = -np.sign(np.dot(Xu, w) + b)
```

In the next graph, it's possible to compare the initial plot (left) with the final one where all points have been assigned a label (right):



As you can see, S^{3}VM succeeded in finding the right label for all unlabeled points, confirming the existence of two very dense regions for *x* between *[0, 2]* (square dots) and *y* between *[0, 2]* (circular dots).

NLOpt is a complete optimization library developed at MIT. It is available for different operating systems and programming languages. The website is `https://nlopt.readthedocs.io`. LIBSVM is an optimized library for solving SVM problems and it is adopted by Scikit-Learn together with LIBLINEAR. It's also available for different environments. The homepage is `https://www.csie.ntu.edu.tw/~cjlin/libsvm/`.

Transductive Support Vector Machines (TSVM)

Another approach to the same problem is offered by the TSVM, proposed by T. Joachims (in *Transductive Inference for Text Classification using Support Vector Machines*, *Joachims T., ICML Vol. 99/1999*). The idea is to keep the original objective with two sets of slack variables: the first for the labeled samples and the other for the unlabeled ones:

$$min_{\bar{w},b,\bar{\eta},\bar{\xi}} \left[\|\bar{w}\| + C_L \sum_{i=1}^{N} \eta_i + C_U \sum_{j=N+1}^{N+M} \xi_j \right]$$

As this is a transductive approach, we need to consider the unlabeled samples as variable-labeled ones (subject to the learning process), imposing a constraint similar to the supervised points. As for the previous algorithm, we assume we have *N* labeled samples and *M* unlabeled ones; therefore, the conditions become as follows:

$$\begin{cases} y_i(\bar{w}^T \cdot x_i + b) \geqslant 1 - \eta_i \;\; and \;\; \eta_i \geqslant 0 \;\; \forall \;\; i = 1..N \\ y_j^{(u)}(\bar{w}^T \cdot x_j + b) \geqslant 1 - \xi_j \;\; and \;\; \xi_j \geqslant 0 \;\; \forall j = N+1, \ldots, N+M \\ y_j^{(u)} \in \{-1, 1\} \end{cases}$$

The first constraint is the classical SVM one and it works only on labeled samples. The second one uses the variable $y^{(u)}_j$ with the corresponding slack variables ξ_j to impose a similar condition on the unlabeled samples, while the third one is necessary to constrain the labels to being equal to -1 and 1.

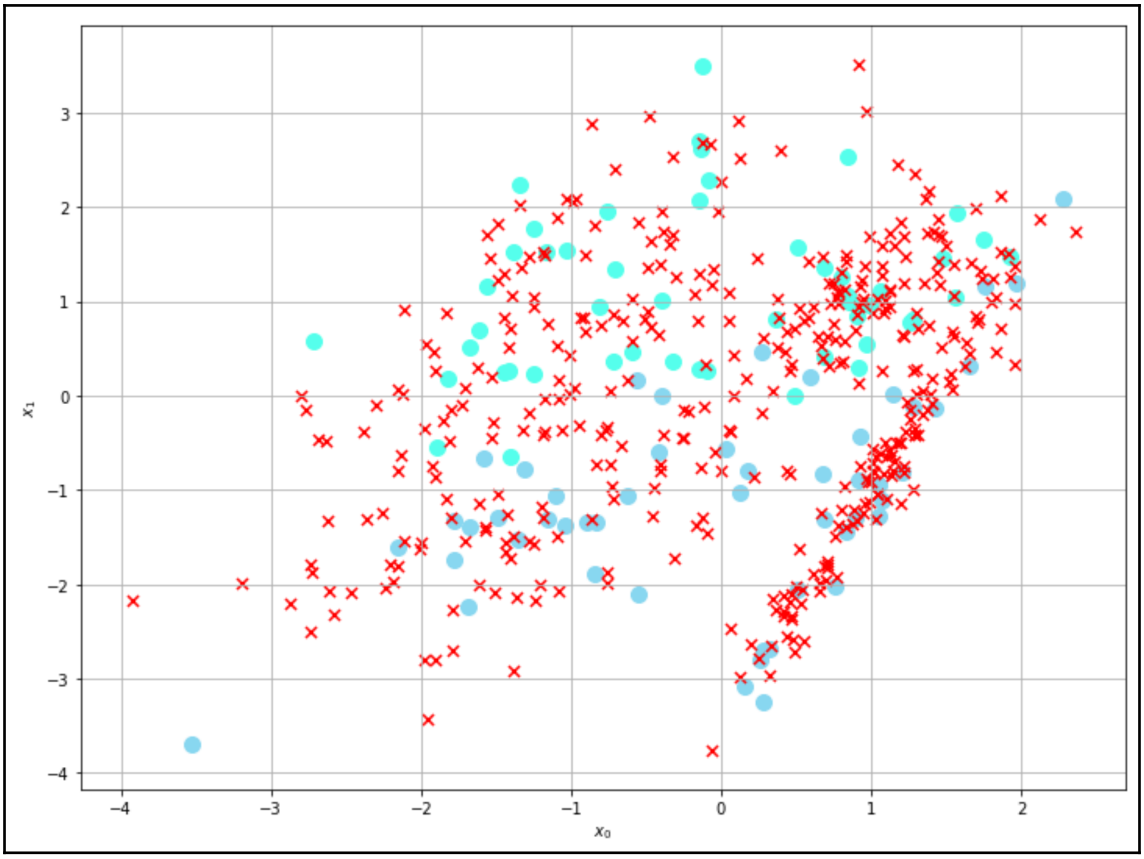
Just like the semi-supervised SVMs, this algorithm is non-convex and it's useful to try different methods to optimize it. Moreover, the author, in the aforementioned paper, showed how TSVM works better when the test set (unlabeled) is large and the training set (labeled) is relatively small (when a standard supervised SVM is outperformed). On the other hand, with large training sets and small test sets, a supervised SVM (or other algorithms) are always preferable because they are faster and yield better accuracy.

Example of TSVM

In our Python implementation, we are going to use a bidimensional dataset similar to one employed in the previous method; however, in this case, we impose 400 unlabeled samples out of a total of 500 points:

```
from sklearn.datasets import make_classification

nb_samples = 500
nb_unlabeled = 400

X, Y = make_classification(n_samples=nb_samples, n_features=2,
n_redundant=0, random_state=1000)
Y[Y==0] = -1
Y[nb_samples - nb_unlabeled:nb_samples] = 0
```

The corresponding plot is shown in the following graph:



Original labeled and unlabeled dataset

The procedure is similar to the one we used before. First of all, we need to initialize our variables:

```
import numpy as np

w = np.random.uniform(-0.1, 0.1, size=X.shape[1])
eta_labeled = np.random.uniform(0.0, 0.1, size=nb_samples - nb_unlabeled)
eta_unlabeled = np.random.uniform(0.0, 0.1, size=nb_unlabeled)
y_unlabeled = np.random.uniform(-1.0, 1.0, size=nb_unlabeled)
b = np.random.uniform(-0.1, 0.1, size=1)

C_labeled = 1.0
C_unlabeled = 10.0

theta0 = np.hstack((w, eta_labeled, eta_unlabeled, y_unlabeled, b))
```

In this case, we also need to define the `y_unlabeled` vector for variable-labels. The author also suggests using two C constants (`C_labeled` and `C_unlabeled`) in order to be able to weight the misclassification of labeled and unlabeled samples differently. We used a value of 1.0 for `C_labeled` and 10.0 for `C_unlabled`, because we want to penalize more the misclassification of unlabeled samples.

The objective function to optimize is as follows:

```
def svm_target(theta, Xd, Yd):
    wt = theta[0:2].reshape((Xd.shape[1], 1))
    s_eta_labeled = np.sum(theta[2:2 + nb_samples - nb_unlabeled])
    s_eta_unlabeled = np.sum(theta[2 + nb_samples - nb_unlabeled:2 +
nb_samples])
    return (C_labeled * s_eta_labeled) + (C_unlabeled * s_eta_unlabeled) +
(0.5 * np.dot(wt.T, wt))
```

While the labeled and unlabeled constraints are as follows:

```
def labeled_constraint(theta, Xd, Yd, idx):
    wt = theta[0:2].reshape((Xd.shape[1], 1))
    c = Yd[idx] * (np.dot(Xd[idx], wt) + theta[-1]) + \
    theta[2:2 + nb_samples - nb_unlabeled][idx] - 1.0
    return (c >= 0)[0]

def unlabeled_constraint(theta, Xd, idx):
    wt = theta[0:2].reshape((Xd.shape[1], 1))
    c = theta[2 + nb_samples:2 + nb_samples + nb_unlabeled][idx -
nb_samples + nb_unlabeled] * \
        (np.dot(Xd[idx], wt) + theta[-1]) + \
        theta[2 + nb_samples - nb_unlabeled:2 + nb_samples][idx -
nb_samples + nb_unlabeled] - 1.0
    return (c >= 0)[0]
```

We need also to impose the constraints on the slack variables and on the $y^{(u)}$:

```
def eta_labeled_constraint(theta, idx):
    return theta[2:2 + nb_samples - nb_unlabeled][idx] >= 0

def eta_unlabeled_constraint(theta, idx):
    return theta[2 + nb_samples - nb_unlabeled:2 + nb_samples][idx -
nb_samples + nb_unlabeled] >= 0

def y_constraint(theta, idx):
    return np.power(theta[2 + nb_samples:2 + nb_samples +
nb_unlabeled][idx], 2) == 1.0
```

As in the previous example, we can create the constraint dictionary needed by SciPy:

```
svm_constraints = []

for i in range(nb_samples - nb_unlabeled):
    svm_constraints.append({
            'type': 'ineq',
            'fun': labeled_constraint,
            'args': (X, Y, i)
        })
    svm_constraints.append({
            'type': 'ineq',
            'fun': eta_labeled_constraint,
            'args': (i,)
        })
for i in range(nb_samples - nb_unlabeled, nb_samples):
    svm_constraints.append({
            'type': 'ineq',
            'fun': unlabeled_constraint,
            'args': (X, i)
        })
    svm_constraints.append({
            'type': 'ineq',
            'fun': eta_unlabeled_constraint,
            'args': (i,)
        })

for i in range(nb_unlabeled):
    svm_constraints.append({
            'type': 'eq',
            'fun': y_constraint,
            'args': (i,)
        })
```

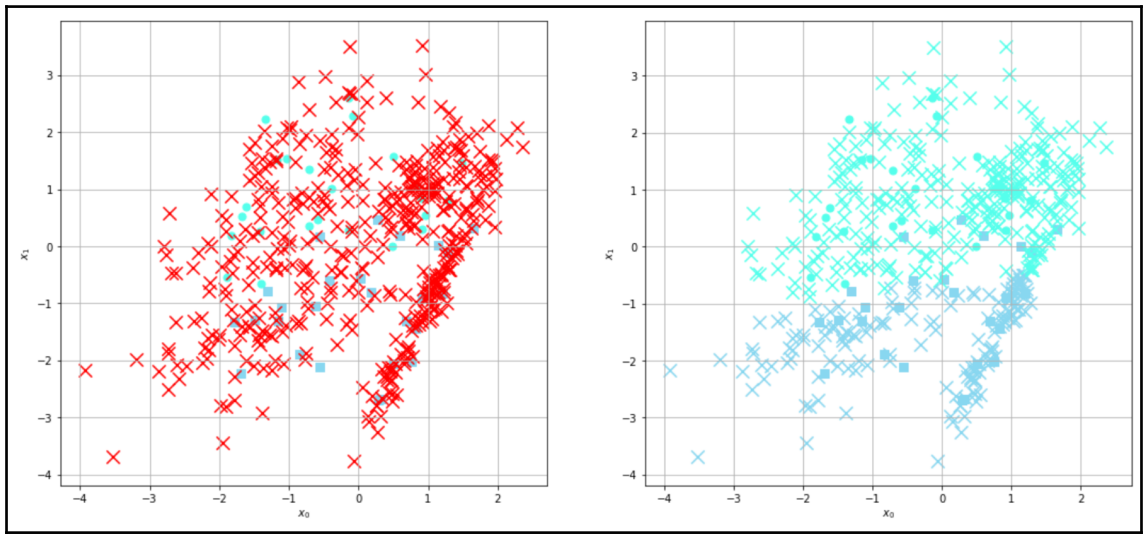
In this case, the last constraint is an equality, because we want to force $y^{(u)}$ to be equal either to -1 or 1. At this point, we minimize the objective function:

```
from scipy.optimize import minimize

result = minimize(fun=svm_target,
                  x0=theta0,
                  constraints=svm_constraints,
                  args=(X, Y),
                  method='SLSQP',
                  tol=0.0001,
                  options={'maxiter': 1000})
```

When the process is complete, we can compute the labels for the unlabeled samples and compare the plots:

```
theta_end = result['x']
w = theta_end[0:2]
b = theta_end[-1]

Xu= X[nb_samples - nb_unlabeled:nb_samples]
yu = -np.sign(np.dot(Xu, w) + b)
```

The plot comparison is shown in the following graph:



Original dataset (left). Final labeled dataset (right)

The misclassification (based on the density distribution) is slightly higher than S^3VM, but it's possible to change the C values and the optimization method until the expected result has been reached. A good benchmark is provided by a supervised SVM, which can have better performances when the training set is huge enough (and when it represents the whole p_{data} correctly).

It's interesting to evaluate different combinations of the C parameters, starting from a standard supervised SVM. The dataset is smaller, with a high number of unlabeled samples:

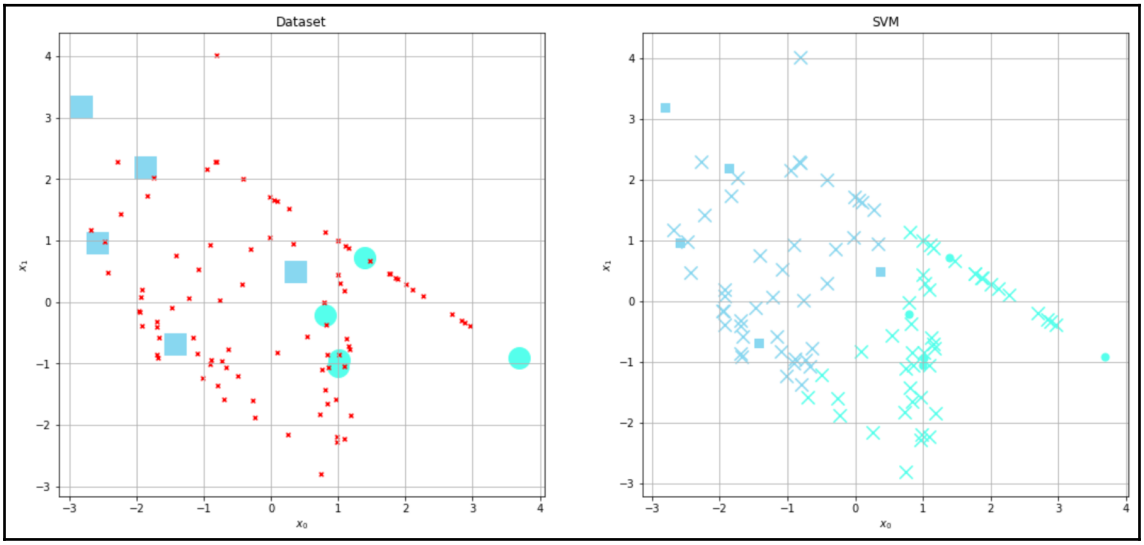
```
nb_samples = 100
nb_unlabeled = 90
```

```
X, Y = make_classification(n_samples=nb_samples, n_features=2,
n_redundant=0, random_state=100)
Y[Y==0] = -1
Y[nb_samples - nb_unlabeled:nb_samples] = 0
```

We use the standard SVM implementation provided by Scikit-Learn (the `SVC()` class) with a linear kernel and `C=1.0`:

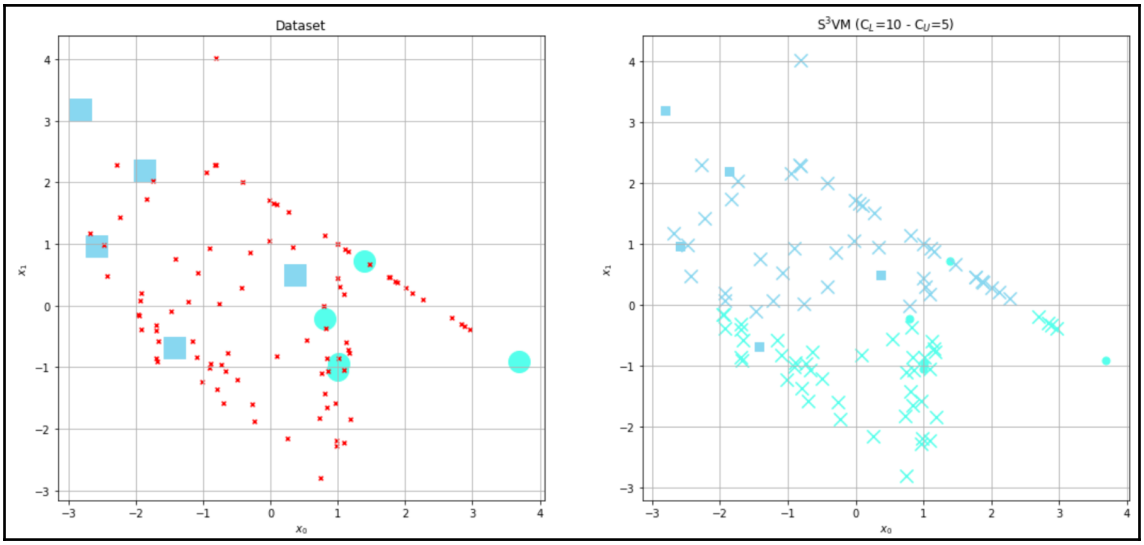
```
from sklearn.svm import SVC

svc = SVC(kernel='linear', C=1.0)
svc.fit(X[Y!=0], Y[Y!=0])

Xu_svc= X[nb_samples - nb_unlabeled:nb_samples]
yu_svc = svc.predict(Xu_svc)
```

The SVM is trained with the labeled samples and the vector `yu_svc` contains the prediction for the unlabeled samples. The resulting plot (in comparison with the original dataset) is shown in the following graph:



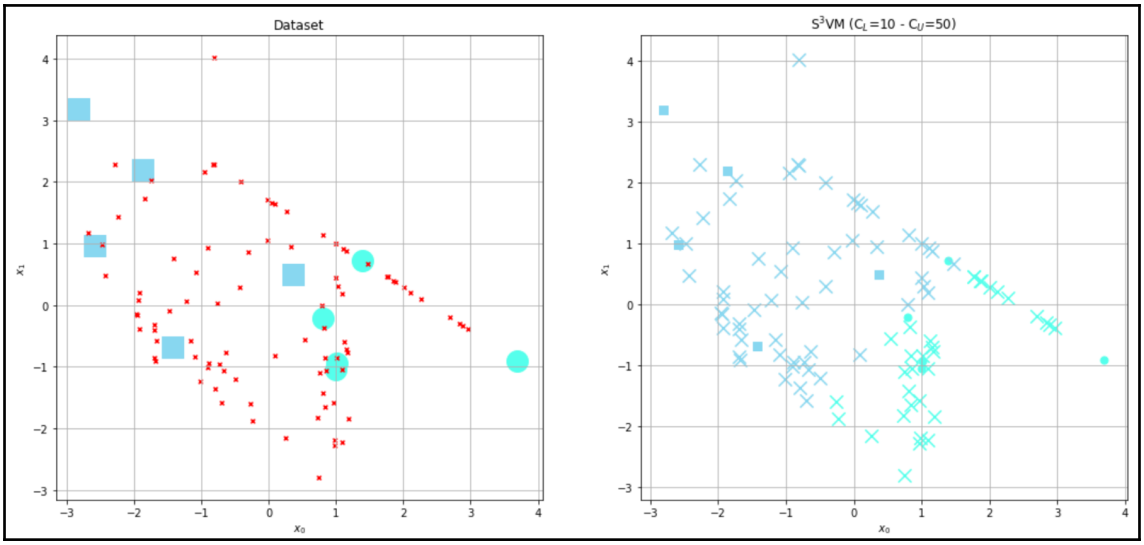
Original dataset (left). Final labeled dataset (right) with C = 1.0

All the labeled samples are represented with bigger squares and circles. The result meets our expectations, but there's an area *(X [-1, 0] - Y [-2, -1])*, where the SVM decided to impose the *circle* class even if the unlabeled points are close to a square. This hypothesis can't be acceptable considering the clustering assumption; in fact, in a high-density region there are samples belonging to two classes. A similar (or even worse) result is obtained using an S^3VM with **CL=10** and **CU=5**:



Original dataset (left). Final labeled dataset (right) with C_L = 10 and C_U = 5

In this case, the classification accuracy is lower because the penalty for the unlabeled samples is lower than the one imposed on the labeled points. A supervised SVM has obviously better performances. Let's try with **C_L=10** and **C_U=50**:



Original dataset (left). Final labeled dataset (right) with $C_L = 10$ and $C_U = 50$

Now, the penalty is quite a lot higher for the unlabeled samples and the result appears much more reasonable considering the clustering assumption. All the high-density regions are coherent and separated by low-density ones. These examples show how the value chosen for the parameters and the optimization method can dramatically change the result. My suggestion is to test several configurations (on sub-sampled datasets), before picking the final one. In *Semi-Supervised Learning*, *Chapelle O.*, *Schölkopf B.*, *Zien A.*, (*edited by*), *The MIT Press*, there are further details about possible optimization strategies, with strengths and weaknesses.

Summary

In this chapter, we introduced semi-supervised learning, starting from the scenario and the assumptions needed to justify the approaches. We discussed the importance of the smoothness assumption when working with both supervised and semi-supervised classifiers in order to guarantee a reasonable generalization ability. Then we introduced the clustering assumption, which is strictly related to the geometry of the datasets and allows coping with density estimation problems with a strong structural condition. Finally, we discussed the manifold assumption and its importance in order to avoid the curse of dimensionality.

The chapter continued by introducing a generative and inductive model: Generative Gaussian mixtures, which allow clustering labeled and unlabeled samples starting from the assumption that the prior probabilities are modeled by multivariate Gaussian distributions.

The next topic was about a very important algorithm: contrastive pessimistic likelihood estimation, which is an inductive, semi-supervised classification framework that can be adopted together with any supervised classifier. The main concept is to define a contrastive log-likelihood based on soft-labels (representing the probabilities for the unlabeled samples) and impose a pessimistic condition in order to minimize the trust in the soft-labels. The algorithm can find the best configuration that maximizes the log-likelihood, taking into account both labeled and unlabeled samples.

Another inductive classification approach is provided by the S^3VM algorithm, which is an extension of the classical SVM approach, based on two extra optimization constraints to address the unlabeled samples. This method is relatively powerful, but it's non-convex and, therefore, very sensitive to the algorithms employed to minimize the objective function.

An alternative to S^3VM is provided by the TSVM, which tries to minimize the objective with a condition based on variable labels. The problem is, hence, divided into two parts: the supervised one, which is exactly the same as standard SVM, and the semi-supervised one, which has a similar structure but without fixed y labels. This problem is non-convex too and it's necessary to evaluate different optimization strategies to find the best trade-off between accuracy and computational complexity. In the reference section, there are some useful resources so you can examine all these problems in depth and find a suitable solution for each particular scenario.

In the next chapter, `Chapter 3`, *Graph-Based Semi-Supervised Learning* we're continuing this exploration by discussing some important algorithms based on the structure underlying the dataset. In particular, we're going to employ graph theory to perform the propagation of labels to unlabeled samples and to reduce the dimensionality of datasets in non-linear contexts.

3
Graph-Based Semi-Supervised Learning

In this chapter, we continue our discussion about semi-supervised learning, considering a family of algorithms that is based on the graph obtained from the dataset and the existing relationships among samples. The problems that we are going to discuss belong to two main categories: the propagation of class labels to unlabeled samples and the use of non-linear techniques based on the manifold assumption to reduce the dimensionality of the original dataset. In particular, this chapter covers the following propagation algorithms:

- Label propagation based on the weight matrix
- Label propagation in Scikit-Learn (based on transition probabilities)
- Label spreading
- Propagation based on Markov random walks

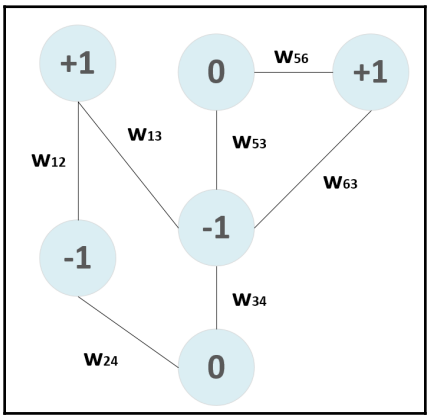
For the manifold learning section, we're discussing:

- Isomap algorithm and multidimensional scaling approach
- Locally linear embedding
- Laplacian Spectral Embedding
- t-SNE

Label propagation

Label propagation is a family of semi-supervised algorithms based on a graph representation of the dataset. In particular, if we have *N* labeled points (with bipolar labels +1 and -1) and *M* unlabeled points (denoted by *y=0*), it's possible to build an undirected graph based on a measure of geometric affinity among samples. If *G = {V, E}* is the formal definition of the graph, the set of vertices is made up of sample labels *V = { -1, +1, 0 }*, while the edge set is based on an **affinity matrix** *W* (often called **adjacency matrix** when the graph is unweighted), which depends only on the *X* values, not on the labels.

In the following graph, there's an example of such a structure:



Example of binary graph

In the preceding example graph, there are four labeled points (two with *y=+1* and two with *y=-1*), and two unlabeled points (*y=0*). The affinity matrix is normally symmetric and square with dimensions equal to *(N+M) x (N+M)*. It can be obtained with different approaches. The most common ones, also adopted by Scikit-Learn, are:

- ***k*-Nearest Neighbors** (we are going to discuss this algorithm with further details in `Chapter 8`, *Clustering Algorithms*):

$$W_{ij} = \begin{cases} 1 \;\; if \;\; \bar{x}_i \in \; kNN(\bar{x}_j) \\ 0 \;\; otherwise \end{cases}$$

- **Radial basis function kernel**:

$$W_{ij} = e^{-\gamma \| \bar{x}_i - \bar{x}_j \|^2}$$

Sometimes, in the radial basis function kernel, the parameter γ is represented as the reciprocal of $2\sigma^2$; however, small γ values corresponding to a large variance increase the radius, including farther points and *smoothing* the class over a number of samples, while large γ values restrict the boundaries to a subset that tends to a single sample. Instead, in the k-nearest neighbors kernel, the parameter *k* controls the number of samples to consider as neighbors.

To describe the basic algorithm, we also need to introduce the **degree matrix** (*D*):

$$D = diag\left(\left| \sum_j W_{ij} \right| \;\; \forall \;\; i = 1..N+M \right) = \begin{pmatrix} deg(v_1) & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & deg(v_{N+M}) \end{pmatrix}$$

It is a diagonal matrix where each non-null element represents the *degree* of the corresponding vertex. This can be the number of incoming edges, or a measure proportional to it (as in the case of *W* based on the radial basis function). The general idea of label propagation is to let each node propagate its label to its neighbors and iterate the procedure until convergence.

Formally, if we have a dataset containing both labeled and unlabeled samples:

$$\begin{cases} X = \{\bar{x}_0, \bar{x}_1, \ldots, \bar{x}_N, \bar{x}_{N+1}, \ldots, \bar{x}_{N+M}\} \;\; where \;\; \bar{x}_i \in \mathbb{R}^k \\ Y = \{y_0, y_1, \ldots, y_N, 0, 0, \ldots, 0\} \;\; where \;\; y_i \in \{0, +1, -1\} \end{cases}$$

The complete steps of the **label propagation** algorithm (as proposed by Zhu and Ghahramani in *Learning from Labeled and Unlabeled Data with Label Propagation, Zhu X., Ghahramani Z., CMU-CALD-02-107*) are:

1. Select an affinity matrix type (KNN or RBF) and compute W
2. Compute the degree matrix D
3. Define $Y^{(0)} = Y$
4. Define $Y_L = \{y_0, y_1, ..., y_N\}$
5. Iterate until convergence of the following steps:

$$\begin{cases} \widetilde{Y}^{(t+1)} = D^{-1}W\widetilde{Y}^{(t)} \\ \widetilde{Y}_L^{(t+1)} = Y_L \end{cases}$$

The first update performs a propagation step with both labeled and unlabeled points. Each label is spread from a node through its outgoing edges, and the corresponding weight, normalized with the degree, increases or decreases the *effect* of each contribution. The second command instead resets all y values for the labeled samples. The final labels can be obtained as:

$$Y_{Final} = sign(\widetilde{Y}^{(t_{end})})$$

The proof of convergence is very easy. If we partition the matrix $D^{-1}W$ according to the relationship among labeled and unlabeled samples, we get:

$$D^{-1}W = \begin{pmatrix} A_{LL} & A_{LU} \\ A_{UL} & A_{UU} \end{pmatrix}$$

If we consider that only the first N components of Y are non-null and they are clamped at the end of each iteration, the matrix can be rewritten as:

$$D^{-1}W = \begin{pmatrix} A_{LL} & A_{LU} \\ A_{UL} & A_{UU} \end{pmatrix} = \begin{pmatrix} I & 0 \\ A_{UL} & A_{UU} \end{pmatrix}$$

We are interested in proving the convergence for the part regarding the unlabeled samples (the labeled ones are fixed), so we can write the update rule as:

$$\widetilde{Y}_U^{(t+1)} = A_{UL}Y_L + A_{UU}\widetilde{Y}_U^{(t)}$$

Transforming the recursion into an iterative process, the previous formula becomes:

$$\widetilde{Y}_U^{(t+1)} = \sum_{k=1}^{t+1}\left((A_{UU})^{k-1}A_{UL}Y_L\right) + (A_{UU})^{t+1}Y_U$$

In the previous expression, the second term is null, so we need to prove that the first term converges; however, it's easy to recognize a truncated matrix geometrical series (Neumann series), and A_{UU} is constructed to have all eigenvalues $|\lambda_i| < 1$, therefore the series converges to:

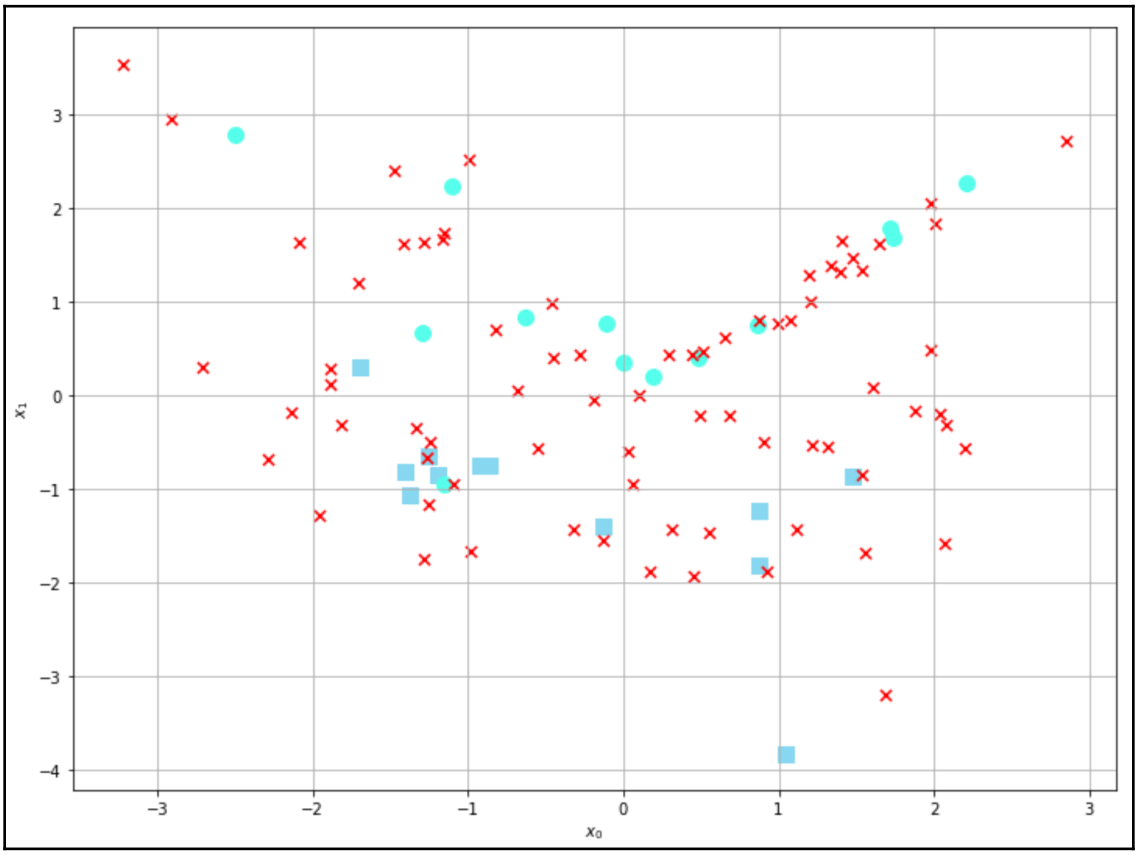
$$\widetilde{Y}_Y^{(\infty)} = \lim_{t\to\infty}\sum_{k=1}^{t+1}(A_{UU})^{k-1}A_{UL}Y_L = (I - A_{UU})^{-1}A_{UL}Y_L$$

Example of label propagation

We can implement the algorithm in Python, using a test bidimensional dataset:

```
from sklearn.datasets import make_classification

nb_samples = 100
nb_unlabeled = 75

X, Y = make_classification(n_samples=nb_samples, n_features=2,
n_informative=2, n_redundant=0, random_state=1000)
Y[Y==0] = -1
Y[nb_samples - nb_unlabeled:nb_samples] = 0
```

As in the other examples, we set $y = 0$ for all unlabeled samples (75 out of 100). The corresponding plot is shown in the following graph:



Partially labeled dataset

The dots marked with a cross are unlabeled. At this point, we can define the affinity matrix. In this case, we compute it using both methods:

```
from sklearn.neighbors import kneighbors_graph

nb_neighbors = 2

W_knn_sparse = kneighbors_graph(X, n_neighbors=nb_neighbors,
mode='connectivity', include_self=True)
W_knn = W_knn_sparse.toarray()
```

The KNN matrix is obtained using the Scikit-Learn function `kneighbors_graph()` with the parameters `n_neighbors=2` and `mode='connectivity'`; the alternative is `'distance'`, which returns the distances instead of 0 and 1 to indicate the absence/presence of an edge. The `include_self=True` parameter is useful, as we want to have $W_{ii} = 1$.

For the RBF matrix, we need to define it manually:

```
import numpy as np

def rbf(x1, x2, gamma=10.0):
    n = np.linalg.norm(x1 - x2, ord=1)
    return np.exp(-gamma * np.power(n, 2))

W_rbf = np.zeros((nb_samples, nb_samples))

for i in range(nb_samples):
    for j in range(nb_samples):
        W_rbf[i, j] = rbf(X[i], X[j])
```

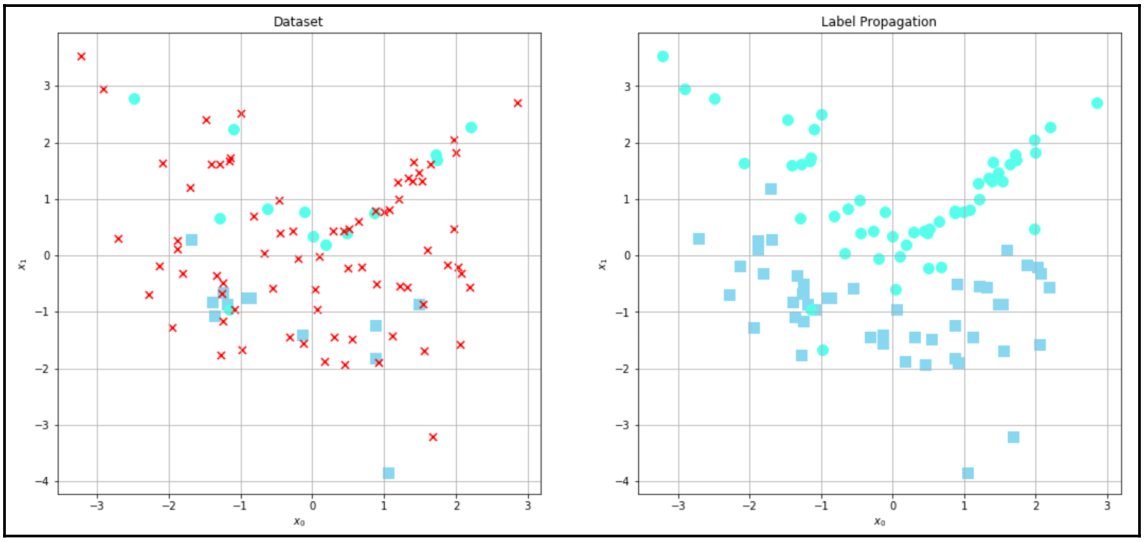
The default value for γ is *10*, corresponding to a standard deviation σ equal to *0.22*. When using this method, it's important to set a correct value for γ; otherwise, the propagation can degenerate in the predominance of a class (γ too small). Now, we can compute the degree matrices and its inverse. As the procedure is identical, from this point on we continue using the RBF affinity matrix:

```
D_rbf = np.diag(np.sum(W_rbf, axis=1))
D_rbf_inv = np.linalg.inv(D_rbf)
```

The algorithm is implemented using a variable threshold. The value adopted here is `0.01`:

```
tolerance = 0.01

Yt = Y.copy()
Y_prev = np.zeros((nb_samples,))
iterations = 0

while np.linalg.norm(Yt - Y_prev, ord=1) > tolerance:
    P = np.dot(D_rbf_inv, W_rbf)
    Yt = np.dot(P, Yt)
    Yt[0:nb_samples - nb_unlabeled] = Y[0:nb_samples - nb_unlabeled]
    Y_prev = Yt.copy()

Y_final = np.sign(Yt)
```

The final result is shown in the following double plot:

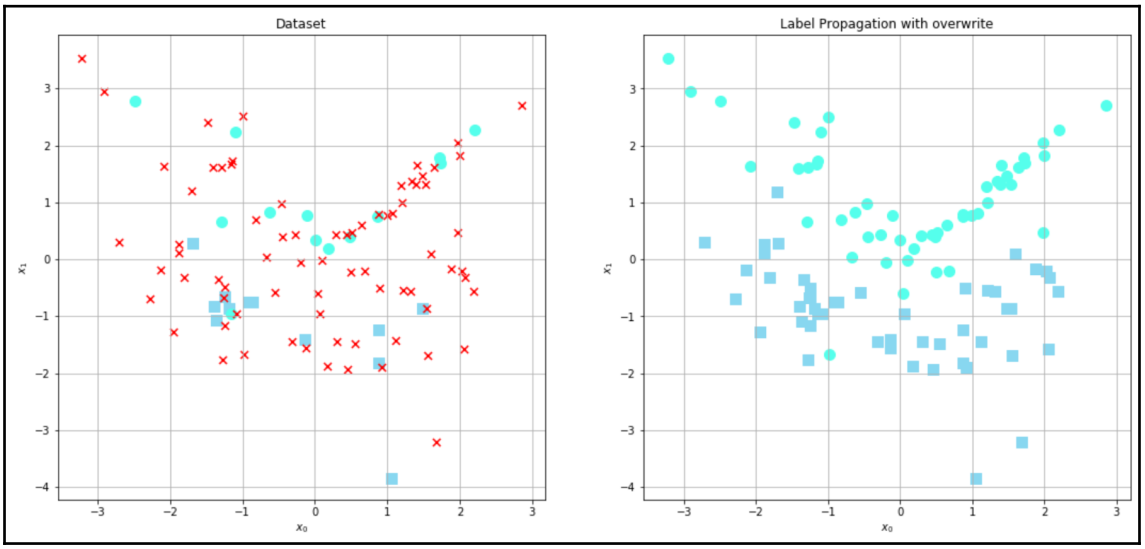


Original dataset (left); dataset after a complete label propagation (right)

As it's possible to see, in the original dataset there's a round dot surrounded by square ones (-0.9, -1). As this algorithm keeps the original labels, we find the same situation after the propagation of labels. This condition could be acceptable, even if both the smoothness and clustering assumptions are contradicted. Assuming that it's reasonable, it's possible to force a *correction* by relaxing the algorithm:

```
tolerance = 0.01

Yt = Y.copy()
Y_prev = np.zeros((nb_samples,))
iterations = 0

while np.linalg.norm(Yt - Y_prev, ord=1) > tolerance:
    P = np.dot(D_rbf_inv, W_rbf)
    Yt = np.dot(P, Yt)
    Y_prev = Yt.copy()

Y_final = np.sign(Yt)
```

In this way, we don't reset the original labels, letting the propagation change all those values that disagree with the neighborhood. The result is shown in the following plot:



Original dataset (left); dataset after a complete label propagation with overwrite (right)

Label propagation in Scikit-Learn

Scikit-Learn implements a slightly different algorithm proposed by Zhu and Ghahramani (in the aforementioned paper) where the affinity matrix *W* can be computed using both methods (KNN and RBF), but it is normalized to become a probability transition matrix:

$$P_{ij}(i \to j) = \frac{w_{ij}}{\sum_k w_{kj}}$$

The algorithm operates like a Markov random walk, with the following sequence (assuming that there are *Q* different labels):

1. Define a matrix $Y^M_i = [P(label=y_0), P(label=y_1), ..., and\ P(label=y_Q)]$, where *P(label=yi)* is the probability of the label *yi*, and each row is normalized so that all the elements sum up to *1*
2. Define $Y^{(0)} = Y^M$

3. Iterate until convergence of the following steps:

$$\begin{cases} \widetilde{Y}_M^{(t+1)} = P\widetilde{Y}_M^{(t)} \\ \widetilde{Y}_M^{(t+1)}(i,j) = \frac{\widetilde{Y}_M^{(t+1)}(i,j)}{\sum_k \widetilde{Y}_M^{(t+1)}(k,j)} \\ \widetilde{Y}_{ML}^{(t+1)} = Y_L \end{cases}$$

The first update performs a label propagation step. As we're working with probabilities, it's necessary (second step) to renormalize the rows so that their element sums up to *1*. The last update resets the original labels for all labeled samples. In this case, it means imposing a $P(label=y_i) = 1$ to the corresponding label, and setting all the others to zero. The proof of convergence is very similar to the one for label propagation algorithms, and can be found in *Learning from Labeled and Unlabeled Data with Label Propagation*, *Zhu X., Ghahramani Z., CMU-CALD-02-107*. The most important result is that the solution can be obtained in closed form (without any iteration) through this formula:

$$Y_U = (I - P_{uu})^{-1} P_{ul} Y_L$$

The first term is the sum of a generalized geometric series, where P_{uu} is the unlabeled-unlabeled part of the transition matrix *P*. P_{ul}, instead, is the unlabeled-labeled part of the same matrix.

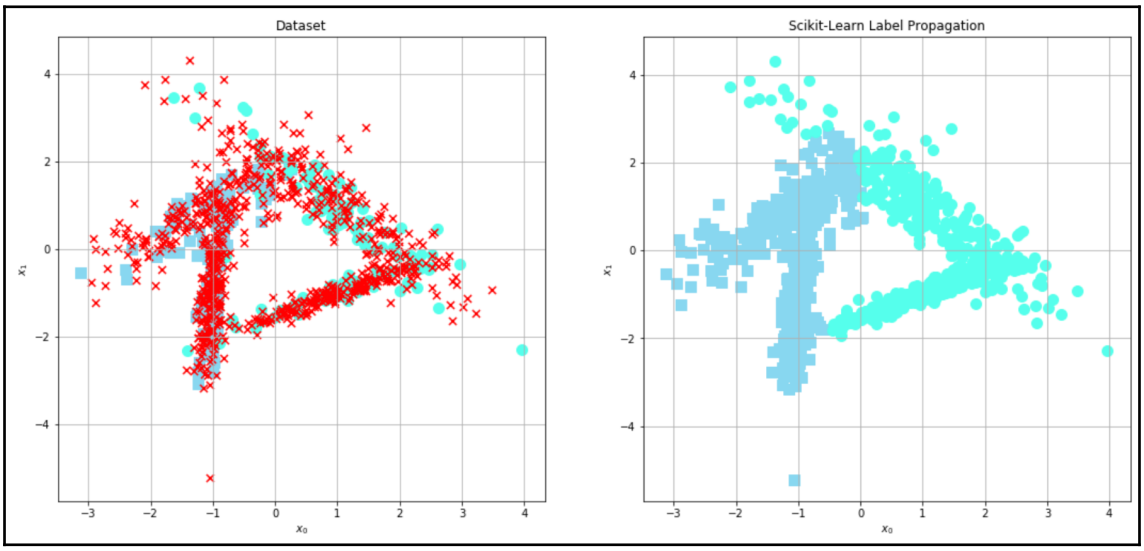
For our Python example, we need to build the dataset differently, because Scikit-Learn considers a sample unlabeled if *y=-1*:

```
from sklearn.datasets import make_classification

nb_samples = 1000
nb_unlabeled = 750

X, Y = make_classification(n_samples=nb_samples, n_features=2,
n_informative=2, n_redundant=0, random_state=100)
Y[nb_samples - nb_unlabeled:nb_samples] = -1
```

We can now train a `LabelPropagation` instance with an RBF kernel and `gamma=10.0`:

```
from sklearn.semi_supervised import LabelPropagation

lp = LabelPropagation(kernel='rbf', gamma=10.0)
lp.fit(X, Y)

Y_final = lp.predict(X)
```

The result is shown in the following double plot:



Original dataset (left). Dataset after a Scikit-Learn label propagation (right)

As expected, the propagation converged to a solution that respects both the smoothness and the clustering assumption.

Label spreading

The last algorithm (proposed by Zhou et al.) that we need to analyze is called **label spreading**, and it's based on the normalized graph Laplacian:

$$\mathcal{L} = D^{-\frac{1}{2}} W D^{-\frac{1}{2}}$$

This matrix has each a diagonal element l_{ii} equal to *1*, if the degree $deg(l_{ii}) > 0$ (0 otherwise) and all the other elements equal to:

$$l_{ij} = -\frac{1}{\sqrt{deg(v_i)}\sqrt{deg(v_j)}} \quad if\ v_i \in NN(v_j)$$

The behavior of this matrix is analogous to a discrete Laplacian operator, whose real-value version is the fundamental element of all diffusion equations. To better understand this concept, let's consider the generic heat equation:

$$\frac{\partial Q}{\partial t} = \rho \nabla^2 Q$$

This equation describes the behavior of the temperature of a room when a point is suddenly heated. From basic physics concepts, we know that heat will spread until the temperature reaches an equilibrium point and the speed of variation is proportional to the Laplacian of the distribution. If we consider a bidimensional grid at the equilibrium (the derivative with respect to when time becomes null) and we discretize the Laplacian operator ($\nabla^2 = \nabla \cdot \nabla$) considering the incremental ratios, we obtain:

$$\rho\left(Q(x+1,y) + Q(x-1,y) + Q(x,y+1) + Q(x,y-1) - 4Q(x,y)\right) = 0 \Rightarrow$$
$$Q(x,y) = \frac{Q(x+1,y) + Q(x-1,y) + Q(x,y+1) + Q(x,y-1)}{4}$$

Therefore, at the equilibrium, each point has a value that is the mean of the direct neighbors. It's possible to prove the finite-difference equation has a single fixed point that can be found iteratively, starting from every initial condition. In addition to this idea, label spreading adopts a clamping factor α for the labeled samples. If $\alpha=0$, the algorithm will always reset the labels to the original values (like for label propagation), while with a value in the interval $(0, 1]$, the percentage of clamped labels decreases progressively until $\alpha=1$, when all the labels are overwritten.

The complete steps of the **label spreading** algorithm are:

1. Select an affinity matrix type (KNN or RBF) and compute W
2. Compute the degree matrix D
3. Compute the normalized graph Laplacian L
4. Define $Y^{(0)} = Y$

5. Define α in the interval *[0, 1]*
6. Iterate until convergence of the following step:

$$\widetilde{Y}^{(t+1)} = \alpha \mathcal{L} \widetilde{Y}^{(t)} + (1 - \alpha) Y^{(0)}$$

It's possible to show (as demonstrated in *Semi-Supervised Learning*, *Chapelle O.*, *Schölkopf B.*, *Zien A.*, (edited by), *The MIT Press*) that this algorithm is equivalent to the minimization of a quadratic cost function with the following structure:

$$L(\widetilde{Y}) = \left\| \widetilde{Y}_L - Y_L \right\|^2 + \left\| \widetilde{Y}_U \right\|^2 + \mu \left(D^{-\frac{1}{2}} \right)^T (D - W)(D^{-\frac{1}{2}} \widetilde{Y})$$

The first term imposes consistency between original labels and estimated ones (for the labeled samples). The second term acts as a normalization factor, forcing the unlabeled terms to become zero, while the third term, which is probably the least intuitive, is needed to guarantee geometrical coherence in terms of smoothness. As we have seen in the previous paragraph, when a hard-clamping is adopted, the smoothness assumption could be violated. By minimizing this term (μ is proportional to α), it's possible to penalize the rapid changes inside the high-density regions. Also in this case, the proof of convergence is very similar to the one for label propagation algorithms, and will be omitted. The interested reader can find it in *Semi-Supervised Learning*, *Chapelle O.*, *Schölkopf B.*, *Zien A.*, (edited by), *The MIT Press.*

Example of label spreading

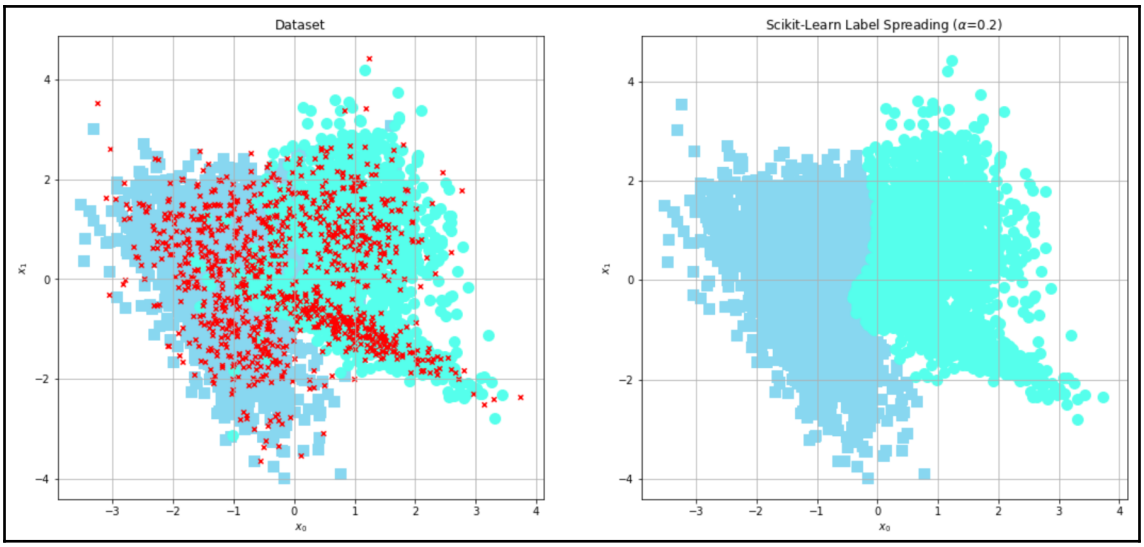
We can test this algorithm using the Scikit-Learn implementation. Let's start by creating a very dense dataset:

```
from sklearn.datasets import make_classification

nb_samples = 5000
nb_unlabeled = 1000

X, Y = make_classification(n_samples=nb_samples, n_features=2,
n_informative=2, n_redundant=0, random_state=100)
Y[nb_samples - nb_unlabeled:nb_samples] = -1
```

We can train a `LabelSpreading` instance with a clamping factor `alpha=0.2`. We want to preserve 80% of the original labels but, at the same time, we need a smooth solution:

```
from sklearn.semi_supervised import LabelSpreading

ls = LabelSpreading(kernel='rbf', gamma=10.0, alpha=0.2)
ls.fit(X, Y)

Y_final = ls.predict(X)
```

The result is shown, as usual, together with the original dataset:



Original dataset (left). Dataset after a complete label spreading (right)

As it's possible to see in the first figure (left), in the central part of the cluster *(x [-1, 0])*, there's an area of circle dots. Using a hard-clamping, this *aisle* would remain unchanged, violating both the smoothness and clustering assumptions. Setting $\alpha > 0$, it's possible to avoid this problem. Of course, the choice of α is strictly correlated with each single problem. If we know that the original labels are absolutely correct, allowing the algorithm to change them can be counterproductive. In this case, for example, it would be better to preprocess the dataset, filtering out all those samples that violate the semi-supervised assumptions. If, instead, we are not sure that all samples are drawn from the same p_{data}, and it's possible to be in the presence of spurious elements, using a higher α value can smooth the dataset without any other operation.

Label propagation based on Markov random walks

The goal of this algorithm proposed by Zhu and Ghahramani is to find the probability distribution of target labels for unlabeled samples given a mixed dataset. This objective is achieved through the simulation of a stochastic process, where each unlabeled sample walks through the graph until it reaches a stationary absorbing state, a labeled sample where it stops acquiring the corresponding label. The main difference with other similar approaches is that in this case, we consider the probability of reaching a labeled sample. In this way, the problem acquires a closed form and can be easily solved.

The first step is to always build a k-nearest neighbors graph with all *N* samples, and define a weight matrix *W* based on an RBF kernel:

$$W_{ij} = e^{-\gamma \|\bar{x}_i - \bar{x}_j\|^2}$$

$W_{ij} = 0$ is x_i and x_j are not neighbors and $W_{ii} = 1$. The transition probability matrix, similarly to the Scikit-Learn label propagation algorithm, is built as:

$$P_{ij}(i \rightarrow j) = \frac{w_{ij}}{\sum_k w_{kj}}$$

In a more compact way, it can be rewritten as $P = D^{-1}W$. If we now consider a *test sample*, starting from the state x_i and randomly walking until an absorbing labeled state is found (we call this label y^{∞}), the probability (referred to as **binary classification**) can be expressed as:

$$P(y^{\infty} = 1|\bar{x}_i) = \begin{cases} I_{y_i=1} \;\; if\ x_i\ is\ labeled \\ \sum_{k=1}^{N} p(y^{\infty} = 1|x_k)p(x_i|x_k) \;\; if\ x_i\ is\ unlabeled \end{cases}$$

When x_i is labeled, the state is final, and it is represented by the indicator function based on the condition $y_i=1$. When the sample is unlabeled, we need to consider the sum of all possible transitions starting from x_i and ending in the closest absorbing state, with label $y=1$ weighted by the relative transition probabilities.

We can rewrite this expression in matrix form. If we create a vector $P^{\infty} = [\ P_L(y^{\infty}=1 \mid X_L)$, $P_U(y^{\infty}=1 \mid X_U)\ ]$, where the first component is based on labeled samples and the second on the unlabeled ones, we can write:

$$P^{\infty} = D^{-1}WP^{\infty}$$

If we now expand the matrices, we get:

$$P^{\infty} = \begin{pmatrix} D_U^{-1} & 0 \\ 0 & D_{uu}^{-1} \end{pmatrix} \begin{pmatrix} W_U & W_{LU} \\ W_{UL} & W_{UU} \end{pmatrix} P^{\infty} = \begin{pmatrix} D_U^{-1}W_U & D_U^{-1}W \\ D_{UU}^{-1}W_{UL} & D_{UU}^{-1}W_{UU} \end{pmatrix} P^{\infty}$$

As we are interested only in the unlabeled samples, we can consider only the second equation:

$$P_U(y^{\infty} = 1|X_U) = D_{UU}^{-1}W_{UL}P_L(y^{\infty} = 1|X_L) + D_{UU}^{-1}W_{UU}P_U(y^{\infty} = 1|X_U)$$

Simplifying the expression, we get the following linear system:

$$(D_{UU} - W_{UU})P_U(y^{\infty} = 1|X_U) = W_{UL}P_L(y^{\infty} = 1|X_L)$$

The term $(D_{uu} - W_{uu})$ is the unlabeled-unlabeled part of the unnormalized graph Laplacian $L = D - W$. By solving this system, we can get the probabilities for the class $y=1$ for all unlabeled samples.

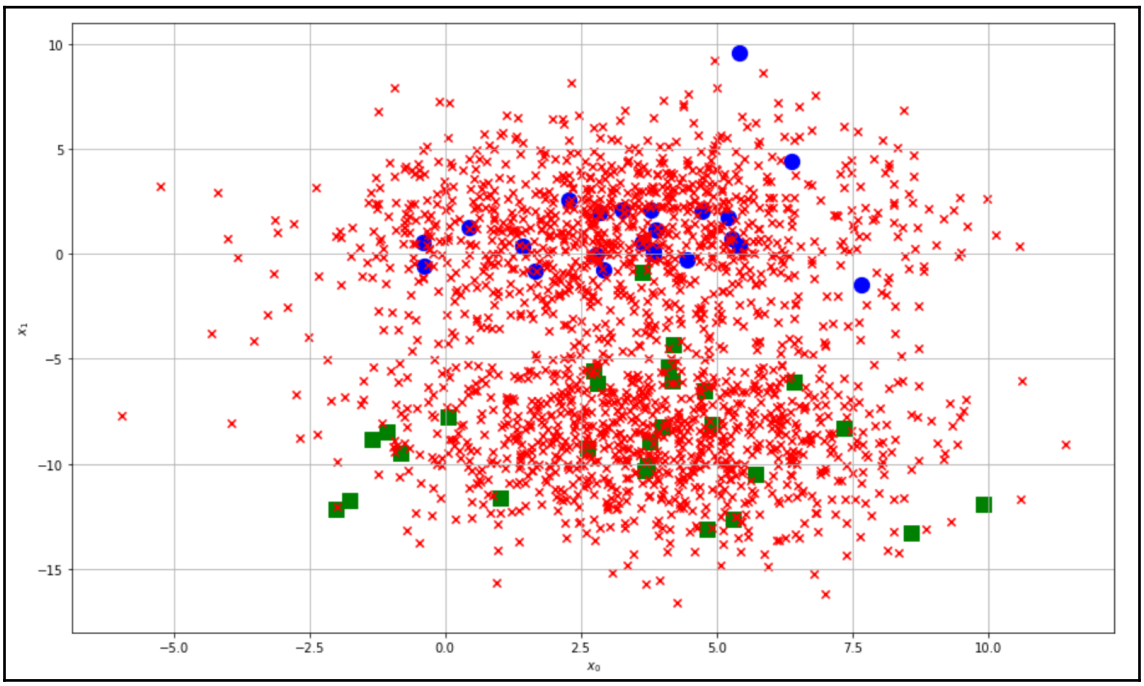
Example of label propagation based on Markov random walks

For this Python example of label propagation based on Markov random walks, we are going to use a bidimensional dataset containing 50 labeled samples belonging to two different classes, and 1,950 unlabeled samples:

```
from sklearn.datasets import make_blobs

nb_samples = 2000
nb_unlabeled = 1950
nb_classes = 2

X, Y = make_blobs(n_samples=nb_samples,
                  n_features=2,
```

```
                    centers=nb_classes,
                    cluster_std=2.5,
                    random_state=500)

Y[nb_samples - nb_unlabeled:] = -1
```

The plot of the dataset is shown in the following diagram (the crosses represent the unlabeled samples):



Partially labeled dataset

We can now create the graph (using `n_neighbors=15`) and the weight matrix:

```
import numpy as np

from sklearn.neighbors import kneighbors_graph

def rbf(x1, x2, sigma=1.0):
    d = np.linalg.norm(x1 - x2, ord=1)
    return np.exp(-np.power(d, 2.0) / (2 * np.power(sigma, 2)))

W = kneighbors_graph(X, n_neighbors=15, mode='connectivity',
include_self=True).toarray()

for i in range(nb_samples):
    for j in range(nb_samples):
        if W[i, j] != 0.0:
            W[i, j] = rbf(X[i], X[j])
```

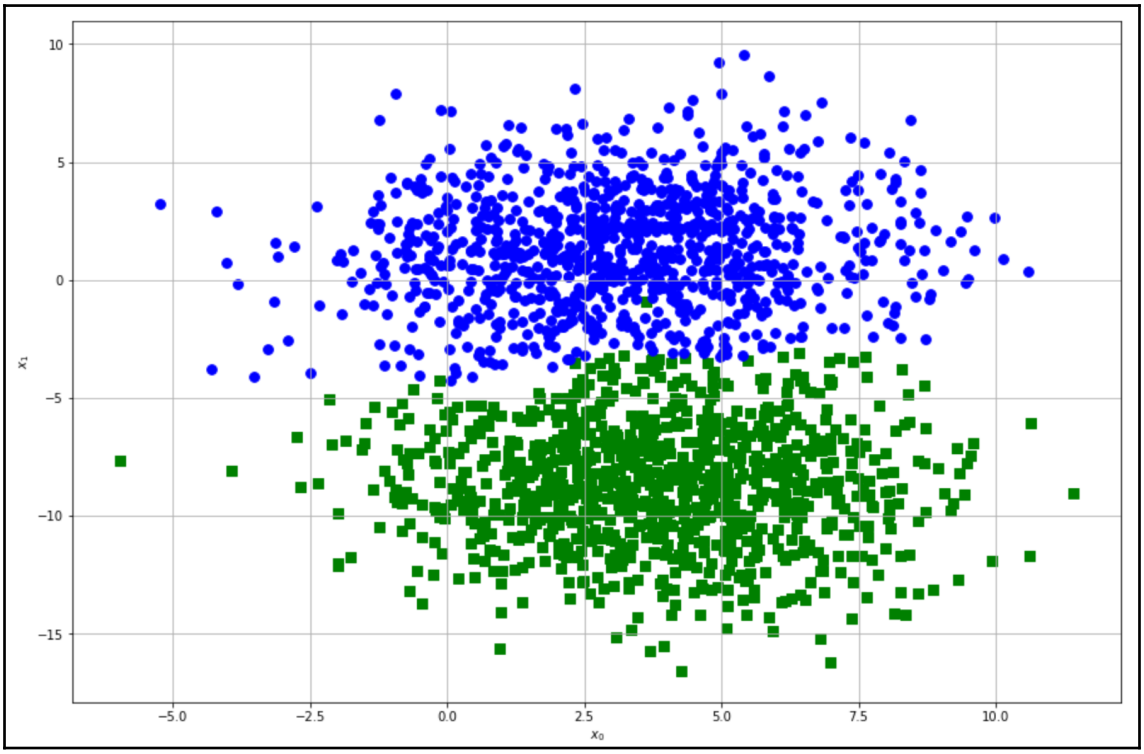
Now, we need to compute the unlabeled part of the unnormalized graph Laplacian and the unlabeled-labeled part of the matrix *W*:

```
D = np.diag(np.sum(W, axis=1))
L = D - W
Luu = L[nb_samples - nb_unlabeled:, nb_samples - nb_unlabeled:]
Wul = W[nb_samples - nb_unlabeled:, 0:nb_samples - nb_unlabeled,]
Yl = Y[0:nb_samples - nb_unlabeled]
```

At this point, it's possible to solve the linear system using the NumPy function `np.linalg.solve()`, which accepts as parameters the matrix *A* and the vector *b* of a generic system in the form *Ax=b*. Once we have the solution, we can merge the new labels with the original ones (where the unlabeled samples have been marked with *-1*). In this case, we don't need to convert the probabilities, because we are using *0* and *1* as labels. In general, it's necessary to use a threshold (0.5) to select the right label:

```
Yu = np.round(np.linalg.solve(Luu, np.dot(Wul, Yl)))
Y[nb_samples - nb_unlabeled:] = Yu.copy()
```

Replotting the dataset, we get:



Dataset after a complete Markov random walk label propagation

As expected, without any iteration, the labels have been successfully propagated to all samples in perfect compliance with the clustering assumption. Both this algorithm and label propagation can work using a closed-form solution, so they are very fast even when the number of samples is high; however, there's a fundamental problem regarding the choice of σ/γ for the RBF kernel. As the same authors Zhu and Ghahramani remark, there is no standard solution, but it's possible to consider when $\sigma \rightarrow 0$ and when $\sigma \rightarrow \infty$. In the first case, only the nearest point has an influence, while in the second case, the influence is extended to the whole sample space, and the unlabeled points tend to acquire the same label. The authors suggest considering the entropy of all samples, trying to find the best σ value that minimizes it. This solution can be very effective, but sometimes the minimum entropy corresponds to a label configuration that isn't impossible to achieve using these algorithms. The best approach is to try different values (at different scales) and select the one corresponding to a valid configuration with the lowest entropy. In our case, it's possible to compute the entropy of the unlabeled samples as:

$$H(X_u) = -\sum_{i=N+1}^{N+M} p(x_i)\ log\ p(x_i)$$

The Python code to perform this computation is:

```
Pu = np.linalg.solve(Luu, np.dot(Wul, Yl))
H = -np.sum(Pu * np.log(Pu + 1e-6))
```

The term `1e-6` has been added to avoid numerical problems when the probability is null. Repeating this process for different values allows us to find a set of candidates that can be restricted to a single value with a direct evaluation of the labeling accuracy (for example, when there is no precise information about the real distribution, it's possible to consider the coherence of each cluster and the separation between them). Another approach is called **class rebalancing**, and it's based on the idea of reweighting the probabilities of unlabeled samples to rebalance the number of points belonging to each class when the new unlabeled samples are added to the set. If we have *N* labeled points and *M* unlabeled ones, with *K* classes, the weight factor w_j for the class *j* can be obtained as:

$$w_j = \frac{\frac{1}{N}\sum_{t=1}^{N} y_t^{(j)}}{\frac{1}{M}\sum_{t=N+1}^{N+M} \tilde{y}_t^{(j)}}$$

The numerator is the average computed over the labeled samples belonging to class *k*, while the denominator is the average over the unlabeled ones whose estimated class is *k*. The final decision about a class is no longer based only on the highest probability, but on:

$$\tilde{y}_t^{(j)} = argmax_j \left(w_j \cdot p(y_t = j) \right)$$

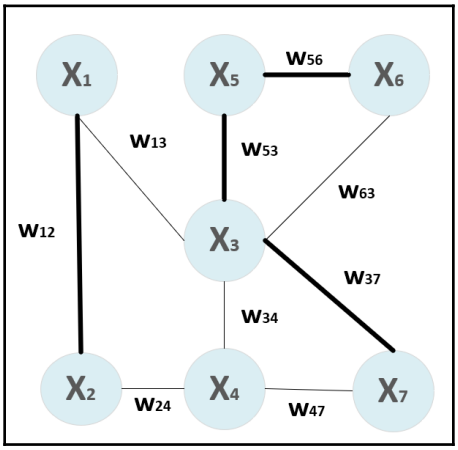
Manifold learning

In `Chapter 02`, *Introduction to Semi-Supervised Learning*, we discussed the manifold assumption, saying that high-dimensional data normally lies on low-dimensional manifolds. Of course, this is not a theorem, but in many real cases, the assumption is proven to be correct, and it allows us to work with non-linear dimensionality reduction algorithms that would be otherwise unacceptable. In this section, we're going to analyze some of these algorithms. They are all implemented in Scikit-Learn, therefore it's easy to try them with complex datasets.

Isomap

Isomap is one of the simplest algorithms, and it's based on the idea of reducing the dimensionality while trying to preserve the geodesic distances measured on the original manifold where the input data lies. The algorithm works in three steps. The first operation is a k-nearest neighbors clustering and the construction of the following graph. The vertices will be the samples, while the edges represent the connections among nearest neighbors, and their weight is proportional to the distance to the corresponding neighbor.

The second step adopts the **Dijkstra algorithm** to compute the shortest pairwise distances on the graph of all couples of samples. In the following graph, there's a portion of a graph, where some shortest distances are marked:



Example of a graph with marked shortest distances

For example, as x_3 is a neighbor of x_5 and x_7, applying the Dijkstra algorithm, we could get the shortest paths $d(x_3, x_5) = w_{53}$ and $d(x_3, x_7) = w_{73}$. The computational complexity of this step is about $O(n^2 \log n + n^2 k)$, which is lower than $O(n^3)$ when $k << n$ (a condition normally met); however, for large graphs (with $n >> 1$), this is often the most expensive part of the whole algorithm.

The third step is called **metric multidimensional scaling**, which is a technique for finding a low-dimensional representation while trying to preserve the inner product among samples. If we have a P-dimensional dataset X, the algorithm must find a Q-dimensional set Φ with $Q < P$ minimizing the function:

$$L_{MDS} = \sum_{i,j} (\bar{x}_i \cdot \bar{x}_j - \bar{\phi}_i \cdot \bar{\phi}_j)^2$$

As proven in *Semi-Supervised Learning Chapelle O., Schölkopf B., Zien A.*, (edited by), *The MIT Press*, the optimization is achieved by taking the top Q eigenvectors of the Gram matrix $G_{ij} = x_i \cdot x_j$ (or in matrix form, $G=XX^T$ if $X \in \Re^{n \times M}$); however, as the **Isomap** algorithm works with pairwise distances, we need to compute the matrix D of squared distances:

$$D_{ij} = \|\bar{x}_i - \bar{x}_j\|^2$$

If the X dataset is zero-centered, it's possible to derive a simplified Gram matrix from D, as described by M. A. A. Cox and T. F. Cox:

$$G_D = -\frac{1}{2}(I - \bar{v}\bar{v}^T)D(I - \bar{v}\bar{v}^T) \;\; \bar{v} \in \mathbb{R}^p \; and \; \bar{v} = \left(\frac{1}{\sqrt{P}}, \frac{1}{\sqrt{P}}, \ldots, \frac{1}{\sqrt{P}}\right)$$

Isomap computes the top Q eigenvalues $\lambda_1, \lambda 2, \ldots, \lambda_Q$ of G_D and the corresponding eigenvectors $v_1, v_2, \ldots, v_Q$ and determines the Q-dimensional vectors as:

$$\bar{\phi}_i = \left(\lambda_1^{\frac{1}{2}}\bar{v}_1, \lambda_2^{\frac{1}{2}}\bar{v}_2, \ldots, \lambda_Q^{\frac{1}{2}}\bar{v}_Q\right)$$

As we're going to discuss in `Chapter 5`, *EM Algorithm and Applications* (and also as pointed out by Saul, Weinberger, Sha, Ham, and Lee in *Spectral Methods for Dimensionality Reduction, Saul L. K., Weinberger K. Q., Sha F., Ham J., and Lee D. D.*), this kind of projection is also exploited by **Principal Component Analysis** (**PCA**), which finds out the direction with the highest variance, corresponding to the top k eigenvectors of the covariance matrix. In fact, when applying the SVD to the dataset X, we get:

$$X = U\Lambda V^T \;\; where \;\; U \in \mathbb{R}^{M \times M}, \;\; \Lambda \; is \; diag(n \times n) \;\; and \;\; V \in \mathbb{R}^{n \times n}$$

The diagonal matrix Λ contains the eigenvalues of both XX^T and X^TX; therefore, the eigenvalues λ_{Gi} of G are equal to $M\lambda_{x_i}$ where λ_{x_i} are the eigenvalues of the covariance matrix $\Sigma = M^{-1}X^TX$. Hence, Isomap achieves the dimensionality reduction, trying to preserve the pairwise distances, while projecting the dataset in the subspace determined by a group of eigenvectors, where the maximum explained variance is achieved. In terms of information theory, this condition guarantees the minimum loss with an effective reduction of dimensionality.

Scikit-Learn also implements the Floyd-Warshall algorithm, which is slightly slower. For further information, please refer to *Introduction to Algorithms*, *Cormen T. H.*, *Leiserson C. E.*, *Rivest R. L.*, *The MIT Press*.

Example of Isomap

We can now test the Scikit-Learn **Isomap** implementation using the Olivetti faces dataset (provided by AT&T Laboratories, Cambridge), which is made up of 400 64 × 64 grayscale portraits belonging to 40 different people. Examples of these images are shown here:

Subset of the Olivetti faces dataset

The original dimensionality is 4096, but we want to visualize the dataset in two dimensions. It's important to understand that using the Euclidean distance for measuring the similarity of images might not the best choice, and it's surprising to see how well the samples are clustered by such a simple algorithm.

The first step is loading the dataset:

```
from sklearn.datasets import fetch_olivetti_faces

faces = fetch_olivetti_faces()
```

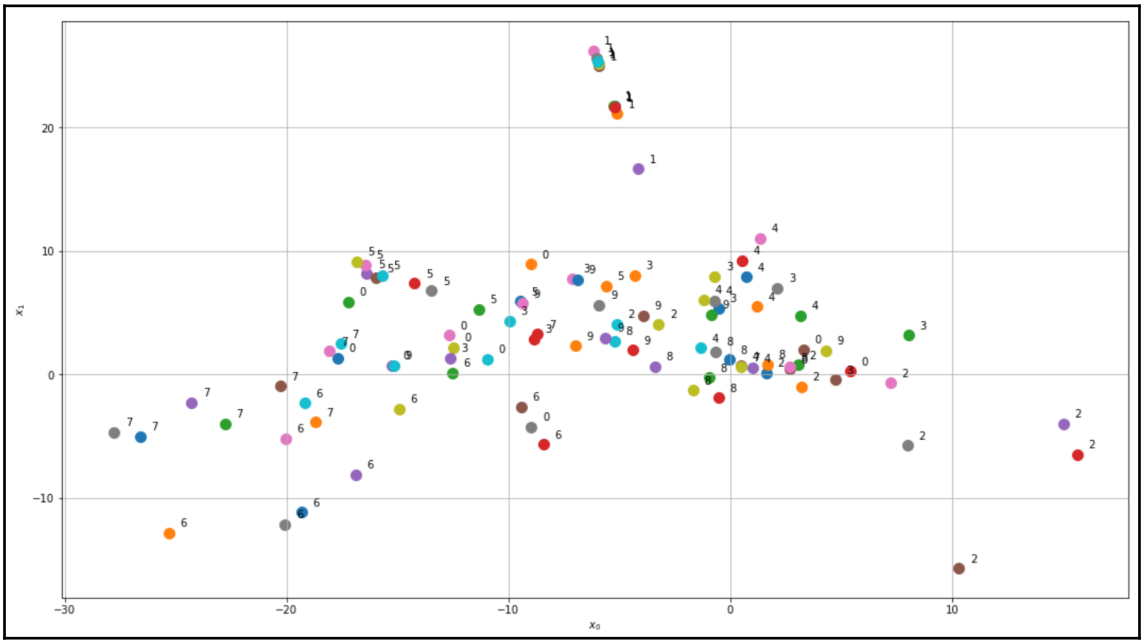
The `faces` dictionary contains three main elements:

- `images`: Image array with shape 400 × 64 × 64
- `data`: Flattened array with shape 400 × 4096
- `target`: Array with shape 400 × 1 containing the labels (0, 39)

At this point, we can instantiate the `Isomap` class provided by Scikit-Learn, setting `n_components=2` and `n_neighbors=5` (the reader can try different configurations), and then fitting the model:

```
from sklearn.manifold import Isomap

isomap = Isomap(n_neighbors=5, n_components=2)
X_isomap = isomap.fit_transform(faces['data'])
```

As the resulting plot with 400 elements is very dense, I preferred to show in the following plot only the first 100 samples:



Isomap applied to 100 samples drawn from the Olivetti faces dataset

As it's possible to see, samples belonging to the same class are grouped in rather dense agglomerates. The classes that seem better separated are 7 and 1. Checking the corresponding faces, for class 7, we get:

Samples belonging to class 7

The set contains portraits of a young woman with a fair complexion, quite different from the majority of other people. Instead, for class 1, we get:

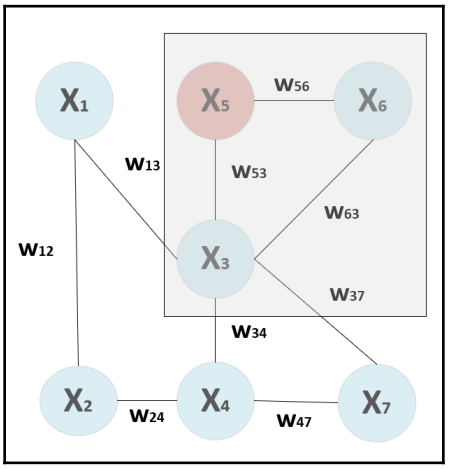
Samples belonging to class 1

In this case, it's a man with big glasses and a particular mouth expression. In the dataset, there are only a few people with glasses, and one of them has a dark beard. We can conclude that **Isomap** created a low-dimensional representation that is really coherent with the original geodesic distances. In some cases, there's a partial clustering overlap that can be mitigated by increasing the dimensionality or adopting a more complex strategy.

Locally linear embedding

Contrary to Isomap, which works with the pairwise distances, this algorithm is based on the assumption that a high-dimensional dataset lying on a smooth manifold can have local linear structures that it tries to preserve during the dimensionality reduction process. **Locally Linear Embedding** (**LLE**), like Isomap, is based on three steps. The first one is applying the *k*-nearest neighbor algorithm to create a directed graph (in Isomap, it was undirected), where the vertices are the input samples and the edges represent a neighborhood relationship. As the graph is direct, a point x_i can be a neighbor of x_j, but the opposite could be false. It means that the weight matrix can be asymmetric.

The second step is based on the main assumption of local linearity. For example, consider the following graph:



Graph where a neighborhood is marked with a shaded rectangle

The rectangle delimits a small neighboorhood. If we consider the point x_5, the local linearity assumption allows us to think that $x_5 = w_{56}x_6 + w_{53}x_3$, without considering the cyclic relationship. This concept can be formalized for all N P-dimensional points through the minimization of the following function:

$$L_W = \sum_{i=1}^{N} \left\| \bar{x}_i - \sum_{k \in NN(\bar{x}_i)} W_{ik}\bar{x}_k \right\|^2 \quad subject\ to \sum_{k \in NN(\bar{x}_i)} W_{ik} = 1$$

In order to address the problem of low-rank neighborhood matrices (think about the previous example, with a number of neighbors equal to 20), Scikit-Learn also implements a regularizer that is based on a small arbitrary additive constant that is added to the local weights (according to a variant called **Modified LLE** or **MLLE**). At the end of this step, the matrix W that better matches the linear relationships among neighbors will be selected for the next phase.

In the third step, locally linear embedding tries to determine the low-dimensional ($Q < P$) representation that best reproduces the original relationship among nearest neighbors. This is achieved by minimizing the following function:

$$L_{\Phi} = \sum_{i=1}^{N} \left\| \bar{\phi}_i - \sum_{k \in NN(\bar{\phi}_i)} W_{ik} \bar{\phi}_k \right\|^2 \quad subject\ to \sum_i \bar{\phi}_i = 0 \ \ and \ \ Cov(\bar{\phi}_i, \bar{\phi}_j) = 1 \ \ \forall\, i, j$$

The solution for this problem is obtained through the adoption of the **Rayleigh-Ritz method**, an algorithm to extract a subset of eigenvectors and eigenvalues from a very large sparse matrix. For further details, read *A spectrum slicing method for the Kohn–Sham problem, Schofield G. Chelikowsky J. R.; Saad Y., Computer Physics Communications. 183*. The initial part of the final procedure consists of determining the matrix *D*:

$$D = (I - W)^T (I - W)$$

It's possible to prove the last eigenvector (if the eigenvalues are sorted in descending order, it's the bottom one) has all components $v_1^{(N)}$, $v_2^{(N)}$, ..., $v_N^{(N)} = v$, and the corresponding eigenvalue is null. As Saul and Roweis (*An introduction to locally linear embedding, Saul L. K., Roweis S. T.*) pointed out, all the other *Q* eigenvectors (from the bottom) are orthogonal, and this allows them to have zero-centered embedding. Hence, the last eigenvector is discarded, while the remaining Q eigenvectors determine the embedding vectors φ_i.

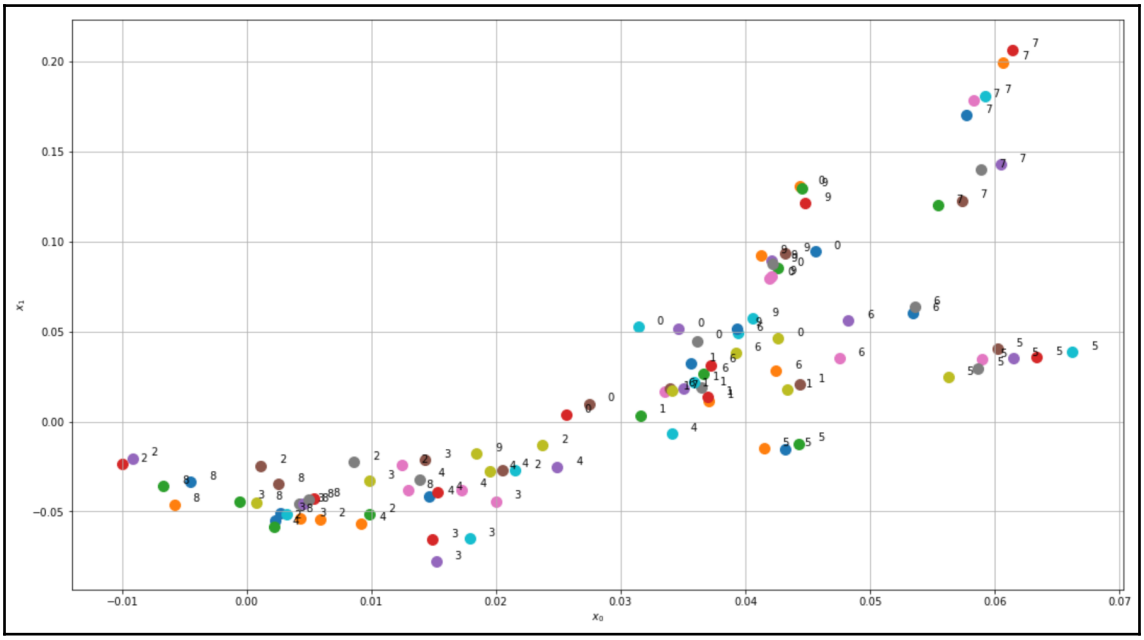
For further details about MLLE, please refer to *MLLE: Modified Locally Linear Embedding Using Multiple Weights, Zhang Z., Wang J.*, `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.70.382`.

Example of locally linear embedding

We can now apply this algorithm to the Olivetti faces dataset, instantiating the Scikit-Learn class `LocallyLinearEmbedding` with `n_components=2` and `n_neighbors=15`:

```
from sklearn.manifold import LocallyLinearEmbedding

lle = LocallyLinearEmbedding(n_neighbors=15, n_components=2)
X_lle = lle.fit_transform(faces['data'])
```

The result (limited to the first 100 samples) is shown in the following plot:



Locally linear embedding applied to 100 samples drawn from the Olivetti faces dataset

Even if the strategy is different from Isomap, we can determine some coherent clusters. In this case, the similarity is obtained through the conjunction of small linear blocks; for the faces, they can represent particular micro-features, like the shape of the nose or the presence of glasses, that remain invariant in the different portraits of the same person. LLE is, in general, preferable when the original dataset is intrinsically locally linear, possibly lying on a smooth manifold. In other words, LLE is a reasonable choice when small parts of a sample are structured in a way that allows the reconstruction of a point given the neighbors and the weights. This is often true for images, but it can be difficult to determine for a generic dataset. When the result doesn't reproduce the original clustering, it's possible to employ the next algorithm or **t-SNE**, which is one the most advanced.

Laplacian Spectral Embedding

This algorithm, based on the spectral decomposition of a graph Laplacian, has been proposed in order to perform a non-linear dimensionality reduction to try to preserve the nearness of points in the *P*-dimensional manifold when remapping on a *Q*-dimensional (with $Q < P$) subspace.

The procedure is very similar to the other algorithms. The first step is a *k*-nearest neighbor clustering to generate a graph where the vertices (we can assume to have *N* elements) are the samples, and the edges are weighted using an RBF kernel:

$$W_{ij} = e^{-\gamma \|\bar{x}_i - \bar{x}_j\|^2}$$

The resulting graph is undirected and symmetric. We can now define a pseudo-degree matrix *D*:

$$D = diag\left(\sum_j W_{1j}, \sum_j W_{2j}, \ldots, \sum_j W_{Nj}\right)$$

The low-dimensional representation Φ is obtained by minimizing the function:

$$L_\Phi = \sum_{i,j} \frac{W_{ij}\left\|\bar{\phi}_i - \bar{\phi}_j\right\|^2}{\sqrt{D_{ii} D_{jj}}} \;\; subject\, to \sum_i \bar{\phi}_i = 0 \;\; and \;\; Cov(\bar{\phi}_i, \bar{\phi}_j) = 1 \; \forall \; i, j$$

If the two points x_i and x_j are near, the corresponding W_{ij} is close to *1*, while it tends to 0 when the distance tends to ∞. D_{ii} is the sum of all weights originating from x_i (and the same for D_{jj}). Now, let's suppose that x_i is very close only to x_j so, to approximate $D_{ii} = D_{jj} \approx W_{ij}$. The resulting formula is a square loss based on the difference between the vectors φ_i and φ_j. When instead there are multiple *closeness* relationships to consider, the factor W_{ij} divided by the square root of $D_{ii}D_{jj}$ allows reweighting the new distances to find the best trade-off for the whole dataset. In practice, L_Φ is not minimized directly. In fact, it's possible to prove that the minimum can be obtained through the spectral decomposition of the symmetric normalized graph Laplacian (the name derives from this procedure):

$$\mathcal{L} = I - D^{-\frac{1}{2}} W D^{-\frac{1}{2}}$$

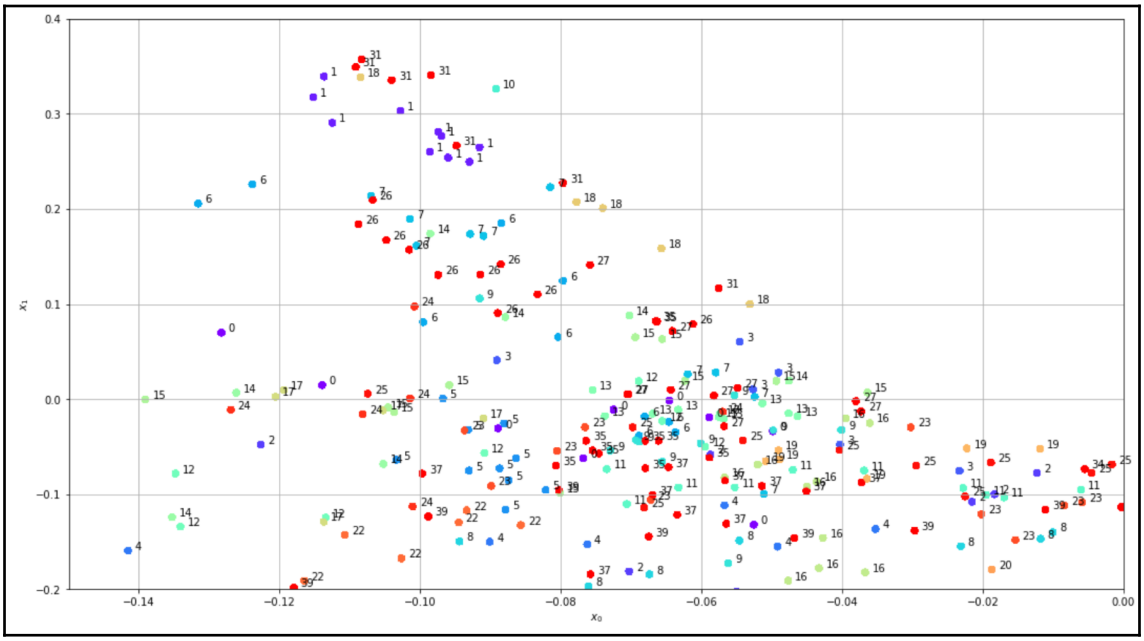
Just like for the LLE algorithm, Laplacian Spectral Embedding also works with the bottom $Q + 1$ eigenvectors. The mathematical theory behind the last step is always based on the application of the Rayleigh-Ritz method. The last one is discarded, and the remaining Q determines the low-dimensional representation φ_i.

Example of Laplacian Spectral Embedding

Let's apply this algorithm to the same dataset using the Scikit-Learn class `SpectralEmbedding`, with `n_components=2` and `n_neighbors=15`:

```
from sklearn.manifold import SpectralEmbedding

se = SpectralEmbedding(n_components=2, n_neighbors=15)
X_se = se.fit_transform(faces['data'])
```

The resulting plot (zoomed in due to the presence of a high-density region) is shown in the following graph:



Laplacian Spectral Embedding applied to the Olivetti faces dataset

Even in this case, we can see that some classes are grouped into small clusters, but at the same time, we observe many agglomerates where there are mixed samples. Both this and the previous method work with local pieces of information, trying to find low-dimensional representations that could preserve the geometrical structure of micro-features. This condition drives to a mapping where close points *share* local features (this is almost always true for images, but it's very difficult to prove for generic samples). Therefore, we can observe small clusters containing elements belonging to the same class, but also some *apparent* outliers, which, on the original manifold, can be globally different even if they share local *patches*. Instead, methods like Isomap or t-SNE work with the whole distribution, and try to determine a representation that is almost isometric with the original dataset considering its global properties.

t-SNE

This algorithm, proposed by Van der Mateen and Hinton and formally known as **t-Distributed Stochastic Neighbor Embedding** (**t-SNE**), is one of the most powerful manifold dimensionality reduction techniques. Contrary to the other methods, this algorithm starts with a fundamental assumption: the similarity between two N-dimensional points x_i and x_j can be represented as the conditional probability $p(x_j|x_i)$ where each point is represented by a Gaussian distribution centered in x_i and with variance σ_i. The variances are selected starting from the desired perplexity, defined as:

$$Perplexity(P) = 2^{H(P)}$$

Low-perplexity values indicate a low uncertainty, and are normally preferable. In common t-SNE tasks, values in the range $10 \div 50$ are normally acceptable.

The assumption on the conditional probabilities can be interpreted thinking that if two samples are very similar, the probability associated with the first sample conditioned to the second one is high, while dissimilar points yield low conditional probabilities. For example, thinking about images, a point centered in the pupil can have as neighbors some points belonging to an eyelash. In terms of probabilities, we can think that $p(eyelash|pupil)$ is quite high, while $p(nose|pupil)$ is obviously lower. t-SNE models these conditional probabilities as:

$$p(\bar{x}_j|\bar{x}_i) = \frac{e^{-\frac{\left\|\bar{x}_i-\bar{x}_j\right\|^2}{2\sigma_i^2}}}{\sum_{k\neq i} e^{-\frac{\left\|\bar{x}_i-\bar{x}_k\right\|^2}{2\sigma_i^2}}}$$

The probabilities $p(x_j|x_i)$ are set to zero, so the previous formula can be extended to the whole graph. In order to solve the problem in an easier way, the conditional probabilities are also symmetrized:

$$p(\bar{x}_j|\bar{x}_i) = \frac{p(\bar{x}_i|\bar{x}_j) + p(\bar{x}_j|\bar{x}_i)}{2N}$$

The probability distribution so obtained represents the high-dimensional input relationship. As our goal is to reduce the dimensionality to a value $M < N$, we can think about a similar probabilistic representation for the target points φ_i, using a student-t distribution with one degree of freedom:

$$q(\bar{\phi}_i|\bar{\phi}_j) = \frac{\left(1+\left\|\bar{\phi}_i-\bar{\phi}_j\right\|^2\right)^{-1}}{\sum_{k\neq j}\left(1+\left\|\bar{\phi}_k-\bar{\phi}_j\right\|^2\right)^{-1}}$$

We want the low-dimensional distribution Q to be as close as possible to the high-dimensional distribution P; therefore, the aim of the **t-SNE** algorithm is to minimize the Kullback-Leibler divergence between P and Q:

$$D_{KL}(P||Q) = \sum_{i,j} p(\bar{x}_j|\bar{x}_i)\ log\frac{p(\bar{x}_j|\bar{x}_i)}{q(\bar{\phi}_j|\bar{\phi}_i)} = H(P) - \sum_{i,j} p(\bar{x}_j|\bar{x}_i) log\ q(\bar{\phi}_j|\bar{\phi}_i)$$

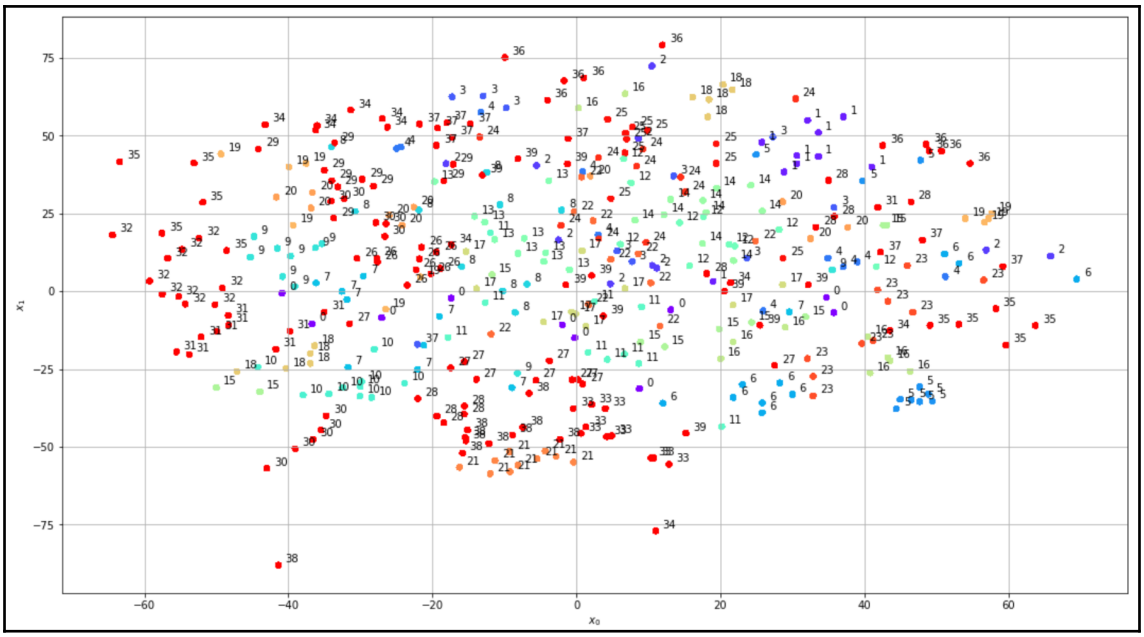
The first term is the entropy of the original distribution *P*, while the second one is the cross-entropy *H(P, Q)*, which has to be minimized to solve the problem. The best approach is based on a gradient-descent algorithm, but there are also some useful variations that can improve the performance discussed in *Visualizing High-Dimensional Data Using t-SNE, Van der Maaten L.J.P., Hinton G.E., Journal of Machine Learning Research 9 (Nov), 2008.*

Example of t-distributed stochastic neighbor embedding

We can apply this powerful algorithm to the same Olivetti faces dataset, using the Scikit-Learn class `TSNE` with `n_components=2` and `perplexity=20`:

```
from sklearn.manifold import TSNE

tsne = TSNE(n_components=2, perplexity=20)
X_tsne = tsne.fit_transform(faces['data'])
```

The result for all 400 samples is shown in the following graph:



t-SNE applied to the Olivetti faces dataset

A visual inspection of the label distribution can confirm that t-SNE recreated the optimal clustering starting from the original high-dimensional distribution. This algorithm can be employed in several non-linear dimensionality reduction tasks, such as images, word embeddings, or complex feature vectors. Its main strength is hidden in the assumption to consider the similarities as probabilities, without the need to impose any constraint on the pairwise distances, either global or local. Under a certain viewpoint, it's possible to consider t-SNE as a reverse multiclass classification problem based on a cross-entropy cost function. Our goal is to find the labels (low-dimensional representation) given the original distribution and an assumption about the output distribution.

At this point, we could try to answer a natural question: which algorithm must be employed? The obvious answer is it depends on the single problem. When it's useful to reduce the dimensionality, preserving the global similarity among vectors (this is the case when the samples are long feature vectors without local properties, such as word embeddings or data encodings), t-SNE or Isomap are good choices. When instead it's necessary to keep the local distances (for example, the structure of a visual patch that can be shared by different samples also belonging to different classes) as close as possible to the original representation, locally linear embedding or spectral embedding algorithms are preferable.

Summary

In this chapter, we have introduced the most important label propagation techniques. In particular, we have seen how to build a dataset graph based on a weighting kernel, and how to use the geometric information provided by unlabeled samples to determine the most likely class. The basic approach works by iterating the multiplication of the label vector times the weight matrix until a stable point is reached and we have proven that, under simple assumptions, it is always possible.

Another approach, implemented by Scikit-Learn, is based on the transition probability from a state (represented by a sample) to another one, until the convergence to a labeled point. The probability matrix is obtained using a normalized weight matrix to encourage transitions associated to close points and discourage all the *long jumps*. The main drawback of these two methods is the hard-clamping of labeled samples; this constraint can be useful if we *trust* our dataset, but it can be a limitation in the presence of outliers whose label has been wrongly assigned.

Label spreading solves this problem by introducing a clamping factor that determines the percentage of clamped labels. The algorithm is very similar to label propagation, but it's based on graph Laplacian and can be employed in all those problems where the data-generating distribution is not well-determined and the probability of noise is high.

The propagation based on Markov random walks is a very simple algorithm that can estimate the class distribution of unlabeled samples through a stochastic process. It's possible to imagine it as a *test sample* that walks through the graph until it reaches a final labeled state (acquiring the corresponding label). The algorithm is very fast and it has a closed-form solution that can be found by solving a linear system.

The next topic was the introduction of manifold learning with the Isomap algorithm, which is a simple but powerful solution based on a graph built using a *k*-nearest neighbors algorithm (this is a common step in most of these algorithms). The original pairwise distances are processed using the multidimensional scaling technique, which allows obtaining a low-dimensional representation where the distances between samples are preserved.

Two different approaches, based on local pieces of information, are locally linear embedding and Laplacian Spectral Embedding. The former tries to preserve the local linearity present in the original manifold, while the latter, which is based on the spectral decomposition of the normalized graph Laplacian, tries to preserve the nearness of original samples. Both methods are suitable for all those tasks where it's important not to consider the whole original distribution, but the similarity induced by small data *patches*.

We closed this chapter by discussing t-SNE, which is a very powerful algorithm that tries to model a low-dimensional distribution that is as similar as possible to the original high-dimensional one. This task is achieved by minimizing the Kullback-Leibler divergence between the two distributions. t-SNE is a state-of-the-art algorithm, useful whenever it's important to consider the whole original distribution and the similarity between entire samples.

In the next chapter, `Chapter 4`, *Bayesian Networks and Hidden Markov Models* we're going to introduce Bayesian networks in both a static and dynamic context, and hidden Markov models, with practical prediction examples. These algorithms allow modeling complex probabilistic scenarios made up of observed and latent variables, and infer future states using optimized sampling methods based only on the observations.

4
Bayesian Networks and Hidden Markov Models

In this chapter, we're going to introduce the basic concepts of Bayesian models, which allow working with several scenarios where it's necessary to consider uncertainty as a structural part of the system. The discussion will focus on static (time-invariant) and dynamic methods that can be employed where necessary to model time sequences.

In particular, the chapter covers the following topics:

- Bayes' theorem and its applications
- Bayesian networks
- Sampling from a Bayesian network using direct methods and **Markov chain Monte Carlo** (**MCMC**) ones (Gibbs and Metropolis-Hastings samplers)
- Modeling a Bayesian network with PyMC3
- **Hidden Markov Models** (**HMMs**)
- Examples with hmmlearn

Conditional probabilities and Bayes' theorem

If we have a probability space S and two events A and B, the probability of A given B is called **conditional probability**, and it's defined as:

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

As *P(A, B) = P(B, A)*, it's possible to derive **Bayes' theorem**:

$$\begin{cases} P(A,B) = P(A|B)P(B) \\ P(B,A) = P(B|A)P(A) \end{cases} \Rightarrow P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

This theorem allows expressing a conditional probability as a function of the opposite one and the two marginal probabilities *P(A)* and *P(B)*. This result is fundamental to many machine learning problems, because, as we're going to see in this and in the next chapters, normally it's easier to work with a conditional probability in order to get the opposite, but it's hard to work directly from the latter. A common form of this theorem can be expressed as:

$$P(A|B) \propto P(B|A)P(A)$$

Let's suppose that we need to estimate the probability of an event *A* given some observations *B*, or using the standard notation, **the posterior probability of A**; the previous formula expresses this value as proportional to the term *P(A)*, which is the marginal probability of *A*, called **prior probability**, and the conditional probability of the observations *B* given the event *A*. *P(B|A)* is called **likelihood**, and defines how event *A* is likely to determine *B*. Therefore, we can summarize the relation as *posterior probability* $\propto$ *likelihood* $\cdot$ *prior probability*. The proportion is not a limitation, because the term *P(B)* is always a normalizing constant that can be omitted. Of course, the reader must remember to normalize *P(A|B)* so that its terms always sum up to one.

This is a key concept of Bayesian statistics, where we don't directly trust the prior probability, but we reweight it using the likelihood of some observations. As an example, we can think to toss a coin 10 times (event *A*). We know that *P(A) = 0.5* if the coin is fair. If we'd like to know what the probability is to get 10 heads, we could employ the Binomial distribution obtaining *P(10 heads) = 0.5^k*; however, let's suppose that we don't know whether the coin is fair or not, but we suspect it's loaded with a prior probability *P(Loaded) = 0.7* in favor of tails. We can define a complete prior probability *P(Coin status)* using the indicator functions:

$$P(Coin\ status) = P(Fair)I_{Coin=Fair} + P(Loaded)I_{Coin=Loaded}$$

Where *P(Fair) = 0.5* and *P(Loaded) = 0.7*, the indicator $I_{Coin=Fair}$ is equal to 1 only if the coin is fair, and 0 otherwise. The same happens with $I_{Coin=Loaded}$ when the coin is loaded. Our goal now is to determine the posterior probability *P(Coin status|* B_1, B_2, ..., B_n*)* to be able to confirm or to reject our hypothesis.

Let's imagine to observe $n = 10$ events with $B_1 = Head$ and $B_2, ..., B_n = Tail$. We can express the probability using the binomial distribution:

$$P(Coin\ status|B_1, B_2, \ldots, B_n) \propto \left[\binom{10}{1}0.5(1-0.5)^9 \cdot 0.3I_{Fair} + \binom{10}{1}0.7(1-0.7)^9 \cdot 0.7I_{Loaded}\right]$$

After simplifying the expression, we get:

$$P(Coin\ status|B_1, B_2, \ldots, B_n) \propto (0.003I_{Fair} + 0.08I_{Loaded})$$

We still need to normalize by dividing both terms by 0.083 (the sum of the two terms), so we get the final posterior probability $P(Coin\ status|B_1, B_2, ..., Bn) = 0.04I_{Fair} + 0.96I_{Loaded}$. This result confirms and strengthens our hypothesis. The probability of a loaded coin is now about 96%, thanks to the sequence of nine tail observations after one head.

This example was presented to show how the data (observations) is plugged into the Bayesian framework. If the reader is interested in studying these concepts in more detail, in *Introduction to Statistical Decision Theory*, *Pratt J., Raiffa H., Schlaifer R., The MIT Press*, it's possible to find many interesting examples and explanations; however, before introducing Bayesian networks, it's useful to define two other essential concepts.

The first concept is called **conditional independence**, and it can be formalized considering two variables A and B, which are conditioned to a third one, C. We say that A and B are conditionally independent given C if:

$$P(A, B|C) = P(A|C)P(B|C)$$

Now, let's suppose we have an event A that is conditioned to a series of causes $C_1, C_2, ..., C_n$; the conditional probability is, therefore, $P(A|C_1, C_2, ..., C_n)$. Applying Bayes' theorem, we get:

$$P(A|C_1, C_2, \ldots, C_n) \propto P(C_1, C_2, \ldots, C_n|A)P(A)$$

If there is conditional independence, the previous expression can be simplified and rewritten as:

$$P(A|C_1, C_2, \ldots, C_n) \propto P(C_1|A)P(C_2|A)\ldots P(C_n|A)P(A) = P(A)\prod_{i=1}^{n} P(C_i|A)$$

This property is fundamental in Naive Bayes classifiers, where we assume that the effect produced by a cause does not influence the other causes. For example, in a spam detector, we could say that the length of the mail and the presence of some particular keywords are independent events, and we only need to compute *P(Length | Spam)* and *P(Keywords | Spam)* without considering the joint probability *P(Length, Keywords | Spam)*.

Another important element is the **chain rule** of probabilities. Let's suppose we have the joint probability $P(X_1, X_2, ..., X_n)$. It can be expressed as:

$$P(X_1, X_2, \ldots, X_n) = P(X_1|X_2, \ldots, X_n)P(X_2, \ldots, X_n)$$

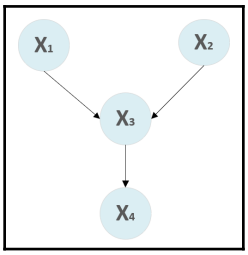
Repeating the procedure with the joint probability on the right side, we get:

$$P(X_1, X_2, \ldots, X_n) = P(X_1|X_2, \ldots, X_n)P(X_2|X_3, \ldots, X_n)\ldots P(X_n) = \prod_{i=1}^{n} P(X_i|X_{i+1}, \ldots, X_n)$$

In this way, it's possible to express a full joint probability as the product of hierarchical conditional probabilities, until the last term, which is a marginal distribution. We are going to use this concept extensively in the next paragraph when exploring Bayesian networks.

Bayesian networks

A **Bayesian network** is a probabilistic model represented by a direct acyclic graph $G = \{V, E\}$, where the vertices are random variables X_i, and the edges determine a conditional dependence among them. In the following diagram, there's an example of simple Bayesian networks with four variables:



Example of Bayesian network

The variable x_4 is dependent on x_3, which is dependent on x_1 and x_2. To describe the network, we need the marginal probabilities $P(x_1)$ and $P(x_2)$ and the conditional probabilities $P(x_3|x_1,x_2)$ and $P(x_4|x_3)$. In fact, using the chain rule, we can derive the full joint probability as:

$$P(x_1, x_2, x_3, x_4) = P(x_4|x_3)P(x_3|x_1, x_2)P(x_2)P(x_1)$$

The previous expression shows an important concept: as the graph is direct and acyclic, each variable is conditionally independent of all other variables that are not successors given its predecessors. To formalize this concept, we can define the function $Predecessors(x_i)$, which returns the set of nodes that influence x_i directly, for example, $Predecessors(x_3) = \{x_1,x_2\}$ (we are using lowercase letters, but we are considering the random variable, not a sample). Using this function, it's possible to write a general expression for the full joint probability of a Bayesian network with N nodes:

$$P(x_1, x_2, \ldots, x_N) = \prod_{i=1}^{N} P\left(x_i|Predecessors(x_i)\right)$$

The general procedure to build a Bayesian network should always start with the first causes, adding their effects one by one, until the last nodes are inserted into the graph. If this rule is not respected, the resulting graph can contain useless relations that can increase the complexity of the model. For example, if x_4 is caused indirectly by both x_1 and x_2, therefore adding the edges $x_1 \rightarrow x_4$ and $x_2 \rightarrow x_4$ could seem a good modeling choice; however, we know that the final influence on x_4 is determined only by the values of x_3, whose probability must be conditioned on x_1 and x_2, hence we can remove the spurious edges. I suggest reading *Introduction to Statistical Decision Theory, Pratt J., Raiffa H., Schlaifer R., The MIT Press* to learn many best practices that should be employed in this procedure.

Sampling from a Bayesian network

Performing a direct inference on a Bayesian network can be a very complex operation when the number of variables and edges is high. For this reason, several sampling methods have been proposed. In this paragraph, we are going to show how to determine the full joint probability sampling from a network using a direct approach, and two MCMC algorithms.

Let's start considering the previous network and, for simplicity, let's assume to have only *Bernoulli* distributions. X_1 and X_2 are modeled as:

$$\begin{cases} X_1 \sim Bernoulli(0.35) \\ X_2 \sim Bernoulli(0.65) \end{cases}$$

The conditional distribution X_3 is defined as:

$$X_3 \sim Bernoulli(p) \;\; with\; p = \begin{cases} 0.75 & if\; x_1 = True\; and\; x_2 = True \\ 0.4 & otherwise \end{cases}$$

While the conditional distribution X_4 is defined as:

$$X_4 \sim Bernoulli(p) \;\; with\; p = \begin{cases} 0.65 & if\; x_3 = True \\ 0.5 & otherwise \end{cases}$$

We can now use a direct sampling to estimate the full joint probability $P(x_1, x_2, x_3, x_4)$ using the chain rule previously introduced.

Direct sampling

With **direct sampling**, our goal is to approximate the full joint probability through a sequence of samples drawn from each conditional distribution. If we assume that the graph is well-structured (without unnecessary edges) and we have *N* variables, the algorithm is made up of the following steps:

1. Initialize the variable $N_{Samples}$.
2. Initialize a vector *S* with shape $(N, N_{Samples})$.
3. Initialize a frequency vector $F_{Samples}$ with shape $(N, N_{Samples})$. In Python, it's better to employ a dictionary where the key is a combination $(x_1, x_2, x_3, ..., x_N)$.
4. For $t=1$ to $N_{Samples}$:
    1. For $i=1$ to N:
        1. Sample from $P(X_i | Predecessors(X_i))$
        2. Store the sample in $S[i, t]$
    2. If $F_{Samples}$ contains the sampled tuple $S[:, t]$:
        1. $F_{Samples}[S[:, t]] \mathrel{+}= 1$
    3. Else:
        1. $F_{Samples}[S[:, t]] = 1$ (both these operations are immediate with

Python dictionaries)

5. Create a vector $P_{Sampled}$ with shape *(N, 1)*.
6. Set $P_{Sampled}[i, 0] = F_{Samples}[i]/N$.

From a mathematical viewpoint, we are first creating a frequency vector $F_{Samples}(x_1, x_2, x_3, ..., x_N; N_{Samples})$ and then we approximate the full joint probability considering $N_{Samples} \rightarrow \infty$:

$$P(x_1, x_2, \ldots, x_N) = \lim_{N_{samples} \to \infty} F_{samples}(x_1, x_2, \ldots, x_N; N_{samples})$$

Example of direct sampling

We can now implement this algorithm in Python. Let's start by defining the sample methods using the NumPy function `np.random.binomial(1, p)`, which draws a sample from a *Bernoulli* distribution with probability `p`:

```
import numpy as np

def X1_sample(p=0.35):
    return np.random.binomial(1, p)

def X2_sample(p=0.65):
    return np.random.binomial(1, p)

def X3_sample(x1, x2, p1=0.75, p2=0.4):
    if x1 == 1 and x2 == 1:
        return np.random.binomial(1, p1)
    else:
        return np.random.binomial(1, p2)
def X4_sample(x3, p1=0.65, p2=0.5):
    if x3 == 1:
        return np.random.binomial(1, p1)
    else:
        return np.random.binomial(1, p2)
```

At this point, we can implement the main cycle. As the variables are Boolean, the total number of probabilities is 16, so we set `Nsamples` to `5000` (smaller values are also acceptable):

```
N = 4
Nsamples = 5000

S = np.zeros((N, Nsamples))
```

```
Fsamples = {}

for t in range(Nsamples):
    x1 = X1_sample()
    x2 = X2_sample()
    x3 = X3_sample(x1, x2)
    x4 = X4_sample(x3)
    sample = (x1, x2, x3, x4)
    if sample in Fsamples:
        Fsamples[sample] += 1
    else:
        Fsamples[sample] = 1
```

When the sampling is complete, it's possible to extract the full joint probability:

```
samples = np.array(list(Fsamples.keys()), dtype=np.bool_)
probabilities = np.array(list(Fsamples.values()), dtype=np.float64) /
Nsamples

for i in range(len(samples)):
    print('P{} = {}'.format(samples[i], probabilities[i]))

P[ True False  True  True] = 0.0286
P[ True  True False  True] = 0.024
P[ True  True  True False] = 0.06
P[False False False False] = 0.0708
P[ True False  True False] = 0.0166
P[False  True  True  True] = 0.1006
P[False False  True  True] = 0.054
...
```

We can also query the model. For example, we could be interested in $P(X_4=True)$. We can do this by looking for all the elements where $X_4=True$, and summing up the relative probabilities:

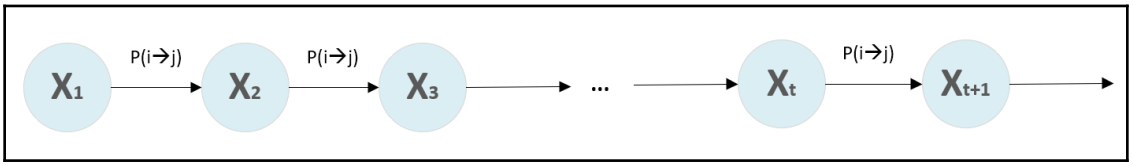
```
p4t = np.argwhere(samples[:, 3]==True)
print(np.sum(probabilities[p4t]))

0.5622
```

This value is coherent with the definition of X_4, which is always $p >= 0.5$. The reader can try to change the values and repeat the simulation.

A gentle introduction to Markov chains

In order to discuss the MCMC algorithms, it's necessary to introduce the concept of Markov chains. In fact, while the direct sample method draws samples without any particular order, the MCMC strategies draw a sequence of samples according to a precise transition probability from a sample to the following one.

Let's consider a time-dependent random variable *X(t)*, and let's assume a discrete time sequence $\mathbf{X_1}$, $\mathbf{X_2}$, ..., $\mathbf{X_t}$, $\mathbf{X_{t+1}}$, ... where $\mathbf{X_t}$ represents the value assumed at time *t*. In the following diagram, there's a schematic representation of this sequence:



Structure of a generic Markov chain

We can suppose to have *N* different states s_i for *i=1..N*, therefore it's possible to consider the probability $P(X_t=s_i | X_{t-1}=s_j, ..., X_1=s_p)$. *X(t)* is defined as a **first-order Markov process** if:

$$P(X_t = s_i | X_{t-1} = s_j, \ldots, X_1 = s_p) = P(X_t = s_i | X_{t-1} = s_j)$$

In other words, in a Markov process (from now on, we omit *first-order*, even if there are cases when it's useful to consider more previous states), the probability that *X(t)* is in a certain state depends only on the state assumed in the previous time instant. Therefore, we can define a **transition probability** for every couple *i*, *j*:

$$P(j \rightarrow i) = P(X_t = s_i | X_{t-1} = s_j)$$

Considering all the couples *(i, j)*, it's also possible to build a transition probability matrix $T(i, j) = P(i \rightarrow j)$. The marginal probability that $X_t=s_i$ using a standard notation is defined as:

$$\pi_i(t) = P(X_t = s_i)$$

At this point, it's easy to prove (**Chapman-Kolmogorov** equation) that:

$$\pi_i(t+1) = \sum_k p(k \rightarrow i)\pi_k(t) \;\Rightarrow\; \bar{\pi}(t+1) = T^T\bar{\pi}(t)$$

In the previous expression, in order to compute $\pi_i(t+1)$, we need to sum over all possible previous states, considering the relative transition probability. This operation can be rewritten in matrix form, using a vector $\pi(t)$ containing all states and the transition probability matrix T (the uppercase superscript T means that the matrix is transposed). The evolution of the chain can be computed recursively:

$$\bar{\pi}(t+1) = T^T \bar{\pi}(t) = T^T \left(T^T \bar{\pi}(t-1)\right) = \ldots = (T^T)^t \bar{\pi}(1)$$

For our purposes, it's important to consider Markov chains that are able to reach a *stationary distribution* π_s:

$$\bar{\pi}_s = T^T \bar{\pi}_s$$

In other words, the state does not depend on the initial condition $\pi(1)$, and it's no longer able to change. The stationary distribution is unique if the underlying Markov process is *ergodic*. This concept means that the process has the same properties if averaged over time (which is often impossible), or averaged vertically (freezing the time) over the states (which is simpler in the majority of cases).

The process of ergodicity for Markov chains is assured by two conditions. The first is aperiodicity for all states, which means that it is impossible to find a positive number p so that the chain returns in the same state sequence after a number of instants equal to a multiple of p. The second condition is that all states must be positive recurrent: this means that, given a random variable $N_{instants}(i)$, describing the number of time instants needed to return to the state s_i, $E[N_{instants}(i)] < \infty$; therefore, potentially, all the states can be revisited in a finite time.

The reason why we need the ergodicity condition, and hence the existence of a unique stationary distribution, is that we are considering the sampling processes modeled as Markov chains, where the next value is sampled according to the current state. The transition from one state to another is done in order to find better samples, as we're going to see in the Metropolis-Hastings sampler, where we can also decide to reject a sample and keep the chain in the same state. For this reason, we need to be sure that the algorithms converge to the unique stable distribution (that approximates the real full joint distribution of our Bayesian network). It's possible to prove that a chain always reaches a stationary distribution if:

$$\forall\, i, j \Rightarrow P(i \to j)\pi_{s_i} = P(j \to i)\pi_{s_j}$$

The previous equation is called detailed balance, and implies the reversibility of the chain. Intuitively, it means that the probability of finding the chain in the state *A* times the probability of a transition to the state *B* is equal to the probability of finding the chain in the state *B* times the probability of a transition to *A*.

For both methods that we are going to discuss, it's possible to prove that they satisfy the previous condition, and therefore their convergence is assured.

Gibbs sampling

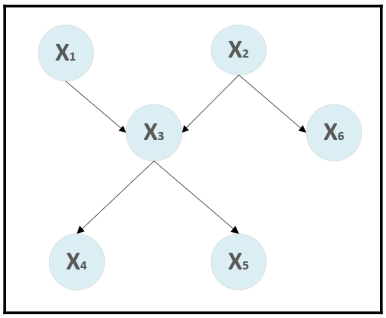
Let's suppose that we want to obtain the full joint probability for a Bayesian network $P(x_1, x_2, x_3, ..., x_N)$; however, the number of variables is large and there's no way to solve this problem easily in a closed form. Moreover, imagine that we would like to get some marginal distribution, such as $P(x_2)$, but to do so we should integrate the full joint probability, and this task is even harder. Gibbs sampling allows approximating of all marginal distributions with an iterative process. If we have N variables, the algorithm proceeds with the following steps:

1. Initialize the variable $N_{Iterations}$
2. Initialize a vector S with shape $(N, N_{Iterations})$
3. Randomly initialize $x_1^{(0)}, x_2^{(0)}, ..., x_N^{(0)}$ (the superscript index is referred to the iteration)
4. For $t=1$ to $N_{Iterations}$:
    1. Sample $x_1^{(t)}$ from $p(x_1 | x_2^{(t-1)}, x_3^{(t-1)}, ..., x_N^{(t-1)})$ and store it in $S[0, t]$
    2. Sample $x_2^{(t)}$ from $p(x_2 | x_1^{(t)}, x_3^{(t-1)}, ..., x_N^{(t-1)})$ and store it in $S[1, t]$
    3. Sample $x_3^{(t)}$ from $p(x_3 | x_1^{(t)}, x_2^{(t)}, ..., x_N^{(t-1)})$ and store it in $S[2, t]$
    4. ...
    5. Sample $x_N^{(t)}$ from $p(x_N | x_1^{(t)}, x_2^{(t)}, ..., x_{N-1}^{(t)})$ and store it in $S[N-1, t]$

At the end of the iterations, vector S will contain $N_{Iterations}$ samples for each distribution. As we need to determine the probabilities, it's necessary to proceed like in the direct sampling algorithm, counting the number of single occurrences and normalizing dividing by $N_{Iterations}$. If the variables are continuous, it's possible to consider intervals, counting how many samples are contained in each of them.

For small networks, this procedure is very similar to direct sampling, except that when working with very large networks, the sampling process could become slow; however, the algorithm can be simplified after introducing the concept of the Markov blanket of X_i, which is the set of random variables that are predecessors, successors, and successors' predecessors of X_i (in some books, they use the terms *parents* and *children*). In a Bayesian network, a variable X_i is a conditional independent of all other variables given its Markov blanket. Therefore, if we define the function $MB(X_i)$, which returns the set of variables in the blanket, the generic sampling step can be rewritten as $p(x_i|MB(X_i))$, and there's no more need to consider all the other variables.

To understand this concept, let's consider the network shown in the following diagram:



Bayesian network for the Gibbs sampling example

The Markov blankets are:

- $MB(X_1) = \{ X_2, X_3 \}$
- $MB(X_2) = \{ X_1, X_3, X_4 \}$
- $MB(X_3) = \{ X_1, X_2, X_4, X_5 \}$
- $MB(X_4) = \{ X_3 \}$
- $MB(X_5) = \{ X_3 \}$
- $MB(X_6) = \{ X_2 \}$

In general, if N is very large, the cardinality of $|MB(X_i)| << N$, thus simplifying the process (the *vanilla* Gibbs sampling needs $N-1$ conditions for each variable). We can prove that the Gibbs sampling generates samples from a Markov chain that is in detailed balance:

$$P(i \to j)\pi_{s_i} = P(x_j|x_1, x_2, \ldots, x_{j-1}, x_{j+1}, \ldots, x_N)P(x_i) =$$
$$= P(x_j, x_i|x_1, x_2, \ldots, x_{j-1}, x_{j+1}, \ldots, x_{i-1}, x_{i+1}, \ldots, x_N) =$$
$$= P(x_i|x_1, x_2, \ldots, x_{i-1}, x_{i+1}, \ldots, x_N)P(x_j) = P(j \to i)\pi_{s_j}$$

Therefore, the procedure converges to the unique stationary distribution. This algorithm is quite simple; however, its performance is not excellent, because the random walks are not tuned up in order to explore the right regions of the state-space, where the probability to find good samples is high. Moreover, the trajectory can also return to bad states, slowing down the whole process. An alternative (also implemented by PyMC3 for continuous random variables) is the **No-U-Turn** algorithm, which we don't discuss in this book. The reader interested in this topic can find a full description in *The No-U-Turn Sampler: Adaptively Setting Path Lengths in Hamiltonian Monte Carlo*, *Hoffmann M. D.*, *Gelman A.*, *arXiv:1111.4246*.

Metropolis-Hastings sampling

We have seen that the full joint probability distribution of a Bayesian network $P(x_1, x_2, x_3, ..., x_N)$ can become intractable when the number of variables is large. The problem can become even harder when it's needed to marginalize it in order to obtain, for example, $P(x_i)$, because it's necessary to integrate a very complex function. The same problem happens when applying the Bayes' theorem in simple cases. Let's suppose we have the expression $p(A|B) = K \cdot P(B|A)P(A)$. I've expressly inserted the normalizing constant K, because if we know it, we can immediately obtain the posterior probability; however, finding it normally requires integrating $P(B|A)P(A)$, and this operation can be impossible in closed form.

The Metropolis-Hastings algorithm can help us in solving this problem. Let's imagine that we need to sample from $P(x_1, x_2, x_3, ..., x_N)$, but we know this distribution up to a normalizing constant, so $P(x_1, x_2, x_3, ..., x_N) \propto g(x_1, x_2, x_3, ..., x_N)$. For simplicity, from now on we collapse all variables into a single vector, so $P(x) \propto g(x)$.

Let's take another distribution $q(x'|x^{(i-1)})$, which is called **candidate-generating distribution**. There are no particular restrictions on this choice, only that q is easy to sample. In some situations, q can be chosen as a function very similar to the distribution $p(x)$, which is our target, while in other cases, it's possible to use a normal distribution with mean equal to $x^{(i-1)}$. As we're going to see, this function acts as a proposal-generator, but we're not obliged to accept all the samples drawn from it therefore, potentially any distribution with the same domain of $P(X)$ can be employed. When a sample is accepted, the Markov chain transitions to the next state, otherwise it remains in the current one. This decisional process is based on the idea that the sampler must explore the most important state-space regions and discard the ones where the probability to find good samples is low.

The algorithm proceeds with the following steps:

1. Initialize the variable $N_{Iterations}$
2. Initialize $x^{(0)}$ randomly
3. For $t=1$ to $N_{Iterations}$:
    1. Draw a candidate sample x' from $q(x' | x^{(i-1)})$
    2. Compute the following value:

$$\alpha = \frac{g(x')q(x^{(t-1)}|x')}{g(x^{(t-1)})q(x'|x^{(t-1)})}$$

    3. If $\alpha \geq 1$:
        1. Accept the sample $x^{(t)} = x'$
    4. Else if $0 < \alpha < 1$:
        1. Accept the sample $x^{(t)} = x'$ with probability α; or
        2. Reject the sample x' setting $x^{(t)} = x^{(t-1)}$ with probability $1 - \alpha$

It's possible to prove (the proof will be omitted, but it's available in *Markov Chain Monte Carlo and Gibbs Sampling, Walsh B., Lecture Notes for EEB 596z*) that the transition probability of the Metropolis-Hastings algorithm satisfies the detailed balance equation, and therefore the algorithm converges to the true posterior distribution.

Example of Metropolis-Hastings sampling

We can implement this algorithm to find the posterior distribution $P(A|B)$ given the product of $P(B|A)$ and $P(A)$, without considering the normalizing constant that requires a complex integration.

Let's suppose that:

$$\begin{cases} p(A) \sim Exponential(\lambda = 0.1) \\ p(B|A) \sim Wald(\mu = 1.0, \lambda = 0.2) \end{cases}$$

Therefore, the resulting $g(x)$ is:

$$g(x) = 0.1e^{-0.1x}\sqrt{\frac{0.2}{2\pi x^3}}e^{-\frac{0.2(x-1)^2}{2x}}$$

To solve this problem, we adopt the random walk Metropolis-Hastings, which consists of choosing $q \sim Normal(\mu=x^{(t-1)})$. This choice allows simplifying the value α, because the two terms $q(x^{(t-1)}|x')$ and $q(x'|x^{(t-1)})$ are equal (thanks to the symmetry around the vertical axis passing through x_{mean}) and can be canceled out, so α becomes the ratio between $g(x')$ and $g(x^{(t-1)})$.

The first thing is defining the functions:

```
import numpy as np

def prior(x):
    return 0.1 * np.exp(-0.1 * x)

def likelihood(x):
    a = np.sqrt(0.2 / (2.0 * np.pi * np.power(x, 3)))
    b = - (0.2 * np.power(x - 1.0, 2)) / (2.0 * x)
    return a * np.exp(b)

def g(x):
    return likelihood(x) * prior(x)

def q(xp):
    return np.random.normal(xp)
```

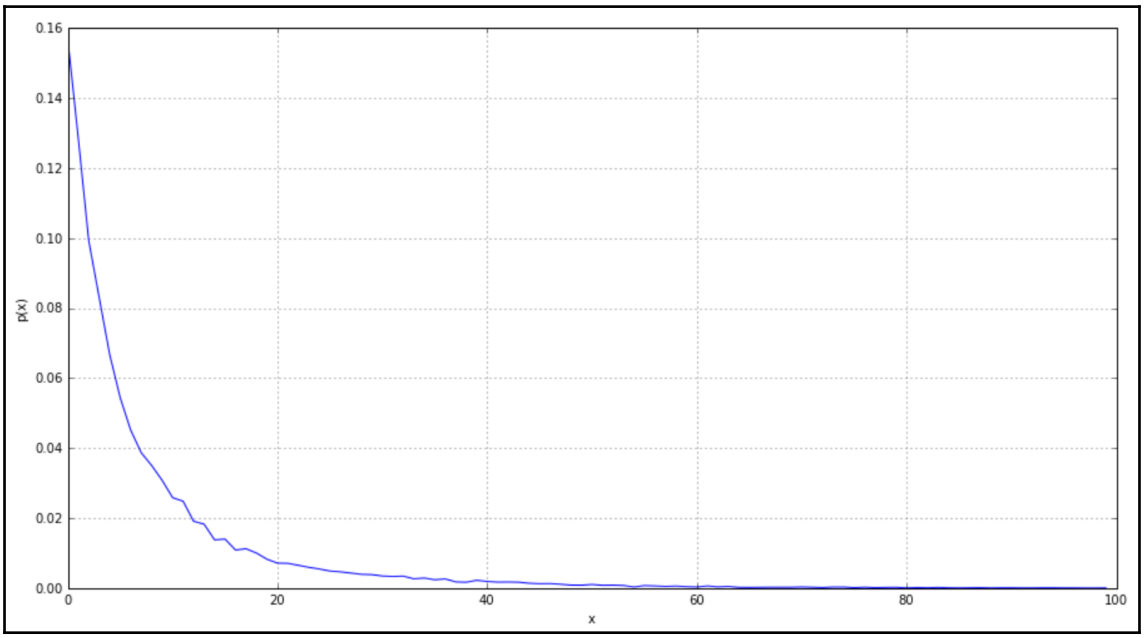
Now, we can start our sampling process with 100,000 iterations and $x^{(0)} = 1.0$:

```
nb_iterations = 100000
x = 1.0
samples = []

for i in range(nb_iterations):
    xc = q(x)
    alpha = g(xc) / g(x)
    if np.isnan(alpha):
        continue
    if alpha >= 1:
        samples.append(xc)
        x = xc
    else:
        if np.random.uniform(0.0, 1.0) < alpha:
            samples.append(xc)
            x = xc
```

To get a representation of the posterior distribution, we need to create a histogram through the NumPy function `np.histogram()`, which accepts an array of values and the number of desired intervals (`bins`); in our case, we set `100` intervals:

```
hist, _ = np.histogram(samples, bins=100)
hist_p = hist / len(samples)
```

The resulting plot of *p(x)* is shown in the following graph:

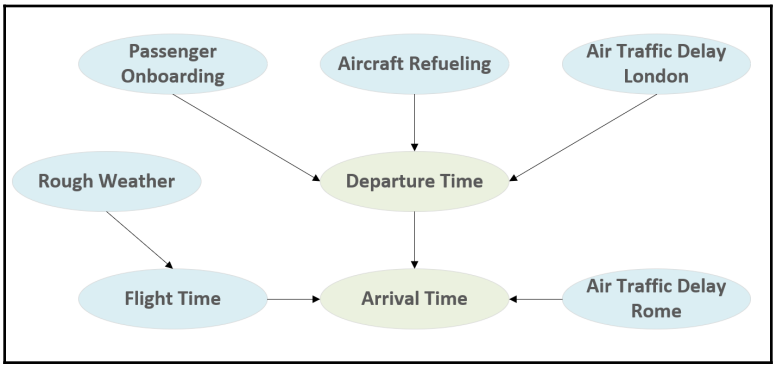


Sampled probability density function

Sampling example using PyMC3

PyMC3 is a powerful Python Bayesian framework that relies on Theano to perform high-speed computations (see the information box at the end of this paragraph for the installation instructions). It implements all the most important continuous and discrete distributions, and performs the sampling process mainly using the No-U-Turn and Metropolis-Hastings algorithms. For all the details about the API (distributions, functions, and plotting utilities), I suggest visiting the documentation home page `http://docs.pymc.io/index.html`, where it's also possible to find some very intuitive tutorials.

The example we want to model and simulate is based on this scenario: a daily flight from London to Rome has a scheduled departure time at 12:00 am, and a standard flight time of two hours. We need to organize the operations at the destination airport, but we don't want to allocate resources when the plane hasn't landed yet. Therefore, we want to model the process using a Bayesian network and considering some common factors that can influence the arrival time. In particular, we know that the onboarding process can be longer than expected, as well as the refueling one, even if they are carried out in parallel. London air traffic control can also impose a delay, and the same can happen when the plane is approaching Rome. We also know that the presence of rough weather can cause another delay due to a change of route. We can summarize this analysis with the following plot:



Bayesian network representing the air traffic control problem

Considering our experience, we decide to model the random variables using the following distributions:

- *Passenger onboarding ~ Wald(μ=0.5, λ=0.2)*
- *Refueling ~ Wald(μ=0.25, λ=0.5)*
- *Departure traffic delay ~ Wald(μ=0.1, λ=0.2)*
- *Arrival traffic delay ~ Wald(μ=0.1, λ=0.2)*
- *Departure time = 12 + Departure traffic delay + max(Passenger onboarding, Refueling)*
- *Rough weather ~ Bernoulli(p=0.35)*
- *Flight time ~ Exponential(λ=0.5 - (0.1 · Rough weather))* (The output of a Bernoulli distribution is *0* or *1* corresponding to False and True)
- *Arrival time = Departure time + Flight time + Arrival traffic delay*

The probability density functions are:

$$\begin{cases} f_{Wald}(x;\mu;\lambda) = \sqrt{\frac{\lambda}{2\pi x^3}} e^{-\frac{\lambda(x-\mu)^2}{2\mu^2 x}} \\ f_{Exponential}(x;\lambda) = \begin{cases} \lambda e^{-\lambda x} \;\; if \;\; x \geqslant 0 \\ 0 \;\; otherwise \end{cases} \\ f_{Bernoulli}(x;p) = \begin{cases} p \;\; if \;\; x = 1 \\ 1 - p \;\; if \;\; x = 0 \end{cases} \end{cases}$$

`Departure Time` and `Arrival Time` are functions of random variables, and the parameter λ of `Flight Time` is also a function of `Rough Weather`.

Even if the model is not very complex, the direct inference is rather inefficient, and therefore we want to simulate the process using PyMC3.

The first step is to create a `model` instance:

```
import pymc3 as pm

 model = pm.Model()
```

From now on, all operations must be performed using the context manager provided by the `model` variable. We can now set up all the random variables of our Bayesian network:

```
import pymc3.distributions.continuous as pmc
import pymc3.distributions.discrete as pmd
import pymc3.math as pmm

with model:
    passenger_onboarding = pmc.Wald('Passenger Onboarding', mu=0.5,
lam=0.2)
    refueling = pmc.Wald('Refueling', mu=0.25, lam=0.5)
    departure_traffic_delay = pmc.Wald('Departure Traffic Delay', mu=0.1,
lam=0.2)
    departure_time = pm.Deterministic('Departure Time',
                                      12.0 + departure_traffic_delay +
                                      pmm.switch(passenger_onboarding >=
refueling,
                                                 passenger_onboarding,
                                                 refueling))
    rough_weather = pmd.Bernoulli('Rough Weather', p=0.35)
    flight_time = pmc.Exponential('Flight Time', lam=0.5 - (0.1 *
```

```
rough_weather))
    arrival_traffic_delay = pmc.Wald('Arrival Traffic Delay', mu=0.1,
lam=0.2)
    arrival_time = pm.Deterministic('Arrival time',
                                    departure_time +
                                    flight_time +
                                    arrival_traffic_delay)
```

We have imported two namespaces, `pymc3.distributions.continuous` and `pymc3.distributions.discrete`, because we are using both kinds of variable. Wald and exponential are continuous distributions, while `Bernoulli` is discrete. In the first three rows, we declare the variables `passenger_onboarding`, `refueling`, and `departure_traffic_delay`. The structure is always the same: we need to specify the class corresponding to the desired distribution, passing the name of the variable and all the required parameters.

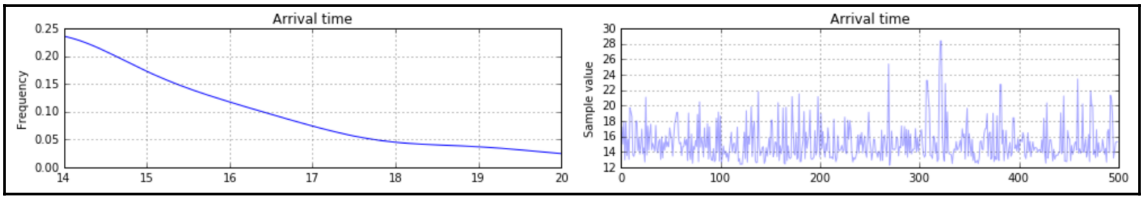
The `departure_time` variable is declared as `pm.Deterministic`. In PyMC3, this means that, once all the random elements have been set, its value becomes completely determined. Indeed, if we sample from `departure_traffic_delay`, `passenger_onboarding`, and `refueling`, we get a determined value for `departure_time`. In this declaration, we've also used the utility function `pmm.switch`, which operates a binary choice based on its first parameter (for example, if $A > B$, return A, else return B).

The other variables are very similar, except for `flight_time`, which is an exponential variable with a parameter λ, which is a function of another variable (`rough_weather`). As a Bernoulli variable outputs *1* with probability p and *0* with probability $1 - p$, $\lambda = 0.4$ if there's rough weather, and *0.5* otherwise.

Once the model has been set up, it's possible to simulate it through a sampling process. PyMC3 picks the best sampler automatically, according to the type of variables. As the model is not very complex, we can limit the process to `500` samples:

```
nb_samples = 500

with model:
    samples = pm.sample(draws=nb_samples, random_seed=1000)
```

The output can be analyzed using the built-in `pm.traceplot()` function, which generates the plots for each of the sample's variables. The following graph shows the detail of one of them:



Distribution and samples for the arrival time random variable

The right column shown the samples generated for the random variable (in this case, the arrival time), while the left column shows the relative frequencies. This plot can be useful to have a visual confirmation of our initial ideas; in fact, the arrival time has the majority of its mass concentrated in the interval 14:00 to 16:00 (the numbers are always decimal, so it's necessary to convert the times); however, we should integrate to get the probabilities. Instead, through the `pm.summary()` function, PyMC3 provides a statistical summary that can help us in making the right decisions. In the following snippet, the output containing the summary of a single variable is shown:

```
pm.summary(samples)

...

Arrival time:

  Mean             SD               MC Error         95% HPD interval
  -------------------------------------------------------------------
  15.174           2.670            0.102            [12.174, 20.484]

  Posterior quantiles:
  2.5            25             50             75             97.5
  |--------------|==============|==============|--------------|
  12.492         13.459         14.419         16.073         22.557
```

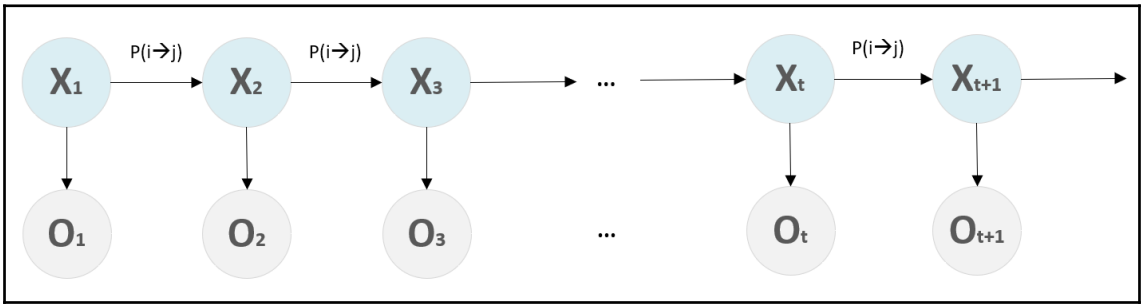
For each variable, it contains mean, standard deviation, Monte Carlo error, 95% highest posterior density interval, and the posterior quantiles. In our case, we know that the plane will land at about 15:10 (`15.174`).

This is only a very simple example to show the power of Bayesian networks. For deep insight, I suggest the book *Introduction to Statistical Decision Theory*, *Pratt J., Raiffa H., Schlaifer R., The MIT Press*, where it's possible to study different Bayesian applications that are out of the scope of this book.

PyMC3 (`http://docs.pymc.io/index.html`) can be installed using the `pip install -U pymc3` command. As it requires Theano (which is installed automatically), it's also necessary to provide it with a C/C++ compiler. I suggest using distributions such as Anaconda (`https://www.anaconda.com/download/`), which allows installing MinGW through the `conda install -c anaconda mingw` command. For any problems, on the website you can find detailed installation instructions. For further information on how to configure Theano to work with GPU support (the default installation is based on CPU NumPy algorithms), please visit this page: `http://deeplearning.net/software/theano/`.

Hidden Markov Models (HMMs)

Let's consider a stochastic process *X(t)* that can assume *N* different states: s_1, s_2, ..., s_N with first-order Markov chain dynamics. Let's also suppose that we cannot observe the state of *X(t)*, but we have access to another process *O(t)*, connected to *X(t)*, which produces observable outputs (often known as **emissions**). The resulting process is called a **Hidden Markov Model** (**HMM**), and a generic schema is shown in the following diagram:



Structure of a generic Hidden Markov Model

For each hidden state s_i, we need to define a transition probability $P(i \rightarrow j)$, normally represented as a matrix if the variable is discrete. For the Markov assumption, we have:

$$P(j \rightarrow i) = P(X_t = s_i | X_{t-1} = s_j)$$

Moreover, given a sequence of observations o_1, o_2, ..., o_M, we also assume the following assumption about the independence of the **emission probability**:

$$P(o_i | o_1, o_2, \ldots, o_k, x_1, x_2, \ldots, x_k) = P(o_i | x_i)$$

In other words, the probability of the observation o_i (in this case, we mean the value at time i) is conditioned only by the state of the hidden variable at time i (x_i). Conventionally, the first state x_0 and the last one x_{Ending} are never emittied, and therefore all the sequences start with the index *1* and end with an extra timestep corresponding to the final state.

HMMs can be employed in all those contexts where it's impossible to measure the state of a system (we can only model it as a stochastic variable with a known transition probability), but it's possible to access some data connected to it. An example can be a complex engine that is made up of a large number of parts. We can define some internal states and learn a transition probability matrix (we're going to learn how to do that), but we can only receive measures provided by specific sensors.

Sometimes, even if not extremely realistic, but it's useful to include the Markov assumption and the emission probability independence into our model. The latter can be justified considering that we can sample all the *peak* emissions corresponding to precise states and, as the random process *O(t)* is implicitly dependent on *X(t)*, it's not unreasonable to think of it like a *pursuer* of *X(t)*.

The Markov assumption holds for many real-life processes if either they are naturally first-order Markov ones, or if the states contain all the history needed to justify a transition. In other words, in many cases, if the state is *A*, then there's a transit to *B* and finally to *C*. We assume that when in *C*, the system moved from a state *(B)* that carries a part of the information provided by *A*.

For example, if we are filling a tank, we can measure the level (the state of our system) at time *t*, *t+1*, ... If the water flow is modeled by a random variable because we don't have a stabilizer, we can find the probability that the water has reached a certain level at time *t*, $p(L_t=x | L_{t-1})$. Of course, it doesn't make sense to condition over all the previous states, because if the level is, for example, 80 m at time t-1, all the information needed to determine the probability of a new level (state) at time *t* is already contained in this state (80 m).

At this point, we can start analyzing how to train a hidden Markov model, and how to determine the most likely hidden states given a sequence of observations. For simplicity, we call *A* the transition probability matrix, and *B* the matrix containing all $P(o_i|x_i)$. The resulting model can be determined by the knowledge of those elements: *HMM = { A, B }*.

Forward-backward algorithm

The **forward-backward algorithm** is a simple but effective method to find the transition probability matrix *T* given a sequence of observations $o_1, o_2, ..., o_t$. The first step is called the *forward phase*, and consists of determining the probability of a sequence of observations $P(o_1, o_2, ..., o_{Sequence\ Length}|A, B)$. This piece of information can be directly useful if we need to know the likelihood of a sequence and it's necessary, together with the *backward phase*, to estimate the structure (*A* and *B*) of the underlying HMM.

Both algorithms are based on the concept of dynamic programming, which consists of splitting a complex problem into sub-problems that can be easily solved, and reusing the solutions to solve more complex steps in a recursive/iterative fashion. For further information on this, please refer to *Dynamic Programming and Markov Process, Ronald A. Howard, The MIT Press*.

Forward phase

If we call p_{ij} the transition probability $P(i \rightarrow j)$, we define a recursive procedure considering the following probability:

$$f_t^i = P(o_1, o_2, \ldots, o_t, x_t = i|A, B)$$

The variable f_t^i represents the probability that the HMM is in the state *i* (at time *t*) after *t* observations (from *1* to *t*). Considering the HMM assumptions, we can state that f_t^i depends on all possible f_{t-1}^j. More precisely, we have:

$$f_t^i = \sum_j f_{t-1}^j p_{ji} P(o_t|x_j)$$

With this process, we are considering that the HMM can reach any of the states at time *t-1* (with the first *t-1* observations), and transition to the state *i* at time *t* with probability p_{ji}. We need also to consider the emission probability for the final state o_t conditioned to each of the possible previous states.

For definition, the initial and ending states are not emitting. It means that we can write any sequence of observations as *0*, o_1, o_2, ..., $o_{Sequence\ Length}$, *0*, where the first and the final values are null. The procedure starts with computing the forward message at time *1*:

$$f_1^i = p_{0i} P(o_1 | x_0)$$

The non-emitting ending state must be also considered:

$$f_{SequenceLength}^{Ending} = \sum_i f_{SequenceLength-1}^i p_{iEnding}$$

The expression for the last state x_{Ending} is interpreted here as the index of the ending state in both *A* and *B* matrices. For example, we indicate p_{ij} as *A[i, j]*, meaning the transition probability at a generic time instant from the state $x_t = i$ to the state $x_{t+1} = j$. In the same way, $p_{iEnding}$ is represented as *A[i, x_{Ending}]*, meaning the transition probability from the penultimate state $x_{Sequence\ Length-1} = i$ to the ending one $x_{Sequence\ Length} = Ending\ State$.

The Forward algorithm can, therefore, be summarized in the following steps (we assume to have *N* states, hence we need to allocate *N*+2 positions, considering the initial and the ending states):

1. Initialization of a *Forward* vector with shape (*N + 2*, *Sequence Length*).
2. Initialization of *A* (transition probability matrix) with shape (*N, N*). Each element is $P(x_i | x_j)$.
3. Initialization of *B* with shape (*Sequence Length*, *N*). Each element is $P(o_i | x_j)$.
4. For *i=1* to *N*:
    1. Set *Forward[i, 1] = A[0, i] · B[1, i]*
5. For *t=2* to *Sequence Length-1*:
    1. For *i=1* to *N*:
        1. Set *S = 0*
    2. For *j=1* to *N*:
        1. Set *S = S + Forward[j, t-1] · A[j, i] · B[t, i]*
    3. Set *Forward[i, t] = S*

6. Set $S = 0$.
7. For $i=1$ to N:
    1. Set $S = S + Forward[i, Sequence\ Length] \cdot A[i, x_{Ending}]$
8. Set $Forward[x_{Ending}, Sequence\ Length] = S$.

Now it should be clear that the name **forward** derives from the procedure to propagate the information from the previous step to the next one, until the ending state, which is not emittied.

Backward phase

During the **backward phase**, we need to compute the probability of a sequence starting at time $t+1$: $o_{t+1}, o_{t+2}, ..., o_{Sequence\ Length}$, given that the state at time t is i. Just like we have done before, we define the following probability:

$$b_t^i = P(o_{t+1}, o_{t+2}, \ldots, o_{SequenceLength} | x_t = i, A, B)$$

The backward algorithm is very similar to the forward one, but in this case, we need to move in the opposite direction, assuming we know that the state at time t is i. The first state to consider is the last one x_{Ending}, which is not emitting, like the initial state; therefore we have:

$$b_{SequenceLenght}^i = p_{iEnding}$$

We terminate the recursion with the initial state:

$$b_1^0 = \sum_i b_1^i p_{0i} P(o_1 | x_i)$$

The steps are the following ones:

1. Initialization of a vector *Backward* with shape $(N + 2, Sequence\ Length)$.
2. Initialization of A (transition probability matrix) with shape (N, N). Each element is $P(x_i | x_j)$.
3. Initialization of B with shape $(Sequence\ Length, N)$. Each element is $P(o_i | x_j)$.

4. For *i=1* to *N*:
    1. Set *Backward[x_{Endind}, Sequence Length] = A[i, x_{Endind}]*
5. For *t=Sequence Length-1* to *1*:
    1. For *i=1* to *N*:
        1. Set *S = 0*
        2. For *j=1* to *N*
            1. Set *S = S + Backward[j, t+1] · A[j, i] · B[t+1, i]*
        3. Set *Backward[i, t] = S*
6. Set *S = 0*.
7. For *i=1* to *N*:
    1. Set *S = S + Backward[i, 1] · A[0, i] · B[1, i]*
8. Set *Backward[0, 1] = S*.

HMM parameter estimation

Now that we have defined both the forward and the backward algorithms, we can use them to estimate the structure of the underlying HMM. The procedure is an application of the Expectation-Maximization algorithm, which will be discussed in the next chapter, `Chapter 5`, *EM Algorithm and Applications*, and its goal can be summarized as defining how we want to estimate the values of *A* and *B*. If we define *N(i, j)* as the number of transitions from the state *i* to the state *j*, and *N(i)* the total number of transitions from the state *i*, we can approximate the transition probability *P(i → j)* with:

$$\tilde{a}_{ij} = \widetilde{P}(i \to j) = \frac{E[N(i,j)]}{E[N(i)]}$$

In the same way, if we define *M(i, p)* the number of times we have observed the emission o_p in the state *i*, we can approximate the emission probability $P(o_p|x_i)$ with:

$$\tilde{b}_{ip} = \widetilde{P}(o_p|x_i) = \frac{E[M(i,p)]}{E[N(i)]}$$

Let's start with the estimation of the transition probability matrix *A*. If we consider the probability that the HMM is in the state *i* at time *t*, and in the state *j* at time *t+1* given the observations, we have:

$$\widetilde{\alpha}_{ij}^{t} = P(x_t = i, x_{t+1} = j | o_1, o_2, \ldots, o_{SequenceLength}, A, B)$$

We can compute this probability using the forward and backward algorithms, given a sequence of observations $o_1, o_2, \ldots, o_{Sequence\ Length}$. In fact, we can use both the forward message f_t^i, which is the probability that the HMM is in the state *i* after *t* observations, and the backward message $b_{t+1}^{\ j}$, which is the probability of a sequence $o_{t+1}, o_{t+1}, \ldots, o_{Sequence\ Length}$ starting at time *t+1*, given that the HMM is in state *j* at time *t+1*. Of course, we need also to include the emission probability and the transition probability p_{ij}, which is what we are estimating. The algorithm, in fact, starts with a random hypothesis and iterates until the values of *A* become stable. The estimation α_{ij} at time *t* is equal to:

$$\widetilde{\alpha}_{ij}^{t} = \frac{f_t^i p_{ij} b_{t+1}^j P(o_{t+1} | x_j)}{f_{SequenceLength}^{Ending}}$$

In this context, we are omitting the full proof due to its complexity; however, the reader can find it in *A tutorial on hidden Markov models and selected applications in speech recognition, Rabiner L. R., Proceedings of the IEEE 77.2.*

To compute the emission probabilities, it's easier to start with the probability of being in the state *i* at time *t* given the sequence of observations:

$$\tilde{\beta}_i^t = P(x_t = i | o_1, o_2, \ldots, o_{SequenceLength}, A, B)$$

In this case, the computation is immediate, because we can multiply the forward and backward messages computed at the same time *t* and state *i* (remember that considering the observations, the backward message is conditioned to $x_t = i$, while the forward message computes the probability of the observations joined with $x_t = i$. Hence, the multiplication is the unnormalized probability of being in the state *i* at time *t*). Therefore, we have:

$$\tilde{\beta}_i^t = \frac{f_t^i b_t^i}{f_{SequenceLength}^{Ending}}$$

The proof of how the normalizing constant is obtained can be found in the aforementioned paper. We can now plug these expressions to the estimation of a_{ij} and b_{ip}:

$$\begin{cases} \tilde{a}_{ij} = \dfrac{\sum_{t=1}^{SequenceLength-1} \tilde{\alpha}_{ij}^{t}}{\sum_{t=1}^{SequenceLength-1} \sum_{j=1}^{N} \tilde{\alpha}_{ij}^{t}} \\ \tilde{b}_{ip} = \dfrac{\sum_{t=1}^{SequenceLength} \tilde{\beta}_{i}^{t} \cdot I_{o_t=p}}{\sum_{t=1}^{SequenceLength} \tilde{\beta}_{i}^{t}} \end{cases}$$

In the numerator of the second formula, we adopted the indicator function (it's *1* only if the condition is true, *0* otherwise) to limit the sum only where those elements are $o_t = p$. During an iteration *k*, p_{ij} is the estimated value a_{ij} found in the previous iteration *k-1*.

The algorithm is based on the following steps:

1. Randomly initialize the matrices *A* and *B*
2. Initialize a tolerance variable *Tol* (for example, *Tol = 0.001*)
3. While $Norm(A^k - A^{k-1}) > Tol$ and $Norm(B^k - B^{k-1}) > Tol$ (*k* is the iteration index):
    1. For *t=1* to *Sequence Length-1*:
        1. For *i=1* to *N*:
            1. For *j=1* to *N*:
                1. Compute α^{t}_{ij}
            2. Compute β^{t}_{i}
    2. Compute the estimations of a_{ij} and b_{ip} and store them in A^k

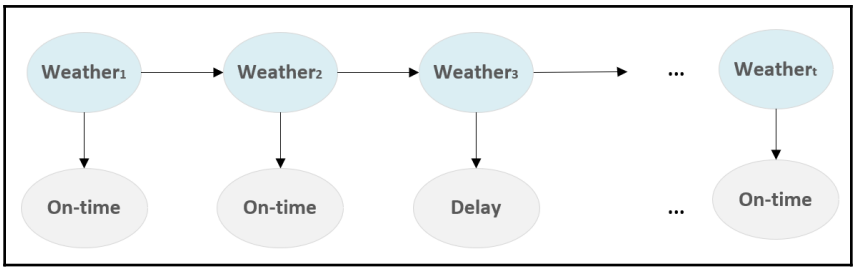
Alternatively, it's possible to fix the number of iterations, even if the best solution is using both a tolerance and a maximum number of iterations, to terminate the process when the first condition is met.

Example of HMM training with hmmlearn

For this example, we are going to use hmmlearn, which is a package for HMM computations (see the information box at the end of this section for further details). For simplicity, let's consider the airport example discussed in the paragraph about the Bayesian networks, and let's suppose we have a single hidden variable that represents the weather (of course, this is not a real hidden variable!), modeled as a multinomial distribution with two components (good and rough).

We observe the arrival time of our flight London-Rome (which partially depends on the weather conditions), and we want to train an HMM to infer future states and compute the posterior probability of hidden states corresponding to a given sequence.

The schema for our example is shown in the following diagram:



HMM for the weather-arrival delay problem

Let's start by defining our observation vector. As we have two states, its values will be `0` and `1`. Let's assume that `0` means **On-time** and `1` means **Delay**:

```
import numpy as np

observations = np.array([[0], [1], [1], [0], [1], [1], [1], [0], [1],
                         [0], [0], [0], [1], [0], [1], [1], [0], [1],
                         [0], [0], [1], [0], [1], [0], [0], [0], [1],
                         [0], [1], [0], [1], [0], [0], [0], [0], [0]],
dtype=np.int32)
```

We have 35 consecutive observations whose values are either `0` or `1`.

To build the HMM, we are going to use the `MultinomialHMM` class, with `n_components=2`, `n_iter=100`, and `random_state=1000` (it's important to always use the same seed to avoid differences in the results). The number of iterations is sometimes hard to determine; for this reason, hmmlearn provides a utility `ConvergenceMonitor` class which can be checked to be sure that the algorithm has successfully converged.

Now we can train our model using the `fit()` method, passing as argument the list of observations (the array must be always bidimensional with shape *Sequence Length* × $N_{Components}$):

```
from hmmlearn import hmm

hmm_model = hmm.MultinomialHMM(n_components=2, n_iter=100,
random_state=1000)
```

```
hmm_model.fit(observations)

print(hmm_model.monitor_.converged)
True
```

The process is very fast, and the monitor (available as instance variable `monitor`) has confirmed the convergence. If the model is very big and needs to be retrained, it's also possible to check smaller values of `n_iter`). Once the model is trained, we can immediately visualize the transition probability matrix, which is available as an instance variable `transmat_`:

```
print(hmm_model.transmat_)

[[ 0.0025384   0.9974616 ]
 [ 0.69191905  0.30808095]]
```

We can interpret these values as saying that the probability to transition from `0` (good weather) to `1` (rough weather) is higher (p_{01} is close to *1*) than the opposite, and it's more likely to remain in state `1` than in state `0` (p_{00} is almost null). We could deduce that the observations have been collected during the winter period! After explaining the Viterbi algorithm in the next paragraph, we can also check, given some observations, what the most likely hidden state sequence is.

hmmlearn (`http://hmmlearn.readthedocs.io/en/latest/index.html`) is a framework originally built to be a part of Scikit-Learn. It supports multinomial and Gaussian HMM, and allows training and inferring using the most common algorithms. It can be installed using the `pip install hmmlearn` command.

Viterbi algorithm

The **Viterbi algorithm** is one of most common decoding algorithms for HMM. Its goal is to find the most likely hidden state sequence corresponding to a series of observations. The structure is very similar to the forward algorithm, but instead of computing the probability of a sequence of observations joined with the state at the last time instant, this algorithm looks for:

$$v_t^i = max_{x_j} P(o_1, o_2, \ldots, o_t, x_1, x_2, \ldots, x_{t-1}, x_t = i | A, B)$$

The variable v_t^i represents that maximum probability of the given observation sequence joint with $x_t = i$, considering all possible hidden state paths (from time instant *1* to *t-1*). We can compute v_t^i recursively by evaluating all the v_{t-1}^j multiplied by the corresponding transition probabilities p_{ji} and emission probability $P(o_t|x_i)$, and always picking the maximum overall possible values of *j*:

$$v_t^i = max_j v_{t-1}^j p_{ji} P(o_t|x_i)$$

The algorithm is based on a backtracking approach, using a backpointer bp_t^i whose recursive expression is the same as v_t^i, but with the *argmax* function instead of *max*:

$$bp_t^i = argmax_j v_{t-1}^j p_{ji} P(o_t|x_i)$$

Therefore, bp_t^i represents the partial sequence of hidden states $x_1, x_2, ..., x_{t-1}$ that maximizes v_t^i. During the recursion, we add the timesteps one by one, so the previous path could be invalidated by the last observation. That's why we need to backtrack the partial result and replace the sequence built at time *t* that doesn't maximize v_{t+1}^i anymore.

The algorithm is based on the following steps (like in the other cases, the initial and ending states are not emitting):

1. Initialization of a vector *V* with shape *(N + 2, Sequence Length)*.
2. Initialization of a vector *BP* with shape *(N + 2, Sequence Length)*.
3. Initialization of *A* (transition probability matrix) with shape *(N, N)*. Each element is $P(x_i|x_j)$.
4. Initialization of *B* with shape *(Sequence Length, N)*. Each element is $P(o_i|x_j)$.
5. For *i=1* to *N*:
    1. Set *V[i, 1] = A[i, 0] · B[1, i]*
    2. *BP[i, 1]* = Null (or any other value that cannot be interpreted as a state)
6. For *t=1* to *Sequence Length*:
    1. For *i=1* to *N*:
        1. Set $V[i, t] = max_j V[j, t-1] \cdot A[j, i] \cdot B[t, i]$
        2. Set $BP[i, t] = argmax_j V[j, t-1] \cdot A[j, i] \cdot B[t, i]$
7. Set $V[x_{Endind}, Sequence\ Length] = max_j V[j, Sequence\ Length] \cdot A[j, x_{Endind}]$.

8. Set $BP[x_{Endind}, Sequence\ Length] = argmax_j\ V[j, Sequence\ Length] \cdot A[j, x_{Endind}]$.
9. Reverse *BP*.

The output of the Viterbi algorithm is a tuple with the most likely sequence *BP*, and the corresponding probabilities *V*.

Finding the most likely hidden state sequence with hmmlearn

At this point, we can continue with the previous example, using our model to find the most likely hidden state sequence given a set of possible observations. We can use either the `decode()` method or the `predict()` method. The first one returns the log probability of the whole sequence and the sequence itself; however, they all use the Viterbi algorithm as a default decoder:

```
sequence = np.array([[1], [1], [1], [0], [1], [1], [1], [0], [1],
                     [0], [1], [0], [1], [0], [1], [1], [0], [1],
                     [1], [0], [1], [0], [1], [0], [1], [0], [1],
                     [1], [1], [0], [0], [1], [1], [0], [1], [1]],
dtype=np.int32)

lp, hs = hmm_model.decode(sequence)

print(hs)
[0 1 1 0 1 1 1 0 1 0 1 0 1 0 1 1 0 1 1 0 1 0 1 0 1 0 1 1 1 1 0 1 1 0 1 1]

print(lp)
-30.489992468878615
```

The sequence is coherent with the transition probability matrix; in fact, it's more likely the persistence of rough weather (`1`) than the opposite. As a consequence, the transition from `1` to X is less likely than the one from `0` to `1`. The choice of state is made by selecting the highest probability; however, in some cases, the differences are minimal (in our example, it can happen to have *p = [0.49, 0.51]*, meaning that there's a high error chance), so it's useful to check the posterior probabilities for all the states in the sequence:

```
pp = hmm_model.predict_proba(sequence)
print(pp)

[[  1.00000000e+00   5.05351938e-19]
 [  3.76687160e-05   9.99962331e-01]
 [  1.31242036e-03   9.98687580e-01]
 [  9.60384736e-01   3.96152641e-02]
```

```
 [  1.27156616e-03   9.98728434e-01]
 [  3.21353749e-02   9.67864625e-01]
 [  1.23481962e-03   9.98765180e-01]

...
```

In our case, there are a couple of states that have $p \sim [0.495, 0.505]$, so even if the output state is *1* (rough weather), it's also useful to consider a moderate probability to observe good weather. In general, if a sequence is coherent with the transition probability previously learned (or manually input), those cases are not very common. I suggest trying different configurations and observations sequences, and to also assess the probabilities for the *strangest* situations (like a sequence of zero second). At that point, it's possible to retrain the model and recheck the new evidence has been correctly processed.

Summary

In this chapter, we have introduced Bayesian networks, describing their structure and relations. We have seen how it's possible to build a network to model a probabilistic scenario where some elements can influence the probability of others. We have also described how to obtain the full joint probability using the most common sampling methods, which allow reducing the computational complexity through an approximation.

The most common sampling methods belong to the family of MCMC algorithms, which model the transition probability from a sample to another one as a first-order Markov chain. In particular, the Gibbs sampler is based on the assumption that it's easier to sample from conditional distribution than work directly with the full joint probability. The method is very easy to implement, but it has some performance drawbacks that can be avoided by adopting more complex strategies. The Metropolis-Hastings sampler, instead, works with a candidate-generating distribution and a criterion to accept or reject the samples. Both methods satisfy the detailed balance equation, which guarantees the convergence (the underlying Markov chain will reach the unique stationary distribution).

In the last part of the chapter, we introduced HMMs, which allow modeling time sequences based on observations corresponding to a series of hidden states. The main concept of such models, in fact, is the presence of unobservable states that condition the emission of a particular observation (which is observable). We have discussed the main assumptions and how to build, train, and infer from a model. In particular, the Forward-Backward algorithm can be employed when it's necessary to learn the transition probability matrix and the emission probabilities, while the Viterbi algorithm is adopted to find the most likely hidden state sequence given a set of consecutive observations.

In the next chapter, `Chapter 5`, *EM Algorithm and Applications*, we're going to briefly discuss the Expectation-Maximization algorithm, focusing on some important applications based on the **Maximum Likelihood Estimation** (**MLE**) approach.

5
EM Algorithm and Applications

In this chapter, we are going to introduce a very important algorithmic framework for many statistical learning tasks: the EM algorithm. Contrary to its name, this is not a method to solve a single problem, but a methodology that can be applied in several contexts. Our goal is to explain the rationale and show the mathematical derivation, together with some practical examples. In particular, we are going to discuss the following topics:

- **Maximum Likelihood Estimation** (**MLE**) and **Maximum A Posteriori** (**MAP**) learning approaches
- The EM algorithm with a simple application for the estimation of unknown parameters
- The Gaussian mixture algorithm, which is one the most famous EM applications
- Factor analysis
- **Principal Component Analysis** (**PCA**)
- **Independent Component Analysis** (**ICA**)
- A brief explanation of the **Hidden Markov Models** (**HMMs**) forward-backward algorithm considering the EM steps

MLE and MAP learning

Let's suppose we have a data generating process $p_{data,}$ used to draw a dataset X:

$$X = \{\bar{x_1}, \bar{x_2}, \ldots, \bar{x_N}\} \ where \ \bar{x_i} \in \mathbb{R}^k$$

In many statistical learning tasks, our goal is to find the optimal parameter set θ according to a maximization criterion. The most common approach is based on the likelihood and is called MLE. In this case, the optimal set θ is found as follows:

$$\theta_{opt} = argmax_\theta \ L(\theta; X) = argmax_\theta \ p(X|\theta)$$

This approach has the advantage of being unbiased by wrong preconditions, but, at the same time, it excludes any possibility of incorporating prior knowledge into the model. It simply looks for the best θ in a wider subspace, so that $p(X|\theta)$ is maximized. Even if this approach is almost unbiased, there's a higher probability of finding a sub-optimal solution that can also be quite different from a reasonable (even if not sure) prior. After all, several models are too complex to allow us to define a suitable prior probability (think, for example, of reinforcement learning strategies where there's a huge number of complex states). Therefore, MLE offers the most reliable solution. Moreover, it's possible to prove that the MLE of a parameter θ converges in probability to the real value:

$$\forall \epsilon > 0 \quad P\left(\left|\tilde{\theta}_k - \theta\right| < \epsilon\right) \rightarrow 1 \;\; when \;\; k \rightarrow \infty$$

On the other hand, if we consider Bayes' theorem, we can derive the following relation:

$$p(\theta|X) = \alpha p(X|\theta)p(\theta)$$

The posterior probability, $p(\theta|X)$, is obtained using both the likelihood and a prior probability, $p(\theta)$, and hence takes into account existing knowledge encoded in $p(\theta)$. The choice to maximize $p(\theta|X)$ is called the MAP approach and it's often a good alternative to MLE when it's possible to formulate trustworthy priors or, as in the case of **Latent Dirichlet Allocation** (**LDA**), where the model is on purpose based on some specific prior assumptions.

Unfortunately, a wrong or incomplete prior distribution can bias the model leading to unacceptable results. For this reason, MLE is often the default choice even when it's possible to formulate reasonable assumptions on the structure of $p(\theta)$. To understand the impact of a prior on an estimation, let's consider to have observed n=1000 binomial distributed (θ corresponds to the parameter p) experiments and k=800 had a successful outcome. The likelihood is as follows:

$$p(X|\theta) = \binom{n}{k}\theta^k(1-\theta)^{n-k}$$

For simplicity, let's compute the log-likelihood:

$$log\, p(X|\theta) = log\, \binom{n}{k} + k\, log\theta + (n-k)\, log(1-\theta)$$

If we compute the derivative with respect to θ and set it equal to zero, we get the following:

$$\frac{\partial}{\partial\theta} log\, p(X|\theta) = \frac{k}{\theta} - \frac{n-k}{1-\theta} = 0 \Rightarrow \theta = \frac{\frac{1}{n-k}}{\frac{1}{k} + \frac{1}{n-k}} = \frac{k}{n}$$

So the MLE for θ is 0.8, which is coherent with the observations (we can say that after observing 1000 experiments with 800 successful outcomes, *p(X|Success)=0.8*). If we have only the data *X*, we could say that a success is more likely than a failure because 800 out of 1000 experiments are positive.

However, after this simple exercise, an expert can tell us that, considering the largest possible population, the marginal probability *p(Success)=0.001* (Bernoulli distributed with *p(Failure) = 1 - P(success)*) and our sample is not representative. If we trust the expert, we need to compute the posterior probability using Bayes' theorem:

$$p(Success|X) = \frac{p(X|Success)p(Success)}{p(X|Success)p(Success) + p(X|Failure)(1-p(Success))} =$$
$$= \frac{0.8 \cdot 0.001}{(0.8 \cdot 0.001) + (0.2 \cdot 0.999)} = \frac{0.0008}{0.0008 + 0.1998} \approx 0.004$$

Surprisingly, the posterior probability is very close to zero and we should reject our initial hypothesis! At this point, there are two options: if we want to build a model based only on our data, the MLE is the only reasonable choice, because, considering the posterior, we need to accept we have a very poor dataset (this is probably a bias when drawing the samples from the data generating process p_{data}).

On the other hand, if we really trust the expert, we have a few options for managing the problem:

- Checking the sampling process in order to assess its quality (we can discover that a better sampling leads to a very lower *k* value)
- Increasing the number of samples
- Computing the MAP estimation of θ

I suggest that the reader tries both approaches with simple models, to be able to compare the relative accuracies. In this book, we're always going to adopt the MLE when it's necessary to estimate the parameters of a model with a statistical approach. This choice is based on the assumption that our datasets are correctly sampled from p_{data}. If this is not possible (think about an image classifier that must distinguish between horses, dogs, and cats, built with a dataset where there are pictures of 500 horses, 500 dogs, and 5 cats), we should expand our dataset or use data augmentation techniques to create artificial samples.

EM algorithm

The EM algorithm is a generic framework that can be employed in the optimization of many generative models. It was originally proposed in *Maximum likelihood from incomplete data via the em algorithm*, *Dempster A. P.*, *Laird N. M.*, *Rubin D. B.*, *Journal of the Royal Statistical Society*, *B*, *39(1):1–38*, *11/1977*, where the authors also proved its convergence at different levels of genericity.

For our purposes, we are going to consider a dataset, X, and a set of latent variables, Z, that we cannot observe. They can be part of the original model or introduced artificially as a trick to simplify the problem. A generative model parameterized with the vector θ has a log-likelihood equal to the following:

$$L(\bar{\theta}|X, Z) = log\ P(X, Z|\bar{\theta})$$

Of course, a large log-likelihood implies that the model is able to generate the original distribution with a small error. Therefore, our goal is to find the optimal set of parameters θ that maximizes the marginalized log-likelihood (we need to sum—or integrate out for continuous variables—the latent variables out because we cannot observe them):

$$\bar{\theta}_{opt} = argmax_{\bar{\theta}}\ log \sum_{z} L(\bar{\theta}|X, z) = argmax_{\bar{\theta}}\ log \sum_{z} P(X, z|\bar{\theta})$$

Theoretically, this operation is correct, but, unfortunately, it's almost always impracticable because of its complexity (in particular, the logarithm of a sum is often very problematic to manage). However, the presence of the latent variables can help us in finding a good proxy that is easy to compute and whose maximization corresponds to the maximization of the original log-likelihood. Let's start by rewriting the expression of the likelihood using the chain rule:

$$log \sum_z P(X, z|\bar{\theta}) = log \sum_z P(X|z, \bar{\theta}) P(z|\bar{\theta})$$

If we consider an iterative process, our goal is to find a procedure that satisfies the following condition:

$$L(\bar{\theta}_{opt}|X, Z) > L(\bar{\theta}_t|X, Z) > L(\bar{\theta}_{t-1}|X, Z) > \ldots > L(\bar{\theta}_0|X, Z)$$

We can start by considering a generic step:

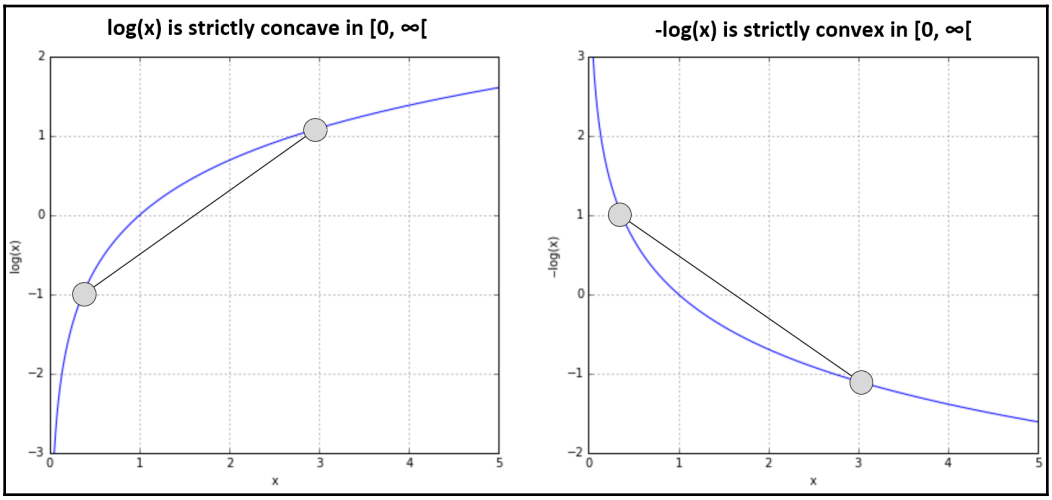
$$L(\bar{\theta}|X) - L(\bar{\theta}_t|X) = log \sum_z P(X|z, \bar{\theta}) P(z|\bar{\theta}) - log\ P(X|\bar{\theta}_t)$$

The first problem to solve is the logarithm of the sum. Fortunately, we can employ the *Jensen's inequality*, which allows us to move the logarithm inside the summation. Let's first define the concept of a *convex function*: a function, *f(x),* defined on a convex set, *D,* is said to be convex if the following applies:

$$f(\lambda x_1 + (1-\lambda)x_2) \leqslant \lambda f(x_1) + (1-\lambda) f(x_2) \quad \forall\, x_1, x_2 \in D \text{ and } \lambda \in [0, 1]$$

If the inequality is strict, the function is said to be *strictly convex*. Intuitively, and considering a function of a single variable *f(x),* the previous definition states that the function is never above the segment that connects two points (x_1, $f(x_1)$) and (x_2, $f(x_2)$). In the case of strict convexity, *f(x)* is always below the segment. Inverting these definitions, we obtain the conditions for a function to be *concave* or *strictly concave*.

If a function *f(x)* is concave in *D*, the function *-f(x)* is convex in *D*; therefore, as *log(x)* is concave in *[0, ∞)* (or with an equivalent notation in *[0, ∞[*), *-log(x)* is convex in *[0, ∞)*, as shown in the following diagram:



The *Jensen's inequality* (the proof is omitted but further details can be found in *Jensen's Operator Inequality*, *Hansen F.*, *Pedersen G. K.*, arXiv:math/0204049 [math.OA] states that if *f(x)* is a convex function defined on a convex set *D*, if we select n points $x_1, x_2, ..., x_n \in D$ and *n* constants $\lambda_1, \lambda_2, ..., \lambda_n \geq 0$ satisfying the condition $\lambda_1 + \lambda_2 + ... + \lambda_n = 1$, then the following applies:

$$f\left(\sum_i \lambda_i x_i\right) \leqslant \sum_i \lambda_i f(x_i)$$

Therefore, considering that *-log(x)* is convex, the *Jensen's inequality* for *log(x)* becomes as follows:

$$log\left(\sum_i \lambda_i x_i\right) \geqslant \sum_i \lambda_i log(x_i)$$

Hence, the generic iterative step can be rewritten, as follows:

$$\Delta L = L(\bar{\theta}|X) - L(\bar{\theta}_t|X) = log \sum_z P(X|z,\bar{\theta})P(z|\bar{\theta}) - log\ P(X|\bar{\theta}_t) =$$

$$= log \sum_z P(z|X,\bar{\theta}_t) \frac{P(X|z,\bar{\theta})P(z|\bar{\theta})}{P(z|X,\bar{\theta}_t)} - log\ P(X|\bar{\theta}_t)$$

Applying the Jensen's inequality, we obtain the following:

$$\Delta L \geqslant \sum_z P(z|X,\bar{\theta}_t)\ log \frac{P(X|z,\bar{\theta})P(z|\bar{\theta})}{P(z|X,\bar{\theta}_t)} - log\ P(X|\bar{\theta}_t) =$$

$$= \sum_z P(z|X,\bar{\theta}_t)\ log \frac{P(X|z,\bar{\theta})P(z|\bar{\theta})}{P(z|X,\bar{\theta}_t)P(X|\bar{\theta}_t)}$$

All the conditions are met, because the terms $P(z_i|X, \theta_t)$ are, by definition, bounded between [0, 1] and the sum over all z must always be equal to 1 (laws of probability). The previous expression implies that the following is true:

$$L(\bar{\theta}|X) \geqslant L(\bar{\theta}_t|X) + \sum_z P(z|X,\bar{\theta}_t)\ log \frac{P(X|z,\bar{\theta})P(z|\bar{\theta})}{P(z|X,\bar{\theta}_t)P(X|\bar{\theta}_t)}$$

Therefore, if we maximize the right side of the inequality, we also maximize the log-likelihood. However, the problem can be further simplified, considering that we are optimizing only the parameter vector θ and we can remove all the terms that don't depend on it. Hence, we can define a *Q function* (there are no relationships with the Q-Learning that we're going to discuss in `Chapter 14`, *Introduction to Reinforcement Learning*) whose expression is as follows:

$$Q(\bar{\theta}|\bar{\theta}_t) = \sum_z P(z|X,\bar{\theta}_t)\ log\ P(X|z,\bar{\theta})P(z|\bar{\theta}) = \sum_z P(z|X,\bar{\theta}_t)\ log\ P(X,z|\bar{\theta}) = E_{Z|X,\bar{\theta}_t}[log\ P(X,z|\bar{\theta})]$$

Q is the expected value of the log-likelihood considering the complete data $Y = (X, Z)$ and the current iteration parameter *set* θ_t. At each iteration, Q is computed considering the current estimation θ_t and it's maximized considering the variable θ. It's now clearer why the latent variables can be often artificially introduced: they allow us to apply the *Jensen's inequality* and transform the original expression into an expected value that is easy to evaluate and optimize.

At this point, we can formalize the EM algorithm:

1. Set a threshold *Thr* (for example, *Thr* = 0.01)
2. Set a random parameter vector θ_0.
3. While $|L(\theta_t|X, Z) - L(\theta_{t-1}|X, Z)| > Thr$:
    - **E-Step**: Compute the $Q(\theta|\theta_t)$. In general, this step consists in computing the conditional probability $p(z|X, \theta_t)$ or some of its moments (sometimes, the sufficient statistics are limited to mean and covariance) using the current parameter estimation θ_t.
    - **M-Step**: Find $\theta_{t+1} = argmax_\theta\ Q(\theta|\theta_t)$. The new parameter estimation is computed to maximize the Q function.

The procedure ends when the log-likelihood stops increasing or after a fixed number of iterations.

An example of parameter estimation

In this example, we see how it's possible to apply the EM algorithm for the estimation of unknown parameters (inspired by an example discussed in the original paper *Maximum likelihood from incomplete data via the em algorithm, Dempster A. P., Laird N. M., Rubin D. B., Journal of the Royal Statistical Society, B, 39(1):1–38, 11/1977*).

Let's consider a sequence of n independent experiments modeled with a multinomial distribution with three possible outcomes x_1, x_2, x_3 and corresponding probabilities p_1, p_2 and p_3. The probability mass function is as follows:

$$f(x_1, x_2, x_3; p_1, p_2, p_3) = \frac{n!}{\prod_{i=1}^{3} x_i!} \prod_{i=1}^{3} p_i^{x_i}$$

Let's suppose that we can observe $z_1 = x_1 + x_2$ and x_3, but we don't have any direct access to the single values x_1 and x_2. Therefore, x_1 and x_2 are latent variables, while z_1 and x_3 are observed ones. The probability vector p is parameterized in the following way:

$$\bar{p} = (p_1 \quad p_2 \quad p_3) = \left(\frac{\theta}{6} \quad 1 - \frac{\theta}{4} \quad \frac{\theta}{12}\right)$$

Our goal is to find the MLE for θ given n, z_1, and x_3. Let's start computing the log-likelihood:

$$L(\theta|x_1,x_2,x_3,z_1) = log\frac{n!}{\prod_{i=1}^{3} x_i!}\prod_{i=1}^{3} p_i^{x_i} = c + \sum_{i=1}^{3} x_i \; log\, p_i =$$
$$= c + x_1 \; log\frac{\theta}{6} + x_2 \; log\left(1-\frac{\theta}{4}\right) + x_3 \; log\,\frac{\theta}{12}$$

We can derive the expression for the corresponding Q function, exploiting the linearity of the expected value operator $E[\bullet]$:

$$Q(\theta|\theta_t) = E\left[x_1|z_1,\bar{p}^{(t)}\right] log\,\frac{\theta}{6} + E\left[x_2|z_1,\bar{p}^{(t)}\right] log\left(1-\frac{\theta}{4}\right) + x_3 \; log\,\frac{\theta}{12}$$

The variables x_1 and x_2, given z_1, are binomially distributed and can be expressed as a function of θ_t (we need to recompute them at each iteration). Hence, the expected value of $x_1^{(t+1)}$ becomes as follows:

$$E\left[x_1|z_1,\bar{p}^{(t)}\right] = z_1\frac{p_1^{(t)}}{p_1^{(t)}+p_2^{(t)}} = z_1\frac{\frac{\theta_t}{6}}{\frac{\theta}{6}+1-\frac{\theta_t}{4}} = z_1\frac{2\theta_t}{12-\theta_t}$$

While the expected value of $x_2^{(t+1)}$ is as follows:

$$E\left[x_2|z_1,\bar{p}^{(t)}\right] = z_1\frac{p_2^{(t)}}{p_1^{(t)}+p_2^{(t)}} = z_1\frac{1-\frac{\theta_t}{4}}{\frac{\theta}{6}+1-\frac{\theta_t}{4}} = z_1\frac{3(4-\theta_t)}{12-\theta_t}$$

If we apply these expressions in $Q(\theta|\theta_t)$ and compute the derivative with respect to θ, we get the following:

$$\frac{\partial Q}{\partial\theta} = 0 \Rightarrow \frac{E\left[x_1|z_1,\bar{p}^{(t)}\right]+x_3}{\theta} + \frac{E\left[x_2|z_1,\bar{p}^{(t)}\right]}{\theta-4} = 0$$

Therefore, solving for θ, we get the following:

$$\theta = \frac{4\left(E\left[x_1^{(t+1)}|z_1, \bar{p}^{(t)}\right] + x_3)\right)}{z_1 + x_3}$$

At this point, we can derive the iterative expression for θ:

$$\theta = \frac{4\left(z_1 \frac{2\theta_t}{12-\theta_t} + x_3)\right)}{z_1 + x_3} = \frac{8z_1\theta_t + 4x_3(12-\theta_t)}{(z_1 + x_3)(12-\theta_t)}$$

Let's compute the value of θ for $z_1 = 50$ and $x_3 = 10$:

```
def theta(theta_prev, z1=50.0, x3=10.0):
    num = (8.0 * z1 * theta_prev) + (4.0 * x3 * (12.0 - theta_prev))
    den = (z1 + x3) * (12.0 - theta_prev)
    return num / den

theta_v = 0.01

for i in range(1000):
    theta_v = theta(theta_v)

print(theta_v)
1.999999999999999

p = [theta_v/6.0, (1-(theta_v/4.0)), theta_v/12.0]

print(p)
[0.33333333333333315, 0.5000000000000002, 0.16666666666666657]
```

In this example, we have parameterized all probabilities and, considering that $z_1 = x_1 + x_2$, we have one degree of freedom for the choice of θ. The reader can repeat the example by setting the value of one of p_1 or p_2 and leaving the other probabilities as functions of θ. The computation is almost identical but in this case, there are no degrees of freedom.

Gaussian mixture

In `Chapter 2`, *Introduction to Semi-Supervised Learning*, we discussed the generative Gaussian mixture model in the context of semi-supervised learning. In this paragraph, we're going to apply the EM algorithm to derive the formulas for the parameter updates.

Let's start considering a dataset, *X*, drawn from a data generating process, p_{data}:

$$X = \{\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_N\} \, where \, \bar{x}_i \in \mathbb{R}^m$$

We assume that the whole distribution is generated by the sum of *k* Gaussian distributions so that the probability of each sample can be expressed as follows:

$$p(\bar{x}_i) = \sum_{j=1}^{k} P(N = j) \, N(\bar{x}_i | \bar{\mu_j}, \Sigma_j) = \sum_{j=1}^{k} w_j N(\bar{x}_i | \bar{\mu_j}, \Sigma_j)$$

In the previous expression, the term $w_j = P(N=j)$ is the relative weight of the j^{th} Gaussian, while μ_j and Σ_j are the mean and the covariance matrix. For consistency with the laws of probability, we also need to impose the following:

$$\sum_j w_j = 1$$

Unfortunately, if we try to solve the problem directly, we need to manage the logarithm of a sum and the procedure becomes very complex. However, we have learned that it's possible to use latent variables as helpers, whenever this trick can simplify the solution.

Let's consider a single parameter set $\theta=(w_j, \mu_j, \Sigma_j)$ and a latent indicator matrix Z where each element z_{ij} is equal to 1 if the point x_i has been generated by the j^{th} Gaussian, and 0 otherwise. Therefore, each z_{ij} is Bernoulli distributed with parameters equal to $p(j | x_i, \theta_t)$.

The joint log-likelihood can hence be expressed using the exponential-indicator notation, as follows:

$$L(\theta; X, Z) = log \prod_i \prod_j p(\bar{x}_i, j|\theta)^{z_{ij}} = \sum_i \sum_j z_{ij} log \, p(\bar{x}_i, j|\theta)$$

The index, *i*, is referred to the samples, while *j* refers to the Gaussian distributions. If we apply the chain rule and the properties of a logarithm, the expression becomes as follows:

$$L(\theta; X, Z) = \sum_i \sum_j z_{ij} log \, p(\bar{x}_i, j|\theta) = \sum_i \sum_j z_{ij} log \, p(\bar{x}_i | j, \theta) + z_{ij} log \, p(j|\theta)$$

The first term represents the probability of x_i under the j^{th} Gaussian, while the second one is the relative weight of the j^{th} Gaussian. We can now compute the $Q(\theta;\theta_t)$ function using the joint log-likelihood:

$$Q(\theta|\theta_t) = E_{Z|X,\theta_t}[L(\theta; X, Z)] = E_{Z|X,\theta_t}\left[\sum_i \sum_j z_{ij} log\, p(\bar{x}_i|j,\theta) + z_{ij} log\, p(j|\theta)\right]$$

Exploiting the linearity of $E[\bullet]$, the previous expression becomes as follows:

$$Q(\theta|\theta_t) = \sum_i \sum_j E_{Z|X,\theta_t}[z_{ij}]\, log\, p(\bar{x}_i|j,\theta) + E_{Z|X,\theta_t}[z_{ij}]\, log\, p(j|\theta) =$$
$$= \sum_i \sum_j p(j|\bar{x}_i,\theta_t)\, log\, p(\bar{x}_i|j,\theta) + p(j|\bar{x}_i,\theta_t)\, log\, p(j|\theta)$$

The term $p(j \mid x_i, \theta_t)$ corresponds to the expected value of z_{ij} considering the complete data, and expresses the probability of the j^{th} Gaussian given the sample x_i. It can be simplified considering Bayes' theorem:

$$p(j|\bar{x}_i,\theta_t) = \alpha\, p(\bar{x}_i|j,\theta_t)\, p(j,\theta_t)$$

The first term is the probability of x_i under the j^{th} Gaussian with parameters θ_t, while the second one is the weight of the j^{th} Gaussian considering the same parameter set θ_t. In order to derive the iterative expressions for the parameters, it's useful to write the complete formula for the logarithm of a multivariate Gaussian distribution:

$$log\, p(\bar{x}_i|j,\theta) = log\, \frac{1}{\sqrt{2\pi\ \det \Sigma_j}} e^{-\frac{1}{2}(\bar{x}_i-\bar{\mu}_j)^T \Sigma^{-1}(\bar{x}_i-\bar{\mu}_j)} =$$
$$= -\frac{m}{2} log\, 2\pi\ -\ \frac{1}{2} log\, det\, \Sigma_j - \frac{1}{2}(\bar{x}_i - \bar{\mu}_j)^T \Sigma^{-1}(\bar{x}_i - \bar{\mu}_j)$$

To simplify this expression, we use the trace trick. In fact, as $(x_i - \mu_j)^T \Sigma^{-1} (x_i - \mu_j)$ is a scalar, we can exploit the properties $tr(AB) = tr(BA)$ and $tr(c) = c$ where A and B are matrices and $c \in \Re$:

$$log\, p(\bar{x}_i|j,\theta) = -\frac{m}{2} log\, 2\pi\ -\ \frac{1}{2} log\, det\, \Sigma_j - \frac{1}{2} tr\left(\Sigma_j^{-1}(\bar{x}_i - \bar{\mu}_j)(\bar{x}_i - \bar{\mu}_j)^T\right)$$

Let's start considering the estimation of the mean (only the first term of $Q(\theta;\theta_t)$ depends on mean and covariance):

$$\frac{\partial Q}{\partial \mu_j} = -\frac{1}{2}\sum_i p\left(j|\bar{x}_i,\theta_t\right)\frac{\partial}{\partial \mu_j} tr\left(\Sigma_j^{-1}\left(\bar{x}_i-\bar{\mu_j}\right)\left(\bar{x}_i-\bar{\mu_j}\right)^T\right) =$$

$$= \sum_i p\left(j|\bar{x}_i,\theta_t\right) tr\left(\Sigma_j^{-1}\left(\bar{x}_i-\bar{\mu_j}\right)\right)$$

Setting the derivative equal to zero, we get the following:

$$\mu_j = \frac{\sum_i p\left(j|\bar{x}_i,\theta_t\right)\bar{x}_i}{\sum_i p\left(j|\bar{x}_i,\theta_t\right)}$$

In the same way, we obtain the expression of the covariance matrix:

$$\Sigma_j = \frac{\sum_i p\left(j|\bar{x}_i,\theta_t\right)\left[\left(\bar{x}_i-\bar{\mu_j}\right)\left(\bar{x}_i-\bar{\mu_j}\right)^T\right]}{\sum_i p\left(j|\bar{x}_i,\theta_t\right)}$$

To obtain the iterative expressions for the weights, the procedure is a little bit more complex, because we need to use the Lagrange multipliers (further information can be found in `http://www.slimy.com/~steuard/teaching/tutorials/Lagrange.html`). Considering that the sum of the weights must always be equal to 1, it's possible to write the following equation:

$$P = Q - \lambda\left(\sum_j w_j - 1\right) \Rightarrow \frac{\partial P}{\partial w_j} = \frac{\partial Q}{\partial w_j} - \lambda \;\; and \;\; \frac{\partial P}{\partial \lambda} = \sum_j w_j - 1$$

Setting both derivatives equal to zero, from the first one, considering that $wj = p(j|\theta)$, we get the following:

$$\frac{\partial P}{\partial w_j} = \frac{\partial Q}{\partial w_j} - \lambda = 0 \;\Rightarrow\; \frac{\partial}{\partial w_j}\sum_i p\left(j|\bar{x}_i,\theta_t\right) log\, p(j|\theta) = \lambda \;\Rightarrow\; \frac{\sum_i p\left(j|\bar{x}_i,\theta_t\right)}{w_j} = \lambda$$

$$w_j = \frac{\sum_i p\left(j|\bar{x}_i,\theta_t\right)}{\lambda}$$

While from the second derivative, we obtain the following:

$$\frac{\partial P}{\partial \lambda} = \sum_j w_j - 1 \;\Rightarrow\; \frac{\partial P}{\partial \lambda} = \frac{\sum_i \sum_j p(j|\bar{x}_i, \theta_t)}{\lambda} - 1 \;\Rightarrow\; \lambda = N$$

The last step derives from the fundamental condition:

$$\sum_j p(j|\bar{x}_i, \theta_t) = 1$$

Therefore, the final expression of the weights is as follows:

$$w_j = \frac{\sum_i p(j|\bar{x}_i, \theta_t)}{N}$$

At this point, we can formalize the Gaussian mixture algorithm:

- Set random initial values for $w_j^{(0)}$, ${\theta^{(0)}}_j$ and ${\Sigma^{(0)}}_j$
- **E-Step**: Compute $p(j|x_i, \theta_t)$ using Bayes' theorem: $p(j|x_i, \theta_t) = \alpha\, {w^{(t)}}_j\, p(x_i|j, \theta_t)$
- **M-Step**: Compute $w_j^{(t+1)}$, ${\theta^{(t+1)}}_j$ and ${\Sigma^{(t+1)}}_j$ using the formulas provided previously

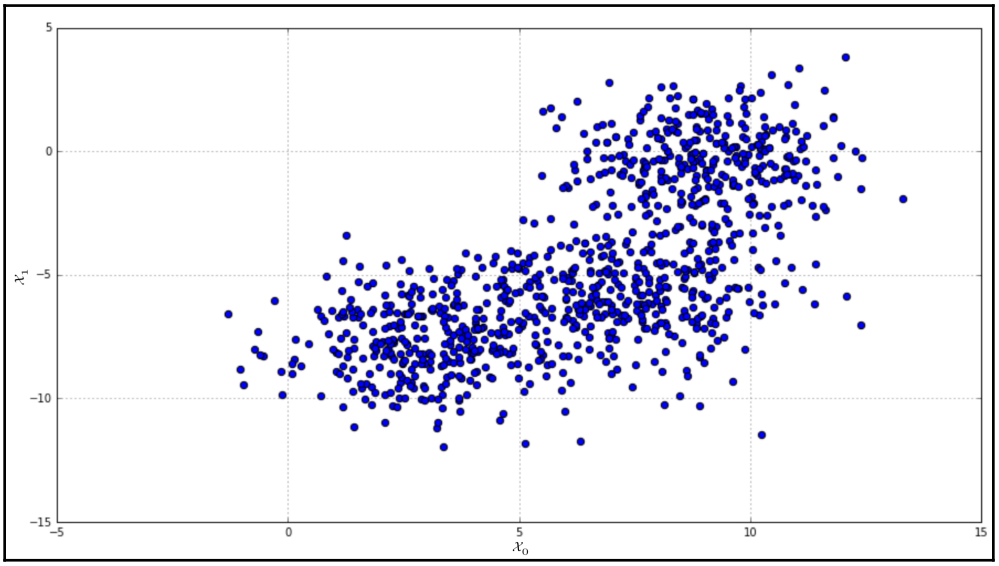
The process must be iterated until the parameters become stable. In general, the best practice is using both a threshold and a maximum number of iterations.

An example of Gaussian Mixtures using Scikit-Learn

We can now implement the Gaussian mixture algorithm using the Scikit-Learn implementation. The direct approach has already been shown in `Chapter 2`, *Introduction to Semi-Supervised Learning*. The dataset is generated to have three cluster centers and a moderate overlap due to a standard deviation equal to 1.5:

```
from sklearn.datasets import make_blobs

nb_samples = 1000
X, Y = make_blobs(n_samples=nb_samples, n_features=2, centers=3,
cluster_std=1.5, random_state=1000)
```

The corresponding plot is shown in the following diagram:



The Scikit-Learn implementation is based on the `GaussianMixture` class , which accepts as parameters the number of Gaussians (`n_components`), the type of covariance (`covariance_type`), which can be `full` (the default value), if all components have their own matrix, `tied` if the matrix is shared, `diag` if all components have their own diagonal matrix (this condition imposes an uncorrelation among the features), and `spherical` when each Gaussian is symmetric in every direction. The other parameters allow setting regularization and initialization factors (for further information, the reader can directly check the documentation). Our implementation is based on full covariance:

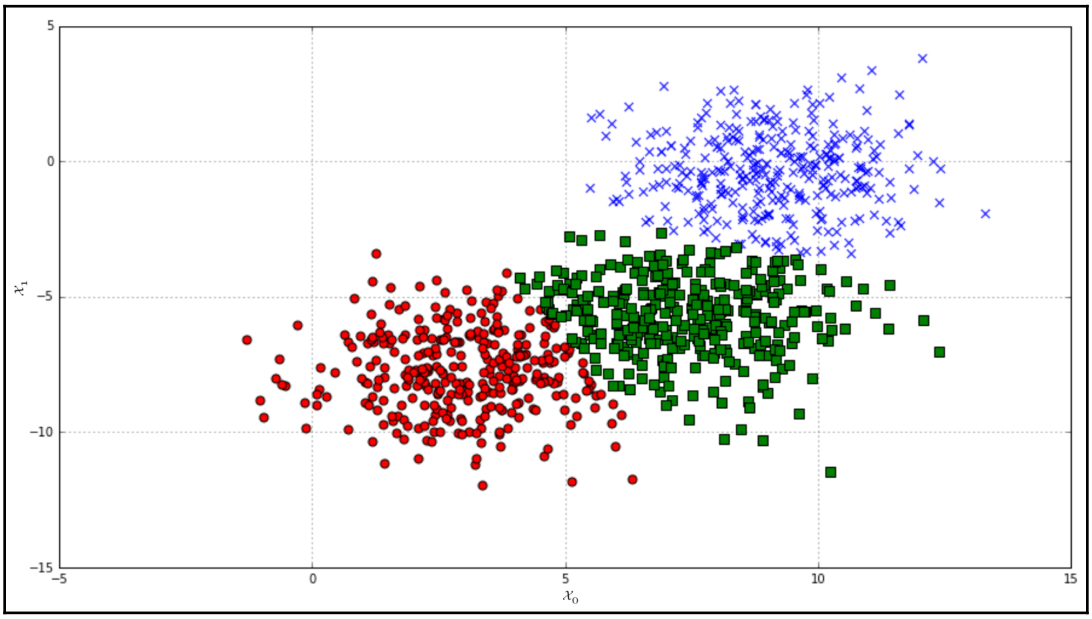
```
from sklearn.mixture import GaussianMixture

gm = GaussianMixture(n_components=3)
gm.fit(X)
```

After fitting the model, it's possible to access to the learned parameters through the instance variables `weights_`, `means_`, and `covariances_`:

```
print(gm.weights_)

[ 0.32904743  0.33027731  0.34067526]

print(gm.means_)
```

```
[[ 3.03902183 -7.69186648]
 [ 9.04414279 -0.37455175]
 [ 7.37103878 -5.77496152]]

print(gm.covariances_)

[[[ 2.34943036  0.08492009]
  [ 0.08492009  2.36467211]]

 [[ 2.10999633  0.02602279]
  [ 0.02602279  2.21533635]]

 [[ 2.71755196 -0.0100434 ]
  [-0.0100434   2.39941067]]]
```

Considering the covariance matrices, we can already understand that the features are very uncorrelated and the Gaussians are almost spherical. The final plot can be obtained by assigning each point to the corresponding cluster (Gaussian distribution) through the `Yp = gm.transform(X)` command:



Labeled dataset obtained through the application of a Gaussian mixture with three components

The reader should have noticed a strong analogy between Gaussian mixture and k-means (which we're going to discuss in `Chapter 7`, *Clustering Algorithms*). In particular, we can state that K-means is a particular case of spherical Gaussian mixture with a covariance $\Sigma \rightarrow 0$. This condition transforms the approach from a soft clustering, where each sample belongs to all clusters with a precise probability distribution, into a hard clustering, where the assignment is done by considering the shortest distance between sample and centroid (or mean). For this reason, in some books, the Gaussian mixture algorithm is also called soft K-means. A conceptually similar approach that we are going to present is Fuzzy K-means, which is based on assignments characterized by membership functions, which are analogous to probability distributions.

Factor analysis

Let's suppose we have a Gaussian data generating process, $p_{data} \sim N(0, \Sigma)$, and M n-dimensional zero-centered samples drawn from it:

$$X = \{\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_M\} \;\; where \;\; \bar{x}_i \in \mathbb{R}^n$$

If p_{data} has a mean $\mu \neq 0$, it's also possible to use this model, but it's necessary to account for this non-null value with slight changes in some formulas. As the zero-centering normally has no drawbacks, it's easier to remove the mean to simplify the model.

One of the most common problems in unsupervised learning is finding a lower dimensional distribution p_{lower} such that the Kullback-Leibler divergence with p_{data} is minimized. When performing a **factor analysis** (**FA**), following the original proposal published in *EM algorithms for ML factor analysis*, *Rubin D.*, *Thayer D.*, *Psychometrika*, 47/1982, Issue 1, and *The EM algorithm for Mixtures of Factor Analyzers*, *Ghahramani Z.*, *Hinton G. E.*, CRC-TG-96-1, 05/1996, we start from the assumption to model the generic sample x as a linear combination of Gaussian latent variables, z, (whose dimension p is normally $p < n$) plus an additive and decorrelated Gaussian noise term, ν:

$$\bar{x} = A\bar{z} + \bar{\nu} \;\; where \;\; \bar{z} \sim N(0, I) \;\; and \;\; \bar{\nu} \sim N(0, \Omega) \;\; with \;\; \Omega = diag(\omega_0^2, \omega_1^2, \ldots, \omega_n^2)$$

The matrix, A, is called a *factor loading matrix* because it determines the contribution of each latent variable (factor) to the reconstruction of x. Factors and input data are assumed to be statistically independent. Instead, considering the last term, if $\omega_0^2 \neq \omega_1^2 \neq ... \neq \omega_n^2$ the noise is called *heteroscedastic*, while it's defined *homoscedastic* if the variances are equal $\omega_0^2 = \omega_1^2 = ... = \omega_n^2 = \omega^2$. To understand the difference between these two kinds of noise, think about a signal x which is the sum of two identical voices, recorded in different places (for example, an airport and a wood). In this case, we can suppose to also have different noise variances (the first one should be higher than the second considering the number of different noise sources). If instead both voices are recorded in a soundproofed room or even in the same airport, homoscedastic noise is surely more likely (we're not considering the power, but the difference between the variances).

One of the most important strengths of FA in respect to other methods (such as PCA) is its intrinsic robustness to heteroscedastic noise. In fact, including the noise term in the model (with only the constraint to be decorrelated) allows partial denoising filtering based on the single components, while one of the preconditions for the PCA is to impose only homoscedastic noise (which, in many cases, is very similar to the total absence of noise). Considering the previous example, we could make the assumption to have the first variance be $\omega_0^2 = k\,\omega_1^2$ with $k > 1$. In this way, the model will be able to understand that a high variance in the first component should be considered (with a higher probability) as the product of the noise and not an intrinsic property of the component.

Let's now analyze the linear relation:

$$\bar{x} = A\bar{z} + \bar{\nu}$$

Considering the properties of Gaussian distributions, we know that $x \sim N(\mu, \Sigma)$ and it's easy to determine either the mean or the covariance matrix:

$$\mu = E[X] = AE[Z] + E[\epsilon] = 0$$
$$\Sigma = E[X^T X] = AE[Z^T Z]A^T + E[\nu^T \nu] = AA^T + \Omega$$

Therefore, in order to solve the problem, we need to find the best $\theta=(A, \Omega)$ so that $AA^T + \Omega \approx \Sigma$ (with a zero-centered dataset, the estimation is limited to the input covariance matrix Σ).The ability to cope with noisy variables should be clearer now. If $AA^T + \Omega$ is exactly equal to Σ and the estimation of Ω is correct, the algorithm will optimize the factor loading matrix A, excluding the interference produced by the noise term; therefore, the components will be approximately denoised.

In order to adopt the EM algorithm, we need to determine the joint probability $p(X, z; \theta) = p(X|z; \theta)p(z|\theta)$. The first term on the right side can be easily determined, considering that $x - Az \sim N(0, \Omega)$; therefore, we get the following:

$$p(X, \bar{z}; \theta) = \prod_{i=1}^{M} \left(\frac{1}{\sqrt{(2\pi)^n det(\Omega)}} e^{-\frac{1}{2}(\bar{x}_i - A\bar{z})^T \Omega^{-1} (\bar{x}_i - A\bar{z})} \right) \left(\frac{1}{\sqrt{(2\pi)^p}} e^{-\frac{1}{2}\bar{z}^T \bar{z}} \right) =$$
$$= (2\pi)^{-\frac{Mn+p}{2}} det(\Omega)^{-\frac{M}{2}} \prod_{i=1}^{M} e^{-\frac{1}{2}\left[(\bar{x}_i - A\bar{z})^T \Omega^{-1} (\bar{x}_i - A\bar{z}) + \bar{z}^T \bar{z}\right]}$$

We can now determine the $Q(\theta;\theta_i)$ function, discarding the constant term $(2\pi)^k$ and term z^Tz, which don't depend on θ (in this particular case, as we're going to see, we don't need to compute the probability $p(z|X;\theta)$ because it's enough to obtain sufficient statistics for expected value and second moment). Moreover, it's useful to expand the multiplication in the exponential:

$$Q(\theta; \theta_t) = E_{Z|X;\theta} \left[log\, p(X|\bar{z}; \theta) \right] =$$
$$E_{Z|X;\theta} \left[-\frac{M}{2} log\, det(\Omega) - \frac{1}{2} \sum_{i=1}^{M} \left(\bar{x}_i^T \Omega^{-1} \bar{x}_i - 2\bar{x}_i^T \Omega^{-1} A\bar{z} + \bar{z}^T A^T \Omega^{-1} A\bar{z} \right) \right]$$

Using the trace trick with the last term (which is a scalar), we can rewrite it as follows:

$$\bar{z}^T A^T \Omega^{-1} A\bar{z} = tr(\bar{z}^T A^T \Omega^{-1} A\bar{z}) = tr(A^T \Omega^{-1} A\bar{z}\bar{z}^T)$$

Exploiting the linearity of $E[\bullet]$, we obtain the following:

$$Q(\theta; \theta_t) = -\frac{M}{2} log\, det(\Omega) - \frac{1}{2} \sum_{i=1}^{M} \left(\bar{x}_i^T \Omega^{-1} \bar{x}_i - 2\bar{x}_i^T \Omega^{-1} A E_{\bar{z}|\bar{x}_i;\theta}[\bar{z}|\bar{x}_i] + A^T \Omega^{-1} A E_{\bar{z}|\bar{x}_i;\theta}[\bar{z}\bar{z}^T|\bar{x}_i] \right)$$

This expression is similar to what we have seen in the Gaussian mixture model, but in this case, we need to compute the conditional expectation and the conditional second moment of z. Unfortunately, we cannot do this directly, but it's possible to compute them exploiting the joint normality of x and z. In particular, using a classic theorem, we can partition the full joint probability $p(z, x)$, considering the following relations:

$$\bar{v} = \begin{pmatrix} \bar{z} \\ \bar{x} \end{pmatrix} \quad \bar{\mu}^* = \begin{pmatrix} E[\bar{z}] \\ E[\bar{x}] \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix} \quad \Sigma^* = \begin{pmatrix} E[\bar{z}\bar{z}^T] & E[\bar{z}\bar{x}^T] \\ E[\bar{x}\bar{z}^T] & E[\bar{x}\bar{x}^T] \end{pmatrix} = \begin{pmatrix} I & A^T \\ A & AA^T + \Omega \end{pmatrix}$$

The conditional distribution $p(z|x=x_i)$ has a mean equal to the following:

$$E[\bar{z}|\bar{x}=\bar{x}_i] = E[\bar{z}] + E[\bar{z}\bar{x}^T]E[\bar{x}\bar{x}^T]^{-1}(\bar{x}_i - E[\bar{x}]) = A^T(AA^T+\Omega)^{-1}\bar{x}_i$$

The conditional variance is as follows:

$$E\left[(\bar{z} - E[\bar{z}|\bar{x}=\bar{x}_i])^2\right] = E[\bar{z}\bar{z}^T|\bar{x}=\bar{x}_i] - E[\bar{z}|\bar{x}=\bar{x}_i]E[\bar{z}|\bar{x}=\bar{x}_i]^T$$

Therefore, the conditional second moment is equal to the following:

$$E[\bar{z}\bar{z}^T|\bar{x}=\bar{x}_i] = E[\bar{z}\bar{z}^T|\bar{x}=\bar{x}_i] - E[\bar{z}\bar{x}^T|\bar{x}=\bar{x}_i]E[\bar{x}\bar{x}^T|\bar{x}=\bar{x}_i]^{-1}E[\bar{x}\bar{z}^T|\bar{x}=\bar{x}_i] + E[\bar{z}|\bar{x}=\bar{x}_i]E[\bar{z}|\bar{x}=\bar{x}_i]^T = \\ = I - A^T(AA^T+\Omega)^{-1}A + E[\bar{z}|\bar{x}=\bar{x}_i]E[\bar{z}|\bar{x}=\bar{x}_i]^T$$

If we define the auxiliary matrix $K = (AA^T + \Omega)^{-1}$, the previous expressions become as follows:

$$E[\bar{z}|\bar{x}=\bar{x}_i] = A^T K\bar{x}_i$$
$$E[\bar{z}\bar{z}^T|\bar{x}=\bar{x}_i] = I - A^T KA + A^T K\bar{x}_i\bar{x}_i^T K^T A$$

The reader in search of further details about this technique can read *Preview Introduction to Statistical Decision Theory*, *Pratt J., Raiffa H., Schlaifer R., The MIT Press*.

Using the previous expression, it's possible to build the inverse model (sometimes called a *recognition model* because it starts with the effects and rebuilds the causes), which is still Gaussian distributed:

$$\bar{z} = B\bar{x} + \bar{\lambda} \;\; where \;\; p(\bar{z}|\bar{x};\theta) \sim N(E[\bar{z}|\bar{x}], E\left[(\bar{z} - E[\bar{z}|\bar{x}=\bar{x}_i])^2\right]) = N(A^T K\bar{x}, I - A^T KA)$$

We are now able to maximize $Q(\theta;\theta_t)$ with respect to A and Ω, considering $\theta_t=(A_t, \Omega_t)$ and both the conditional expectation and the second moment computed according to the previous estimation $\theta_{t-1}=(A_{t-1}, \Omega_{t-1})$. For this reason, they are not involved in the derivation process. We are adopting the convention that the term subject to maximization is computed at time t, while all the others are obtained through the previous estimations (t - 1):

$$\frac{\partial Q}{\partial A} = -\sum_{i=1}^{M} \Omega_{t-1}^{-1}\bar{x}_i E_{Z|\bar{x}_i;\theta}[\bar{z}|\bar{x}=\bar{x}_i]^T + \sum_{j=1}^{M} \Omega_{t-1}^{-1} A_t E_{Z|\bar{x}_j;\theta}[\bar{z}\bar{z}^T|\bar{x}=\bar{x}_j] = 0$$

The expression for A_t is therefore as follows (Q is the biased input covariance matrix $E[X^TX]$ for a zero-centered dataset):

$$A_t = \left(\sum_{i=1}^{M} \bar{x}_i E_{Z|\bar{x}_i;\theta}[\bar{z}|\bar{x} = \bar{x}_i]^T\right)\left(\sum_{j=1}^{M} E_{Z|\bar{x}_j;\theta}[\bar{z}\bar{z}^T|\bar{x} = \bar{x}_j]\right)^{-1} =$$
$$= (QK_{t-1}^T A_{t-1})(I - A_{t-1}^T K_{t-1} A_{t-1} + A_{t-1}^T K_{t-1} Q K_{t-1}^T A_{t-1})^{-1}$$

In the same way, we can obtain an expression for Ω_t by computing the derivative with respect to Ω^{-1} (this choice simplifies the calculation and doesn't affect the result, because we must set the derivative equal to zero):

$$\frac{\partial Q}{\partial \Omega^{-1}} = \frac{M}{2}\Omega_t - \frac{1}{2}\sum_{i=1}^{M}\left(\bar{x}_i\bar{x}_i^T - 2A_{t-1}E_{Z|\bar{x}_i;\theta}[\bar{z}|\bar{x} = \bar{x}_i] + A_{t-1}E[\bar{z}\bar{z}^T|\bar{x} = \bar{x}_i]A_{t-1}^T\right) =$$
$$\frac{M}{2}\Omega_t - \frac{1}{2}\sum_{i=1}^{M}\left(\bar{x}_i\bar{x}_i^T - A_{t-1}E_{Z|\bar{x}_i;\theta}[\bar{z}|\bar{x} = \bar{x}_i]\bar{x}_i^T\right) = 0$$

The derivative of the first term, which is the determinant of a real diagonal matrix, is obtained using the adjugate matrix $Adj(\Omega)$ and exploiting the properties of the inverse matrix $T^{-1} = det(T)^{-1}Adj(T)$ and the properties $det(T)^{-1} = det(T^{-1})$ and $det(T^T) = det(T)$:

$$\frac{\partial}{\partial \Omega^{-1}} log\ det(\Omega) = -\frac{\partial}{\partial \Omega^{-1}} log\ det(\Omega^{-1}) = det(\Omega)\frac{\partial}{\partial \Omega^{-1}} det(\Omega^{-1}) =$$
$$= det(\Omega)\frac{\partial}{\partial \Omega^{-1}}\Omega^{-1}\left(Adj(\Omega)^{-1}\right)^T = det(\Omega)\left(Adj(\Omega)^T\right)^{-1} = \Omega^T = \Omega$$

The expression for Ω_t (imposing the diagonality constraint) is as follows:

$$\Omega_t = \frac{1}{M} diag\left[\sum_{i=1}^{M}\left(\bar{x}_i\bar{x}_i^T - A_{t-1}E_{Z|\bar{x}_i;\theta}[\bar{z}|\bar{x} = \bar{x}_i]\bar{x}_i^T\right)\right] =$$
$$= diag\left(Q - A_{t-1}A_{t-1}^T K_{t-1} Q\right)$$

Summarizing the steps, we can define the complete FA algorithm:

1. Set random initial values for $A^{(0)}$ and $\Omega^{(0)}$
2. Compute the biased input covariance matrix $Q = E[X^T X]$
3. E-Step: Compute $A^{(t)}$, $\Omega^{(t)}$, and $K^{(t)}$
4. M-Step: Compute $A^{(t+1)}$, $\Omega^{(t+1)}$, and $K^{(t+1)}$ using the previous estimations and the formulas provided previously
5. Compute the matrices B and Ψ for the inverse model

The process must be repeated until $A^{(t)}$, $\Omega^{(t)}$, and $K^{(t)}$ stop modifying their values (using a threshold) together with a constraint on the maximum number of iterations. The factors can be easily obtained using the inverse model $z = Bx + \lambda$.

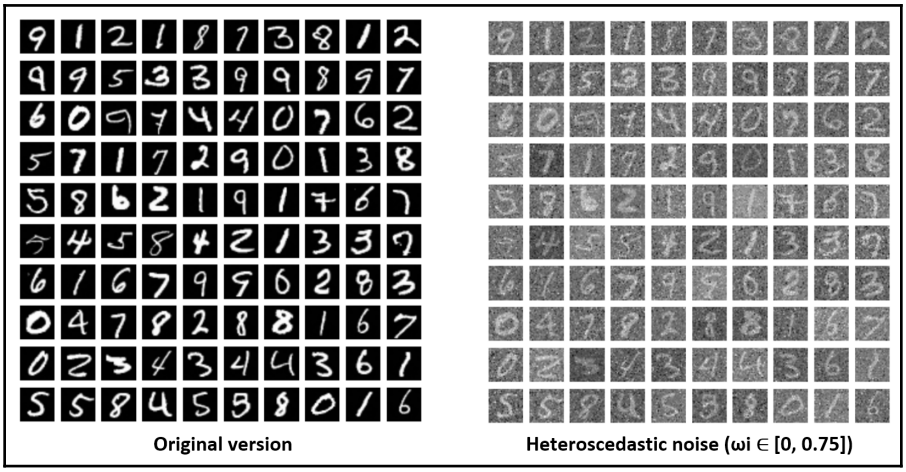
An example of factor analysis with Scikit-Learn

We can now make an example of FA with Scikit-Learn using the MNIST handwritten digits dataset (70,000 28 × 28 grayscale images) in the original version and with added heteroscedastic noise (ω_i randomly selected from [0, 0.75]).

The first step is to load and zero-center the original dataset (I'm using the functions defined in the first chapter, `Chapter 1`, *Machine Learning Model Fundamentals*):

```
import numpy as np

from sklearn.datasets import fetch_mldata

digits = fetch_mldata('MNIST original')
X = zero_center(digits['data'].astype(np.float64))
np.random.shuffle(X)

Omega = np.random.uniform(0.0, 0.75, size=X.shape[1])
Xh = X + np.random.normal(0.0, Omega, size=X.shape)
```

After this step, the `X` variable will contain the zero-center original dataset, while `Xh` is the noisy version. The following screenshot shows a random selection of samples from both versions:



We can perform FA on both datasets using the Scikit-Learn `FactorAnalysis` class with the `n_components=64` parameter and check the score (the average log-likelihood over all samples). If the noise variance is known (or there's a good estimation), it's possible to include the starting point through the `noise_variance_init` parameter; otherwise, it will be initialized with the identity matrix:

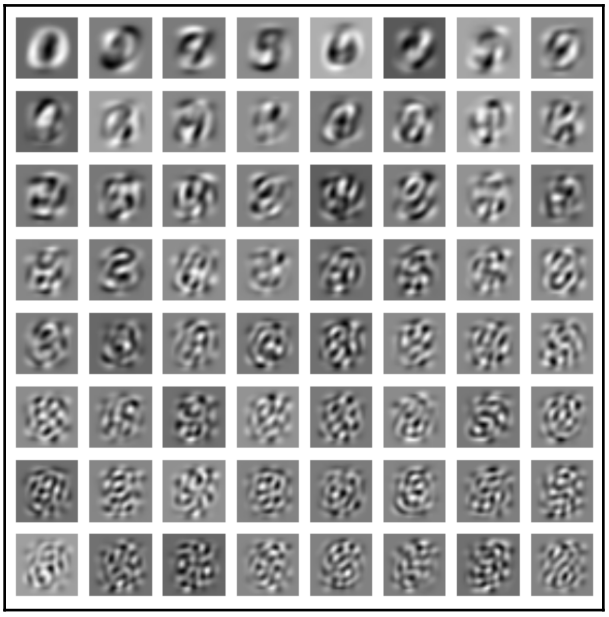
```
from sklearn.decomposition import FactorAnalysis

fa = FactorAnalysis(n_components=64, random_state=1000)
fah = FactorAnalysis(n_components=64, random_state=1000)

Xfa = fa.fit_transform(X)
Xfah = fah.fit_transform(Xh)

print(fa.score(X))
-2162.70193446

print(fah.score(Xh))
-3046.19385694
```

As expected, the presence of noise has reduced the final accuracy (MLE). Following an example provided by *A. Gramfort* and *D. A. Engemann* in the original Scikit-Learn documentation, we can create a benchmark for the MLE using the *Lodoit-Wolf* algorithm (a shrinking method for improving the condition of the covariance that is beyond the scope of this book.

For further information, read *A Well-Conditioned Estimator for Large-Dimensional Covariance Matrices, Ledoit O., Wolf M., Journal of Multivariate Analysis*, 88, 2/2004":

```
from sklearn.covariance import LedoitWolf

ldw = LedoitWolf()
ldwh = LedoitWolf()

ldw.fit(X)
ldwh.fit(Xh)

print(ldw.score(X))
-2977.12971009

print(ldwh.score(Xh))
-2989.27874799
```

With the original dataset, FA performs much better than the benchmark, while it's slightly worse in the presence of heteroscedastic noise. The reader can try other combinations using the grid search with different numbers of components and noise variances, and experiment with the effect of removing the zero-centering step. It's possible to plot the extracted components using the `components_` instance variable:



A plot of the 64 components extracted with the factor analysis on the original dataset

A careful analysis shows that the components are a superimposition of many low-level visual features. This is a consequence of the assumption to have a Gaussian prior distribution over the components ($z \sim N(0, I)$). In fact, one of the disadvantages of this distribution is its intrinsic denseness (the probability of sampling values far from the mean is often too high, while in some case, it would be desirable to have a peaked distribution that discourages values not close to its mean, to be able to observe more selective components). Moreover, considering the distribution $p[Z|X; \theta]$, the covariance matrix ψ could not be diagonal (trying to impose this constraint can lead to an unsolvable problem), leading to a resulting multivariate Gaussian distribution, which isn't normally made up of independent components. In general, the single variables $z_{i,}$ (conditioned to an input sample, x_i) are statistically dependent and the reconstruction $x_{i,}$ is obtained with the participation of almost all extracted features. In all these cases, we say that the *coding is dense* and the dictionary of features in *under-complete* (the dimensionality of the components is lower than $dim(x_i)$).

The lack of independence can be also an issue considering that any orthogonal transformation Q applied to A (the factor loading matrix) don't affect the distribution $p[X|Z, \theta]$. In fact, as $QQ^T=I$, the following applies:

$$AA^T + \Omega = AQQ^T A^T + \Omega$$

In other words, any feature rotation $(x = AQz + \nu)$ is always a solution to the original problem and it's impossible to decide which is the real loading matrix. All these conditions lead to the further conclusion that the mutual information among components is not equal to zero and neither close to a minimum (in this case, each of them carries a specific portion of information). On the other side, our main goal was to reduce the dimensionality. Therefore, it's not surprising to have dependent components because we aim to preserve the maximum amount of original information contained in $p(X)$ (remember that the amount of information is related to the entropy and the latter is proportional to the variance).

The same phenomenon can be observed in the PCA (which is still based on the Gaussian assumption), but in the last paragraph, we're going to discuss a technique, called ICA, whose goal is to create a representation of each sample (without the constraint of the dimensionality reduction) after starting from a set of statistically independent features. This approach, even if it has its peculiarities, belongs to a large family of algorithms called *sparse coding*. In this scenario, if the corresponding dictionary has $dim(z_i) > dim(x_i)$,it is called *over-complete* (of course, the main goal is no longer the dimensionality reduction).

However, we're going to consider only the case when the dictionary is at most complete $dim(z_i) = dim(x_i)$, because ICA with over-complete dictionaries requires a more complex approach. The level of sparsity, of course, is proportional to $dim(z_i)$ and with ICA, it's always achieved as a secondary goal (the primary one is always the independence between components).

Principal Component Analysis

Another common approach to the problem of reducing the dimensionality of a high-dimensional dataset is based on the assumption that, normally, the total variance is not explained equally by all components. If p_{data} is a multivariate Gaussian distribution with covariance matrix Σ, then the entropy (which is a measure of the amount of information contained in the distribution) is as follows:

$$H(p) = \frac{1}{2} log\left(det(2\pi e\Sigma)\right)$$

Therefore, if some components have a very low variance, they also have a limited contribution to the entropy, providing little additional information. Hence, they can be removed without a high loss of accuracy.

Just as we've done with FA, let's consider a dataset drawn from $p_{data} \sim N(0, \Sigma)$ (for simplicity, we assume that it's zero-centered, even if it's not necessary):

$$X = \{\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_M\} \;\; where \;\; \bar{x}_i \in \mathbb{R}^n$$

Our goal is to define a linear transformation, $z = A^T x$ (a vector is normally considered a column, therefore x has a shape $(n \times 1)$), such as the following:

$$\begin{cases} dim(\bar{z}_i) \ll n \\ H(p(z)) \approx H(p(x)) \end{cases}$$

As we want to find out the directions where the variance is higher, we can build our transformation matrix, A, starting from the eigen decomposition of the input covariance matrix, Σ (which is real, symmetric, and positive definite):

$$\Sigma = V\Omega V^T$$

V is an *(n × n)* matrix containing the eigenvectors (as columns), while Ω is a diagonal matrix containing the eigenvalues. Moreover, *V* is also orthogonal, hence the eigenvectors constitute a basis. An alternative approach is based on the **singular value decomposition** (**SVD**), which has an incremental variant and there are algorithms that can perform a decomposition truncated at an arbitrary number of components, speeding up the convergence process (such as the Scikit-Learn implementation `TruncatedSVD`).

In this case, it's immediately noticeable that the sample covariance is as follows:

$$\Sigma_s = \frac{1}{M} X^T X \;\; where \;\; X \in \mathbb{R}^{M \times n} \; and \; \Sigma_s \in \mathbb{R}^{n \times n}$$

If we apply the SVD to the matrix *X* (each row represents a single sample with a shape *(1, n)*), we obtain the following:

$$X = U \Lambda V^T \;\; where \;\; U \in \mathbb{R}^{M \times M}, \;\; \Lambda \; is \; diag(n \times n) \;\; and \;\; V \in \mathbb{R}^{n \times n}$$

U is a unitary matrix containing (as rows) the left singular vectors (the eigenvectors of XX^T), *V* (also unitary) contains (as rows) the right singular vectors (corresponding to the eigenvectors of X^TX), while *Λ* is a diagonal matrix containing the singular values of Σ_s (which are the square roots of the eigenvalues of both XX^T and X^TX). Conventionally, the eigenvalues are sorted by descending order and the eigenvectors are rearranged to match the corresponding position.

Hence, we can directly use the matrix *Λ* to select the most relevant eigenvalues (the square root is an increasing function and doesn't change the order) and the matrix *V* to retrieve the corresponding eigenvectors (the factor *1/M* is a proportionality constant). In this way, we don't need to compute and eigen decompose the covariance matrix *Σ* (contains $n \times n$ elements) and we can exploit some very fast approximate algorithms that work only with the dataset (without computing X^TX). Using the SVD, the transformation of *X* can be done directly, considering that *U* and *V* are unitary matrices (this means that $UU^T = U^TU = I$; therefore, the conjugate transpose is also the inverse):

$$Z = XA = U \Lambda V^T V = U \Lambda$$

Right now, *X* has only been projected in the eigenvector space (it has been simply rotated) and its dimensionality hasn't changed. However, from the definition of the eigenvector, we know that the following is true:

$$\Sigma \bar{v} = \lambda \bar{v}$$

If λ is large, the projection of v will be amplified proportionally to the variance explained by the direction of the corresponding eigenvector. Therefore, if it has not been already done, we can sort (and rename) the eigenvalues and the corresponding eigenvectors to have the following:

$$\lambda_1 > \lambda_2 > \ldots > \lambda_n$$

If we select the first top *k* eigenvalues, we can build a transformation matrix based on the corresponding eigenvectors (principal components) that projects X onto a subspace of the original eigenvector space:

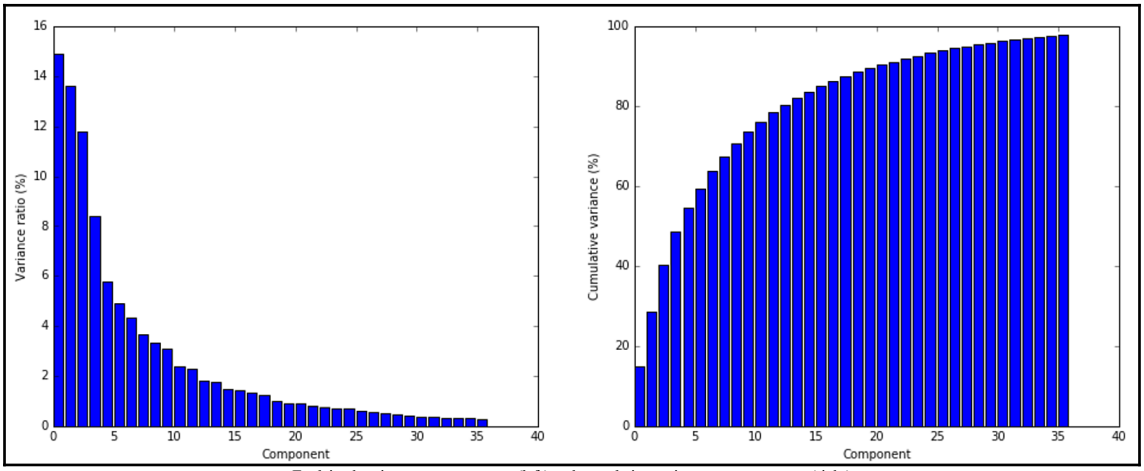
$$A_k = \{\bar{v}_1, \bar{v}_2, \ldots, \bar{v}_k\} \;\; where \;\; A_k \in \mathbb{R}^{n \times k}$$

Using the SVD, instead of A_k, we can directly truncate U and Λ, creating the matrices U_k (which contains only the top k eigenvectors) and Λ_k, a diagonal matrix with the top k eigenvalues.

When choosing the value for *k*, we are assuming that the following is true:

$$ExplainedVariance[A_k] \approx ExplainedVariance[V]$$

To achieve this goal, it is normally necessary to compare the performances with a different number of components. In the following graph, there's a plot where the variance ratio (variance explained by component n/total variance) and the cumulative variance are plotted as functions of the components:



Explained variance per component (left) and cumulative variance per component (right)

In this case, the first 10 components are able to explain 80% of the total variance. The remaining 25 components have a slighter and slighter impact and could be removed. However, the choice must be always based on the specific context, considering the loss of value induced by the loss of information.

A trick for determining the right number of components is based on the analysis of the eigenvalues of X. After sorting them, it's possible to consider the differences between subsequent values d = {$\lambda_1 - \lambda_2$, $\lambda_2 - \lambda_3$, ..., $\lambda_{n-1} - \lambda_n$}. The highest difference $\lambda_k - \lambda_{k+1}$ determines the index *k* of a potential optimal reduction (obviously, it's necessary to consider a constraint on the minimum value, because normally $\lambda_1 - \lambda_2$ is the highest difference). For example, if d = {4, 4, 3, 0.2, 0.18, 0.05} the original dimensionality is n=6; however, $\lambda_4 - \lambda_5$ is the smallest difference, so, it's reasonable to reduce the dimensionality to $(n + 1) - k = 3$. The reason is straightforward, the eigenvalues determine the magnitude of each component, but we need a relative measure because the scale changes. In the example, the last three eigenvectors point to directions where the explained variance is negligible when compared to the first three components.

Once we've defined the transformation matrix A_k, it's possible to perform the actual projection of the original vectors in the new subspace, through the relation:

$$\bar{z} = A_k^T \bar{x} \;\; where \;\; \bar{z} \in \mathbb{R}^{k \times 1}, A_k^T \in \mathbb{R}^{n \times k} \; and \; \bar{x} \in \mathbb{R}^{n \times 1}$$

The complete transformation of the whole dataset is simply obtained as follows:

$$Z = XA_k = U_k \Lambda_k$$

Now, let's analyze the new covariance matrix $E[Z^T Z]$. If the original distribution p_{data} $x \sim N(0, \Sigma)$, $p(z)$ will also be Gaussian with mean and covariance:

$$\mu_z = E[Z] = A^T E[X] = 0$$
$$\Sigma_z = E[Z^T Z] = A^T E[X^T X] A = A^T \Sigma A = A^T V \Omega V^T A$$

We know that Σ is orthogonal; therefore, $v_i \bullet v_j = 0$ if $i \neq j$. If we analyze the term A^TV, we get the following:

$$A^TV = \begin{pmatrix} \bar{v}_1 \\ \bar{v}_2 \\ \vdots \\ \bar{v}_k \end{pmatrix} \begin{pmatrix} \bar{v}_1 & \bar{v}_2 & \cdots & \bar{v}_n \end{pmatrix} =$$

$$= \begin{pmatrix} \bar{v}_1 \cdot \bar{v}_1 & \bar{v}_1 \cdot \bar{v}_2 & \cdots & \bar{v}_1 \cdot \bar{v}_n \\ \bar{v}_2 \cdot \bar{v}_1 & \bar{v}_2 \cdot \bar{v}_2 & \cdots & \bar{v}_2 \cdot \bar{v}_n \\ \vdots & \vdots & \ddots & \vdots \\ \bar{v}_k \cdot \bar{v}_1 & \bar{v}_k \cdot \bar{v}_2 & \cdots & \bar{v}_k \cdot \bar{v}_n \end{pmatrix} = \begin{pmatrix} \bar{v}_1 \cdot \bar{v}_1 & 0 & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \cdots & 0 \\ 0 & 0 & \bar{v}_k \cdot \bar{v}_k & 0 & \cdots & 0 \end{pmatrix}$$

Considering that Ω is diagonal, the resulting matrix Σ_z will be diagonal as well. This means that the PCA decorrelates the transformed covariance matrix. At the same time, we can state that every algorithm that decorrelates the input covariance matrix performs a PCA (with or without dimensionality reduction). For example, the *whitening process* is a particular PCA without dimensionality reduction, while Isomap (see `Chapter 3`, *Graph-Based Semi-Supervised Learning*) performs the same operation working with the Gram matrix with a more geometric approach. This result will be used in `Chapter 6`, *Hebbian Learning*, to show how some particular neural networks can perform a PCA without eigen decomposing Σ.

Let's now consider a FA with homoscedastic noise. We have seen that the covariance matrix of the conditional distribution, $p(X|Z; \theta)$, is equal to $AA^T + \Omega$. In the case of homoscedastic noise, it becomes $AA^T + \omega I$. For a generic covariance matrix, Σ, it's possible to prove that adding a constant diagonal matrix $(\Sigma + aI)$ doesn't modify the original eigenvectors and shifts the eigenvalues by the same quantity:

$$\Sigma + aI = V\Psi V^T + aI = V\Psi V^T + aVIV^T = V(\Psi + aI)V^T$$

Therefore, we can consider the generic case of absence of noise without loss of generality. We know that the goal of FA (with $\Omega = (0)$) is finding the matrix, A, so that $AA^T \approx Q$ (the input covariance). Hence, thanks to the symmetry and imposing the asymptotic equality, we can write the following:

$$A_\infty A_\infty^T = Q \;\Rightarrow\; A_\infty A_\infty^T = V\Omega V^T = \left(V \left(\Omega^{\frac{1}{2}} \right) \left(\Omega^{\frac{1}{2}} \right)^T V^T \right) \;\Rightarrow\; A_\infty = V\Omega^{\frac{1}{2}}$$

This result implies that the FA is a more generic (and robust) way to manage the dimensionality reduction in the presence of heteroscedastic noise, and the PCA is a restriction to homoscedastic noise. When a PCA is performed on datasets affected by heteroscedastic noise, the MLE worsens because the different noise components, altering the magnitude of the eigenvalues at different levels, can drive to the selection of eigenvectors that, in the original dataset, explain only a low percentage of the variance (and in a noiseless scenario, it would be normally discarded in favor of more important directions). If you think of the example discussed at the beginning of the previous paragraph, we know that the noise is strongly heteroscedastic, but we don't have any tools to inform the PCA to cope with it and the variance of the first component will be much higher than expected, considering that the two sources are identical. Unfortunately, in a real- life scenario, the noise is correlated and neither a factor nor a PCA can efficiently solve the problem when the noise power is very high. In all those cases, more sophisticated denoising techniques must be employed. Whenever, instead, it's possible to define an approximate diagonal noise covariance matrix, FA is surely more robust and efficient than PCA. The latter should be considered only in noiseless or *quasi*-noiseless scenarios. In both cases, the results can never lead to well-separated features. For this reason, the ICA has been studied and many different strategies have been engineered.

The complete algorithm for the PCA is as follows:

1. Create a matrix $X^{(M \times n)}$ containing all the samples x_i as rows
    1. Eigen decomposition version:
        1. Compute the covariance matrix $\Sigma = [X^T X]$
        2. Eigen decompose $\Sigma = V \Omega V^T$
    2. SVD version:
        1. Compute the SVD on the matrix $X = U \Lambda V^T$
    3. Select the top k eigenvalues (from Ω or Λ) and the corresponding eigenvectors (from V)
    4. Create the matrix A with shape $(n \times k)$, whose columns are the top k eigenvectors (each of them has a shape (n × 1))
    5. Project the dataset into the low-dimensional space Z = XA (eigen decomposition) or $Z = U\Lambda$ *(SVD)*

Some packages (such as Scipy, which is the backend for many NumPy function, such as`np.linalg.svd()`) return the matrix V (right singular vectors) already transposed. In this case, it's necessary to use V^T instead of V in step 3 of the algorithm. I suggest always checking the documentation when implementing these kinds of algorithms.

An example of PCA with Scikit-Learn

We can repeat the same experiment made with the FA and heteroscedastic noise to assess the MLE score of the PCA. We are going to use the `PCA` class with the same number of components (`n_components=64`). To achieve the maximum accuracy, we also set the `svd_solver='full'` parameter, to force Scikit-Learn to apply a full SVD instead of the truncated version. In this way, the top eigenvalues are selected only after the decomposition, avoiding the risk of imprecise estimations:

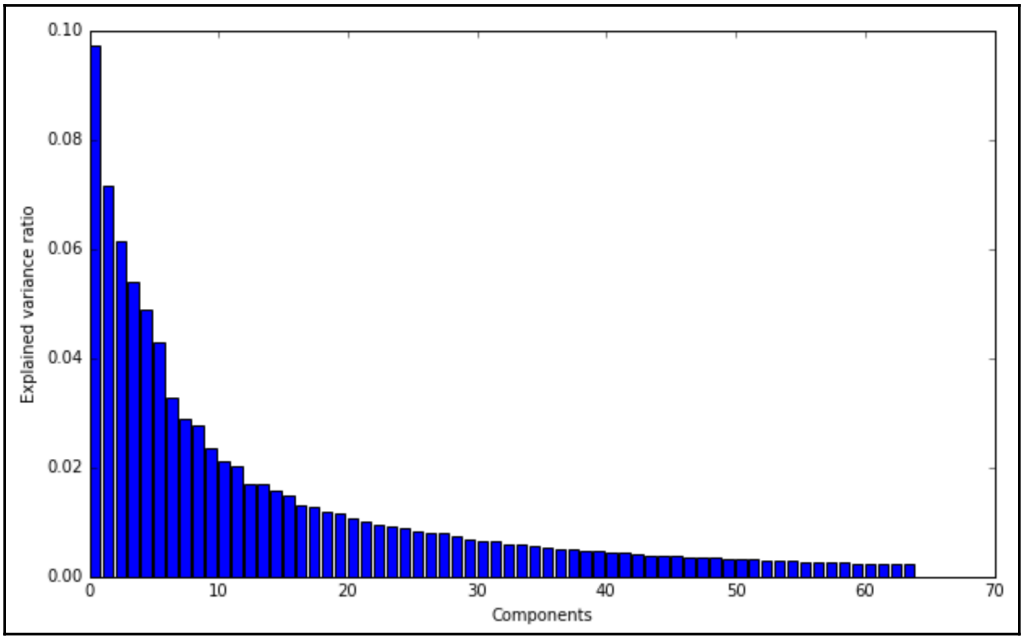
```
from sklearn.decomposition import PCA

pca = PCA(n_components=64, svd_solver='full', random_state=1000)
Xpca = pca.fit_transform(Xh)

print(pca.score(Xh))
-3772.7483580391995
```

The result is not surprising: the MLE is much lower than FA, because of the wrong estimations made due to the heteroscedastic noise. I invite the reader to compare the results with different datasets and noise levels, considering that the training performance of PCA is normally higher than FA. Therefore, when working with large datasets, a good trade-off is surely desirable. As with FA, it's possible to retrieve the components through the `components_` instance variable.

It's interesting to check the total explained variance (as a fraction of the total input variance) through the component-wise instance array `explained_variance_ratio_`:

```
print(np.sum(pca.explained_variance_ratio_))
0.862522337381
```

With 64 components, we are explaining 86% of the total input variance. Of course, it's also useful to compare the explained variance using a plot:



As usual, the first components explain the largest part of the variance; however, after about the twentieth component, each contribution becomes lower than 1% (decreasing till about 0%). This analysis suggests two observations: it's possible to further reduce the number of components with an acceptable loss (using the previous snippet, it's easy to extend the sum only the first *n* components and compare the results) and, at the same time, the PCA will be able to overcome a higher threshold (such as 95%) only by adding a large number of new components. In this particular case, we know that the dataset is made up of handwritten digits; therefore, we can suppose that the tail is due to secondary differences (a line slightly longer than average, a marked stroke, and so on); hence, we can drop all the components with n > 64 (or less) without problems (it's also easy to verify visually a rebuilt image using the `inverse_transform()` method). However, it is always best practice to perform a complete analysis before moving on to further processing steps, particularly when the dimensionality of X is high.

Another interesting approach to determine the optimal number of components has been proposed by Minka (*Automatic Choice of Dimensionality for PCA*, *Minka T.P.*, *NIPS* 2000") and it's based on the Bayesian model selection. The idea is to use the MLE to optimize the likelihood $p(X|k)$ where k is a parameter indicating the number of components. In other words, it doesn't start analyzing the explained variance, but determines a value of $k < n$ so that the likelihood keeps being the highest possible (implicitly, k will explain the maximum possible variance under the constraint of $max(k) = k_{max}$). The theoretical foundation (with tedious mathematical derivations) of the method is presented in the previously mentioned paper however, it's possible to use this method with Scikit-Learn by setting the `n_components='mle'` and `svd_solver='full'` parameters.

Independent component analysis

We have seen that the factors extracted by a PCA are decorrelated, but not independent. A classic example is the *cocktail party:* we have a recording of many overlapped voices and we would like to separate them. Every single voice can be modeled as a random process and it's possible to assume that they are statistically independent (this means that the joint probability can be factorized using the marginal probabilities of each source). Using FA or PCA, we are able to find uncorrelated factors, but there's no way to assess whether they are also independent (normally, they aren't). In this section, we are going to study a model that is able to produce sparse representations (when the dictionary isn't under-complete) with a set of statistically independent components.

Let's assume we have a zero-centered and whitened dataset X sampled from $N(0, I)$ and noiseless linear transformation:

$$\bar{x} = A\bar{z} \;\; where \;\; x \sim N(0, I) \;\; and \;\; p(\bar{z}; \theta) = \alpha \prod_k e^{f_k(\bar{z})}$$

In this case, the prior over, z, is modeled as a product of independent variables (α is the normalization factor), each of them represented as a generic exponential where the function $f_k(z)$ must be non-quadratic, that is, $p(z; \theta)$ cannot be Gaussian. Furthermore, we assume that the variance of z_i is equal to 1, therefore, $p(x|z; \theta) \sim N(Az, AA^T)$. The joint probability $p(X, z; \theta) = p(X|z; \theta)p(z|\theta)$ is equal to the following:

$$p(X,z;\theta) = \left(\prod_{i=1}^{M} \frac{1}{\sqrt{(2\pi)^n det(AA^T)}} e^{-\frac{1}{2}(\bar{x}_i - A\bar{z})^T (AA^T)^{-1}(\bar{x}_i - A\bar{z})} \right) \left(\alpha \prod_k e^{f_k(\bar{z})} \right)$$

If *X* has been whitened, *A* is orthogonal (the proof is straightforward); hence, the previous expression can be simplified. However, applying the EM algorithm requires determining *p(z|X; θ)* and this is quite difficult. The process could be easier after choosing a suitable prior distribution for *z*, that is, $f_k(z)$, but as we discussed at the beginning of the chapter, this assumption can have dramatic consequences if the real factors are distributed differently. For these reasons, other strategies have been studied.

The main concept that we need to enforce is having a non-Gaussian distribution of the factors. In particular, we'd like to have a peaked distribution (inducing sparseness) with heavy tails. From the theory, we know that the standardized fourth moment (also called *Kurtosis*) is a perfect measure:

$$Kurt(X) = E_{x \sim X} \left[\left(\frac{x - \mu_x}{\sigma_x} \right)^4 \right]$$

For a Gaussian distribution, *Kurt[X]* is equal to three (which is often considered as the reference point, determining the so called *Excess Kurtosis* = *Kurtosis - 3*), while it's larger for a family of distributions, called *Leptokurtotic* or super-Gaussian, which are peaked and heavy-tailed (also, the distributions with *Kurt[X] < 3*, called *Platykurtotic* or sub-Gaussian, can be good candidates, but they are less peaked and normally only the super-Gaussian distributions are taken into account). However, even if accurate, this measure is very sensitive to outliers because of the fourth power. For example, if $x \sim N(0, 1)$ and $z = x + v$, where v is a noise term that alters a few samples, increasing their value to two, the result can be a super-Gaussian (*Kurt[x] > 3*) even if, after filtering the outliers out, the distribution has *Kurt[x] = 3 (Gaussian)*.

To overcome this problem, Hyvärinen and Oja (*Independent Component Analysis: Algorithms and Applications*, *Hyvarinen A.*, *Oja E.*, Neural Networks 13/2000) proposed a solution based on another measure, the *negentropy*. We know that the entropy is proportional to the variance and, given the variance, the Gaussian distribution has the maximum entropy (for further information, read *Mathematical Foundations of Information Theory*, *Khinchin A. I.*, *Dover Publications*); therefore, we can define the measure:

$$H_N(X) = H\left(X_{\bar{x} \sim N(0,\Sigma)}\right) - H(X)$$

Formally, the negentropy of *X* is the difference between the entropy of a Gaussian distribution with the same covariance and the entropy of *X* (we are assuming both zero-centered). It's immediately possible to understand that $H_N(X) \geq 0$, hence the only way to maximize it is by reducing *H(X)*. In this way, *X* becomes less random, concentrating the probability around the mean (in other words, it becomes super-Gaussian). However, the previous expression cannot be easily adapted to closed-form solutions, because *H(X)* needs to be computed over all the distribution of *X*, which must be estimated. For this reason, the same authors proposed an approximation based on non-quadratic functions (remember that in the context of ICA, a quadratic function can be never be employed because it would lead to a Gaussian distribution) that is useful to derive a fixed-point iterative algorithm called *FastICA* (indeed, it's really faster than EM).

Using k functions $f_k(x)$, the approximation becomes as follows:

$$H_N(X) \approx \sum_{i=1}^{k} \alpha_i (E[f_i(\bar{x})] - E[f_i(\bar{n})])^2 \;\; where \;\; \bar{n} \sim N(0, I) \;\; and \;\; \alpha_i > 0$$

In many real-life scenarios, a single function is enough to achieve a reasonable accuracy and one of the most common choices for f(x) is as follows:

$$f(x) = \frac{1}{a} log(cosh(ax)) = \frac{1}{a} log \, \frac{e^{ax} + e^{-ax}}{2}$$

In the aforementioned paper, the reader can find some alternatives that can be employed when this function fails in forcing statistical independence between components.

If we invert the model, we get $z = Wx$ with $W = A^{-1}$; therefore, considering a single sample, the approximation becomes as follows:

$$H_N(X) \approx \left(E[f(\bar{w}^T \bar{x})] - E[f(\bar{n})]\right)^2 \;\; where \;\; \bar{n} \sim N(0, I)$$

Clearly, the second term doesn't depend on *w* (in fact, it's only a reference) and can be excluded from the optimization. Moreover, considering the initial assumptions, $E[Z^T Z]=W$ $E[X^T X]\, W^T = I$, therefore $WW^T = I$, *i.e.* $||w||^2 = 1$. Hence, our goal is to find the following:

$$\bar{w}_{opt} = argmax_{\bar{w}} \; E[f(\bar{w}^T \bar{x})]^2 \;\; subject \; to \;\; \|\bar{w}\|^2 = 1$$

In this way, we are forcing the matrix *W* to transform the input vector *x*, so that *z* has the lowest possible entropy; therefore, it's super-Gaussian. The maximization process is based on convex optimization techniques that are beyond the scope of this book (the reader can find all the details of Lagrange theorems in *Luenberger D. G., Optimization by Vector Space Methods, Wiley*); therefore, we directly provide the iterative step that must be performed:

$$w_{t+1} = E\left[\bar{x} f'(\bar{w}_t^T \bar{x})\right] - E\left[f''(\bar{w}_t^T \bar{x})\right] \bar{w}_t$$

Of course, to ensure $||w||^2 = 1$, after each step, the weight vector w must be normalized ($w_{t+1} = w_{t+1} / ||w_{t+1}||$).

In a more general context, the matrix *W* contains more than one weight vector and, if we apply the previous rule to find out the independent factors, it can happen that some elements, $w_i^T x$, are correlated. A strategy to avoid this problem is based on the gram-schmidt orthonormalization process, which decorrelates the components one by one, subtracting the projections of the current component (w_n) onto all the previous ones ($w_1, w_2, ..., w_{n-1}$) to w_n. In this way, w_n is forced to be orthogonal to all the other components.

Even if this method is simple and doesn't require much effort, it's preferable a global approach that can work directly with the matrix *W* at the end of an iteration (so that the order of the weights is not fixed). As explained in *Fast and robust fixedpoint algorithms for independent component analysis*, *Hyvarinen A.*, *IEEE Transactions on Neural Networks* this result can be achieved with a simple sub-algorithm that we are including in the final *FastICA* algorithm:

1. Set random initial values for W_0
2. Set a threshold *Thr* (for example 0.001)
    1. Independent component extraction
    2. For each *w* in *W*:
        1. While $||w_{t+1} - w_t|| > Thr$:
            1. Compute $w_{t+1} = E[x \cdot f'(w_t^T x)] - E[f''(w_t^T x)] w_t$
            2. $w_{t+1} = w_{t+1} / ||w_{t+1}||$
    3. Orthonormalization
    4. While $||W_{t+1} - W_t||_F > Thr$:
        1. $W_t = W_t / sqrt(||W_t W_t^T||)$
        2. $W_{t+1} = (3/2)W_t - (1/2)WW^TW$

This process can be also iterated for a fixed number of times, but the best approach is based on using both a threshold and a maximum number of iterations.

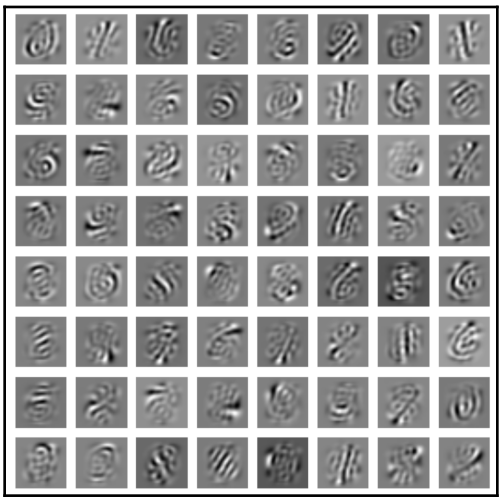
An example of FastICA with Scikit-Learn

Using the same dataset, we can now test the performance of the ICA. However, in this case, as explained, we need to zero-center and whiten the dataset, but fortunately these preprocessing steps are done by the Scikit-Learn implementation (if the parameter `whiten=True` is omitted).

To perform the ICA on the MNIST dataset, we're going to instantiate the `FastICA` class, passing the arguments `n_components=64` and the maximum number of iterations `max_iter=5000`. It's also possible to specify which function will be used to approximate the negentropy; however, the default is *log cosh(x)*, which is normally a good choice:

```
from sklearn.decomposition import FastICA

fastica = FastICA(n_components=64, max_iter=5000, random_state=1000)
fastica.fit(X)
```

At this point, we can visualize the components (which are always available through the `components_` instance variance):

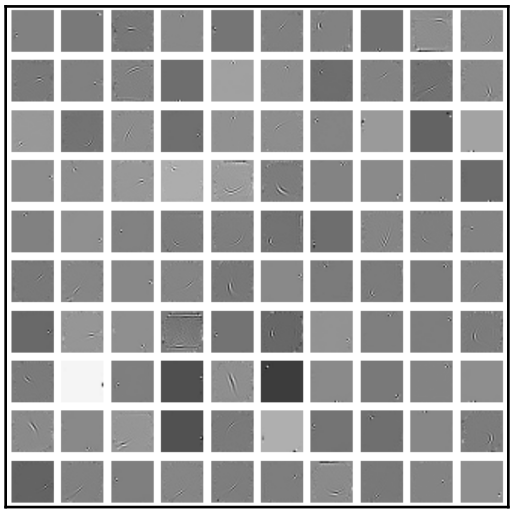


Independent components of the MNIST dataset extracted by the FastICA algorithm (64 components)

There are still some redundancies (the reader can try to increase the number of components) and background noise; however, it's now possible to distinguish some low-level features (such as oriented stripes) that are common to many digits. This representation isn't very sparse yet. In fact, we're always using 64 components (like for FA and PCA); therefore, the dictionary is under-complete (the input dimensionality is 28 × 28 = 784). To see the difference, we can repeat the experiment with a dictionary ten times larger, setting `n_components=640`:

```
fastica = FastICA(n_components=640, max_iter=5000, random_state=1000)
fastica.fit(Xs)
```

A subset of the new components (100) is shown in the following screenshot:



Independent components of the MNIST dataset extracted by the FastICA algorithm (640 components)

The structure of these components is almost elementary. They represent oriented stripes and positional dots. To check how an input is rebuilt, we can consider the mixing matrix *A* (which is available as the `mixing_` instance variable). Considering the first input sample, we can check how many factors have a weight less than half of the average:

```
M = fastica.mixing_
M0 = M[0] / np.max(M[0])

print(len(M0[np.abs(M0) < (np.mean(np.abs(M0)) / 2.0)]))
233
```

The sample is rebuilt using approximately 410 components. The level of sparsity is higher, but considering the granularity of the factors, it's easy to understand that many of them are needed to rebuild even a single structure (like the image of a 1) where long lines are present. However, this is not a drawback because, as already mentioned, the main goal of the ICA is to extract independent components. Considering an analogy with the *cocktail party* example, we could deduce that each component represents a phoneme, not the complete sound of a word or a sentence.

The reader can test a different number of components and compare the results with the ones achieved by other sparse coding algorithms (such as Dictionary Learning or Sparse PCA).

Addendum to HMMs

In the previous chapter, we discussed how it's possible to train a HMM using the forward-backward algorithm and we have seen that it is a particular application of the EM algorithm. The reader can now understand the internal dynamic in terms of E and M steps. In fact, the procedure starts with randomly initialized A and B matrices and proceeds in an alternating manner:

- **E-Step**:
  - The estimation of the probability α^t_{ij} that the HMM is in the state *i* at time *t* and in the state *j* at time *t+1* given the observations and the current parameter estimations (A and B)
  - The estimation of the probability β^t_i that the HMM is in the state *i* at time *t* given the observations and the current parameter estimations (A and B)
- **M-Step**:
  - Computing the new estimation for the transition probabilities a_{ij} *(A)* and for the emission probabilities b_{ip} *(B)*

The procedure is repeated until the convergence is reached. Even if there's no explicit definition of a Q function, the E-step determines a split expression for the expected complete data likelihood of the model given the observations (using both the Forward and Backward algorithms), while the M-Step corrects parameters A and B to maximize this likelihood.

Summary

In this chapter, we presented the EM algorithm, explaining the reasons that justify its application in many statistical learning contexts. We also discussed the fundamental role of hidden (latent) variables, in order to derive an expression that is easier to maximize (the Q function).

We applied the EM algorithm to solve a simple parameter estimation problem and afterward to prove the Gaussian Mixture estimation formulas. We showed how it's possible to employ the Scikit-Learn implementation instead of writing the whole procedure from scratch (like in `Chapter 2`, *Introduction to Semi-Supervised Learning*).

Afterward, we analyzed three different approaches to component extraction. FA assumes that we have a small number of Gaussian latent variables and a Gaussian decorrelated noise term. The only restriction on the noise is to have a diagonal covariance matrix, so two different scenarios are possible. When we are in the presence of heteroscedastic noise, the process is an actual FA. When, instead, the noise is homoscedastic, the algorithm becomes the equivalent of a PCA. In this case, the process is equivalent to check the sample space in order to find the directions where the variance is higher. Selecting only the most important directions, we can project the original dataset onto a low-dimensional subspace, where the covariance matrix becomes decorrelated.

One of the problems of both FA and PCA is their assumption to model the latent variables with Gaussian distributions. This choice simplifies the model, but at the same time, yields dense representations where the single components are statistically dependent. For this reason, we have investigated how it's possible to force the factor distribution to become sparse. The resulting algorithm, which is generally faster and more accurate than the MLE, is called FastICA and its goal is to extract a set of statistically independent components with the maximization of an approximation of the negentropy.

In the end, we provided a brief explanation of the HMM forward-backward algorithm (discussed in the previous chapter) considering the subdivision into E and M steps. Other EM-specific applications will be discussed in the next chapters.

In the next chapter, we are going to introduce the fundamental concepts of Hebbian learning and self-organizing maps, which are still very useful to solve many specific problems, such as principal component extraction, and have a strong neurophysiological foundation.

6
Hebbian Learning and Self-Organizing Maps

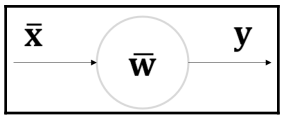
In this chapter, we're going to introduce the concept of Hebbian learning, based on the methods defined by the psychologist Donald Hebb. These theories immediately showed how a very simple biological law is able to describe the behavior of multiple neurons in achieving complex goals and was a pioneering strategy that linked the research activities in the fields of artificial intelligence and computational neurosciences.

In particular, we are going to discuss the following topics:

- The Hebb rule for a single neuron, which is a simple but biologically plausible behavioral law
- Some variants that have been introduced to overcome a few stability problems
- The final result achieved by a Hebbian neuron, which consists of computing the first principal component of the input dataset
- Two neural network models (Sanger's network and Rubner-Tavan's network) that can extract a generic number of principal components
- The concept of **Self-Organizing Maps** (**SOMs**) with a focus on the Kohonen Networks

Hebb's rule

Hebb's rule has been proposed as a conjecture in 1949 by the Canadian psychologist Donald Hebb to describe the synaptic plasticity of natural neurons. A few years after its publication, this rule was confirmed by neurophysiological studies, and many research studies have shown its validity in many application, of Artificial Intelligence. Before introducing the rule, it's useful to describe the generic Hebbian neuron, as shown in the following diagram:



Generic Hebbian neuron with a vectorial input

The neuron is a simple computational unit that receives an input vector x, from the pre-synaptic units (other neurons or perceptive systems) and outputs a single scalar value, y. The internal structure of the neuron is represented by a weight vector, w, that models the strength of each synapse. For a single multi-dimensional input, the output is obtained as follows:

$$y = \bar{w} \cdot \bar{x}$$

In this model, we are assuming that each input signal is encoded in the corresponding component of the vector, x; therefore, x_i is processed by the synaptic weight $w_{i,}$ and so on. In the original version of Hebb's theory, the input vectors represent neural firing rates, which are always non-negative. This means that the synaptic weights can only be strengthened (the neuroscientific term for this phenomenon is **long-term potentiation** (**LTP**)). However, for our purposes, we assume that x is a real-valued vector, as is w. This condition allows modeling more artificial scenarios without a loss of generality.

The same operation performed on a single vector holds when it's necessary to process many input samples organized in a matrix. If we have N m-dimensional input vectors, the formula becomes as follows:

$$\bar{y} = X\bar{w} \;\; where \; X \in \mathbb{R}^{N \times m}, \; \bar{w} \in \mathbb{R}^m \; and \; \bar{y} \in \mathbb{R}^N$$

The basic form of Hebb's rule in a discrete form can be expressed (for a single input) as follows:

$$\Delta\bar{w} = \eta y\bar{x} = \eta(\bar{x} \cdot \bar{w})\bar{x}$$

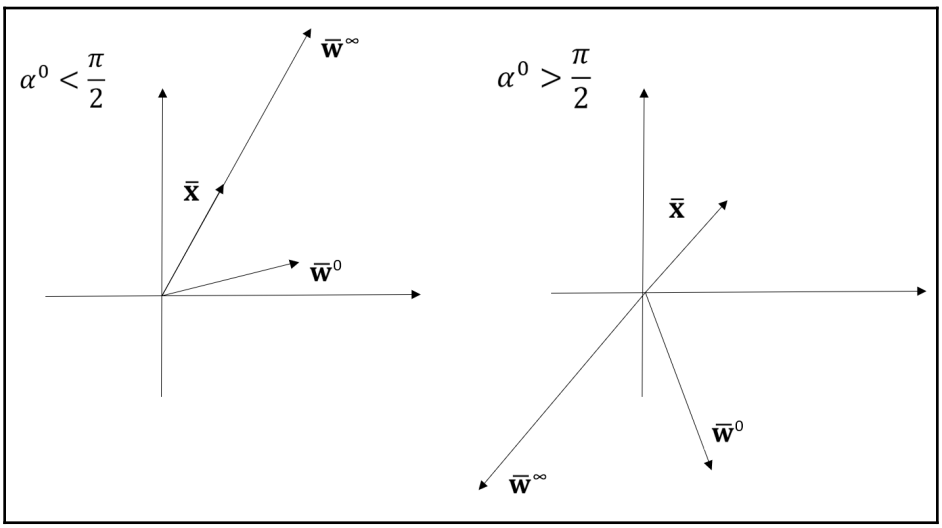
The weight correction is hence a vector that has the same orientation of x and magnitude equal to $|x|$ multiplied by a positive parameter, η, which is called the learning rate and the corresponding output, y (which can have either a positive or a negative sign). The sense of Δw is determined by the sign of y; therefore, under the assumption that x and y are real values, two different scenarios arise from this rule:

- If $x_i > 0$ (< 0) and $y > 0$ (< 0), w_i is strengthened
- If $x_i > 0$ (< 0) and $y < 0$ (> 0), w_i is weakened

It's easy to understand this behavior considering two-dimensional vectors:

$$sign(y) = sign(\bar{w} \cdot \bar{x}) = sign(|w|\,|x|\,cos(\alpha))$$

Therefore, if the initial angle α between w and x is less than 90°, w will have the same orientation of x and viceversa if α is greater than 90°. In the following diagram, there's a schematic representation of this process:



Vectorial analysis of Hebb's rule

It's possible to simulate this behavior using a very simple Python snippet. Let's start with a scenario where α is less than 90° and 50 iterations:

```
import numpy as np

w = np.array([1.0, 0.2])
x = np.array([0.1, 0.5])
alpha = 0.0

for i in range(50):
    y = np.dot(w, x.T)
    w += x*y
    alpha = np.arccos(np.dot(w, x.T) / (np.linalg.norm(w) *
np.linalg.norm(x)))

print(w)
[  8028.48942243  40137.64711215]

print(alpha * 180.0 / np.pi)
0.00131766983584
```

As expected, the final angle, α, is close to zero and w has the same orientation and sense of x. We can now repeat the experiment with α greater than 90° (we change only the value of w because the procedure is the same):

```
w = np.array([1.0, -1.0])

...

print(w)
[-16053.97884486 -80275.89422431]

print(alpha * 180.0 / np.pi)
179.999176456
```

In this case, the final angle, α, is about 180° and, of course, w has the opposite sense with respect to x.

The scientist S. Löwel expressed this concept with the famous sentence:

> *"Neurons that fire together wire together"*

We can re-express this concept (adapting it to a machine learning scenario) by saying that the main assumption of this approach is based on the idea that when pre- and post-synaptic units are coherent (their signals have the same sign), the connection between neurons becomes stronger and stronger. On the other side, if they are discordant, the corresponding synaptic weight is decreased. For the sake of precision, if *x* is a spiking rate, it should be represented as a real function *x(t)* as well as *y(t)*. According to the original Hebbian theory, the discrete equation must be replaced by a differential equation:

$$\frac{\mathrm{d}\bar{w}}{\mathrm{d}t} = \eta y \bar{x}$$

If *x(t)* and *y(t)* have the same fire rate, the synaptic weight is strengthened proportionally to the product of both rates. If instead there's a relatively long delay between the pre-synaptic activity *x(t)* and the post-synaptic one *y(t)*, the corresponding weight is weakened. This is a more biologically plausible explanation of the relation *fire together → wire together*.

However, even if the theory has a strong neurophysiological basis, some modifications are necessary. In fact, it's easy to understand that the resulting system is always unstable. If two inputs are repeatedly applied (both real values and firing rates), the norm of the vector, w, grows indefinitely and this isn't a plausible assumption for a biological system. In fact, if we consider a discrete iteration step, we have the following equation:

$$\bar{w}_{k+1} = \bar{w}_k + \eta(\bar{w}_k \cdot \bar{x})\bar{x} \;\Rightarrow\; \bar{w}_{k+1} \cdot \bar{x} = \bar{w}_k \cdot \bar{x} + \eta(\bar{w}_k \cdot \bar{x})\bar{x} \cdot \bar{x} \;\Rightarrow\; y_{k+1} = y_k(1 + \eta|\bar{x}|^2)$$

The previous output, y_k, is always multiplied by a factor greater than *1* (except in the case of null input), therefore it grows without a bound. As $y = w \cdot x$, this condition implies that the magnitude of *w* increases (or remains constant if the magnitude of *x* is null) at each iteration (a more rigorous proof can be easily obtained considering the original differential equation).

Such a situation is not only biologically unacceptable, but it's also necessary to properly manage it in machine learning problems in order to avoid a numerical overflow after a few iterations. In the next paragraph, we're going to discuss some common methods to overcome this issue. For now, we can continue our analysis without introducing a correction factor.

Let's now consider a dataset, *X*:

$$X = \{\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_N\} \; where \; \bar{x}_i \in \mathbb{R}^m$$

We can apply the rule iteratively to all elements, but it's easier (and more useful) to average the weight modifications over the input samples (the index now refers to the whole specific vector, not to the single components):

$$\Delta\bar{w} = \frac{\eta}{N}\sum_{i=1}^{N} y_i \bar{x}_i = \frac{\eta}{N}\sum_{i=1}^{N}(\bar{w}\cdot\bar{x}_i)\bar{x}_i = \frac{\eta}{N}\sum_{i=1}^{N}(\bar{x}_i^T\cdot\bar{x}_i)\bar{w} = \eta C\bar{w}$$

In the previous formula, *C* is the input correlation matrix:

$$C = \begin{pmatrix} \frac{1}{N}\sum_i x_1^i x_1^i & \cdots & \frac{1}{N}\sum_i x_1^i x_m^i \\ \vdots & \ddots & \vdots \\ \frac{1}{N}\sum_i x_m^i x_1^i & \cdots & \frac{1}{N}\sum_i x_m^i x_m^i \end{pmatrix} = \frac{1}{N}X^T X$$

For our purposes, however, it's useful to consider a slightly different Hebbian rule based on a threshold θ for the input vector (there's also a biological reason that justifies this choice, but it's beyond the scope of this book; the reader who is interested can find it in *Theoretical Neuroscience*, *Dayan P.*, *Abbott F. L.*, *The MIT Press*).

It's easy to understand that in the original theory where *x(t)* and *y(t)* are firing rates, this modification allows a phenomenon opposite to LTP and called **long-term depression** (**LTD**). In fact, when $x(t) < \theta$ and *y(t)* is positive, the product $(x(t) - \theta)y(t)$ is negative and the synaptic weight is weakened.

If we set $\theta = \langle x \rangle \approx E[X]$, we can derive an expression very similar to the previous one, but based on the input covariance matrix (unbiased through the Bessel's correction):

$$\Delta\bar{w} = \frac{\eta}{N-1}\sum_{i=1}^{N} y_i(\bar{x}_i - \langle\bar{x}\rangle) = \frac{\eta}{N-1}\sum_{i=1}^{N}(\bar{w}\cdot(\bar{x}_i - \langle\bar{x}\rangle))(\bar{x}_i - \langle\bar{x}\rangle) =$$

$$= \frac{\eta}{N-1}\sum_{i=1}^{N}(\bar{x}_i - \langle\bar{x}\rangle)^T(\bar{x}_i - \langle\bar{x}\rangle)\bar{w} = \eta\Sigma\bar{w}$$

For obvious reasons, this variant of the original Hebb's rule is called the **covariance rule**.

It's also possible to use the **Maximum Likelihood Estimation** (**MLE**) (or biased) covariance matrix (dividing by *N*), but it's important to check which version is adopted by the mathematical package that is employed. When using NumPy, it's possible to decide the version using the `np.cov()` function and setting the `bias=True/False` parameter (the default value is `False`). However, when $N \gg 1$, the difference between versions decreases and can often be discarded. In this book, we'll use the unbiased version. The reader who wants to see further details about the Bessel's correction can read *Applied Statistics*, *Warner R.*, *SAGE Publications*.

Analysis of the covariance rule

The covariance matrix Σ is real and symmetric. If we apply the eigendecomposition, we get (for our purposes it's more useful to keep V^{-1} instead of the simplified version V^{T}):

$$\Sigma = V\Omega V^{-1}$$

V is an orthogonal matrix (thanks to the fact that Σ is symmetric) containing the eigenvectors of Σ (as columns), while Ω is a diagonal matrix containing the eigenvalues. Let's suppose we sort both eigenvalues (λ_1, λ_2, ..., λ_m) and the corresponding eigenvectors (v_1, v_2, ..., v_m) so that:

$$\lambda_1 > \lambda_2 > \ldots > \lambda_m$$

Moreover, let's suppose that λ_1 is dominant over all the other eigenvalues (it's enough that $\lambda_1 > \lambda_i$ with $i \neq 1$). As the eigenvectors are orthogonal, they constitute a basis and it's possible to express the vector w, with a linear combination of the eigenvectors:

$$\bar{w} = u_1\bar{v}_1 + u_2\bar{v}_2 + \ldots + u_m\bar{v}_m = V\bar{u}$$

The vector u contains the coordinates in the new basis. Let's now consider the modification to the covariance rule:

$$\Delta\bar{w} = \eta\Sigma\bar{w} = \eta V\Omega V^{-1}V\bar{u} = \eta\Sigma\bar{w} = \eta V\Omega\bar{u}$$

If we apply the rule iteratively, we get a matrix polynomial:

$$\begin{gathered}\bar{w}^{(0)} \\ \bar{w}^{(1)} = \bar{w}^{(0)} + \eta\Sigma\bar{w}^{(0)} \\ \bar{w}^{(2)} = \bar{w}^{(1)} + \eta\Sigma\bar{w}^{(1)} = \bar{w}^{(0)} + 2\eta\Sigma\bar{w}^{(0)} + \eta^2\Sigma^2\bar{w}^{(0)} \\ \bar{w}^{(3)} = \bar{w}^{(2)} + \eta\Sigma\bar{w}^{(2)} = \bar{w}^{(0)} + 3\eta\Sigma\bar{w}^{(0)} + 3\eta^2\Sigma^2\bar{w}^{(0)} + \eta^3\Sigma^3\bar{w}^{(0)} \\ \cdots\end{gathered}$$

Exploiting the Binomial theorem and considering that $\Sigma^0=I$, we can get a general expression for $w^{(k)}$ as a function of $w^{(0)}$:

$$\bar{w}^{(k)} = \sum_{i=0}^{k}\binom{k}{i}\eta^i\Sigma^i\bar{w}^{(0)}$$

Let's now rewrite the previous formula using the change of basis:

$$\bar{w}^{(k)} = \sum_{i=0}^{k}\binom{k}{i}\eta^i\Sigma^i\bar{w}^{(0)} = \sum_{i=0}^{k}\binom{k}{i}\eta^i V\Omega^i V^{-1}\bar{w}^{(0)} = \sum_{i=0}^{k}\binom{k}{i}\eta^i V\Omega^i\bar{u}^{(0)}$$

The vector $u^{(0)}$ contains the coordinates of $w^{(0)}$ in the new basis; hence, $w^{(k)}$ is expressed as a polynomial where the generic term is proportional to $V\Omega^i u^{(0)}$.

Let's now consider the diagonal matrix Ω^k:

$$\Omega^k = \begin{pmatrix}\lambda_1^k & 0 & 0 \\ \vdots & \ddots & \vdots \\ 0 & 0 & \lambda_m^k\end{pmatrix} \approx \begin{pmatrix}\lambda_1^k & 0 & 0 \\ \vdots & \ddots & \vdots \\ 0 & 0 & 0\end{pmatrix}$$

The last step derives from the hypothesis that λ_1 is greater than any other eigenvalue and when $k \to \infty$, all ${\lambda_{i\neq 1}}^k \ll {\lambda_1}^k$. Of course, if $\lambda_{i\neq 1} > 1$, ${\lambda_{i\neq 1}}^k$ will grow as well as ${\lambda_1}^k$ however, the contribution of the *secondary* eigenvalues to $w^{(k)}$ becomes significantly weaker when $k \to \infty$. Just to understand the validity of this approximation, let's consider the following situation where λ_1 is slightly larger that λ_2:

$$\Omega = \begin{pmatrix} 1.1 & 0 \\ 0 & 1.05 \end{pmatrix} \Rightarrow \Omega^{1000} \approx \begin{pmatrix} 2.5 \cdot 10^{41} & 0 \\ 0 & 1.5 \cdot 10^{21} \end{pmatrix}$$

The result shows a very important property: not only is the approximation correct, but as we're going to show, if an eigenvalue λ_i is larger than all the other ones, the covariance rule will always converge to the corresponding eigenvector v_i. No other stable fixed points exist!

This hypothesis is no more valid if $\lambda_1 = \lambda_2 = ... = \lambda_n$. In this case, the total variance is explained equally by the direction of each eigenvector (a condition that implies a symmetry which isn't very common in real-life scenarios). This situation can also happen when working with finite-precision arithmetic, but in general, if the difference between the largest eigenvalue and the second one is less than the maximum achievable precision (for example, 32-bit floating point), it's plausible to accept the equality.

Of course, we assume that the dataset is not whitened, because our goal (also in the next paragraphs) is to reduce the original dimensionality considering only a subset of components with the highest total variability (the decorrelation, like in **Principal Component Analysis** (**PCA**), must be an outcome of the algorithm, not a precondition). On the other side, zero-centering the dataset could be useful, even if not really necessary for this kind of algorithm.

If we rewrite the expression for w_k considering this approximation, we obtain the following:

$$\bar{w}^{(k)} = \sum_{i=0}^{k} \binom{k}{i} \eta^i V \Omega^i \bar{u}^{(0)} \approx \sum_{i=0}^{k} \binom{k}{i} \eta^i \left((\bar{v}_1\ \bar{v}_2\ \dots\ \bar{v}_m) \begin{pmatrix} \lambda_1^k & 0 & 0 \\ \vdots & \ddots & \vdots \\ 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} u_1^{(0)} \\ \vdots \\ u_m^{(0)} \end{pmatrix} \right) =$$

$$= \left(\sum_{i=0}^{k} \binom{k}{i} \eta^i \lambda_1^i u_1^{(0)} \right) \bar{v}_1$$

As $a_1 v + a_2 v + ... + a_k v \propto v$, this result shows that, when $k \rightarrow \infty$, w_k will become proportional to the first eigenvector of the covariance matrix Σ (if $u_1^{(0)}$ is not null) and its magnitude, without normalization, will grow indefinitely. The spurious effect due to the other eigenvalues becomes negligible (above all, if w is divided by its norm, so that the length is always $||w|| = 1$) after a limited number of iterations.

However, before drawing our conclusions, an important condition must be added:

$$\bar{w}^{(0)} \cdot \bar{v}_1 \neq 0$$

In fact, if *w(0)* were orthogonal to *v1*, we would get (the eigenvectors are orthogonal to each other):

$$\bar{w}^{(0)} \cdot \bar{v}_1 = u_1^{(0)} \bar{v}_1 \cdot \bar{v}_1 + u_2^{(0)} \bar{v}_2 \cdot \bar{v}_1 + \ldots + u_m^{(0)} \bar{v}_m \cdot \bar{v}_1 =$$
$$= u_1^{(0)} \bar{v}_1 \cdot \bar{v}_1 =$$
$$= u_1^{(0)} |\bar{v}_1|^2 = 0 \Rightarrow u_1^{(0)} = 0$$

This important result shows how a Hebbian neuron working with the covariance rule is able to perform a PCA limited to the first component without the need for eigendecomposing Σ. In fact, the vector w (we're not considering the problem of the increasing magnitude, which can be easily managed) will rapidly converge to the orientation where the input dataset X has the highest variance. In `Chapter 5`, *EM Algorithm and Applications,* we discussed the details of PCA; in the next paragraph, we're going to discuss a couple of methods to find the first N principal components using a variant of the Hebb's rule.

Example of covariance rule application

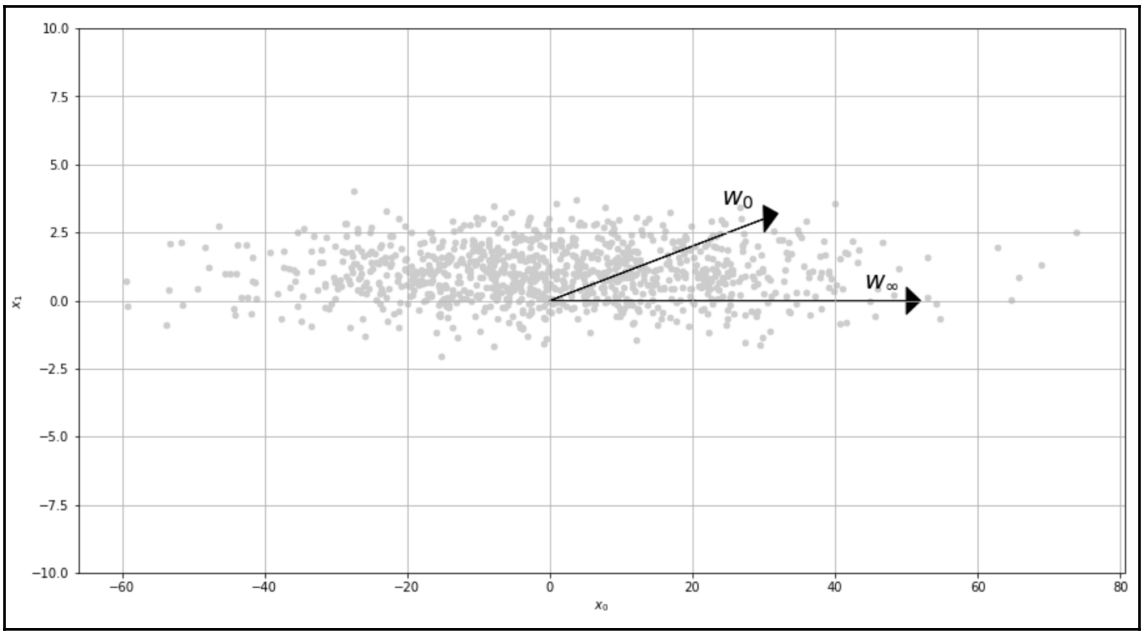
Before moving on, let's simulate this behavior with a simple Python example. We first generate `1000` values sampled from a bivariate Gaussian distribution (the variance is voluntarily asymmetric) and then we apply the covariance rule to find the first principal component ($w^{(0)}$ has been chosen so not to be orthogonal to v_1):

```
import numpy as np

rs = np.random.RandomState(1000)
X = rs.normal(loc=1.0, scale=(20.0, 1.0), size=(1000, 2))

w = np.array([30.0, 3.0])

S = np.cov(X.T)

for i in range(10):
    w += np.dot(S, w)
    w /= np.linalg.norm(w)
w *= 50.0
```

```
print(np.round(w, 1))
[ 50.  -0.]
```

The algorithm is straightforward, but there are a couple of elements that we need to comment on. The first one is the normalization of vector w at the end of each iteration. This is one of the techniques needed to avoid the uncontrolled growth of w. The second *tricky* element is the final multiplication, $w \bullet 50$. As we are multiplying by a positive scalar, the direction of w is not impacted, but it's easier to show the vector in the complete plot.

The result is shown in the following diagram:



Application of the covariance rule. w_∞ becomes proportional to the first principal component

After a limited number of iterations, w_∞ has the same orientation of the principal eigenvector which is, in this case, parallel to the x axes. The sense depends on the initial value w_0; however, in a PCA, this isn't an important element.

Weight vector stabilization and Oja's rule

The easiest way to stabilize the weight vector is normalizing it after each update. In this way, its length will be always kept equal to one. In fact, in this kind of neural networks we are not interested in the magnitude, but only in the direction (that remains unchanged after the normalization). However, there are two main reasons that discourage this approach:

- It's non-local. To normalize vector w, we need to know all its values and this isn't biologically plausible. A real synaptic weight model should be self-limiting, without the need to have access to external pieces of information that cannot be available.
- The normalization must be performed after having applied the correction and hence needs a double iterative step.

In many machine learning contexts, these conditions are not limiting and they can be freely adopted, but when it's necessary to work with neuroscientific models, it's better to look for other solutions. In a discrete form, we need to determine a correction term for the standard Hebb's rule:

$$\bar{w}_{k+1} - \bar{w}_k = \eta y \bar{x} - f(\bar{w}_k, y_k, \bar{x})$$

The *f* function can work both as a local and non-local normalizer. An example of the first type is **Oja's rule**:

$$\Delta \bar{w} = \eta y_k \bar{x}_k - \alpha y_k^2 \bar{w}_k$$

The α parameter is a positive number that controls the strength of the normalization. A non-rigorous proof of the stability of this rule can be obtained considering the condition:

$$\Delta \bar{w} \cdot \bar{w} \rightarrow 0 \quad \Rightarrow \quad y_k(\bar{x} \cdot \bar{w}) - \alpha y_k^2 (\bar{w} \cdot \bar{w}) \rightarrow 0$$

The second expression implies that:

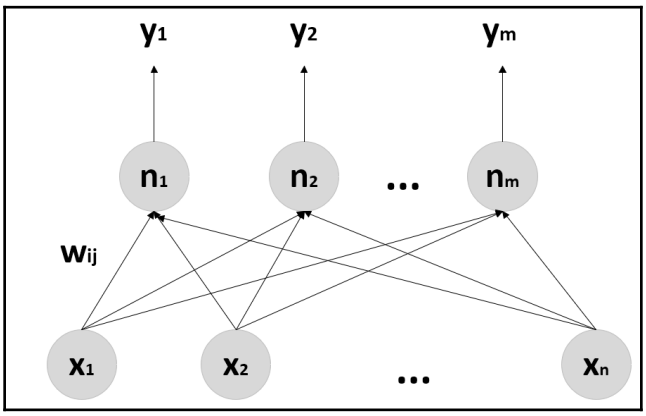
$$y_k^2 (1 - \alpha |\bar{w}|^2) \rightarrow 0 \quad \Rightarrow \quad |\bar{w}|^2 \rightarrow \frac{1}{\alpha}$$

Therefore, when $t \to \infty$, the magnitude of the weight correction becomes close to zero and the length of the weight vector w will approach a finite limit value:

$$\lim_{k \to \infty} |\bar{w}_k| = \frac{1}{\sqrt{\alpha}}$$

Sanger's network

A **Sanger's network** is a neural network model for online *Principal Component* extraction proposed by T. D. Sanger in *Optimal Unsupervised Learning in a Single-Layer Linear Feedforward Neural Network, Sanger T. D., Neural Networks, 1989/2*. The author started with the standard version of Hebb's rule and modified it to be able to extract a variable number of principal components (v_1, v_2, ..., v_m) in descending order ($\lambda_1 > \lambda_2 > ... > \lambda_m$). The resulting approach, which is a natural extension of Oja's rule, has been called the **Generalized Hebbian Rule** (**GHA**) (or Learning). The structure of the network is represented in the following diagram:

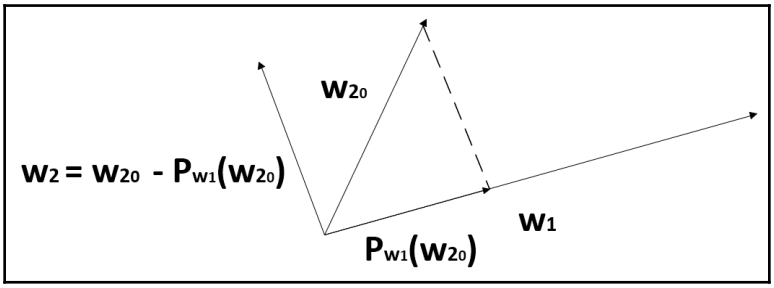


The network is fed with samples extracted from an n-dimensional dataset:

$$X = \{\bar{x}_1, \bar{x}_2, \ldots \bar{x}_N\} \;\; where \;\; \bar{x}_i \in \mathbb{R}^n$$

The *m* output neurons are connected to the input through a weight matrix, $W = \{w_{ij}\}$, where the first index refers to the input components (pre-synaptic units) and the second one to the neuron. The output of the network can be easily computed with a scalar product; however, in this case, we are not interested in it, because just like for the covariance (and Oja's) rules, the principal components are extracted through the weight updates.

The problem that arose after the formulation of Oja's rule was about the extraction of multiple components. In fact, if we applied the original rule to the previous network, all weight vectors (the rows of *w*) would converge to the first principal component. The main idea (based on the **Gram-Schmidt** orthonormalization method) to overcome this limitation is based on the observation that once we have extracted the first component w_1, the second one w_2 can be forced to be orthogonal to w_1, the third one w_3 can be forced to be orthogonal to w_1 and w_2, and so on. Consider the following representation:



Orthogonalization of two weight vectors

In this case, we are assuming that w_1 is stable and w_{20} is another weight vector that is converging to w_1. The projection of w_{20} onto w_1 is as follows:

$$P_{\bar{w}_1}(\bar{w}_{2_0}) = (\bar{w}_1^T \bar{w}_{2_0}) \frac{\bar{w}_1}{\|\bar{w}_1\|}$$

In the previous formula, we can omit the norm if we don't need to normalize (in the network, this process is done after a complete weight update). The orthogonal component of w_{20} is simply obtained with a difference:

$$\bar{w}_2 = \bar{w}_{2_0} - P_{\bar{w}_1}(\bar{w}_{2_0})$$

Applying this method to the original Oja's rule, we obtain a new expression for the weight update (called Sanger's rule):

$$\Delta w_{ij} = \eta \left(y_i x_j - y_i \sum_{k=1}^{i} w_{kj} y_k \right)$$

The rule is referred to a single input vector *x*, hence x_j is the j^{th} component of *x*. The first term is the classic Hebb's rule, which forces weight *w* to become parallel to the first principal component, while the second one acts in a way similar to the Gram-Schmidt orthogonalization, by subtracting a term proportional to the projection of *w* onto all the weights connected to the previous post-synaptic units and considering, at the same time, the normalization constraint provided by Oja's rule (which is proportional to the square of the output).

In fact, expanding the last term, we get the following:

$$y_i \sum_{k=1}^{i} w_{kj} y_k = y_i \sum_{k=1}^{i} w_{kj} (\bar{w}_k^T \bar{x}) = (\bar{w}_i^T \bar{x}) \left[w_{1j} (\bar{w}_1^T \bar{x}) + w_{2j} (\bar{w}_2^T \bar{x}) + \ldots + w_{ij} (\bar{w}_i^T \bar{x}) \right]$$

The term subtracted to each component w_{ij} is proportional to all the components where the index *j* is fixed and the first index is equal to *1, 2, ..., i*. This procedure doesn't produce an immediate orthogonalization but requires several iterations to converge. The proof is non-trivial, involving convex optimization and dynamic systems methods, but, it can be found in the aforementioned paper. Sanger showed that the algorithm converges always to the sorted first *n* principal components (from the largest eigenvalue to the smallest one) if the `learning_rate` $\eta(t)$ decreases monotonically and converges to zero when $t \to \infty$. Even if necessary for the formal proof, this condition can be relaxed (a stable $\eta < 1$ is normally sufficient). In our implementation, matrix *W* is normalized after each iteration, so that, at the end of the process, W^T (the weights are in the rows) is orthonormal and constitutes a basis for the eigenvector subspace.

In matrix form, the rule becomes as follows:

$$\Delta W = \eta \left(\bar{y} \bar{x}^T - Tril(\bar{y} \bar{y}^T) W \right)$$

Tril(•) is a matrix function that transforms its argument into a lower-triangular matrix and the term yy^T is equal to Wxx^TW.

The algorithm for a Sanger's network is as follows:

1. Initialize $W^{(0)}$ with random values. If the input dimensionality is n and m principal components must be extracted, the shape will be $(m \times n)$.
2. Set a `learning_rate` η (for example, `0.01`).
3. Set a `threshold` Thr (for example, `0.001`).
4. Set a counter $T = 0$.
5. While $||W^{(t)} - W^{(t-1)}||_F > Thr$:
    1. Set $\Delta W = 0$ (same shape of W)
    2. For each x in X:
        1. Set $T = T + 1$
        2. Compute $y = W^{(t)}x$
        3. Compute and accumulate $\Delta W \mathrel{+}= \eta(yx^T - Tril(yy^T)W^{(t)}$
    3. Update $W^{(t+1)} = W^{(t)} + (\eta / T)\Delta W$
    4. Set $W^{(t+1)} = W^{(t+1)} / ||W^{(t+1)}||^{(rows)}$ (the norm must be computed row-wise)

The algorithm can also be iterated a fixed number of times (like in our example), or the two stopping approaches can be used together.

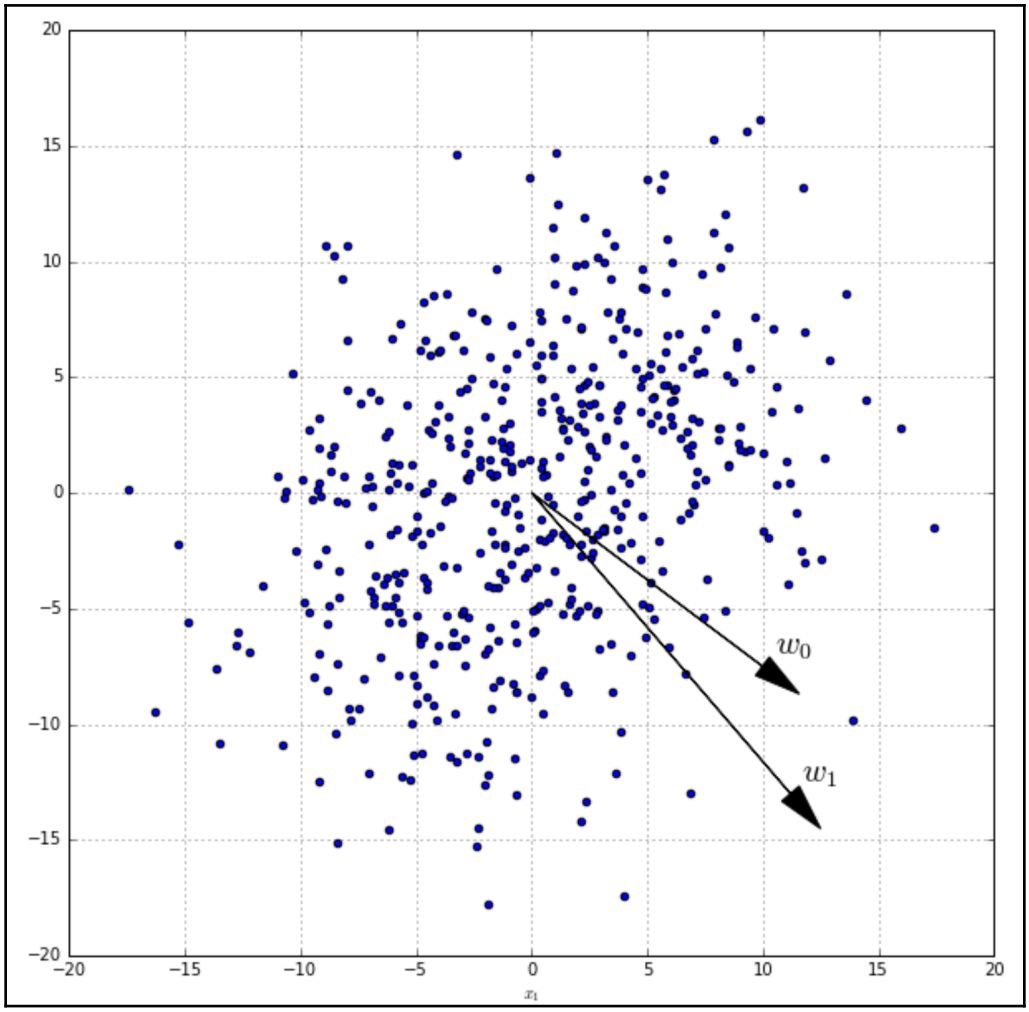
Example of Sanger's network

For this Python example, we consider a bidimensional zero-centered dataset `X` with 500 samples (we are using the function defined in the first chapter). After the initialization of `X`, we also compute the eigendecomposition, to be able to double-check the result:

```
import numpy as np

from sklearn.datasets import make_blobs

X, _ = make_blobs(n_samples=500, centers=2, cluster_std=5.0,
random_state=1000)
Xs = zero_center(X)

Q = np.cov(Xs.T)
eigu, eigv = np.linalg.eig(Q)

print(eigu)
[ 24.5106037   48.99234467]

print(eigv)
```

```
[[-0.75750566 -0.6528286 ]
 [ 0.6528286  -0.75750566]]

n_components = 2

W_sanger = np.random.normal(scale=0.5, size=(n_components, Xs.shape[1]))
W_sanger /= np.linalg.norm(W_sanger, axis=1).reshape((n_components, 1))
```

The eigenvalues are in reverse order; therefore, we expect to have a final *W* with the rows swapped. The initial condition (with the weights multiplied by 15) is shown in the following diagram:



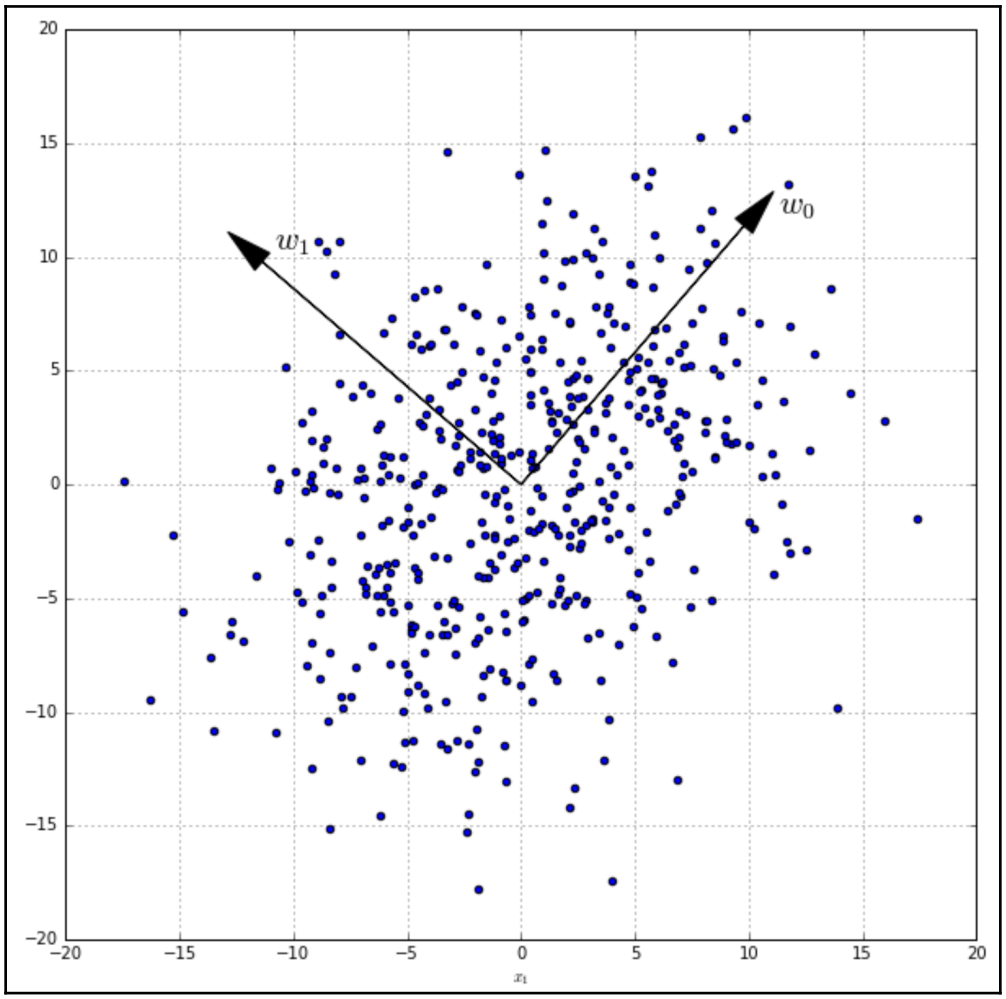
Dataset with *W* initial condition, we can implement the algorithm. For simplicity, we preferred a fixed number of iterations (`5000`) with a `learning_rate` of $\eta=0.01$. The reader can modify the snippet to stop when the weight matrix becomes stable:

```
learning_rate = 0.01
nb_iterations = 5000
t = 0.0

for i in range(nb_iterations):
    dw = np.zeros((n_components, Xs.shape[1]))
    t += 1.0
    for j in range(Xs.shape[0]):
        Ysj = np.dot(W_sanger, Xs[j]).reshape((n_components, 1))
        QYd = np.tril(np.dot(Ysj, Ysj.T))
        dw += np.dot(Ysj, Xs[j].reshape((1, X.shape[1]))) - np.dot(QYd,
W_sanger)
    W_sanger += (learning_rate / t) * dw
    W_sanger /= np.linalg.norm(W_sanger, axis=1).reshape((n_components, 1))
```

The first thing to check is the final state of *W* (we transposed the matrix to be able to compare the columns):

```
print(W_sanger.T)
[[-0.6528286  -0.75750566]
 [-0.75750566  0.6528286 ]]
```

As expected, *W* has converged to the eigenvectors of the input correlation matrix (the sign – which is associated with the sense of *w*—is not important because we care only about the orientation). The second eigenvalue is the highest, so the columns are swapped. Replotting the diagram, we get the following:



Final condition, w has converged to the two principal components

The two components are perfectly orthogonal (the final orientations can change according to the initial conditions or the random state) and w_0 points in the direction of the first principal component, while w_1 points in the direction of the second component. Considering this nice property, it's not necessary to check the magnitude of the eigenvalues; therefore, this algorithm can operate without eigendecomposing the input covariance matrix. Even if a formal proof is needed to explain this behavior, it's possible to understand it intuitively. Every single neuron converges to the first principal component given a full eigenvector subspace. This property is always maintained, but after the orthogonalization, the subspace is implicitly reduced by a dimension. The second neuron will always converge to the first component, which now corresponds to the global second component, and so on.

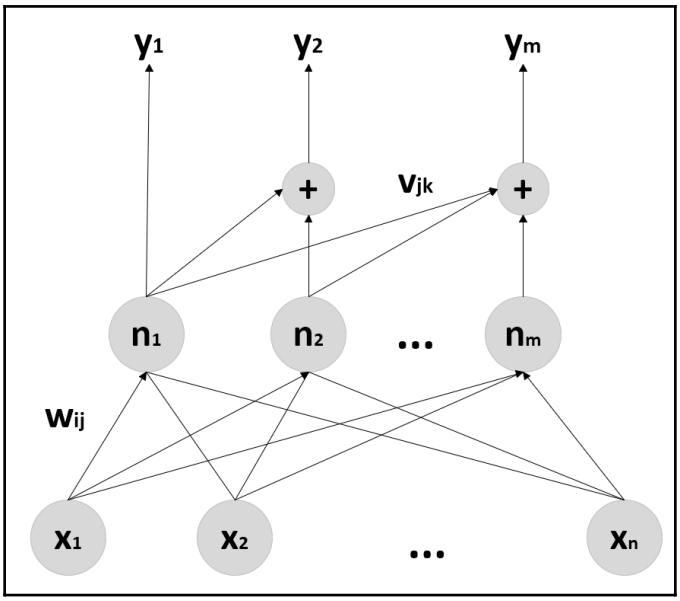
One of the advantages of this algorithm (and also of the next one) is that a standard PCA is normally a bulk process (even if there are batch algorithms), while a Sanger's network is an online algorithm that is trained incrementally. In general, the time performance of a Sanger's network is worse than the direct approach because of the iterations (some optimizations can be achieved using more vectorization or GPU support). On the other side, a Sanger's network is memory-saving when the number of components is less than the input dimensionality (for example, the covariance matrix for *n=1000* has 10^6 elements, if $m = 100$, the weight matrix has 10^4 elements).

Rubner-Tavan's network

In `Chapter 5`, *EM Algorithm and Applications*, we said that any algorithm that decorrelates the input covariance matrix is performing a PCA without dimensionality reduction. Starting from this approach, Rubner, and Tavan (in the paper *A Self-Organizing Network for Principal-Components Analysis*, *Rubner J., Tavan P., Europhysics. Letters, 10(7), 1989*) proposed a neural model whose goal is decorrelating the output components to force the consequent decorrelation of the output covariance matrix (in lower-dimensional subspace). Assuming a zero-centered dataset and $E[y] = 0$, the output covariance matrix for m principal components is as follows:

$$Q = \begin{pmatrix} \frac{1}{N}\sum_i y_1^i y_1^i & \cdots & \frac{1}{N}\sum_i y_1^i y_m^i \\ \vdots & \ddots & \vdots \\ \frac{1}{N}\sum_i y_m^i y_1^i & \cdots & \frac{1}{N}\sum_i y_m^i y_m^i \end{pmatrix}$$

Hence, it's possible to achieve an approximate decorrelation, forcing the terms $y_i y_j$ with $i \neq j$ to become close to zero. The main difference with a standard approach (such as whitening or vanilla PCA) is that this procedure is local, while all the standard methods operate globally, directly with the covariance matrix. The neural model proposed by the authors is shown in the following diagram (the original model was proposed for binary units, but it works quite well also for linear ones):



Rubner-Tavan network. The connections v_{jk} are based on the anti-Hebbian rule

The network has *m* output units and the last *m-1* neurons have a summing node that receives the weighted output of the previous units (hierarchical lateral connections). The dynamic is simple: the first output isn't modified. The second one is forced to become decorrelated with the first one. The third one is forced to become decorrelated with both the first and the second one and so on. This procedure must be iterated a number of times because the inputs are presented one by one and the cumulative term that appears in the correlation/covariance matrix (it's always easier to zero-center the dataset and work with the correlation matrix) must be implicitly split into its addends. It's not difficult to understand that the convergence to the only stable fixed point (which has been proven to exist by the authors) needs some iterations to correct the wrong output estimations.

The output of the network is made up of two contributions:

$$\bar{y}^{(i)} = \sum_{j=1}^{m} w_{ij}\bar{x}^{(j)} + \sum_{k=1}^{i-1} v_{jk}\bar{y}^{(k)}$$

The notation $y/x^{(i)}$ indicates the i^{th} element of y/x. The first term produces a partial output based only on the input, while the second one uses hierarchical lateral connections to correct the values and enforce the decorrelation. The internal weights w_{ij} are updated using the standard version of Oja's rule (this is mainly responsible for the convergence of each weight vector to the first principal component):

$$\Delta w_{ij} = \eta y_i (x_j - w_{ij} y_i)$$

Instead, the external weights v_{jk} are updated using an anti-Hebbian rule:

$$\Delta v_{jk} = -\eta y_j (y_k + v_{jk} y_j) \;\; valid\ only\ for\ i \neq k$$

The previous formula can be split into two parts: the first term $-\eta y_j y_k$ acts in the opposite direction of a standard version of Hebb's rule (that's why it's called anti-Hebbian) and forces the decorrelation. The second one $-\eta y_j y_k v_{jk}$ acts as a regularizer and it's analogous to Oja's rule. The term $-\eta y_j y_k$ works as a feedback signal for the Oja's rule that readapts the updates according to the new magnitude of the actual output. In fact, after modifying the lateral connections, the outputs are also forced to change and this modification impacts on the update of w_{ij}. When all the outputs are decorrelated, the vectors w_i are implicitly obliged to be orthogonal. It's possible to imagine an analogy with the Gram-Schmidt orthogonalization, even if in this case the relation between the extraction of different components and the decorrelation is more complex. Like for Sanger's network, this model extracts the first m principal components in descending order (the reason is the same that has been intuitively explained), but for a complete (non-trivial) mathematical proof, please refer to the aforementioned paper.

If input dimensionality is n and the number of components is equal to m, it's possible to use a lower-triangular matrix $V\ (m \times m)$ with all diagonal elements set to 0 and a standard matrix for $W\ (n \times m)$.

The structure of W is as follows:

$$W = (\bar{w}_1 \;\; \bar{w}_2 \;\; \dots \;\; \bar{w}_m)$$

Therefore, w_i is a column-vector that must converge to the corresponding eigenvector. The structure of V is instead:

$$V = Tril_{(if\ i=j\ V[i,j]=0)} \begin{pmatrix} \bar{v}_1 \\ \vdots \\ \bar{v}_m \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 & 0 \\ v_{21} & 0 & 0 & 0 \\ v_{31} & v_{32} & 0 & 0 \\ v_{41} & v_{42} & v_{43} & 0 \end{pmatrix}$$

Using this notation, the output becomes as follows:

$$y^{(t+1)} = W^T \bar{x} + V \bar{y}^{(t)}$$

As the output is based on recurrent lateral connections, its value must be stabilized by iterating the previous formula for a fixed number times or until the norm between two consecutive values becomes smaller than a predefined threshold. In our example, we use a fixed number of iterations equal to five. The update rules cannot be written directly in matrix notation, but it's possible to use the vectors w_i(columns) and v_j(rows):

$$\begin{cases} \Delta \bar{w}_i = \eta \bar{y}^{(i)} (\bar{x} - \bar{y}^{(i)} \bar{w}_i) \\ \Delta \bar{v}_i = -\eta \bar{y}^{(i)} (\bar{y} + \bar{y}^{(i)} \bar{v}_i) \end{cases}$$

In this case, $y^{(i)}$ means the i^{th} component of y. The two matrices must be populated with a loop.

The complete Rubner-Tavan's network algorithm is (the dimensionality of x is n, the number of components is denoted with m):

1. Initialize $W^{(0)}$ randomly. The shape is $(n \times m)$.
2. Initialize $V^{(0)}$ randomly. The shape is $(m \times m)$.
3. Set $V^{(0)} = Tril(V^{(0)})$. $Tril(\bullet)$ transforms the input argument in a lower-triangular matrix.
4. Set all diagonal components of $V^{(0)}$ equal to 0.
5. Set the `learning_rate` η (for example, `0.001`).
6. Set a `threshold` Thr (for example, `0.0001`).
7. Set a cycle counter $T=0$.

8. Set a maximum number of iterations `max_iterations` (for example, 1000).
9. Set a number of `stabilization_cycles` (for example, `5`):
    1. While $||W^{(t)} - W^{(t-1)}||_F > Thr$ and T < `max_iterations`:
        1. Set $T = T + 1$.
        2. For each x in X:
            1. Set y_{prev} to zero. The shape is $(m, 1)$.
            2. For $i=1$ to `stabilization_cycles`:
                1. $y = W^T x + V y_{prev}$.
                2. $y_{prev} = y$.
            3. Compute the updates for W and V:
                1. Create two empty matrices $\Delta W\ (n \times m)$ and $\Delta V\ (m \times m)$
                2. for $t=1$ to m:
                    1. $\Delta w_t = \eta y^{(t)}(x - y^{(t)} w_t)$
                    2. $\Delta v_t = -\eta y^{(t)}(y + y^{(t)} v_t)$
                3. Update W and V:
                    1. $W^{(t+1)} = W^{(t)} + \Delta W$
                    2. $V^{(t+1)} = V^{(t)} + \Delta V$
                4. Set $V = Tril(V)$ and set all the diagonal elements to 0
                5. Set $W^{(t+1)} = W^{(t+1)} / ||W^{(t+1)}||^{(columns)}$ (The norm must be computed column-wise)

In this case, we have adopted both a threshold and a maximum number of iterations because this algorithms normally converges very quickly. Moreover, I suggest the reader always checks the shapes of vectors and matrices when performing dot products.

In this example, as well as in all the other ones, the NumPy random seed is set equal to `1000` (`np.random.seed(1000)`). Using different values (or repeating more times the experiments without resetting the seed) can lead to slightly different results (which are always coherent).

Example of Rubner-Tavan's network

For our Python example, we are going to use the same dataset already created for the Sanger's network (which is expected to be available in the variable `Xs`). Therefore, we can start setting up all the constants and variables:

```
import numpy as np

n_components = 2
learning_rate = 0.0001
max_iterations = 1000
stabilization_cycles = 5
threshold = 0.00001

W = np.random.normal(0.0, 0.5, size=(Xs.shape[1], n_components))
V = np.tril(np.random.normal(0.0, 0.01, size=(n_components, n_components)))
np.fill_diagonal(V, 0.0)

prev_W = np.zeros((Xs.shape[1], n_components))
t = 0
```

At this point, it's possible to implement the training loop:

```
while(np.linalg.norm(W - prev_W, ord='fro') > threshold and t <
max_iterations):
    prev_W = W.copy()
    t += 1
    for i in range(Xs.shape[0]):
        y_p = np.zeros((n_components, 1))
        xi = np.expand_dims(Xs[i], 1)
        y = None

        for _ in range(stabilization_cycles):
            y = np.dot(W.T, xi) + np.dot(V, y_p)
            y_p = y.copy()
        dW = np.zeros((Xs.shape[1], n_components))
        dV = np.zeros((n_components, n_components))
        for t in range(n_components):
            y2 = np.power(y[t], 2)
            dW[:, t] = np.squeeze((y[t] * xi) + (y2 * np.expand_dims(W[:,
t], 1)))
            dV[t, :] = -np.squeeze((y[t] * y) + (y2 * np.expand_dims(V[t,
:], 1)))

        W += (learning_rate * dW)
        V += (learning_rate * dV)
        V = np.tril(V)
```

```
        np.fill_diagonal(V, 0.0)
        W /= np.linalg.norm(W, axis=0).reshape((1, n_components))
```

The final `W` and the output covariance matrix are as follows:

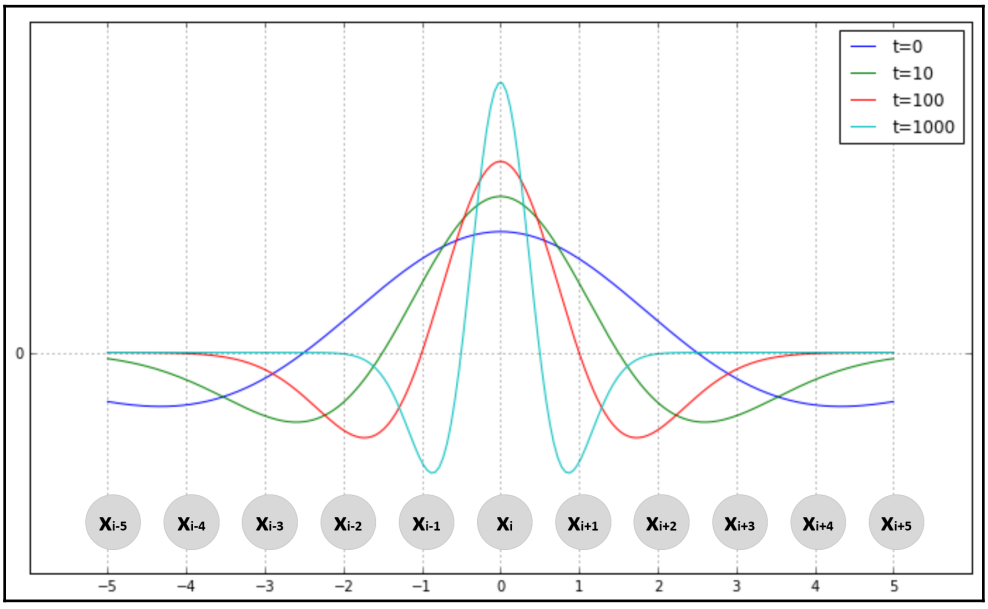
```
print(W)
[[-0.65992841  0.75897537]
 [-0.75132849 -0.65111933]]

Y_comp = np.zeros((Xs.shape[0], n_components))

for i in range(Xs.shape[0]):
        y_p = np.zeros((n_components, 1))
        xi = np.expand_dims(Xs[i], 1)

        for _ in range(stabilization_cycles):
            Y_comp[i] = np.squeeze(np.dot(W.T, xi) + np.dot(V.T, y_p))
            y_p = y.copy()

print(np.cov(Y_comp.T))
[[ 48.9901765   -0.34109965]
 [ -0.34109965  24.51072811]]
```

As expected, the algorithm has successfully converged to the eigenvectors (in descending order) and the output covariance matrix is almost completely decorrelated (the sign of the non-diagonal elements can be either positive or negative). Rubner-Tavan's networks are generally faster than Sanger's network, thanks to the feedback signal created by the anti-Hebbian rule; however, it's important to choose the right value for the learning rate. A possible strategy is to implement a temporal decay (as done in Sanger's network) starting with a value not greater than `0.0001`. However, it's important to reduce η when n increases (for example, $\eta = 0.0001 / n$), because the normalization strength of Oja's rule on the lateral connections v_{jk} is often not enough to avoid over and underflows when $n >> 1$. I don't suggest any extra normalization on V (which must be carefully analyzed considering that V is singular) because it can slow down the process and reduce the final accuracy.

Self-organizing maps

Self-organizing maps (SOMs) have been proposed by Willshaw and Von Der Malsburg (*Willshaw D. J., Von Der Malsburg C., How patterned neural connections can be set up by self-organization, Proceedings of the Royal Society of London, B/194, N. 1117*) to model different neurobiological phenomena observed in animals. In particular, they discovered that some areas of the brain develop structures with different areas, each of them with a high sensitivity for a specific input pattern. The process behind such a behavior is quite different from what we have discussed up until now, because it's based on competition among neural units based on a principle called **winner-takes-all**. During the training period, all the units are excited with the same signal, but only one will produce the highest response. This unit is automatically candidate to become the receptive basin for that specific pattern. The particular model we are going to present has been introduced by **Kohonen** (in the paper *Kohonen T., Self-organized formation of topologically correct feature maps, Biological Cybernetics, 43/1*) and it's named after him.

The main idea is to implement a gradual winner-takes-all paradigm, to avoid the premature convergence of a neuron (as a definitive winner) and increment the level of plasticity of the network. This concept is expressed graphically in the following graph (where we are considering a linear sequence of neurons):



Mexican-hat dynamic implemented by a Kohonen network

In this case, the same pattern is presented to all the neurons. At the beginning of the training process (**t=0**), a positive response is observed in $\mathbf{x}_{i-2}$ to $\mathbf{x}_{i+2}$ with a peak in $\mathbf{x}_i$. The potential winner is obviously $\mathbf{x}_i$, but all these units are potentiated according to their distance from $\mathbf{x}_i$. In other words, the network (which is trained sequentially) is still receptive to change if other patterns produce a stronger activation. If instead $\mathbf{x}_i$ keeps on being the winner, the radius is slightly reduced, until the only potentiated unit will be $\mathbf{x}_i$. Considering the shape of this function, this dynamic is often called *Mexican Hat*. With this approach, the network remains plastic until all the patterns have been repeatedly presented. If, for example, another pattern elicits a stronger response in $\mathbf{x}_i$, it's important that its activation is still not too high, to allow a fast reconfiguration of the network. At the same time, the new winner will probably be a neighbor of $\mathbf{x}_i$, which has received a partial potentiation and can easily take the place of $\mathbf{x}_i$.

A **Kohonen SOM** (also known as Kohonen network or simply Kohonen map) is normally represented as a bidimensional map (for example, a square matrix $m \times m$, or any other rectangular shape), but 3D surfaces, such as spheres or toruses are also possible (the only necessary condition is the existence of a suitable metric). In our case, we always refer to a square matrix where each cell is a receptive neuron characterized by a synaptic weight w with the dimensionality of the input patterns:

$$X = \{\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_N\} \;\; where \;\; \bar{x}_i \in \mathbb{R}^n$$

During both training and working phases, the winning unit is determined according to a similarity measure between a sample and each weight vector. The most common metric is the Euclidean; hence, if we consider a bidimensional map W with a shape $(k \times p)$ so that $W \in \Re^{k \times p \times n}$, the winning unit (in terms of its coordinates) is computed as follows:

$$u^* = argmin_{k,p} \|W[k,p] - \bar{x}\|_2$$

As explained before, it's important to avoid the premature convergence because the complete final configuration could be quite different from the initial one. Therefore, the training process is normally subdivided into two different stages. During the first one, whose duration is normally about 10-20% of the total number of iterations (let's call this value t_{max}), the correction is applied to the winning unit and its neighbors (computed by adopting a decaying radius). Instead, during the second one, the radius is set to 1.0 and the correction is applied only to the winning unit. In this way, it's possible to analyze a larger number of possible configurations, automatically selecting the one associated with the least error. The neighborhood can have different shapes; it can be square (in closed 3D maps, the boundaries don't exist anymore), or, more easily, it's possible to employ a radial basis function based on an exponentially decaying distance-weight:

$$n(i,j) = e^{-\frac{\|u^* - (i,j)\|^2}{2\sigma(t)^2}} \quad where \quad \sigma(t) = \sigma_0 e^{-\frac{t}{\tau}}$$

The relative weight of each neuron is determined by the $\sigma(t)$. σ_0 function is the initial radius and τ is a time-constant that must be considered as a hyperparameter which determines the slope of the decaying weight. Suitable values are 5-10% of the total number of iterations. Adopting a radial basis function, it's not necessary to compute an actual neighborhood because the multiplication factor $n(i, j)$ becomes close to zero outside of the boundaries. A drawback is related to the computational cost, which is higher than a square neighborhood (as the function must be computed for the whole map); however, it's possible to speed up the process by precomputing all the squared distances (the numerator) and exploiting the vectorization features offered by packages such as NumPy (a single exponential is computed every time).

The update rule is very simple and it's based on the idea to move the winning unit synaptic weights closer to the pattern, x_i, (repeated for the whole dataset, X):

$$\Delta \bar{w}_{ij} = \eta(t) n(i,j)(\bar{x}_i - \bar{w}_{ij})$$

The $\eta(t)$ function is the learning rate, which can be fixed, but it's preferable to start with a higher value, η_0 and let it decay to a target final value, η_∞:

$$\eta(t) = \begin{cases} \eta_0 e^{-\frac{t}{\tau}} & if \ t < t_{max} \\ \eta_\infty & if \ t \geqslant t_{max} \end{cases}$$

In this way, the initial changes force the weights to align with the input patterns, while all the subsequent updates allow slight modifications to improve the overall accuracy. Therefore, each update is proportional to the learning rate, the neighborhood weighted distance, and the difference between each pattern and the synaptic vector. Theoretically, if Δw_{ij} is equal to 0.0 for the winning unit, it means that a neuron has become the attractor of a specific input pattern, and its neighbors will be receptive to noisy/altered versions. The most interesting aspect is that the complete final map will contain the attractors for all patterns which are organized to maximize the similarity between adjacent units. In this way, when a new pattern is presented, the area of neurons that maps the most similar shapes will show a higher response. For example, if the patterns are made up of handwritten digits, attractors for the digit 1 and for digit 7 will be closer than the attractor, for example, for digit 8. A malformed 1 (which could be interpreted as 7) will elicit a response that is between the first two attractors, allowing us to assign a relative probability based on the distance. As we're going to see in the example, this feature yields to a smooth transition between different variants of the same pattern class avoiding rigid boundaries that oblige a binary decision (like in a K-means clustering or in a hard classifier).

The complete Kohonen SOM algorithm is as follows:

1. Randomly initialize $W^{(0)}$. The shape is $(k \times p \times n)$.
2. Initialize `nb_iterations`, the total number of iterations, and t_{max} (for example, `nb_iterations` = 1000 and t_{max} = 150).
3. Initialize τ (for example, τ = 100).
4. Initialize η_0 and η_∞ (for example, η_0 = 1.0 and η_∞ = 0.05).
5. For `t = 0` to `nb_iterations`:
    1. If $t < t_{max}$:
        1. Compute $\eta(t)$
        2. Compute $\sigma(t)$
    2. Otherwise:
        1. Set $\eta(t) = \eta_\infty$
        2. Set $\sigma(t) = \sigma_\infty$

3. For each x_i in X:
    1. Compute the winning unit u^* (let's assume that the coordinates are *i, j*)
    2. Compute *n(i, j)*
    3. Apply the weight correction $\Delta w_{ij}^{(t)}$ to all synaptic weights $W^{(t)}$
4. Renormalize $W^{(t)} = W^{(t)} / ||W^{(t)}||^{(columns)}$ (the norm must be computed column-wise)

Example of SOM

We can now implement an SOM using the Olivetti faces dataset. As the process can be very long, in this example we limit the number of input patterns to 100 (with a 5 × 5 matrix). The reader can try with the whole dataset and a larger map.

The first step is loading the data, normalizing it so that all values are bounded between 0.0 and 1.0, and setting the constants:

```
import numpy as np

from sklearn.datasets import fetch_olivetti_faces

faces = fetch_olivetti_faces(shuffle=True)

Xcomplete = faces['data'].astype(np.float64) / np.max(faces['data'])
np.random.shuffle(Xcomplete)

nb_iterations = 5000
nb_startup_iterations = 500
pattern_length = 64 * 64
pattern_width = pattern_height = 64
eta0 = 1.0
sigma0 = 3.0
tau = 100.0

X = Xcomplete[0:100]
matrix_side = 5
```

At this point, we can initialize the weight matrix using a normal distribution with a small standard deviation:

```
W = np.random.normal(0, 0.1, size=(matrix_side, matrix_side,
pattern_length))
```

Now, we need to define the functions to determine the winning unit based on the least distance:

```
def winning_unit(xt):
    distances = np.linalg.norm(W - xt, ord=2, axis=2)
    max_activation_unit = np.argmax(distances)
    return int(np.floor(max_activation_unit / matrix_side)),
max_activation_unit % matrix_side
```

It's also useful to define the functions $\eta(t)$ and $\sigma(t)$:

```
def eta(t):
    return eta0 * np.exp(-float(t) / tau)

def sigma(t):
    return float(sigma0) * np.exp(-float(t) / tau)
```

As explained before, instead of computing the radial basis function for each unit, it's preferable to use a precomputed distance matrix (in this case, $5 \times 5 \times 5 \times 5$) containing all the possible distances between couples of units. In this way, NumPy allows a faster calculation thanks to its vectorization features:

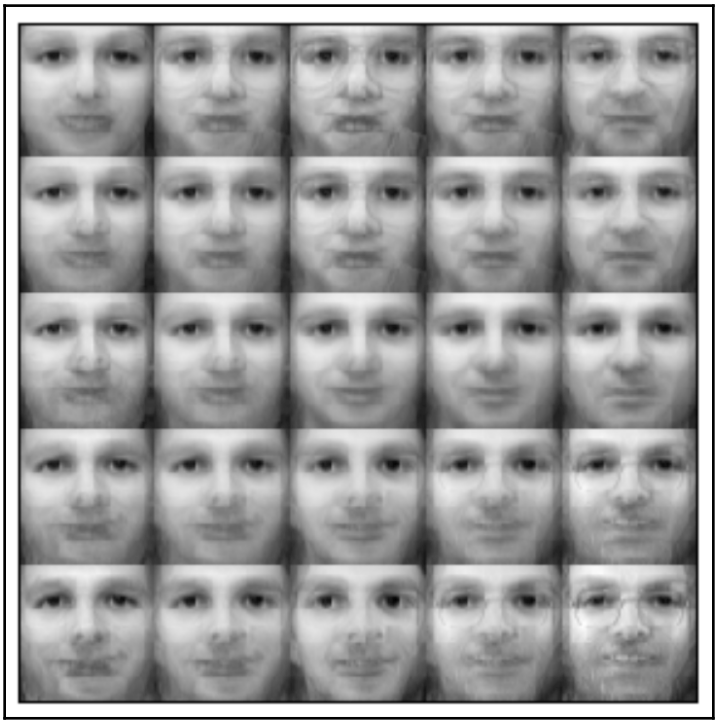
```
precomputed_distances = np.zeros((matrix_side, matrix_side, matrix_side,
matrix_side))

for i in range(matrix_side):
    for j in range(matrix_side):
        for k in range(matrix_side):
            for t in range(matrix_side):
                precomputed_distances[i, j, k, t] = \
                    np.power(float(i) - float(k), 2) + np.power(float(j) -
float(t), 2)

def distance_matrix(xt, yt, sigmat):
    dm = precomputed_distances[xt, yt, :, :]
    de = 2.0 * np.power(sigmat, 2)
    return np.exp(-dm / de)
```

The `distance_matrix` function returns the value of the radial basis function for the whole map given the center point (the winning unit) `xt, yt` and the current value of σ `sigmat`. Now, it's possible to start the training process (in order to avoid correlations, it's preferable to shuffle the input sequence at the beginning of each iteration):

```
sequence = np.arange(0, X.shape[0])
t = 0

for e in range(nb_iterations):
```

```
np.random.shuffle(sequence)
t += 1
if e < nb_startup_iterations:
    etat = eta(t)
    sigmat = sigma(t)
else:
    etat = 0.2
    sigmat = 1.0
for n in sequence:
    x_sample = X[n]
    xw, yw = winning_unit(x_sample)
    dm = distance_matrix(xw, yw, sigmat)
    dW = etat * np.expand_dims(dm, axis=2) * (x_sample - W)
    W += dW
W /= np.linalg.norm(W, axis=2).reshape((matrix_side, matrix_side, 1))
```

In this case, we have set η_∞ = `0.2` but I invite the reader to try different values and evaluate the final result. After training for `5000` epochs, we got the following weight matrix (each weight is plotted as a bidimensional array):



As it's possible to see, the weights have converged to faces with slightly different features. In particular, looking at the shapes of the faces and the expressions, it's easy to notice the transition between different attractors (some faces are smiling, while others are more serious; some have glasses, mustaches, and beards, and so on). It's also important to consider that the matrix is larger than the minimum capacity (there are ten different individuals in the dataset). This allows mapping more patterns that cannot be easily attracted by the right neuron. For example, an individual can have pictures with and without a beard and this can lead to confusion. If the matrix is too small, it's possible to observe an instability in the convergence process, while if it's too large, it's easy to see redundancies. The right choice depends on each different dataset and on the internal variance and there's no way to define a standard criterion. A good starting point is picking a matrix whose capacity is between 2.0 and 3.0 times larger than the number of desired attractors and then increasing or reducing its size until the accuracy reaches a maximum. The last element to consider is the labeling phase. At the end of the training process, we have no knowledge about the weight distribution in terms of winning neurons, so it's necessary to process the dataset and annotate the winning unit for each pattern. In this way, it's possible to submit new patterns to get the most likely label. This process has not been shown, but it's straightforward and the reader can easily implement it for every different scenario.

Summary

In this chapter, we have discussed Hebb's rule, showing how it can drive the computation of the first principal component of the input dataset. We have also seen that this rule is unstable because it leads to the infinite growth of the synaptic weights and how it's possible to solve this problem using normalization or Oja's rule.

We have introduced two different neural networks based on Hebbian learning (Sanger's and Rubner-Tavan's networks), whose internal dynamics are slightly different, which are able to extract the first n principal components in the right order (starting from the largest eigenvalue) without eigendecomposing the input covariance matrix.

Finally, we have introduced the concept of SOM and presented a model called a Kohonen network, which is able to map the input patterns onto a surface where some attractors (one per class) are placed through a competitive learning process. Such a model is able to recognize new patterns (belonging to the same distribution) by eliciting a strong response in the attractor, that is most similar to the pattern. In this way, after a labeling process, the model can be employed as a soft classifier that can easily manage noisy or altered patterns.

In the next chapter, we're going to discuss some important clustering algorithms, focusing on the difference (already discussed in the previous chapters) between hard and soft clustering and discussing the main techniques employed to evaluate the performance of an algorithm.

7
Clustering Algorithms

In this chapter, we are going to introduce some fundamental clustering algorithms, discussing both their strengths and weaknesses. The field of unsupervised learning, as well as any other machine learning approach, must be always based on the concept of Occam's razor. Simplicity must always be preferred when performance meets the requirements. However, in this case, the ground truth can be unknown. When a clustering algorithm is adopted as an exploratory tool, we can only assume that the dataset represents a precise data generating process. If this assumption is correct, the best strategy is to determine the number of clusters to maximize the internal cohesion (denseness) and the external separation. This means that we expect to find blobs (or isles) whose samples share some common and partially unique features.

In particular, the algorithms we are going to present are:

- **k-Nearest Neighbors** (**KNN**) based on KD Trees and Ball Trees
- K-means and K-means++
- Fuzzy C-means
- Spectral clustering based on the Shi-Malik algorithm

k-Nearest Neighbors

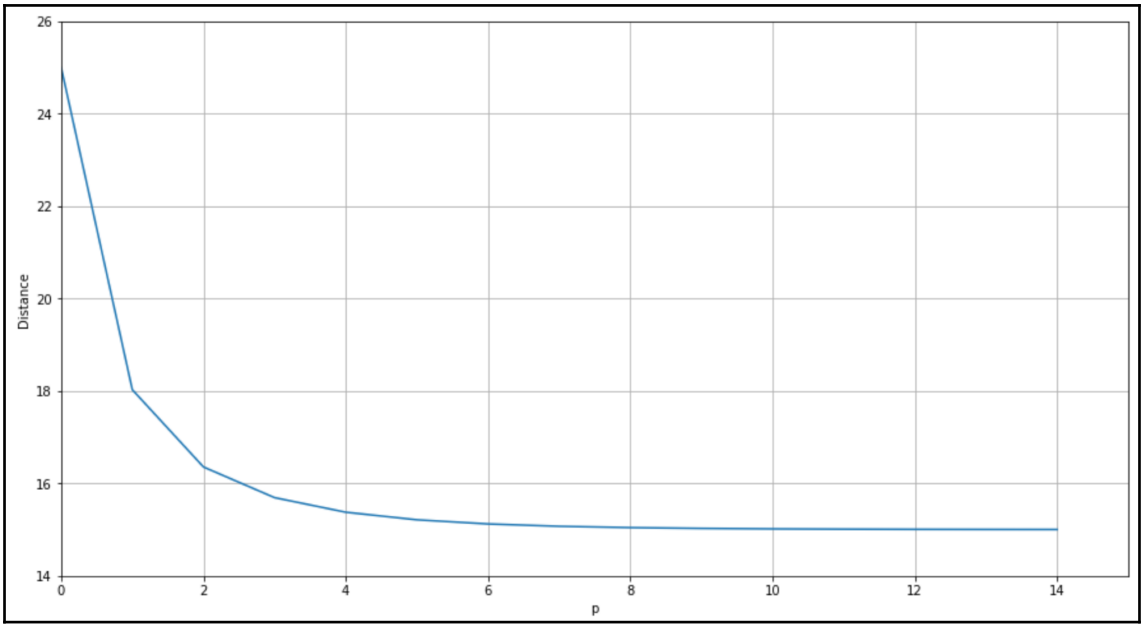
This algorithm belongs to a particular family called **instance-based** (the methodology is called **instance-based learning**). It differs from other approaches because it doesn't work with an actual mathematical model. On the contrary, the inference is performed by direct comparison of new samples with existing ones (which are defined as instances). KNN is an approach that can be easily employed to solve clustering, classification, and regression problems (even if, in this case, we are going to consider only the first technique). The main idea behind the clustering algorithm is very simple. Let's consider a data generating process p_{data} and a finite a dataset drawn from this distribution:

$$X = \{\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_n\} \quad where \quad \bar{x}_i \in \mathbb{R}^N$$

Each sample has a dimensionality equal to *N*. We can now introduce a distance function $d(x_1, x_2)$, which in the majority of cases can be generalized with the Minkowski distance:

$$d_p(\bar{x}_1, \bar{x}_2) = \left(\sum_{j=1}^{N} \left| x_1^{(j)} - x_2^{(j)} \right|^p \right)^{\frac{1}{p}}$$

When $p = 2$, d_p represents the classical Euclidean distance, that is normally the default choice. In particular cases, it can be useful to employ other variants, such as $p = 1$ (which is the Manhattan distance) or $p > 2$. Even if all the properties of a metric function remain unchanged, different values of *p* yield results that can be *semantically* diverse. As an example, we can consider the distance between points $x_1 = (0, 0)$ and $x_2 = (15, 10)$ as a function of *p*:



Minkowski distance between (0, 0) and (15, 10) as a function of parameter p

The distance decreases monotonically with p and converges to the largest component absolute difference, $|x_1^{(j)} - x_2^{(j)}|$, when $p \to \infty$. Therefore, whenever it's important to weight all the components in the same way in order to have a consistent metric, small values of p are preferable (for example, $p=1$ or 2). This result has also been studied and formalized by Aggarwal, Hinneburg, and Keim (in *On the Surprising Behavior of Distance Metrics in High Dimensional Space*, *Aggarwal C. C., Hinneburg A., Keim D. A., ICDT 2001*), who proved a fundamental inequality. If we consider a generic distribution G of M points $x_i \in (0, 1)^d$, a distance function based on the L_p norm, and the maximum D_{max}^p and minimum D_{min}^p distances (computed using the L_p norm) between two points, x_j and x_k drawn from G and $(0, 0)$, the following inequality holds:

$$C_p \leq \lim_{d \to \infty} E\left[\frac{D_{max}^p - D_{min}^p}{d^{\frac{1}{p}-\frac{1}{2}}}\right] \leqslant (M-1)C_p \;\; where\; C_p \geqslant 0$$

It's clear that when the input dimensionality is very high and $p >> 2$, the expected value, $E[D_{max}^{p} - D_{min}^{p}]$, becomes bounded between two constants, $k_1 (C_p d^{1/p-1/2})$ and $k_2 ((M-1)C_p d^{1/p-1/2}) \rightarrow 0$, reducing the actual effect of almost any distance. In fact, given two generic couples of points (x_1, x_2) and (x_3, x_4) drawn from G, the natural consequence of the following inequality is that $d_p(x_1, x_2) \approx d_p(x_3, x_4)$ when $p \rightarrow \infty$, independently of their relative positions. This important result confirms the importance of choosing the right metric according to the dimensionality of the dataset and that $p = 1$ is the best choice when $d >> 1$, while $p >> 1$ can produce inconsistent results due the ineffectiveness of the metric. To see direct confirmation of this phenomenon, it's possible to run the following snippet, which computes the average difference between maximum and minimum distances considering `100` sets containing `100` samples drawn from a uniform distribution, $G \sim U(0, 1)$. In the snippet, the case of `d=2`, `100`, `1000` is analyzed with Minkowski metrics with `P=` `1`, `2`, `10`, `100` (the final values depend on the random seed and how many times the experiment is repeated):

```
import numpy as np

from scipy.spatial.distance import pdist

nb_samples = 100
nb_bins = 100

def max_min_mean(p=1.0, d=2):
    Xs = np.random.uniform(0.0, 1.0, size=(nb_bins, nb_samples, d))
    pd_max = np.zeros(shape=(nb_bins, ))
    pd_min = np.zeros(shape=(nb_bins, ))

    for i in range(nb_bins):
        pd = pdist(Xs[i], metric='minkowski', p=p)
        pd_max[i] = np.max(pd)
        pd_min[i] = np.min(pd)
    return np.mean(pd_max - pd_min)

print('P=1 -> {}'.format(max_min_mean(p=1.0)))
print('P=2 -> {}'.format(max_min_mean(p=2.0)))
print('P=10 -> {}'.format(max_min_mean(p=10.0)))
print('P=100 -> {}'.format(max_min_mean(p=100.0)))

P=1 -> 1.79302317381
P=2 -> 1.27290283592
P=10 -> 0.989257369005
P=100 -> 0.983016242436

print('P=1 -> {}'.format(max_min_mean(p=1.0, d=100)))
print('P=2 -> {}'.format(max_min_mean(p=2.0, d=100)))
```

```
print('P=10 -> {}'.format(max_min_mean(p=10.0, d=100)))
print('P=100 -> {}'.format(max_min_mean(p=100.0, d=100)))

P=1 -> 17.1916057948
P=2 -> 1.76155714836
P=10 -> 0.340453945928
P=100 -> 0.288625281313

print('P=1 -> {}'.format(max_min_mean(p=1.0, d=1000)))
print('P=2 -> {}'.format(max_min_mean(p=2.0, d=1000)))
print('P=10 -> {}'.format(max_min_mean(p=10.0, d=1000)))
print('P=100 -> {}'.format(max_min_mean(p=100.0, d=1000)))

P=1 -> 55.2865105705
P=2 -> 1.77098913218
P=10 -> 0.130444336657
P=100 -> 0.0925427145923
```

A particular case, that is a direct consequence of the previous inequality is when the largest absolute difference between components determines the most important factor of a distance, large values of p can be employed. For example, if we consider three points, $x_1 = (0, 0)$, $x_2 = (15, 10)$, and $x_3 = (15, 0)$, $d_2(x_1, x_2) \approx 18$ and $d_2(x_1, x_3) = 15$. So, if we set a threshold at $d = 16$ centered at x_1, x_2 is outside the boundaries. If instead $p = 15$, both distances become close to 15 and the two points (x_2 and x_3) are inside the boundaries. A particular use of large values of p is when it's important to take into account the inhomogeneity among components. For example, some feature vectors can represent the age and height of a set of people. Considering a test person $x = (30, 175)$, with large p values, the distances between x and two samples $(35, 150)$ and $(25, 151)$ are almost identical (about 25.0), and the only dominant factor becomes the height difference (independent from the age).

The KNN algorithm determines the k closest samples of each training point. When a new sample is presented, the procedure is repeated with two possible variants:

- With a predefined value of k, the KNN are computed
- With a predefined radius/threshold r, all the neighbors whose distance is less than or equal to the radius are computed

The philosophy of KNN is that similar samples can share their features. For example, a recommendation system can cluster users using this algorithm and, given a new user, find the most similar ones (based, for example, on the products they bought) to recommend the same category of items. In general, a similarity function is defined as the reciprocal of a distance (there are some exceptions, such as the cosine similarity):

$$s(\bar{x}_1, \bar{x}_2) = f(d_p(\bar{x}_1, \bar{x}_2)) = \frac{1}{d_p(\bar{x}_1, \bar{x}_2)} \quad for \;\; d_p(\bar{x}_1, \bar{x}_2) \neq 0$$

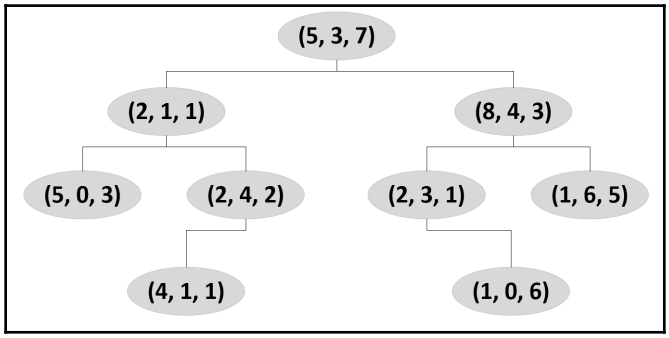
Two different users, *A* and *B*, who are classified as neighbors, will differ under some viewpoints, but, at the same time, they will share some peculiar features. This statement authorizes us to increase the homogeneity by *suggesting the differences*. For example, if *A* liked book b_1 and *B* liked b_2, we can recommend b_1 to *B* and b_2 to *A*. If our hypothesis was correct, the similarity between *A* and *B* will be increased; otherwise, the two users will move towards other clusters that better represent their behavior.

Unfortunately, the *vanilla* algorithm (in Scikit-Learn it is called the **brute-force** algorithm) can become extremely slow with a large number of samples because it's necessary to compute all the pairwise distances in order to answer any query. With *M* points, this number is equal to M^2, which is often unacceptable (if *M* = 1,000, each query needs to compute a million distances). More precisely, as the computation of a distance in an N-dimensional space requires *N* operations, the total complexity becomes $O(M^2N)$, which can be reasonable only for small values of both *M* and *N*. That's why some important strategies have been implemented to reduce the computational complexity.

KD Trees

As all KNN queries can be considered search problems, one of the most efficient way to reduce the overall complexity is to reorganize the dataset into a tree structure. In a binary tree (one-dimensional data), the average computational complexity of a query is $O(log\ M)$, because we assume we have almost the same number of elements in each branch (if the tree is completely unbalanced, all the elements are inserted sequentially and the resulting structure has a single branch, so the complexity becomes $O(M)$). In general, the real complexity is slightly higher than $O(log\ M)$, but the operation is always much more efficient than a vanilla search, which is $O(M^2)$.

However, we normally work with N-dimensional data and the previous structure cannot be immediately employed. KD Trees extend the concept of a binary for $N > 1$. In this case, a split cannot be immediately performed and a different strategy must be chosen. The easiest way to solve this problem is to select a feature at each level *(1, 2, ..., N)* and repeat the process until the desired depth is reached. In the following diagram, there's an example of KD Trees with three-dimensional points:



Example of three-dimensional KD Tree

The root is point **(5, 3, 7)**. The first split is performed considering the first feature, so two children are **(2, 1, 1)** and **(8, 4, 3)**. The second one operates on the second feature and so on. The average computational complexity is *O(N log M)*, but if the distribution is very asymmetric, the probability that the tree becomes unbalanced is very high. To mitigate this issue, it's possible to select the feature corresponding to the median of the (sub-)dataset and to continue splitting with this criterion. In this way, the tree is guaranteed to be balanced. However, the average complexity is always proportional to the dimensionality and this can dramatically affect the performance.

For example, if $M = 10{,}000$ and $N = 10$, using the log_{10}, $O(N \log M) = O(40)$, while, with $N = 1{,}000$, the complexity becomes $O(40{,}000)$. Generally, KD Trees suffers the *curse of dimensionality* and when N becomes large, the average complexity is about $O(MN)$, which is always better than the *vanilla* algorithm, but often too expensive for real-life applications. Therefore, KD Trees is really effective only when the dimensionality is not too high. In all other cases, the probability of having an unbalanced tree and the resulting computational complexity suggest employing a different method.

Ball Trees

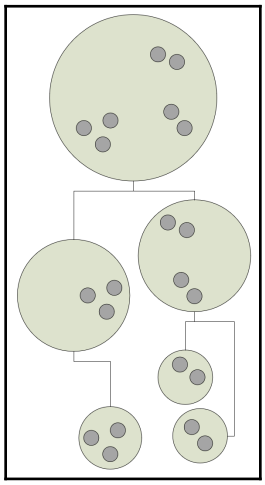
An alternative to KD Trees is provided by **Ball Trees**. The idea is to rearrange the dataset in a way that is almost insensitive to high-dimensional samples. A ball is defined as a set of points whose distance from a center sample is less than or equal to a fixed radius:

$$B_R(\bar{x}_c) = \{\bar{x}_i : d_p(\bar{x}_i, \bar{x}_c) \leqslant R\}$$

Starting from the first main ball, it's possible to build smaller ones nested into the parent ball and stop the process when the desired depth has been reached. A fundamental condition is that a point can always belong to a single ball. In this way, considering the cost of the N-dimensional distance, the computational complexity is *O(N log M)* and doesn't suffer the curse of dimensionality like KD Trees. The structure is based on hyperspheres, whose boundaries are defined by the equations (given a center point *x* and a radius R_i):

$$x_1^2 + x_2^2 + \ldots + x_N^2 = R_i^2$$

Therefore, the only operation needed to find the right ball is measuring the distance between a sample and the centers starting from the smallest balls. If a point is outside the ball, it's necessary to move upwards and check the parents, until the ball containing the sample is found. In the following diagram, there's an example of Ball Trees with two levels:



Example of Ball Trees with seven bidimensional points and two levels

In this example, the seven bidimensional points are split first into two balls containing respectively three and four points. At the second level, the second ball is split again into two smaller balls containing two points each. This procedure can be repeated until a fixed depth is reached or by imposing the maximum number of elements that a leaf must contain (in this case, it can be equal to *3*).

Both KD Trees and Ball Trees can be efficient structures to reduce the complexity of KNN queries. However, when fitting a model, it's important to consider both the *k parameter* (which normally represents the average or the standard number of neighbors computed in a query) and the maximum tree depth. These particular structures are not employed for common tasks (such as sorting) and their efficiency is maximized when all the requested neighbors can be found in the same sub-structure (with a size $K << M$, to avoid an implicit fallback to the *vanilla* algorithm). In other words, the tree has the role of reducing the dimensionality of the search space by partitioning it into reasonably small regions.

At the same time, if the number of samples contained in a leaf is small, the number of tree nodes grows and the complexity is subsequently increased. The negative impact is doubled because on average it's necessary to explore more nodes and if *k* is much greater than the number of elements contained in a node, it's necessary to merge the samples belonging to different nodes. On the other side, a very large number of samples per node leads to a condition that is close to the *vanilla* algorithm. For example, if $M = 1{,}000$ and each node contains 250 elements, once the right node is computed, the number of distances to compute is comparable with the initial dataset size and no real advantage is achieved by employing a tree structure. An acceptable practice is to set the size of a life equal to $5 \div 10$ times the average value of *k*, to maximize the probability to find all the neighbors inside the same leaf. However, every specific problem must be analyzed (while also benchmarking the performances) in order to find the most appropriate value. If different values for *k* are necessary, it's important to consider the relative frequencies of the queries. For example, if a program needs 10 *5-NN* queries and 1 *50-NN* query, it's probably better to set a leaf size equal to 25, even if the *50-NN* query will be more expensive. In fact, setting a good value for a second query (for example, 200) will dramatically increase the complexity of the first 10 queries, driving to a performance loss.

Example of KNN with Scikit-Learn

In order to test the KNN algorithm, we are going to use the MNIST handwritten digit dataset provided directly by Scikit-Learn. It is made up of 1,797 8 × 8 grayscale images representing the digits from 0 to 9. The first step is loading it and normalizing all the values to be bounded between 0 and 1:

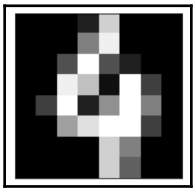
```
import numpy as np

from sklearn.datasets import load_digits

digits = load_digits()
X_train = digits['data'] / np.max(digits['data'])
```

The dictionary `digits` contains both the images, `digits['images']`, and the flattened 64-dimensional arrays, `digits['data']`. Scikit-Learn implements different classes (for example, it's possible to work directly with KD Trees and Ball Trees using the KDTree and BallTree classes) that can be used in the context of KNN (as clustering, classification, and regression algorithms). However, we're going to employ the main class, `NearestNeighbors`, which allows performing clustering and queries based either on the number of neighbors or on the radius of a ball centered on a sample:

```
from sklearn.neighbors import NearestNeighbors

knn = NearestNeighbors(n_neighbors=50, algorithm='ball_tree')
knn.fit(X_train)
```

We have chosen to have a default number of neighbors equal to `50` and an algorithm based on a `ball_tree`. The leaf size (`leaf_size`) parameter has been kept to its default value equal to `30`. We have also employed the default metric (Euclidean), but it's possible to change it using the `metric` and `p` parameters (which is the order of the Minkowski metric). Scikit-Learn supports all the metrics implemented by SciPy in the `scipy.spatial.distance` package. However, in the majority of cases, it's sufficient to use a Minkowski metric and adjust the value of `p` if the results are not acceptable with any number of neighbors. Other metrics, such as the cosine distance, can be employed when the similarity must not be affected by the Euclidean distance, but only by the angle between two vectors pointing at the samples. Applications that use this metric include, for example, deep learning models for natural language processing, where the words are embedded into feature vectors whose semantic similarity is proportional to their Cosine distance.

We can now query the model in order to find 50 neighbors of a sample. For our purposes, we have selected the sample with index 100, which represents a 4 (the images have a very low resolution, but it's always possible to distinguish the digit):



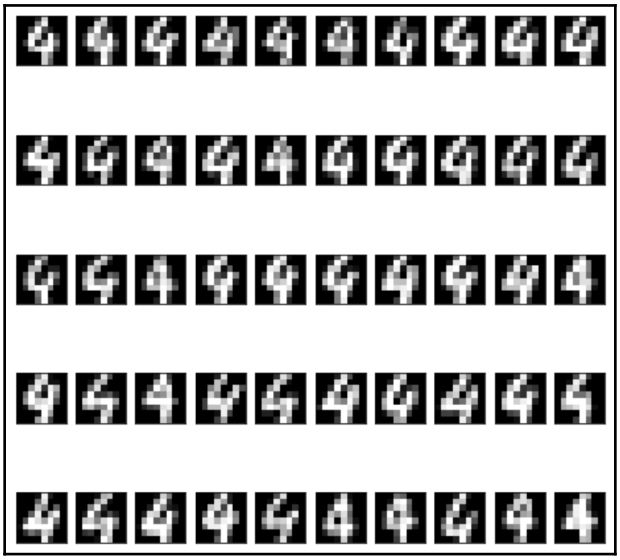
Sample digit used to query the KNN model

The query can be performed using the instance method `kneighbors`, which allows specifying the number of neighbors (`n_neighbors` parameter the default is the value selected during the instantiation of the class) and whether we want to also get the distances of each neighbor (the `return_distance` parameter). In this example, we are also interested in evaluating *how far* the neighbors are from the center, so we set `return_distance=True`:

```
distances, neighbors = knn.kneighbors(X_train[100].reshape(1, -1),
return_distance=True)

print(distances[0])
```

```
[ 0.          0.91215747  1.16926793  1.22633855  1.24058958  1.32139841
  1.3564084   1.36645069  1.41972709  1.43341812  1.45236875  1.50130152
  1.52709897  1.5499496   1.62379763  1.62620148  1.6345871   1.64292993
  1.66770801  1.70934929  1.71619128  1.71619128  1.72187216  1.73317808
  1.74888357  1.75445861  1.75668367  1.75779514  1.76555586  1.77878118
  1.788636    1.79408751  1.79626348  1.80169191  1.80277564  1.80385871
  1.80494113  1.8125      1.81572988  1.83498978  1.84771819  1.87291551
  1.87916205  1.88020112  1.88538789  1.88745861  1.88952706  1.90906554
  1.91213232  1.92333532]
```

The first neighbor is always the center, so its distance is `0`. The other ones range from 0.9 to 1.9. Considering that, in this case, the maximum possible distance is 8 (between a 64-dimensional vector *a = (1, 1, ..., 1)* and the null vector), the result could be acceptable. In order to get confirmation, we can plot the neighbors as bidimensional 8 × 8 arrays (the returned array, `neighbors`, contains the indexes of the samples). The result is shown in the following screenshot:



50 neighbors selected by the KNN model

As it's possible to see, there are no errors, but all the shapes are slightly different. In particular, the last one, which is also the farthest, has a lot of white pixels (corresponding to the value 1.0), explaining the reason of a distance equal to about 2.0. I invite the reader to test the `radius_neighbors` method until spurious values appear among the results. It's also interesting to try this algorithm with the Olivetti faces dataset, whose complexity is higher and many more geometrical parameters can influence the similarity.

K-means

When we discussed the Gaussian mixture algorithm, we defined it as *Soft K-means*. The reason is that each cluster was represented by three elements: mean, variance, and weight. Each sample always belongs to all clusters with a probability provided by the Gaussian distributions. This approach can be very useful when it's possible to manage the probabilities as weights, but in many other situations, it's preferable to determine a single cluster per sample. Such an approach is called hard clustering and K-means can be considered the hard version of a Gaussian mixture. In fact, when all variances $\Sigma_i \rightarrow 0$, the distributions degenerate to Dirac's Deltas, which represent perfect spikes centered at a specific point. In this scenario, the only possibility to determine the most appropriate cluster is to find the shortest distance between a sample point and all the centers (from now on, we are going to call them *centroids*). This approach is also based on an important double principle that should be taken into account in every clustering algorithm. The clusters must be set up to maximize:

- The intra-cluster cohesion
- The inter-cluster separation

This means that we expect to label high-density regions that are well separated from each other. When this is not possible, the criterion must try to minimize the intra-cluster average distance between samples and centroid. This quantity is also called *inertia* and it's defined as:

$$S = \sum_{j=1}^{k} \sum_{\bar{x}_j \in C_j} \left\| \bar{x}_i - \bar{\mu}_j \right\|^2$$

High levels of inertia imply low cohesion because there are probably too many points belongings to clusters whose centroids are too far away. The problem can be solved by minimizing the previous quantity. However, the computational complexity needed to find the global minimum is exponential (K-means belongs to the class of NP-Hard problems). The alternative approach employed by the K-means algorithm, also known as **Lloyd's algorithm**, is iterative and starts from selecting *k* random centroids (in the next section, we're going to analyze a more efficient method) and adjusting them until their configuration becomes stable.

The dataset to cluster (with *M* samples) is represented as:

$$X = \{\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_M\} \;\; where \;\; \bar{x}_i \in \mathbb{R}^N$$

An initial guess for the centroids is:

$$M^{(0)} = \left\{\bar{\mu}_0^{(0)}, \bar{\mu}_1^{(0)}, \ldots, \bar{\mu}_k^{(0)}\right\} \;\; where \;\; \bar{\mu}_i^{(0)} \in \mathbb{R}^N$$

There are no particular restrictions on the initial values. However, the choice can influence both the convergence speed and the minimum that is found. The iterative procedure will loop over the dataset, computing the Euclidean distance between x_i and each μ_j and assigning a cluster based on the criterion:

$$C^{(t)}(\bar{x}_i) = argmin_j \; d(\bar{x}_i, \bar{\mu}_j^{(t)})$$

Once all the samples have been clustered, the new centroids are computed:

$$\bar{\mu}_j^{(t)} = \frac{1}{N_{C_j}} \sum_{i \in C_j} \bar{x}_i \;\; \forall \, j \in [1, k]$$

The quantity N_{Cj} represents the number of points belonging to cluster *j*. At this point, the inertia is recomputed and the new value is compared with the previous one. The procedure will stop either after a fixed number of iterations or when the variations in the inertia become smaller than a predefined threshold. Lloyd's algorithm is very similar to a particular case of the EM algorithm. In fact, the first step of each iteration is the computation of an *expectation* (the centroid configuration), while the second step maximizes the intra-cluster cohesion by minimizing the inertia.

The complete vanilla K-means algorithm is:

1. Set a maximum number of iterations N_{max}.
2. Set a tolerance *Thr*.
3. Set the value of *k* (number of expected clusters).
4. Initialize vector $C^{(0)}$ with random values. They can be points belonging to the dataset or sampled from a suitable distribution.
5. Compute the initial inertia $S^{(0)}$

6. Set $N = 0$.
7. While $N < N_{max}$ or $||S^{(t)} - S^{(t-1)}|| > Thr$:
    1. $N = N + 1$
    2. For x_i in X:
        1. Assign x_i to a cluster using the shortest distance between x_i and μ_j
    3. Recompute the centroid vector $C^{(t)}$
    4. Recompute the inertia $S^{(t)}$

The algorithm is quite simple and intuitive, and there are many real-life applications based on it. However, there are two important elements to consider. The first one is the convergence speed. It's easy to show that every initial guess drives to a convergence point, but the number of iterations is dramatically influenced by this choice and there's no guarantee to find the global minimum. If the initial centroids are close to the final ones, the algorithm needs only a few steps to correct the values, but when the choice is totally random, it's not uncommon to need a very high number of iterations. If there are N samples and k centroids, Nk distances must be computed at each iteration, leading to an inefficient result. In the next paragraph, we'll show how it's possible to initialize the centroids to minimize the convergence time.

Another important aspect is that, contrary to KNN, K-means needs to predefine the number of expected clusters. In some cases, this is a secondary problem because we already know the most appropriate value for k. However, when the dataset is high-dimensional and our knowledge is limited, this choice could be hazardous. A good approach to solve the issue is to analyze the final inertia for a different number of clusters. As we expect to maximize the intra-cluster cohesion, a small number of clusters will lead to an increased inertia. We try to pick the highest point below a maximum tolerable value. Theoretically, we can also pick $k = N$. In this case, the inertia becomes zero because each point represents the centroid of its cluster, but a large value for k transforms the clustering scenario into a fine-grained partitioning that might not be the best strategy to capture the feature of a consistent group. It's impossible to define a rule for the upper bound k_{max}, but we assume that this value is always much less than N. The best choice is achieved by selecting k to minimize the inertia, selecting the values from a set bounded, for example, between 2 and k_{max}.

K-means++

We have said that a good choice for the initial centroids can improve the convergence speed and leads to a minimum that is closer to the global optimum of the inertia S. Arthur and Vassilvitskii (in *The Advantages of Careful Seeding, Arthur, D., Vassilvitskii S., k-means++: Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms*) proposed a method called K-means++, which allows increasing the accuracy of the initial centroid guess considering the most likely final configuration.

In order to expose the algorithm, it's useful to introduce a function, *D(x, i)*, which is defined as:

$$D(\bar{x}, i) = min_i \ d(\bar{x}, \bar{\mu}_i) \ for \ i = 1 \ to \ p \leqslant k$$

D(x, i) defines the shortest distance between each sample and one of the centroids already selected. As the process is incremental, this function must be recomputed after all steps. For our purposes, let's also define an auxiliary probability distribution (we omit the index variable for simplicity):

$$G(\bar{x}) = \frac{D(\bar{x})^2}{\sum_{j=1}^{M} D(\bar{x}_j)^2}$$

The first centroid μ_0 is sampled from *X* using a uniform distribution. The next steps are:

1. Compute *D(x, i)* for all $x \in X$ considering the centroids already selected
2. Compute *G(x)*
3. Select the next centroid μ_i from *X* with a probability *G(x)*

In the aforementioned paper, the authors showed a very important property. If we define S^* as the global optimum of *S*, a K-means++ initialization determines an upperbound for the expected value of the actual inertia:

$$E[S] \leqslant 8S^* (log \ k + 2)$$

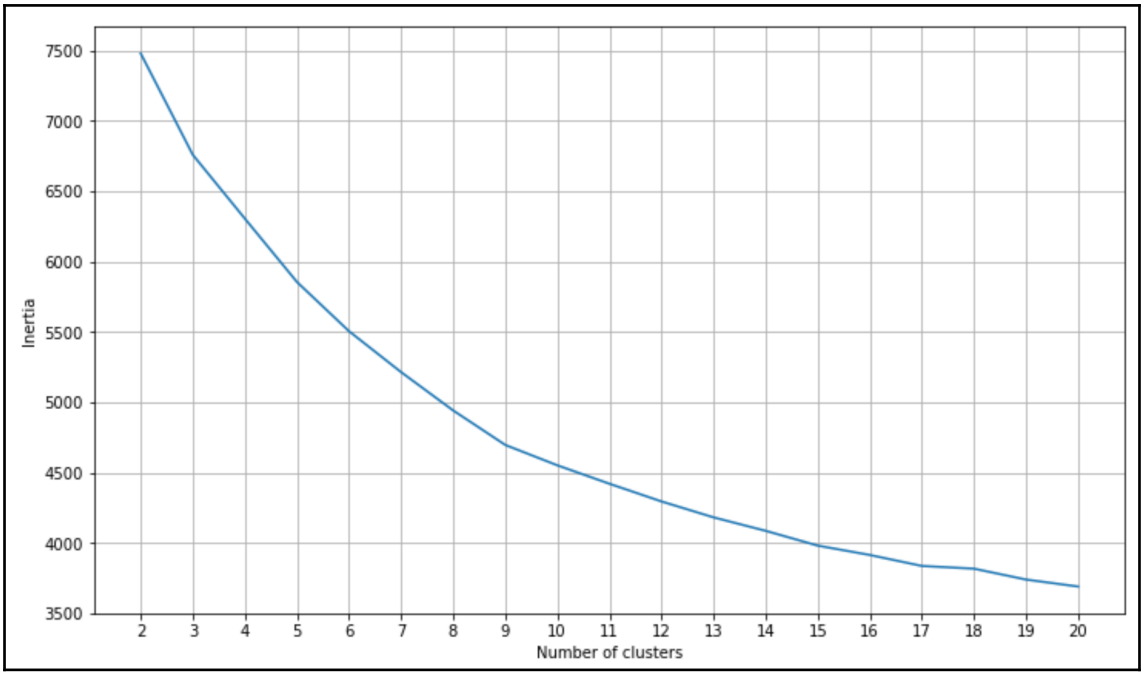
This condition is often expressed by saying that K-means++ is *O(log k)*-competitive. When *k* is sufficiently small, the probability of finding a local minimum close to the global one increases. However, K-means++ is still a probabilistic approach and different initializations on the same dataset lead to different initial configurations. A good practice is to run a limited number of initializations (for example, ten) and pick the one associated with the smallest inertia. When training complexity is not a primary issue, this number can be increased, but different experiments showed that the improvement achievable with a very large number of trials is negligible when compared to the actual computational cost. The default value in Scikit-Learn is ten and the author suggests to keep this value in the majority of cases. If the result continues to be poor, it's preferable to pick another method. Moreover, there are problems that cannot be solved using K-means (even with the best possible initialization), because one of the assumptions of the algorithm is that each cluster is a hypersphere and the distances are measured using a Euclidean function. In the following sections, we're going to analyze other algorithms that are not constrained to work with such limitations and can easily solve clustering problems using asymmetric cluster geometries.

Example of K-means with Scikit-Learn

In this example, we continue using the MNIST dataset (the `X_train` array is the same defined in the paragraph dedicated to KNN), but we want also to analyze different clustering evaluation methods. The first step is visualizing the inertia corresponding to different numbers of clusters. We are going to use the `KMeans` class, which accepts the `n_clusters` parameter and employs the K-means++ initialization as the default method (as explained in the previous section, in order to find the best initial configuration, Scikit-Learn performs several attempts and selects the configuration with the lowest inertia; it's possible to change the number of attempts through the `n_iter` parameter):

```
import numpy as np

from sklearn.cluster import KMeans

min_nb_clusters = 2
max_nb_clusters = 20

inertias = np.zeros(shape=(max_nb_clusters - min_nb_clusters + 1,))

for i in range(min_nb_clusters, max_nb_clusters + 1):
    km = KMeans(n_clusters=i, random_state=1000)
    km.fit(X_train)
    inertias[i - min_nb_clusters] = km.inertia_
```

We are supposing to analyze the range [2, 20]. After each training session, the final inertia can be retrieved using the `inertia_` instance variable. The following graph shows the plot of the values as a function of the number of clusters:

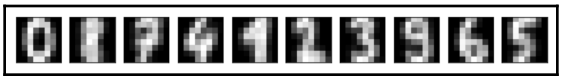


Inertia as a function of the number of clusters

As expected, the function is decreasing, starting from a value of about 7,500 and reaching about 3,700 with **20** clusters. In this case, we know that the real number is **10**, but it's possible to discover it by observing the trend. The slope is quite high before **10**, but it starts decreasing more and more slowly after this threshold. This is a signal that informs us that some clusters are not well separated, even if their internal cohesion is high. In order to confirm this hypothesis, we can set `n_clusters=10` and, first of all, check the centroids at the end of the training process:

```
km = KMeans(n_clusters=10, random_state=1000)
Y = km.fit_predict(X_train)
```

The centroids are available through the `cluster_centers_` instance variable. In the following screenshot, there's a plot of the corresponding bidimensional arrays:

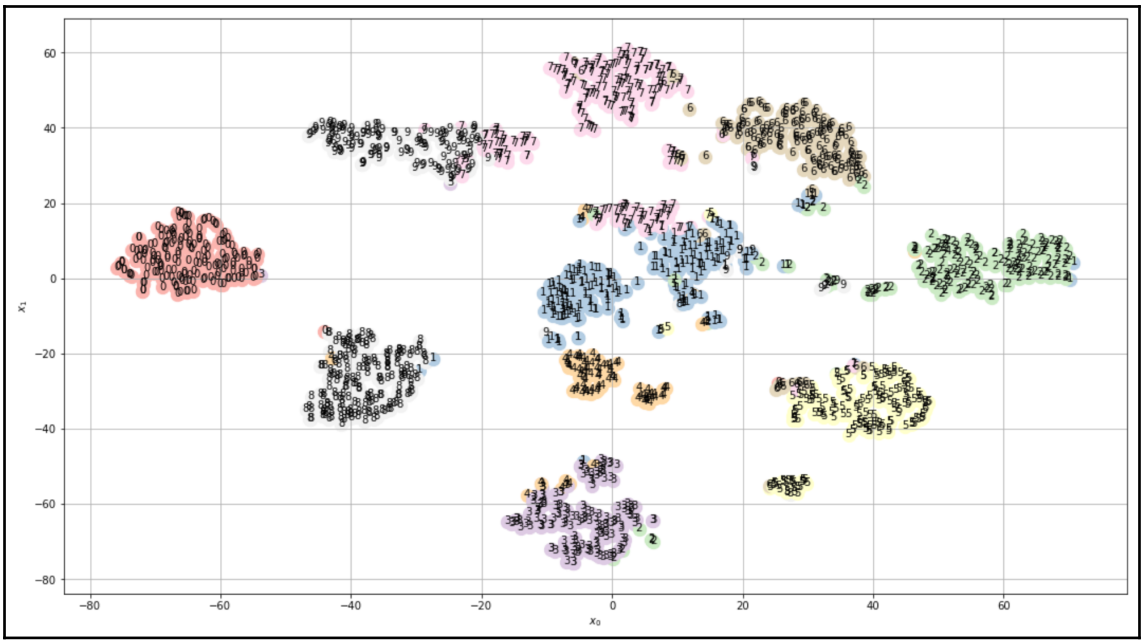


K-means centroid at the end of the training process

All the digits are present and there are no duplicates. This confirms that the algorithm has successfully separated the sets, but the final inertia (which is about 4,500) informs us that there are probably wrong assignments. To obtain confirmation, we can plot the dataset using a dimensionality-reduction method, such as t-SNE (see `Chapter 3`, *Graph-Based Semi-Supervised Learning* for further details):

```
from sklearn.manifold import TSNE

tsne = TSNE(n_components=2, perplexity=20.0, random_state=1000)
X_tsne = tsne.fit_transform(X_train)
```

At this point, we can plot the bidimensional dataset with the corresponding cluster labels:



t-SNE representation of the MNIST dataset; the labels correspond to the clusters

The plot confirms that the dataset is made up of well-separated blobs, but a few samples are assigned to the wrong cluster (this is not surprising considering the similarity between some pairs of digits). An important observation can further explain the trend of the inertia. In fact, the point where the slope changes almost abruptly corresponds to 9 clusters. Observing the t-SNE plot, we can immediately discover the reason: the cluster corresponding to the digit **7** is indeed split into 3 blocks. The main one contains the majority of samples, but there are another 2 smaller blobs that are wrongly *attached* to clusters **1** and **9**. This is not surprising, considering that the digit **7** can be very similar to a distorted **1** or **9**. However, these two spurious blobs are always at the boundaries of the wrong clusters (remember that the geometric structures are hyperspheres), confirming that the metric has successfully detected a low similarity. If a group of wrongly assigned samples were in the middle of a cluster, it would have meant that the separation failed dramatically and another method should be employed.

Evaluation metrics

In many cases, it's impossible to evaluate the performance of a clustering algorithm using only a visual inspection. Moreover, it's important to use standard objective metrics that allow for comparing different approaches. We are now going to introduce some methods based on the knowledge of the ground truth (the correct assignment for each sample) and one common strategy employed when the true labels are unknown.

Before discussing the scoring functions, we need to introduce a standard notation. If there are *k* clusters, we define the true labels as:

$$Y_{true} = \left\{ y_1^{true}, y_2^{true}, \ldots, y_M^{true} \right\} \;\; where \;\; y_i^{true} \in \{1, 2, \ldots, k\}$$

In the same way, we can define the predicted labels:

$$Y_{pred} = \left\{ y_1^{pred}, y_2^{pred}, \ldots, y_M^{pred} \right\} \;\; where \;\; y_i^{pred} \in \{1, 2, \ldots, k\}$$

Both sets can be considered as sampled from two discrete random variables (for simplicity, we denote them with the same names), whose probability mass functions are $P_{true}(y)$ and $P_{pred}(y)$ with a generic $y \in \{y_1, y_2, ..., y_k\}$ (y_i represents the index of the i^{th} cluster). These two probabilities can be approximated with a frequency count; so, for example, the probability $P_{true}(1)$ is computed as the number of samples whose true label is *1* $n_{true}(1)$ over the total number of samples *M*. In this way, we can define the entropies:

$$\begin{cases} H(Y_{true}) = -\sum_{i=1}^{k} p(y_i^{true}) \; log \; p(y_i^{true}) \\ H(Y_{pred}) = -\sum_{i=1}^{k} p(y_i^{pred}) \; log \; p(y_i^{pred}) \end{cases}$$

These quantities describe the intrinsic uncertainty of the random variables. They are maximized when all the classes have the same probability, while, for example, they are null if all the samples belong to a single class (minimum uncertainty). We also need to know the uncertainty of a random variable *Y* given another one *X*. This can be achieved using the conditional entropy $H(Y|X)$. In this case, we need to compute the joint probability $p(x, y)$ because the definition of $H(Y|X)$ is:

$$H(Y|X) = -\sum_{x} \sum_{y} p(x, y) \; log \; \frac{p(x, y)}{p(x)}$$

In order to approximate the previous expression, we can define the function $n(i_{true}, j_{pred})$, which counts the number of samples with the true label *i* assigned to cluster *j*. In this way, if there are *M* samples, the approximated conditional entropies become:

$$\begin{cases} H(Y_{true}|Y_{pred}) = -\sum_{i_{true}=1}^{M} \sum_{j_{pred}=1}^{M} \frac{n(i_{true}, \, j_{pred})}{M} \; log \; \frac{n(i_{true}, \, j_{pred})}{n_{pred}(j_{pred})} \\ H(Y_{pred}|Y_{true}) = -\sum_{i_{pred}=1}^{M} \sum_{j_{true}=1}^{M} \frac{n(i_{true}, \, j_{pred})}{M} \; log \; \frac{n(i_{true}, \, j_{pred})}{n_{true}(i_{true})} \end{cases}$$

Homogeneity score

This score is useful to check whether the clustering algorithm meets an important requirement: a cluster should contain only samples belonging to a single class. It's defined as:

$$h = 1 - \frac{H(Y_{true}|Y_{pred})}{H(Y_{true})}$$

It's bounded between *0* and *1*, with low values indicating a low homogeneity. In fact, when the knowledge of Y_{pred} reduces the uncertainty of Y_{true}, $H(Y_{true}|Y_{pred})$ becomes smaller ($h \to 1$) and viceversa. For our example, the homogeneity score can be computed as:

```
from sklearn.metrics import homogeneity_score

print(homogeneity_score(digits['target'], Y))
0.739148799605
```

The `digits['target']` array contains the true labels while `Y` contains the predictions (all the functions we are going to use accept the true labels as the first parameter and the predictions as the second one). The homogeneity score confirms that the clusters are rather homogeneous, but there's still a moderate level of uncertainty because some clusters contain wrong assignments. This method, together with the other ones, can be used to search for the right number of clusters and tune up all supplementary hyperparameters (such as the number of iterations or the metric function).

Completeness score

This score is complementary to the previous one. Its purpose is to provide a piece of information about the assignment of samples belonging to the same class. More precisely, a good clustering algorithm should assign all samples with the same true label to the same cluster. From our previous analysis, we know that, for example, the digit 7 has been wrongly assigned to both clusters 9 and 1; therefore, we expect a non-perfect completeness score. The definition is symmetric to the homogeneity score:

$$c = 1 - \frac{H(Y_{pred}|Y_{true})}{H(Y_{pred})}$$

The rationale is very intuitive. When $H(Y_{pred}|Y_{true})$ is low $(c \to 1)$, it means that the knowledge of the ground truth reduces the uncertainty about the predictions. Therefore, if we know that all the sample of subset A have the same label y_i, we are quite sure that all the corresponding predictions have been assigned to the same cluster. The completeness score for our example is:

```
from sklearn.metrics import completeness_score

print(completeness_score(digits['target'], Y))
0.747718831945
```

Again, the value confirms our hypothesis. The residual uncertainty is due to a lack of completeness because a few samples with the same label have been split into blocks that are assigned to wrong clusters. It's obvious that a perfect scenario is characterized by having both homogeneity and completeness scores equal to 1.

Adjusted Rand Index

This score is useful to compare the original label distribution with the clustering prediction. Ideally, we'd like to reproduce the exact ground truth distribution, but in general, this is very difficult in real-life scenarios. A way to measure the discrepancy is provided by the Adjusted Rand Index. In order to compute this score, we need to define the auxiliary variables:

- a: Number of sample pairs (y_i, y_j) that have the same true label and that are assigned to the same cluster
- b: Number of sample pairs (y_i, y_j) that have a different true label and that are assigned to different clusters

The Rand Index is defined as:

$$R = \frac{a+b}{\binom{M}{2}}$$

The Adjusted Rand Index is the Rand Index corrected for chance and it's defined as:

$$R_A = \frac{R - E[R]}{max(R) - E[R]}$$

The R_A measure is bounded between *-1* and *1*. A value close to *-1* indicates a prevalence of wrong assignments, while a value close to *1* indicates that the clustering algorithm is correctly reproducing the ground truth distribution. The Adjusted Rand Score for our example is:

```
from sklearn.metrics import adjusted_rand_score

print(adjusted_rand_score(digits['target'], Y))
0.666766395716
```

This value confirms that the algorithm is working well (because it's positive), but it can be further optimized by trying to reduce the number of wrong assignments. The Adjusted Rand Score is a very powerful tool when the ground truth is known and can be employed as a single method to optimize all the hyperparameters.

Silhouette score

This measure doesn't need to know the ground truth and can be used to check, at the same time, the intra-cluster cohesion and the inter-cluster separation. In order to define the Silhouette score, we need to introduce two auxiliary functions. The first one is the average intra-cluster distance of a sample x_i belonging to a cluster C_j:

$$a(\bar{x}_i) = \frac{1}{n(j)} \sum_p d(\bar{x}_i, \bar{x}_p) \ \ \forall \ \bar{x}_p \in C_j$$

In the previous expression, *n(k)* is the number of samples assigned to the cluster C_j and *d(a, b)* is a standard distance function (in the majority of cases, the Euclidean distance is chosen). We need also to define the lowest inter-cluster distance which can be interpreted as the average nearest-cluster distance. In the sample $x_i \in C_j$, let's call C_t the nearest cluster; therefore, the function is defined as:

$$b(\bar{x}_i) = \frac{1}{n(t)} \sum_t d(\bar{x}_i, \bar{x}_t) \ \ \forall \ \bar{x}_t \in C_t$$

The Silhouette score for sample x_i is:

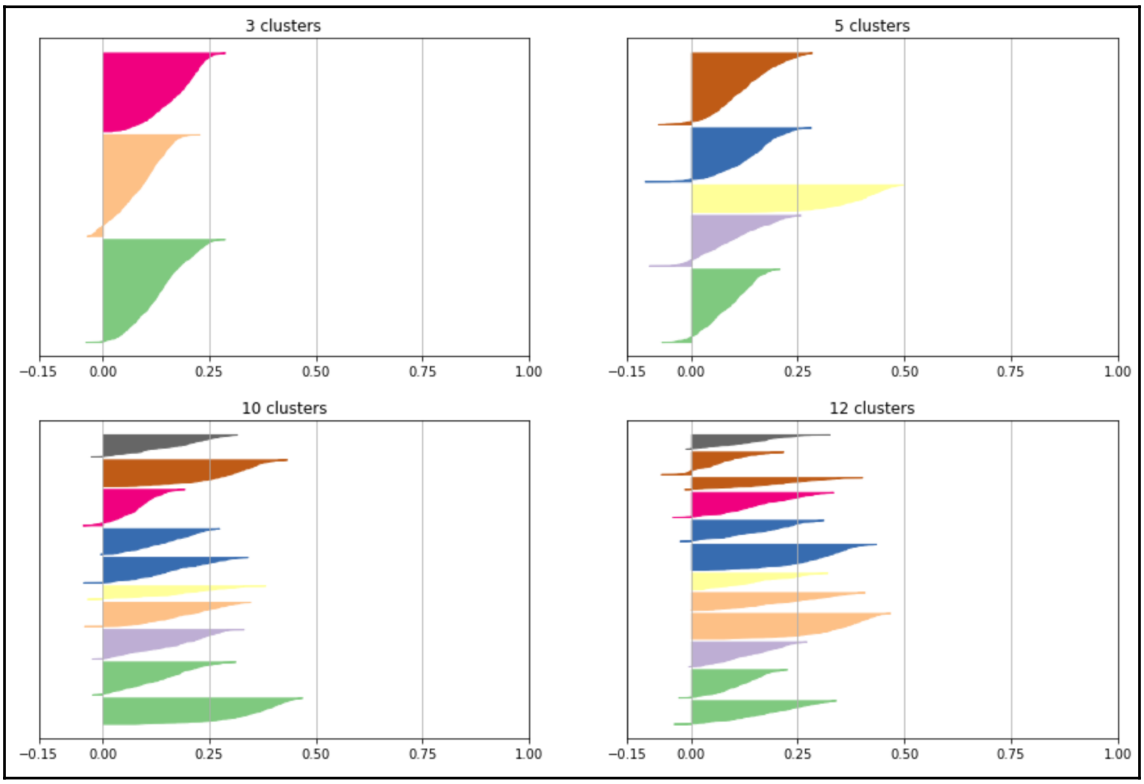
$$s(\bar{x}_i) = \frac{b(\bar{x}_i) - a(\bar{x}_i)}{max\left(a(\bar{x}_i), b(\bar{x}_i)\right)}$$

The value of $s(x_i)$, like for the Adjusted Rand Index, is bounded between *-1* and *1*. A value close to *-1* indicates that $b(x_i) << a(x_i)$, so the average intra-cluster distance is greater than the average nearest-cluster index and sample x_i is wrongly assigned. Viceversa, a value close to *1* indicates that the algorithm achieved a very good level of internal cohesion and inter-cluster separation (because $a(x_i) << b(x_i)$). Contrary to the other measure, the Silhouette score isn't a cumulative function and must be computed for each sample. A feasible strategy is to analyze the average value, but in this way, it's not possible to determine which clusters have the highest impact on the result. Another approach (the most common), is based on Silhouette plots, which display the score for each cluster in descending order. In the following snippet, we create plots for four different values of `n_clusters` (`3`, `5`, `10`, `12`):

```
import matplotlib.pyplot as plt
import matplotlib.cm as cm

import numpy as np

from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_samples

fig, ax = plt.subplots(2, 2, figsize=(15, 10))

nb_clusters = [3, 5, 10, 12]
mapping = [(0, 0), (0, 1), (1, 0), (1, 1)]

for i, n in enumerate(nb_clusters):
    km = KMeans(n_clusters=n, random_state=1000)
    Y = km.fit_predict(X_train)

    silhouette_values = silhouette_samples(X_train, Y)

    ax[mapping[i]].set_xticks([-0.15, 0.0, 0.25, 0.5, 0.75, 1.0])
    ax[mapping[i]].set_yticks([])
    ax[mapping[i]].set_title('%d clusters' % n)
    ax[mapping[i]].set_xlim([-0.15, 1])
    ax[mapping[i]].grid()
    y_lower = 20

    for t in range(n):
        ct_values = silhouette_values[Y == t]
```

```
            ct_values.sort()

            y_upper = y_lower + ct_values.shape[0]

            color = cm.Accent(float(t) / n)
            ax[mapping[i]].fill_betweenx(np.arange(y_lower, y_upper), 0,
ct_values, facecolor=color, edgecolor=color)

            y_lower = y_upper + 20
```

The result is shown in the following graph:



Silhouette plots for different number of clusters

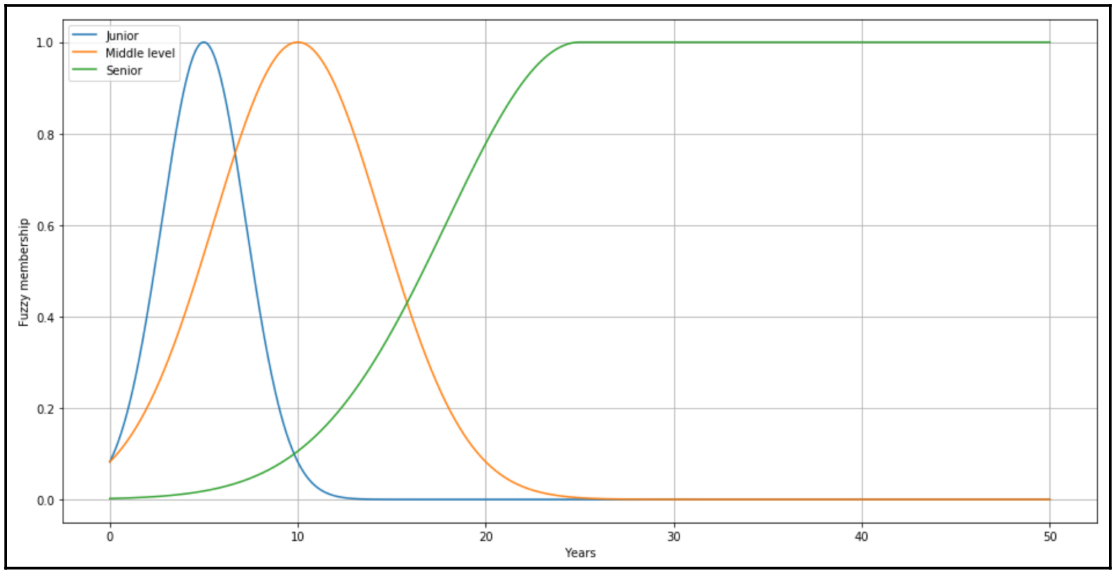
The analysis of a Silhouette plot should follow some common guidelines:

- The width of each block must be proportional to the number of samples that are expected to belong to the corresponding cluster. If the label distribution is uniform, all the blocks must have a similar width. Any asymmetry indicates wrong assignments. For example, in our case, we know that the right number of clusters is ten, but a couple of blocks are thinner than the other ones. This means that a cluster contains fewer samples than expected and the remaining ones have been assigned to wrong partitions.
- The shape of a block shouldn't be sharp and peaked (like a knife) because it means that many samples have a low Silhouette score. The ideal (realistic) scenario is made up of shapes similar to cigars with a minimum difference between the highest and lowest values. Unfortunately, this is not always possible to achieve, but it's always preferable to tune up the algorithm if the shapes are like the ones plotted in the first diagram (three clusters).
- The maximum Silhouette score should be close to *1*. Lower values (like in our example) indicate the presence of partial overlaps and wrong assignments. Negative values must be absolutely avoided (or limited to a very small number of samples) because they show a failure in the clustering process. Moreover, it's possible to prove that convex clusters (like K-means hyperspheres) lead to higher values. This is due to the properties of the commons distance functions (like the Euclidean distance) that can suggest a low internal cohesion whenever the shape of a cluster is concave (think about a circle and a half-moon). In this case, the process of embedding the shape into a convex geometry leads to a lower density and this negatively affects the Silhouette score.

In our particular case, we cannot accept having a number of clusters different from ten. However, the corresponding Silhouette plot is not perfect. We know the reasons for such imperfections (the structure of the samples and the high similarity of different digits) and it's quite difficult to avoid them using an algorithm like K-means. The reader can try to improve the performances by increasing the number of iterations, but in these cases, if the result doesn't meet the requirements, it's preferable to adopt another method (like the spectral clustering method, which can manage asymmetric clusters and more complex geometries).

Fuzzy C-means

We have already talked about the difference between hard and soft clustering, comparing K-means with Gaussian mixtures. Another way to address this problem is based on the concept of **fuzzy logic**, which was proposed for the first time by Lotfi Zadeh in 1965 (for further details, a very good reference is *An Introduction to Fuzzy Sets*, *Pedrycz W.*, *Gomide F.*, *The MIT Press*). Classic logic sets are based on the law of excluded middle that, in a clustering scenario, can be expressed by saying that a sample x_i can belong only to a single cluster c_j. Speaking more generally, if we split our universe into labeled partitions, a hard clustering approach will assign a label to each sample, while a fuzzy (or soft) approach allows managing a membership degree (in Gaussian mixtures, this is an actual probability), w_{ij} which expresses how strong the relationship is between sample x_i and cluster c_j. Contrary to other methods, by employing fuzzy logic it's possible to define asymmetric sets that are not representable with continuous functions (such as trapezoids). This allows for achieving further flexibility and an increased ability to adapt to more complex geometries. In the following graph, there's an example of fuzzy sets:



Example of fuzzy sets representing the seniority level of an employee according to years of experience

The graph represents the seniority level of an employee given his/her years of experience. As we want to cluster the entire population into three groups (**Junior**, **Middle level**, and **Senior**), three fuzzy sets have been designed. We have assumed that a young employee is keen and can quickly reach a **Junior** level after an initial apprenticeship period. The possibility to work with complex problems allows him/her to develop skills that are fundamental to allowing the transition between the **Junior** and **Middle** levels. After about **10** years, the employee can begin to consider himself/herself as a s*enior apprentice* and, after about 25 years, the experience is enough to qualify him/her as a full **Senior** until the end of his/her career. As this is an imaginary example, we haven't tuned all the values up, but it's easy to compare, for example, employee A with 9 years of experience with another employee B with 18 years of experience. The former is about 50% **Junior** (decreasing), 90% **Middle level** (reaching its climax), and 10% **Senior** (increasing). The latter, instead, is 0% **Junior** (ending plateau), 30% **Middle level** (decreasing), and 60% **Senior** (increasing). In both cases, the values are not normalized so always sum up to 1 because we are more interested in showing the process and the proportions. The fuzziness level is lower in extreme cases, while it becomes higher when two sets intersect. For example, at about 15%, the **Middle level** and **Senior** are about 50%. As we're going to discuss, it's useful to avoid a very high fuzziness when clustering a dataset because it can lead to a lack of precision as the boundaries *fade out*, becoming completely fuzzy.

Fuzzy C-means is a generalization of a standard K-means, with a soft assignment and more *flexible* clusters. The dataset to cluster (containing *M* samples) is represented by:

$$X = \{\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_M\} \;\; where \;\; \bar{x}_i \in \mathbb{R}^N$$

If we assume we have *k* clusters, it's necessary to define a matrix $W \in \Re^{M \times k}$ containing the membership degrees for each sample:

$$W = \begin{pmatrix} w_{11} & \cdots & w_{1k} \\ \vdots & \ddots & \vdots \\ w_{M1} & \cdots & w_{Mk} \end{pmatrix}$$

Each degree $w_{ij} \in [0, 1]$ and all rows must be normalized so that they always sum up to *1*. In this way, the membership degrees can be considered as probabilities (with the same semantics) and it's easier to make decisions with a prediction result. If a hard assignment is needed, it's possible to employ the same approach normally used with Gaussian mixtures: the winning cluster is selected by applying the *argmax* function. However, it's a good practice to employ soft clustering only when it's possible to manage the vectorial output. For example, the probabilities/membership degrees can be fed into a classifier in order to yield more complex predictions.

As with K-means, the problem can be expressed as the minimization of a *generalized inertia*:

$$S_f = \sum_{j=1}^{k} \sum_{\bar{x}_i \in C_j} w_{ij}^m \left\| \bar{x}_i - \bar{\mu}_j \right\|^2$$

The constant m $(m > 1)$ is an exponent employed to re-weight the membership degrees. A value very close to *1* doesn't affect the actual values. Greater m values reduce their magnitude. The same parameter is also used when recomputing the centroids and the new membership degrees and can drive to a different clustering result. It's rather difficult to define a global acceptable value; therefore, a good practice is to start with an average m (for example, 1.5) and perform a grid search (it's possible to sample from a Gaussian or uniform distribution) until the desired accuracy has been achieved.

Minimizing the previous expression is even more difficult than with a standard inertia; therefore, a *pseudo-Lloyd's algorithm* is employed. After a random initialization, the algorithm proceeds, alternating two steps (like an EM procedure) in order to determine the centroids, and recomputing the membership degrees to maximize the internal cohesion. The centroids are determined by a weighted average:

$$\bar{\mu}_j = \frac{\sum_{i=1}^{M} w_{ij}^m \ \bar{x}_i}{\sum_{i=1}^{M} w_{ij}^m}$$

Contrary to K-means, the sum is not limited to the points belonging to a specific cluster because the weight factor will force the farthest points ($w_{ij} \approx 0.0$) to produce a contribution close to *0*. At the same time, as this is a soft-clustering algorithm, no exclusions are imposed, to allow a sample to belong to any number of clusters with different membership degrees. Once the centroids have been recomputed, the membership degrees must be updated using this formula:

$$w_{ij} = \frac{1}{\sum_{p=1}^{k} \left(\frac{\left\| \bar{x}_i - \bar{\mu}_j \right\|}{\left\| \bar{x}_i - \bar{\mu}_p \right\|} \right)^{\frac{2}{m-1}}}$$

This function behaves like a similarity. In fact, when sample x_i is very close to centroid μ_j (and relatively far from μ_p with $p \neq j$), the denominator becomes small and w_{ij} increases. The exponent *m* directly influences the fuzzy partitioning, because when $m \approx 1$ $(m > 1)$, the denominator is a sum of *quasi*-squared terms and the closest centroid can dominate the sum, yielding to a higher preference for a specific cluster. When $m >> 1$, all the terms in the sum tend to *1*, producing a more flat weight distribution with no well-defined preference. It's important to understand that, even when working with soft clustering, a fuzziness excess leads to inaccurate decisions because there are no factors that push a sample to clearly belong to a specific cluster. This means that problem is either ill-posed or, for example, the number of expected clusters is too high and doesn't represent the real underlying data structure. A good way to measure how much this algorithm is similar to a hard-clustering approach (such as K-means) is provided by the normalized **Dunn's partitioning coefficient**:

$$P_C = \frac{w_c - \frac{1}{k}}{1 - \frac{1}{k}} \quad where \;\; w_c = \frac{1}{M} \sum_{i=1}^{M} \sum_{j=1}^{k} w_{ij}^2$$

When P_c is bounded between *0* and *1*, when it's close to *0*, it means that the membership degrees have a flat distribution and the level of fuzziness is the highest possible. On the other side, if it's close to *1*, each row of *W* has a single dominant value, while all the others are negligible. This scenario resembles a hard-clustering approach. Higher P_c values are normally preferable because, even without renouncing to a degree of fuzziness, it allows making more precise decisions. Considering the previous example, P_c tends to *1* when the sets don't intersect, while it becomes 0 (complete fuzziness) if, for example, the three seniority levels are chosen to be identical and overlapping. Of course, we are interested in avoiding such extreme scenarios by limiting the number of borderline cases. A grid search can be performed by analyzing different numbers of clusters and *m* values (in the example, we're going to do it with the MNIST handwritten digit dataset). A reasonable rule of thumb is to accept P_c values higher than *0.8*, but in some cases, that can be impossible. If we are sure that the problem is well-posed, the best approach is to choose the configuration that maximizes P_c, considering, however, that a final value less than *0.3-0.5* will lead to a very high level of uncertainty because the clusters are extremely overlapping.

The complete **Fuzzy C-means** algorithm is:

1. Set a maximum number of iteration N_{max}
2. Set a tolerance *Thr*
3. Set the value of *k* (number of expected clusters)
4. Initialize the matrix $W^{(0)}$ with random values and normalize each row, dividing it by its sum
5. Set $N = 0$
6. While $N < N_{max}$ or $||W^{(t)} - W^{(t-1)}|| > Thr$:
    1. $N = N + 1$
    2. For $j = 1$ to k:
        1. Compute the centroid vectors μ_j
    3. Recompute the weight matrix $W^{(t)}$
    4. Normalize the rows of $W^{(t)}$

Example of fuzzy C-means with Scikit-Fuzzy

Scikit-Fuzzy (`http://pythonhosted.org/scikit-fuzzy/`) is a Python package based on SciPy that allows implementing all the most important fuzzy logic algorithms (including fuzzy C-means). In this example, we continue using the MNIST dataset, but with a major focus on fuzzy partitioning. To perform the clustering, Scikit-Fuzzy implements the `cmeans` method (in the `skfuzzy.cluster` package) which requires a few mandatory parameters: `data`, which must be an array $D \in \Re^{N \times M}$ (N is the number of features; therefore, the array used with Scikit-Learn must be transposed); `c`, the number of clusters; the coefficient `m`, `error`, which is the maximum tolerance; and `maxiter`, which is the maximum number of iterations. Another useful parameter (not mandatory) is the `seed` parameter which allows specifying the random seed to be able to easily reproduce the experiments. I invite the reader to check the official documentation for further information.

The first step of this example is performing the clustering:

```
from skfuzzy.cluster import cmeans

fc, W, _, _, _, _, pc = cmeans(X_train.T, c=10, m=1.25, error=1e-6,
maxiter=10000, seed=1000)
```

The `cmeans` function returns many values, but for our purposes, the most important are: the first one, which is the array containing the cluster centroids; the second one, which is the final membership degree matrix; and the last one, the partition coefficient. In order to analyze the result, we can start with the partition coefficient:

```
print(pc)
0.632070870735
```

This value informs us that the clustering is not very far from a hard assignment, but there's still a residual fuzziness. In this particular case, such a situation may be reasonable because we know that many digits are partially distorted and may appear very similar to other ones (such as 1, 7, and 9). However, I invite the reader to try different values for `m` and check how the partition coefficient changes. We can now display the centroids:

Centroids obtained by fuzzy C-means

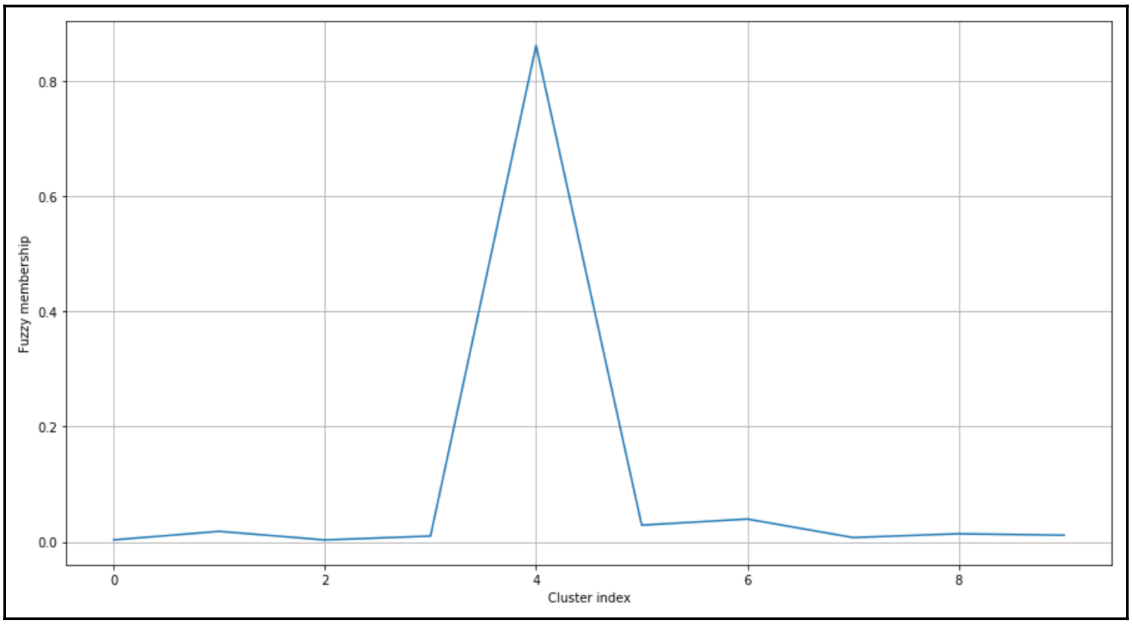
All the different digit classes have been successfully found, but now, contrary to K-means, we can check the fuzziness of a *problematic* digit (representing a 7, with index 7), as shown in the following diagram:

Sample digit (a 7) selected to test the fuzziness

The membership degrees associated with the previous sample are:

```
print(W[:, 7])
[ 0.00373221  0.01850326  0.00361638  0.01032591  0.86078292  0.02926149
  0.03983662  0.00779066  0.01432076  0.0118298 ]
```

The corresponding plot is:



Fuzzy membership plot corresponding to a digit representing a 7

In this case, the choice of *m* has forced the algorithm to reduce the fuzziness. However, it's still possible to see three smaller peaks corresponding to the clusters centered respectively on 1, 8, and 5 (remember that the cluster indexes correspond to digits shown previously in the centroid plot). I invite the reader to analyze the fuzzy partitioning of different digits and replot it with different values of the `m` parameter. It will be possible to observe an increased fuzziness (corresponding also to smaller partitioning coefficients) with larger *m* values. This effect is due to a stronger overlap among clusters (observable also by plotting the centroids) and could be useful when it's necessary to detect the distortion of a sample. In fact, even if the main peak indicates the right cluster, the secondary ones, in descending order, inform us how much the sample is similar to other centroids and, therefore, if it contains features that are characteristics of other subsets.

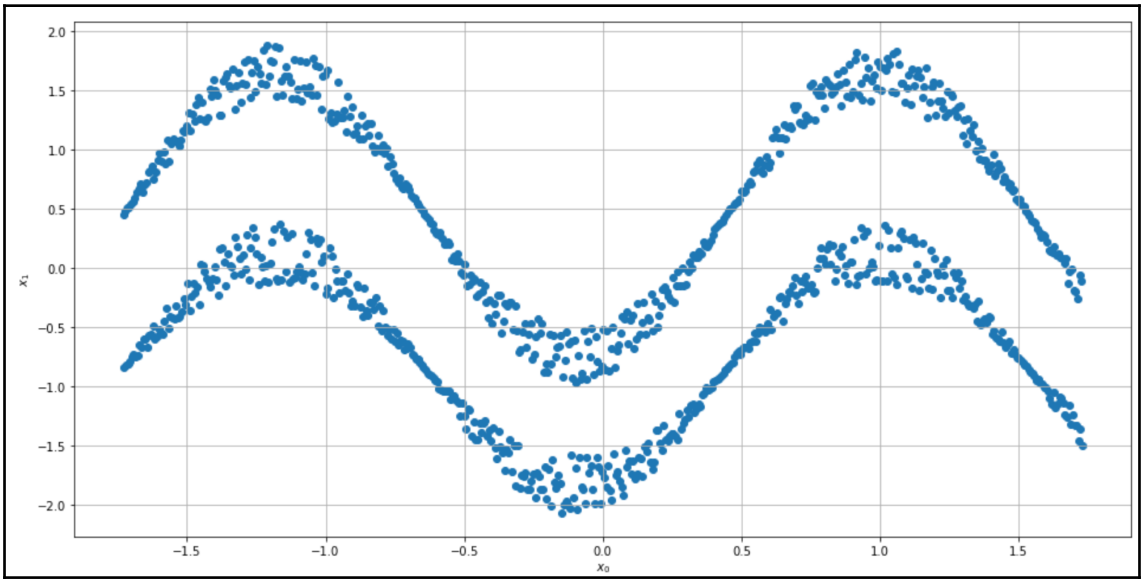
Contrary to Scikit-Learn, in order to perform predictions, Scikit-Fuzzy implements the `cmeans_predict` method (in the same package), which requires the same parameters of `cmeans`, but instead of the number of clusters, `c` needs the final centroid array (the name of the parameter is `cntr_trained`). The function returns as a first value the corresponding membership degree matrix (the other ones are the same as `cmeans`). In the following snippet, we repeat the prediction for the same sample digit (representing a `7`):

```
import numpy as np

from skfuzzy.cluster import cmeans_predict

new_sample = np.expand_dims(X_train[7], axis=1)
Wn, _, _, _, _, _ = cmeans_predict(new_sample, cntr_trained=fc, m=1.25,
error=1e-6, maxiter=10000, seed=1000)

print(Wn.T)
[[ 0.00373221  0.01850326  0.00361638  0.01032591  0.86078292  0.02926149
   0.03983662  0.00779066  0.01432076  0.0118298 ]]
```

Scikit-Fuzzy can be installed using the `pip install -U scikit-fuzzy` command. For further instructions, please visit `http://pythonhosted.org/scikit-fuzzy/install.html`

Spectral clustering

One of the most common problems of K-means and other similar algorithms is the assumption we have only hyperspherical clusters. This condition can be acceptable when the dataset is split into blobs that can be easily embedded into a regular geometric structure. However, it fails whenever the sets are not separable using regular shapes. Let's consider, for example, the following bidimensional dataset:



Sinusoidal dataset

As we are going to see in the example, any attempt to separate the upper sinusoid from the lower one using K-means will fail. The reason is quite obvious: a circle that contains the upper set will also contain part of the (or the whole) lower set. Considering the criterion adopted by K-means and imposing two clusters, the inertia will be minimized by a vertical separation corresponding to about $x_0 = 0$. Therefore, the resulting clusters are completely mixed and only a dimension is contributing to the final configuration. However, the two sinusoidal sets are well-separated and it's not difficult to check that, selecting a point x_i from the lower set, it's always possible to find a ball containing only samples belonging to the same set. We have already discussed this kind of problem when Label Propagation algorithms were discussed and the logic behind **spectral clustering** is essentially the same (for further details, I invite the reader to check `Chapter 2`, *Graph-Based Semi-Supervised Learning*).

Let's suppose we have a dataset *X* sampled from a data generating process p_{data}:

$$X = \{\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_M\} \;\; where \;\; \bar{x}_i \in \mathbb{R}^N$$

We can build a graph *G = {V, E}*, where the vertices are the points and the edges are determined using an *affinity matrix W*. Each element w_{ij} must express the affinity between sample x_i and sample x_j. *W* is normally built using two different approaches:

- KNN: In this case, we can build the number of neighbors to take into account for each point x_i. *W* can be built as a *connectivity matrix* (expressing only the existence of a connection between two samples) if we adopt the criterion:

$$w_{ij} = \begin{cases} 1 \;\; if \;\; \bar{x}_j \in kNN(\bar{x}_i) \\ 0 \;\; otherwise \end{cases}$$

  Alternatively, it's possible to build a *distance matrix*:

$$w_{ij} = \begin{cases} d(\bar{x}_i, \bar{x}_j) \;\; if \;\; \bar{x}_j \in kNN(\bar{x}_i) \\ 0 \;\; otherwise \end{cases}$$

- **Radial basis function** (**RBF**): The previous methods can lead to graphs which are not fully connected because samples can exist that have no neighbors. In order to obtain a fully connected graph, it's possible to employ an RBF (this approach has also been used in the Kohonen map algorithm):

$$w_{ij} = e^{-\gamma \|\bar{x}_i - \bar{x}_j\|^2}$$

  The γ parameter allows controlling the amplitude of the Gaussian function, reducing or increasing the number of samples with a high weight (so *actual neighbors*). However, a weight is assigned to all points and the resulting graph will always be connected (even if many elements are close to zero).

In both cases, the elements of *W* will represent a measure of affinity (or *closeness*) between points and no restrictions are imposed on the global geometry (contrary to K-means). In particular, using a KNN connectivity matrix, we are implicitly segmenting the original dataset into smaller regions with a high level of internal cohesion. The problem that we need to solve now is to find out a way to merge all the regions belonging to the same cluster. The approach we are going to present here has been proposed by *Normalized Cuts and Image Segmentation, J. Shi* and *J. Malik, IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. 22, 08/2000*, and it's based on the normalized graph Laplacian:

$$L_n = I - D^{-1}W$$

The matrix *D*, called the degree matrix, is the same as discussed in `Chapter 3`, *Graph-Based Semi-Supervised Learning* and it's defined as:

$$D = diag\left(\left[\sum_j w_{ij}\right] \forall\, i \in [1, M]\right)$$

It's possible to prove the following properties (the formal proofs are omitted but they can be found in texts such as *Functions and Graphs Vol. 2, Gelfand I. M., Glagoleva E. G., Shnol E. E., The MIT Press*:

- The eigenvalues λ_i and the eigenvectors v_i of L_n can be found by solving the problem $Lv = \lambda Dv$, where *L* is the unnormalized graph Laplacian $L = D - W$
- L_n always has an eigenvalue equal to *0* (with a multiplicity *k*) with a corresponding eigenvector $v_o = (1, 1, ..., 1)$
- As *G* is undirected and all $w_{ij} \geq 0$, the number of connected components *k* of *G* is equal to the multiplicity of the null eigenvalue

In other words, the normalized graph Laplacian encodes the information about the number of connected components and provides us with a new reference system where the clusters can be separated using regular geometric shapes (normally hyperspheres). To better understand how this approach works without a non-trivial mathematical approach, it's important to expose another property of L_n.

From linear algebra, we know that each eigenvalue λ of a matrix $M \in \Re^{n \times n}$ spans a corresponding eigenspace, which is a subset of $\Re^n$ containing all eigenvectors associated with λ plus the null vector. Moreover, given a set $S \subseteq \Re n$ and a countable subset C (it's possible to extend the definition to generic subsets but in our context the datasets are always countable), we can define a vector $v \in \Re^n$ as an *indicator vector*, if $v^{(i)} = 1$ if the vector $c_i \in S$ and $v^{(i)} = 0$ otherwise. If we consider the null eigenvalues of L_n and we assume that their number is k (corresponding to the multiplicity of the eigenvalue *0*), it's possible to prove that the corresponding eigenvectors are indicator vectors for eigenspaces spanned by each of them. From the previous statements, we know that these eigenspaces correspond to the connected components of the graph G; therefore, performing a standard clustering (like K-means or K-means++) with the points projected into these subspaces allows for an easy separation with symmetric shapes.

As $L_n \in \Re^{M \times M}$, its eigenvectors $v_i \in \Re^M$. Selecting the first k eigenvectors, it's possible to build a matrix $A \in \Re^{M \times k}$:

$$A = \begin{pmatrix} v_1^{(1)} & v_2^{(1)} & \cdots & v_k^{(1)} \\ v_1^{(2)} & v_2^{(2)} & \cdots & v_1^{(k)} \\ \vdots & \vdots & \ddots & \vdots \\ v_1^{(M)} & v_2^{(M)} & \cdots & v_k^{(M)} \end{pmatrix}$$

Each row of A, $a_j \in \Re^k$ can be considered as the projection of an original sample x_j in the low-dimensional subspace spanned by each of the null eigenvalues of L_n. At this point, the separability of the new dataset $A = \{a_j\}$ depends only on the structure of the graph G and, in particular, on the number of neighbors or the γ parameter for RBFs. As in many other similar cases, it's impossible to define a standard value suitable for all problems, above all when the dimensionality doesn't allow a visual inspection. A reasonable approach should start with a small number of neighbors (for example, five) or $\gamma = 1.0$ and increase the values until a performance metric (such as the Adjusted Rand Index) reaches its maximum. Considering the nature of the problems, it can also be useful to measure the homogeneity and the completeness because these two measures are more sensitive to irregular geometric structures and can easily show when the clustering is not separating the sets correctly. If the ground truth is unknown, the Silhouette score can be employed to assess the intra-cluster cohesion and the inter-cluster separation as functions of all hyperparameters (number of clusters, number of neighbors, or γ).

The complete **Shi-Malik spectral clustering** algorithm is:

1. Select a graph construction a method between KNN (1) and RBF (2):
    1. Select parameter k
    2. Select parameter γ
2. Select the expected number of clusters N_k.
3. Compute the matrices W and D.
4. Compute the normalized graph Laplacian L_n.
5. Compute the first k eigenvectors of L_n.
6. Build the matrix A.
7. Cluster the rows of A using K-means++ (or any other symmetric algorithm). The output of this process is this set of clusters: $C_{km}^{(1)}$, $C_{km}^{(2)}$, ..., $C_{km}^{(Nk)}$.

Example of spectral clustering with Scikit-Learn

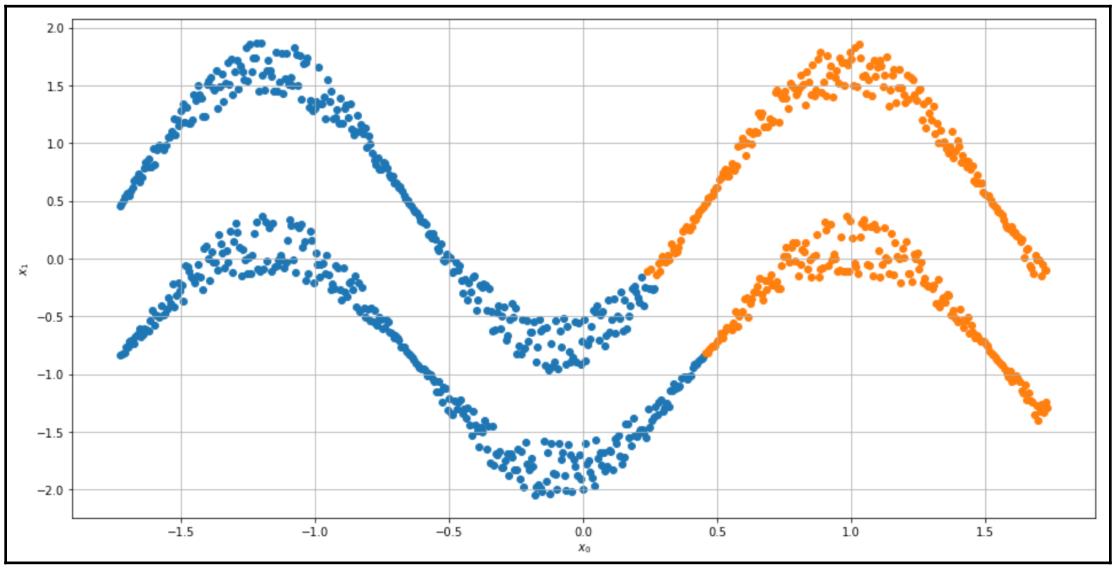
In this example, we are going to use the sinusoidal dataset previously shown. The first step is creating it (with 1,000 samples):

```
import numpy as np

from sklearn.preprocessing import StandardScaler

nb_samples = 1000

X = np.zeros(shape=(nb_samples, 2))

for i in range(nb_samples):
    X[i, 0] = float(i)
    if i % 2 == 0:
        X[i, 1] = 1.0 + (np.random.uniform(0.65, 1.0) * np.sin(float(i) /
100.0))
    else:
        X[i, 1] = 0.1 + (np.random.uniform(0.5, 0.85) * np.sin(float(i) /
100.0))
ss = StandardScaler()
Xs = ss.fit_transform(X)
```

At this point, we can try to cluster it using K-means (with `n_clusters=2`):

```
from sklearn.cluster import KMeans

km = KMeans(n_clusters=2, random_state=1000)
Y_km = km.fit_predict(Xs)
```

The result is shown in the following graph:

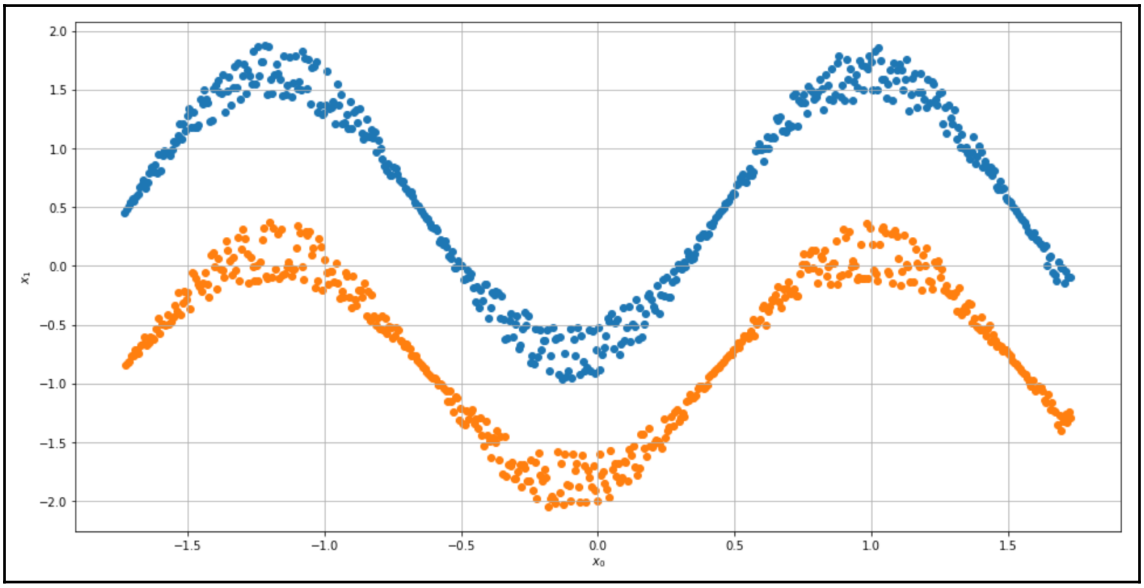


K-means clustering result using the sinusoidal dataset

As expected, K-means isn't able to separate the two sinusoids. The reader is free to try with different parameters, but the result will always be unacceptable because K-means bidimensional clusters are circles and no valid configurations exist. We can now employ spectral clustering using an affinity matrix based on the KNN algorithm (in this case, Scikit-Learn can produce a warning because the graph is not fully connected, but this normally doesn't affect the results). Scikit-Learn implements the `SpectralClustering` class, whose most important parameters are `n_clusters`, the number of expected clusters; `affinity`, which can be either `'rbf'` or `'nearest_neighbors'`; `gamma` (only for RBF); and `n_neighbors` (only for KNN). For our test, we have chosen to have `20` neighbors:

```
from sklearn.cluster import SpectralClustering

sc = SpectralClustering(n_clusters=2, affinity='nearest_neighbors',
n_neighbors=20, random_state=1000)
Y_sc = sc.fit_predict(Xs)
```

The result of the spectral clustering is shown in the following graph:



Spectral clustering result using the sinusoidal dataset

As expected, the algorithm was able to separate the two sinusoids perfectly. As an exercise, I invite the reader to apply this method to the MNIST dataset, using both an RBF (with different gamma values) and KNN (with different numbers of neighbors). I also suggest to replot the t-SNE diagram and compare all the assignment errors. As the clusters are strictly non-convex, we don't expect a high Silhouette score. Other useful exercises can be: drawing the Silhouette plot and checking the result, assigning ground truth labels, and measuring the homogeneity and the completeness.

Summary

In this chapter, we presented some fundamental clustering algorithms. We started with KNN, which is an instance-based method that restructures the dataset to find the most similar samples given a query point. We discussed three approaches: a naive one, which is also the most expensive in terms of computational complexity, and two strategies based respectively on the construction of a KD Tree and a Ball Tree. These two data structures can dramatically improve performance even when the number of samples is very large.

The next topic was a classic algorithm: K-means, which is a symmetric partitioning strategy, comparable to a Gaussian mixture with variances close to zero, that can solve many real-life problems. We discussed both a vanilla algorithm, which wasn't able to find a valid sub-optimal solution, and an optimized initialization method, called K-means++, which was able to speed up the convergence towards solutions quite close to the global minimum. In the same section, we also presented some evaluation methods that can be employed to assess the performance of a generic clustering algorithm.

We also presented a soft-clustering method called fuzzy C-means, which resembles the structure of a standard K-means, but allows managing membership degrees (analogous to probabilities) that encode the similarity of a sample with all cluster centroids. This kind of approach allows processing the membership vectors in a more complex pipeline, where the output of a clustering process, for example, is fed into a classifier.

One of the most important limitations of K-means and similar algorithms is the symmetric structure of the clusters. This problem can be solved with methods such as spectral clustering, which is a very powerful approach based on the dataset graph and is quite similar to non-linear dimensionality reduction methods. We analyzed an algorithm proposed by Shi and Malik, showing how it can easily separate a non-convex dataset.

In the next chapter, `Chapter 8`, *Ensemble Learning*, we're going to discuss some common ensemble learning methods, which are based on the use of a large set of weak classifiers. We focused on their peculiarities, comparing the performances of different ensembles with single strong classifiers.

8
Ensemble Learning

In this chapter, we are going to discuss some important algorithms that exploit different estimators to improve the overall performance of an ensemble or committee. These techniques work either by introducing a medium level of randomness in every estimator belonging to a predefined set or by creating a sequence of estimators where, each new model is forced to improve the performance of the previous ones. These techniques allow us to reduce both the bias and the variance (thereby increasing validation accuracy) when employing models with a limited capacity or more prone to overfit the training set.

In particular, the topics covered in the chapter are as follows:

- Introduction to ensemble learning
- A brief introduction to decision trees
- Random forest and extra randomized forests
- AdaBoost (algorithms M1, SAMME, SAMME.R, and R2)
- Gradient boosting
- Ensembles of voting classifiers, stacking, and bucketing

Ensemble learning fundamentals

The main concept behind ensemble learning is the distinction between strong and weak learners. In particular, a strong learner is a classifier or a regressor which has enough capacity to reach the highest potential accuracy, minimizing both bias and variance (thus achieving also a satisfactory level of generalization). More formally, if we consider a parametrized binary classifier *f(x; θ)*, we define it as a strong learner if the following is true:

$$\forall \epsilon > 0 \;\; and \;\; \delta \leqslant \frac{1}{2} \;\; \exists\, \bar{\theta} : with\, p \geqslant 1 - \delta \;\Rightarrow\; p[f(\bar{x}_i, \bar{\theta}) \neq y_i] \leqslant \epsilon$$

This expression can appear cryptic; however, it's very easy to understand. It simply expresses the concept that a strong learner is theoretically able to achieve any non-null probability of misclassification with a probability greater than or equal to 0.5 (that is, the threshold for a binary random guess). All the models normally employed in Machine Learning tasks are normally strong learners, even if their domain can be limited (for example, a logistic regression cannot solve non-linear problems). On the other hand, a weak learner is a model that is generically able to achieve an accuracy slightly higher than a random guess, but whose complexity is very low (they can be trained very quickly, but can never be used alone to solve complex problems). There is a formal definition also in this case, but it's simpler to consider that the real main property of a weak learner is a limited ability to achieve a reasonable accuracy. In some very particular and small regions of the training space, a weak learner could reach a low probability of misclassification, but in the whole space its performance is only a little bit superior to a random guess. The previous one is more a theoretical definition than a practice one, because all the models currently available are normally quite better than a random oracle. However, an ensemble is defined as a set of weak learners that are trained together (or in a sequence) to make up a committee. Both in classification and regression problems, the final result is obtained by averaging the predictions or employing a majority vote.

At this point, a reasonable question is—Why do we need to train many weak learners instead of a single strong one? The answer is double—in ensemble learning, we normally work with medium-strong learners (such as decision trees or **support vector machines** (**SVMs**)) and we use them as a committee to increase the overall accuracy and reduce the variance thanks to a wider exploration of the sample space. In fact, while a single strong learner is often able to overfit the training set, it's more difficult to keep a high accuracy over the whole sample subspace without saturating the capacity. In order to avoid overfitting, a trade-off must be found and the result is a less accurate classifier/regressor with a simpler separation hyperplane. The adoption of many weak learners (that are actually quite strong, because even the simplest models are more accurate than a random guess), allows us to force them to focus only on a limited subspace, so as to be able to reach a very high local accuracy with a low variance. The committee, employing an averaging technique, can easily find out which prediction is the most suitable. Alternatively, it can ask each learner to vote, assuming that a successful training process must always lead the majority to propose the most accurate classification or prediction.

The most common approaches to ensemble learning are as follows:

- **Bagging (bootstrap aggregating)**: This approach trains *n* weak learners *fw1*, *fw2*, ..., *fwn* (very often they are decision trees) using *n* training sets (*D1*, *D2*, ..., *Dn*) created by randomly sampling the original dataset *D*. The sampling process (called **bootstrap sampling**) is normally performed with replacement, so as to determine different data distributions. Moreover, in many real algorithms, the weak learners are also initialized and trained using a medium degree of randomness. In this way, the probability of having clones becomes very small and, at the same, time it's possible to increase the accuracy by keeping the variance under a tolerable threshold (thus avoiding overfitting).
- **Boosting**: This is an alternative approach that builds an incremental ensemble starting with a single weak learner *fw1* and adding a new one *fwi* at each iteration. The goal is to reweight the dataset, so as to force the new learner to focus on the samples that were previously misclassified. This strategy yields a very high accuracy because the new learners are trained with a positively-biased dataset that allows them to adapt to the most difficult internal conditions. However, in this way, the control over the variance is weakened and the ensemble can more easily overfit the training set. It's possible to mitigate this problem by reducing the complexity of the weak learners or imposing a regularization constraint.
- **Stacking**: This method can be implemented in different ways but the philosophy is always the same—use different algorithms (normally a few strong learners) trained on the same dataset and filter the final result using another classifier, averaging the predictions or using a majority vote. This strategy can be very powerful if the dataset has a structure that can be partially managed with different approaches. Each classifier or regressor should discover some data aspects that are peculiar; that's why the algorithms must be structurally different. For example, it can be useful to mix a decision tree with a SVM or linear and kernel models. The evaluation performed on the test set should clearly show the prevalence of a classifier only in some cases. If an algorithm is finally the only one that produces the best prediction, the ensemble becomes useless and it's better to focus on a single strong learner.

Random forests

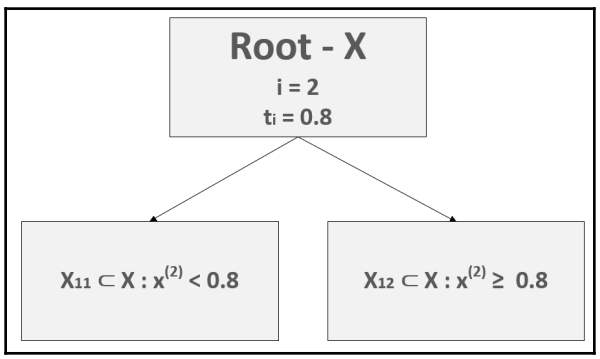
A random forest is the bagging ensemble model based on decision trees. If the reader is not familiar with this kind of model, I suggest reading the *Introduction to Machine Learning*, *Alpaydin E., The MIT Press*, where a complete explanation can be found. However, for our purposes, it's useful to provide a brief explanation of the most important concepts. A decision tree is a model that resembles a standard hierarchical decision process. In the majority of cases, a special family is employed, called binary decision trees, as each decision yields only two outcomes. This kind of tree is often the simplest and most reasonable choice and the training process (which consists in building the tree itself) is very intuitive. The root contains the whole dataset:

$$X = \{\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_M\} \;\; where\ \bar{x}_i \in \mathbb{R}^n$$

Each level is obtained by applying a selection tuple, defined as follows:

$$\sigma_i = \langle i, t_i \rangle \;\; where\ i \in [1, n] \;\; and\ t_i \in \left[min\left(x^{(i)}\right), max\left(x^{(i)}\right) \right]$$

The first index of the tuple corresponds to an input feature, while the threshold t_i is a value chosen in the specific range of each feature. The application of a selection tuple leads to a split and two nodes that contain each a non-overlapping subset of the input dataset. In the following diagram, there's an example of a slip performed at the level of the root (initial split):



Example of initial split in a decision tree

The set *X* is split into two subsets defined as *X11* and *X12* whose samples have respectively the feature with *i=2* less or greater than the threshold *ti=0.8*. The intuition behind classification decision trees is to continue splitting until the leaves contain samples belonging to a single category *yi* (these nodes are defined as pure). In this way, a new sample *xj* can traverse the tree with a computation complexity *O(log(M))* and reach a final node that determines its category. In a very similar way, it's possible to build regression trees whose output is continuous (even if, for our purposes, we are going to consider only classification scenarios).

At this point, the main problem is how to perform each split. We cannot pick any feature and any threshold, because the final tree will be completely unbalanced and very deep. Our goal is to find the optimal selection tuple at each node considering the final goal, which is classification into discrete categories (the process is almost identical for regressions). The technique is very similar to a problem based on a cost function that must be minimized, but, in this case, we operate locally, applying an impurity measure proportional to the heterogeneity of a node. A high impurity indicates that samples belonging to many different categories are present, while an impurity equal to 0 indicates that a single category is present. As we need to continue splitting until a pure leaf appears, the optimal choice is based on a function that scores each selection tuple, allowing us to select the one that yields the lowest impurity (theoretically, the process should continue until all the leaves are pure, but normally a maximum depth is provided, so as to avoid excessive complexity). If there are p classes, the category set can be defined as follows:

$$Y = \{y_1, y_2, \ldots, y_M\} \quad where\ y_i \in [1, p]$$

A very common impurity measure is called **Gini impurity** and it's based on the probability of a misclassification if a sample is categorized using a label randomly chosen from the node subset distribution. Intuitively, if all the samples belong to the same category, any random choice leads to a correct classification (and the impurity becomes *0*). On the other side, if the node contains samples from many categories, the probability of a misclassification increases. Formally, the measure is defined as follows:

$$I_{Gini}(X_k) = \sum_{j=1}^{p} p(j|k)\,(1 - p(j|k))$$

The subset is indicated by *Xk* and *p(j|k)* is obtained as the ratio of the samples belonging to the class *j* over the total number of samples. The selection tuple must be chosen so as to minimize the Gini impurity of the children. Another common approach is the cross-entropy impurity, defined as follows:

$$I_{Cross-Entropy}(X_k) = -\sum_{j=1}^{p} p(j|k)\ log\ p(j|k)$$

The main difference between this measure and the previous one is provided by some fundamental information theory concepts. In particular, the goal we want to reach is the minimization of the uncertainty, which is measured using the *(Cross-)Entropy*. If we have a discrete distribution and all the samples belong to the same category, a random choice is can fully describe the distribution; therefore, the uncertainty is null. On the contrary, if, for example, we have a fair die, the probability of each outcome is 1/6 and the corresponding entropy is about 2.58 bits (if the base of the logarithm is 2). When the nodes become purer and purer, the cross-entropy impurity decreases and reaches 0 in an optimal scenario. Moreover, adopting the concept of mutual information, we can define the information gain obtained after a split has been performed:

$$IG(\sigma) = H(X_{parent}) - H(X_{parent}|X_{children})$$

Given a node, we want to create two children to maximize the information gain. In other words, by choosing the cross-entropy impurity we implicitly grow the tree until the information gain becomes null. Considering again the example of a fair die, we need 2.58 bits of information to decide which is the right outcome. If, instead, the die is loaded and the probability of an outcome is 1.0, we need no information to make a decision. In a decision tree, we'd like to resemble this situation, so that, when a new sample has completely traversed the tree, we don't need any further information to classify it. If a maximum depth is imposed, the final information gain cannot be null. This means that we need to pay an extra cost to finalize the classification. This cost is proportional to the residual uncertainty and should be minimized to increase the accuracy.

Other methods can also be employed (even if Gini and cross-entropy are the most common) and I invite the reader to check the references for further information. However, at this point, a consideration naturally arises. Decision trees are simple models (they are not weak learners!), but the procedure for building them is more complex than, for example, training a logistic regression or a SVM. Why are they so popular? One reason is already clear—they represent a structural process that can be shown using a diagram; however, this is not enough to justify their usage. Two important properties allow the employment of decision trees without any data preprocessing.

In fact, it's easy to understand that, contrary to other methods, there's no need for any scaling or whitening and it's possible to use continuous and categorical features at the same time. For example, if in a bidimensional dataset a feature has a variance equal to 1 and the other equal to 100, the majority of classifiers will achieve a low accuracy; therefore, a preprocessing step becomes necessary. In a decision tree, a selection tuple has the same effect also when the ranges are very different. It goes without saying that a split can be easily performed considering also categorical features and there's no need, for example, to use techniques such as one-hot encoding (which is necessary in most cases to avoid generalization errors). However, unfortunately, the separation hypersurface obtained with a decision tree is normally much more complex than the one obtained using other algorithms and this drives to a higher variance with a consequential loss of generalization ability.

To understand the reason, it's possible to imagine a very simple bidimensional dataset made up of two blobs located in the second and fourth quarters. The first set is characterized by $(x < 0, y > 0)$, but the second one by $(x < 0, y < 0)$. Let's also suppose that we have a few outliers, but our knowledge about the data generating process is not enough to qualify them as noisy samples (the original distribution can have tails that are extended over the axes; for example, it may be a mixture of two Gaussians). In this scenario, the simplest separation line is a diagonal splitting the plane into two subplanes containing regions belonging also to the first and third quarters. However, this decision can be made only considering both coordinates at the same time. Using a decision tree, we need to split initially, for example, using the first feature and again with the second one. The result is a piece-wise separation line (for example, splitting the plane into the region corresponding to the second quarter and its complement), leading to a very high classification variance. Paradoxically, a better solution can be obtained with an incomplete tree (limiting the process, for example, to a single split) and with the selection of the y-axis as the separation line (this is why it's important to impose a maximum depth), but the price you pay is an increased bias (and a consequently worse accuracy).

Another important element to consider when working with decision trees (and related models) is the maximum depth. It's possible to grow the tree until the all leaves are pure, but sometimes it's preferable to impose a maximum depth (and, consequently, a maximum number of terminal nodes). A maximum depth equal to 1 drives to binary models called **decision stumps**, which don't allow any interaction among the features (they can simply be represented as *If... Then* conditions). Higher values yield more terminal nodes and allow an increasing interaction among features (it's possible to think about a combination of many *If... Then* statements together with `AND` logical operators). The right value must be tuned considering every single problem and it's important to remember that very deep trees are more prone to overfitting than pruned ones.

In some contexts, it's preferable to achieve a slightly worse accuracy with a higher generalization ability and, in those case, a maximum depth should be imposed. The common tool to determine the best value is always a grid search together with a cross-validation technique.

Random forests provide us with a powerful tool to solve the bias-variance trade-off problem. They were proposed by L. Breiman (in *Breiman L., Random Forests, Machine Learning, 45, 2001*) and their logic is very simple. As already explained in the previous section, the bagging method starts with the choice of the number of weak learners, *Nc*. The second step is the generation of Nc datasets (called bootstrap samples) *D1, D2, ..., DNc*:

$$D_p = \left\{ \bar{x}_1^s, \bar{x}_2^s, \ldots, \bar{x}_M^s \right\} \;\; where\; \bar{x}_i^s \; is\; sampled\; from\; X\; and\; p \in [1, N_c]$$

Each decision tree is trained using the corresponding dataset using a common impurity criterion; however, in a random forest, in order to reduce the variance, the selection splits are not computed considering all the features, but only via a random subset containing a quite smaller number of features (common choices are the rounded square root, log2 or natural logarithm). This approach indeed weakens each learner, as the optimality is partially lost, but allows us to obtain a drastic variance reduction by limiting the over-specialization. At the same time, a bias reduction and an increased accuracy are a result of the ensemble (in particular for a large number of estimators). In fact, as the learners are trained with slightly different data distributions, the average of a prediction converges to the right value when $Nc \rightarrow \infty$ (in practice, it's not always necessary to employ a very large number of decision trees, however, the correct value must be tuned using a grid search with cross-validation). Once all the models, represented with a function *di(x)*, have been trained, the final prediction can be obtained as an average:

$$\hat{y} = \frac{1}{N_c} \sum_{i=1}^{N_c} d_i(\bar{x})$$

Alternatively, it's possible to employ a majority vote (but only for classifications):

$$\hat{y} = argmax_{d_i(\bar{x})} \left(d_i(\bar{x}) \right)$$

These two methods are very similar and, in most cases, they yield the same result. However, averaging is more robust and allows an improved flexibility when the samples are almost on the boundaries. Moreover, it can be used for both classification and regression tasks.

Random forests limit their randomness by picking the best selection tuple from a smaller sample subset. In some cases, for example, when the number of features is not very large, this strategy drives to a minimum variance reduction and the computational cost is no longer justified by the result. It's possible to achieve better performances with a variant called extra-randomized trees (or simply extra-trees). The procedure is almost the same; however, in this case, before performing a split, *n* random thresholds are computed (for each feature) and the one which leads to the least impurity is chosen. This approach further weakens the learners but, at the same time, reduces residual variance and prevents overfitting. The dynamic is not very different from many techniques such as regularization or dropout (we're going to discuss this approach in the next chapter); in fact, the extra-randomness reduces the capacity of the model, forcing it to a more linearized solution (which is clearly sub-optimal). The price to pay for this limitation is a consequent bias worsening, which, however, is compensated by the presence of many different learners. Even with random splits, when *Nc* is large enough, the probability of a wrong classification (or regression prediction) becomes smaller and smaller because both the average and the majority vote tend to compensate the outcome of trees whose structure is strongly sub-optimal in particular regions. This result is easier to obtain, in particular, when the number of training samples is large. In this case, in fact, sampling with replacement leads to slightly different distributions that could be considered (even if this is not formally correct) as partially and randomly boosted. Therefore, every weak learner will implicitly focus on the whole dataset with extra-attention to a smaller subset that, however, is randomly selected (differently from actual boosting).

The complete random forest algorithm is as follows:

1. Set the number of decision trees *Nc*
2. For *i=1* to *Nc*:
    1. Create a dataset *Di* sampling with replacements from the original dataset *X*
3. Set the number of features to consider during each split *Nf* (for example, *sqrt(n)*)
4. Set an impurity measure (for example, Gini impurity)
5. Define an optional maximum depth for each tree
6. For *i=1* to *Nc*:
    1. Random forest:
        1. Train the decision tree *di(x)* using the dataset *Di* and selecting the best split among *Nf* features randomly sampled

2. Extra-trees:
    1. Train the decision tree *di(x)* using the dataset *Di*, computing before each split *n* random thresholds and selecting the one that yields the least impurity

4. Define an output function averaging the single outputs or employing a majority vote

Example of random forest with Scikit-Learn

In this example, we are going to use the famous Wine dataset (178 13-dimensional samples split into three classes) that is directly available in Scikit-Learn. Unfortunately, it's not so easy to find good and simple datasets for ensemble learning algorithms, as they are normally employed with large and complex sets that require too long a computational time. As the Wine dataset is not particularly complex, the first step is to assess the performances of different classifiers (logistic regression, decision tree, and polynomial SVM) using a k-fold cross-validation:

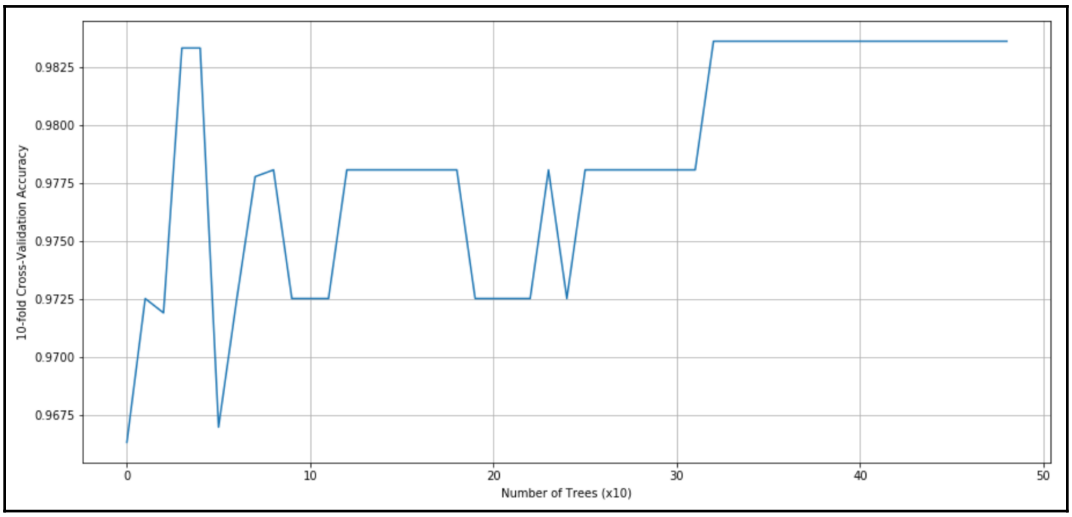
```
import numpy as np

from sklearn.datasets import load_wine
from sklearn.model_selection import cross_val_score
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.svm import SVC

X, Y = load_wine(return_X_y=True)

lr = LogisticRegression(max_iter=1000, random_state=1000)
print(np.mean(cross_val_score(lr, X, Y, cv=10)))
0.956432748538

dt = DecisionTreeClassifier(criterion='entropy', random_state=1000)
print(np.mean(cross_val_score(dt, X, Y, cv=10)))
0.933298933609

svm = SVC(kernel='poly', random_state=1000)
print(np.mean(cross_val_score(svm, X, Y, cv=10)))
0.961403508772
```

As expected, the performances are quite good, with a top value of average cross-validation accuracy equal to about 96% achieved by the polynomial (the default degree is 3) SVM. A very interesting element is the performance of the decision tree, the worst of the set (with Gini impurity it's lower). Even if it's not correct, we can define this model as the weakest of the group and it's a perfect candidate for our bagging test. We can now fit a Random Forest by instantiating the class `RandomForestClassifier` and selecting `n_estimators=50` (I invite the reader to try different values):

```
from multiprocessing import cpu_count

from sklearn.ensemble import RandomForestClassifier

rf = RandomForestClassifier(n_estimators=50, n_jobs=cpu_count(),
random_state=1000)
print(np.mean(cross_val_score(rf, X, Y, cv=10)))
0.983333333333
```

As expected, the average cross-validation accuracy is the highest, about 98.3%. Therefore, the random forest has successfully found a global configuration of decision trees, so as to specialize them in almost any region of the sample space. The parameter `n_jobs=cpu_count()` tells Scikit-Learn to parallelize the training process using all of the CPU cores available in the machine.

To better understand the dynamics of this model, it's useful to plot the cross-validation accuracy as a function of the number of trees:



Cross-validation accuracy of a random forest as a function of the number of trees

It's not surprising to observe some oscillations and a plateau when the number of trees becomes greater at about 320. The effect of the randomness can cause a performance loss, even increasing the number of learners. In fact, even if the training accuracy grows, the validation accuracy on different folds can be affected by an over-specialization. Moreover, in this case, it's very interesting to notice that the top accuracy is achievable with 50 trees instead of 400 or more. For this reason, I always suggest performing at least a grid search, in order not only to achieve the best accuracy but also to minimize the complexity of the model.

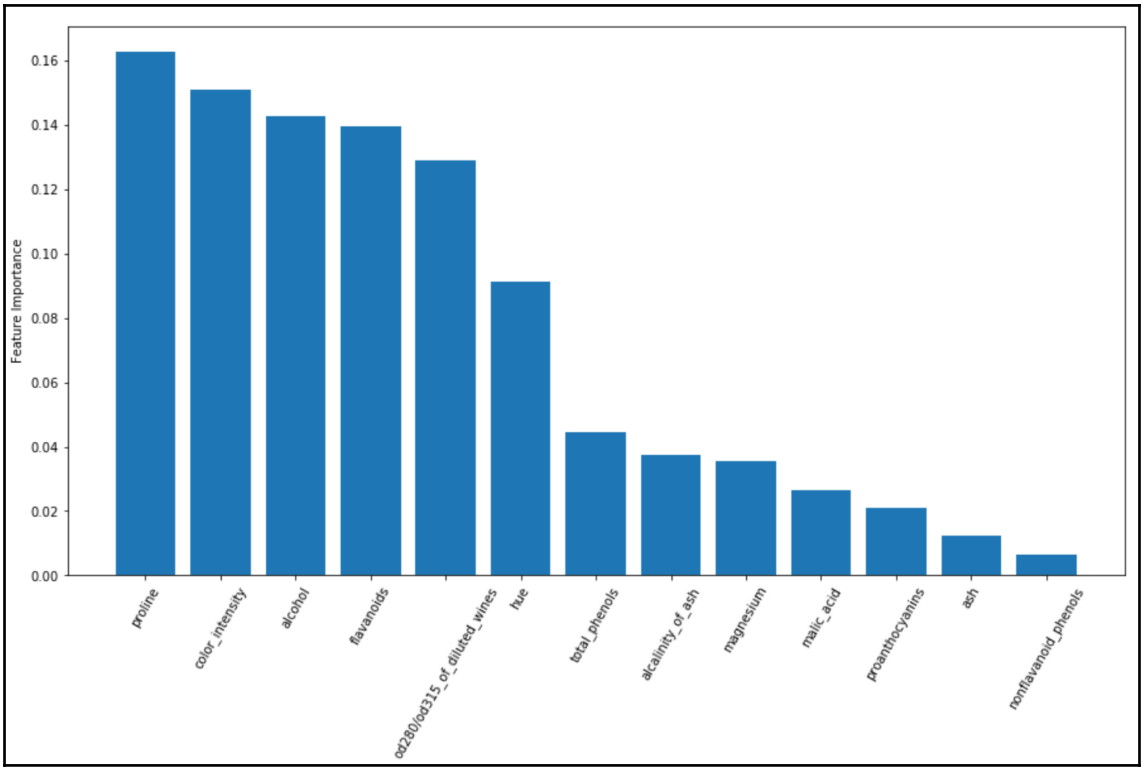
Another important element to consider when working with decision trees and random forests is feature importance (also called Gini importance when this criterion is chosen), which is a measure proportional to the impurity reduction that a particular feature allows us achieve. For a decision tree, it is defined as follows:

$$Importance(\bar{x}^{(i)}) = \sum_{j} \frac{n(j)}{M} \Delta I_j^i$$

In the previous formula, *n(j)* denotes the number of samples reaching the node *j* (the sum must be extended to all nodes where the feature is chosen) and *ΔIi* is the impurity reduction achieved at node *j* after splitting using the feature *i*. In a random forest, the importance must be computed by averaging over all trees:

$$Importance(\bar{x}^{(i)}) = \frac{1}{N_c} \sum_{k=1}^{N_c} \sum_{j} \frac{n(j)}{M} \Delta I_j^i$$

After fitting a model (decision tree or random forest), Scikit-Learn outputs the feature importance vector in the `feature_importances_ instance` variable. In the following graph, there's a plot showing the importance of each feature (the labels can be obtained with the command `load_wine()['feature_names'])` in descending order:



Feature importances for Wine dataset

We don't want to analyze the chemical meaning of each element, but it's clear that, for example, the presence of proline and the color intensity are much more important than the presence of non-flavonoid phenols. As the model is working with features that are semantically independent (it's not the same for the pixels of an image), it's possible to reduce the dimensionality of a dataset by removing all those features whose importance doesn't have a high impact on the final accuracy. This process, called **feature selection**, should be performed using more complex statistical techniques, such as Chi-squared, but when a classifier is able to produce an importance index, it's also possible to use a Scikit-Learn class called `SelectFromModel`. Passing an estimator (that can be fitted or not) and a threshold, it's possible to transform the dataset by filtering out all the features whose value is below the threshold. Applying it to our model and setting a minimum importance equal to `0.02`, we get the following:

```
from sklearn.feature_selection import SelectFromModel

sfm = SelectFromModel(estimator=rf, prefit=True, threshold=0.02)
```

```
X_sfm = sfm.transform(X)

print(X_sfm.shape)
(178, 10)
```

The new dataset now contains 10 features instead of the 13 of the original Wine dataset (for example., it's easy to verify that ash and non-flavonoid phenols have been removed). Of course, as for any other dimensionality reduction method, it's always suggested you verify the final accuracy with a cross-validation and make decisions only if the trade-off between loss of accuracy and complexity reduction is reasonable.

AdaBoost

In the previous section, we have seen that sampling with a replacement leads to datasets where the samples are randomly reweighted. However, if *M* is very large, most of the samples will appear only once and, moreover, all the choices are totally random. AdaBoost is an algorithm proposed by Schapire and Freund that tries to maximize the efficiency of each weak learner by employing adaptive boosting (the name derives from this). In particular, the ensemble is grown sequentially and the data distribution is recomputed at each step so as to increase the weight of those samples that were misclassified and reduce the weight of the ones that were correctly classified. In this way, every new learner is forced to focus on those regions that were more problematic for the previous estimators. The reader can immediately understand that, contrary to random forests and other bagging methods, boosting doesn't rely on randomness to reduce the variance and improve the accuracy. Rather, it works in a deterministic way and each new data distribution is chosen with a precise goal. In this paragraph, we are going to consider a variant called **Discrete AdaBoost** (formally *AdaBoost.M1*), which needs a classifier whose output is thresholded (for example, *-1* and *1*). However, real-valued versions (whose output behaves like a probability) have been developed (a classical example is shown in *Additive Logistic Regression: a Statistical View of Boosting*, *Friedman J.*, *Hastie T.*, *Tibshirani R.*, *Annals of Statistics*, 28/1998). As the main concepts are always the same, the reader interested in the theoretical details of other variants can immediately find them in the referenced papers.

For simplicity, the training dataset of **AdaBoost.M1** is defined as follow:

$$\begin{cases} X = \{\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_M\} \;\; where\; \bar{x}_i \in \mathbb{R}^n \\ Y = \{y_1, y_2, \ldots, y_M\} \;\; where\; y_i \in \{-1, 1\} \end{cases}$$

This choice is not a limitation because, in multi-class problems, a one-versus-the-rest strategy can be easily employed, even if algorithms like **AdaBoost.SAMME** guarantee a much better performance. In order to manipulate the data distribution, we need to define a weight set:

$$\begin{cases} W^{(t)} = \left\{w_1^{(t)}, w_2^{(t)}, \ldots, w_M^{(t)}\right\} \;\; where \;\; w_i^{(t)} \in \mathbb{R}^+ \cup \{0\} \\ W^{(1)} = \left\{\frac{1}{M}, \frac{1}{M}, \ldots, \frac{1}{M}\right\} \end{cases}$$

The weight set allows defining an implicit data distribution *D(t)(x)*, which initially is equivalent to the original one but that can be easily reshaped by changing the values *wi*. Once the family and the number of estimators, *Nc*, have been chosen, it's possible to start the global training process. The algorithm can be applied to any kind of learner that is able to produce thresholded estimations (while the real-valued variants can work with probabilities, for example, obtained through the Platt scaling method).

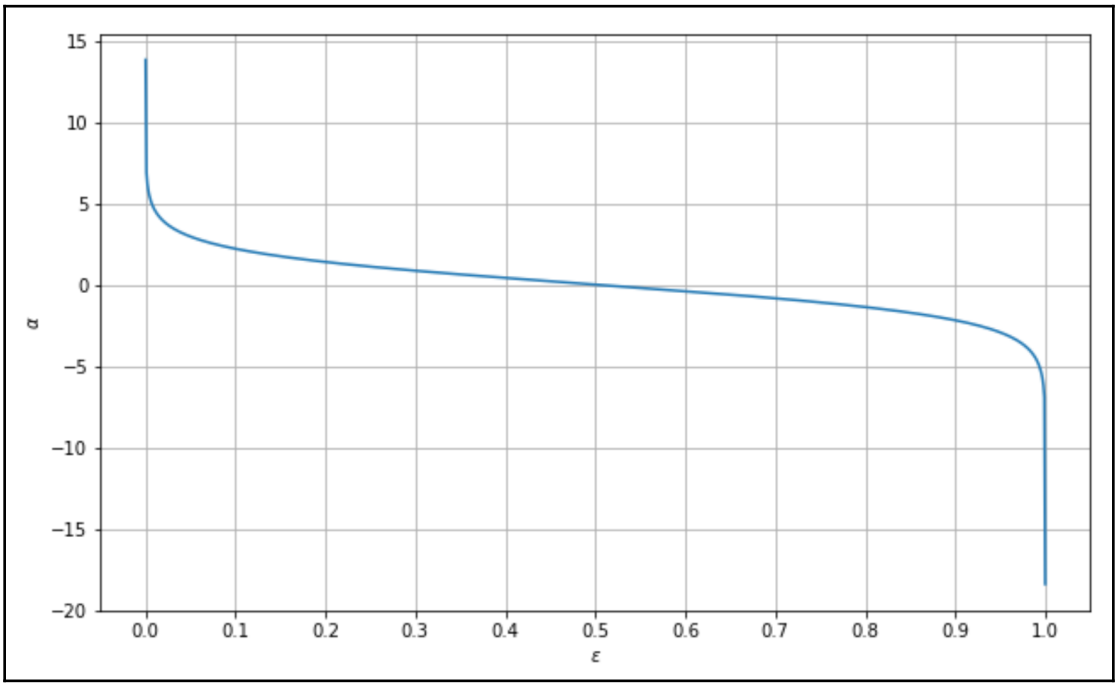
The first instance *d1(x)* is trained with the original dataset, which means with the data distribution *D(1)(x)*. The next instances, instead, are trained with the reweighted distributions *D(2)(x), D(3)(x), ..., D(Nc)(x)*. In order to compute them, after each training process, the normalized weighted error sum is computed:

$$\epsilon^{(t)} = \frac{\sum_{d_t(\bar{x}_i) \neq y_i} w_i}{\sum_{i=1}^{M} w_i}$$

This value is bounded between *0* (no misclassifications) and *1* (all samples have been misclassified) and it's employed to compute the estimator weight *α(t)*:

$$\alpha^{(t)} = log\left(\frac{1 - \epsilon^{(t)}}{\epsilon^{(t)}}\right)$$

To understand how this function works, it's useful to consider its plot (shown in the following diagram):



Estimator weight plot as a function of the normalized weighted error sum

This diagram unveils an implicit assumption: the worst classifier is not the one that misclassifies all samples ($\varepsilon(t) = 1$), but a totally random binary guess (corresponding to $\varepsilon(t) = 0.5$). In this case, $\alpha(t)$ is null and, therefore, the outcome if the estimator is completely discarded. When $\varepsilon(t) < 0.5$, a boosting is applied (between about 0.05 and 0.5, the trend is almost linear), but it becomes greater than 1 only when $\varepsilon(t)$ < about 0.25 (larger values drive to a penalty because the weight is smaller than 1). This value is a threshold to qualify an estimator as trusted or very strong and $\alpha(t) \to +\infty$ in the particular case of a perfect estimator (no errors).

In practice, an upper bound should be imposed in order to avoid overflows or divisions by zero. Instead, when $\varepsilon(t) > 0.5$, the estimator is unacceptably weak, because it's worse than a random guess and the resulting boosting would be negative. To avoid this problem, real implementations must invert the output of such estimators, transforming them de facto into learners with $\varepsilon(t) < 0.5$ (this is not an issue, as the transformation is applied to all output values in the same way). It's important to consider that this algorithm shouldn't be directly applied to multi-class scenarios because, as pointed out in *Multi-class AdaBoost*, *Zhu J.*, *Rosset S.*, *Zou H.*, *Hastie T.*, *01/2006*, the threshold 0.5 corresponds to a random guess accuracy only for binary choices. When the number of classes is larger than two, a random estimator outputs a class with a probability $1/Ny$ (where Ny is the number of classes) and, therefore, AdaBoost.M1 will boost the classifiers in a wrong way, yielding poor final accuracies (the real threshold should be $1 - 1/Ny$, which is larger than 0.5 when $Ny > 2$). The AdaBoost.SAMME algorithm (implemented by Scikit-Learn) has been proposed to solve this problem and exploit the power of boosting also in multi-class scenarios.

The global decision function is defined as follows:

$$d(\bar{x}) = sign\left(\sum_{i=1}^{N_c} \alpha^{(i)} d_i(\bar{x})\right)$$

In this way, as the estimators are added sequentially, the importance of each of them will decrease while the accuracy of $di(x)$ increases. However, it's also possible to observe a plateau if the complexity of X is very high. In this case, many learners will have a high weight, because the final prediction must take into account a sub-combination of learners in order to achieve an acceptable accuracy. As this algorithm specializes the learners at each step, a good practice is to start with a small number of estimators (for example, 10 or 20) and increase the number until no improvement is achieved. Sometimes, a minimum number of good learners (like SVM or decision trees) is sufficient to reach the highest possible accuracy (limited to this kind of algorithm), but in some other cases, the number of estimators can be some thousands. Grid search and cross-validation are again the only good strategies to make the right choice.

After each training step it is necessary to update the weights in order to produce a boosted distribution. This is achieved using an exponential function (based on bipolar outputs *{-1, 1}*):

$$w_i^{(t+1)} = w_i^{(t)} e^{\alpha^{(t)} o_i} \quad where\ o_i = \begin{cases} 1 & if\ \ d_i(\bar{x}_i) \neq y_i \\ -1 & if\ \ d_i(\bar{x}_i) = y_i \end{cases}$$

Given a sample x_i, if it has been misclassified, its weight will be increased considering the overall estimator weight. This approach allows a further adaptive behavior because a classifier with a high $\alpha(t)$ is already very accurate and it's necessary a higher attention level to focus only on the (few) misclassified samples. On the contrary, if $\alpha(t)$ is small, the estimator must improve its overall performance and the over-weighting process must be applied to a large subset (therefore, the distribution won't peak around a few samples, but will penalize only the small subset that has been correctly classified, leaving the estimator free to explore the remaining space with the same probability). Even if not present in the original proposal, it's also possible to include a learning rate η that multiplies the exponent:

$$w_i^{(t+1)} = w_i^{(t)} e^{\eta \alpha^{(t)} o_i}$$

A value $\eta = 1$ has no effect, while smaller values have been proven to increase the accuracy by avoiding a premature specialization. Of course, when $\eta << 1$, the number of estimators must be increased in order to compensate the minor reweighting and this can drive to a training performance loss. As for the other hyperparameters, the right value for η must be discovered using a cross-validation technique (alternatively, if it's the only value that must be fine-tuned, it's possible to start with one and proceed by decreasing its value until the maximum accuracy has been reached).

The complete AdaBoost.M1 algorithm is as follows:

1. Set the family and the number of estimators *Nc*
2. Set the initial weights *W(1)* equal to *1/M*
3. Set the learning rate η (for example, $\eta = 1$)
4. Set the initial distribution *D(1)* equal to the dataset *X*

5. For *i=1* to *Nc*:
    1. Train the i^{th} estimator *di(x)* with the data distribution *D(i)*
    2. Compute the normalized weighted error sum $\varepsilon(i)$:
        1. If $\varepsilon(i) > 0.5$, invert all estimator outputs
    3. Compute the estimator weight $\alpha(i)$
    4. Update the weights using the exponential formula (with or without the learning rate)
    5. Normalize the weights
6. Create the global estimator applying the sign(•) function to the weighted sum $\alpha(i)di(x)$ (for *i=1* to *Nc*)

AdaBoost.SAMME

This variant, called **Stagewise Additive Modeling using a Multi-class Exponential loss** (**SAMME**), was proposed by Zhu, Rosset, Zou, and Hastie in *Multi-class AdaBoost*, *Zhu J.*, *Rosset S.*, *Zou H.*, *Hastie T.*, *01/2006*. The goal is to adapt AdaBoost.M1 in order to work properly in multi-class scenarios. As this is a discrete version, its structure is almost the same, with a difference in the estimator weight computation. Let's consider a label dataset, *Y*:

$$Y = \{y_1, y_2, \ldots, y_M\} \quad where\ y_i \in [1, p]$$

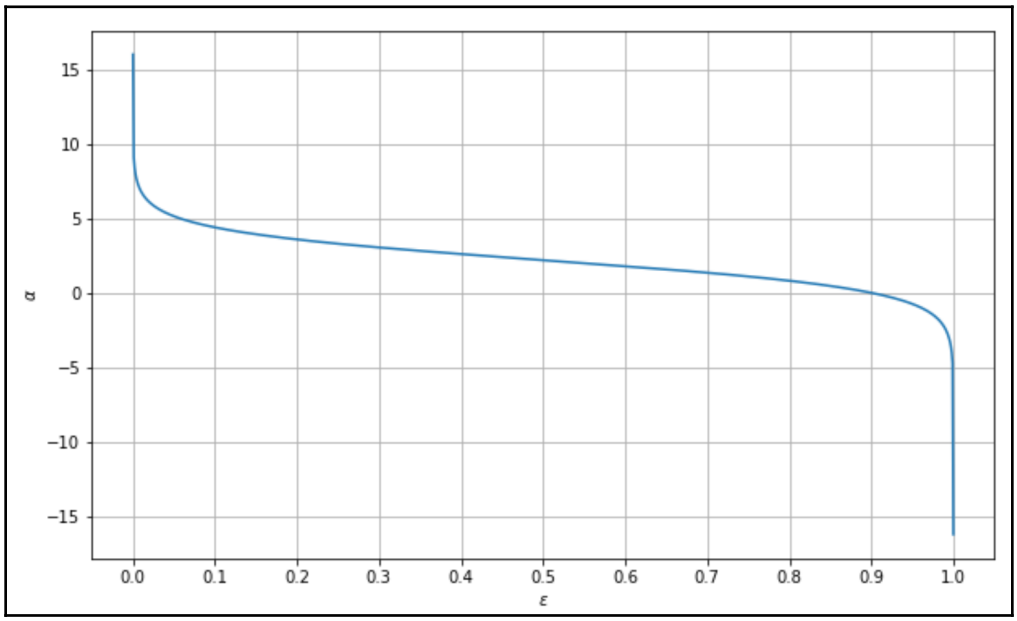
Now, there are *p* different classes and it's necessary to consider that a random guess estimator cannot reach an accuracy equal to 0.5; therefore, the new estimator weights are computed as follows:

$$\alpha^{(t)} = log\left(\frac{1-\epsilon^{(t)}}{\epsilon^{(t)}}\right) + log(p-1) = log\left(\frac{(1-\epsilon^{(t)})(p-1)}{\epsilon^{(t)}}\right)$$

In this way, the threshold is pushed forward and $\alpha(t)$ will be zero when the following is true:

$$\epsilon^{(t)} = 1 - \frac{1}{p}$$

The following graph shows the plot of $\alpha(t)$ with $p = 10$:



Estimator weight plot as a function of the normalized weighted error sum when p = 10

Employing this correction, the boosting process can successfully cope with multi-class problems without the bias normally introduced by AdaBoost.M1 when $p > 2$ ($\alpha(t) > 0$ when the error is less than an actual random guess, which is a function of the number of classes). As the performance of this algorithm is clearly superior, the majority of AdaBoost implementations aren't based on the original algorithm anymore (as already mentioned, for example, Scikit-Learn implements AdaBoost.SAMME and the real-valued version AdaBoost.SAMME.R). Of course, when $p = 2$, AdaBoost.SAMME is exactly equivalent to AdaBoost.M1.

AdaBoost.SAMME.R

AdaBoost.SAMME.R is a variant that works with classifiers that can output prediction probabilities. This is normally possible employing techniques such as Platt scaling, but it's important to check whether a specific classifier implementation is able to output the probabilities without any further action. For example, SVM implementations provided by Scikit-Learn don't compute the probabilities unless the parameter `probability=True` (because they require an extra step that could be useless in some cases).

In this case, we assume that the output of each classifier is a probability vector:

$$\hat{y}^{(t)} = d_t(\bar{x}_t) \;\; where \; \hat{y}^{(t)} = \left(p^{(t)}(y_t = 1|\bar{x}_t), p^{(t)}(y_t = 2|\bar{x}_t), \ldots, p^{(t)}(y_t = p|\bar{x}_t)\right)$$

Each component is the conditional probability that the j^{th} class is output given the input xi. When working with a single estimator, the winning class is obtained through the argmax(•) function; however, in this case, we want to re-weight each learner so as to obtain a sequentially grown ensemble. The basic idea is the same as AdaBoost.M1, but, as now we manage probability vectors, we also need an estimator weighting function that depends on the single sample *xi* (this function indeed wraps every single estimator that is now expressed as a probability vectorial function *pi(t)(y=i|x))*:

$$\alpha_i^{(t)}(\bar{x}) = (p-1)log\left(p_i^{(t)}(y = i|\bar{x})\right) - \frac{p-1}{p}\sum_{j=1}^{p} log\left(p_j^{(t)}(y = i|\bar{x})\right)$$

Considering the properties of logarithms, the previous expression is equivalent to a discrete $\alpha(t)$; however, in this case, we don't rely on a weighted error sum (the theoretical explanation is rather complex and is beyond the scope of this book. The reader can find it in the aforementioned paper, even if the method presented in the next chapter discloses a fundamental part of the logic). To better understand the behavior of this function, let's consider a simple scenario with *p = 2*. The first case is a sample that the learner isn't able to classify (*p=(0.5, 0.5)*):

$$\alpha_i^{(t)}(\bar{x}) = log\left(\frac{1}{2}\right) - \frac{1}{2}\left(log\left(\frac{1}{2}\right) + log\left(\frac{1}{2}\right)\right) = 0$$

In this case, the uncertainty is maximal and the classifier cannot be trusted for this sample, so the weight becomes null for all output probabilities. Now, let's apply the boosting, obtaining the probability vector *p=(0.7, 0.3)*:

$$\begin{cases} \alpha_1^{(t)}(\bar{x}) = log(0.7) - \frac{1}{2}(log(0.7) + log(0.3)) \approx 0.42 \\ \alpha_2^{(t)}(\bar{x}) = log(0.3) - \frac{1}{2}(log(0.7) + log(0.3)) \approx -0.42 \end{cases}$$

The first class will become positive and its magnitude will increase when $p \rightarrow 1$, while the other one is the opposite value. Therefore, the functions are symmetric and allow working with a sum:

$$d(\bar{x}) = argmax_j \sum_{i=1}^{N_c} \alpha_j^{(i)}(\bar{x})$$

This approach is very similar to a weighted majority vote because the winning class *yi* is computed taking into account not only the number of estimators whose output is *yi* but also their relative weight and the negative weight of the remaining classifiers. A class can be selected only if the strongest classifiers predicted it and the impact of the other learners is not sufficient to overturn this result.

In order to update the weights, we need to consider the impact of all probabilities. In particular, we want to reduce the uncertainty (which can degenerate to a purely random guess) and force a superior attention focused on all those samples that have been misclassified. To achieve this goal, we need to define the *yi* and *p(t)(xi)* vectors, which contain, respectively, the one-hot encoding of the true class (for example, *(0, 0, 1, ..., 0)*) and the output probabilities yielded by the estimator (as a column vector). Hence, the update rule becomes as follows:

$$w_i^{(t+1)} = w_i^{(t)} e^{-\frac{\eta(p-1)}{p}\left(\bar{y}_i \cdot log\left(\bar{p}^{(t)}(\bar{x}_i)\right)\right)}$$

If, for example, the true vector is (1, 0) and the output probabilities are (0.1, 0.9), with η=1, the weight of the sample will be multiplied by about 3.16. If instead, the output probabilities are (0.9, 0.1), meaning the sample has been successfully classified, the multiplication factor will become closer to 1. In this way, the new data distribution *D(t+1)*, analogously to AdaBoost.M1, will be more peaked on the samples that need more attention. All implementations include the learning rate as a hyperparameter because, as already explained, the default value equal to 1.0 cannot be the best choice for specific problems. In general, a lower learning rate allows reducing the instability when there are many outliers and improves the generalization ability thanks to a slower convergence towards the optimum. When $\eta < 1$, every new distribution is slightly more focused on the misclassified samples, allowing the estimators to search for a better parameter set without big jumps (that can lead the estimator to skip an optimal point). However, contrary to Neural Networks that normally work with small batches, AdaBoost can often perform quite well also with $\eta=1$ because the correction is applied only after a full training step. As usual, I recommend performing a grid search to select the right values for each specific problem.

The complete AdaBoost.SAMME.R algorithm is as follows:

1. Set the family and the number of estimators *Nc*
2. Set the initial weights *W(1)* equal to *1/M*
3. Set the learning rate η (for example, $\eta = 1$)
4. Set the initial distribution *D(1)* equal to the dataset *X*
5. For *i=1* to *Nc*:
    1. Train the i^{th} estimator *di(x)* with the data distribution *D(i)*
    2. Compute the output probabilities for each class and each training sample
    3. Compute the estimator weights $\alpha j(i)$
    4. Update the weights using the exponential formula (with or without the learning rate)
    5. Normalize the weights
6. Create the global estimator applying the argmax(•) function to the sum $\alpha j(i)$ (for *i=1* to *Nc*)

AdaBoost.R2

A slightly more complex variant has been proposed by Drucker (in *Improving Regressors using Boosting Techniques*, *Drucker H., ICML 1997*) to manage regression problems. The weak learners are commonly decision trees and the main concepts are very similar to the other variants (in particular, the re-weighting process applied to the training dataset). The real difference is the strategy adopted in order to choose the final prediction *yi* given the input sample *xi*. Assuming that there are Nc estimators and each of them is represented as function *dt(x)*, we can compute the absolute residual *ri(t)* for every input sample:

$$r_i^{(t)} = \left| d_t(\bar{x}_i) - y_i \right|$$

Once the set *Ri* containing all the absolute residuals has been populated, we can compute the quantity *Sr = sup Ri* and compute the values of a cost function that must be proportional to the error. The common choice that is normally implemented (and suggested by the author himself) is a linear loss:

$$L_i^{(t)} = \frac{r_i^{(t)}}{S_r}$$

This loss is very flat and it's directly proportional to the error. In most cases, it's a good choice because it avoids premature over-specialization and allows the estimators to readapt their structure in a gentler way. The most obvious alternative is the square loss, which starts giving more importance to those samples whose prediction error is larger. It is defined as follows:

$$L_i^{(t)} = \frac{{r_i^{(t)}}^2}{S_r^2}$$

The last cost function is strictly related to AdaBoost.M1 and it's exponential:

$$L_i^{(t)} = e^{-\frac{r_i^{(t)}}{S_r}}$$

This is normally a less robust choice because, as we are also going to discuss in the next section, it penalizes small errors in favor of larger ones. Considering that these functions are also employed in the re-weighting process, an exponential loss can force the distribution to assign very high probabilities to samples whose misclassification error is high, driving the estimators to become over-specialized with effect from the first iterations. In many cases (such as in neural networks), the loss functions are normally chosen according to their specific properties but, above all, to the ease to minimize them. In this particular scenario, loss functions are a fundamental part of the boosting process and they must be chosen considering the impact on the data distribution. Testing and cross-validation provide the best tool to make a reasonable decision.

Once the loss function has been evaluated for all training samples, it's possible to build the global cost function as the weighted average of all losses. Contrary to many algorithms that simply sum or average the losses, in this case, it's necessary to consider the structure of the distribution. As the boosting process reweights the samples, also the corresponding loss values must be filtered to avoid a bias. At the iteration *t*, the cost function is computed as follows:

$$C^{(t)} = \frac{1}{\sum_j w_j^{(t)}} \sum_{i=1}^{M} \frac{L_i^{(t)}}{w_i^{(t)}}$$

This function is proportional to the weighted errors, which can be either linearly filtered or emphasized using a quadratic or exponential function. However, in all cases, a sample whose weight is lower will yield a smaller contribution, letting the algorithm focus on the samples more difficult to be predicted. Consider that, in this case, we are working with classifications; therefore, the only measure we can exploit is the loss. Good samples yield low losses, hard samples yield proportionally higher losses. Even if it's possible to use *C(t)* directly, it's preferable to define a confidence measure:

$$\gamma^{(t)} = \frac{C^{(t)}}{1 - C^{(t)}}$$

This index is inversely proportional to the average confidence at the iteration *t*. In fact, when $C(t) \rightarrow 0$, $\gamma(t) \rightarrow 0$ and when $C(t) \rightarrow \infty$, $\gamma(t) \rightarrow 1$. The weight update is performed considering the overall confidence and the specific loss value:

$$w_i^{(t+1)} = w_i \gamma^{(t)^{1-L_i^{(t)}}}$$

A weight will be decreased proportionally to the loss associated with the corresponding absolute residual. However, instead of using a fixed base, the global confidence index is chosen. This strategy allows a further degree of adaptability, because an estimator with a low confidence doesn't need to focus only on a small subset and, considering that $\gamma(t)$ is bounded between 0 and 1 (worst condition), the exponential becomes ineffective when the cost function is very high (1x = 1), so that the weights remain unchanged. This procedure is not very dissimilar to the one adopted in other variants, but it tries to find a trade-off between global accuracy and local misclassification problems, providing an extra degree of robustness.

The most complex part of this algorithm is the approach employed to output a global prediction. Contrary to classification algorithms, we cannot easily compute an average, because it's necessary to consider the global confidence at each iteration. Drucker proposed a method based on the weighted median of all outputs. In particular, given a sample xi, we define the set of predictions:

$$Y_i = \left\{ y_i^{(1)}, y_i^{(2)}, \ldots, y_i^{(Nc)} \right\}$$

As weights, we consider the $log(1 / \gamma(t))$, so we can define a weight set:

$$\Gamma = \left\{ log\left(\frac{1}{\gamma^{(1)}}\right), log\left(\frac{1}{\gamma^{(2)}}\right), \ldots, log\left(\frac{1}{\gamma^{(Nc)}}\right) \right\}$$

The final output is the median of Y weighted according to Γ (normalized so that the sum is 1.0). As $\gamma(t) \rightarrow 1$ when the confidence is low, the corresponding weight will tend to 0. In the same way, when the confidence is high (close to 1.0), the weight will increase proportionally and the chance to pick the output associated with it will be higher. For example, if the outputs are $Y = \{1, 1.2, 1.3, 2.0, 2.2, 2.5, 2.6\}$ and the weights are $\Gamma = \{ 0.35, 0.15, 0.12, 0.11, 0.1, 0.09, 0.08 \}$, the weighted median corresponds to the second index, therefore the global estimator will output 1.2 (which is, also intuitively, the most reasonable choice).

The procedure to find the median is quite simple:

1. The $yi(t)$ must be sorted in ascending order, so that $yi(1) < yi(2) < ... < yi(Nc)$
2. The set Γ is sorted accordingly to the index of $yi(t)$ (each output $yi(t)$ must carry its own weight)
3. The set Γ is normalized, dividing it by its sum
4. The index corresponding to the smallest element that splits Γ into two blocks (whose sums are less than or equal to 0.5) is selected
5. The output corresponding to this index is chosen

The complete AdaBoost.R2 algorithm is as follows:

1. Set the family and the number of estimators Nc
2. Set the initial weights $W(1)$ equal to $1/M$
3. Set the initial distribution $D(1)$ equal to the dataset X
4. Select a loss function L
5. For $i=1$ to Nc:
    1. Train the i^{th} estimator $di(x)$ with the data distribution $D(i)$
    2. Compute the absolute residuals, the loss values, and the confidence measures
    3. Compute the global cost function
    4. Update the weights using the exponential formula
6. Create the global estimator using the weighted median

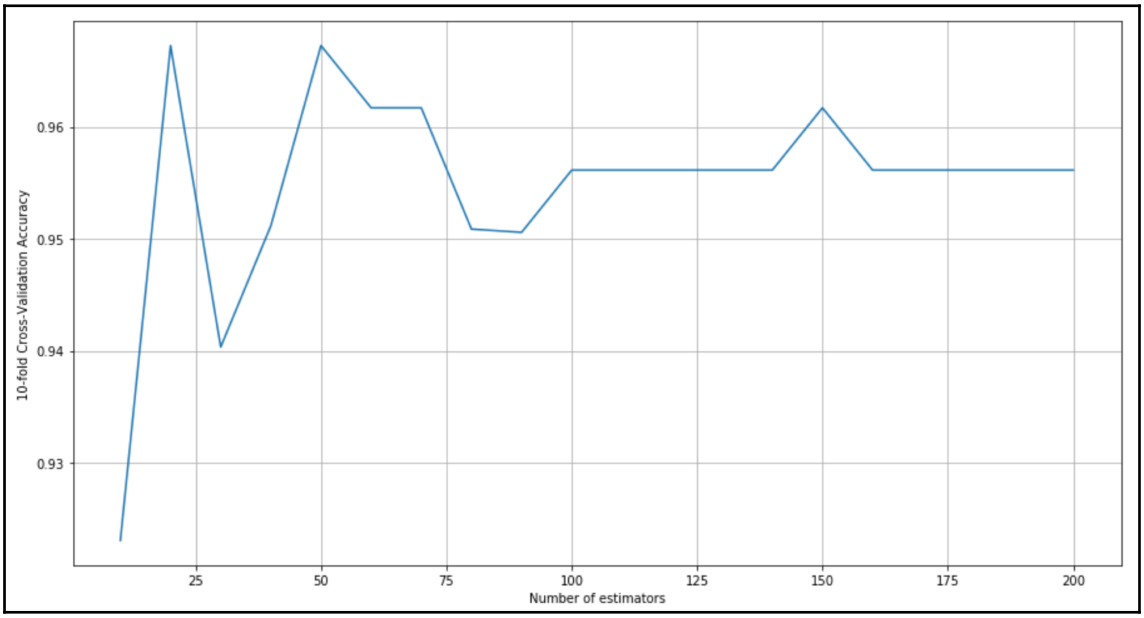
Example of AdaBoost with Scikit-Learn

Let's continue using the Wine dataset in order to analyze the performance of AdaBoost with different parameters. Scikit-Learn, like almost all algorithms, implements both a classifier `AdaBoostClassfier` (based on the algorithm SAMME and SAMME.R) and a regressor `AdaBoostRegressor` (based on the algorithm R2). In this case, we are going to use the classifier, but I invite the reader to test the regressor using a custom dataset or one of the built-in toy datasets. In both classes, the most important parameters are `n_estimators` and `learning_rate` (default value set to `1.0`). The default underlying weak learner is always a decision tree, but it's possible to employ other models creating a base instance and passing it through the parameter `base_estimator`. As explained in the chapter, real-valued AdaBoost algorithms require an output based on a probability vector. In Scikit-Learn, some classifiers/regressors (such as SVM) don't compute the probabilities unless it is explicitly required (setting the parameter `probability=True`); therefore, if an exception is raised, I invite you to check the documentation in order to learn how to force the algorithm to compute them.

The examples we are going to discuss have only a didactic purpose because they focus on a single parameter. In a real-world scenario, it's always better to perform a grid search (which is more expensive), so as to analyze a set of combinations. Let's start analyzing the cross-validation score as a function of the number of estimators (the vectors X and Y are the ones defined in the previous example):

```
import numpy as np

from sklearn.ensemble import AdaBoostClassifier
from sklearn.model_selection import cross_val_score

scores_ne = []

for ne in range(10, 201, 10):
    adc = AdaBoostClassifier(n_estimators=ne, learning_rate=0.8,
random_state=1000)
    scores_ne.append(np.mean(cross_val_score(adc, X, Y, cv=10)))
```

We have considered a range starting from 10 trees and ending with 200 trees with steps of 10 trees. The learning rate is kept constant and equal to 0.8. The resulting plot is shown in the following graph:



10-fold cross-validation accuracy as a function of the number of estimators

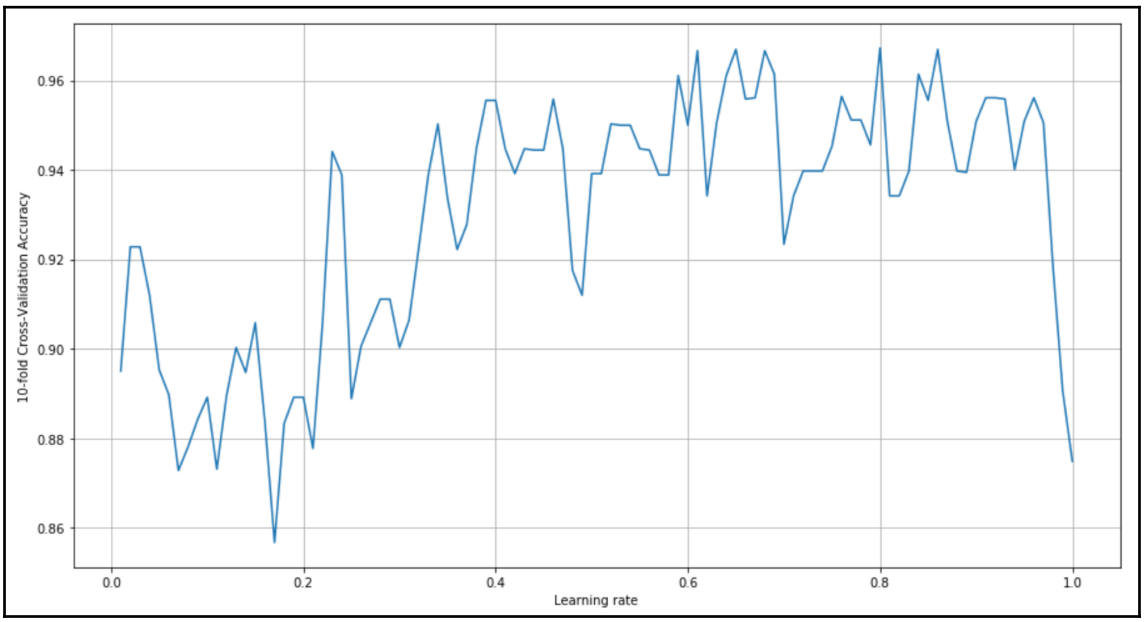
The maximum is reached with 50 estimators. Larger values cause performance worsening due to the over-specialization and a consequent variance increase. As explained also in other chapters, the capacity of a model must be tuned according to the Occam's Razor principle, not only because the resulting model can be faster to train, but also because a capacity excess is normally saturated, overfitting the training set and reducing the scope for generalization. Cross-validation can immediately show this effect, which, instead, can remain hidden when a standard training/test set split is done (above all when the samples are not shuffled).

Let's now check the performance with different learning rates (keeping the number of trees fixed):

```
import numpy as np

scores_eta_adc = []

for eta in np.linspace(0.01, 1.0, 100):
    adc = AdaBoostClassifier(n_estimators=50, learning_rate=eta,
```

```
random_state=1000)
    scores_eta_adc.append(np.mean(cross_val_score(adc, X, Y, cv=10)))
```

The final plot is shown in the following graph:



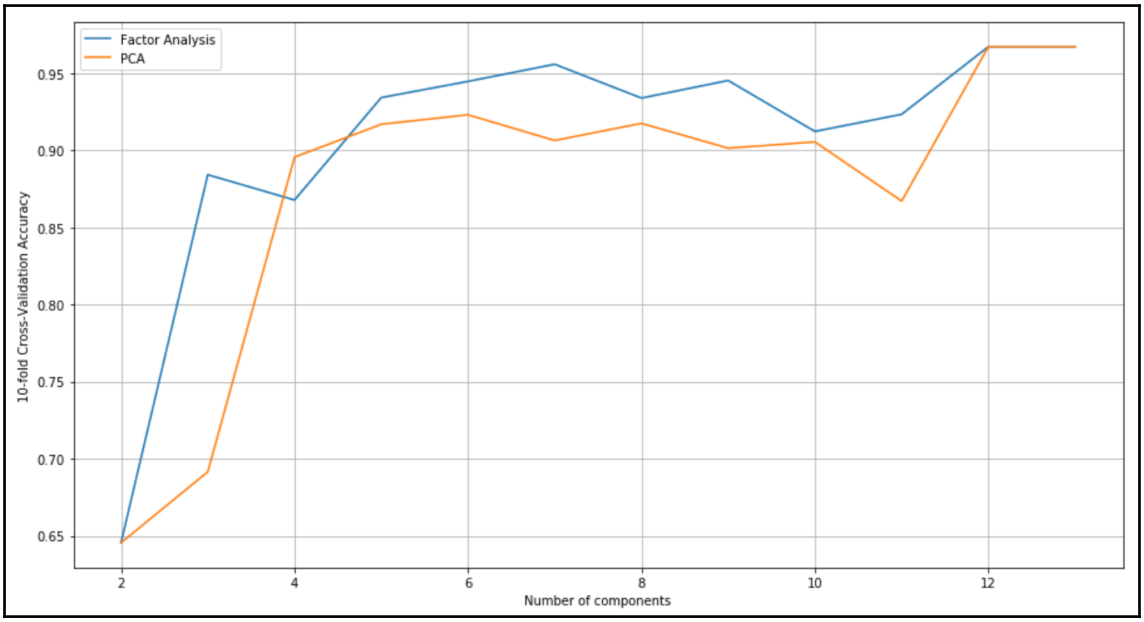
10-fold Cross-validation accuracy as a function of the learning rate (number of estimators = 50)

Again, different learning rates yield different accuracies. The choice of $\eta = 0.8$ seems to be the most effective, as higher and lower values lead to performance worsening. As explained, the learning rate has a direct impact on the re-weighting process. Very small values require a larger number of estimators because subsequent distributions are very similar. On the other side, large values can lead to a premature over-specialization. Even if the default value is `1.0`, I always suggest checking the accuracy also with smaller values. There's no golden rule for picking the right learning rate in every case, but it's important to remember that lower values allow the algorithm to smoothly adapt to fit the training set in a gentler way, while higher values reduce the robustness to outliers, because the samples that have been misclassified are immediately boosted and the probability of sampling them increases very rapidly. The result of this behavior is a constant focus on those samples that may be affected by noise, almost forgetting the structure of the remaining sample space.

The last experiment we want to make is analyzing the performance after a dimensionality reduction performed with **Principal Component Analysis** (**PCA**) and **Factor Analysis** (**FA**) (with 50 estimators and $\eta = 0.8$):

```
import numpy as np

from sklearn.decomposition import PCA, FactorAnalysis

scores_pca = []

for i in range(13, 1, -1):
    if i < 12:
        pca = PCA(n_components=i, random_state=1000)
        X_pca = pca.fit_transform(X)
    else:
        X_pca = X
    adc = AdaBoostClassifier(n_estimators=50, learning_rate=0.8,
random_state=1000)
    scores_pca.append(np.mean(cross_val_score(adc, X_pca, Y, cv=10)))

scores_fa = []

for i in range(13, 1, -1):
    if i < 12:
        fa = FactorAnalysis(n_components=i, random_state=1000)
        X_fa = fa.fit_transform(X)
    else:
        X_fa = X
    adc = AdaBoostClassifier(n_estimators=50, learning_rate=0.8,
random_state=1000)
    scores_fa.append(np.mean(cross_val_score(adc, X_fa, Y, cv=10)))
```

The resulting plot is shown in the following graph:



10-fold cross-validation accuracy as a function of the number of components (PCA and factor analysis)

This exercise confirms some important features analyzed in `Chapter 5`, *EM Algorithm and Applications*. First of all, performances are not dramatically affected even by a 50% dimensionality reduction. This consideration is further confirmed by the feature importance analysis performed in the previous example. Decision trees can perform quite a good classification considering only 6/7 features because the remaining ones offer a marginal contribution to the characterization of a sample. Moreover, FA is almost always superior to PCA. With 7 components, the accuracy achieved using the FA algorithm is higher than 0.95 (very close to the value achieved with no reduction), while a PCA reaches this value with 12 components. The reader should remember that PCA is a particular case of FA, with the assumption of homoscedastic noise. The diagram confirms that this condition is not acceptable with the Wine dataset. Assuming different noise variances allows remodeling the reduced dataset in a more accurate way, minimizing the cross-effect of the missing features. Even if PCA is normally the first choice, with large datasets, I suggest you always compare the performance with a Factor Analysis and choose the technique that guarantees the best result (given also that FA is more expensive in terms of computational complexity).

Gradient boosting

At this point, we can introduce a more general method of creating boosted ensembles. Let's choose a generic algorithm family, represented as follows:

$$d_i(\bar{x}) = f(\bar{x}; \bar{\theta}_i)$$

Each model is parametrized using the vector θi and there are no restrictions on the kind of method that is employed. In this case, we are going to consider decision trees (which is one of the most diffused algorithms when this boosting strategy is employed—in this case, the algorithm is known as gradient tree boosting), but the theory is generic and can be easily applied to more complex models, such as neural networks. In a decision tree, the parameter vector θi is made up of selection tuples, so the reader can think of this method as a pseudo-random forest where, instead of randomness, we look for extra optimality exploiting the previous experience. In fact, as with AdaBoost, a gradient boosting ensemble is built sequentially, using a technique that is formally defined as **Forward Stage-wise Additive Modeling**. The resulting estimator is represented as a weighted sum:

$$d(\bar{x}) = \sum_{i=1}^{N_c} \alpha_i d_i(\bar{x}) = \sum_{i=1}^{N_c} \alpha_i f(\bar{x}; \bar{\theta}_i)$$

Therefore the variables to manage are the single estimator weights αi and the parameter vectors θi. However, we don't have to work with the whole set, but with a single tuple $(\alpha i, \theta i)$, without modifying the values already chosen during the previous iterations. The general procedure can be summarized with a loop:

1. The estimator sum is initialized to a null value
2. For $i=1$ to Nc:
    1. The best $tuple(\alpha i, \theta i)$ is chosen and the estimator $f(x; \theta i)$ is trained
    2. $di(x) = di\text{-}1(x) + \alpha i f(x; \theta i)$
3. The final estimator $d(x)$ is output

How is it possible to find out the best tuple? We have already presented a strategy for improving the performance of every learner through boosting the dataset. In this case, instead, the algorithm is based on a cost function that we need to minimize:

$$C(X;Y;\bar{\alpha};\bar{\theta}) = \sum_{i=1}^{M} L(y_i, \alpha_i f(\bar{x}_i;\bar{\theta}))$$

In particular, the generic optimal tuple is obtained as follows:

$$(\bar{\alpha}^*, \bar{\theta}^*) = argmin_{\bar{\alpha},\bar{\theta}}\ C(X;Y;\bar{\alpha};\bar{\theta})$$

As the process is sequential, each estimator is optimized to improve the previous one's accuracy. However, contrary to AdaBoost, we are not constrained to impose a specific loss function (it's possible to prove that AdaBoost.M1 is equivalent to this algorithm with an exponential loss but the proof is beyond the scope of this book). As we are going to discuss, other cost functions can yield better performances in several different scenarios, because they avoid the premature convergence towards sub-optimal minima.

The problem could be considered as solved by employing the previous formula to optimize each new learner; however, the `argmin(•)` function needs a complete exploration of the cost function space and, as `C(•)` depends on each specific model instance and, therefore, on *θi*, it's necessary to perform several retraining processes in order to find the optimal solution. Moreover, the problem is generally non-convex and the number of variables can be very high. Numerical algorithms such as L-BFGS or other quasi-Newton methods need too many iterations and a prohibitive computational time. It's clear that such an approach is not affordable in the vast majority of cases and the Gradient Boosting algorithm has been proposed as an intermediate solution. The idea is to find a sub-optimal solution with a gradient descent strategy limited to a single step for each iteration.

In order to present the algorithm, it's useful to rewrite the additive model with an explicit reference to the optimal goal:

$$d_i(\bar{x}) = d_{i-1}(\bar{x}) + argmin_f \sum_{j=1}^{M} L(y_j, d_{i-1}(\bar{x}_j) + f(\bar{x}_j;\bar{\theta}_i))$$

Note that the cost function is computed carrying on all the previously trained models; therefore, the correction is always incremental. If the cost function *L* is differentiable (a fundamental condition that is not difficult to meet), it's possible to compute the gradient with respect to the current additive model (at the i^{th} iteration, we need to consider the additive model obtained summing all the previous *i-1* models):

$$\nabla_d \sum_{j=1}^{M} L(y_j, d_{i-1}(\bar{x}_j) + f(\bar{x}_j; \bar{\theta}_i)) = \sum_{j=1}^{M} \nabla_d L(y_j, d_{i-1}(\bar{x}_j))$$

At this point, a new classifier can be added by moving the current additive model into the negative direction of the gradient:

$$d_i(\bar{x}) = d_{i-1}(\bar{x}) - \eta\, \alpha_i \sum_{j=1}^{M} \nabla_d L(y_j, d_{i-1}(\bar{x}_j))$$

We haven't considered the parameter αi yet (nor the learning rate η, which is a constant), however the reader familiar with some basic calculus can immediately understand the effect of an update is to reduce the value of the global loss function by forcing the next model to improve its accuracy with respect to its predecessors. However, a single gradient step isn't enough to guarantee an appropriate boosting strategy. In fact, as discussed previously, we also need to weight each classifier according to its ability to reduce the loss. Once the gradient has been computed, it's possible to determine the best value for the weight αi with a direct minimization of the loss function (using a line search algorithm) computed considering the current additive model with α as an extra variable:

$$\alpha_i = argmin_\alpha \sum_{j=1}^{M} L(y_j, d_i(\bar{x}_j, \alpha)) = argmin_\alpha \sum_{j=1}^{M} L(y_j, d_{i-1}(\bar{x}) - \eta\, \alpha \nabla_d L(\bar{x}_i, f(\bar{x}_i; \bar{\theta}))$$

When using the gradient tree boosting variant, an improvement can be achieved by splitting the weight αi into m sub-weights $\alpha i(j)$ associated with each terminal node of the tree. The computational complexity is slightly increased, but the final accuracy can be higher than the one obtained with a single weight. The reason derives from the functional structure of a tree. As the boosting forces a specialization in specific regions, a single weight could drive to an over-estimation of a learner also when a specific sample cannot be correctly classified. Instead, using different weights, it's possible to operate a fine-grained filtering of the result, accepting or discarding an outcome according to its value and to the properties of the specific tree.

This solution cannot provide the same accuracy of a complete optimization, but it's rather fast and it's possible to compensate for this loss using more estimators and a lower learning rate. Like many other algorithms, gradient boosting must be tuned up in order to yield the maximum accuracy with a low variance. The learning rate is normally quite smaller than 1.0 and its value should be found by validating the results and considering the total number of estimators (it's better to reduce it when more learners are employed). Moreover, a regularization technique could be added in order to prevent overfitting. When working with specific classifier families (such as logistic regression or neural networks), it's very easy to include an *L1* or *L2* penalty, but it's not so easy with other estimators. For this reason, a common regularization technique (implemented also by Scikit-Learn) is the downsampling of the training dataset. Selecting $P < N$ random samples allows the estimators to reduce the variance and prevent overfitting. Alternatively, it's possible to employ a random feature selection (for gradient tree boosting only) as in a random forest; picking a fraction of the total number of features increases the uncertainty and avoids over-specialization. Of course, the main drawback to these techniques is a loss of accuracy (proportional to the downsampling/feature selection ratio) that must be analyzed in order to find the most appropriate trade-off.

Before moving to the next section, it's useful to briefly discuss the main cost functions that are normally employed with this kind of algorithms. In the first chapter, we have presented some common cost functions, like mean squared error, Huber Loss (very robust in regression contexts), and cross-entropy. They are all valid examples, but there are other functions that are peculiar to classification problems. The first one is Exponential Loss, defined as follows:

$$L(y_i, f(\bar{x}_i; \bar{\theta})) = e^{-y_i f(\bar{x}_i; \bar{\theta})}$$

As pointed out by Hastie, Tibshirani and, Friedman, this function transforms the gradient boosting into an AdaBoost.M1 algorithm. The corresponding cost function has a very precise behavior that sometimes is not the most adequate to solve particular problems. In fact, the result of an exponential loss has a very high impact when the error is large, yielding distributions that are strongly peaked around a few samples. The subsequent classifiers can be consequently driven to over-specialize their structure to cope only with a small data region, with a concrete risk of losing the ability to correctly classify other samples. In many situations, this behavior is not dangerous and the final bias-variance trade-off is absolutely reasonable; however, there are problems where a softer loss function can allow a better final accuracy and generalization ability. The most common choice for real-valued binary classification problems is Binomial Negative Log-Likelihood Loss (deviance), defined as follows (in this case we are assuming that the classifier *f(•)* is not thresholded, but outputs a positive-class probability):

$$L(y_i, f(\bar{x}_i; \bar{\theta})) = y_i log(f(\bar{x}_i; \bar{\theta})) + (1 - y_i) log(1 - f(\bar{x}_i; \bar{\theta}))$$

This loss function is the same employed in Logistic Regressions and, contrary to Exponential Loss, doesn't yield peaked distributions. Two misclassified samples with different probabilities are boosted proportionally to the error (not the exponential value), so as to force the classifiers to focus on all the misclassified population with almost the same probability (of course, a higher probability assigned to samples whose error is very large is desirable, assuming that all the other misclassified samples have always a good chance to be selected). The natural extension of the Binomial Negative Log-Likelihood Loss to multi-class problems is the Multinomial Negative Log-Likelihood Loss, defined as follows (the classifier *f(•)* is represented as probability vector with *p* components):

$$L(y_i, \bar{f}(\bar{x}_i; \bar{\theta})) = -\sum_{j=1}^{p} I_{y_i=j} \; log(f_j(\bar{x}_i; \bar{\theta}))$$

In the previous formula, the notation *Iy=j* must be interpreted as an indicator function, which is equal to 1 when *y=j* and 0 otherwise. The behavior of this loss function is perfectly analogous to the binomial variant and, in general, it is the default choice for classification problems. The reader is invited to test the examples with both exponential loss and deviance and compare the results.

The complete gradient boosting algorithm is as follows:

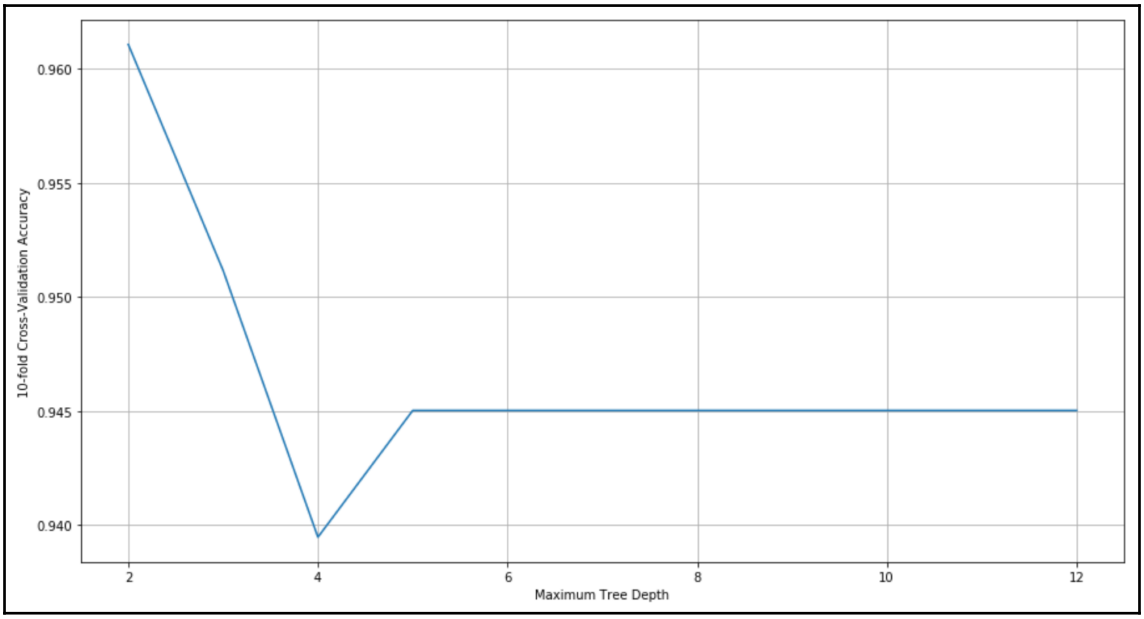
1. Set the family and the number of estimators *Nc*
2. Select a loss function *L* (for example, deviance)
3. Initialize the base estimator *d0(x)* as a constant (such as 0) or using another model
4. Set the learning rate η (such as $\eta = 1$)
5. For *i=1* to *Nc*:
    1. Compute the gradient $\nabla d\ L(\bullet)$ using the additive model at the step *i-1*
    2. Train the i^{th} estimator *di(x)* with the data distribution *{ (xi, ∇d L(yi, di-1(xi)) }*
    3. Perform a line search to compute αi
    4. Add the estimator to the ensemble

Example of gradient tree boosting with Scikit-Learn

In this example, we want to employ a gradient tree boosting classifier (class `GradientBoostingClassifier`) and check the impact of the maximum tree depth (`parameter max_depth`) on the performance. Considering the previous example, we start by setting `n_estimators=50` and `learning_rate=0.8`:

```
import numpy as np

from sklearn.ensemble import GradientBoostingClassifier
from sklearn.model_selection import cross_val_score

scores_md = []
eta = 0.8

for md in range(2, 13):
    gbc = GradientBoostingClassifier(n_estimators=50, learning_rate=eta,
max_depth=md, random_state=1000)
    scores_md.append(np.mean(cross_val_score(gbc, X, Y, cv=10)))
```

The result is shown in the following diagram:



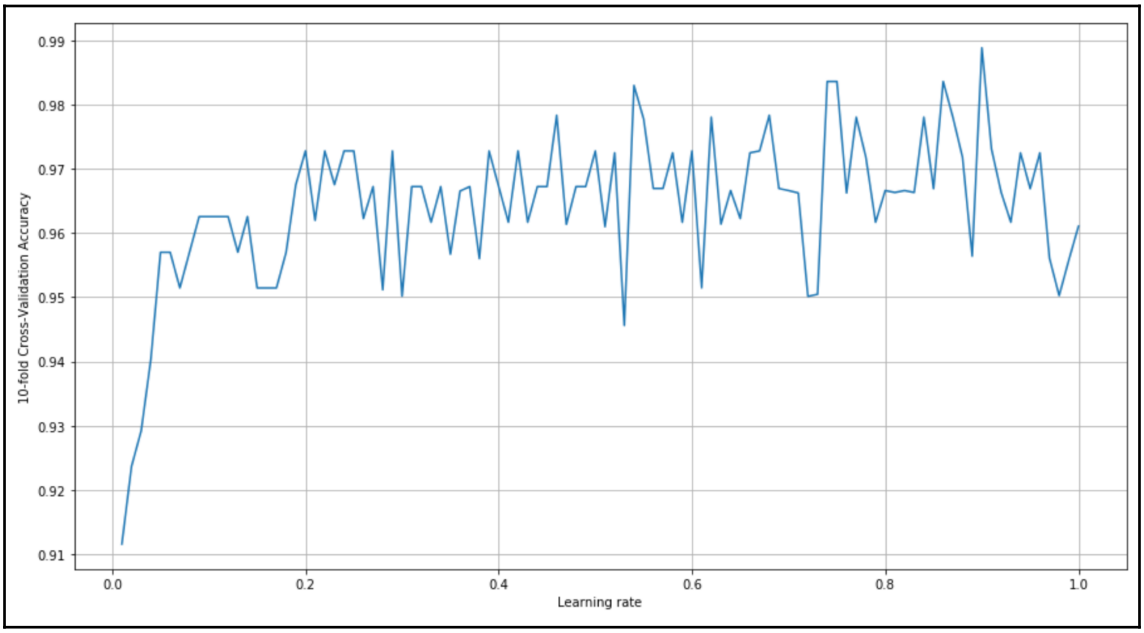
10-fold Cross-validation accuracy as a function of the maximum tree depth

As explained in the first section, the maximum depth of a decision tree is strictly related to the possibility of interaction among features. This can be a positive or negative aspect when the trees are employed in an ensemble. A very high interaction level can create over-complex separation hyperplanes and reduce the overall variance. In other cases, a limited interaction results in a higher bias. With this particular (and simple) dataset, the gradient boosting algorithm can achieve better performances when the max depth is 2 (consider that the root has a depth equal to zero) and this is partially confirmed by both the feature importance analysis and dimensionality reductions. In many real-world situations, the result of such a research could be completely different, with increased performance, therefore I suggest you cross-validate the results (it's better to employ a grid search) starting from a minimum depth and increasing the value until the maximum accuracy has been achieved. With `max_depth=2`, we want now to tune up the learning rate, which is a fundamental parameter in this algorithm:

```
import numpy as np

scores_eta = []

for eta in np.linspace(0.01, 1.0, 100):
```

```
    gbr = GradientBoostingClassifier(n_estimators=50, learning_rate=eta,
max_depth=2, random_state=1000)
    scores_eta.append(np.mean(cross_val_score(gbr, X, Y, cv=10)))
```

The corresponding plot is shown in the following diagram:



10-fold Cross-validation accuracy as a function of the learning rate (max depth equal to 2)

Unsurprisingly, gradient tree boosting outperforms AdaBoost with $\eta \approx 0.9$, achieving a cross-validation accuracy slightly lower than 0.99. The example is very simple, but it clearly shows the power of this kind of techniques. The main drawback is the complexity. Contrary to single models, ensembles are more sensitive to changes to the hyperparameters and more detailed research must be conducted in order to optimize the models. When the datasets are not excessively large, cross-validation remains the best choice. If, instead, we are pretty sure that the dataset represents almost perfectly the underlying data generating process, it's possible to shuffle it and split it into two (training/test) or three blocks (training/test/validation) and proceed by optimizing the hyperparameters and trying to overfit the test set (this expression can seem strange, but overfitting the test set means maximizing the generalization ability while learning perfectly the training set structure).

Ensembles of voting classifiers

A simpler but no less effective way to create an ensemble is based on the idea of exploiting a limited number of strong learners whose peculiarities allow them to yield better performances in particular regions of the sample space. Let's start considering a set of *Nc* discrete-valued classifiers *f1(x), f2(x), ..., fNc(x)*. The algorithms are different, but they are all trained with the same dataset and output the same label set. The simplest strategy is based on a hard-voting approach:

$$\hat{y}_i = argmax\left(n(y_1), n(y_2), \ldots, n(y_{N_c})\right)$$

In this case, the function *n(•)* counts the number of estimators that output the label *yi*. This method is rather powerful in many cases, but has some limitations. If we rely only on a majority vote, we are implicitly assuming that a correct classification is obtained by a large number of estimators. Even if, *Nc/2 + 1* votes are necessary to output a result, in many cases their number is much higher. Moreover, when k is not very large, also *Nc/2 + 1* votes imply a symmetry that involves a large part of the population. This condition often drives to the training of useless models that could be simply replaced by a single well-fitted strong learner. In fact, let's suppose that the ensemble is made up of three classifiers and one of them is more specialized in regions where the other two can easily be driven to misclassifications. A hard-voting strategy applied to this ensemble could continuously penalize the more complex estimator in favor of the other classifiers. A more accurate solution can be obtained by considering real-valued outcomes. If each estimator outputs a probability vector, the confidence of a decision is implicitly encoded in the values. For example, a binary classifier whose output is *(0.52, 0.48)* is much more uncertain than another classifier outputting *(0.95, 0.05)*. Applying a threshold is equivalent to flattening the probability vectors and discarding the uncertainty. Let's consider an ensemble with three classifiers and a sample that is hard to classify because it's very close to the separation hyperplane. A hard-voting strategy decides for the first class because the thresholded output is *(1, 1, 2)*. Then we check the output probabilities, obtaining *(0.51, 0.49)*, *(0.52, 0.48)*, *(0.1, 0.9)*. After averaging the probabilities, the ensemble output becomes about (0.38, 062) and by applying `argmax(•)`, we get the second class as the final decision. In general, it's also a good practice to consider a weighted average, so that the final class is obtained as follows (assuming the output of the classifier is a probability vector):

$$\hat{y}_i = argmax \frac{1}{N_c} \sum_{j=1}^{N_c} w_j \bar{f}_j(\bar{x}_i) \;\; where \;\; \bar{f}_j(\bar{x}_i) = \left(p_j(\hat{y}_j = 1), p_j(\hat{y}_j = 2), \ldots, p_j(\hat{y}_j = p)\right)$$

The weights can be simply equal to 1.0 if no weighting is required or they can reflect the level of trust we have for each classifier. An important rule is to avoid the dominance of a classifier in the majority of cases because it would be an implicit fallback to a single estimator scenario. A good voting example should always allow a minority to overturn a result when their confidence is quite higher than the majority. In this strategies, the weights can be considered as hyperparameters and tuned up using a grid search with cross-validation. However, contrary to other ensemble methods, they are not fine-grained, therefore the optimal value is often a compromise among some different possibilities.

A slightly more complex technique is called **stacking** and consists of using an extra classifier as a post-filtering step. The classical approach consists of training the classifiers separately, then the whole dataset is transformed into a prediction set (based on class labels or probabilities) and the combining classifier is trained to associate the predictions to the final classes. Using even very simple models like Logistic Regressions or Perceptrons, it's possible to mix up the predictions so as to implement a dynamic reweighting that is a function of the input values. A more complex approach is feasible only when a single training strategy can be used to train the whole ensemble (including the combiner). For example, it could be employed with neural networks that, however, have already an implicit flexibility and can often perform quite better than complex ensembles.

Example of voting classifiers with Scikit-Learn

In this example, we are going to employ the MNIST handwritten digits dataset. As the concept is very simple, our goal is to show how to combine two completely different estimators to improve the overall cross-validation accuracy. For this reason, we have selected a Logistic Regression and a decision tree, which are structurally different. In particular, while the former is a linear model that works with the whole vectors, the latter is a feature-wise estimator that can support the decision only in particular cases (images are not made up of semantically consistent features, but the over-complexity of a Decision Tree can help with particular samples which are very close to the separation hyperplane and, therefore, more difficult to classify with a linear method). The first step is loading and normalizing the dataset (this operation is not important with a Decision Tree, but has a strong impact on the performances of a Logistic Regression):

```
import numpy as np

from sklearn.datasets import load_digits

X, Y = load_digits(return_X_y=True)
X /= np.max(X)
```

At this point, we need to evaluate the performances of both estimators individually:

```
import numpy as np

from sklearn.tree import DecisionTreeClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import cross_val_score

dt = DecisionTreeClassifier(criterion='entropy', random_state=1000)
print(np.mean(cross_val_score(dt, X, Y, cv=10)))
0.830880960443

lr = LogisticRegression(C=2.0, random_state=1000)
print(np.mean(cross_val_score(lr, X, Y, cv=10)))
0.937021649942
```

As expected, the Logistic Regression (~94% accuracy) outperforms the decision tree (83% accuracy); therefore, a hard-voting strategy is not the best choice. As we trust the Logistic Regression more, we can employ soft voting with a weight vector set to *(0.9, 0.1)*. The class `VotingClassifier` accepts a list of tuples (name of the estimator, instance) that must be supplied through the `estimators` parameter. The strategy can be specified using parameter voting (it can be either "soft" or "hard") and the optional weights, using the parameter with the same name:

```
import numpy as np

from sklearn.ensemble import VotingClassifier

vc = VotingClassifier(estimators=[
    ('LR', LogisticRegression(C=2.0, random_state=1000)),
    ('DT', DecisionTreeClassifier(criterion='entropy',
random_state=1000))],
     voting='soft', weights=(0.9, 0.1))

print(np.mean(cross_val_score(vc, X, Y, cv=10)))
0.944835154373
```

Using a soft-voting strategy, the estimator is able to outperform Logistic Regression by reducing the global uncertainty. I invite the reader to test this algorithm with other datasets, using more estimators, and try to find out the optimal combination using both the hard and soft voting strategies.

Ensemble learning as model selection

This is not a proper ensemble learning technique, but it is sometimes known as **bucketing**. In the previous section, we have discussed how a few strong learners with different peculiarities can be employed to make up a committee. However, in many cases, a single learner is enough to achieve a good bias-variance trade-off but it's not so easy to choose among the whole Machine Learning algorithm population. For this reason, when a family of similar problems must be solved (they can differ but it's better to consider scenarios that can be easily compared), it's possible to create an ensemble containing several models and use cross-validation to find the one whose performances are the best. At the end of the process, a single learner will be used, but its choice can be considered like a grid search with a voting system. Sometimes this technique can unveil important differences even using similar datasets. For example, during the development of a system, a first dataset (*X1, Y1*) is provided. Everybody expects that it is correctly sampled from an underlying data generating process p_{data} and, so, a generic model is fitted and evaluated. Let's imagine that a SVM achieves a very high validation accuracy (evaluated using a k-fold cross-validation) and, therefore, it is chosen as the final model. Unfortunately, a second, larger dataset (*X2, Y2*) is provided and the final mean accuracy worsens. We might simply think that the residual variance of the model cannot let it generalize correctly or, as sometimes happens, we can say the second dataset contains many outliers which are not correctly classified. The real situation is a little more complex: given a dataset, we can only suppose that it represents a complete data distribution. Even when the number of samples is very high or we use data augmentation techniques, the population might not represent some particular samples that will be analyzed by the system we are developing. Bucketing is a good way to create a security buffer that can be exploited whenever the scenario changes. The ensemble can be made up of completely different models, models belonging to the same family but differently parametrized (for example, different kernel SVMs) or a mixture of composite algorithms (like PCA + SVM, PCA + decision trees/random forests, and so on). The most important element is the cross-validation. As explained in the first chapter, splitting the dataset into training and test sets can be an acceptable solution only when the number of samples and their variability is high enough to justify the belief that it correctly represents the final data distribution. This often happens in deep learning, where the dimensions of the datasets are quite large and the computational complexity doesn't allow retraining the model too many times. Instead, in classical Machine Learning contexts, cross-validation is the only way to check the behavior of a model when trained with a large random subset and tested on the remaining samples. Ideally, we'd like to observe the same performances, but it can also happen that the accuracy is higher in some folds and quite lower in other. When this phenomenon is observed and the dataset is the final one, it probably means that the model is not able to manage one or more regions of the sample space and a boosting approach could dramatically improve the final accuracy.

Summary

In this chapter, we introduced the main concepts of ensemble learning, focusing on both bagging and boosting techniques. In the first section, we explained the difference between strong and weak learners and we presented the big picture of how it's possible to combine the estimators to achieve specific goals.

The next topic focused on the properties of decision trees and their main strengths and weaknesses. In particular, we explained that the structure of a tree causes a natural increase in the variance. The bagging technique called random forests allow mitigating this problem, improving at the same time the overall accuracy. A further variance reduction can be achieved by increasing the randomness and employing a variant called **extra randomized trees**. In the example, we have also seen how it's possible to evaluate the importance of each input feature and perform dimensionality reduction without involving complex statistical techniques.

In the third section, we presented the most famous boosting techniques, AdaBoost, which is based on the concept of creating a sequential additive model, when each new estimator is trained using a reweighted (boosted) data distribution. In this way, every learner is added to focus on the misclassified samples without interfering with the previously added models. We analyzed the original M1 discrete variant and the most effective alternatives called SAMME and SAMME.R (real-valued), and R2 (for regressions), which are implemented in many Machine Learning packages.

After AdaBoost, we extended the concept to a generic Forward Stage-wise Additive Model, where the task of each new estimator is to minimize a generic cost function. Considering the complexity of a full optimization, a gradient descent technique was presented that, combined with an estimator weight line search, can yield excellent performances both in classification and in regression problems.

The final topics concerned how to build ensembles using a few strong learners, averaging their prediction or considering a majority vote. We discussed the main drawback of thresholded classifiers and we showed how it's possible to build a soft-voting model that is able to trust the estimator that show less uncertainty. Other useful topics are the Stacking method, which consists of using an extra classifier to process the prediction of each member of the ensemble and how it's possible to create candidate ensembles that are evaluated using a cross-validation technique to find out the best estimator for each specific problem.

In the next chapter, we are going to begin discussing the most important deep learning techniques, introducing the fundamental concepts regarding neural networks and the algorithms involved in their training processes.

9
Neural Networks for Machine Learning

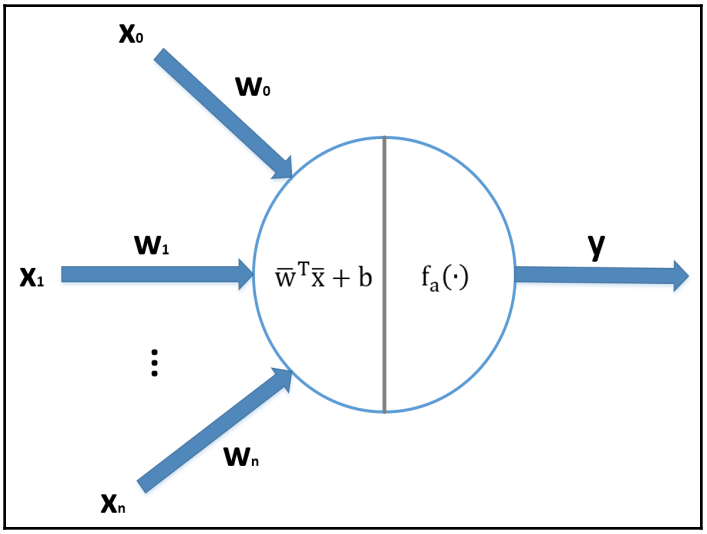
This chapter is the introduction to the world of deep learning, whose methods make it possible to achieve the state-of-the-art performance in many classification and regression fields often considered extremely difficult to manage (such as image segmentation, automatic translation, voice synthesis, and so on). The goal is to provide the reader with the basic instruments to understand the structure of a fully connected neural network and model it using the Python tool Keras (employing all the modern techniques to speed the training process and prevent overfitting).

In particular, the topics covered in the chapter are as follows:

- The structure of a basic artificial neuron
- Perceptrons, linear classifiers, and their limitations
- Multilayer perceptrons with the most important activation functions (such as ReLU)
- Back-propagation algorithms based on **stochastic gradient descent** (**SGD**) optimization method
- Optimized SGD algorithms (Momentum, RMSProp, Adam, AdaGrad, and AdaDelta)
- Regularization and dropout
- Batch normalization

The basic artificial neuron

The building block of a neural network is the abstraction of a biological neuron, a quite simplistic but powerful computational unit that was proposed for the first time by F. Rosenblatt in 1957, to make up the simplest neural architecture, called a perceptron, that we are going to analyze in the next section. Contrary to Hebbian Learning, which is more biologically plausible but has some strong limitations, the artificial neuron has been designed with a pragmatic viewpoint and, of course, only its structure is based on a few elements characterizing a biological cell. However, recent deep learning research activities have unveiled the enormous power of this kind of architecture. Even if there are more complex and specialized computational cells, the basic artificial neuron can be summarized as the conjunction of two blocks, which are clearly shown in the following diagram:



The input of a neuron is a real-valued vector $x \in \mathcal{R}^n$, while the output is a scalar $y \in \mathcal{R}$. The first operation is linear:

$$z = \bar{w}^T \bar{x} + b$$

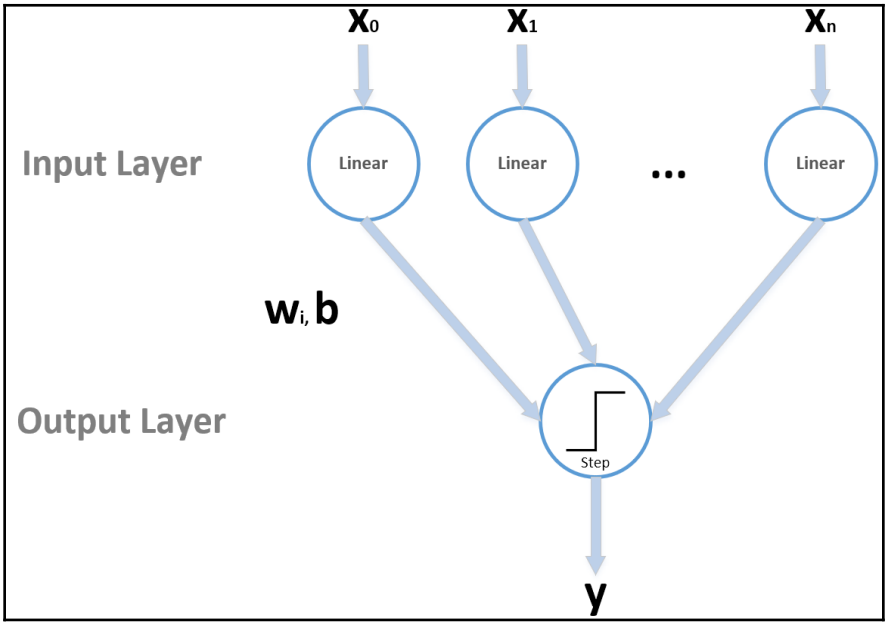
The vector $w \in \mathcal{R}^n$ is called **weight-vector** (or **synaptic weight vector**, because, analogously to a biological neuron, it reweights the input values), while the scalar term $b \in \mathcal{R}$ is a constant called **bias**. In many cases, it's easier to consider only the weight vector. It's possible to get rid of the bias by adding an extra input feature equal to 1 and a corresponding weight:

$$\bar{x}^* = (x_1, x_2, \ldots, x_n, 1)$$

In this way, the only element that must be learned is the weight vector. The following block is called an **activation function**, and it's responsible for remapping the input into a different subset. If the function is $f_a(z) = z$, the neuron is called linear and the transformation can be omitted. The first experiments were based on linear neurons that are much less powerful than non-linear ones, and this was a reason that led many researchers to consider the perceptron as a failure, but, at the same time, this limitation opened the door for a new architecture that, instead, showed its excellent abilities. Let's now start this analysis with the first neural network ever proposed.

Perceptron

Perceptron was the name that Frank Rosenblatt gave to the first neural model in 1957. A perceptron is a neural network with a single layer of input linear neurons, followed by an output unit based on the *sign(•)* function (alternatively, it's possible to consider a bipolar unit whose output is -1 and 1). The architecture of a perceptron is shown in the following diagram:



Even if the diagram can appear as quite complex, a perceptron can be summarized by the following equation:

$$y_i = sign\left(\bar{w}^T \bar{x}_i + b\right) \;\; where \;\; \bar{w}, \bar{x}_i \in \mathbb{R}^n \; and \;\; y_i \in \{0, 1\}$$

All the vectors are conventionally column-vectors; therefore, the dot product $w^T x_i$ transforms the input into a scalar, then the bias is added, and the binary output is obtained using the step function, which outputs 1 when $z > 0$ and 0 otherwise. At this point, a reader could object that the step function is non-linear; however, a non-linearity applied to the output layer is only a filtering operation that has no effect on the actual computation. Indeed, the output is already decided by the linear block, while the step function is employed only to impose a binary threshold. Moreover, in this analysis, we are considering only single-value outputs (even if there are multi-class variants) because our goal is to show the dynamics and also the limitations, before moving to more generic architectures that can be used to solve extremely complex problems.

A perceptron can be trained with an online algorithm (even if the dataset is finite) but it's also possible to employ an offline approach that repeats for a fixed number of iterations or until the total error becomes smaller than a predefined threshold. The procedure is based on the squared error loss function (remember that, conventionally, the term *loss* is applied to single samples, while the term *cost* refers to the sum/average of every single loss):

$$L(\bar{x}_i, y_i; \bar{w}, b) = \frac{1}{2}(\bar{w}^T \bar{x}_i + b - y_i)^2 = \frac{1}{2}(w_1 x_i^{(1)} + w_2 x_i^{(2)} + \ldots + w_n x_i^{(n)} + b - y_i)$$

When a sample is presented, the output is computed, and if it is wrong, a weight correction is applied (otherwise the step is skipped). For simplicity, we don't consider the bias, as it doesn't affect the procedure. Our goal is to correct the weights so as to minimize the loss. This can be achieved by computing the partial derivatives with respect to w_i:

$$\frac{\partial L}{\partial w_j} = (w_j x_i^{(j)} - y_i) x_i^{(j)}$$

Let's suppose that $w^{(0)} = (0, 0)$ (ignoring the bias) and the sample, $x = (1, 1)$, has $y = 1$. The perceptron misclassifies the sample, because $sign(w^T x) = 0$. The partial derivatives are both equal to -1; therefore, if we subtract them from the current weights, we obtain $w^{(1)} = (1, 1)$ and now the sample is correctly classified because $sign(w^T x) = 1$. Therefore, including a learning rate η, the weight update rule becomes as follows:

$$w_i^{(t+1)} = \begin{cases} w_i^{(t)} - \eta(w_j x_i^{(j)} - y_i)x_i^{(j)} & if\ \ sign(\bar{w}^T \bar{x}_i) \neq y_i \\ w_i^{(t)} & otherwise \end{cases}$$

When a sample is misclassified, the weights are corrected proportionally to the difference between actual linear output and true label. This is a variant of a learning rule called the **delta rule**, which represented the first step toward the most famous training algorithm, employed in almost any supervised deep learning scenario (we're going to discuss it in the next sections). The algorithm has been proven to converge to a stable solution in a finite number of states as the dataset is linearly separable. The formal proof is quite tedious and very technical, but the reader who is interested can find it in *Perceptrons, Minsky M. L., Papert S. A., The MIT Press*.

In this chapter, the role of the learning rate becomes more and more important, in particular when the update is performed after the evaluation of a single sample (like in a perceptron) or a small batch. In this case, a high learning rate (that is, one greater than 1.0) can cause an instability in the convergence process because of the magnitude of the single corrections. When working with neural networks, it's preferable to use a small learning rate and repeat the training session for a fixed number of epochs. In this way, the single corrections are limited, and only if they are *confirmed* by the majority of samples/batches, they can become stable, driving the network to converge to an optimal solution. If, instead, the correction is the consequence of an outlier, a small learning rate can limit its action, avoiding destabilizing the whole network only for a few noisy samples. We are going to discuss this problem in the next sections.

Now, we can describe the full perceptron algorithm and close the paragraph with some important considerations:

1. Select a value for the learning rate η (such as `0.1`).
2. Append a constant column (set to `1.0`) to the sample vector X. Therefore $X_b \in \Re^{M \times (n+1)}$.
3. Initialize the weight vector $w \in \Re^{n+1}$ with random values sampled from a normal distribution with a small variance (such as `0.05`).
4. Set an error threshold `Thr` (such as `0.0001`).
5. Set a maximum number of iterations N_i.
6. Set `i = 0`.
7. Set `e = 1.0`.

8. While $i < N_i$ and $e > Thr$:
    1. Set `e = 0.0`.
    2. For $k=1$ to M:
        1. Compute the linear output $l_k = w^T x_k$ and the threshold one $t_k = sign(l_k)$.
        2. If t_k != y_k:
            1. Compute $\Delta w_j = \eta(l_k - y_k)x_k^{(j)}$.
            2. Update the weight vector.
        3. Set e += $(l_k - y_k)^2$ (alternatively it's possible to use the absolute value $|l_k - y_k|$).
    3. Set `e /= M`.

The algorithm is very simple, and the reader should have noticed an analogy with a logistic regression. Indeed, this method is based on a structure that can be considered as a perceptron with a sigmoid output activation function (that outputs a real value that can be considered as a probability). The main difference is the training strategy—in a logistic regression, the correction is performed after the evaluation of a cost function based on the negative log likelihood:

$$L(X, Y; \bar{w}, b) = -log \prod_{i=1}^{M} P(y_i|\bar{x}_i; \bar{w}, b) = -\sum_{i=1}^{M} log\left(P(y_i|\bar{x}_i; \bar{w}, b)\right) =$$

$$-\sum_{i=1}^{M} y_i \left(log\left(\sigma(\bar{w}^T \bar{x}_i + b)\right) + (1 - y_i) log\left(1 - \sigma(\bar{w}^T \bar{x}_i + b)\right)\right)$$

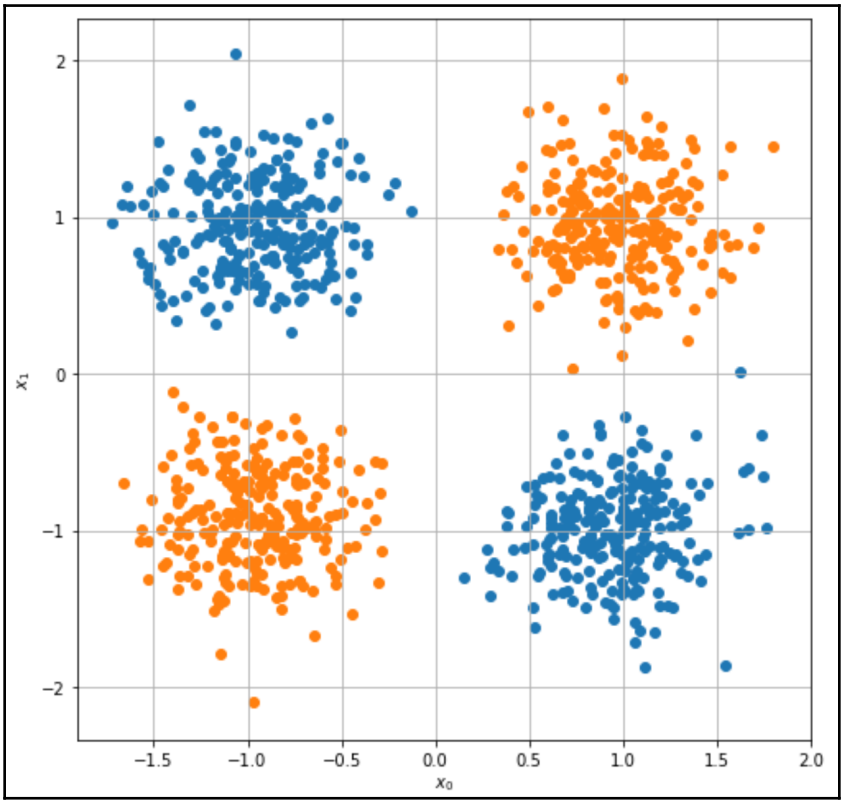
This cost function is the well-known cross-entropy and, in the first chapter, we showed that minimizing it is equivalent to reducing the Kullback-Leibler divergence between the true and predicted distribution. In almost all deep learning classification tasks, we are going to employ it, thanks to its robustness and convexity (this is a convergence guarantee in a logistic regression, but unfortunately the property is normally lost in more complex architectures).

Example of a perceptron with Scikit-Learn

Even if the algorithm is very simple to implement from scratch, I prefer to employ the Scikit-Learn implementation `Perceptron`, so as to focus the attention on the limitations that led to non-linear neural networks. The *historical* problem that showed the main weakness of a perceptron is based on the XOR dataset. Instead of explaining, it's better to build it and visualize the structure:

```
import numpy as np

from sklearn.preprocessing import StandardScaler
from sklearn.utils import shuffle

np.random.seed(1000)

nb_samples = 1000
nsb = int(nb_samples / 4)

X = np.zeros((nb_samples, 2))
Y = np.zeros((nb_samples, ))

X[0:nsb, :] = np.random.multivariate_normal([1.0, -1.0], np.diag([0.1,
0.1]), size=nsb)
Y[0:nsb] = 0.0

X[nsb:(2 * nsb), :] = np.random.multivariate_normal([1.0, 1.0],
np.diag([0.1, 0.1]), size=nsb)
Y[nsb:(2 * nsb)] = 1.0

X[(2 * nsb):(3 * nsb), :] = np.random.multivariate_normal([-1.0, 1.0],
np.diag([0.1, 0.1]), size=nsb)
Y[(2 * nsb):(3 * nsb)] = 0.0

X[(3 * nsb):, :] = np.random.multivariate_normal([-1.0, -1.0],
np.diag([0.1, 0.1]), size=nsb)
Y[(3 * nsb):] = 1.0

ss = StandardScaler()
X = ss.fit_transform(X)

X, Y = shuffle(X, Y, random_state=1000)
```

The plot showing the true labels is shown in the following diagram:



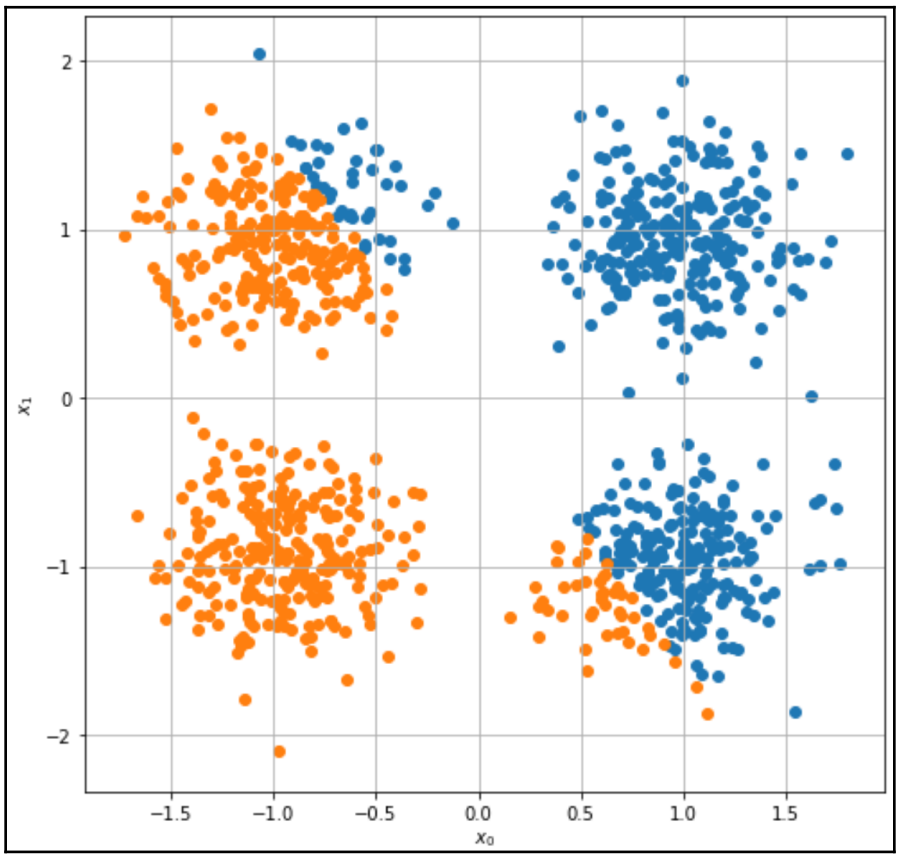
Example of XOR dataset

As it's possible to see, the dataset is split into four blocks that are organized as the output of a logical XOR operator. Considering that the separation hypersurface of a two-dimensional perceptron (as well as the one of a logistic regression) is a line; it's easy to understand that any possible final configuration can achieve an accuracy that is about 50% (a random guess). To have a confirmation, let's try to solve this problem:

```
import numpy as np

from multiprocessing import cpu_count

from sklearn.linear_model import Perceptron
from sklearn.model_selection import cross_val_score

pc = Perceptron(penalty='l2', alpha=0.1, max_iter=1000, n_jobs=cpu_count(),
```

```
random_state=1000)
print(np.mean(cross_val_score(pc, X, Y, cv=10)))
0.498
```

The Scikit-Learn implementation offers the possibility to add a regularization term (see `Chapter 1`, *Machine Learning Models Fundamentals*) through the parameter `penalty` (it can be `'l1'`, `'l2'` or `'elasticnet'`) to avoid overfitting and improve the convergence speed (the strength can be specified using the parameter `alpha`). This is not always necessary, but as the algorithm is offered in a production-ready package, the designers decided to add this feature. Nevertheless, the average cross-validation accuracy is slightly higher than 0.5 (the reader is invited to test any other possible hyperparameter configuration). The corresponding plot (that can change with different random states or subsequent experiments) is shown in the following diagram:



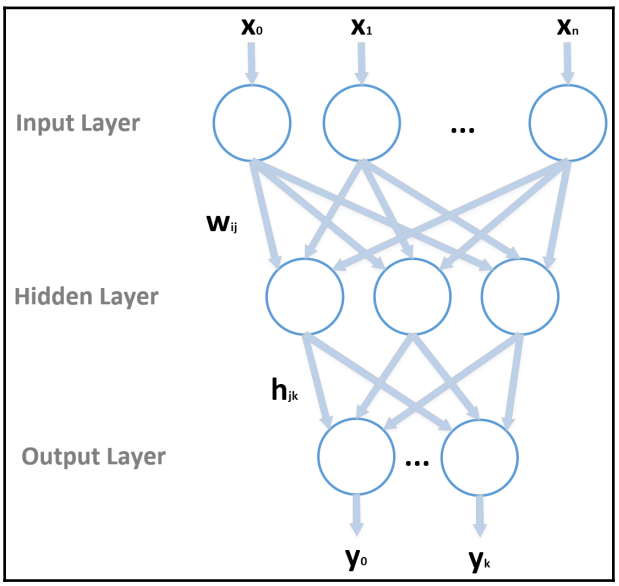
XOR dataset labeled using a perceptron

It's obvious that a perceptron is another linear model without specific peculiarities, and its employment is discouraged in favor of other algorithms like logistic regression or SVM. After 1957, for a few years, many researchers didn't hide their delusion and considered the neural network like a promise never fulfilled. It was necessary to wait until a simple modification to the architecture, together with a powerful learning algorithm, opened officially the door of a new fascinating machine learning branch (later called **deep learning**).

In Scikit-Learn > 0.19, the class `Perceptron` allows adding `max_iter` or `tol` (tolerance) parameters. If not specified, a warning will be issued to inform the reader about the future behavior. This piece of information doesn't affect the actual results.

Multilayer perceptrons

The main limitation of a perceptron is its linearity. How is it possible to exploit this kind of architecture by removing such a constraint? The solution is easier than any speculation. Adding at least a non-linear layer between input and output leads to a highly non-linear combination, parametrized with a larger number of variables. The resulting architecture, called **Multilayer Perceptron** (**MLP**) and containing a single (only for simplicity) **Hidden Layer**, is shown in the following diagram:



This is a so-called **feed-forward network**, meaning that the flow of information begins in the first layer, proceeds always in the same direction and ends at the output layer. Architectures that allow a partial feedback (for example, in order to implement a local memory) are called **recurrent networks** and will be analyzed in the next chapter.

In this case, there are two weight matrices, *W* and *H*, and two corresponding bias vectors, *b* and *c*. If there are *m* hidden neurons, $x_i \in \Re^{n \times 1}$ (column vector), and $y_i \in \Re^{k \times 1}$, the dynamics are defined by the following transformations:

$$\begin{cases} \bar{z} = f_h\left(W^T \bar{x} + \bar{b}\right) \;\; where \;\; W \in \mathbb{R}^{n \times m} \;\; and \;\; \bar{b} \in \mathbb{R}^{m \times 1} \\ \bar{y} = f_a\left(H^T \bar{z} + \bar{c}\right) \;\; where \;\; H \in \mathbb{R}^{m \times k} \;\; and \;\; \bar{c} \in \mathbb{R}^{k \times 1} \end{cases}$$

A fundamental condition for any MLP is that at least one hidden-layer activation function $f_h(\bullet)$ is non-linear. It's straightforward to prove that *m* linear hidden layers are equivalent to a single linear network and, hence, an MLP falls back into the case of a standard perceptron. Conventionally, the activation function is fixed for a given layer, but there are no limitations in their combinations. In particular, the output activation is normally chosen to meet a precise requirement (such as multi-label classification, regression, image reconstruction, and so on). That's why the first step of this analysis concerns the most common activation functions and their features.

Activation functions

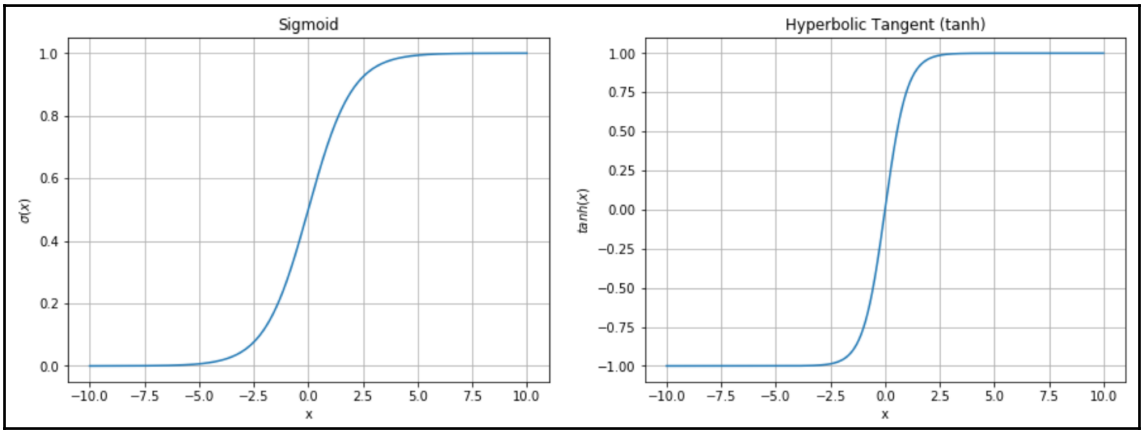
In general, any continuous and differentiable function could be employed as activation; however, some of them have particular properties that allow achieving a good accuracy while improving the learning process speed. They are commonly used in the state-of-the-art models, and it's important to understand their properties in order to make the most reasonable choice.

Sigmoid and hyperbolic tangent

These two activations are very similar but with an important difference. Let's start defining them:

$$f_{Sigmoid}(x) = \frac{1}{1 + e^{-x}} \quad and \quad f_{tanh}(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

The corresponding plots are shown in the following diagram:



Sigmoid and hyperbolic tangent plots

A sigmoid $\sigma(x)$ is bounded between 0 and 1, with two asymptotes ($\sigma(x) \to 0$ *when* $x \to -\infty$ and $\sigma(x) \to 1$ *when* $x \to \infty$). Similarly, the hyperbolic tangent (`tanh`) is bounded between -1 and 1 with two asymptotes corresponding to the extreme values. Analyzing the two plots, we can discover that both functions are almost linear in a short range (about *[-2, 2]*), and they become almost flat immediately after. This means that the gradient is high and about constant when x has small values around 0 and it falls down to about 0 for larger absolute values. A sigmoid perfectly represents a probability or a set of weights that must be bounded between 0 and 1, and therefore, it can be a good choice for some output layers. However, the hyperbolic tangent is completely symmetric, and, for optimization purposes, it's preferable because the performances are normally superior. This activation function is often employed in intermediate layers, whenever the input is normally small. The reason will be clear when the back-propagation algorithm is analyzed; however, it's obvious that large absolute inputs lead to almost constant outputs, and as the gradient is about 0, the weight correction can become extremely slow (this problem is formally known as **vanishing gradients**). For this reason, in many real-world applications, the next family of activation functions is often employed.

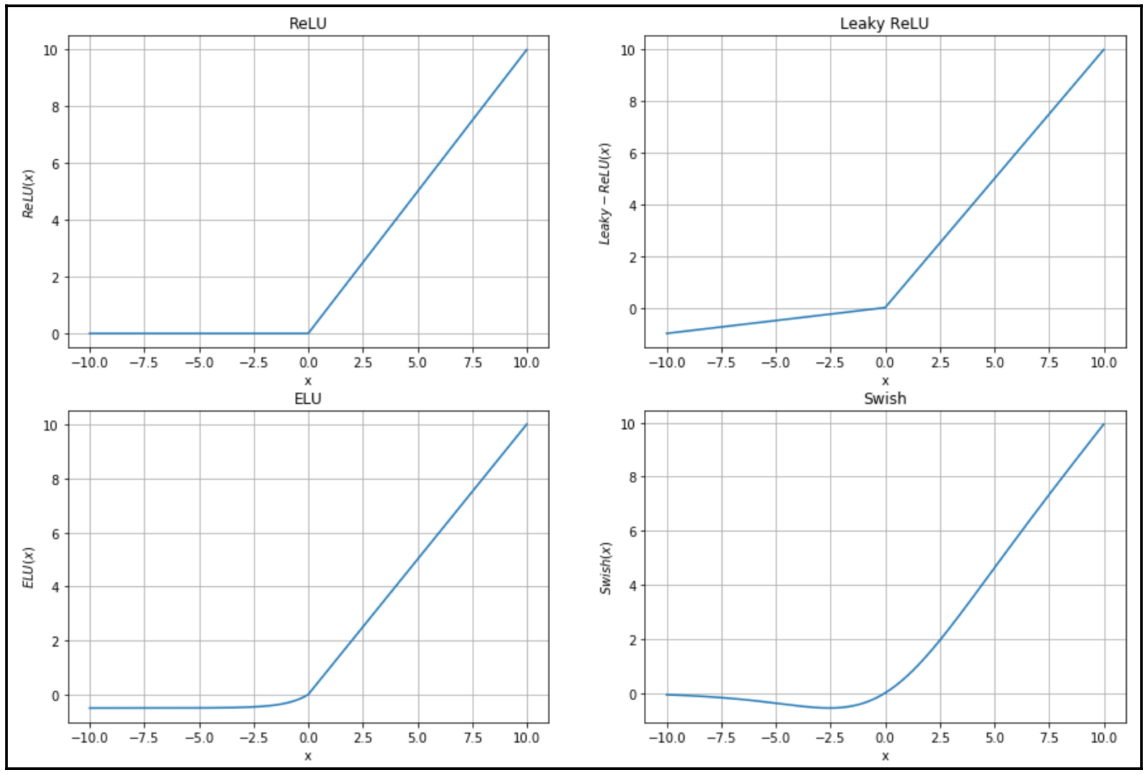
Rectifier activation functions

These functions are all linear (or quasi-linear for Swish) when $x > 0$, while they differ when $x < 0$. Even if some of them are differentiable when $x = 0$, the derivative is set to `0` in this case. The most common functions are as follows:

$$f_{ReLU}(x) = max(0, x) \qquad f_{Leaky-ReLU}(x) = max(0, \alpha x) \;\; if \; \alpha \leqslant 1$$

$$f_{ELU}(x) = \begin{cases} x & if \;\; x > 0 \\ \alpha(e^x - 1) & otherwise \end{cases} \qquad f_{Swish}(x) = \frac{x}{1 + e^{-\alpha x}}$$

The corresponding plots are shown in the following diagram:



The basic function (and also the most commonly employed) is the ReLU, which has a constant gradient when $x > 0$, while it is null for $x < 0$. This function is very often employed in visual processing when the input is normally greater than 0 and has the extraordinary advantage to mitigate the vanishing gradient problem, as a correction based on the gradient is always possible. On the other side, ReLU is null (together with its first derivative) when $x < 0$, therefore every negative input doesn't allow any modification. In general, this is not an issue, but there are some deep networks that perform much better when a small negative gradient was allowed. This consideration drove to the other variants, which are characterized by the presence of the hyperparameter α, that controls the strength of the *negative tail*. Common values between 0.01 and 0.1 allow a behavior that is almost identical to ReLU, but with the possibility of a small weight update when $x < 0$. The last function, called Swish and proposed in *Searching for Activation Functions, Ramachandran P., Zoph P., Le V. L., arXiv:1710.05941 [cs.NE]*, is based on the sigmoid and offers the extra advantage to converge to 0 when $x \rightarrow 0$, so the *non-null effect* is limited to a short region bounded between $[-b, 0]$ with $b > 0$. This function can improve the performance of some particular visual processing deep networks, as discussed in the aforementioned paper. However, I always suggest starting the analysis with ReLU (that is very robust and computationally inexpensive) and switch to an alternative only if no other techniques can improve the performance of a model.

Softmax

This function characterized the output layer of almost all classification networks, as it can immediately represent a discrete probability distribution. If there are k outputs y_i, the softmax is computed as follows:

$$f^{(i)}_{Softmax}(x) = \frac{e^{y_i}}{\sum_{j=1}^{k} e^{y_j}}$$

In this way, the output of a layer containing k neurons is normalized so that the sum is always 1. It goes without saying that, in this case, the best cost function is the cross-entropy. In fact, if all true labels are represented with a one-hot encoding, they implicitly become probability vectors with 1 corresponding to the true class. The goal of the classifier is hence to reduce the discrepancy between the training distribution of its output by minimizing the function (see `Chapter 1`, *Machine Learning Models Fundamentals*, for further information):

$$L(Y; \hat{Y}; \bar{\theta}) = -\sum_{i=1}^{k} y_i log(\hat{y}_i) \;\; where \;\; \hat{y}_i \; is \; a \; prediction$$

Back-propagation algorithm

We can now discuss the training approach employed in an MLP (and almost all other neural networks). This algorithm is more a methodology than an actual one; therefore I prefer to define the main concepts without focusing on a particular case. The reader who is interested in implementing it will be able to apply the same techniques to different kinds of networks with minimal effort (assuming that all requirements are met).

The goal of a training process using a deep learning model is normally achieved by minimizing a cost function. Let's suppose to have a network parameterized with a global vector θ, the cost function (using the same notation for loss and cost but with different parameters to disambiguate) is defined as follows:

$$L(\bar{\theta}) = \frac{1}{M} \sum_{i=1}^{M} L(\bar{x}_i, \bar{y}_i; \bar{\theta})$$

We have already explained that the minimization of the previous expression (which is the empirical risk) is a way to minimize the real expected risk and, therefore, to maximize the accuracy. Our goal is, hence, to find an optimal parameter set so that the following applies:

$$\bar{\theta}_{opt} = argmin_{\bar{\theta}} L(\bar{\theta})$$

If we consider a single loss function (associated with a sample x_i and a true label y_i), we know that such a function can be expressed with an explicit dependence on the predicted value:

$$L(\bar{x}_i, \bar{y}_i; \bar{\theta}) = L(\bar{y}_i, \hat{y}_i)$$

Now, the parameters have been *embedded* into the prediction. From calculus (without an excessive mathematical rigor that can be found in many books about optimization techniques), we know that the gradient of *L*, a scalar function, computed at any point (we are assuming the *L* is differentiable) is a vector with components:

$$\nabla_{\bar{\theta}} L = \left(\frac{\partial L}{\partial \theta_1}, \frac{\partial L}{\partial \theta_2}, \dots, \frac{\partial L}{\partial \theta_T} \right)^T$$

As ∇L points always in the direction of the closest maximum, so the negative gradient points in the direction of the closest minimum. Hence, if we compute the gradient of L, we have a ready-to-use piece of information that can be used to minimize the cost function. Before proceeding, it's useful to expose an important mathematical property called the **chain rule of derivatives**:

$$\frac{\partial f_1(f_2(\dots f_n(x)\dots))}{\partial x} = \frac{\partial f_1}{\partial f_2}\frac{\partial f_2}{\partial f_3}\cdots\frac{\partial f_n}{\partial x}$$

Now, let's consider a single step in an MLP (starting from the bottom) and let's exploit the chain rule:

$$\bar{y} = f_a\left(H^T\bar{z} + \bar{c}\right)$$

Each component of the vector y is independent of the others, so we can simplify the example by considering only an output value:

$$\hat{y}_i = f_a\left(\sum_{j=1}^{k} h_{ji}\bar{z}_j + c_i\right)$$

In the previous expression (discarding the bias), there are two important elements—the weights, h_j (that are the columns of H), and the expression, z_j, which is a function of the previous weights. As L is, in turn, a function of all predictions, y_i, applying the chain rule (using the variable t as the generic argument of the activation functions), we get the following:

$$\frac{\partial L}{\partial h_{ij}} = \frac{\partial L}{\partial \hat{y}_i}\frac{\partial \hat{y}_i}{\partial t}\frac{\partial t}{\partial h_{ij}} = \frac{\partial L}{\partial \hat{y}_i}\frac{\partial \hat{y}_i}{\partial t}\bar{z}_j = \delta_i\bar{z}_j$$

As we normally cope with vectorial functions, it's easier to express this concept using the gradient operator. Simplifying the transformations performed by a generic layer, we can express the relations (with respect to a row of H, so to a weight vector h_i corresponding to a hidden unit, z_i) as follows:

$$\begin{cases} L = p(\bar{y}) \;\; where \;\; \bar{y} \in \mathbb{R}^{k\times 1} \\ \bar{y} = \bar{q}(\bar{h}_i) \;\; where \;\; \bar{h}_i \in \mathbb{R}^{m\times 1} \end{cases}$$

Employing the gradient and considering the vectorial output y can be written as $y = (y_1, y_2, ..., y_m)$, we can derive the following expression:

$$\nabla_{\bar{h}_i} L = J_{\bar{h}_i}(\bar{y})^T \nabla_{\bar{y}} L = \begin{pmatrix} \frac{\partial \bar{y}_1}{\partial \bar{h}_1} & \cdots & \frac{\partial \bar{y}_k}{\partial \bar{h}_1} \\ \vdots & \ddots & \vdots \\ \frac{\partial \bar{y}_1}{\partial \bar{h}_m} & \cdots & \frac{\partial \bar{y}_k}{\partial \bar{h}_m} \end{pmatrix} \begin{pmatrix} \frac{\partial L}{\partial \bar{y}_1} \\ \vdots \\ \frac{\partial L}{\partial \bar{y}_k} \end{pmatrix} = \begin{pmatrix} \frac{\partial L}{\partial \bar{h}_1} \\ \vdots \\ \frac{\partial L}{\partial \bar{h}_m} \end{pmatrix}$$

In this way we get all the components of the gradient of L computed with respect to the weightvectors, h_i. If we move back, we can derive the expression of z_j:

$$\bar{z}_j = f_h \left(\sum_{p=1}^{m} w_{pj} \bar{x}_p + b_j \right)$$

Reapplying the chain rule, we can compute the partial derivative of L with respect to w_{pj} (to avoid confusion, the argument of the prediction y_i is called t_1, while the argument of z_j is called t_2):

$$\frac{\partial L}{\partial w_{pj}} = \frac{\partial L}{\partial \hat{y}_i} \frac{\partial \hat{y}_i}{\partial t_1} \frac{\partial t_1}{\partial \bar{z}_j} \frac{\partial \bar{z}_j}{\partial t_2} \frac{\partial t_2}{\partial w_{pj}} = \delta_i h_{ji} \frac{\partial \bar{z}_j}{\partial t_2} \bar{x}_p$$

Observing this expression (that can be easily rewritten using the gradient) and comparing it with the previous one, it's possible to understand the philosophy of the **back-propagation algorithm**, presented for the first time in *Learning representations by back-propagating errors, Rumelhart D. E., Hinton G. E., Williams R. J., Nature 323/1986*. The samples are fed into the network and the cost function is computed. At this point, the process starts from the bottom, computing the gradients with respect to the closest weights and reusing a part of the calculation δ_i (proportional to the error) to move back until the first layer is reached.

The correction is indeed propagated from the source (the cost function) to the origin (the input layer), and the effect is proportional to the responsibility of each different weight (and bias).

Considering all the possible different architectures, I think that writing all the equations for a single example is useless. The methodology is conceptually simple, and it's purely based on the chain rule of derivatives. Moreover, all existing frameworks, such as Tensorflow, Caffe, CNTK, PyTorch, Theano, and so on, can compute the gradients for all weights of a complete network with a single operation, so as to allow the user to focus attention on more pragmatic problems (like finding the best way to avoid overfitting and improving the training process).

A very important phenomenon that is worth considering was already outlined in the previous section and now it should be clearer: the chain rule is based on multiplications; therefore, when the gradients start to become smaller than 1, the multiplication effect forces the last values to be close to 0. This problem is known as **vanishing gradients** and can really stop the training process of very deep models that use saturating activation functions (like `sigmoid` or `tanh`). Rectifier units provide a good solution to many specific issues, but sometimes when functions like hyperbolic tangent are necessary, other methods, like normalization, must be employed to mitigate the phenomenon. We are going to discuss some specific techniques in this chapter and in the next one, but a generic best practice is to work always with normalized datasets and, if necessary, also testing the effect of whitening.

Stochastic gradient descent

Once the gradients have been computed, the cost function can be *moved* in the direction of its minimum. However, in practice, it is better to perform an update after the evaluation of a fixed number of training samples (batch). Indeed, the algorithms that are normally employed don't compute the global cost for the whole dataset, because this operation could be very computationally expensive. An approximation is obtained with partial steps, limited to the experience accumulated with the evaluation of a small subset. According to some literature, the expression **stochastic gradient descent** (**SGD**) should be used only when the update is performed after every single sample. When this operation is carried out on every *k* sample, the algorithm is also known as **mini-batch gradient descent**; however, conventionally SGD is referred to all batches containing $k \geq 1$ samples, and we are going to use this expression from now on.

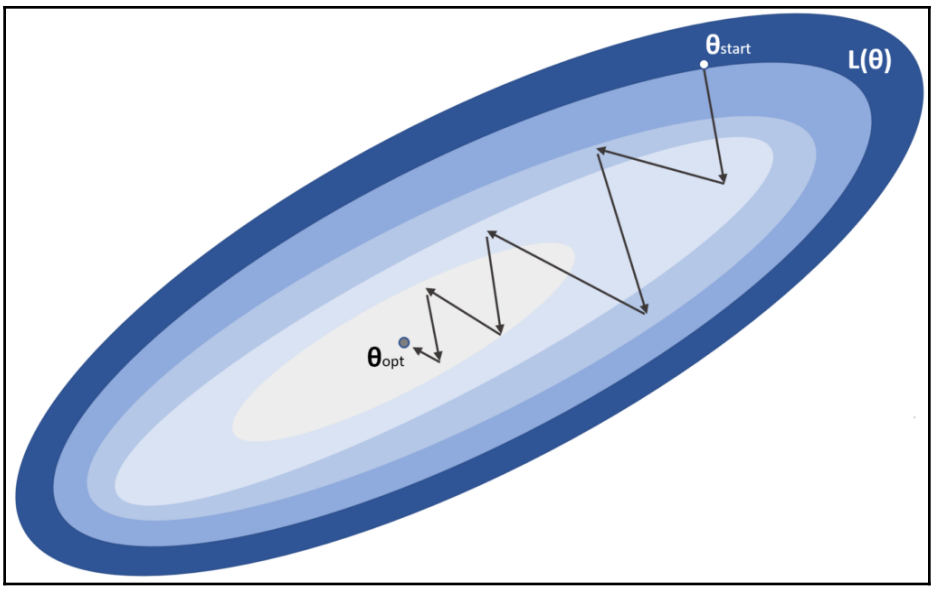
The process can be expressed considering a partial cost function computed using a batch containing *k* samples:

$$L(\bar{\theta}) = \frac{1}{k}\sum_{i=1}^{k} L(\bar{x}_i, \bar{y}_i; \bar{\theta})$$

The algorithm performs a gradient descent by updating the weights according to the following rule:

$$\bar{\theta}^{(t+1)} = \bar{\theta}^{(t)} - \eta \nabla_{\bar{\theta}} L$$

If we start from an initial configuration θ_{start}, the stochastic gradient descent process can be imagined like the path shown in the following diagram:



The weights are moved towards the minimum $\theta_{opt,}$ with many subsequent corrections that could also be wrong considering the whole dataset. For this reason, the process must be repeated several times (epochs), until the validation accuracy reaches its maximum. In a perfect scenario, with a convex cost function L, this simple procedure converges to the optimal configuration. Unfortunately, a deep network is a very complex and non-convex function where plateaus and saddle points are quite common (see `Chapter 1`, *Machine Learning Models Fundamentals*). In such a scenario, a *vanilla* SGD wouldn't be able to find the global optimum and, in many cases, could not even find a close point. For example, in flat regions, the gradients can become so small (also considering the numerical imprecisions) as to slow down the training process until no change is possible (so $\theta^{(t+1)} \approx \theta^{(t)}$). In the next section, we are going to present some common and powerful algorithms that have been developed to mitigate this problem and dramatically accelerate the convergence of deep models.

Before moving on, it's important to mark two important elements. The first one concerns the learning rate, η. This hyperparameter plays a fundamental role in the learning process. As also shown in the figure, the algorithm proceeds jumping from a point to another one (which is not necessarily closer to the optimum). Together with the optimization algorithms, it's absolutely important to correctly tune up the learning rate. A high value (such as 1.0) could move the weights too rapidly increasing the instability. In particular, if a batch contains a few outliers (or simply non-dominant samples), a large η will consider them as representative elements, correcting the weights so to minimize the error. However, subsequent batches could better represent the data generating process, and, therefore, the algorithm must partially *revert* its modifications in order to compensate the wrong update. For this reason, the learning rate is usually quite small with common values bounded between 0.0001 and 0.01 (in some particular cases, $\eta = 0.1$ can be also a valid choice). On the other side, a very small learning rate leads to minimum corrections, slowing down the training process. A good trade-off, which is often the best practice, is to let the learning rate decay as a function of the epoch. In the beginning, η can be higher, because the probability to be close to the optimum is almost null; so, larger jumps can be easily adjusted. While the training process goes on, the weights are progressively moved towards their final configuration and, hence, the corrections become smaller and smaller. In this case, large jumps should be avoided, preferring a fine-tuning. That's why the learning rate is decayed. Common techniques include the exponential decay or a linear one. In both cases, the initial and final values must be chosen according to the specific problem (testing different configurations) and the optimization algorithm. In many cases, the ratio between the start and end value is about 10 or even larger.

Another important hyperparameter is the batch size. There are no silver bullets that allow us to automatically make the right choice, but some considerations can be made. As SGD is an approximate algorithm, larger batches drive to corrections that are probably more similar to the ones obtained considering the whole dataset. However, when the number of samples is extremely high, we don't expect the deep model to map them with a one-to-one association, but instead our efforts are directed to improving the generalization ability. This feature can be re-expressed saying that the network must learn a smaller number of abstractions and reuse them in order to classify new samples. A batch, if sampled correctly, contains a part of these *abstract elements* and part of the corrections automatically improve the evaluation of a subsequent batch. You can imagine a waterfall process, where a new training step never starts from scratch. However, the algorithm is also called mini-batch gradient descent, because the usual batch size normally ranges from 16 to 512 (larger sizes are uncommon, but always possible), which are values smaller than the number of total samples (in particular in deep learning contexts). A reasonable default value could be 32 samples, but I always invite the reader to test larger values, comparing the performances in terms of training speed and final accuracy.

When working with deep neural networks, all the values (number of neurons in a layer, batch size, and so on) are normally powers of two. This is not a constraint, but only an optimization tip (above all when using GPUs), as the memory can be more efficiently filled when the blocks are based on a 2^N elements. However, this is only a suggestion, whose benefits could also be negligible; so, don't be afraid to test architectures with different values. For example, in many papers, the batch size is 100 or some layers have 1,000 neurons.

Weight initialization

A very important element is the initial configuration of a neural network. How should the weights be initialized? Let's imagine we that have set them all to zero. As all neurons in a layer receive the same input, if the weights are 0 (or any other common, constant number), the output will be equal. When applying the gradient correction, all neurons will be treated in the same way; so, the network is equivalent to a sequence of single neuron layers. It's clear that the initial weights must be different to achieve a goal called symmetry breaking, but which is the best choice?

If we knew (also approximately) the final configuration, we could set them to easily reach the optimal point in a few iterations, but, unfortunately, we have no idea where the minimum is located. Therefore, some empirical strategies have been developed and tested, with the goal of minimizing the training time (obtaining state-of-the-art accuracies). A general rule of thumb is that the weights should be small (compared to the input sample variance). Large values lead to large outputs that negatively impact on saturating functions (such as `tanh` and `sigmoid`), while small values can be more easily optimized because the corresponding gradients are larger and the corrections have a stronger effect. The same is true also for rectifier units because the maximum efficiency is achieved by working in a segment crossing the origin (where the non-linearity is actually *located*). For example, when coping with images, if the values are positive and large, a ReLU neuron becomes almost a linear unit, losing a lot of its advantages (that's why images are normalized, so as to bound each pixel value between 0 and 1 or -1 and 1).

At the same time, ideally, the activation variances should remain almost constant throughout the network, as well as the weight variances after every back-propagation step. These two conditions are fundamental in order to improve the convergence process and to avoid the vanishing and exploding gradient problems (the latter, which is the opposite of vanishing gradients, will be discussed in the section dedicated to recurrent network architectures).

A very common strategy considers the number of neurons in a layer and initializes the weights as follows:

$$w_{ij} \sim N\left(0, \frac{1}{n}\right)$$

This method is called **variance scaling** and can be applied using the number of input units (Fan-In), the number of output units (Fan-Out), or their average. The idea is very intuitive: if the number of incoming or outgoing connections is large, the weights must be smaller, so as to avoid large outputs. In the degenerate case of a single neuron, the variance is set to `1.0`, which is the maximum value allowed(in general, all methods keep the initial values for the biases equal to 0.0 because it's not necessary to initialize them with a random value).

Other variations have been proposed, even if they all share the same basic ideas. **LeCun** proposed initializing the weights as follows:

$$w_{ij} \sim U\left(-\sqrt{\frac{3}{n_{Fan-In}}}, \sqrt{\frac{3}{n_{Fan-In}}}\right)$$

Another method called **Xavier initialization** (presented in *Understanding the difficulty of training deep feedforward neural networks, Glorot X., Bengio Y., Proceedings of the 13th International Conference on Artificial Intelligence and Statistics*), is similar to **LeCun initialization**, but it's based on the average between the number of units of two consecutive layers (to mark the sequentiality, we have substituted the terms Fan-In and Fan-Out with explicit indices):

$$w_{ij} \sim U\left(-\sqrt{\frac{6}{n_k + n_{k+1}}}, \sqrt{\frac{6}{n_k + n_{k+1}}}\right)$$

This is a more robust variant, as it considers both the incoming connections and also the outgoing ones (which are in turn incoming connections). The goal (widely discussed by the authors in the aforementioned papers) is trying to meet the two previously presented requirements. The first one is to avoid oscillations in the variance of the activations of each layer (ideally, this condition can avoid saturation). The second one is strictly related to the back-propagation algorithm, and it's based on the observation that, when employing a variance scaling (or an equivalent uniform distribution), the variance of a weight matrix is proportional to the reciprocal of $3n_k$.

Therefore, the averages of Fan-In and Fan-Out are multiplied by three, trying to avoid large variations in the weights after the updates. Xavier initialization has been proven to be very effective in many deep architectures, and it's often the default choice.

Other methods are based on a different way to measure the variance during both the feed-forward and back-propagation phases and trying to correct the values to minimize residual oscillations in specific contexts. For example, He, Zhang, Ren, and Sun (in *Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification, He K., Zhang X., Ren S., Sun J., arXiv:1502.01852 [cs.CV]*) analyzed the initialization problem in the context of convolutional networks (we are going to discuss them in the next chapter) based on ReLU or variable Leaky-ReLU activations (also known as PReLU, parametric ReLU), deriving an optimal criterion (often called the **He initializer**), which is slightly different from the Xavier initializer:

$$w_{ij} \sim U\left(-\sqrt{\frac{6}{n_{Fan-In}}}, \sqrt{\frac{6}{n_{Fan-In}}}\right)$$

All these methods share some common principles and, in many cases, they are interchangeable. As already mentioned, Xavier is one of the most robust and, in the majority of real-life problems, there's no need to look for other methods; however, the reader should be always aware that the complexity of deep models must be often faced using empirical methods based on sometimes simplistic mathematical assumptions. Only the validation with real dataset can confirm if a hypothesis is correct or it's better to continue the investigation in another direction.

Example of MLP with Keras

Keras (`https://keras.io`) is a powerful Python toolkit that allows modeling and training complex deep learning architectures with minimum effort. It relies on low-level frameworks, such as Tensorflow, Theano, or CNTK, and provides high-level blocks to build the single layers of a model. In this book, we need to be very pragmatic because there's no room for a complete explanation; however, all the examples will be structured to allow the reader to try different configurations and options without a full knowledge (for further details, I suggest the book *Deep Learning with Keras, Gulli A, Pal S., Packt Publishing*).

In this example, we want to build a small MLP with a single hidden layer to solve the XOR problem (the dataset is the same created in the previous example). The simplest and most common way is to instantiate the class `Sequential`, which defines an *empty container* for an indefinite model. In this initial part, the fundamental method is `add()`, which allows adding a layer to the model. For our example, we want to employ four hidden layers with hyperbolic tangent activation and two softmax output layers. The following snippet defines the MLP:

```
from keras.models import Sequential
from keras.layers import Dense, Activation

model = Sequential()

model.add(Dense(4, input_dim=2))
model.add(Activation('tanh'))

model.add(Dense(2))
model.add(Activation('softmax'))
```

The `Dense` class defines a fully connected layer (a *classical* MLP layer), and the first parameter is used to declare the number of desired units. The first layer must declare the `input_shape` or `input_dim`, which specify the dimensions (or the shape) of a single sample (the batch size is omitted as it's dynamically set by the framework). All the subsequent layers compute the dimensions automatically. One of the strengths of Keras is the possibility to avoid setting many parameters (like weight initializers), as they will be automatically configured using the most appropriate default values (for example, the default weight initializer is Xavier). In the next examples, we are going to explicitly set some of them, but I suggest that the reader checks the official documentation to get acquainted with all the possibilities and features. The other layer involved in this experiment is `Activation`, which specifies the desired activation function (it's also possible to declare it using the parameter `activation` implemented by almost all layers, but I prefer to decouple the operations to emphasize the single roles, and also because some techniques—such as batch normalization—are normally applied to the linear output, before the activation).

At this point, we must ask Keras to compile the model (using the preferred backend):

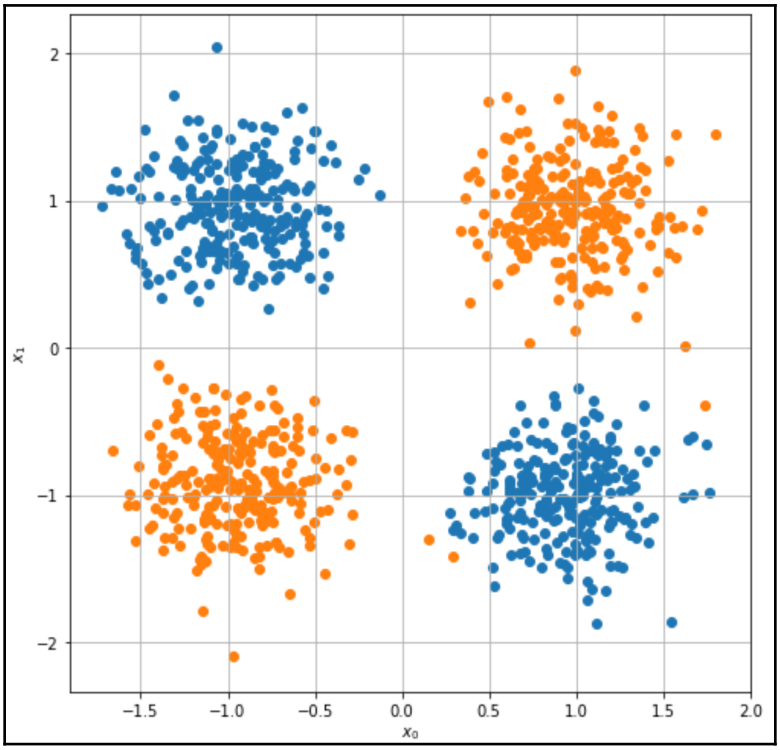
```
model.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
```

The parameter `optimizer` defines the stochastic gradient descent algorithm that we want to employ. Using `optimizer='sgd'`, it's possible to implement a standard version (as described in the previous paragraph). In this case, we are employing Adam (with the default parameters), which is a much more performant variant that will be discussed in the next section. The parameter `loss` is used to define the cost function (in this case, cross-entropy) and `metrics` is a list of all the evaluation score we want to be computed (`'accuracy'` is enough for many classification tasks). Once the model is compiled, it's possible to train it:

```
from keras.utils import to_categorical

from sklearn.model_selection import train_test_split

X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.3,
random_state=1000)

model.fit(X_train,
          to_categorical(Y_train, num_classes=2),
          epochs=100,
          batch_size=32,
          validation_data=(X_test, to_categorical(Y_test, num_classes=2)))

Train on 700 samples, validate on 300 samples
Epoch 1/100
700/700 [==============================] - 1s 2ms/step - loss: 0.7227 -
acc: 0.4929 - val_loss: 0.6943 - val_acc: 0.5933
Epoch 2/100
700/700 [==============================] - 0s 267us/step - loss: 0.7037 -
acc: 0.5371 - val_loss: 0.6801 - val_acc: 0.6100
Epoch 3/100
700/700 [==============================] - 0s 247us/step - loss: 0.6875 -
acc: 0.5871 - val_loss: 0.6675 - val_acc: 0.6733

...

Epoch 98/100
700/700 [==============================] - 0s 236us/step - loss: 0.0385 -
acc: 0.9986 - val_loss: 0.0361 - val_acc: 1.0000
Epoch 99/100
700/700 [==============================] - 0s 261us/step - loss: 0.0378 -
acc: 0.9986 - val_loss: 0.0355 - val_acc: 1.0000
Epoch 100/100
700/700 [==============================] - 0s 250us/step - loss: 0.0371 -
acc: 0.9986 - val_loss: 0.0347 - val_acc: 1.0000
```

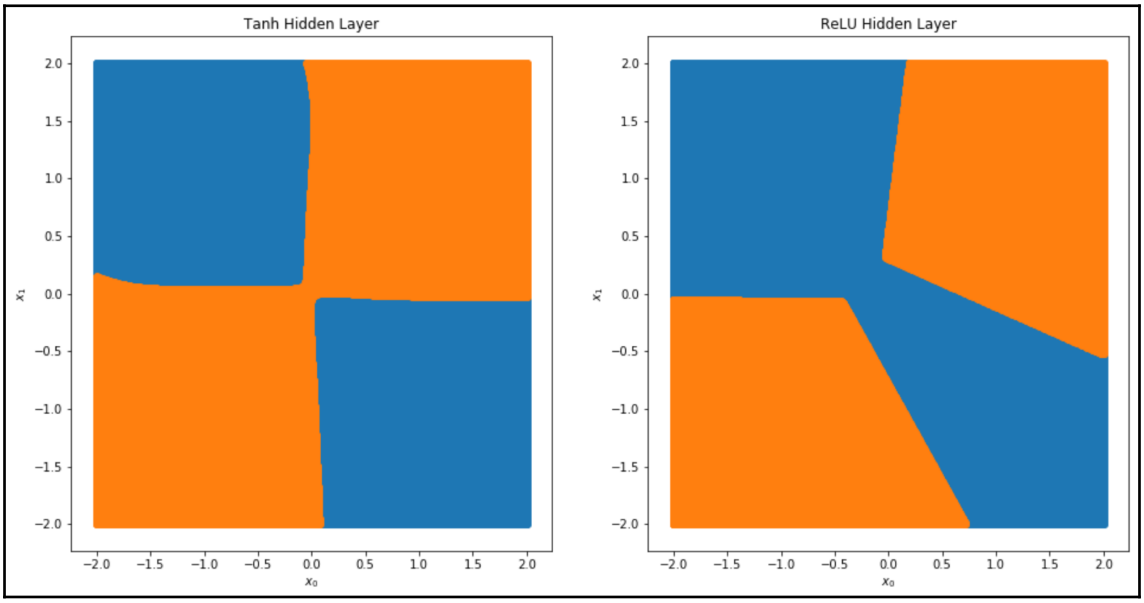
The operations are quite simple. We have split the dataset into training and test/validation sets (in deep learning, cross-validation is seldom employed) and, then, we have trained the model setting `batch_size=32` and `epochs=100`. The dataset is automatically shuffled at the beginning of each epoch, unless setting `shuffle=False`. In order to convert the discrete labels into one-hot encoding, we have used the utility function `to_categorical`. In this case, the label 0 becomes (1, 0) and the label 1 (0, 1). The model converges before reaching 100 epochs; therefore, I invite the reader to optimize the parameters as an exercise. However, at the end of the process, the training accuracy is about 0.999 and the validation accuracy is 1.0.

The final classification plot is shown in the following diagram:



MLP classification of the XOR dataset

Only three points have been misclassified, but it's clear that the MLP successfully separated the XOR dataset. To have a confirmation of the generalization ability, we've plotted the decision surfaces for a hyperbolic tangent hidden layer and ReLU one:



MLP decision surfaces with Tanh (left) and ReLU (right) hidden layer

In both cases, the MLPs delimited the areas in a reasonable way. However, while a `tanh` hidden layer seems to be overfitted (this is not true in our case, as the dataset represents exactly the data generating process), the ReLU layer generates less smooth boundaries with an apparent lower variance (in particular for considering the outliers of a class). We know that the final validation accuracies confirm an almost perfect fit, and the decision plots (which is easy to create with two dimensions) show in both cases acceptable boundaries, but this simple exercise is useful to understand the complexity and the sensitivity of a deep model. For this reason, it's absolutely necessary to select a valid training set (representing the ground-truth) and employ all possible techniques to avoid the overfitting (as we're going to discuss later). The easiest way to detect such a situation is checking the validation loss. A good model should reduce both training and validation loss after each epoch, reaching a plateau for the latter. If, after *n* epochs, the validation loss (and, consequently, the accuracy) begins to increase, while the training loss keeps decreasing, it means that the model is overfitting the training set.

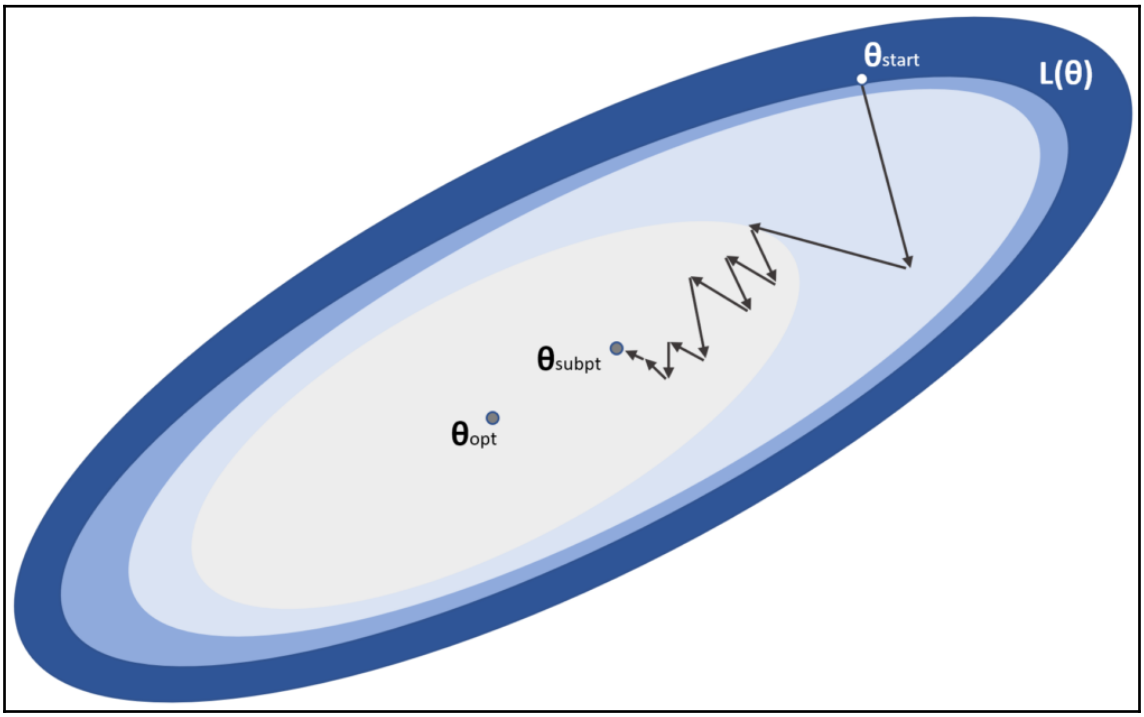
Another empirical indicator that the training process is evolving correctly is that, at least at the beginning, the validation accuracy should be higher than the training one. This can seem strange, but we need to consider that the validation set is slightly smaller and less complex than the training set; therefore, if the capacity of the model is not saturated with training samples, the probability of misclassification is higher for the training set than for the validation set. When this trend is inverted, the model is very likely to overfit after a few epochs. To verify these concepts, I invite the reader to repeat the exercise using a large number of hidden neurons (so as to increase dramatically the capacity), but they will be clearer when working with much more complex and unstructured datasets.

Keras can be installed using the command `pip install -U keras`. The default framework is Theano with CPU support. In order to use other frameworks (such as Tensorflow GPU), I suggest reading the instructions reported on the home page `https://keras.io`. As also suggested by the author, the best backend is Tensorflow, which is available for Linux, Mac OSX, and Windows. To install it (together with all dependencies), please follow the instructions on the following page: `https://www.tensorflow.org/install/`

Optimization algorithms

When discussing the back-propagation algorithm, we have shown how the SGD strategy can be easily employed to train deep networks with large datasets. This method is quite robust and effective; however, the function to optimize is generally non-convex and the number of parameters is extremely large. These conditions increase dramatically the probability to find saddle points (instead of local minima) and can slow down the training process when the surface is almost flat.

A common result of applying a *vanilla* SGD algorithm to these systems is shown in the following diagram:



Instead of reaching the optimal configuration, θ_{opt}, the algorithm reaches a sub-optimal parameter configuration, $\theta_{subopt,}$ and loses the ability to perform further corrections. To mitigate all these problems and their consequences, many SGD optimization algorithms have been proposed, with the purpose of speeding up the convergence (also when the gradients become extremely small) and avoiding the instabilities of ill-conditioned systems.

Gradient perturbation

A common problem arises when the hypersurface is flat (plateaus) the gradients become close to zero. A very simple way to mitigate this problem is based on adding a small homoscedastic noise component to the gradients:

$$\nabla_{\bar{\theta}} L^n = \nabla_{\bar{\theta}} L + \bar{n}(t) \;\; where \;\; \bar{n}(t) \sim N(0, \Sigma(t))$$

The covariance matrix is normally diagonal with all elements set to $\sigma^2(t)$, and this value is decayed during the training process to avoid perturbations when the corrections are very small. This method is conceptually reasonable, but its implicit randomness can yield undesired effects when the noise component is dominant. As it's very difficult to tune up the variances in deep models, other (more deterministic) strategies have been proposed.

Momentum and Nesterov momentum

A more robust way to improve the performance of SGD when plateaus are encountered is based on the idea of momentum (analogously to physical momentum). More formally, a momentum is obtained employing the weighted moving average of subsequent gradient estimations instead of the punctual value:

$$\bar{v}^{(t+1)} = \mu \bar{v}^{(t)} - \eta \nabla_{\bar{\theta}} L$$

The new vector $v^{(t)}$, contains a component which is based on the past history (and weighted using the parameter μ which is a forgetting factor) and a term referred to the current gradient estimation (multiplied by the learning rate). With this approach, abrupt changes become more difficult, and when the exploration leaves a sloped region to enter a plateau, the momentum doesn't become immediately null (but for a time proportional to μ) a portion of the previous gradients will be kept, making it possible to traverse flat regions. The value assigned to the hyperparameter μ is normally bounded between 0 and 1. Intuitively, small values imply a short memory as the first term decays very quickly, while values close to 1.0 (for example, 0.9) allow a longer memory, less influenced by local oscillations. Like for many other hyperparameters, μ needs to be tuned according to the specific problem, considering that a high momentum is not always the best choice. High values could slow down the convergence when very small adjustments are needed, but, at the same time, values close to 0.0 are normally ineffective because the memory contribution decays too early. Using momentum, the update rule becomes as follows:

$$\bar{\theta}^{(t+1)} = \bar{\theta}^{(t)} + \bar{v}^{(t+1)}$$

A variant is provided by **Nesterov momentum**, which is based on the results obtained in the field of mathematical optimization by Nesterov that have been proven to speed up the convergence of many algorithms. The idea is to determine a temporary parameter update based on the current momentum and then apply the gradient to this vector to determine the next momentum (it can be interpreted as a *look-ahead* gradient evaluation aimed to mitigate the risk of a wrong correction considering the moving history of each parameter):

$$\begin{cases} \bar{\theta}_N^{(t+1)} = \bar{\theta}^{(t)} + \mu \bar{v}^{(t)} \\ \bar{v}^{(t+1)} = \mu \bar{v}^{(t)} - \eta \nabla_{\bar{\theta}} L \left(\bar{\theta}_N^{(t+1)} \right) \end{cases}$$

This algorithm showed a performance improvement in several deep models; however, its usage is still limited because the next algorithms very soon outperformed the standard SGD with momentum, and they became the first choice in almost any real-life task.

SGD with momentum in Keras

When using Keras, it's possible to customize the SGD optimizer by directly instantiating the `SGD` class and using it while compiling the model:

```
from keras.optimizers import SGD

...

sgd = SGD(lr=0.0001, momentum=0.8, nesterov=True)

model.compile(optimizer=sgd,
              loss='categorical_crossentropy',
              metrics=['accuracy'])
```

The class SGD accepts the parameter `lr` (the learning rate η with a default set to `0.01`), `momentum` (the parameter μ), `nesterov` (a boolean indicating whether employing the Nesterov momentum), and an optional `decay` parameter to indicate whether the learning rate must be decayed over the updates with the following formula:

$$\eta^{(t+1)} = \frac{\eta^{(t)}}{1 + decay}$$

RMSProp

RMSProp was proposed by Hinton as an adaptive algorithm, partially based on the concept of momentum. Instead of considering the whole gradient vector, it tries to optimize each parameter separately to increase the corrections of slowly changing weights (that probably need more drastic modifications) and decreasing the update magnitudes of quickly changing ones (which are normally the more unstable). The algorithm computes the exponentially weighted moving average of the *changing speed* of every parameter considering the square of the gradient (which is insensitive to the sign):

$$\bar{v}^{(t+1)}(\bar{\theta}_i) = \mu \bar{v}^{(t)} + (1-\mu)\left(\nabla_{\bar{\theta}} L(\bar{\theta}_i)\right)^2$$

The weight update is then performed, as follows:

$$\bar{\theta}_i^{(t+1)} = \bar{\theta}_i^{(t)} - \frac{\eta}{\sqrt{\bar{v}^{(t+1)}(\bar{\theta}_i) + \delta}} \nabla_{\bar{\theta}} L(\bar{\theta}_i)$$

The parameter δ is a small constant (such as 10^{-6}) that is added to avoid numerical instabilities when the changing speed becomes null. The previous expression could be rewritten in a more compact way:

$$\bar{\theta}_i^{(t+1)} = \bar{\theta}_i^{(t)} - \eta(\bar{\theta}_i) \nabla_{\bar{\theta}} L(\bar{\theta}_i)$$

Using this notation, it is clear that the role of RMSProp is adapting the learning rate for every parameter so it can increase it when necessary (almost *frozen* weights) and decrease it when the risk of oscillations is higher. In a practical implementation, the learning rate is always decayed over the epochs using an exponential or linear function.

RMSProp with Keras

The following snippet shows the usage of RMSProp with Keras:

```
from keras.optimizers import RMSprop

...

rms_prop = RMSprop(lr=0.0001, rho=0.8, epsilon=1e-6, decay=1e-2)

model.compile(optimizer=rms_prop,
```

```
loss='categorical_crossentropy',
metrics=['accuracy'])
```

The learning rate and decay are the same as SGD. The parameter `rho` corresponds to the exponential moving average weight, μ, and `epsilon` is the constant added to the changing speed to improve the stability. As with any other algorithm, if the user wants to use the default values, it's possible to declare the optimizer without instantiating the class (for example, `optimizer='rmsprop'`).

Adam

Adam (the contraction of Adaptive Moment Estimation) is an algorithm proposed by Kingma and Ba (in *Adam: A Method for Stochastic Optimization, Kingma D. P., Ba J., arXiv:1412.6980 [cs.LG])* to further improve the performance of RMSProp. The algorithm determines an adaptive learning rate by computing the exponentially weighted averages of both the gradient and its square for every parameter:

$$\begin{cases} \bar{g}^{(t+1)}(\bar{\theta}_i) = \mu_1 \bar{g}^{(t)}(\bar{\theta}_i) + (1-\mu_1)\nabla_{\bar{\theta}} L(\bar{\theta}_i) \\ \bar{v}^{(t+1)}(\bar{\theta}_i) = \mu_2 \bar{v}^{(t)}(\bar{\theta}_i) + (1-\mu_2)\left(\nabla_{\bar{\theta}} L(\bar{\theta}_i)\right)^2 \end{cases}$$

In the aforementioned paper, the authors suggest to unbias the two estimations (which concern the first and second moment) by dividing them by *1 - μ_i*, so the new moving averages become as follows:

$$\begin{cases} \hat{g}^{(t+1)}(\bar{\theta}_i) = \frac{\bar{g}^{(t+1)}(\bar{\theta}_i)}{1-\mu_1} \\ \hat{v}^{(t+1)}(\bar{\theta}_i) = \frac{\bar{v}^{(t+1)}(\bar{\theta}_i)}{1-\mu_2} \end{cases}$$

The weight update rule for Adam is as follows:

$$\bar{\theta}_i^{(t+1)} = \bar{\theta}_i^{(t)} - \frac{\eta g^{(t+1)}(\bar{\theta}_i)}{\sqrt{v^{(t+1)}(\bar{\theta_1}) + \delta}}$$

Analyzing the previous expression, it is possible to understand why this algorithm is often called RMSProp with momentum. In fact, the term $g(\bullet)$ acts just like the standard momentum, computing the moving average of the gradient for each parameter (with all the advantages of this procedure), while the denominator acts as an adaptive term with the same exact semantics of RMSProp. For this reason, Adam is very often one of the most widely employed algorithms, even if, in many complex tasks, its performances are comparable to a standard RMSProp. The choice must be made considering the extra complexity due to the presence of two forgetting factors. In general, the default values (0.9) are acceptable, but sometimes it's better to perform an analysis of several scenarios before deciding on a specific configuration. Another important element to remember is that all momentum based methods can lead to instabilities (oscillations) when training some deep architectures. That's why RMSProp is very diffused in almost any research paper; however, don't consider this statement as a limitation, because Adam has shown outstanding performances in many tasks. It's helpful to remember that, whenever the training process seems unstable also with low learning rates, it's preferable to employ methods that are not based on momentum (the inertial term, in fact, can slow down the fast modifications necessary to avoid oscillations).

Adam with Keras

The following snippet shows the usage of Adam with Keras:

```
from keras.optimizers import Adam

...

adam = Adam(lr=0.0001, beta_1=0.9, beta_2=0.9, epsilon=1e-6, decay=1e-2)

model.compile(optimizer=adam,
              loss='categorical_crossentropy',
              metrics=['accuracy'])
```

The forgetting factors, μ_1 and μ_2, are represented by the parameters `beta_1` and `beta_2`. All the other elements are the same as the other algorithms.

AdaGrad

This algorithm has been proposed by Duchi, Hazan, and Singer (in *Adaptive Subgradient Methods for Online Learning and Stochastic Optimizatioln, Duchi J., Hazan E., Singer Y., Journal of Machine Learning Research 12/2011).* The idea is very similar to RMSProp, but, in this case, the whole history of the squared gradients is taken into account:

$$\bar{v}^{(t+1)}(\bar{\theta}_i) = \bar{v}^{(t)}(\bar{\theta}_i) + \left(\nabla_{\bar{\theta}} L(\bar{\theta}_i)\right)^2$$

The weights are updated exactly like in RMSProp:

$$\bar{\theta}_i^{(t+1)} = \bar{\theta}_i^{(t)} - \frac{\eta}{\sqrt{\bar{v}^{(t+1)}(\bar{\theta}_i) + \delta}} \nabla_{\bar{\theta}} L(\bar{\theta}_i)$$

However, as the squared gradients are non-negative, the implicit sum $v^{(t)}(\bullet) \rightarrow \infty$ when $t \rightarrow \infty$. As the growth continues until the gradients are non-null, there's no way to keep the contribution stable while the training process proceeds. The effect is normally quite strong at the beginning, but vanishes after a limited number of epochs, yielding a null learning rate. **AdaGrad** keeps on being a powerful algorithm when the number of epochs is very limited, but it cannot be a first-choice solution for the majority of deep models (the next algorithm has been proposed to solve this problem).

AdaGrad with Keras

The following snippet shows the use of AdaGrad with Keras:

```
from keras.optimizers import Adagrad

...

adagrad = Adagrad(lr=0.0001, epsilon=1e-6, decay=1e-2)

model.compile(optimizer=adagrad,
              loss='categorical_crossentropy',
              metrics=['accuracy'])
```

The AdaGrad implementation has no other parameters but the common ones.

AdaDelta

AdaDelta is an algorithm (proposed in *ADADELTA: An Adaptive Learning Rate Method, Zeiler M. D., arXiv:1212.5701 [cs.LG]*) in order to address the main issue of AdaGrad, which arises to managing the whole squared gradient history. First of all, instead of the accumulator, AdaDelta employs an exponentially weighted moving average, like RMSProp:

$$\bar{v}^{(t+1)}(\bar{\theta}_i) = \mu\bar{v}^{(t)} + (1-\mu)\big(\nabla_{\bar{\theta}} L(\bar{\theta}_i)\big)^2$$

However, the main difference with RMSProp is based on the analysis of the update rule. When we consider the operation $x + \Delta x$, we assume that both terms have the same unit; however, the author noticed that the adaptive learning rate $\eta(\theta_i)$ obtained with RMSProp (as well as AdaGrad) is unitless (instead of having the unit of θ_i). In fact, as the gradient is split into partial derivatives that can be approximated as $\Delta L/\Delta\theta_i$ and the cost function L is assumed to be unitless, we obtain the following relations:

$$unit_{\nabla_{\bar{\theta}} L(\bar{\theta}_i)} = \frac{1}{unit_{\bar{\theta}_i}} \quad and \quad unit_{\Delta\bar{\theta}_i} \propto \frac{unit_{\nabla_{\bar{\theta}} L(\bar{\theta}_i)}}{\sqrt{unit_{v^{(t+1)}(\bar{\theta}_i)}}} \propto \frac{\frac{1}{unit_{\bar{\theta}_i}}}{\sqrt{\frac{1}{\left(unit_{\Delta\bar{\theta}_i}\right)^2}}} \propto 1$$

Therefore, Zeiler proposed to apply a correction term proportional to the unit of each weight θ_i. This factor is obtained by considering the exponentially weighted moving average of every squared difference:

$$u^{(t+1)}(\bar{\theta}_i) = \mu u^{(t)}(\bar{\theta}_i) + (1-\mu)\big(\Delta(\bar{\theta}_i)\big)^2$$

The resulting updated rule hence becomes as follows:

$$\bar{\theta}_i^{(t+1)} = \bar{\theta}_i^{(t)} - \frac{\eta\sqrt{u^{(t)}(\bar{\theta}_i)}}{\sqrt{\bar{v}^{(t+1)}(\bar{\theta}_i) + \delta}}\nabla_{\bar{\theta}} L(\bar{\theta}_i)$$

This approach is indeed more similar to RMSProp than AdaGrad, but the boundaries between the two algorithms are very thin, in particular when the history is limited to a finite sliding window. AdaDelta is a powerful algorithm, but it can outperform Adam or RMSProp only in very particular tasks. My suggestion is to employ a method and, before moving to another one, try to optimize the hyperparameters until the accuracy reaches its maximum. If the performances keep on being bad and the model cannot be improved in any other way, it's a good idea to test other optimization algorithms.

AdaDelta with Keras

The following snippet shows the usage of AdaDelta with Keras:

```
from keras.optimizers import Adadelta

...

adadelta = Adadelta(lr=0.0001, rho=0.9, epsilon=1e-6, decay=1e-2)

model.compile(optimizer=adadelta,
              loss='categorical_crossentropy',
              metrics=['accuracy'])
```

The forgetting factor, μ, is represented by the parameter `rho`.

Regularization and dropout

Overfitting is a common issue in deep models. Their extremely high capacity can often become problematic even with very large datasets because the ability to learn the structure of the training set is not always related to the ability to generalize. A deep neural network can easily become an associative memory, but the final internal configuration couldn't be the most suitable to manage samples belonging to the same distribution but was never presented during the training process. It goes without saying that this behavior is proportional to the complexity of the separation hypersurface. A linear classifier has a minimum chance to overfit, and a polynomial classifier is incredibly more prone to do it. A combination of hundreds, thousands, or more non-linear functions yields a separation hypersurface, which is beyond any possible analysis. In 1991, Hornik (in *Approximation Capabilities of Multilayer Feedforward Networks,Hornik K., Neural Networks, 4/2*) generalized a very important result obtained two years before by the mathematician Cybenko (and published in *Approximations by Superpositions of Sigmoidal Functions, Cybenko G., Mathematics of Control, Signals, and Systems, 2 /4*). Without any mathematical detail (which is, however, not very complex), the theorem states that an MLP (not the most complex architecture!) can approximate any function that is continuous in a compact subset of $\Re^n$. It's clear that such a result formalized what almost any researcher already intuitively knew, but its *power* goes beyond the first impact, because the MLP is a finite system (not a mathematical series) and the theorem assumes a finite number of layers and neurons. Obviously, the precision is proportional to the complexity; however, there are no unacceptable limitations for almost any problem. However, our goal is not learning an existing continuous function, but managing samples drawn from an unknown data generating process with the purpose to maximize the accuracy when a new sample is presented. There are no guarantees that the function is continuous or that the domain is a compact subset.

In `Chapter 1`, *Machine Learning Models Fundamentals*, we have presented the main regularization techniques based on a slightly modified cost function:

$$L_r(X, Y; \bar{\theta}) = \frac{1}{M} \sum_{i=1}^{M} L(\bar{x}_i, \bar{y}_i; \bar{\theta}) + g(\bar{\theta})$$

The additional term $g(\theta)$ is a non-negative function of the weights (such as L2 norm) that forces the optimization process to keep the parameters as small as possible. When working with saturating functions (such as `tanh`), regularization methods based on the L2 norm try to limit the operating range of the function to the linear part, reducing *de facto* its capacity. Of course, the final configuration won't be the optimal one (that could be the result of an overfitted model) but the suboptimal trade-off between training and validation accuracy (alternatively, we can say between bias and variance). A system with a bias close to 0 (and a training accuracy close to 1.0) could be extremely rigid in the classification, succeeding only when the samples are very similar to ones evaluated during the training process. That's why this *price* is often paid considering the advantages obtained when working with new samples. L2 regularization can be employed with any kind of activation function, but the effect could be different. For example, ReLU units have an increased probability to become linear (or constantly null) when the weights are very large. Trying to keep them close to 0.0 means forcing the function to exploit its non-linearity without the risk of extremely large outputs (that can negatively affect very deep architectures). This result can sometimes be more useful, because it allows training bigger models in a smoother way, obtaining better final performances. In general, it's almost impossible to decide whether a regularization can improve the result without several tests, but there are some scenarios where it's very common to introduce a dropout (we discuss this approach in the next paragraph) and tune up its hyperparameter. This is more an empirical choice than a precise architectural decision because many real-life examples (including state-of-the-art models) obtained outstanding results employing this regularization technique. I suggest the reader prefer a rational skepticism to blind trust and double-checking its models before picking a specific solution. Sometimes, an extremely high-performing network turns to being ineffective when a different (but analogous) dataset is chosen. That's why testing different alternatives can provide the best experience in order to solve specific problem classes.

Before moving on, I want to show how it's possible to implement an L1 (useful to enforce sparsity), L2, or ElasticNet (the combination of L1 and L2) regularization using Keras. The framework provides a fine-grained approach that allows imposing a different constraint to each layer. For example, the following snippet shows how to add a `l2` constraint with the strength parameter set to `0.05` to a generic fully connected layer:

```
from keras.layers import Dense
from keras.regularizers import l2

...

model.add(Dense(128, kernel_regularizer=l2(0.05)))
```

The `keras.regularizers` package contains the functions `l1()`, `l2()`, and `l1_l2()`, which can be applied to `Dense` and convolutional layers (we're going to discuss them in the next chapter). These layers allow us to impose a regularization on the weights (`kernel_regularizer`), on the bias (`bias_regularizer`), and on the activation output (`activation_regularizer`), even if the first one is normally the most widely employed.

Alternatively, it's possible to impose specific constraints on the weights and biases that in a more selective way. The following snippet shows how to set a maximum norm (equal to `1.5`) on the weights of a layer:

```
from keras.layers import Dense
from keras.constraints import maxnorm

...

model.add(Dense(128, kernel_constraint=maxnorm(1.5)))
```

Keras, in the `keras.constraints` package, provides some functions that can be used to impose a maximum norm on the weights or biases `maxnorm()`, a unit norm along an axis `unit_norm()`, non-negativity `non_neg()`, and upper and lower bounds for the norm `min_max_norm()`. The difference between this approach and regularization is that it is applied only if necessary. Considering the previous example, imposing an L2 regularization always has an effect, while a constraint on the maximum norm is inactive until the value is lower than the predefined threshold.

Dropout

This method has been proposed by Hinton and co. (in *Improving neural networks by preventing co-adaptation of feature detectors, Hinton G. E., Srivastava N., Krizhevsky A., Sutskever I., Salakhutdinov R. R., arXiv:1207.0580 [cs.NE]*) as an alternative to prevent overfitting and allow bigger networks to explore more regions of the sample space. The idea is rather simple—during every training step, given a predefined percentage n_d, a **dropout** layer randomly selects n_dN incoming units and sets them to `0.0` (the operation is only active during the training phase, while it's completely removed when the model is employed for new predictions).

This operation can be interpreted in many ways. When more dropout layers are employed, the result of their selection is a sub-network with a reduced capacity that can, with more difficultly, overfit the training set. The overlap of many trained sub-networks makes up an implicit ensemble whose prediction is an average over all models. If the dropout is applied on input layers, it works like a weak data augmentation, by adding a random noise to the samples (setting a few units to zero can lead to potential corrupted patterns). At the same time, employing several dropout layers allows exploring several potential configurations that are continuously combined and refined.

This strategy is clearly probabilistic, and the result can be affected by many factors that are impossible to anticipate; however, several tests confirmed that the employment of a dropout is a good choice when the networks are very deep because the resulting sub-networks have a residual capacity that allows them to model a wide portion of the samples, without driving the whole network to *freeze* its configuration overfitting the training set. On the other hand, this method is not very effective when the networks are shallow or contain a small number of neurons (in these cases, L2 regularization is probably a better choice).

According to the authors, dropout layers should be used in conjunction with high learning rates and maximum norm constraints on the weights. In this way, in fact, the model can easily learn more potential configurations that would be avoided when the learning rate is kept very small. However, this is not an absolute rule because many state-of-the-art models use a dropout together with optimization algorithms, such as RMSProp or Adam, and not excessively high learning rates.

The main drawback of a dropout is that it slows down the training process and can lead to an unacceptable sub-optimality. The latter problem can be mitigated by adjusting the percentages of dropped units, but, in general, it's very difficult to solve it completely. For this reason, some new image-recognition models (like residual networks) avoid the dropout and employ more sophisticated techniques to train very deep convolutional networks that overfit both training and validation sets.

Example of dropout with Keras

We cannot test the effectiveness of the dropout with a more challenging classification problem. The dataset is the *classical* MNIST handwritten digits, but Keras allows downloading and working with the original version that is made up of 70 thousand (60 thousand training and 10 thousand test) 28×28 grayscale images. Even if this is not the best strategy, because a convolutional network should be the first choice to manage images, we want to try to classify the digits considering them as flattened 784-dimensional arrays.

The first step is loading and normalizing the dataset so that each value becomes a float bounded between 0 and 1:

```
import numpy as np

from keras.datasets import mnist
from keras.utils import to_categorical

(X_train, Y_train), (X_test, Y_test) = mnist.load_data()

width = height = X_train.shape[1]

X_train = X_train.reshape((X_train.shape[0], width *
height)).astype(np.float32) / 255.0
X_test = X_test.reshape((X_test.shape[0], width *
height)).astype(np.float32) / 255.0

Y_train = to_categorical(Y_train, num_classes=10)
Y_test = to_categorical(Y_test, num_classes=10)
```

At this point, we can start testing a model without dropout. The structure, which is common to all experiments, is based on three fully connected ReLU layers (2048-1024-1024) followed by a softmax layer with 10 units. Considering the problem, we can try to train the model using an Adam optimizer with $\eta = 0.0001$ and a decay set to 10^{-6}:

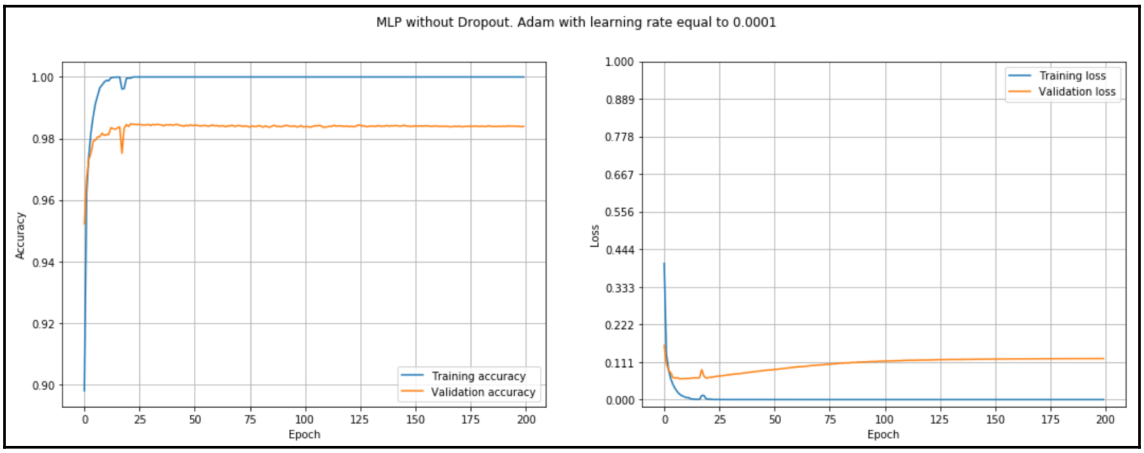
```
from keras.models import Sequential
from keras.layers import Dense, Activation
from keras.optimizers import Adam

model = Sequential()

model.add(Dense(2048, input_shape=(width * height, )))
model.add(Activation('relu'))

model.add(Dense(1024))
model.add(Activation('relu'))

model.add(Dense(1024))
model.add(Activation('relu'))

model.add(Dense(10))
model.add(Activation('softmax'))

model.compile(optimizer=Adam(lr=0.0001, decay=1e-6),
              loss='categorical_crossentropy',
              metrics=['accuracy'])
```

The model is trained for `200` epochs with a batch size of `256` samples:

```
history = model.fit(X_train, Y_train,
                    epochs=200,
                    batch_size=256,
                    validation_data=(X_test, Y_test))

Train on 60000 samples, validate on 10000 samples
Epoch 1/200
60000/60000 [==============================] - 11s 189us/step - loss:
0.4026 - acc: 0.8980 - val_loss: 0.1601 - val_acc: 0.9523
Epoch 2/200
60000/60000 [==============================] - 7s 116us/step - loss: 0.1338
- acc: 0.9621 - val_loss: 0.1062 - val_acc: 0.9669
Epoch 3/200
60000/60000 [==============================] - 7s 124us/step - loss: 0.0872
- acc: 0.9744 - val_loss: 0.0869 - val_acc: 0.9732

...

Epoch 199/200
60000/60000 [==============================] - 7s 114us/step - loss:
1.1935e-07 - acc: 1.0000 - val_loss: 0.1214 - val_acc: 0.9838
Epoch 200/200
60000/60000 [==============================] - 7s 116us/step - loss:
1.1935e-07 - acc: 1.0000 - val_loss: 0.1214 - val_acc: 0.9840
```

Even without a further analysis, we can immediately notice that the model is overfitted. After 200 epochs, the training accuracy is 1.0 with a loss close to 0.0, while the validation accuracy is reasonably high, but with a validation loss slightly lower than the one obtained at the end of the second epoch.

To better understand what happened, it's useful to plot both accuracy and loss during the training process:



As it's possible to see, the validation loss reached a minimum during the first 10 epochs and immediately restarted to grow (this is sometimes called a U-curve because of its shape). At the same moment, the training accuracy reached 1.0. From that epoch on, the model started overfitting, learning a perfect structure of the training set, but losing the generalization ability. In fact, even if the final validation accuracy is rather high, the loss function indicates a lack of robustness when new samples are presented. As the loss is a categorical cross-entropy, the result can be interpreted as saying that the model has learned a distribution that partially mismatches the validation set one. As our goal is to use the model to predict new samples, this configuration could not be acceptable. Therefore, we try again, using some dropout layers. As suggested by the authors, we also increment the learning rate to 0.1 (switching to a Momentum SGD optimizer in order to avoid *explosions* due to adaptivity of RMSProp or Adam), initialize the weight with a uniform distribution (*-0.05, 0.05*), and impose a maximum norm constraint set to 2.0. This choice allows the exploration of more sub-configurations without the risk of excessively high weights. The dropout is applied to the 25% of input units and to all ReLU fully connected layers with a percentage set to 50%:

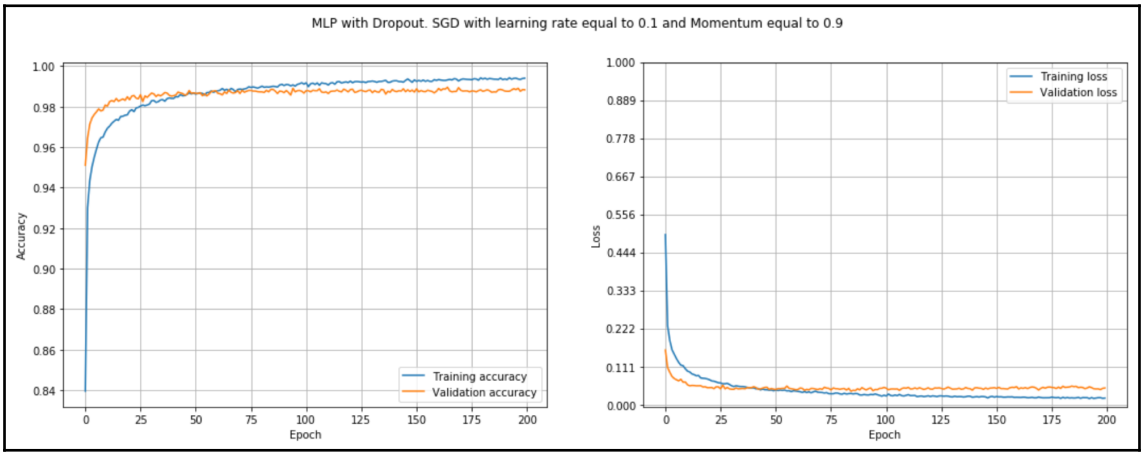
```
from keras.constraints import maxnorm
from keras.models import Sequential
from keras.layers import Dense, Activation, Dropout
from keras.optimizers import SGD

model = Sequential()

model.add(Dropout(0.25, input_shape=(width * height, ), seed=1000))
```

```
model.add(Dense(2048, kernel_initializer='uniform',
kernel_constraint=maxnorm(2.0)))
model.add(Activation('relu'))
model.add(Dropout(0.5, seed=1000))

model.add(Dense(1024, kernel_initializer='uniform',
kernel_constraint=maxnorm(2.0)))
model.add(Activation('relu'))
model.add(Dropout(0.5, seed=1000))

model.add(Dense(1024, kernel_initializer='uniform',
kernel_constraint=maxnorm(2.0)))
model.add(Activation('relu'))
model.add(Dropout(0.5, seed=1000))

model.add(Dense(10))
model.add(Activation('softmax'))

model.compile(optimizer=SGD(lr=0.1, momentum=0.9),
              loss='categorical_crossentropy',
              metrics=['accuracy'])
```

The training process is performed with the same parameters:

```
history = model.fit(X_train, Y_train,
                    epochs=200,
                    batch_size=256,
                    validation_data=(X_test, Y_test))

Train on 60000 samples, validate on 10000 samples
Epoch 1/200
60000/60000 [==============================] - 11s 189us/step - loss:
0.4964 - acc: 0.8396 - val_loss: 0.1592 - val_acc: 0.9511
Epoch 2/200
60000/60000 [==============================] - 6s 97us/step - loss: 0.2300
- acc: 0.9300 - val_loss: 0.1081 - val_acc: 0.9645
Epoch 3/200
60000/60000 [==============================] - 6s 93us/step - loss: 0.1867
- acc: 0.9435 - val_loss: 0.0941 - val_acc: 0.9713

...

Epoch 199/200
60000/60000 [==============================] - 6s 99us/step - loss: 0.0184
- acc: 0.9939 - val_loss: 0.0473 - val_acc: 0.9884
Epoch 200/200
60000/60000 [==============================] - 6s 101us/step - loss: 0.0190
- acc: 0.9941 - val_loss: 0.0484 - val_acc: 0.9883
```

The final condition is dramatically changed. The model is no longer overfitted (even if it's possible to improve it in order to increase the validation accuracy) and the validation loss is lower than the initial one. To have a confirmation, let's analyze the accuracy/loss plots:



The result shows some imperfections because the validation loss is almost flat for many epochs; however, the same model, with a higher learning rate and a weaker algorithm achieved a better final performance (0.988 validation accuracy) and a superior generalization ability. State-of-the-art models can also reach a validation accuracy equal to 0.995, but our goal was to show the effect of dropout layers in preventing the overfitting and, moreover, yielding a final configuration that is much more robust to new samples or noisy ones. I invite the reader to repeat the experiment with different parameters, bigger or smaller networks, and other optimization algorithms, trying to further reduce the final validation loss.

Keras also implements two additional dropout layers: `GaussianDropout`, which multiplies the input samples by a Gaussian noise:

$$\hat{x}_i = \bar{x}_i \, \bar{n} \;\; where \;\; \bar{n} \sim N\left(1.0, \frac{\rho}{1-\rho}\right)$$

The value for the constant ρ can be set through the parameter `rate` (bounded between 0 and 1). When $\varrho \to 1$, $\sigma^2 \to \infty$, while small values yield a null effect as $n \approx 1$. This layer can be very useful as input one, in order to simulate a random data augmentation process. The other class is `AlphaDropout`, which works like the previous one, but renormalizing the output to keep the original mean and variance (this effect is very similar to the one obtained employing the technique described in the next paragraph together with noisy layers).

When working with probabilistic layers (such as dropout), I always suggest setting the random seed (`np.random.seed(...)` and `tf.set_random_seed(...)` when Tensorflow backend is used). In this way, it's possible to repeat the experiments comparing the results without any bias. If the random seed is not explicitly set, every new training process will be different and it's not easy to compare the performances, for example, after a fixed number of epochs.

Batch normalization

Let's consider a mini-batch of *k* samples:

$$X_b = \{\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_k\}$$

Before traversing the network, we can measure a mean and a variance:

$$\bar{X} = \frac{1}{k}\sum_{i=1}^{k}\bar{x}_i \quad and \quad Var(X) = \frac{1}{k}\sum_{i=1}^{k}(\bar{x}_i - \bar{X})^2$$

After the first layer (for simplicity, let's suppose that the activation function, *f(•)*, is the always the same), the batch is transformed into the following:

$$X_b^{(1)} = \{f(\bar{w}_1{}^T\bar{x}_1 + \bar{b}_1), f(\bar{w}_2{}^T\bar{x}_2 + \bar{b}_2), \ldots, f(\bar{w}_k{}^T\bar{x}_k + \bar{b}_k)\}$$

In general, there's no guarantee that the new mean and variance are the same. On the contrary, it's easy to observe a modification that increases throughout the network. This phenomenon is called **covariate shift**, and it's responsible for a progressive training speed decay due to the different adaptations needed in each layer. Ioffe and Szegedy (in *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift, Ioffe S., Szegedy C., arXiv:1502.03167 [cs.LG]*) proposed a method to mitigate this problem, which has been called **batch normalization** (**BN**).

The idea is to renormalize the linear output of a layer (before or after applying the activation function), so that the batch has null mean and unit variance. Therefore, the first task of a BN layer is to compute:

$$\bar{X}^{(j)} = \frac{1}{k}\sum_{i=1}^{k}\bar{x}_i^{(j)} \quad and \quad Var(X^{(j)}) = \frac{1}{k}\sum_{i=1}^{k}\left(\bar{x}_i^{(j)} - \bar{X}^{(j)}\right)^2$$

Then each sample is transformed into a normalized version (the parameter δ is included to improve the numerical stability):

$$\hat{x}_i^{(j)} = \frac{\bar{x}_i^{(j)} - \bar{X}^{(j)}}{\sqrt{Var(X^{(j)}) + \delta}}$$

However, as the batch normalization has no computational purposes other than speeding up the training process, the transformation must always be an identity (in order to avoid to distort and bias the data); therefore, the actual output will be obtained by applying the linear operation:

$$\bar{y}_i^{(j)} = \alpha^{(j)}\hat{x}_i^{(j)} + \beta^{(j)}$$

The two parameters $\alpha^{(j)}$ and $\beta^{(j)}$ are variables optimized by the SGD algorithm; therefore, each transformation is guaranteed not to alter the scale and the position of data. These layers are active only during the training phase (like dropout), but, contrary to other algorithms, they cannot be simply discarded when the model is used to make predictions on new samples because the output would be constantly biased. To avoid this problem, the authors suggest approximating both mean and variance of *X* by averaging over the batches (assuming that there are N_b batches with *k* samples):

$$\mu = \frac{1}{N_b}\sum_{i=1}^{N_b}\bar{X}^{(i)} \quad and \quad \sigma^2 = \frac{k}{N_b(k-1)}\sum_{i=1}^{N_b}Var(X^{(i)})$$

Using these values, the batch normalization layers can be transformed into the following linear operations:

$$y_i^{(j)} = \frac{\alpha^{(j)}}{\sqrt{\sigma^2 + \delta}}\bar{x}_i + \left(\beta^{(j)} - \frac{\alpha^{(j)}\mu}{\sqrt{\sigma^2 + \delta}}\right)$$

It's not difficult to prove that this approximation becomes more and more accurate when the number of batches increases and that the error is normally negligible. However, when the batch size is very small, the statistics can be quite inaccurate; therefore, this method should be used considering the *representativeness* of a batch. If the data generating process is simple, even a small batch can be enough to describe the actual distribution. When, instead, p_{data} is more complex, batch normalization requires larger batches to avoid wrong adjustments (a feasible strategy is to compare global mean and variance with the ones computed sampling some batches and trying to set the batch size that minimizes the discrepancy). However, this simple process can dramatically reduce the covariate shift and improve the convergence speed of very deep networks (including the famous residual networks). Moreover, it allows employing higher learning rates as the layers are implicitly *saturated* and can never *explode*. Additionally, it has been proven that batch normalization has also a secondary regularization effect even if it doesn't work on the weights. The reason is not very different from the one proposed for L2, but, in this case, there's a residual effect due to the transformation itself (partially caused by the variability of the parameters $\alpha^{(j)}$ and $\beta^{(j)}$) that can encourage the exploration of different regions of the sample space. However, this is not the primary effect, and it's not a good practice employing this method as a regularizer.

Example of batch normalization with Keras

In order to show the feature of this technique, let's repeat the previous example using an MLP without dropout but applying a batch normalization after each fully connected layer before the ReLU activation. The example is very similar to the first one, but, in this case, we increase the Adam learning rate to 0.001 keeping the same decay:

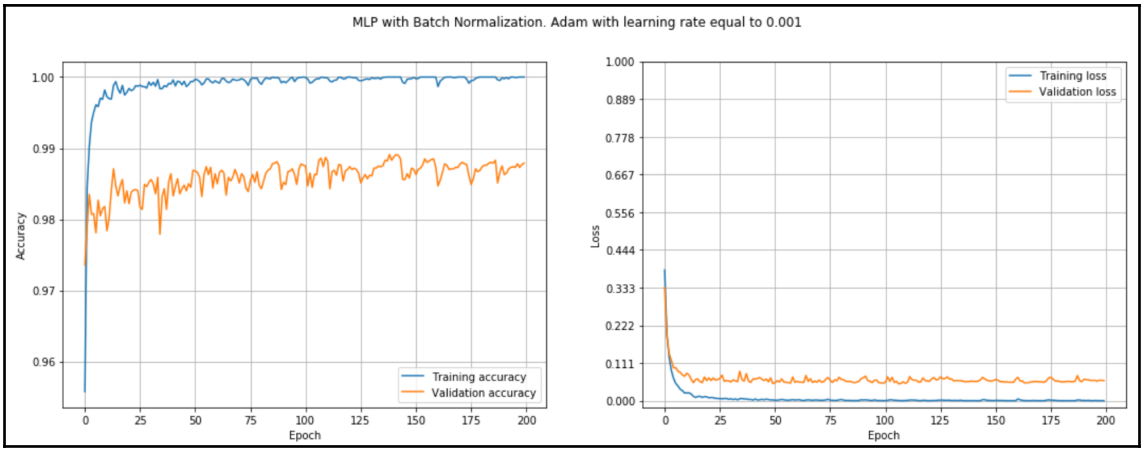
```
from keras.models import Sequential
from keras.layers import Dense, Activation, BatchNormalization
from keras.optimizers import Adam

model = Sequential()

model.add(Dense(2048, input_shape=(width * height, )))
model.add(BatchNormalization())
model.add(Activation('relu'))

model.add(Dense(1024))
model.add(BatchNormalization())
model.add(Activation('relu'))

model.add(Dense(1024))
model.add(BatchNormalization())
model.add(Activation('relu'))

model.add(Dense(10))
model.add(BatchNormalization())
model.add(Activation('softmax'))

model.compile(optimizer=Adam(lr=0.001, decay=1e-6),
              loss='categorical_crossentropy',
              metrics=['accuracy'])
```

We can now train using the same parameters again:

```
history = model.fit(X_train, Y_train,
                    epochs=200,
                    batch_size=256,
                    validation_data=(X_test, Y_test))

Train on 60000 samples, validate on 10000 samples
Epoch 1/200
60000/60000 [==============================] - 16s 274us/step - loss:
0.3848 - acc: 0.9558 - val_loss: 0.3338 - val_acc: 0.9736
Epoch 2/200
60000/60000 [==============================] - 8s 139us/step - loss: 0.1977
- acc: 0.9844 - val_loss: 0.1904 - val_acc: 0.9789
```

```
Epoch 3/200
60000/60000 [==============================] - 8s 137us/step - loss: 0.1292
- acc: 0.9903 - val_loss: 0.1397 - val_acc: 0.9835

...

Epoch 199/200
60000/60000 [==============================] - 8s 132us/step - loss:
4.7805e-05 - acc: 1.0000 - val_loss: 0.0599 - val_acc: 0.9877
Epoch 200/200
60000/60000 [==============================] - 8s 133us/step - loss:
2.6056e-05 - acc: 1.0000 - val_loss: 0.0593 - val_acc: 0.9879
```

The model is again overfitted, but now the final validation accuracy is only slightly higher than the one achieved using the dropout layers. Let's plot accuracy and loss to better analyze the training process:



The effect of the batch normalization improved the performances and slowed down the overfitting. At the same time, the elimination of the covariate shift avoided the U-curve keeping a quite low validation loss. Moreover, the model reached a validation accuracy of about 0.99 during the epochs 135-140 with a residual positive trend. Analogously to the previous example, this solution is imperfect, but it's a good starting point for further optimization. It would be a good idea to continue the training process for a larger number of epochs, monitoring both the validation loss and accuracy. Moreover, it's possible to mix dropout and batch normalization or experiment with the Keras AlphaDropout layer. However, if, in the first example (without dropout), the climax of training accuracy was associated with a starting positive trend for the validation loss, in this case, the learned distribution doesn't seem to be very different from the validation set one. In other words, batch normalization is not preventing overfitting the training set, but it's avoiding a decay in the generalization ability (observed when there was no batch normalization). I suggest repeating the test with other hyperparameter and architectural configurations in order to decide whether this model can be used for prediction purposes or it's better to look for other solutions.

Summary

In this chapter, we started the exploration of the deep learning world by introducing the basic concepts that led the first researchers to improve the algorithms until they achieved the top results we have nowadays. The first part explained the structure of a basic artificial neuron, which combines a linear operation followed by an optional non-linear scalar function. A single layer of linear neurons was initially proposed as the first neural network, with the name of the perceptron.

Even though it was quite powerful for many problems, this model soon showed its limitations when working with non-linear separable datasets. A perceptron is not very different from a logistic regression, and there's no concrete reason to employ it. Nevertheless, this model opened the doors to a family of extremely powerful models obtained combining multiple non-linear layers. The multilayer perceptron, which has been proven to be a universal approximator, is able to manage almost any kind of dataset, achieving high-level performances when other methods fail.

In the next section, we analyzed the building bricks of an MLP. We started with the activation functions, describing their structure and features, and focusing on the reasons they lead the choice for specific problems. Then, we discussed the training process, considering the basic idea behind the back-propagation algorithm and how it can be implemented using the stochastic gradient descent method. Even if this approach is quite effective, it can be slow when the complexity of the network is very high. For this reason, many optimization algorithms were proposed. In this chapter, we analyzed the role of momentum and how it's possible to manage adaptive corrections using RMSProp. Then, we combined both, momentum and RMSProp to derive a very powerful algorithm called Adam. In order to provide a complete vision, we also presented two slightly different adaptive algorithms, called AdaGrad and AdaDelta.

In the next sections, we discussed the regularization methods and how they can be plugged into a Keras model. An important section was dedicated to a very diffused technique called dropout, which consists in setting to zero (dropping) a fixed percentage of samples through a random selection. This method, although very simple, prevents the overfitting of very deep networks and encourages the exploration of different regions of the sample space, obtaining a result not very dissimilar to the ones analyzed in `Chapter 8`, *Ensemble Learning*. The last topic was the batch normalization technique, which is a method to reduce the mean and variance shift (called covariate shift) caused by subsequent neural transformations. This phenomenon can slow down the training process as each layer requires different adaptations and it's more difficult to move all the weights in the best direction. Applying batch normalization means very deep networks can be trained in a shorter time, thanks also to the possibility of employing higher learning rates.

In the next chapter, we are going to continue this exploration, analyzing very important advanced layers like convolutions (that achieve extraordinary performances in image-oriented tasks) and recurrent units (for the processing of time series) and discussing some practical applications that can be experimented on and readapted using Keras and Tensorflow.

10
Advanced Neural Models

In this chapter, we continue our pragmatic exploration of the world of deep learning, analyzing two very important elements: deep convolutional networks and **recurrent neural networks** (**RNN**). The former represents the most accurate and best performing visual processing technique for almost any purpose. Results like the ones obtained in fields such as real-time image recognition, self-driving cars, and Deep Reinforcement Learning have been possible thanks to the expressivity of this kind of network. On the other hand, in order to fully manage the temporal dimension, it is necessary to introduce advanced recurrent layers, whose performance must be greater than any other regression method. Employing these two techniques together with all the elements already discussed in the previous chapter makes it possible to achieve extraordinary results in the field of video processing, decoding, segmentation, and generation.

In particular, in this chapter, we are going to discuss the following topics:

- Deep convolutional networks
- Convolutions, atrous convolutions, separable convolutions, and transpose convolutions
- Pooling and other support layers
- Recurrent neural networks
- LSTM and GRU cells
- Transfer learning

Deep convolutional networks

In the previous chapter, `Chapter 9`, *Neural Networks for Machine Learning* we have seen how a multi-layer perceptron can achieve a very high accuracy when working with an complex image dataset that is not very complex, such as the MNIST handwritten digits one. However, as the fully-connected layers are *horizontal*, the images, which in general are three-dimensional structures (*width* × *height* × *channels*), must be flattened and transformed into one-dimensional arrays where the geometric properties are definitively lost. With more complex datasets, where the distinction between classes depends on more details and on their relationships, this approach can yield moderate accuracies, but it can never reach the precision required by production-ready applications.

The conjunction of neuroscientific studies and image processing techniques suggested experimenting with neural networks where the first layers work with bidimensional structures (without the channels), trying to extract a hierarchy of features that are strictly dependent on the geometric properties of the image. In fact, as confirmed by neuroscientific research about the visual cortex, a human being doesn't decode an image directly. The process is sequential and starts by detecting low-level elements such as lines are orientations; progressively, it proceeds by focusing on sub-properties that define more and more complex shapes, different colors, structural features, and so on, until the amount of information is enough to resolve any possible ambiguity (for further scientific details, I recommend the book *Vision and Brain: How We Perceive the World, Stone J. V., MIT Press*).

For example, we can image the decoding process of an eye as a sequence made up of these filters (of course, this is only a didactic example): directions (dominant horizontal dimension), a central circle inside an ellipsoidal shape, a darker center (pupil) and a clear background (bulb), a smaller darker circle in the middle of the pupil, the presence of eyebrows, and so on. Even if the process is not biologically correct, it can be considered as a reasonable hierarchical process where a higher level sub-feature is obtained after a lower-level filtering.

This approach has been synthesized using the bidimensional convolutional operator, which was already known as a powerful image processing tool. However, in this case, there's a very important difference: the structure of the filters is not pre-imposed but learned by the network using the same back-propagation algorithm employed for MLPs. In this way, the model can adapt the weights considering a final goal (which is the classification output), without taking into account any pre-processing steps. In fact, a deep convolutional network, more than an MLP, is based on the concept of end-to-end learning, which is a different way to express what we have described before. The input is the source; in the middle, there's a flexible structure; and, at the end, we define a global cost function, measuring the accuracy of the classification. The learning process has to back-propagate the errors and correct the weights to reach a specific goal, but we don't know exactly how this process works. What we can easily do is analyze the structure of the filters at the end of the learning phase, discovering that the network has specialized the first layers on low-level details (such as orientations) and the last ones on high-level, sometimes recognizable, ones (such as the components of a face). It's not surprising that such models achieved state-of-the-art performance in tasks such as image recognition, segmentation (detecting the boundaries of different parts composing an image), and tracking (detecting the position of moving objects). Nevertheless, deep convolutional networks have become the first block of many different architectures (such as deep reinforcement learning or neural style transfer) and, even with a few known limitations, continue to be the first choice for solving several complex real-life problems. The main drawback of such models (which is also a common objection) is that they require very large datasets to reach high accuracies. All the most important models are trained with millions of images and their generalization ability (that is, the main goal) is proportional to the number of different samples. There were researchers who noticed that a human being learns to generalize without this huge amount of experience and, in the coming decades, we are likely to observe improvements under this viewpoint. However, deep convolutional networks have revolutionized many Artificial Intelligence fields, allowing results that were considered almost impossible just a few years ago.

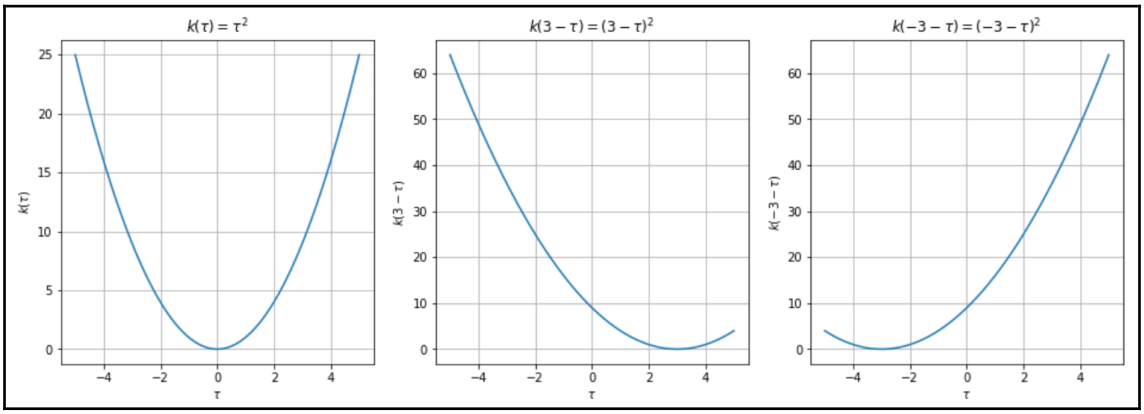
In this section, we are going to discuss different kinds of convolutions and how they can be implemented using Keras; therefore, for specific technical details I continue suggesting to check the official documentation and the book *Deep Learning with Keras, Gulli A, Pal S., Packt*.

Convolutions

Even if we work only with finite and discrete convolutions, it's useful to start providing the standard definition based on integrable functions. For simplicity, let's suppose that $f(\tau)$ and $k(\tau)$ are two real functions of a single variable defined in $\mathfrak{R}$. The convolution of $f(\tau)$ and $k(\tau)$ (conventionally denoted as $f * k$), which we are going to call kernel, is defined as follows:

$$f * k = \int_{-\infty}^{\infty} f(\tau)k(t-\tau)d\tau$$

The expression may not be very easy to understand without a mathematical background, but it can become exceptionally simple with a few considerations. First of all, the integral sums over all values of τ; therefore, the convolution is a function of the remaining variable, t. The second fundamental element is a sort of dynamic property: the kernel is reversed ($-\tau$) and transformed into a function of a new variable $z = t - \tau$. Without deep mathematical knowledge, it's possible to understand that this operation shifts the function along the τ (independent variable) axis. In the following graphs, there's an example based on a parabola:



The first diagram is the original kernel (which is also symmetric). The other two plots show, respectively, a forward and a backward shift. It should be clearer now that a convolution multiplies the function $f(\tau)$ times the shifted kernel and computes the area under the resulting curve. As the variable t is not integrated, the area is a function of t and defines a new function, which is the convolution itself. In other words, the value of convolution of $f(\tau)$ and $k(\tau)$ computed for $t = 5$ is the area under the curve obtained by the multiplication $f(\tau)k(5 - \tau)$. By definition, a convolution is commutative $(f * k = k * f)$ and distributive $(f * (k + g) = (f * k) + (f * g))$. Moreover, it's also possible to prove that it's associative $(f * (k * g) = (f * k) * g)$.

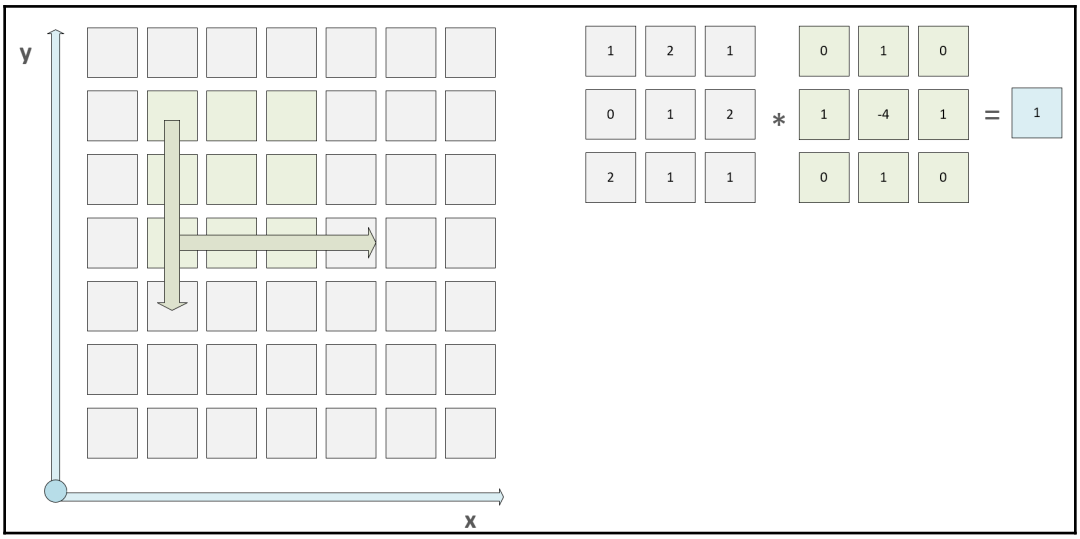
However, in deep learning, we never work with continuous convolutions; therefore, I omit all the properties and mathematical details, focusing the attention on the discrete case. The reader who is interested in the theory can find further details in *Circuits, Signals, and Systems, Siebert W. M., MIT Press*. A common practice is, instead, to stack multiple convolutions with different kernels (often called filters), to transform an input containing n channels into an output with m channels, where m corresponds to the number of kernels. This approach allows the unleashing of the full power of convolutions, thanks to the synergic actions of different outputs. Conventionally, the output of a convolution layer with n filters is called a **feature map** ($w^{(t)} \times h^{(t)} \times n$), because its structure is no longer related to a specific image but resembles the overlap of different feature detectors. In this chapter, we often talk about images (considering a hypothetical first layer), but all the considerations are implicitly extended to any feature map.

Bidimensional discrete convolutions

The most common type of convolution employed in deep learning is based on bidimensional arrays with any number of channels (such as grayscale or RGB images). For simplicity, let's analyze a single layer (channel) convolution because the extension to n layers is straightforward. If $X \in \mathfrak{R}^{w \times h}$ and $k \in \mathfrak{R}^{n \times m}$, the convolution $X * k$ is defined as (the indexes start from 0):

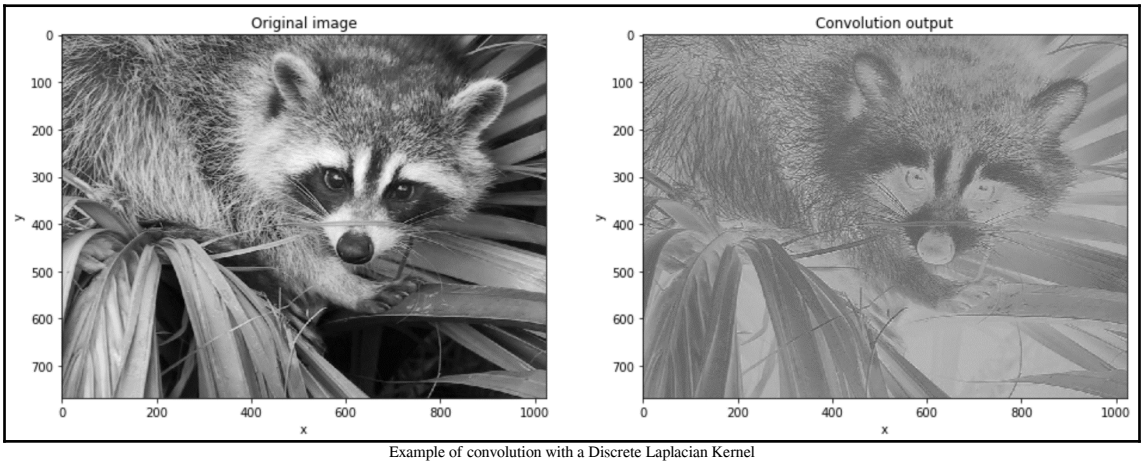
$$(X * k)(x, y) = \sum_{i \in [0, n-1],\ j \in [0, m-1]} k(i, j) X(x + i, y + j)$$

It's clear that the previous expression is a natural derivation of the continuous definition. In the following graph, there's an example with a 3×3 kernel:



Example of bidimensional convolution with a 3x3 kernel

The kernel is shifted horizontally and vertically, yielding the sum of the element-wise multiplication of corresponding elements. Therefore, every operation leads to the output of a single pixel. The kernel employed in the example is called the **discrete Laplacian operator** (because it's obtained by discretizing the real Laplacian); let's observe the effect of this kernel on a complete greyscale diagram:

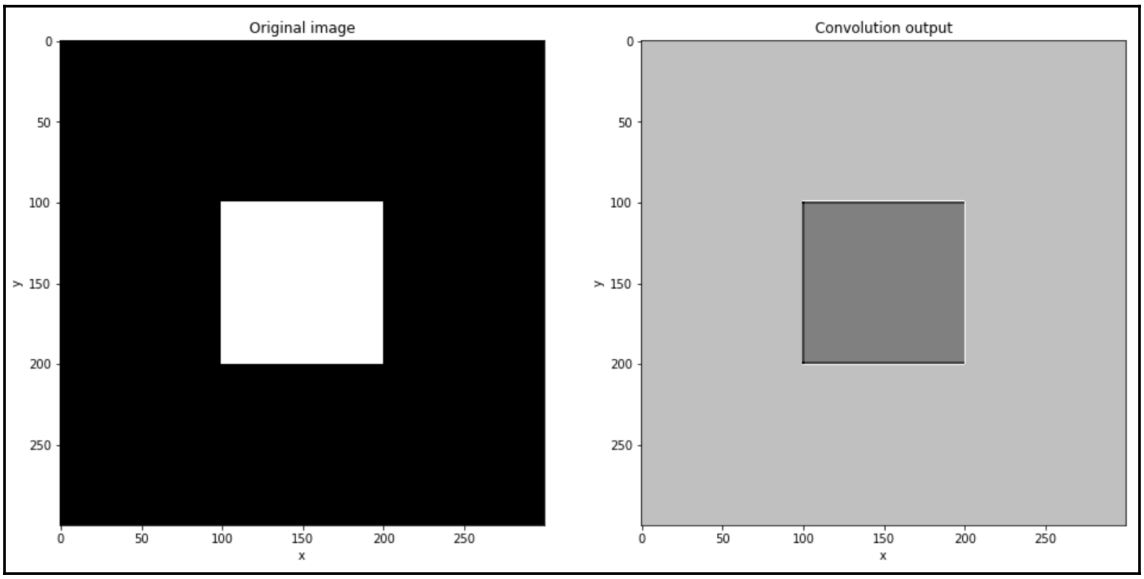


Example of convolution with a Discrete Laplacian Kernel

As it's possible to notice, the effect of the convolution is to emphasize the borders of the various shapes. The reader can now understand how variable kernels can be tuned up in order to fulfill precise requirements. However, instead of trying to do it manually, a deep convolutional network leaves this tasks to the learning process, which is subject to a precise goal expressed as the minimization of a cost function. A parallel application of different filters yields complex overlaps that can simplify the extraction of those features that are really important for a classification. The main difference between a fully-connected layer and a convolutional one is the ability of the latter to work with an existing geometry, which encodes all the elements needed to distinguish an object from another one. These elements cannot be immediately generalizable (think about the branches of a decision tree, where a split defines a precise path towards a final class), but require subsequent processing steps to perform a necessary disambiguation. Considering the previous photo, for example, eyes and nose are rather similar. How is it possible to segment the picture correctly? The answer is provided by a double analysis: there are subtle differences that can be discovered by fine-grained filters and, above all, the global geometry of real objects is based on internal relationships that are almost invariant. For example (only for didactic purposes), eyes and nose should make up an isosceles triangle, because the symmetry of a face implies the same distance between each eye and the nose. This consideration can be made *apriori*, like in many visual processing techniques, or, thanks to the power of deep learning, it can be left to the training process. As the cost function and the output classes implicitly control the differences, a deep convolutional network can learn what is important to reach a specific goal, discarding at the same time all those details that are useless.

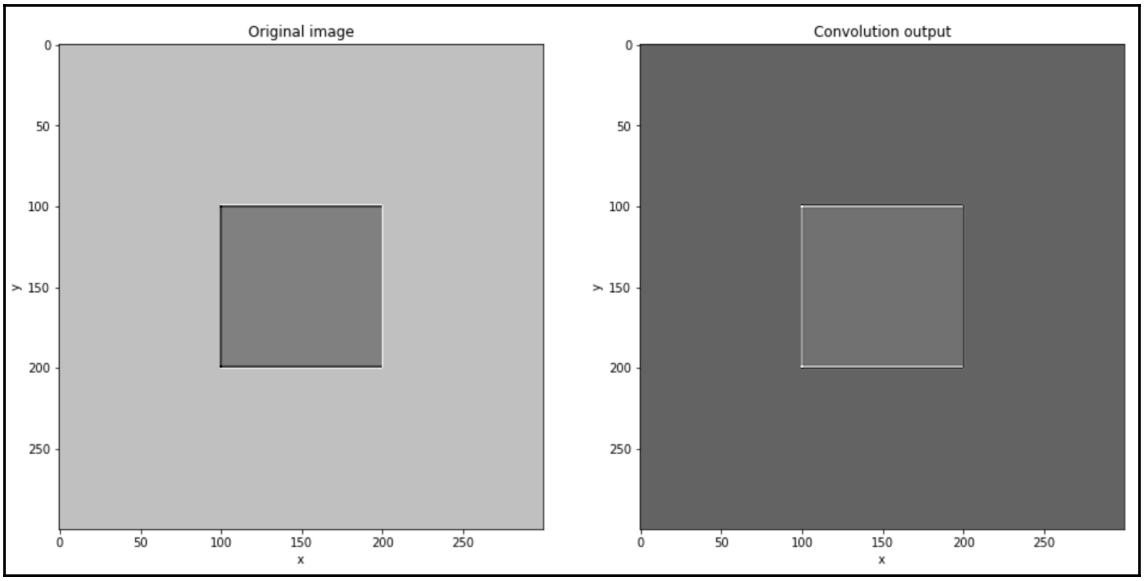
In the previous section, we have said that the feature extraction process is mainly hierarchical. Now, it should be clear that different kernel sizes and subsequent convolutions achieve exactly this objective. Let's suppose that we have a 100×100 image and a (3×3) kernel. The resulting image will be 98×98 pixels (we will explain this concept later). However, each pixel encodes the information of a 3×3 block and, as these blocks are overlapping, two consecutive pixels will share some knowledge but, at the same time, they emphasize the difference between the corresponding blocks.

In the following diagram, the same Laplacian Kernel is applied to a simple white square on a black background:



Orginal image (left); convolution with Laplacian kernel result (right)

Even if the image is very simple, it's possible to notice that the result of a convolution enriched the output image with some very important pieces of information: the borders of the square are now clearly visible (they are black and white) and they can be immediately detected by thresholding the image. The reason is straightforward: the effect of the kernel on the compact surfaces is compact too but, when the kernel is shifted upon the border, the effect of the difference becomes visible. Three adjacent pixels in the original image can be represented as (*0, 1, 1*), indicating the horizontal transition between black and white. After the convolution, the result is approximately (*0.75, 0.0, 0.25*). All the original black pixels have been transformed into a light gray, the white square became darker, and the border (which is not marked in the original picture) is now black (or white, depending on the shift direction). Reapplying the same filter to the output of the previous convolution, we obtain the following:



Second application of the Laplacian kernel

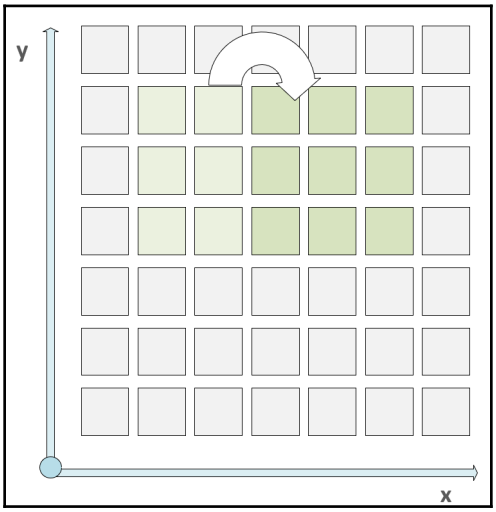
A sharp eye can immediately notice three results: the compact surfaces (black and white) are becoming more and more similar, the borders are still visible, and, above all, the top and lower left corners are now more clearly marked with white pixels. Therefore, the result of the second convolution added a finer-grained piece of information, which was much more difficult to detect in the original image. Indeed, the effect of the Laplacian operator is very straightforward and it's useful only for didactic purposes. In real deep convolutional networks, the filters are trained to perform more complex processing operations that can reveal details (together with their internal and external relationships) that are not immediately exploited to classify the image. Their isolation (obtained thanks to the effect of many parallel filters) allows the network to mark similar elements (like the corners of the square) in a different way and make more accurate decisions.

The purpose of this example is to show how a sequence of convolutions allows the generation of a hierarchical process that will extract coarse-grained features at the beginning and very high-level ones at the end, without losing the information already collected. Metaphorically, we could say that a deep convolutional network starts placing labels indicating lines, orientations, and borders and proceeds by enriching the existing ontology with further details (such as corners, particular shapes, and so on). Thanks to this ability, such models can easily outperform any MLP and reach almost to the Bayes level if the number of training samples is large enough. The main drawback of this models is their inability to easily recognize objects after the application of affine transformations (such as rotations or translations). In other words, if a network is trained with a dataset containing only faces in their natural position, it will achieve poor performance when a rotated (or upside-down) sample is presented. In the next sections, we are going to discuss a couple of methods that are helpful for mitigating this problem (in the case of translations); however, a new experimental architecture called a **capsule network** (which is beyond the scope of this book) has been proposed in order to solve this problem with a slightly different and much more robust approach (the reader can find further details in *Dynamic Routing Between Capsules, Sabour S., Frosst N., Hinton G. E., arXiv:1710.09829 [cs.CV]*).

Strides and padding

Two important parameters common to all convolutions are **padding** and **strides**. Let's consider the bidimensional case, but keep in mind that the concepts are always the same. When a kernel ($n \times m$ with $n, m > 1$) is shifted upon an image and it arrives at the end of a dimension, there are two possibilities. The first one, called **valid padding**, consists of not continuing even if the resulting image is smaller than the original. In particular, if X is a $w \times h$ matrix, the resulting convolution output will have dimensions equal to $(w - n + 1) \times (h - m + 1)$. However, there are many cases when it's useful to keep the original dimensions, for example, to be able to sum different outputs. This approach is called **same padding** and it's based on the simple idea to add $n - 1$ blank columns and $m - 1$ blank rows to allow the kernel to shift over the original image, yielding a number of pixels equal to the initial dimensions. In many implementations, the default value is set to valid padding.

The other parameter, called **strides**, defines the number of pixels to skip during each shift. For example, a value set to (*1, 1*) corresponds to a standard convolution, while strides set to (*2, 1*) are shown in the following diagram:

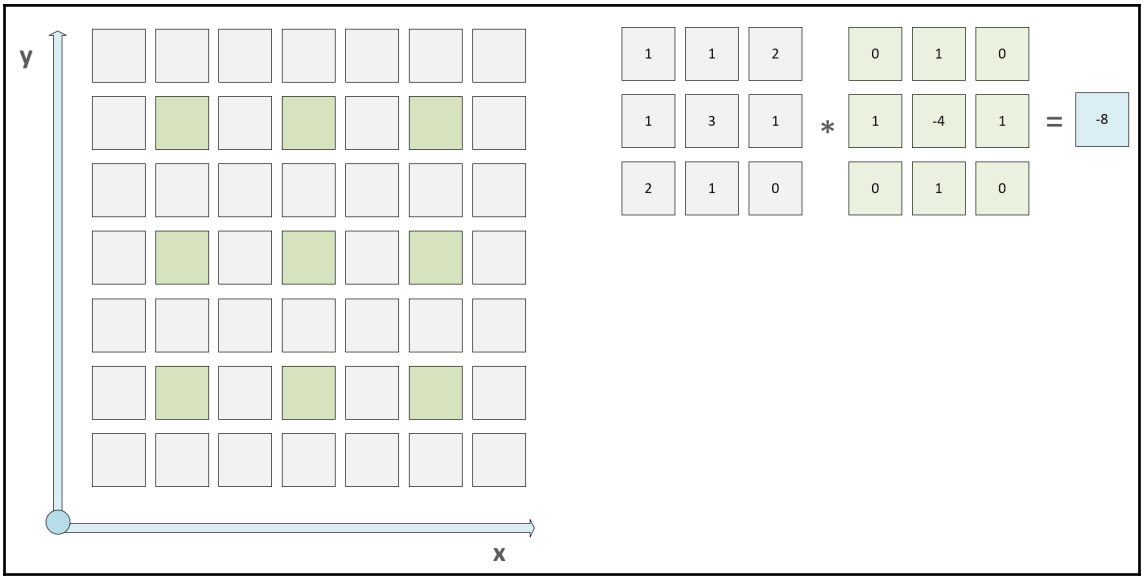


Example of bidimensional convolution with strides=2 on the x-axis

In this case, every horizontal shift skips a pixel. Larger strides force a dimensionality reduction when a high granularity is not necessary (for example, in the first layers), while strides set to (*1, 1*) are normally employed in the last layers to capture smaller details. There are no standard rules to find out the optimal value and testing different configurations is always the best approach. Like any other hyperparameter, too many elements should be taken into account when determining whether a choice is acceptable or not; however, some general pieces of information about the dataset (and therefore about the underlying data generating process) can help in making a reasonable initial decision. For example, if we are working with pictures of buildings whose dimension is vertical, it's possible to start picking a value of (*1, 2*), because we can assume that there's more informative redundancy in the *y*-axis than in the *x*-axis. This choice can dramatically speed up the training process, as the output has one dimension, which is half (with the same padding) of the original one. In this way, larger strides produce a partial denoising and can improve the training speed. At the same time, the information loss could have a negative impact on the accuracy. If that happens, it probably means that the scale isn't high enough to allow skipping some elements without compromising the *semantics*. For example, an image with very small faces could be irreversibly *damaged* with large strides, yielding an inability to detect the right feature and a consequent worsening of the classification accuracy.

Atrous convolution

In some cases, a stride larger than one could be a good solution because it reduces the dimensionality and speeds up the training process, but it can lead to distorted images where the main features are not detectable anymore. An alternative approach is provided by the **atrous convolution** (also known as **dilated convolution**). In this case, the kernel is applied to a larger image patch, but skips some pixels inside the area itself (that's why someone called it convolution with holes). In the following graph, there's an example with (3×3) and dilation rate set to 2:



Example of atrous convolution with a Laplacian kernel

Every patch is now 9×9, but the kernel remains a 3×3 Laplacian operator. The effect of this approach is more robust than increasing the strides because the kernel *perimeter* will always contain a group of pixels with the same geometrical relationships. Of course, fine-grained features could be distorted, but as the strides are normally set to (*1, 1*), the final result is normally more coherent. The main difference with a standard convolution is that in this case, we are assuming that farther elements can be taken into account to determine the nature of an output pixel. For example, if the main features don't contain very small details, an atrous convolution can consider larger areas, focusing directly on elements that a standard convolution can detect only after several operations. The choice of this technique must be made considering the final accuracy, but just like for the strides, it can be considered from the beginning whenever the geometric properties can be detected more efficiently, considering larger patches with a few representative elements. Even if this method can be very effective in particular contexts, it isn't normally the first choice for very deep models. In the most important image classification models, standard convolutions (with or without larger strides) are employed because they have been proven to yield the best performance with very generic datasets (such as ImageNet or Microsoft Coco). However, I suggest the reader experiment with this method and compare the results. In particular, it would be a good idea to analyze which classes are better classified and try to find a rational explanation for the observed behavior.

In some frameworks, such as Keras, there are no explicit layers to define an atrous convolution. Instead, a standard convolutional layer normally has a parameter to define the dilation rate (in Keras, it's called `dilation_rate`). Of course, the default value is 1, meaning that the kernel will be applied to patches matching its size.

Separable convolution

If we consider an image $X \in \mathcal{R}^{w \times h}$ (single channel) and a kernel $k \in \mathcal{R}^{n \times m}$, the number of operations is $nmwh$. When the kernel is not very small and the image is large, the cost of this computation can be quite high, even with GPU support. An improvement can be achieved by taking into account the associated property of convolutions. In particular, if the original kernel can be split into the dot product of two vectorial kernels, $k^{(1)}$ with dimensions $(n \times 1)$ and $k^{(2)}$ with dimensions $(1 \times m)$, the convolution is said to be **separable**. This means that we can perform a $(n \times m)$ convolution with two subsequent operations:

$$X * k \sim \left(X * \begin{pmatrix} k_1^{(1)} \\ \vdots \\ k_n^{(1)} \end{pmatrix} \right) * \begin{pmatrix} k_1^{(2)} & \cdots & k_m^{(2)} \end{pmatrix}$$

The advantage is clear, because now the number of operations is $(n + m)wh$. In particular, when $nm >> n + m$, it's possible to avoid a large number of multiplications and speed up both the training and the prediction process.

A slightly different approach has been proposed in *Xception: Deep Learning with Depthwise Separable Convolutions, Chollet F., arXiv:1610.02357 [cs.CV]*. In this case, which is properly called **depthwise separable convolution**, the process is split into two steps. The first one operates along the channel axis, transforming it into a single dimensional map with a variable number of channels (for example, if the original diagram is $768 \times 1024 \times 3$, the output of the first stage will be $n \times 768 \times 1024 \times 1$). Then, a standard convolution is applied to the single layer (which can have indeed more than one channel). In the majority of implementations, the default number of output channels for the depthwise convolution is 1 (this is conventionally expressed by saying that the **depth multiplier** is 1). This approach allows a dramatic parameter reduction with respect to a standard convolution. In fact, if the input generic feature map is $X \in \mathcal{R}^{w \times h \times p}$ and we want to perform a standard convolution with q kernels $k^{(i)} \in \mathcal{R}^{n \times m}$, we need to learn $nmqp$ parameters (each kernel $k^{(i)}$ is applied to all input channels). Employing the Depthwise Separable Convolution, the first step (working with only the channels) requires nmp parameters. As the output has still p feature maps and we need to output q channels, the process employs a *trick*: processing each feature map with q 1×1 kernels (in this way, the output will have q layers and the same dimensions). The number of parameters required for the second step is pq, so the total number of parameters becomes $nmp + pq$. Comparing this value with the one required for a standard convolution, we obtain an interesting result:

$$nmp + pq < nmpq \;\Rightarrow\; nm + q < nmq \;\Rightarrow\; nm < q(nm-1) \;\Rightarrow\; q > \frac{nm}{nm-1}$$

As this condition is easily true, this approach is extremely effective in optimizing the training and prediction processes, as well as the memory consumption in any scenario. It's not surprising that the Xception model has been immediately implemented in mobile devices, allowing real-time image classification with very limited resources. Of course, depthwise separable convolutions don't always have the same accuracy as standard ones, because they are based on the assumption that the geometrical features observable inside a channel of a composite feature map are independent of each other. This is not always true, because we know that the effect of multiple layers is based also on their combinations (which increases the expressivity of a network). However, in many cases the final result has an accuracy comparable to some state-of-the-art models; therefore, this technique can very often be considered as a valid alternative to a standard convolution.

Since version 2.1.5, Keras has introduced a layer called `DepthwiseConv2D` that implements a depthwise separable convolution. This layer extends the existing `SeparableConv2D`.

Transpose convolution

A **transpose convolution** (sometimes wrongly called deconvolution, even if the mathematical definition is different) is not very different from a standard convolution, but its goal is to rebuild a structure with the same features as the input sample. Let's suppose that the output of a convolutional network is the feature map $X \in \mathbb{R}^{w' \times h' \times p}$ and we need to build an output element $Y \in \mathbb{R}^{w \times h \times 3}$ (assuming the w and h are the original dimensions). We can achieve this result by applying a transpose convolution with appropriate strides and padding to X. For example, let's suppose that $X \in \mathbb{R}^{128 \times 128 \times 256}$ and our output must be $512 \times 512 \times 3$. The last transpose convolution must learn three filters with strides set to four and same padding. We are going to see some practical examples of this method in the next chapter `Chapter 11`, *Autoencoders* when discussing autoencoders; however, there are no very important differences between transpose and standard convolution in terms of internal dynamics. The main difference is the cost function, because when a transpose convolution is used as the last layer, the comparison must be done between a target image and a reconstructed one. In the next chapter, `Chapter 11`, *Autoencoders* we are also going to analyze some techniques to improve the quality of the output even when the cost function doesn't focus on specific areas of the image.

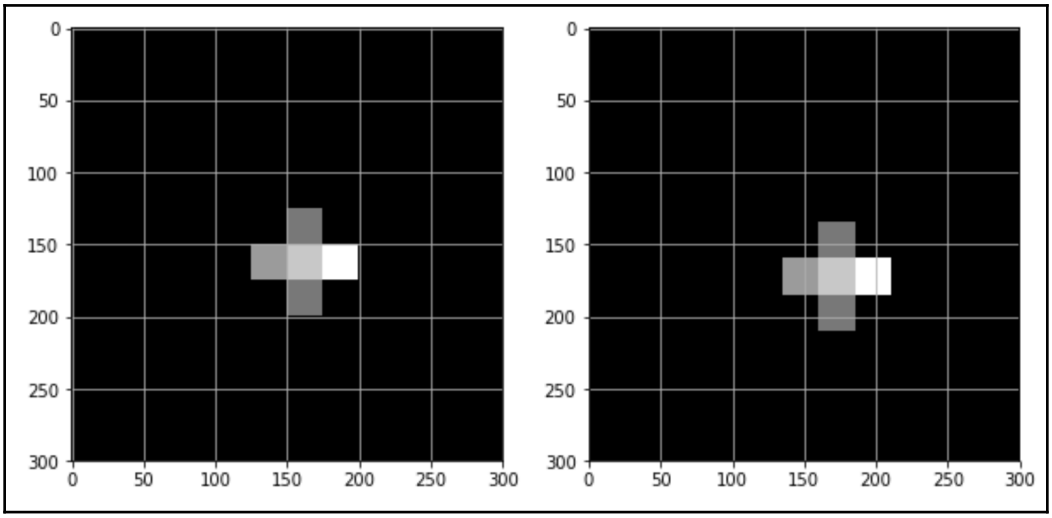
Pooling layers

In a deep convolutional network, **pooling layers** are extremely useful elements. There are mainly two kinds of these structures: **max pooling** and **average pooling**. They both work on patches $p \in \Re^{n \times m}$, shifting horizontally and vertically according to the predefined stride value and transforming the patches into single pixels according to the following rules:

$$\begin{cases} f_{MaxPooling}(X) = max_{i,j} X(i,j) \\ f_{AveragePooling}(X) = \frac{1}{n+m} \sum_{i=1}^{n} \sum_{j=1}^{m} X(i,j) \end{cases}$$

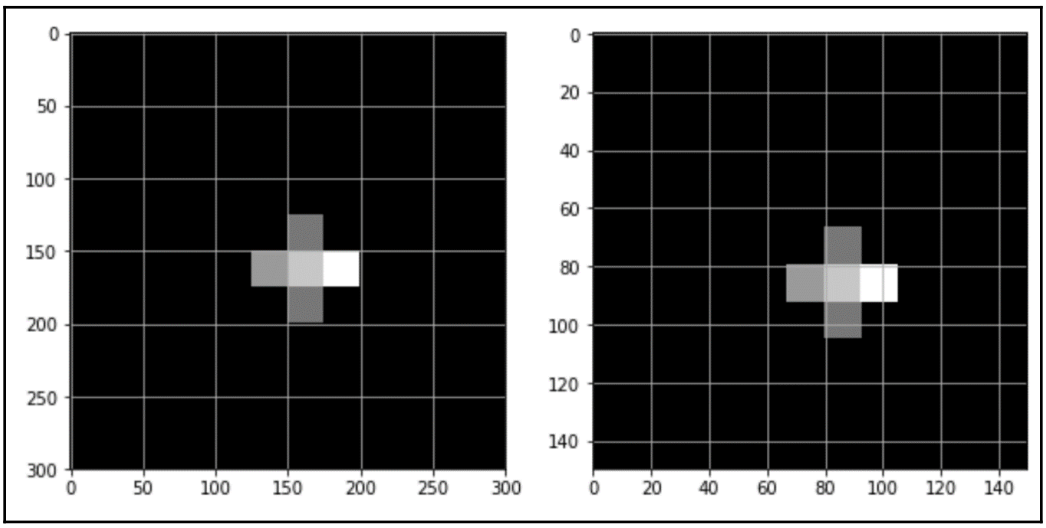
There are two main reasons that justify the use of these layers. The first one is a dimensionality reduction with limited information loss (for example, setting the strides to (*2*, *2*), it's possible to halve the dimensions of an image/feature map). Clearly, all pooling techniques can be more or less lossy (in particular max pooling) and the specific result depends on the single image. In general, pooling layers try to summarize the information contained in a small chunk into a single pixel. This idea is supported by a perceptual-oriented approach; in fact, when the pools are not too large, it's rather unlikely to find high variances in subsequent shifts (natural images have very few isolated pixels). Therefore, all the pooling operations allow us to set up strides greater than one with a mitigated risk of compromising the information content. However, considering several experiments and architectures, I suggest that you set up larger strides in the convolutional layers (in particular, in the first layer of a convolutional sequence) instead of in pooling ones. In this way, it's possible to apply the transformation with a minimum loss and to fully exploit the next fundamental property.

The second (and probably the most important) reason is that they slightly increase the robustness to translations and limited distortions with an effect that is proportional to the pool size. Let's consider the following diagram, representing an original image of a cross and the version after a 10-pixel diagonal translation:



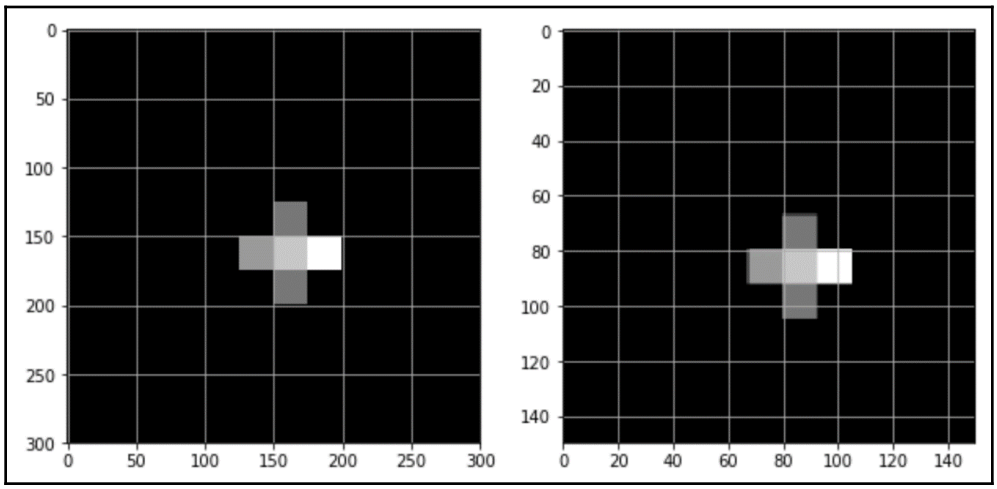
Original image (left); diagonally translated image (right)

This is a very simple example and the translated image is not very different from the original one. However, in a more complex scenario, a classifier could also fail to correctly classify an object in similar conditions. Applying a max pooling (with a (2×2) pool size and 2-pixel strides) on the translated image, we get the following:



Original image (left); result of a max pooling on the translated image (right)

The result is a larger cross, whose arms are slightly more aligned to the axis. When compared with the original image, it's easier for a classifier with a good generalization ability to filter out the spurious elements and recognize the original shape (which can be considered a cross surrounded by a noisy frame). Repeating the same experiment with average pooling (same parameters), we obtain the following:



Original image (left); result of an average pooling on the translated image (right)

In this case, the picture is partially smoothed, but it's still possible to see a better alignment (thanks mainly to the fading effect). Also, if these methods are simple and somewhat effective, the robustness to invariant transformations is never dramatically improved and higher levels of invariance are possible only by increasing the pool size. This choice leads to coarser-grained feature maps whose amount of information is drastically reduced; therefore, whenever it's necessary to extend the classification to samples that can be distorted or rotated, it can be a good idea (which allows working with a dataset that better represents the real data generating process) to use a data augmentation technique to produce artificial images and to also train the classifier on them. However, as pointed out in *Deep Learning, Goodfellow I., Bengio Y., Courville A., MIT Press*, pooling layers can also provide a robust invariance to rotations when they are used together with the output of a multiple convolution layer or a rotated image stack. In fact, in these cases, a single pattern response is elicited and the effect of the pooling layer becomes similar to a collector that standardizes the output. In other words, it will produce the same result without an explicit selection of the best matching pattern. For this reason, if the dataset contains enough samples, pooling layers in intermediate positions of the network can provide a moderate robustness to small rotations, increasing the generalization ability of the whole deep architecture.

As it's easy to see in the previous example, the main difference between the two variants is the final result. Average pooling performs a sort of very simple interpolation, smoothing the borders and avoiding abrupt changes. On the other hand, max pooling is less noisy and can yield better results when the features need to be detected without any kind of smoothing (which could alter their geometry). I always suggest testing both techniques, because it's almost impossible to pick the best method with the right pool size according only to heuristic considerations (above all, when the datasets are not made up of very simple images).

Clearly, it's always preferable to use these layers after a group of convolutions, avoiding very large pool sizes that can irreversibly destroy the information content. In many important deep architectures, the pooling layers are always based on (*2*, *2*) or (*3*, *3*) pools, independently of their position, and the strides are always set to 1 or 2. In both cases, the information loss is proportional to the pool size/strides; therefore, large pools are normally avoided when small features must be detected together with larger ones (for example, foreground and background faces).

Other useful layers

Even if convolution and pooling layers are the backbone of almost all deep convolutional networks, other layers can be helpful to manage specific situations. They are as follows:

- **Padding layers**: These can be employed to increase the size of a feature map (for example, to align it with another one) by surrounding it with a blank frame (*n* black pixels are added before and after each side).
- **Upsampling layers**: These increase the size of a feature map by creating larger blocks out of a single pixel. To a certain extent, they can be considered as a transformation opposite to a pooling layer, even if, in this case, the upsampling is not based on any kind of interpolation. These kinds of layers can be used to prepare the feature maps for transformations similar to the ones obtained with a transpose convolution, even if many experiments confirmed that using larger strides can yield very accurate results without the need of an extra computational step.

- **Cropping layers**: These are helpful for selecting specific rectangular areas of an image/feature map. They are particularly useful in modular architectures, where the first part determines the cropping boundaries (for example, of a face), while the second part, after having removed the background, can perform high-level operations such as detail segmentation (marking the areas of eyes, nose, mouth, and so on). The possibility of inserting these layers directly into a deep neural model avoids multiple data transfers. Unfortunately, many frameworks (such as Keras) don't allow us to use variable boundaries, limiting *de facto* the number of possible use cases.
- **Flattening layers:** These are the conjunction link between feature maps and fully-connected layers. Normally, a single flattening layer is used before processing the output of the convolutional blocks, with a few dense layers terminating in a final Softmax layer (for classifications). The operation is computationally very cheap as it works only with the metadata and doesn't perform any calculations.

Examples of deep convolutional networks with Keras

In the first example, we want to consider again the complete MNIST handwritten digit dataset, but instead of using an MLP, we are going to employ a small deep convolutional network. The first step consists of loading and normalizing the dataset:

```
import numpy as np

from keras.datasets import mnist
from keras.utils import to_categorical

(X_train, Y_train), (X_test, Y_test) = mnist.load_data()

width = height = X_train.shape[1]

X_train = X_train.reshape((X_train.shape[0], width, height,
1)).astype(np.float32) / 255.0
 X_test = X_test.reshape((X_test.shape[0], width, height,
1)).astype(np.float32) / 255.0

Y_train = to_categorical(Y_train, num_classes=10)
Y_test = to_categorical(Y_test, num_classes=10)
```

We can now define the model architecture. The samples are rather small (28×28); therefore it can be helpful to use small kernels. This is not a general rule and it's useful to also evaluate larger kernels (in particular in the first layers); however, many state-of-the-art architectures confirmed large kernel sizes with small images can lead to a performance loss. In my personal experiments, I've always obtained the best results when the largest kernels were $8 \div 10$ smaller than the image dimensions. Our model is made up of the following layers:

1. Input dropout 25%.
2. Convolution with 16 filters, (3 × 3) kernel, strides equal to 1, ReLU activation, and the same padding (the default weight initializer is Xavier). Keras implements the `Conv2D` class, whose main parameters are immediately understandable.
3. Dropout 50%.
4. Convolution with 32 filters, (3 × 3) kernel, strides equal to 1, ReLU activation, and the same padding.
5. Dropout 50%.
6. Average pooling with (2 × 2) pool size and strides equal to 1 (using the Keras class `AveragePooling2D`).
7. Convolution with 64 filters, (3 × 3) kernel, strides equal to 1, ReLU activation, and the same padding.
8. Average pooling with (2 × 2) pool size and strides equal to 1.
9. Convolution with 64 filters, (3 × 3) kernel, strides equal to 1, ReLU activation, and the same padding.
10. Dropout 50%.
11. Average pooling with (2 × 2) pool size and strides equal to 1.
12. Fully-connected layer with 1024 ReLU units.
13. Dropout 50%.
14. Fully-connected layer with 10 Softmax units.

The goal is to capture the low-level features (horizontal and vertical lines, intersections, and so on) in the first layers and use the pooling layers and all the subsequent convolutions to increase the accuracy when distorted samples are presented. At this point, we can create and compile the model (using the Adam optimizer with $\eta = 0.001$ and a decay rate equal to 10^{-5}):

```
from keras.models import Sequential
from keras.layers import Dense, Activation, Dropout, Conv2D,
AveragePooling2D, Flatten
from keras.optimizers import Adam
```

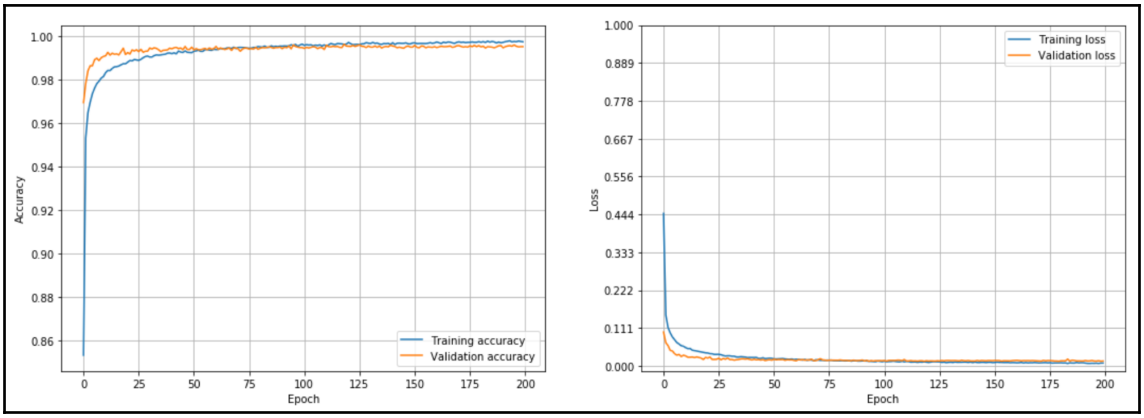
```
model = Sequential()

model.add(Dropout(0.25, input_shape=(width, height, 1), seed=1000))

model.add(Conv2D(16, kernel_size=(3, 3), padding='same'))
model.add(Activation('relu'))
model.add(Dropout(0.5, seed=1000))

model.add(Conv2D(32, kernel_size=(3, 3), padding='same'))
model.add(Activation('relu'))
model.add(Dropout(0.5, seed=1000))

model.add(AveragePooling2D(pool_size=(2, 2), padding='same'))

model.add(Conv2D(64, kernel_size=(3, 3), padding='same'))
model.add(Activation('relu'))

model.add(AveragePooling2D(pool_size=(2, 2), padding='same'))

model.add(Conv2D(64, kernel_size=(3, 3), padding='same'))
model.add(Activation('relu'))
model.add(Dropout(0.5, seed=1000))

model.add(AveragePooling2D(pool_size=(2, 2), padding='same'))

model.add(Flatten())

model.add(Dense(1024))
model.add(Activation('relu'))
model.add(Dropout(0.5, seed=1000))

model.add(Dense(10))
model.add(Activation('softmax'))

model.compile(optimizer=Adam(lr=0.001, decay=1e-5),
              loss='categorical_crossentropy',
              metrics=['accuracy'])
```

We can now proceed to train the model with 200 epochs and a batch size of 256 samples:

```
history = model.fit(X_train, Y_train,
                    epochs=200,
                    batch_size=256,
                    validation_data=(X_test, Y_test))

Train on 60000 samples, validate on 10000 samples
Epoch 1/200
60000/60000 [==============================] - 30s 496us/step - loss:
```

```
0.4474 - acc: 0.8531 - val_loss: 0.0993 - val_acc: 0.9693
Epoch 2/200
60000/60000 [==============================] - 20s 338us/step - loss:
0.1497 - acc: 0.9530 - val_loss: 0.0682 - val_acc: 0.9780
Epoch 3/200
60000/60000 [==============================] - 21s 346us/step - loss:
0.1131 - acc: 0.9647 - val_loss: 0.0598 - val_acc: 0.9839

...

Epoch 199/200
60000/60000 [==============================] - 21s 349us/step - loss:
0.0083 - acc: 0.9974 - val_loss: 0.0137 - val_acc: 0.9950
Epoch 200/200
60000/60000 [==============================] - 22s 373us/step - loss:
0.0083 - acc: 0.9972 - val_loss: 0.0143 - val_acc: 0.9950
```

The final validation accuracy is now `0.9950`, which means that only 50 samples (out of 10,000) have been misclassified. To better understand the behavior, we can plot the accuracy and loss diagrams:



As it's possible to see, both validation accuracy and loss easily reach the optimal values. In particular, the initial validation accuracy is about 0.97 and the remaining epochs are necessary to improve the performance with all those samples, whose shapes can lead to confusion (for example, malformed 8s that resemble 0s, or 7s that are very similar to 1s). It's evident that the *geometric* approach employed by convolutions guarantees a much higher robustness than a standard fully-connected network, thanks also to the contribution of pooling layers, which reduce the variance due to noisy samples.

Example of a deep convolutional network with Keras and data augmentation

In this example, we are going to use the Fashion MNIST dataset, which was freely provided by Zalando as a more difficult replacement for the standard MNIST dataset. In this case, instead of handwritten digits, there are greyscale photos of different articles of clothing. An example of a few samples is shown in the following screenshot:

However, in this case, we want to employ a utility class provided by Keras (`ImageDataGenerator`) in order to create a data-augmented sample set to improve the generalization ability of the deep convolutional network. This class allows us to add random transformations (such as standardization, rotations, shifting, flipping, zooming, shearing, and so on) and output the samples using a Python generator (with an infinite loop). Let's start loading the dataset (we don't need to standardize it, as this transformation is performed by the generator):

```
from keras.datasets import fashion_mnist

(X_train, Y_train), (X_test, Y_test) = fashion_mnist.load_data()
```

At this point, we can create the generators, selecting the transformation that best suits our case. As the dataset is rather *standard* (all the samples are represented only in a few positions), we've decided to augment the dataset by applying a sample-wise standardization (which doesn't rely on the entire dataset), horizontal flip, zooming, small rotations, and small shears. This choice has been made according to an objective analysis, but I suggest the reader repeat the experiment with different parameters (for example, adding whitening, vertical flip, horizontal/vertical shifting, and extended rotations). Of course, increasing the augmentation variability needs larger processed sets. In our case, we are going to use 384,000 training samples (the original size is 60,000), but larger values can be employed to train deeper networks:

```
import numpy as np

from keras.preprocessing.image import ImageDataGenerator
from keras.utils import to_categorical

nb_classes = 10
train_batch_size = 256
test_batch_size = 100
```

```
train_idg = ImageDataGenerator(rescale=1.0 / 255.0,
                               samplewise_center=True,
                               samplewise_std_normalization=True,
                               horizontal_flip=True,
                               rotation_range=10.0,
                               shear_range=np.pi / 12.0,
                               zoom_range=0.25)

train_dg = train_idg.flow(x=np.expand_dims(X_train, axis=3),
                          y=to_categorical(Y_train,
num_classes=nb_classes),
                          batch_size=train_batch_size,
                          shuffle=True,
                          seed=1000)

test_idg = ImageDataGenerator(rescale=1.0 / 255.0,
                              samplewise_center=True,
                              samplewise_std_normalization=True)

test_dg = train_idg.flow(x=np.expand_dims(X_test, axis=3),
                         y=to_categorical(Y_test, num_classes=nb_classes),
                         shuffle=False,
                         batch_size=test_batch_size,
                         seed=1000)
```

Once an image data generator has been initialized, it must be fitted, specifying the input dataset and the desired batch size (the output of this operation is the actual Python generator). The test image generator is voluntarily kept without transformations except for normalization and standardization, in order to avoid a validation on a dataset drawn from a different distribution. At this point, we can create and compile our network, using 2D convolutions based on Leaky ReLU activations (using the `LeakyReLU` class, which replaces the standard layer `Activation`), batch normalizations, and max poolings:

```
from keras.models import Sequential
from keras.layers import Activation, Dense, Flatten, LeakyReLU, Conv2D,
MaxPooling2D, BatchNormalization
from keras.optimizers import Adam

model = Sequential()

model.add(Conv2D(filters=32,
                 kernel_size=(3, 3),
                 padding='same',
                 input_shape=(X_train.shape[1], X_train.shape[2], 1)))

model.add(BatchNormalization())
model.add(LeakyReLU(alpha=0.1))
```

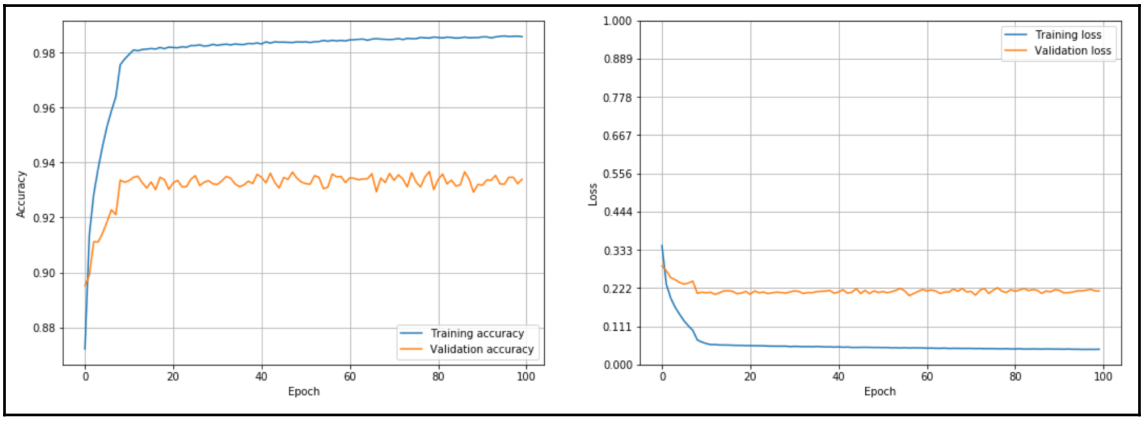
```
model.add(Conv2D(filters=64,
                 kernel_size=(3, 3),
                 padding='same'))

model.add(BatchNormalization())
model.add(LeakyReLU(alpha=0.1))

model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(filters=64,
                 kernel_size=(3, 3),
                 padding='same'))

model.add(BatchNormalization())
model.add(LeakyReLU(alpha=0.1))

model.add(Conv2D(filters=128,
                 kernel_size=(3, 3),
                 padding='same'))

model.add(BatchNormalization())
model.add(LeakyReLU(alpha=0.1))

model.add(Conv2D(filters=128,
                 kernel_size=(3, 3),
                 padding='same'))

model.add(BatchNormalization())
model.add(LeakyReLU(alpha=0.1))

model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Flatten())

model.add(Dense(units=1024))
model.add(BatchNormalization())
model.add(LeakyReLU(alpha=0.1))

model.add(Dense(units=1024))
model.add(BatchNormalization())
model.add(LeakyReLU(alpha=0.1))

model.add(Dense(units=nb_classes))
model.add(Activation('softmax'))

model.compile(loss='categorical_crossentropy',
              optimizer=Adam(lr=0.0001, decay=1e-5),
              metrics=['accuracy'])
```

All the batch normalizations are always applied to the linear transformation before the activation function. Considering the additional complexity, we are also going to use a callback, which is a class that Keras uses in order to perform in-training operations. In our case, we want to reduce the learning rate when the validation loss stops improving. The specific callback is called `ReduceLROnPlateau` and it's tuned in order to reduce η multiplying it by `0.1` (after a number of epochs equal to the value of the `patience` parameter) with a cooldown period (the number of epochs to wait before restoring the original learning rate) of 1 epoch and a minimum $\eta = 10^{-6}$. The training method is now `fit_generator()`, which accepts Python generators instead of finite datasets and the number of iterations per epoch (all the other parameters are the same as implemented by `fit()`):

```
from keras.callbacks import ReduceLROnPlateau

nb_epochs = 100
steps_per_epoch = 1500

history = model.fit_generator(generator=train_dg,
                              epochs=nb_epochs,
                              steps_per_epoch=steps_per_epoch,
                              validation_data=test_dg,
                              validation_steps=int(X_test.shape[0] /
test_batch_size),
                              callbacks=[
                                 ReduceLROnPlateau(factor=0.1, patience=1,
cooldown=1, min_lr=1e-6)
                              ])

Epoch 1/100
1500/1500 [==============================] - 471s 314ms/step - loss: 0.3457
- acc: 0.8722 - val_loss: 0.2863 - val_acc: 0.8952
Epoch 2/100
1500/1500 [==============================] - 464s 309ms/step - loss: 0.2325
- acc: 0.9138 - val_loss: 0.2721 - val_acc: 0.8990
Epoch 3/100
1500/1500 [==============================] - 460s 307ms/step - loss: 0.1929
- acc: 0.9285 - val_loss: 0.2522 - val_acc: 0.9112

...

Epoch 99/100
1500/1500 [==============================] - 449s 299ms/step - loss: 0.0438
- acc: 0.9859 - val_loss: 0.2142 - val_acc: 0.9323
Epoch 100/100
1500/1500 [==============================] - 449s 299ms/step - loss: 0.0443
```

```
- acc: 0.9857 - val_loss: 0.2136 - val_acc: 0.9339
```

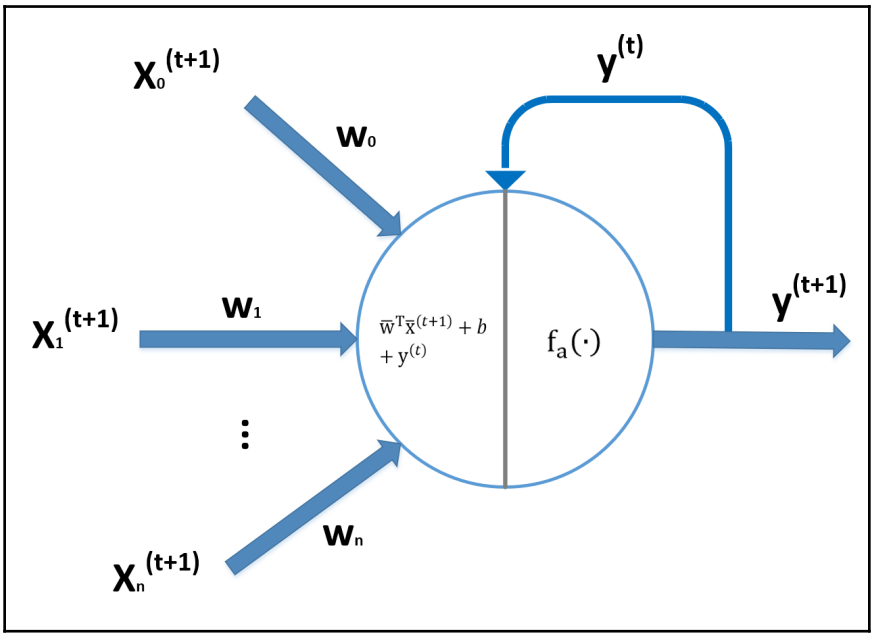
In this case, the complexity is higher and the result is not as accurate as the one obtained with the standard MNIST dataset. The validation and loss plots are shown in the following graph:



The loss plot doesn't show a U-curve, but it seems that there are no real improvements starting from the 20th epoch. This is also confirmed by the validation plot, which continues oscillating between 0.935 and about 0.94. On the other side, the training loss hasn't reached its minimum (nor has the training accuracy), mainly because of the batch normalizations. However, considering several benchmarks, the result is not bad (even if state-of-the-art models can reach a validation accuracy of about 0.96). I suggest that the reader try different configurations (with and without dropout and other activations) based on deeper architectures with larger training sets. This example offers many chances to practice with this kind of models, as the complexity is not as high as to require dedicated hardware, but at the same time, there are many ambiguities (for example, between shirts and t-shirts) that can reduce the generalization ability.

Recurrent networks

All the models that we have analyzed until now have a common feature. Once the training process is completed, the weights are frozen and the output depends only on the input sample. Clearly, this is the expected behavior of a classifier, but there are many scenarios where a prediction must take into account the history of the input values. A time series is a classic example. Let's suppose that we need to predict the temperature for the next week. If we try to use only the last known $x^{(t)}$ value and an MLP trained to predict $x^{(t+1)}$, it's impossible to take into account temporal conditions like the season, the history of the season over the years, the position in the season, and so on. The regressor will be able to associate the output that yields the minimum average error, but in real-life situations, this isn't enough. The only reasonable way to solve this problem is to define a new architecture for the artificial neuron, to provide it with a memory. This concept is shown in the following diagram:



Now the neuron is no longer a pure feed-forward computational unit because the feedback connection forces it to remember its past and use it in order to predict new values. The new dynamic rule is now as follows:

$$\begin{cases} y^{(t+1)} = f_a\left(\bar{w}^T \bar{x}^{(t+1)} + b + y^{(t)}\right) \quad for \;\; t > 0 \\ y^{(0)} = 0 \end{cases}$$

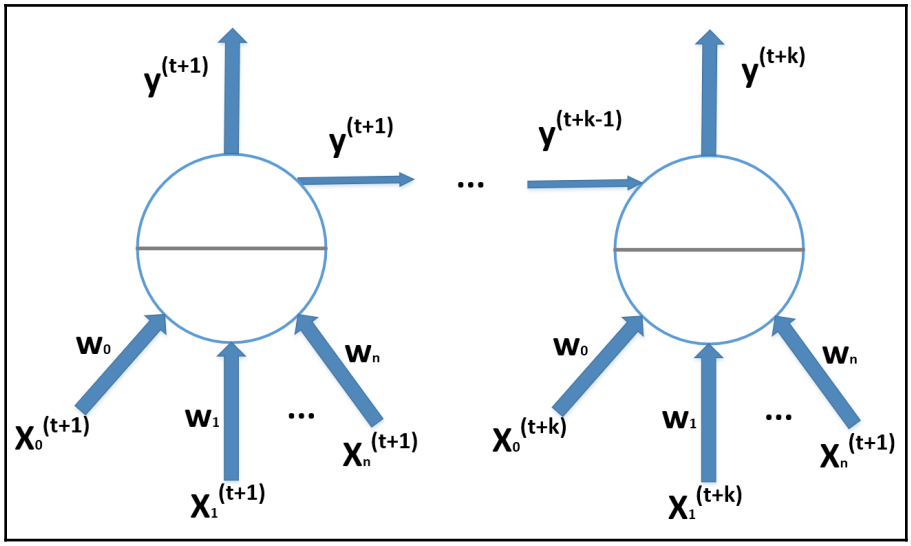
The previous prediction is fed back and summed to new linear output. The resulting value is transformed by the activation function in order to produce the actual new output (conventionally the first output is null, but this is not a constraint). An immediate consideration concerns the activation function—this is a dynamic system that could easily become unstable. The only way to prevent this phenomenon is to employ saturating functions (such as the sigmoid or hyperbolic tangent). In fact, whatever the input is, the output can never *explode* by moving towards +∞ or -∞.

Suppose that, instead, we were to use a ReLU activation—under some conditions, the output will grow indefinitely, leading to an overflow. Clearly, the situation is even worse with a linear activation and could be very similar even when using a Leaky ReLU or ELU. Hence, it's obvious that we need to select saturating functions, but is this enough to ensure stability? Even if a hyperbolic tangent (as well as a sigmoid) has two stable points (*-1* and *+1*), this isn't enough to ensure stability. Let's imagine that the output is affected by noise and oscillates around 0.0. The unit cannot converge towards a value and remains trapped in a limit cycle.

Luckily, the possibility to learn the weights allows us to increase the robustness to noise, avoiding that limited changes in the input could invert the dynamic of the neuron. This is a very important (and easy to prove) result that guarantees stability under very simple conditions, but again, what is the price that we need to pay? Is it anything simple and straightforward? Unfortunately, the answer is negative and the price for stability is extremely high. However, before discussing this problem, let's show how a simple recurrent network can be trained.

Backpropagation through time (BPTT)

The simplest way to train an RNN is based on a representational trick. As the input sequences are limited and their length can be fixed, it's possible to restructure the simple neuron with a feedback connection as an unrolled feed-forward network. In the following diagram, there's an example with *k* timesteps:



Example of unrolled recurrent network

This network (which can be easily extended to more complex architecture with several layers) is exactly like an MLP, but in this case, the weights of each *clone* are the same. The algorithm called **BPTT** is the natural extension of the standard learning technique to unrolled recurrent networks. The procedure is straightforward. Once all the outputs have been computed, it's possible to determine the value of the cost function for every single network. At this point, starting from the last step, the corrections (the gradients) are computed and stored, and the process is repeated until the initial step. Then, all of the gradients are summed and applied to the network. As every single contribution is based on a precise *temporal experience* (made up of a local sample and a previous memory element), the standard backpropagation will learn how to manage a dynamic condition as if it were a point-wise prediction. However, we know that the actual network is not unrolled and the past dependencies are theoretically propagated and remembered. I voluntarily used the word *theoretically*, because all practical experiments show a completely different behavior that we are going to discuss. This technique is very easy to implement, but it can be very expensive for deep networks that must be unrolled for a large number of timesteps. For this reason, a variant called **truncated backpropagation through time** (**TBPTT**) has been proposed (in *Subgrouping reduces complexity and speeds up learning in recurrent networks, Zipser D., Advances in Neural Information Processing Systems, II 1990*).

The idea is to use two sequence lengths t_1 and t_2 (with $t_1 >> t_2$)—the longer one (t_1) is employed for the feed-forward phase, while the shorter length (t_2) is used to train the network. At first sight, this version seems like a normal BPTT with a short sequence; however, the key idea is to force the network to update the hidden states with more pieces of information and then compute the corrections according to the result of the longer sequence (even if the updates are propagated to a limited number of previous timesteps). Clearly, this is an approximation that can speed up the training process, but the final result is normally comparable with the one obtained by processing long sequences, in particular when the dependencies can be split into shorter temporal chunks (and therefore the assumption is that there are no very long dependencies).

Even if the BPTT algorithm is mathematically correct and it's not difficult to learn short-term dependencies (corresponding to short unrolled networks), several experiments confirmed that it's extremely difficult (or almost impossible) learning long-term dependencies. In other words, it's easy to exploit past experiences whose contribution is limited to a short window (and therefore whose importance is limited because they cannot manage the most complex trends) but the network cannot easily learn all behaviors that, for example, have a periodicity of hundreds of timesteps. In 1994, Bengio, Simard, and Frasconi provided a theoretical explanation of the problem (in *Learning Long-Term Dependencies with Gradient Descent is Difficult, Bengio Y., Simard P., Frasconi P., IEEE Transactions on Neural Networks, 5/1994*). The mathematical details are rather complex, because they involve dynamic system theory; however, the final result is that a network whose neurons are forced to become robust to noise (the normal expected behavior) is affected by the vanishing gradients problem when $t \rightarrow \infty$. More generally, we can represent a vectorial recurrent neuron dynamic as follows:

$$\bar{y}^{(t+1)} = f_a \left(W^T \bar{x}^{(t+1)} + \bar{b} + \bar{y}^{(t)} \right)$$

The multiplicative effect of BPTT forces the gradients to be proportional to W^t. If the largest absolute eigenvalue (also known as spectral radius) of W is smaller than 1, then the following applies:

$$\lim_{t \to \infty} W^t = 0$$

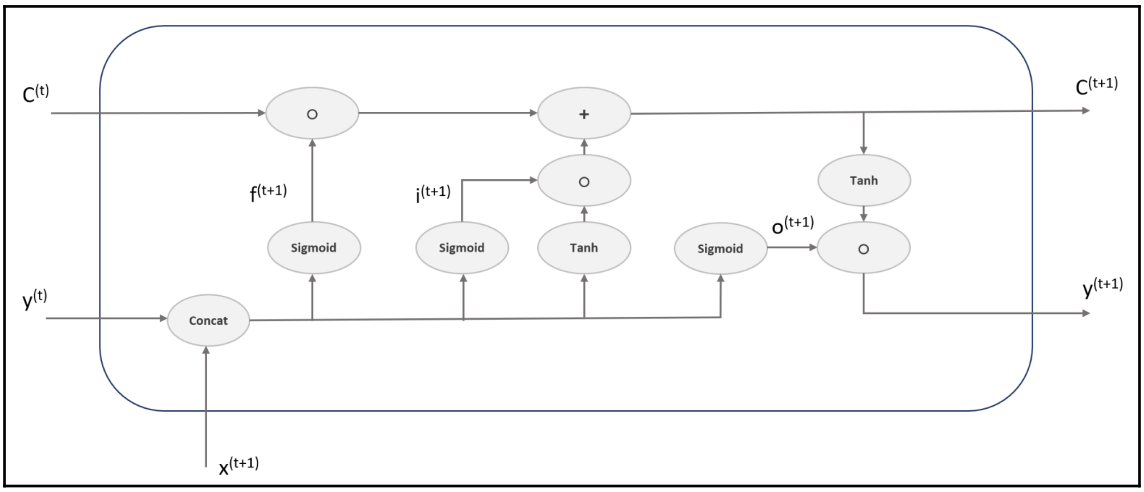
More simply, we can re-express the result saying that the magnitude of the gradients is proportional to the length of the sequences and even if the condition is asymptotically valid, many experiments confirmed that the limited precision of numeric computations and the exponential decay due to subsequent multiplications can force the gradients to vanish even when the sequences are not extremely long. This seems to be the end of any RNN architecture, but luckily more recent approaches have been designed and proposed to resolve this problem, allowing RNNs to learn both short and long-term dependencies without particular complications. A new era of RNNs started and the results were immediately outstanding.

LSTM

This model (which represents the state-of-the-art recurrent cell in many fields) was proposed in 1997 by Hochreiter and Schmidhuber (in *Long Short-Term Memory, Hochreiter S., Schmidhuber J., Neural Computation, Vol. 9, 11/1997*) with the emblematic name **long-short-term memory** (**LSTM**). As the name suggests, the idea is to create a more complex artificial recurrent neuron that can be plugged into larger networks and trained without the risk of vanishing and, of course, exploding gradients. One of the key elements of classic recurrent networks is that they are focused on learning, but not on selectively forgetting. This ability is indeed necessary for optimizing the memory in order to remember what is really important and removing all those pieces of information that are not necessary to predict new values.

To achieve this goal, LSTM exploits two important features (it's helpful to expose them before discussing the model). The first one is an explicit state, which is a separate set of variables that store the elements necessary to build long and short-term dependencies, including the current state. These variables are the building blocks of a mechanism called **constant error carousel** (**CEC**), named in this way because it's responsible for the cyclical and internal management of the error provided by the backpropagation algorithm. This approach allows the correction of the weights without suffering the multiplicative effect anymore. The internal LSTM dynamics allow better understanding of how the error is safely fed back; however, the exact explanation of the training procedure (which is always based on the gradient descent) is beyond the scope of this book and can be found in the aforementioned paper.

The second feature is the presence of gates. We can simply define a gate as an element that can modulate the amount of information flowing through it. For example, if $y = ax$ and a is a variable bounded between 0 and 1, it can be considered as a gate, because when it's equal to 0, it blocks the input x; when it's equal to 1, it allows the input to flow in without restrictions; and when it has an intermediate value, it reduces the amount of information proportionally. In LSTMs, gates are managed by sigmoid functions, while the activations are based on hyperbolic tangents (whose symmetry guarantees better performances). At this point, we can show the structural diagram of an LSTM cell and discuss its internal dynamics:



The first (and most important) element is the memory state, which is responsible for the dependencies and for the actual output. In the diagram, it is represented by the upper line and its dynamics are represented by the following general equation:

$$C^{(t+1)} = g_1(C^{(t)}) + g_2(\bar{x}^{(t+1)}, \bar{y}^{(t)})$$

So, the state depends on the previous value, on the current input, and on the previous output. Let's start with the first term, introducing the forget gate. As the name says, it's responsible for the persistence of the existing memory elements or for their deletion. In the diagram, it's represented by the first vertical block and its value is obtained by considering the concatenation of previous output and current input:

$$\bar{f}^{(t+1)} = \sigma\left(W_f \cdot \begin{pmatrix} \bar{y}^{(t)} \\ \bar{x}^{(t+1)} \end{pmatrix} + \bar{b}_f\right)$$

The operation is a classical neuron activation with a vectorial output. An alternative version can use two weight matrices and keep the input elements separated:

$$\bar{f}^{(t+1)} = \sigma\left(W_f \cdot \bar{x}^{(t+1)} + V_f \cdot \bar{y}^{(t)} + \bar{b}_f\right)$$

However, I prefer the previous version, because it can better express the homogeneity of input and output, and also their consequentiality. Using the forget gate, it's possible to determine the value of $g_1(C^{(t)})$ using the Hadamard (or element-wise) product:

$$g_1(C^{(t)}) = \bar{f}^{(t+1)} \circ C^{(t)}$$

The effect of this computation is filtering the content of $C^{(t)}$ that must be preserved and the validity degree (which is proportional to the value of $f^{(t+1)}$). If the forget gate outputs a value close to 1, the corresponding element is still considered valid, while lower values determine a sort of obsolescence that can even lead the cell to completely remove an element when the forget gate value is 0 or close to it. The next step is to consider the amount of the input sample that must be considered to update the state. This task is achieved by the input gate (second vertical block). The equation is perfectly analogous to the previous one:

$$\bar{i}^{(t+1)} = \sigma\left(W_i \cdot \begin{pmatrix} \bar{y}^{(t)} \\ \bar{x}^{(t+1)} \end{pmatrix} + \bar{b}_i\right)$$

However, in this case, we also need to compute the term that must be added to the current state. As already mentioned, LSTM cells employ hyperbolic tangents for the activations; therefore, the new contribution to the state is obtained as follows:

$$\hat{C}^{(t+1)} = tanh\left(W_c \cdot \begin{pmatrix} \bar{y}^{(t)} \\ \bar{x}^{(t+1)} \end{pmatrix} + \bar{b}_c\right)$$

Using the input gate and the state contribution, it's possible to determine the function $g_2(x^{(t+1)}, y^{(t)})$:

$$g_2\left(\bar{x}^{(t+1)}, \bar{y}^{(t)}\right) = \bar{i}^{(t+1)} \circ \hat{C}^{(t+1)}$$

Hence, the complete state equation becomes as follows:

$$C^{(t+1)} = \left(\bar{f}^{(t+1)} \circ C^{(t)}\right) + \left(\bar{i}^{(t+1)} \circ \hat{C}^{(t+1)}\right)$$

Now, the inner logic of an LSTM cell is more evident. The state is based on the following:

- A dynamic balance between previous experience and its re-evaluation according to new experience (modulated by the forget gate)
- The *semantic* effect of the current input (modulated by the input gate) and the potential additive activation

Realistic scenarios are many. It's possible that a new input forces the LSTM to reset the state and store the new incoming value. On the other hand, the input gate can also remain closed, giving a very low priority to the new input (together with the previous output). In this case, the LSTM, considering the long-term dependencies, can decide to discard a sample that is considered noisy and not necessarily able to contribute to an accurate prediction. In other situations, both the forget and input gates can be partially open, letting only some values influence the state. All these possibilities are managed by the learning process through the correction of the weight matrices and the biases. The difference with BPTT is that the long-term dependencies are no longer impeded by the vanishing gradients problem.

The last step is determining the output. The third vertical block is called the output gate and controls the information that must transit from the state to the output unit. Its equation is as follows:

$$\bar{o}^{(t+1)} = \sigma\left(W_o \cdot \begin{pmatrix} \bar{y}^{(t)} \\ \bar{x}^{(t+1)} \end{pmatrix} + \bar{b}_o\right)$$

The actual output is hence determined as follows:

$$\bar{y}^{(t+1)} = \bar{o}^{(t+1)} \circ tanh(C^{(t+1)})$$

An important consideration concerns the gates. They are all fed with the same vector, containing the previous output and the current input. As they are homogenous values, the concatenation yields a coherent entity that encodes a sort of *inverse* cause-effect relationship (this is an improper definition, as we work with previous effect and current cause). The gates work like logistic regressions without thresholding; therefore, they can be considered as pseudo-probability vectors (not distributions, as each element is independent). The forget gate expresses the probability that last sequence (effect, cause) is more important than the current state; however, only the input gate has the responsibility to grant it the right to influence the new state. Moreover, the output gate expresses the probability that the current sequence is able to let the current state flow out. The dynamic is indeed very complex and has some drawbacks. For example, when the output gate remains closed, the output is close to zero and this influences both forget and input gates. As they control the new state and the CEC, they could limit the amount of incoming information and consequent corrections, leading to poor performance.

A simple solution that can mitigate this problem is provided by a variant called **peephole LSTM**. The idea is to feed the previous state to every gate so that they can take decisions more independently. The generic gate equation becomes as follows:

$$\bar{g}^{(t+1)} = \sigma\left(W_g \cdot \begin{pmatrix} \bar{y}^{(t)} \\ \bar{x}^{(t+1)} \end{pmatrix} + U_g \cdot C^{(t)} + \bar{b}_g\right)$$

The new set of weights U_g (for all three gates) must be learned in the same way as the standard W_g and b_g. The main difference with a classic LSTM is that the sequential dynamic: forget gate | input gate | new state | output gate | actual output is now partially shortcutted. The presence of the state in every gate activation allows them to exploit multiple recurrent connections, yielding a better accuracy in many complex situations. Another important consideration is about the learning process: in this case, the peepholes are closed and the only feedback channel is the output gate. Unfortunately, not every LSTM implementation support peepholes; however, several studies confirmed that in most cases all the models yield similar performances.

Xingjian et al. (in *Convolutional LSTM Network: A Machine Learning Approach for Precipitation Nowcasting, Xingjian S., Zhourong C., Hao W., Dit-Yan Y., Wai-kin W., Wang-Chun W., arXiv:1506.04214 [cs.CV]*) proposed a variant called **convolutional LSTM**, which clearly mixes Convolutions and LSTM cells. The main internal difference concerns the gate computations, which now become (without peepholes, which however, can always be added):

$$\bar{g}^{(t+1)} = \sigma\left(W_g * \begin{pmatrix} \bar{y}^{(t)} \\ \bar{x}^{(t+1)} \end{pmatrix} + \bar{b}_g\right)$$

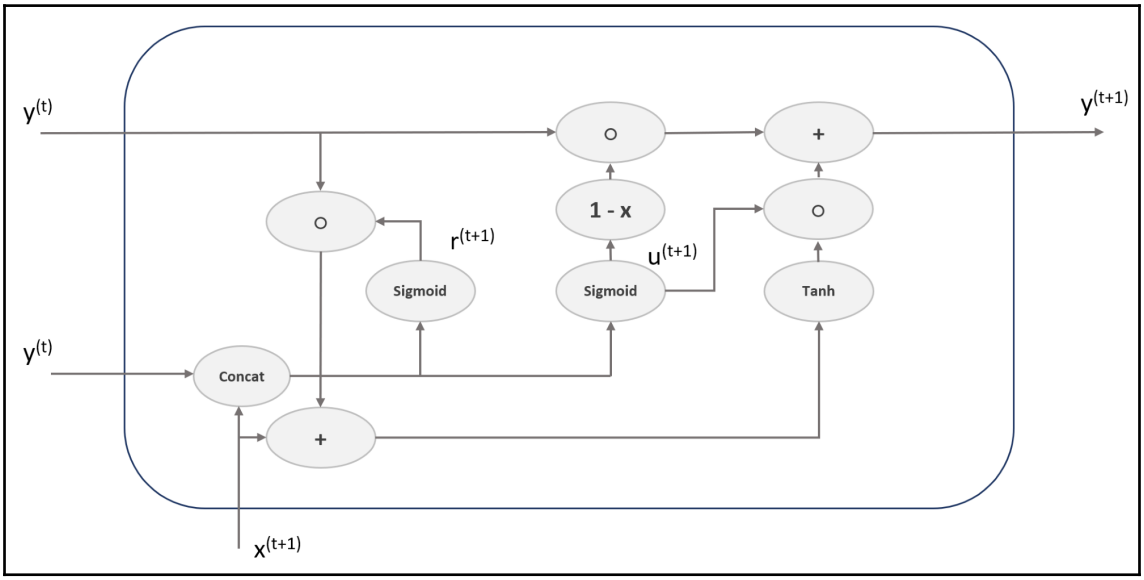
W_g is now a kernel that is convoluted with the input-output vector (which is usually the concatenation of two images). Of course, it's possible to train any number of kernels to increase the decoding power of the cell and the output will have a shape equal to (*batch size* × *width* × *height* × *kernels*). This kind of cell is particularly useful for joining spatial processing with a robust temporal approach. Given a sequence of images (for example, satellite images, game screenshots, and so on), a convolutional LSTM network can learn long-term relationships that are manifested through geometric feature evolutions (for example, cloud movements or specific sprite strategies that it's possible to anticipate considering a long history of events). This approach (even with a few modifications) is widely employed in Deep Reinforcement Learning in order to solve complex problems where the only input is provided by a sequence of images. Of course, the computational complexity is very high, in particular when many subsequent layers are used; however, the results outperformed any existing method and this approach became one of the first choices to manage this kind of problem.

Another important variant, which is common to many Recurrent Neural Networks, is provided by a bidirectional interface. This isn't an actual layer, but a strategy that is employed in order to join the forward analysis of a sequence with the backward one. Two cellblocks are fed with a sequence and its inverse and the output, for example, is concatenated and used for further processing steps. In fields such as NLP, this method allows us to dramatically improve the accuracy of classifications and real-time translations. The reason is strictly related to the rules underlying the structure of a sequence. In natural language, a sentence $w_1\ w_2\ ...\ w_n$ has forward relationships (for example, a singular noun can be followed by *is*), but the knowledge of backward relationships (for example, the sentence *this place is pretty awful*) permits avoiding common mistakes that, in the past, had to be corrected using post-processing steps (the initial translation of *pretty* could be similar to the translation of *nice*, but a subsequent analysis can reveal that the adjective mismatches and a special rule can be applied). Deep learning, on the other side, is not based on *special rules*, but on the ability to learn an internal representation that should be autonomous in making final decisions (without further external aids) and bidirectional LSTM networks help in reaching this goal in many important contexts.

Keras implements the classes `LSTM` since its origins. It also provides a `Bidirectional` class wrapper that can be used with every RNN layer in order to obtain a double output (computed with the forward and backward sequences). Moreover, in Keras 2 there are optimized versions of LSTM based on NVIDIA CUDA (`CuDNNLSTM`), which provide very high performance when a compatible GPU is available. In the same package, it's possible to also find the `ConvLSTM2D` class, which implements a convolutional LSTM layer. In this case, the reader can immediately identify many of the parameters, as they are the same as a standard convolutional layer.

GRU

This model, named **Gated recurrent unit** (**GRU**), proposed by Cho et al. (in *Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation, Cho K., Van Merrienboer B., Gulcehre C., Bahdanau D., Bougares F., Schwenk H., Bengio Y., arXiv:1406.1078 [cs.CL]*) can be considered as a simplified LSTM with a few variations. The structure of a generic full-gated unit is represented in the following diagram:



The main differences from LSTM are the presence of only two gates and the absence of an explicit state. These simplifications can speed both the training and the prediction phases while avoiding the vanishing gradient problem.

The first gate is called the **reset gate** (conventionally denoted with the letter *r*) and its function is analogous to the forget gate:

$$\bar{r}^{(t+1)} = \sigma\left(W_r \cdot \begin{pmatrix} \bar{y}^{(t)} \\ \bar{x}^{(t+1)} \end{pmatrix} + \bar{b}_r\right)$$

Similar to the forget gate, its role is to decide what content of the previous output must be preserved and the relative degree. In fact, the additive contribution to new output is obtained as follows:

$$\hat{y}^{(t+1)} = tanh\left(W_y \cdot \bar{x}^{(t+1)} + \bar{r}^{(t+1)} \circ \left(V_y \cdot \bar{y}^{(t)}\right) + \bar{b}_y\right)$$

In the previous expression, I've preferred to separate the weight matrices to better exposes the behavior. The argument of *tanh(•)* is the sum of a linear function of the new input and a weighted term that is a function of the previous state. Now, it's clear how the reset gate works: it modulates the amount of history (accumulated in the previous output value) that must be preserved and what instead can be discarded. However, the reset gate is not enough to determine the right output with enough accuracy, considering both short and long-term dependencies. In order to increase the expressivity of the unit, an update gate (with a role similar to the LSTM input gate) has been added:

$$\bar{u}^{(t+1)} = \sigma\left(W_u \cdot \begin{pmatrix} \bar{y}^{(t)} \\ \bar{x}^{(t+1)} \end{pmatrix} + \bar{b}_u\right)$$

The update gate controls the amount of information that must contribute to the new output (and hence to the state). As it's a value bounded between *0* and *1*, GRUs are trained to mix old output and new additive contribution with an operation similar to a weighted average:

$$\bar{y}^{(t+1)} = \bar{u}^{(t+1)} \circ \hat{y}^{(t+1)} + \left(I - \bar{u}^{(t+1)}\right) \circ \bar{y}^{(t)}$$

Therefore, the update gate becomes a modulator that can select which components of each flow must be output and stored for the next operation. This unit is structurally simpler than an LSTM, but several studies confirmed that its performance is on average, equivalent to LSTM, with some particular cases when GRU has even outperformed the more complex cell. My suggestion is that you test both models, starting with LSTM. The computational cost has been dramatically reduced by modern hardware and in many contexts the advantage of GRUs is negligible. In both cases, the philosophy is the same: the error is kept inside the cell and the weights of the gates are corrected in order to maximize the accuracy. This behavior prevents the multiplicative cascade of small gradients and increases the ability to learn very complex temporal behaviors.

However, a single cell/layer would not be able to successfully achieve the desired accuracy. In all these cases, it's possible to stack multiple layers made up of a variable number of cells. Every layer can normally output the last value or the entire sequence. The former is used when connecting the LSTM/GRU layer to a fully-connected one, while the whole sequence is necessary to feed another recurrent layer. We are going to see how to implement these techniques with Keras in the following example.

Just like for LSTMs, Keras implements the`GRU` class and its NVIDIA CUDA optimized version `CuDNNGRU`.

Example of an LSTM network with Keras

In this example, we want to test the ability of an LSTM network to learn long-term dependencies. For this reason, we employ a dataset called Zuerich Monthly Sunspots (freely provided by Andrews and Herzberg in 1985) containing the numbers observed in all the months starting from 1749 to 1983 (please read the information box for how to download the dataset). As we are not interested in the dates, we need to parse the file in order to extract only the values needed for the time series (which contains 2,820 steps):

```
import numpy as np

dataset_filename = '<YOUR_PATH>\dataset.csv'

n_samples = 2820
data = np.zeros(shape=(n_samples, ), dtype=np.float32)

with open(dataset_filename, 'r') as f:
    lines = f.readlines()
for i, line in enumerate(lines):
    if i == 0:
        continue
    if i == n_samples + 1:
        break
    _, value = line.split(',')
    data[i-1] = float(value)
```

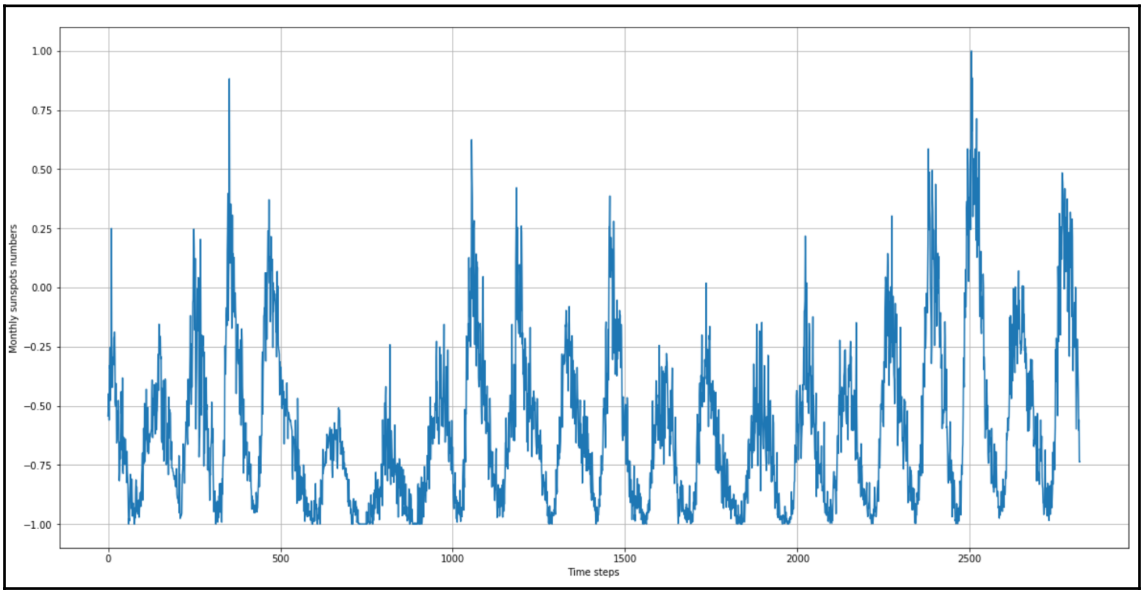
Alternatively, it's possible to load the CSV dataset using pandas (`https://pandas.pydata.org`), which is a powerful data manipulation/analysis library (for further information, please refer to *Learning pandas Second Edition, Heydt M., Packt*):

```
import pandas as pd

dataset_filename = '<YOUR_PATH>\dataset.csv'

df = pd.read_csv(dataset_filename, index_col=0, header=0).dropna()
data = df.values.astype(np.float32).squeeze()
```

The values are unnormalized and as LSTMs work with hyperbolic tangents, it's helpful to normalize them in the interval `-1` and `1`. We can easily perform this step using the Scikit-Learn class `MinMaxScaler`:

```
from sklearn.preprocessing import MinMaxScaler

mmscaler = MinMaxScaler((-1.0, 1.0))
data = mmscaler.fit_transform(data.reshape(-1, 1))
```

The complete dataset is shown in the following diagram:



In order to train the model, we have decided to use 2,300 samples for training and the remaining 500 for validation (corresponding to about 42 years). The input of the model is a batch of sequences of 15 samples (shifted along the time axis) and the output is the subsequent month; therefore, before training, we need to prepare the dataset:

```
sequence_length = 15

X_ts = np.zeros(shape=(n_samples - sequence_length, sequence_length, 1),
dtype=np.float32)
Y_ts = np.zeros(shape=(n_samples - sequence_length, 1), dtype=np.float32)

for i in range(0, data.shape[0] - sequence_length):
    X_ts[i] = data[i:i + sequence_length]
    Y_ts[i] = data[i + sequence_length]

X_ts_train = X_ts[0:2300, :]
Y_ts_train = Y_ts[0:2300]

X_ts_test = X_ts[2300:2800, :]
Y_ts_test = Y_ts[2300:2800]
```

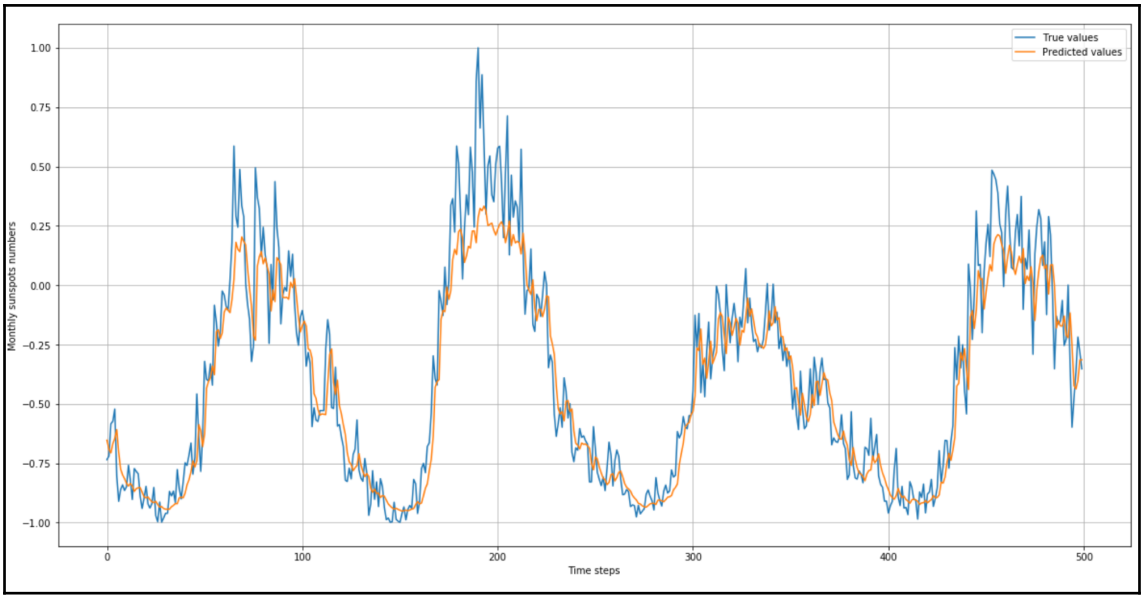
Now, we can create and compile a simple model with a single stateful LSTM layer containing four cells, followed by a hyperbolic tangent output neuron (I always suggest that the reader experiment with more complex architectures and different parameters):

```
from keras.models import Sequential
from keras.layers import LSTM, Dense, Activation
from keras.optimizers import Adam

model = Sequential()

model.add(LSTM(4, stateful=True, batch_input_shape=(20, sequence_length,
1)))

model.add(Dense(1))
model.add(Activation('tanh'))

model.compile(optimizer=Adam(lr=0.001, decay=0.0001),
              loss='mse',
              metrics=['mse'])
```

Setting the `stateful=True` parameter in the `LSTM` class forces Keras not to reset the state after each batch. In fact, our goal is learning long-term dependencies and the internal LSTM state must reflect the overall trend. When an LSTM network is stateful, it's also necessary to specify the batch size in the input shape (through the `batch_input_shape` parameter). In our case, we have selected a batch size equal to 20 samples. The optimizer is `Adam` with a higher decay (to avoid instabilities) and a loss based on the mean squared error (which is the most common choice in this kind of scenario). At this point, we can train the model (for 100 epochs):

```
model.fit(X_ts_train, Y_ts_train,
          batch_size=20,
          epochs=100,
          shuffle=False,
          validation_data=(X_ts_test, Y_ts_test))

Train on 2300 samples, validate on 500 samples
Epoch 1/100
2300/2300 [==============================] - 11s 5ms/step - loss: 0.4905 -
mean_squared_error: 0.4905 - val_loss: 0.1827 - val_mean_squared_error:
0.1827
Epoch 2/100
2300/2300 [==============================] - 4s 2ms/step - loss: 0.1214 -
mean_squared_error: 0.1214 - val_loss: 0.1522 - val_mean_squared_error:
0.1522
Epoch 3/100
2300/2300 [==============================] - 4s 2ms/step - loss: 0.0796 -
mean_squared_error: 0.0796 - val_loss: 0.1154 - val_mean_squared_error:
0.1154

...

Epoch 99/100
2300/2300 [==============================] - 4s 2ms/step - loss: 0.0139 -
mean_squared_error: 0.0139 - val_loss: 0.0247 - val_mean_squared_error:
0.0247
Epoch 100/100
2300/2300 [==============================] - 4s 2ms/step - loss: 0.0139 -
mean_squared_error: 0.0139 - val_loss: 0.0247 - val_mean_squared_error:
0.0247
```

This is an example whose purpose is only didactic; therefore, the final validation mean squared error is not extremely low. However, as it's possible to see in the following diagram (representing the predictions on the validation set), the model has successfully learned the global trend:



LSTM predictions on the Zuerich dataset

The model is still unable to achieve a very high accuracy in correspondence of all the very rapid spikes, but it's able to correctly model the amplitude of the oscillations and the length of the tails. For the sake of intellectual honesty, we must consider that this validation is performed on true data; however, when working with time series, it's normal to predict a new value using the ground truth. In this case, it's like a moving prediction where each value is obtained using the training history and a set of real observations. It's clear that the model is able to predict the long-term oscillations and also some local ones (for example, the sequence starting from step 300), but it can be improved in order to have better performance on the whole validation set. To achieve this goal, it is necessary to increase the network complexity and tune up the learning rate (it's a very interesting exercise on a real dataset).

Observing the previous diagram, it's possible to see that the model is relatively more accurate at some high frequencies (rapid changes), while it's more imprecise on others. This is not a strange behavior, because very oscillating functions *need more non-linearity* (think about the Taylor expansion and the relative error when it's truncated to a specific degree) to achieve high accuracies (this means employing more layers). My suggestion is that you repeat the experiment using more LSTM layers, considering that we need to pass the whole output sequence to the following recurrent layer (this can be achieved by setting the `return_sequences=True` parameter). The last layer, instead, must return only the final value (which is the default behavior). I also suggest testing the GRU layers, comparing the performance with the LSTM version and picking the simplest (benchmarking the training time) and most accurate solution.

The dataset can be freely downloaded in CSV format from `https://datamarket.com/data/set/22ti/zuerich-monthly-sunspot-numbers-1749-1983#!ds=22tidisplay=line`.

Transfer learning

We have discussed how deep learning is fundamentally based on gray-box models that learn how to associate input patterns to specific classification/regression outcomes. All the processing pipeline that is often employed to prepare the data for specific detections is absorbed by the complexity of the neural architecture. However, the price to pay for high accuracies is a proportionally large number of training samples. State-of-the-art visual networks are trained with millions of images and, obviously, each of them must be properly labeled. Even if there are many free datasets that can be employed to train several models, many specific scenarios need hard preparatory work that sometimes is very difficult to achieve.

Luckily, deep neural architectures are hierarchical models that learn in a structured way. As we have seen in the examples of deep convolutional networks, the first layers become more and more sensitive to detect low-level features, while the higher ones concentrate their work on extracting more detailed high-level features. In several tasks, it's reasonable to think that a network trained, for example, with a large visual dataset (such as ImageNet or Microsoft Coco) could be reused to achieve a specialization in a slightly different task. This concept is known as **transfer learning** and it's one of the most useful techniques when it's necessary to create state-of-the-art models with brand new datasets and specific objectives. For example, a customer can ask for a system to monitor a few cameras with the goal to segment the images and highlight the boundaries of specific targets.

The input is made up of video frames with the same geometric properties as thousands of images employed in training very powerful models (for example, Inception, ResNet, or VGG); therefore, we can take a pre-trained model, remove the highest layers (normally dense ones ending in a softmax classification layer) and connect the flattening layer to an MLP that outputs the coordinates of the bounding boxes. The first part of the network can be *frozen* (the weights are not modified anymore), while the SGD is applied to tune up the weights of the newly specialized sub-network.

Clearly, such an approach can dramatically speed up the training process, because the most complex part of the model is already trained and can also guarantee an extremely high accuracy (with respect to a naive solution), thanks to the optimization already performed on the original model. Obviously, the most natural question is how does this method work? Is there any formal proof? Unfortunately, there are no mathematical proofs, but there's enough evidence to assure about us of this approach. Generally speaking, the goal of a neural training process is to specialize each layer in order to provide a more particular (detailed, filtered, and so on) representation to the following one. Convolutional networks are a clear example of this behavior, but the same is observable in MLPs as well. The analysis of very deep convolutional networks showed how the content is still *visual* until reaching the flattening layer, where it's sent to a series of dense layers that are responsible for feeding the final softmax layer. In other words, the output of the convolutional block is a higher-level, segmented representation of the input, which is seldom affected by the specific classification problem. For this reason, transfer learning is generally sound and doesn't normally require a retraining of the lower layers. However, it's difficult to understand which model can yield the best performances and it's very useful to know which dataset has been used to train the original network. General purpose datasets (for example, ImageNet) are very useful in many contexts, while specific ones (such as Cifar-10 or Fashion; MNIST can be too restrictive). Luckily, Keras offers (in the package `keras.applications`) many models (even quite complex ones) that are always trained with ImageNet datasets and that can be immediately employed in a production-ready application. Even if using them is extremely simple, it requires a deeper knowledge of this framework, which is beyond the scope of this book. I invite the reader interested in this topic to check the book *Deep Learning with Keras, Gulli A., Pal S., Packt*.

Summary

In this chapter, we have presented the concept of a deep convolutional network, which is a generic architecture that can be employed in any visual processing task. The idea is based on hierarchical information management, aimed at extracting the features starting from low-level elements and moving forward until the high-level details that can be helpful to achieve specific goals.

The first topic was the concept of convolution and how it's applied in discrete and finite samples. We discussed the properties of standard convolution, before analyzing some important variants such as atrous (or dilated convolution), separable (and depthwise separable) convolution and, eventually, transpose convolution. All these methods can work with 1D, 2D, and 3D samples, even if the most diffused applications are based on bidimensional (not considering the channels) matrices representing static images. In the same section, we also discussed how pooling layers can be employed to reduce the dimensionality and improve the robustness to small translations.

In the next section, we introduced the concept of RNN, emphasizing the issues that normally arise when classic models are trained using the backpropagation through time algorithm. In particular, we explained why these networks cannot easily learn long-term dependencies. For this reason, new models have been proposed, whose performance was immediately outstanding. We discussed the most famous recurrent cell, called **Long-short-term memory** (**LSTM**), which can be used in layers that can easily learn all the most important dependencies of a sequence, allowing us to minimize the prediction error even in contexts with a very high variance (such as stock market quotations). The last topic was a simplified version of the idea implemented in LSTMs, which led to a model called a **Gated recurrent unit** (**GRU**). This cell is simpler and more computationally efficient, and many benchmarks confirmed that its performance is approximately the same as LSTM.

In the next chapter, `Chapter 11`, *Autoencoders* we are going to discuss some particular models called autoencoders, whose main property is to create internal representations of an arbitrarily complex input distribution.

11
Autoencoders

In this chapter, we are going to look at an unsupervised model family whose performance has been boosted by modern deep learning techniques. Autoencoders offer a different approach to classic problems such as dimensionality reduction or dictionary learning, but unlike many other algorithms, they don't suffer the capacity limitations that affect many famous models. Moreover, they can exploit specific neural layers (such as convolutions) to extract pieces of information based on specialized criteria. In this way, the internal representations can be more robust to different kinds of distortions and much more efficient in terms of the amount of information they can process.

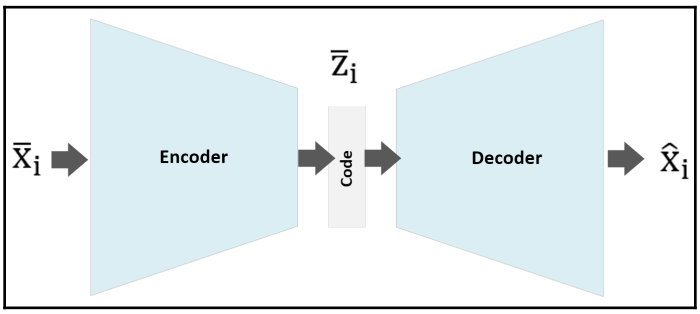
In particular, we are going to discuss the following:

- Standard autoencoders
- Denoising autoencoders
- Sparse autoencoders
- Variational autoencoders

Autoencoders

In the previous chapters, we discussed how real datasets are very often high-dimensional representations of samples that lie on low-dimensional manifolds (this is one of the semi-supervised pattern's assumptions, but it's generally true). As the complexity of a model is proportional to the dimensionality of the input data, many techniques have been analyzed and optimized in order to reduce the actual number of *valid components*. For example, PCA selects the features according to the relative explained variance, while ICA and generic dictionary learning techniques look for basic atoms that can be combined to rebuild the original samples. In this chapter, we are going to analyze a family of models based on a slightly different approach, but whose capabilities are dramatically increased by the employment of deep learning methods.

A generic **autoencoder** is a model that is split into two separate (but not completely autonomous) components called an **Encoder** and a **Decoder**. The task of the encoder is to transform an input sample into an encoded feature vector, while the task of the decoder is the opposite: rebuilding the original sample using the feature vector as input. The following diagram shows a schematic representation of a generic model:



Schema of a generic autoencoder

More formally, we can describe the encoder as a parametrized function:

$$\bar{z}_i = e\left(\bar{x}_i; \bar{\theta}_e\right) \quad where \;\; \bar{x}_i \in X$$

The output z_i is a vectorial code whose dimensionality is normally quite lower than the inputs. Analogously, the decoder is described as the following:

$$\hat{x}_i = d\left(\bar{z}_i; \bar{\theta}_d\right)$$

The goal of a standard algorithm is to minimize a cost function that is proportional to the reconstruction error. A classic method is based on the mean squared error (working on a dataset with *M* samples):

$$C\left(X; \bar{\theta}_e, \bar{\theta}_d\right) = \frac{1}{M}\sum_{i=1}^{M}\left\|\bar{x}_i - \hat{x}_i\right\|^2 = \frac{1}{M}\sum_{i=1}^{M}\left\|\bar{x}_i - d\left(e\left(\bar{x}_i; \bar{\theta}_e\right); \bar{\theta}_d\right)\right\|^2$$

This function depends only on the input samples (which are constant) and the parameter vectors; therefore, this is *de facto* an unsupervised method where we can control the internal structure and the constraints imposed on the z_i code. From a probabilistic viewpoint, if the input x_i samples are drawn from a $p(X)$ data-generating process, our goal is to find a $q(\bullet)$ parametric distribution that minimizes the Kullback–Leibler divergence with $p(X)$. Considering the previous definitions, we can define $q(\bullet)$ as follows:

$$q\left(d\left(e\left(\bar{x}_i;\bar{\theta}_e\right);\bar{\theta}_d\right)|\bar{x}_i\right)$$

Therefore, the Kullback–Leibler divergence becomes the following:

$$D_{KL}(p||q) = \sum_i p(\bar{x}_i)\ log\ \frac{p(\bar{x}_i)}{q\left(d\left(e\left(\bar{x}_i;\bar{\theta}_e\right);\bar{\theta}_d\right)|\bar{x}_i\right)} = -H(p) + H(p,q)$$

The first term represents the negative entropy of the original distribution, which is constant and isn't involved in the optimization process. The other term is the cross-entropy between the p and q. If we assume Gaussian distributions for p and q, the mean squared error is proportional to the cross-entropy (for optimization purposes, it's equivalent to it), and therefore this cost function is still valid under a probabilistic approach. Alternatively, it's possible to consider Bernoulli distributions for p and q, and the cross-entropy becomes the following:

$$H(p,q) = -\sum_i \bar{x}_i log\hat{x}_i + (1-\bar{x}_i)log(1-\hat{x}_i)$$

The main difference between the two approaches is that while a mean squared error can be applied to $x_i \in \Re^q$ (or multidimensional matrices), Bernoulli distributions need $x_i \in [0, 1]^q$ (formally, this condition should be $x_i \in \{0, 1\}^q$; however, the optimization can also be successfully performed when the values are not binary). The same constraint is necessary for the reconstructions; therefore, when using neural networks, the most common choice is to employ sigmoid layers.

An example of a deep convolutional autoencoder with TensorFlow

This example (like all the others in this and the following chapters) is based on TensorFlow (for information about the installation of TensorFlow, please refer to the information box at the end of the section), because this framework allows a greater flexibility that is sometimes much more problematic with Keras. We will approach this example pragmatically, and so we are not going to explore all the features because they are beyond the scope of this book; however, interested readers can refer to *Deep Learning with TensorFlow - Second Edition, Zaccone G., Karim R., Packt*.

In this example, we are going to create a deep convolutional autoencoder and train it using the Fashion MNIST dataset. The first step is loading the data (using the Keras helper function), normalizing, and in order to speed up the computation, limiting the training set to 1,000 samples:

```
import numpy as np

from keras.datasets import fashion_mnist

(X_train, _), (_, _) = fashion_mnist.load_data()

nb_samples = 1000
nb_epochs = 400
batch_size = 200
code_length = 256

X_train = X_train.astype(np.float32)[0:nb_samples] / 255.0

width = X_train.shape[1]
height = X_train.shape[2]
```

At this point, we can create the `Graph`, setting up the whole architecture, which is made up of the following:

- The encoder (all layers have padding "same" and ReLU activation):
    - Convolution with 32 filters, kernel size equal to (3 × 3), and strides (2 × 2)
    - Convolution with 64 filters, kernel size equal to (3 × 3), and strides (1× 1)
    - Convolution with 128 filters, kernel size equal to (3 × 3), and strides (1 × 1)

- The decoder:
    - Transpose convolution with 128 filters, kernel size equal to (3 × 3), and strides (2 × 2)
    - Transpose convolution with 64 filters, kernel size equal to (3 × 3), and strides (1× 1)
    - Transpose convolution with 32 filters, kernel size equal to (3 × 3), and strides (1 × 1)
    - Transpose convolution with 1 filter, kernel size equal to (3 × 3), strides (1 × 1), and sigmoid activation

As the images are (28 × 28), we prefer to resize each batch to the dimensions of (32 × 32) to easily manage all the subsequent operations that are based on sizes which are a power of 2:

```
import tensorflow as tf

graph = tf.Graph()

with graph.as_default():
    input_images = tf.placeholder(tf.float32, shape=(None, width, height,
1))
    r_input_images = tf.image.resize_images(input_images, (32, 32))
    # Encoder
    conv_0 = tf.layers.conv2d(inputs=r_input_images,
                              filters=32,
                              kernel_size=(3, 3),
                              strides=(2, 2),
                              activation=tf.nn.relu,
                              padding='same')
    conv_1 = tf.layers.conv2d(inputs=conv_0,
                              filters=64,
                              kernel_size=(3, 3),
                              activation=tf.nn.relu,
                              padding='same')
    conv_2 = tf.layers.conv2d(inputs=conv_1,
                              filters=128,
                              kernel_size=(3, 3),
                              activation=tf.nn.relu,
                              padding='same')
    # Code layer
    code_input = tf.layers.flatten(inputs=conv_2)
    code_layer = tf.layers.dense(inputs=code_input,
                                 units=code_length,
                                 activation=tf.nn.sigmoid)
    # Decoder
```

```
    decoder_input = tf.reshape(code_layer, (-1, 16, 16, 1))
    convt_0 = tf.layers.conv2d_transpose(inputs=decoder_input,
                                         filters=128,
                                         kernel_size=(3, 3),
                                         strides=(2, 2),
                                         activation=tf.nn.relu,
                                         padding='same')
    convt_1 = tf.layers.conv2d_transpose(inputs=convt_0,
                                         filters=64,
                                         kernel_size=(3, 3),
                                         activation=tf.nn.relu,
                                         padding='same')
    convt_2 = tf.layers.conv2d_transpose(inputs=convt_1,
                                         filters=32,
                                         kernel_size=(3, 3),
                                         activation=tf.nn.relu,
                                         padding='same')
    convt_3 = tf.layers.conv2d_transpose(inputs=convt_2,
                                         filters=1,
                                         kernel_size=(3, 3),
                                         activation=tf.sigmoid,
                                         padding='same')
    # Loss
    loss = tf.nn.l2_loss(convt_3 - r_input_images)
    # Training step
    training_step = tf.train.AdamOptimizer(0.001).minimize(loss)
```

The loss function is a standard L2 without any other constraint. I invite the reader to test different optimizers and learning rates to employ a solution that guarantees the minimum loss value. After defining the `Graph`, it's possible to set up an `InteractiveSession` (or a standard one), initialize all variables, and begin the training process:

```
import numpy as np
import tensorflow as tf

session = tf.InteractiveSession(graph=graph)
tf.global_variables_initializer().run()

for e in range(nb_epochs):
    np.random.shuffle(X_train)
    total_loss = 0.0
    for i in range(0, nb_samples - batch_size, batch_size):
        X = np.zeros((batch_size, width, height, 1), dtype=np.float32)
        X[:, :, :, 0] = X_train[i:i + batch_size, :, :]
        _, n_loss = session.run([training_step, loss],
                                feed_dict={
                                    input_images: X
```

```
                                    })
        total_loss += n_loss
    print('Epoch {}) Total loss: {}'.format(e + 1, total_loss))
```

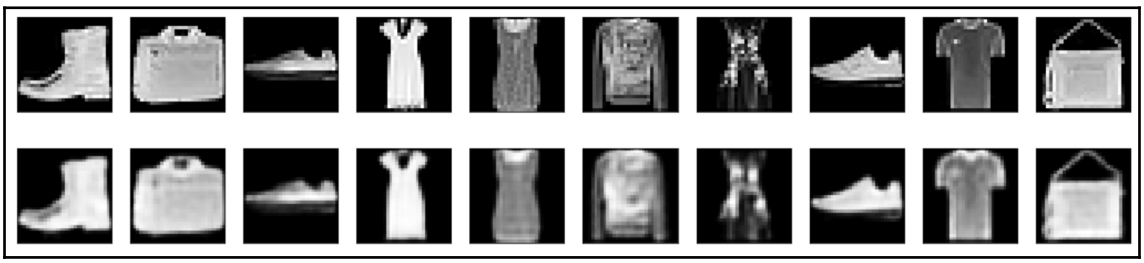
Once the training process is finished, we can check the average code length for the whole dataset (this information is useful to compare this result with the one achieved by imposing a sparsity constraint):

```
import numpy as np

codes = session.run([code_layer],
                    feed_dict={
                        input_images: np.expand_dims(X_train, axis=3),
                    })[0]

print(np.mean(codes))
0.5545144
```

This value is very small, indicating that the representations are already rather sparse; however, we are going to compare it with the mean obtained by a sparse autoencoder. We can now process a few images (10) by encoding and decoding them:

```
import numpy as np

Xs = np.reshape(X_train[0:10], (10, width, height, 1))

Ys = session.run([convt_3],
                 feed_dict={
                     input_images: Xs
                 })

Ys = np.squeeze(Ys[0] * 255.0)
```

The result is shown in the following figure:



Original images (upper row); decoded images (lower row)

As you can see, the reconstructions are rather lossy, but the autoencoder successfully learned how to reduce the dimensionality of the input samples. As an exercise, I invite the reader to split the code into two separate sections (encoder and decoder) and to optimize the architecture in order to achieve better accuracy on the whole Fashion MNIST dataset.

TensorFlow is available for Linux, Windows, and OS X with both CPU and CUDA GPU support. In many cases, it's possible to install it using the `pip install -U tensorflow` command; however, I suggest that you read the updated instructions for each platform at `https://www.tensorflow.org/install/`.

Denoising autoencoders

Autoencoders can be used to determine under-complete representations of a dataset; however, Bengio et al. (in P. Vincent, H. Larochelle, I. Lajoie, Y. Bengio, and P. Manzagol's book *Stacked Denoising Autoencoders: Learning Useful Representations in a Deep Network with a Local Denoising Criterion*, from the *Journal of Machine Learning Research 11/2010*) proposed to use them not to learn the exact representation of a sample in order to rebuild it from a low-dimensional code, but rather to denoise input samples. This is not a brand new idea, because, for example, Hopfield networks (proposed a few decades ago) had the same purpose, but its limitations in terms of capacity led researchers to look for different methods. Nowadays, deep autoencoders can easily manage high-dimensional data (such as images) with a consequent space requirement, that's why many people are now reconsidering the idea of teaching a network how to rebuild a sample image starting from a corrupted one.

Formally, there are not many differences between denoising autoencoders and standard autoencoders. However, in this case, the encoder must work with noisy samples:

$$\bar{z}_i = e\left(\bar{x}_i + \bar{n}_i; \bar{\theta}_e\right) \;\; where \;\; \bar{x}_i \in X$$

The decoder's cost function remains the same. If the noise is sampled for each batch, repeating the process for a sufficiently large number of iterations allows the autoencoder to learn how to rebuild the original image when some fragments are missing or corrupted. To achieve this goal, the authors suggested different possible kinds of noise. The most common choice is to sample Gaussian noise, which has some helpful features and is coherent with many real noisy processes:

$$\bar{z}_i = e\left(\bar{x}_i + \bar{n}_i(t); \bar{\theta}_e\right) \;\; where \;\; \bar{x}_i \in X \;\; and \;\; \bar{n}_i(t) \sim N(0, \sigma^2)$$

Another possibility is to employ an input dropout layer, zeroing some random elements:

$$\bar{z}_i = e\left(\bar{x}_i \circ \bar{n}_i; \bar{\theta}_e\right) \;\; where \;\; \bar{x}_i \in X \;\; and \;\; \bar{n}_i(x, y) \sim B(0, 1)$$

This choice is clearly more drastic, and the rate must be properly tuned. A very large number of dropped pixels can irreversibly delete many pieces of information and the reconstruction can become more difficult and *rigid* (our purpose is to extend the autoencoder's ability to other samples drawn from the same distribution). Alternatively, it's possible to mix up Gaussian noise and the dropout's, switching between them with a fixed probability. Clearly, the models must be more complex than standard autoencoders because now they have to cope with missing information; the same concept applies to the code length: very under-complete code wouldn't be able to provide all the elements needed to reconstruct the original image in the most accurate way. I suggest testing all the possibilities, in particular when the noise is constrained by external conditions (for example, old photos or messages transmitted through channels affected by precise noise processes). If the model must also be employed for never-before-seen samples, it's extremely important to select samples that represent the true distribution, using data augmentation techniques (limited to operations compatible with the specific problem) whenever the number of elements is not enough to reach the desired level of accuracy.

An example of a denoising autoencoder with TensorFlow

In this example (based on the previous one), we are going to employ a very similar architecture, but as the goal is denoising the images, we will impose a code length equal to (width × height), setting all the strides to (1 × 1), and therefore we won't need to resize the images anymore:

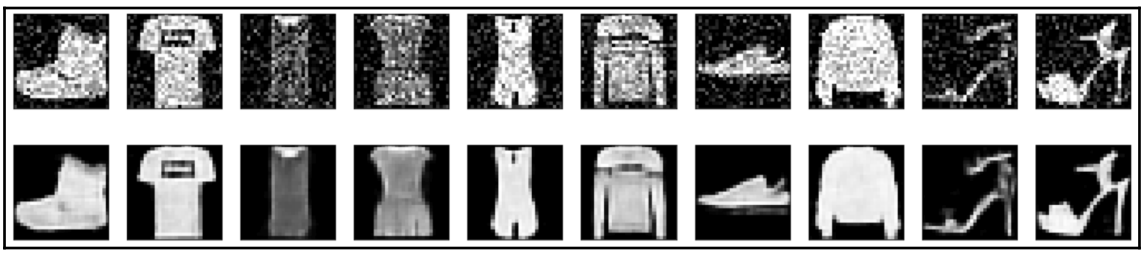
```
import tensorflow as tf

graph = tf.Graph()

with graph.as_default():
    input_noisy_images = tf.placeholder(tf.float32, shape=(None, width,
height, 1))
    input_images = tf.placeholder(tf.float32, shape=(None, width, height,
1))
    # Encoder
    conv_0 = tf.layers.conv2d(inputs=input_noisy_images,
                              filters=32,
                              kernel_size=(3, 3),
```

```
                            activation=tf.nn.relu,
                            padding='same')
conv_1 = tf.layers.conv2d(inputs=conv_0,
                            filters=64,
                            kernel_size=(3, 3),
                            activation=tf.nn.relu,
                            padding='same')
conv_2 = tf.layers.conv2d(inputs=conv_1,
                            filters=128,
                            kernel_size=(3, 3),
                            activation=tf.nn.relu,
                            padding='same')
# Code layer
code_input = tf.layers.flatten(inputs=conv_2)
code_layer = tf.layers.dense(inputs=code_input,
                               units=width * height,
                               activation=tf.nn.sigmoid)
# Decoder
decoder_input = tf.reshape(code_layer, (-1, width, height, 1))
convt_0 = tf.layers.conv2d_transpose(inputs=decoder_input,
                                     filters=128,
                                     kernel_size=(3, 3),
                                     activation=tf.nn.relu,
                                     padding='same')
convt_1 = tf.layers.conv2d_transpose(inputs=convt_0,
                                     filters=64,
                                     kernel_size=(3, 3),
                                     activation=tf.nn.relu,
                                     padding='same')
convt_2 = tf.layers.conv2d_transpose(inputs=convt_1,
                                     filters=32,
                                     kernel_size=(3, 3),
                                     activation=tf.nn.relu,
                                     padding='same')
convt_3 = tf.layers.conv2d_transpose(inputs=convt_2,
                                     filters=1,
                                     kernel_size=(3, 3),
                                     activation=tf.sigmoid,
                                     padding='same')
# Loss
loss = tf.nn.l2_loss(convt_3 - input_images)
# Training step
training_step = tf.train.AdamOptimizer(0.001).minimize(loss)
```

In this case, we need to pass both the noisy images (through the `placeholder input_noisy_images`) and the original ones (which are used to compute the final L2 loss function). For our example, we have decided to employ Gaussian noise with a standard deviation of σ = `0.2` (clipping the final values so that they are always constrained between 0 and 1):

```
import numpy as np
import tensorflow as tf

session = tf.InteractiveSession(graph=graph)
tf.global_variables_initializer().run()

for e in range(nb_epochs):
    total_loss = 0.0
    for i in range(0, nb_samples - batch_size, batch_size):
        X = np.zeros((batch_size, width, height, 1), dtype=np.float32)
        X[:, :, :, 0] = X_train[i:i + batch_size, :, :]
        Xn = np.clip(X + np.random.normal(0.0, 0.2, size=(batch_size,
width, height, 1)), 0.0, 1.0)
        _, n_loss = session.run([training_step, loss],
                                feed_dict={
                                    input_images: X,
                                    input_noisy_images: Xn
                                })
        total_loss += n_loss
    print('Epoch {}) Total loss: {}'.format(e + 1, total_loss))
```

The result after 200 epochs is shown in the following figure:



Noisy samples (upper row); denoised samples (lower row)

The denoising autoencoder has successfully learned to rebuild the original images in the presence of Gaussian noise. I invite the reader to test other methods (such as using an initial dropout) and increase the noise level to understand what the maximum corruption is that this model can effectively remove.

Sparse autoencoders

In general, standard autoencoders produce dense internal representations. This means that most of the values are different from zero. In some cases, however, it's more useful to have sparse codes that can better represent the atoms belonging to a dictionary. In this case, if z_i = (0, 0, z_i^n, ..., 0, z_i^m, ...), we can consider each sample as the overlap of specific atoms weighted accordingly. To achieve this objective, we can simply apply an L1 penalty to the code layer, as explained in `Chapter 1`, *Machine Learning Models Fundamentals*. The loss function for a single sample therefore becomes the following:

$$\hat{L}\left(\bar{x}_i; \bar{\theta}_e, \bar{\theta}_d\right) = L\left(\bar{x}_i; \bar{\theta}_e, \bar{\theta}_d\right) + \alpha \|\bar{z}_i\|_1$$

In this case, we need to consider the extra hyperparameter α, which must be tuned to increase the sparsity without a negative impact on the accuracy. As a general rule of thumb, I suggest starting with a value equal to 0.01 and reducing it until the desired result has been achieved. In most cases, higher values yield very poor performance, and therefore they are generally avoided.

A different approach has been proposed by Andrew Ng (in his book *Sparse Autoencoder, CS294A, Stanford University*). If we consider the code layer as a set of independent Bernoulli random variables, we can enforce sparsity by considering a generic reference Bernoulli variable with a very low mean (for example, p_r = 0.01) and adding the Kullback–Leibler divergence between the generic element $z_i^{(j)}$ and p_r to the cost function. For a single sample, the extra term is as follows (p is the code length):

$$L_{D_i} = \sum_{j=1}^{p} D_{KL}(\bar{z}_i^j || p_r) = \sum_{j=1}^{p} p_r \, log\frac{p_r}{\bar{z}_i^j} + (1 - p_r) \, log\frac{1 - p_r}{1 - \bar{z}_i^j}$$

The resulting loss function becomes the following:

$$\hat{L}\left(\bar{x}_i; \bar{\theta}_e, \bar{\theta}_d\right) = L\left(\bar{x}_i; \bar{\theta}_e, \bar{\theta}_d\right) + \alpha L_{D_i}$$

The effect of this penalty is similar to L1 (with the same considerations about the α hyperparameter), but many experiments have confirmed that the resulting cost function is easier to optimize, and it's possible to achieve the same level of sparsity that reaches higher reconstruction accuracies. When working with sparse autoencoders, the code length is often larger because of the assumption that a single element is made up of a small number of atoms (compared to the dictionary size). As a result, I suggest that you evaluate the level of sparsity with different code lengths and select the combination that maximizes the former and minimizes the latter.

Adding sparseness to the Fashion MNIST deep convolutional autoencoder

In this example, we are going to add an L1 regularization term to the cost function that was defined in the first exercise:

```
import tensorflow as tf

...

# Loss
sparsity_constraint = tf.reduce_sum(0.001 * tf.norm(code_layer, ord=1,
axis=1))
loss = tf.nn.l2_loss(convt_3 - r_input_images) + sparsity_constraint

...
```

The training process is exactly the same, and therefore we can directly show the final code mean after 200 epochs:

```
import numpy as np

codes = session.run([code_layer],
                    feed_dict={
                        input_images: np.expand_dims(X_train, axis=3),
                    })[0]

print(np.mean(codes))
0.45797634
```

As you can see, the mean is now lower, indicating that more code values are close to 0. I invite the reader to implement the other strategy, considering that it's easier to create a constant vector filled with small values (for example, 0.01) and exploit the vectorization properties offered by TensorFlow. I also suggest simplifying the Kullback–Leibler divergence by splitting it into an entropy term $H(p_r)$ (which is constant) and a cross-entropy $H(z, p_r)$ term.

Variational autoencoders

A **variational autoencoder** (**VAE**) is a generative model proposed by Kingma and Wellin (in their work *Auto-Encoding Variational Bayes, arXiv:1312.6114 [stat.ML]*) that partially resembles a standard autoencoder, but it has some fundamental internal differences. The goal, in fact, is not finding an encoded representation of a dataset, but determining the parameters of a generative process that is able to yield all possible outputs given an input data-generating process.

Let's take the example of a model based on a learnable parameter vector θ and a set of latent variables z that have a probability density function $p(z;\theta)$. Our goal can therefore be expressed as the research of the θ parameters that maximize the likelihood of the marginalized distribution $p(x;\theta)$ (obtained through the integration of the joint probability $p(x,z;\theta)$):

$$p\left(\bar{x};\bar{\theta}\right) = \int p\left(\bar{x},\bar{z};\bar{\theta}\right) dz = \int p\left(\bar{x}|\bar{z};\bar{\theta}\right) p\left(\bar{z};\bar{\theta}\right) dz$$

If this problem could be easily solved in closed form, a large set of samples drawn from the $p(x)$ data generating process would be enough to find a $p(x;\theta)$ good approximation. Unfortunately, the previous expression is intractable in the majority of cases because the true prior $p(z)$ is unknown (this is a secondary issue, as we can easily make some helpful assumptions) and the posterior distribution $p(x|z;\theta)$ is almost always close to zero. The first problem can be solved by selecting a simple prior (the most common choice is $z \sim N(0, I)$), but the second one is still very hard because only a few z values can lead to the generation of acceptable samples. This is particularly true when the dataset is very high dimensional and complex (for example, images). Even if there are millions of combinations, only a small number of them can yield realistic samples (if the images are photos of cars, we expect four wheels in the lower part, but it's still possible to generate samples where the wheels are on the top). For this reason, we need to exploit a method to reduce the sample space. Variational Bayesian methods (read *C.* Fox and S. Roberts's work *A Tutorial on Variational Bayesian Inference* from *Orchid* for further information) are based on the idea of employing *proxy* distributions, which are easy to sample and, in this case, whose density is very high (that is, the probability of generating a reasonable output is much higher than the true posterior).

In this case, we define an approximate posterior, considering the architecture of a standard autoencoder. In particular, we can introduce a $q(z|x;\theta_q)$ distribution that acts as an encoder (that doesn't behave determinastically anymore), which can be easily modeled with a neural network. Our goal, of course, is to find the best θ_q parameter set to maximize the similarity between q and the true posterior distribution $p(z|x;\theta)$. This result can be achieved by minimizing the Kullback–Leibler divergence:

$$D_{KL}\left(q(\bar{z}|\bar{x};\bar{\theta}_q)||p(\bar{z}|\bar{x};\bar{\theta})\right) = \sum_z q(\bar{z}|\bar{x};\bar{\theta}_q) log \frac{q(\bar{z}|\bar{x};\bar{\theta}_q)}{p(\bar{z}|\bar{x};\bar{\theta})} =$$

$$= E_z\left[log\, q(\bar{z}|\bar{x};\bar{\theta}_q)\right] - E_z\left[log\, p(\bar{z}|\bar{x};\bar{\theta})\right] =$$

$$= E_z\left[log\, q(\bar{z}|\bar{x};\bar{\theta}_q)\right] - E_z\left[log\, p(\bar{x}|\bar{z};\bar{\theta}) - log\, p(\bar{z};\bar{\theta}) + log\, p(\bar{x};\bar{\theta})\right]$$

In the last formula, the term *log p(x;θ)* doesn't depend on *z*, and therefore it can be extracted from the expected value operator and the expression can be manipulated to simplify it:

$$\begin{aligned} &log\, p(\bar{x};\bar{\theta}) - D_{KL}\left(q(\bar{z}|\bar{x};\bar{\theta}_q)||p(\bar{z}|\bar{x};\bar{\theta})\right) = \\ &= E_z\left[log\, q(\bar{z}|\bar{x};\bar{\theta}_q) - log\, p(\bar{x}|\bar{z};\bar{\theta}) - log\, p(\bar{z};\bar{\theta})\right] = \\ &= E_z\left[log\, p(\bar{x}|\bar{z};\bar{\theta})\right] - D_{KL}\left(q(\bar{z}|\bar{x};\bar{\theta}_q)||p(\bar{z};\bar{\theta})\right) \end{aligned}$$

The equation can be also rewritten as the following:

$$\begin{aligned} log\, p(\bar{x};\bar{\theta}) = E_z\left[log\, p(\bar{x}|\bar{z};\bar{\theta})\right] - D_{KL}\left(q(\bar{z}|\bar{x};\bar{\theta}_q)||p(\bar{z};\bar{\theta})\right) + D_{KL}\left(q(\bar{z}|\bar{x};\bar{\theta}_q)||p(\bar{z}|\bar{x};\bar{\theta})\right) = \\ = ELBO_{\bar{\theta}} + D_{KL}\left(q(\bar{z}|\bar{x};\bar{\theta}_q)||p(\bar{z}|\bar{x};\bar{\theta})\right) \end{aligned}$$

On the right-hand side, we now have the term **ELBO** (short for **evidence lower bound**) and the Kullback–Leibler divergence between the probabilistic encoder $q(z|x;\theta_q)$ and the true posterior distribution $p(z|x;\theta)$. As we want to maximize the log-probability of a sample under the θ parametrization, and considering that the KL divergence is always non-negative, we can only work with the ELBO (which is a lot easier to manage than the other term). Indeed, the loss function that we are going to optimize is the negative ELBO. To achieve this goal, we need two more important steps.

The first one is choosing an appropriate structure for $q(z|x;\theta_q)$. As $p(z;\theta)$ is assumed to be normal, we can supposedly model $q(z|x;\theta_q)$ as a multivariate Gaussian distribution, splitting the probabilistic encoder into two blocks fed with the same lower layers:

- A mean $\mu(z|x;\theta_q)$ generator that outputs a $\mu_i \in \Re^p$ vector
- A $\Sigma(z|x;\theta_q)$ covariance generator (assuming a diagonal matrix) that outputs a $\sigma_i \in \Re^p$ vector so that Σ_i=diag(σ_i)

In this way, $q(z|x;\theta_q) = N(\mu(z|x;\theta_q), \Sigma(z|x;\theta_q))$, and therefore the second term on the right-hand side is the Kullback-Leibler divergence between two Gaussian distributions that can be easily expressed as follows (p is the dimension of both the mean and covariance vector):

$$D_{KL}\left(N\left(\mu(\bar{z}|\bar{x};\bar{\theta}_q),\Sigma(\bar{z}|\bar{x};\bar{\theta}_q)\right)||N(0,I)\right) = \frac{1}{2}\left(tr\left(\Sigma(\bar{z}|\bar{x};\bar{\theta}_q)\right) + \mu(\bar{z}|\bar{x};\bar{\theta}_q)^T\mu(\bar{z}|\bar{x};\bar{\theta}_q) - log\left|\Sigma(\bar{z}|\bar{x};\bar{\theta}_q)\right| - p\right)$$

This operation is simpler than expected because, as Σ is diagonal, the trace corresponds to the sum of the elements $\Sigma_1 + \Sigma_2 + [...] + \Sigma_p$ and $\log(|\Sigma|) = \log(\Sigma_1\Sigma_2...\Sigma_p) = \log \Sigma_1 + \log \Sigma_2 + ... + \log \Sigma_p$.

At this point, maximizing the right-hand side of the previous expression is equivalent to maximizing the expected value of the log probability to generate acceptable samples and minimizing the discrepancy between the normal prior and the Gaussian distribution synthesized by the encoder. Everything seems much simpler now, but there is still a problem to solve. We want to use neural networks and the stochastic gradient descent algorithm, and therefore we need differentiable functions. As the Kullback-Leibler divergence can be computed only using minibatches with n elements (the approximation becomes close to the true value after a sufficient number of iterations), it's necessary to sample n values from the distribution $N(\mu(z|x;\theta_q), \Sigma(z|x;\theta_q))$ and, unfortunately, this operation is not differentiable. To solve this problem, the authors suggested a reparameterization trick: instead of sampling from $q(z|x;\theta_q)$, we can sample from a normal distribution, $\varepsilon \sim N(0, I)$, and build the actual samples as $\mu(z|x;\theta_q) + \varepsilon \cdot \Sigma(z|x;\theta_q)^2$. Considering that ε is a constant vector during a batch (both the forward and backward phases), it's easy to compute the gradient with respect to the previous expression and optimize both the decoder and the encoder.

The last element to consider is the first term on the right-hand side of the expression that we want to maximize:

$$E_z\left[log\ p(\bar{x}|\bar{z};\bar{\theta})\right] = \sum_z p(\bar{z}|\bar{x};\bar{\theta})\ log\ p(\bar{x}|\bar{z};\bar{\theta}) = -H\left(p(\bar{z}|\bar{x};\bar{\theta}), p(\bar{x}|\bar{z};\bar{\theta})\right)$$

This term represents the negative cross-entropy between the actual distribution and the reconstructed one. As discussed in the first section, there are two feasible choices: Gaussian or Bernoulli distributions. In general, variational autoencoders employ a Bernoulli distribution with input samples and reconstruction values constrained between 0 and 1. However, many experiments have confirmed that the mean squared error can speed up the training process, and therefore I suggest that the reader test both methods and pick the one that guarantees the best performance (both in terms of accuracy and training speed).

An example of a variational autoencoder with TensorFlow

Let's continue working with the Fashion MNIST dataset to build a variational autoencoder. As explained, the output of the encoder is now split into two components: the mean and covariance vectors (both with dimensions equal to *(width · height)*) and the decoder input is obtained by sampling from a normal distribution and projecting the code components. The complete `Graph` is as follows:

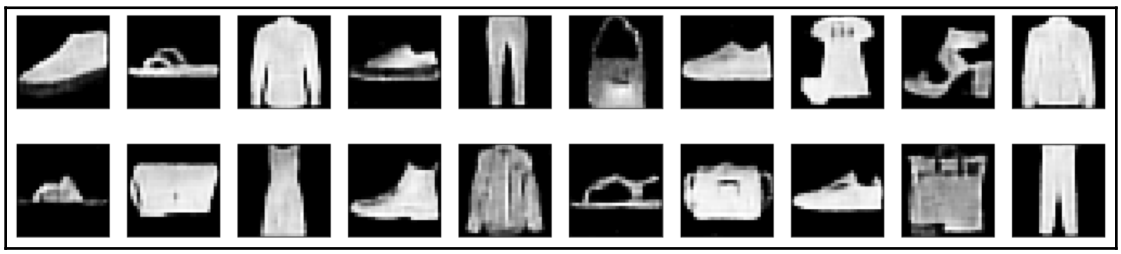
```
import tensorflow as tf

graph = tf.Graph()

with graph.as_default():
    input_images = tf.placeholder(tf.float32, shape=(batch_size, width,
height, 1))
    # Encoder
    conv_0 = tf.layers.conv2d(inputs=input_images,
                              filters=32,
                              kernel_size=(3, 3),
                              strides=(2, 2),
                              activation=tf.nn.relu,
                              padding='same')
    conv_1 = tf.layers.conv2d(inputs=conv_0,
                              filters=64,
                              kernel_size=(3, 3),
                              strides=(2, 2),
                              activation=tf.nn.relu,
                              padding='same')
    conv_2 = tf.layers.conv2d(inputs=conv_1,
                              filters=128,
                              kernel_size=(3, 3),
                              activation=tf.nn.relu,
                              padding='same')
    # Code layer
    code_input = tf.layers.flatten(inputs=conv_2)
    code_mean = tf.layers.dense(inputs=code_input,
                                units=width * height)
    code_log_variance = tf.layers.dense(inputs=code_input,
                                        units=width * height)
    code_std = tf.sqrt(tf.exp(code_log_variance))
    # Normal samples
    normal_samples = tf.random_normal(mean=0.0, stddev=1.0,
shape=(batch_size, width * height))
    # Sampled code
```

```
    sampled_code = (normal_samples * code_std) + code_mean
    # Decoder
    decoder_input = tf.reshape(sampled_code, (-1, 7, 7, 16))
    convt_0 = tf.layers.conv2d_transpose(inputs=decoder_input,
                                         filters=64,
                                         kernel_size=(3, 3),
                                         strides=(2, 2),
                                         activation=tf.nn.relu,
                                         padding='same')
    convt_1 = tf.layers.conv2d_transpose(inputs=convt_0,
                                        filters=32,
                                        kernel_size=(3, 3),
                                        strides=(2, 2),
                                        activation=tf.nn.relu,
                                        padding='same')
    convt_2 = tf.layers.conv2d_transpose(inputs=convt_1,
                                        filters=1,
                                        kernel_size=(3, 3),
                                        padding='same')
    convt_output = tf.nn.sigmoid(convt_2)
    # Loss
    reconstruction =
tf.nn.sigmoid_cross_entropy_with_logits(logits=convt_2,
labels=input_images)
    kl_divergence = 0.5 * tf.reduce_sum(tf.square(code_mean) +
tf.square(code_std) - tf.log(1e-8 + tf.square(code_std)) - 1, axis=1)
    loss = tf.reduce_sum(reconstruction) + kl_divergence
    # Training step
    training_step = tf.train.AdamOptimizer(0.001).minimize(loss)
```

As you can see, the only differences are as follows:

- The generation of the encoder input is `(normal_samples * code_std) + code_mean`
- The use of sigmoid cross-entropy as reconstruction loss
- The presence of the Kullback-Leibler divergence as a regularization term

The training process is identical to the first example in this chapter, as the sampling operations are performed directly by TensorFlow. The result after 200 epochs is shown in the following figure:



Variational autoencoder output

As an exercise, I invite the reader to use RGB datasets (such as Cifar-10, which is found at `https://www.cs.toronto.edu/~kriz/cifar.html`) to test the generation ability of the VAE by comparing the output samples with the one drawn from the original distribution.

In these kinds of experiments, where the random numbers are generated by both NumPy and TensorFlow, the random seeds are always set equal to 1,000 (`np.random.seed(1000)` and `tf.set_random_seed(1000)`). Other values or subsequent tests without resetting the seeds can yield slightly different results.

Summary

In this chapter, we presented autoencoders as unsupervised models that can learn to represent high-dimensional datasets with lower-dimensional codes. They are structured into two separate blocks (which, however, are trained together): an encoder, responsible for mapping the input sample to an internal representation, and a decoder, which must perform the inverse operation, rebuilding the original image starting from the code.

We have also discussed how autoencoders can be used to denoise samples and how it's possible to impose a sparsity constraint on the code layer to resemble the concept of standard dictionary learning. The last topic was about a slightly different pattern called a variational autoencoder. The idea is to build a generative model that is able to reproduce all the possible samples belonging to a training distribution.

In the next chapter, we are going to briefly introduce a very important model family called **generative adversarial networks** (**GANs**), which are not very different from the purposes of a variational autoencoder, but which have a much more flexible approach.

12
Generative Adversarial Networks

In this chapter, we are going to provide a brief introduction to a family of generative models based on some game theory concepts. Their main peculiarity is an adversarial training procedure that is aimed at learning to distinguish between true and fake samples, driving, at the same time, another component that generates samples more and more similar to the training examples.

In particular, we will be discussing:

- Adversarial training and standard **Generative Adversarial Networks** (**GANs**)
- **Deep Convolutional GANs** (**DCGAN**)
- **Wasserstein GANs** (**WGAN**)

Adversarial training

The brilliant idea of adversarial training, proposed by Goodfellow and others (in *Generative Adversarial Networks*, *Goodfellow I. J.*, *Pouget-Abadie J.*, *Mirza M.*, *Xu B.*, *Warde-Farley D.*, *Ozair S.*, *Courville A.*, *Bengio Y.*, *arXiv:1406.2661* [*stat.ML*]), ushered in a new generation of generative models that immediately outperformed the majority of existing algorithms. All of the derived models are based on the same fundamental concept of adversarial training, which is an approach partially inspired by game theory.

Let's suppose that we have a data generating process, $p_{data}(x)$, that represents an actual data distribution and a finite number of samples that we suppose are drawn from p_{data}:

$$X = \{\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_M\} \;\; where \;\; \bar{x}_i \in \mathbb{R}^n$$

Our goal is to train a model called a generator, whose distribution must be as close as possible to p_{data}. This is the trickiest part of the algorithm, because instead of standard methods (for example, variational autoencoders), adversarial training is based on a minimax game between two players (we can simply say that, given an objective, the goal of both players is to minimize the maximum possible loss; but in this case, each of them works on different parameters). One player is the generator, we can define as a parameterized function of a noise sample:

$$\hat{x}_i = G\left(\bar{z}_i; \bar{\theta}_g\right) \;\; where \;\; \bar{z}_i \sim U(-1, 1)$$

The generator is fed with a noise vector (in this case, we have employed a uniform distribution, but there are no particular restrictions; therefore, we are simply going to say that z is drawn from a noise distribution p_{noise}), and outputs a value that has the same dimensionality of the samples drawn from p_{data}. Without any further control, the generator distribution will be completely different from the data generating process, but this is the moment for the other player to enter the scene. The second model is called the *discriminator* (or Critic), and it has the responsibility of evaluating the samples drawn from p_{data} and the ones produced by the generator:

$$p_i = D\left(\bar{x}_i; \bar{\theta}_d\right) \;\; where \;\; p_i \in [0, 1]$$

The role of this model is to output a probability that must reflect the fact that the sample is drawn from p_{data}, instead of being generated by $G(z; \theta_g)$. What happens is very simple: the first player (the generator) outputs a sample, x. If x actually belongs to p_{data}, the discriminator will output a value close to 1, while if it's very different from the other true samples, $D(x; \theta_d)$ will output a very low probability. The real structure of the game is based on the idea of training the generator to deceive the discriminator, by producing samples that can potentially be drawn from p_{data}. This result can be achieved by trying to maximize the log-probability, $log(D(x; \theta_d))$, when x is a true sample (drawn from p_{data}), while minimizing the log-probability, $log(1 - D(G(z; \theta_g); \theta_d))$, with z sampled from a noise distribution.

The first operation forces the discriminator to become more and more aware of the true samples (this condition is necessary to avoid being deceived too easily). The second objective is a little bit more complex, because the discriminator has to evaluate a sample that can be acceptable or not. Let's suppose that the generator is not smart enough, and outputs a sample that cannot belong to p_{data}. As the discriminator is learning how p_{data} is structured, it will very soon distinguish the wrong sample, outputting a low probability. Hence, by minimizing $log(1 - D(G(z; \theta_g); \theta_d))$, we are forcing the discriminator to become more and more critical when the samples are quite different from the ones drawn from p_{data}, and the becomes generator more and more able to produce acceptable samples. On the other hand, if the generator outputs a sample that belongs to the data generating process, the discriminator will output a high probability, and the minimization falls back in the previous case.

The authors expressed this minimax game using a shared value function, *V(G, D)*, that must be minimized by the generator and maximized by the discriminator:

$$V(G,D) = E_{\bar{x} \sim p_{data}} \left[log \left(D \left(\bar{x}; \bar{\theta}_d \right) \right) \right] + E_{\bar{z} \sim p_{noise}} \left[log \left(1 - D \left(G \left(\bar{z}; \bar{\theta}_g \right); \bar{\theta}_d \right) \right) \right] = V_{data}(D) + V_{noise}(G,D)$$

This formula represents the dynamics of a non-cooperative game between two players (for further information, refer to *Tadelis S., Game Theory, Princeton University Press*) that theoretically admits a special configuration, called a *Nash equilibrium*, that can be described by saying that if the two players know each other's strategy, they have no reason to change their own strategy if the other player doesn't. In this case, both the discriminator and generator will pursue their strategies until no change is needed, reaching a final, stable configuration, which is potentially a Nash equilibrium (even if there are many factors that can prevent reaching this goal). A common problem is the premature convergence of the discriminator, which forces the gradients to vanish because the loss function becomes flat in a region close to 0. As this is a game, a fundamental condition is the possibility to provide information to allow the player to make corrections. If the discriminator learns how to separate true samples from fake ones too quickly, the generator convergence slows down, and the player can remain trapped in a sub-optimal configuration. In general, when the distributions are rather complex, the discriminator is slower than the generator; but, in some cases, it is necessary to update the generator more times after each single discriminator update. Unfortunately, there are no rules of thumb; but, for example, when working with images, it's possible to observe the samples generated after a sufficiently large number of iterations. If the discriminator loss has become very small and the samples appear corrupted or incoherent, it means that the generator did not have enough time to learn the distribution, and it's necessary to slow down the discriminator.

The authors in the aforementioned paper showed that, given a generator characterized by a distribution $p_g(x)$, the optimal discriminator is:

$$D_{opt}^{G}(\bar{x}) = \frac{p_{data}(\bar{x})}{p_{data}(\bar{x}) + p_g(\bar{x})}$$

At this point, considering the previous value function *V(G, D)* and using the optimal discriminator, we can rewrite it in a single objective (as a function of *G*) that must be minimized by the generator:

$$V'(G) = E_{\bar{x} \sim p_{data}} \left[log \left(D_{opt}^{G} \left(\bar{x}; \bar{\theta}_d \right) \right) \right] + E_{\bar{x} \sim p_g} \left[log \left(1 - D_{opt}^{G} \left(\bar{x}; \bar{\theta}_d \right) \right) \right]$$

To better understand how a **GAN** works, we need to expand the previous expression:

$$V'(G) = E_{\bar{x} \sim p_{data}} \left[log \left(\frac{p_{data}(\bar{x})}{p_{data}(\bar{x}) + p_g(\bar{x})} \right) \right] + E_{\bar{x} \sim p_g} \left[log \left(\frac{p_g(\bar{x})}{p_{data}(\bar{x}) + p_g(\bar{x})} \right) \right]$$

Applying some simple manipulations, we get the following:

$$\frac{1}{2} V'(G) + 2log(2) = \frac{1}{2} E_{\bar{x} \sim p_{data}} \left[log \left(\frac{p_{data}(\bar{x})}{p_{data}(\bar{x}) + p_g(\bar{x})} \right) \right] + \frac{1}{2} E_{\bar{x} \sim p_g} \left[log \left(\frac{p_g(\bar{x})}{p_{data}(\bar{x}) + p_g(\bar{x})} \right) \right] + \frac{1}{2} 2log(2) =$$

$$= \frac{1}{2} D_{KL} \left(p_{data} || \frac{p_{data} + p_g}{2} \right) + \frac{1}{2} D_{KL} \left(p_g || \frac{p_{data} + p_g}{2} \right) = D_{JS}(p_{data} || p_g)$$

The last term represents the Jensen-Shannon divergence between p_{data} and p_g. This measure is similar to the Kullback-Leibler divergence, but it's symmetric and bounded between *0* and *log(2)*. When the two distributions are identical, $D_{JS} = 0$, but if their supports (the value sets where $p(x) > 0$) are disjoint, $D_{JS} = log(2)$ (while $D_{KL} = \infty$). Therefore, the value function can be expressed as:

$$V'(G) = 2D_{JS}(p_{data} || p_g) - 2log(2)$$

Now, it should be clearer that a GAN tries to minimize the Jensen-Shannon divergence between the data generating process and the generator distribution. In general, this procedure is quite effective; however, when the supports are disjointed, a GAN has no pieces of information about the true distance. This consideration (analyzed with more mathematical rigor in *Improved Techniques for Training GANs*, *Salimans T.*, *Goodfellow I.*, *Zaremba W.*, *Cheung V.*, *Radford A.*, and *Chen X.*, *arXiv:1606.03498 [cs.LG]*) explains why training a GAN can become quite difficult, and, consequently, why the Nash equilibrium cannot be found in many cases. For these reasons, we are going to analyze an alternative approach in the next section.

The complete GAN algorithm (as proposed by the authors) is:

1. Set the number of epochs, N_{epochs}
2. Set the number of discriminator iterations, N_{iter} (in most cases, $N_{iter} = 1$)
3. Set the batch size, k
4. Define a noise generating process, M (for example, $U(-1, 1)$)
5. For $e=1$ to N_{epochs}:
    1. Sample k values from X
    2. Sample k values from N
    3. For $i=1$ to N_{iter}:
        1. Compute the gradients, $\nabla_d V(G, D)$ (only with respect to the discriminator variables). The expected value is approximated with a sample mean.
        2. Update the discriminator parameters by *Stochastic Gradient Ascent* (as we are working with logarithms, it's possible to minimize the negative loss).
    4. Sample k values from N
    5. Compute the gradients, $\nabla_g V_{noise}(G, D)$ (only with respect to the generator variables)
    6. Update the generator parameters by Stochastic Gradient Descent

As these models need to sample noisy vectors in order to guarantee the reproducibility, I suggest setting the random seed in both NumPy (`np.random.seed(...)`) and TensorFlow (`tf.set_random_seed(...)`). The default value chosen for all of these experiments is 1,000.

Example of DCGAN with TensorFlow

In this example, we want to build a DCGAN (proposed in *Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks, Radford A., Metz L., Chintala S., , arXiv:1511.06434 [cs.LG]*) with the Fashion-MNIST dataset (obtained through the `keras` helper function). As the training speed is not very high, we limit the number of samples to 5,000, but I suggest repeating the experiment with larger values. The first step is loading and normalizing (between -1 and 1) the dataset:

```
import numpy as np

from keras.datasets import fashion_mnist

nb_samples = 5000

(X_train, _), (_, _) = fashion_mnist.load_data()
X_train = X_train.astype(np.float32)[0:nb_samples] / 255.0
X_train = (2.0 * X_train) - 1.0

width = X_train.shape[1]
height = X_train.shape[2]
```

According to the original paper, the generator is based on four transpose convolutions with kernel sizes equal to *(4, 4)* and strides equal to *(2, 2)*. The input is a single multi-channel pixel (*1 × 1 × code_length*) that is expanded by subsequent convolutions. The number of filters is 1024, 512, 256, 128, and 1 (we are working with grayscale images). The authors suggest employing a symmetric-valued dataset (that's why we have normalized between -1 and 1), batch normalization after each layer, and leaky ReLU activation (with a default negative slope set to 0.2):

```
import tensorflow as tf

def generator(z, is_training=True):
    with tf.variable_scope('generator'):
        conv_0 = tf.layers.conv2d_transpose(inputs=z,
                                            filters=1024,
                                            kernel_size=(4, 4),
                                            padding='valid')

        b_conv_0 = tf.layers.batch_normalization(inputs=conv_0,
training=is_training)

        conv_1 =
tf.layers.conv2d_transpose(inputs=tf.nn.leaky_relu(b_conv_0),
                                            filters=512,
```

```
                                                kernel_size=(4, 4),
                                                strides=(2, 2),
                                                padding='same')

        b_conv_1 = tf.layers.batch_normalization(inputs=conv_1,
training=is_training)
        conv_2 =
tf.layers.conv2d_transpose(inputs=tf.nn.leaky_relu(b_conv_1),
                                                filters=256,
                                                kernel_size=(4, 4),
                                                strides=(2, 2),
                                                padding='same')

        b_conv_2 = tf.layers.batch_normalization(inputs=conv_2,
training=is_training)
        conv_3 =
tf.layers.conv2d_transpose(inputs=tf.nn.leaky_relu(b_conv_2),
                                                filters=128,
                                                kernel_size=(4, 4),
                                                strides=(2, 2),
                                                padding='same')

        b_conv_3 = tf.layers.batch_normalization(inputs=conv_3,
training=is_training)

        conv_4 =
tf.layers.conv2d_transpose(inputs=tf.nn.leaky_relu(b_conv_3),
                                                filters=1,
                                                kernel_size=(4, 4),
                                                strides=(2, 2),
                                                padding='same')

        return tf.nn.tanh(conv_4)
```

The strides are set to work with 64 × 64 images (unfortunately, the Fashion-MNIST dataset has 28 × 28 samples, which cannot be generated with power-of-two modules); therefore, we are going to resize the samples while training. As we need to compute the gradients of the discriminator and generator separately, it's necessary to set the variable scope (using the context manager `tf.variable_scope()`) to immediately extract only the variables whose names have the scope as a prefix (for example, `generator/Conv_1_1/...`).
The `is_training` parameter is necessary to disable the batch normalization during the generation phase.

The discriminator is almost the same as a generator (the only main differences are the inverse convolution sequence and the absence of batch normalization after the first layer):

```
import tensorflow as tf

def discriminator(x, is_training=True, reuse_variables=True):
    with tf.variable_scope('discriminator', reuse=reuse_variables):
        conv_0 = tf.layers.conv2d(inputs=x,
                                  filters=128,
                                  kernel_size=(4, 4),
                                  strides=(2, 2),
                                  padding='same')

        conv_1 = tf.layers.conv2d(inputs=tf.nn.leaky_relu(conv_0),
                                  filters=256,
                                  kernel_size=(4, 4),
                                  strides=(2, 2),
                                  padding='same')
        b_conv_1 = tf.layers.batch_normalization(inputs=conv_1,
training=is_training)
        conv_2 = tf.layers.conv2d(inputs=tf.nn.leaky_relu(b_conv_1),
                                  filters=512,
                                  kernel_size=(4, 4),
                                  strides=(2, 2),
                                  padding='same')
        b_conv_2 = tf.layers.batch_normalization(inputs=conv_2,
training=is_training)
        conv_3 = tf.layers.conv2d(inputs=tf.nn.leaky_relu(b_conv_2),
                                  filters=1024,
                                  kernel_size=(4, 4),
                                  strides=(2, 2),
                                  padding='same')
        b_conv_3 = tf.layers.batch_normalization(inputs=conv_3,
training=is_training)
        conv_4 = tf.layers.conv2d(inputs=tf.nn.leaky_relu(b_conv_3),
                                  filters=1,
                                  kernel_size=(4, 4),
                                  padding='valid')
        return conv_4
```

In this case, we have an extra parameter (`reuse_variables`) that is necessary when building the loss functions. In fact, we need to declare two discriminators (fed with real samples and with the generator output), but they are not made up of separate layers; hence, the second one must reuse the variables defined by the first one. We can now create a graph and define all of the placeholders and operations:

```
import tensorflow as tf

code_length = 100

graph = tf.Graph()

with graph.as_default():
    input_x = tf.placeholder(tf.float32, shape=(None, width, height, 1))
    input_z = tf.placeholder(tf.float32, shape=(None, code_length))
    is_training = tf.placeholder(tf.bool)
    gen = generator(z=tf.reshape(input_z, (-1, 1, 1, code_length)),
is_training=is_training)
    r_input_x = tf.image.resize_images(images=input_x, size=(64, 64))
    discr_1_l = discriminator(x=r_input_x, is_training=is_training,
reuse_variables=False)
    discr_2_l = discriminator(x=gen, is_training=is_training,
reuse_variables=True)
    loss_d_1 =
tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(labels=tf.ones_like(
discr_1_l), logits=discr_1_l))
    loss_d_2 =
tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(labels=tf.zeros_like
(discr_2_l), logits=discr_2_l))
    loss_d = loss_d_1 + loss_d_2
    loss_g =
tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(labels=tf.ones_like(
discr_2_l), logits=discr_2_l))
    variables_g = [variable for variable in tf.trainable_variables() if
variable.name.startswith('generator')]
    variables_d = [variable for variable in tf.trainable_variables() if
variable.name.startswith('discriminator')]
    with
tf.control_dependencies(tf.get_collection(tf.GraphKeys.UPDATE_OPS)):
        training_step_d = tf.train.AdamOptimizer(0.0002,
beta1=0.5).minimize(loss=loss_d, var_list=variables_d)
        training_step_g = tf.train.AdamOptimizer(0.0002,
beta1=0.5).minimize(loss=loss_g, var_list=variables_g)
```

The first step is defining the placeholders:

- `input_x` contains the true samples drawn from *X*
- `input_z` contains the noise samples
- `is_training` is a Boolean indicating whether or not the batch normalization must be active

Then, we define the generator instance after reshaping the noise sample as a *(1 × 1 × code_length)* matrix (this is necessary to work efficiently with transpose convolutions). As this is a fundamental hyperparameter, I suggest testing different values and comparing the final performances.

As explained previously, the input images are resized before defining the two discriminators (the second one reuses the variables previously defined).
The `discr_1_l` instance is fed with the true samples, while `discr_2_l` works with the generator output.

The next step is defining the loss functions. As we are working with logarithms, there can be stability problems when the values become close to 0. For this reason, it's preferable to employ the TensorFlow built-in function `tf.nn.sigmoid_cross_entropy_with_logits()`, which guarantees numerical stability in every case. This function takes a *logit* as input and applies the sigmoid transformation internally. In general, the output is:

$$-x_{label}\ log(\sigma(x_{logit})) - (1 - x_{label})\ log(1 - \sigma(x_{logit}))$$

Therefore, setting the label equal to 1 forces the second term to be null, and vice versa. At this point, we need to create two lists containing the variables belonging to each scope (this can be easily achieved by using the `tf.trainable_variables()` function, which outputs a list of all variables). The last step consists of defining the optimizers. As suggested in the official TensorFlow documentation, when working with batch normalizations, it's necessary to wrap the training operations in a context manager that checks whether all dependencies (in this case, batch average and variance) have been computed. We have employed the Adam optimizer with $\eta = 0.0002$, and a gradient momentum forgetting factor ($\mu 1$) equal to 0.5 (this is a choice motivated by the potential instability that a high momentum can yield). As it's possible to see, in both cases, the minimization is limited to a specific subset of the variables (providing a list through the `var_list` parameter).

At this point, we can create a `Session` (we are going to use an `InteractiveSession`), initialize all variables, and start the training procedure (with 200 epochs and a batch size equal to 128):

```
import numpy as np
import tensorflow as tf

nb_epochs = 200
batch_size = 128
nb_iterations = int(nb_samples / batch_size)

session = tf.InteractiveSession(graph=graph)
tf.global_variables_initializer().run()

samples_range = np.arange(nb_samples)

for e in range(nb_epochs * 5):
    d_losses = []
    g_losses = []
    for i in range(nb_iterations):
        Xi = np.random.choice(samples_range, size=batch_size)
        X = np.expand_dims(X_train[Xi], axis=3)
        Z = np.random.uniform(-1.0, 1.0, size=(batch_size,
code_length)).astype(np.float32)
        _, d_loss = session.run([training_step_d, loss_d],
                                feed_dict={
                                    input_x: X,
                                    input_z: Z,
                                    is_training: True
                                })
        d_losses.append(d_loss)
        Z = np.random.uniform(-1.0, 1.0, size=(batch_size,
code_length)).astype(np.float32)
        _, g_loss = session.run([training_step_g, loss_g],
                                feed_dict={
                                    input_x: X,
                                    input_z: Z,
                                    is_training: True
                                })
        g_losses.append(g_loss)
    print('Epoch {}) Avg. discriminator loss: {} - Avg. generator loss:
{}'.format(e + 1, np.mean(d_losses), np.mean(g_losses)))
```

The training step (with a single discriminator iteration) is split into two phases:

1. Discriminator training with a batch of true images and noise samples
2. Generator training with a batch of noise samples

Once the training process has finished, we can generate some images (50) by executing the generator with a matrix of noise samples:

```
Z = np.random.uniform(-1.0, 1.0, size=(50, code_length)).astype(np.float32)

Ys = session.run([gen],
                 feed_dict={
                     input_z: Z,
                     is_training: False
                 })

Ys = np.squeeze((Ys[0] + 1.0) * 0.5 * 255.0).astype(np.uint8)
```

The result is shown in the following screenshot:

Samples generated by a DCGAN trained with the Fashion-MNIST dataset

As an exercise, I invite the reader to employ more complex convolutional architectures and an RGB dataset such as CIFAR-10 (`https://www.cs.toronto.edu/~kriz/cifar.html`).

The training phase of this example and the following one, even if limited to 5,000 samples, can be quite slow (around 12-15 hours) particularly when no GPU is available. The reader can simplify the examples by reducing the complexity of the networks (paying attention to the shapes) and reducing the number of samples. To avoid mismatches, I suggest adding the `print(gen.shape)` command after the generator instance. The expected shape should be `(?, 64, 64, 1)`. Alternatively, it's possible to employ smaller target dimensions (like 32 × 32), setting one of the strides (possibly the last one) equal to `(1, 1)`.

Wasserstein GAN (WGAN)

As explained in the previous section, one of the most difficult problems with standard GANs is caused by the loss function based on the Jensen-Shannon divergence, whose value becomes constant when two distributions have disjointed supports. This situation is quite common with high-dimensional, semantically structured datasets. For example, images are constrained to having particular features in order to represent a specific subject (this is a consequence of the manifold assumption discussed in `Chapter 2`, *Introduction to Semi-Supervised Learning*). The initial generator distribution is very unlikely to overlap a true dataset, and in many cases, they are also very far from each other. This condition increases the risk of learning a wrong representation (a problem known as mode collapse), even when the discriminator is able to distinguish between true and generated samples (such a condition arises when the discriminator learns too quickly, with respect to the generator). Moreover, the Nash equilibrium becomes harder to achieve, and the GAN can easily remain blocked in a sub-optimal configuration.

In order to mitigate this problem, Arjovsky, Chintala, and Bottou (in *Wasserstein GAN*, *Arjovsky M.*, *Chintala S.*, *Bottou L.*, *arXiv:1701.07875 [stat.ML]*) proposed employing a different divergence, called the *Wasserstein distance* (or Earth Mover's distance), which is formally defined as follows:

$$D_W(p_{data}||p_g) = inf_{\mu \sim \prod(p_{data}, p_g)} \; E_{(x,y)\sim\mu}[||x - y||]$$

The term $\Pi(p_{data}, p_g)$ represents the set of all possible joint probability distributions between p_{data}, p_g. Hence, the Wasserstein distance is the infimum (considering all joint distributions) of the set of expected values of $||x - y||$, where x and y are sampled from the joint distribution μ. The main property of D_W is that, even when two distributions have disjointed support, its value is proportional to the actual distributional distance. The formal proof is not very complex, but it's easier to understand the concept intuitively. In fact, given two distributions with disjointed support, the infimum operator forces taking the shortest distance between each possible couple of samples. Clearly, this measure is more robust than the Jensen-Shannon divergence, but there's a practical drawback: it's extremely difficult to compute. As we cannot work with all possible joint distributions (nor with an approximation), a further step is necessary to employ this loss function. In the aforementioned paper, the authors proved that it's possible to apply a transformation, thanks to the Kantorovich-Rubinstein theorem (the topic is quite complex, but the reader can find further information in *On the Kantorovich–Rubinstein Theorem*, *Edwards D. A.*, *Expositiones Mathematicae*, 2011):

$$D_W(p_{data}||p_g) = \frac{1}{L} sup_{\|f\| \leqslant L} E_{x \sim p_{data}}[f(x)] - E_{x \sim p_g}[f(x)]$$

The first element to consider is the nature of $f(\bullet)$. The theorem imposes considering only L-Lipschitz functions, which means that $f(\bullet)$ (assuming a real-valued function of a single variable) must obey:

$$|f(x_1) - f(x_2)| \leqslant L|x_1 - x_2| \quad \forall x_1, x_2 \in \mathbb{R}$$

At this point, the Wasserstein distance is proportional to the supremum (with respect to all L-Lipschitz functions) of the difference between two expected values, which are extremely easy to compute. In a WGAN, the $f(\bullet)$ function is represented by a neural network; therefore, we have no warranties about the Lipschitz condition. To solve this problem, the author suggested a very simple procedure: clipping the discriminator (which is normally called Critic, and whose responsibility is to represent the parameterized function $f(\bullet)$) variables after applying the corrections. If the input is bounded, all of the transformations will yield a bounded output; however, the clipping factor must be small enough (0.01, or even smaller) to avoid the additive effect of multiple operations leading to an inversion of the Lipschitz condition. This is not an efficient solution (because it slows down the training process when it's not necessary), but it allows for exploiting the Kantorovich-Rubinstein theorem, even when there are no formal constraints imposed on the function family.

Using a parameterized function (such as a Deep Convolutional Network), the Wasserstein distance becomes as follows (omitting the term *L*, which is constant):

$$D_W(p_{data}||p_g) = max_{\bar{\theta}_c \in \Theta_c} E_{x \sim p_{data}}[f(x; \bar{\theta}_c)] - E_{z \sim p_{noise}}[f(g(\bar{z}; \bar{\theta}_g); \bar{\theta}_c)] = max_{\bar{\theta}_c \in \Theta_c} (W_{data} - W_{noise})$$

In the previous expression, we explicitly extracted the generator output, and in the last step, separated the term that will be optimized separately. The reader has probably noticed that the computation is simpler than a standard GAN because, in this case, we have to average over only the *f(•)* values of a batch (there's no more need for a logarithm). However, as the Critic variables are clipped, the number of required iterations is normally larger, and in order to compensate the difference between the training speeds of the Critic and generator, it's often necessary to set $N_{critic} > 1$ (the authors suggest a value equal to 5, but this is a hyperparameter that must be tuned in every specific context).

The complete WGAN algorithm is:

1. Set the number of epochs, $N_{epochs.}$
2. Set the number of Critic iterations, N_{critic} (in most cases, $N_{iter} = 5$).
3. Set the batch size, *k.*
4. Set a clipping constant, c (for example, c = 0.01).
5. Define a noise generating process, M (for example, *U(-1, 1)*).
6. For *e=1* to N_{epochs}:
    1. Sample *k* values from *X*.
    2. Sample *k* values from *N*.
    3. For *i=1* to N_{critic}:
        1. Compute the gradients, $\nabla_c D_W(p_{data}||p_g)$ (only with respect to the Critic variables). The expected values are approximated by sample means.
        2. Update the Critic parameters by Stochastic Gradient Ascent.
        3. Clip the Critic parameters in the range [*-c, c*].
    4. Sample *k* values from *N*.
    5. Compute the gradients, $\nabla_g W_{noise}$ (only with respect to the generator variables).
    6. Update the generator parameters by Stochastic Gradient Descent.

Example of WGAN with TensorFlow

This example can be considered a variant of the previous one because it uses the same dataset, generator, and discriminator. The only main difference is that in this case, the discriminator (together with its variable scope) has been renamed `critic()`:

```
import tensorflow as tf

def critic(x, is_training=True, reuse_variables=True):
    with tf.variable_scope('critic', reuse=reuse_variables):
...
```

At this point, we can step directly to the creation of the `Graph` containing all of the placeholders, operations, and loss functions:

```
import tensorflow as tf

graph = tf.Graph()

with graph.as_default():
    input_x = tf.placeholder(tf.float32, shape=(None, width, height, 1))
    input_z = tf.placeholder(tf.float32, shape=(None, code_length))
    is_training = tf.placeholder(tf.bool)
    gen = generator(z=tf.reshape(input_z, (-1, 1, 1, code_length)),
is_training=is_training)
    r_input_x = tf.image.resize_images(images=input_x, size=(64, 64))
    crit_1_l = critic(x=r_input_x, is_training=is_training,
reuse_variables=False)
    crit_2_l = critic(x=gen, is_training=is_training, reuse_variables=True)
    loss_c = tf.reduce_mean(crit_2_l - crit_1_l)
    loss_g = tf.reduce_mean(-crit_2_l)
    variables_g = [variable for variable in tf.trainable_variables() if
variable.name.startswith('generator')]
    variables_c = [variable for variable in tf.trainable_variables() if
variable.name.startswith('critic')]
    with
tf.control_dependencies(tf.get_collection(tf.GraphKeys.UPDATE_OPS)):
        optimizer_c = tf.train.AdamOptimizer(0.00005, beta1=0.5,
beta2=0.9).minimize(loss=loss_c, var_list=variables_c)
        with tf.control_dependencies([optimizer_c]):
            training_step_c = tf.tuple(tensors=[tf.assign(variable,
tf.clip_by_value(variable, -0.01, 0.01))
                                                for variable in
variables_c])
        training_step_g = tf.train.AdamOptimizer(0.00005, beta1=0.5,
beta2=0.9).minimize(loss=loss_g, var_list=variables_g)
```

As it's possible to see, there are no differences in the placeholder section, in the definition of the generator, and in the image resizing to the target dimensions of 64 × 64. In the next block, we define the two Critic instances (which are perfectly analogous to the ones declared in the previous example).

The two loss functions are simpler than a standard GAN, as they work directly with the Critic outputs, computing the sample mean over a batch. In the original paper, the authors suggest using RMSProp as the standard optimizer, in order to avoid the instabilities that a momentum-based algorithm can produce. However, Adam, with lower forgetting factors ($\mu_1 = 0.5$ and $\mu_2 = 0.9$) and a learning rate $\eta = 0.00005$, is faster than RMSProp, and doesn't lead to instabilities. I suggest testing both options, trying to maximize the training speed while preventing the mode collapse. Contrary to the previous example, in this case we need to clip all of the Critic variables after each training step. To avoid that, the internal concurrency can alter the order of some operations; it's necessary to employ a nested dependency control context manager. In this way, the actual `training_step_c` (responsible for clipping and reassigning the values to each variable) will be executed only after the `optimizer_c` step has completed.

Now, we can create the `InteractiveSession`, initialize the variables, and start the training process, which is very similar to the previous one:

```
import numpy as np
import tensorflow as tf

nb_epochs = 200
nb_critic = 5
batch_size = 64
nb_iterations = int(nb_samples / batch_size)

session = tf.InteractiveSession(graph=graph)
tf.global_variables_initializer().run()

samples_range = np.arange(nb_samples)

for e in range(nb_epochs):
    c_losses = []
    g_losses = []
    for i in range(nb_iterations):
        for j in range(nb_critic):
            Xi = np.random.choice(samples_range, size=batch_size)
            X = np.expand_dims(X_train[Xi], axis=3)
            Z = np.random.uniform(-1.0, 1.0, size=(batch_size,
code_length)).astype(np.float32)
            _, c_loss = session.run([training_step_c, loss_c],
                                    feed_dict={
```

```
                                        input_x: X,
                                        input_z: Z,
                                        is_training: True
                                    })
            c_losses.append(c_loss)
        Z = np.random.uniform(-1.0, 1.0, size=(batch_size,
code_length)).astype(np.float32)
        _, g_loss = session.run([training_step_g, loss_g],
                                feed_dict={
                                    input_x: np.zeros(shape=(batch_size,
width, height, 1)),
                                    input_z: Z,
                                    is_training: True
                                })
        g_losses.append(g_loss)
    print('Epoch {}) Avg. critic loss: {} - Avg. generator loss:
{}'.format(e + 1, np.mean(c_losses), np.mean(g_losses)))
```

The main difference is that, in this case, the Critic is trained `n_critic` times before each generator training step. The result of the generation of 50 random samples is shown in the following screenshot:

Samples generated by a WGAN trained with the Fashion MNIST dataset

As it's possible to see, the quality is slightly higher, and the samples are smoother. I invite the reader to also test this model with an RGB dataset, because the final quality is normally excellent.

When working with these models, the training time can be very long. To avoid waiting to see the initial results (and to perform the required tuning), I suggest using Jupyter. In this way, it's possible to stop the learning process, check the generator ability, and restart it without any problem. Of course, the graph must remain the same, and the variable initialization must be performed only at the beginning.

Summary

In this chapter, we discussed the main principles of adversarial training, and explained the roles of two players: the generator and discriminator. We described how to model and train them using a minimax approach whose double goal is to force the generator to learn the true data distribution p_{data}, and get the discriminator to distinguish perfectly between true samples (belonging to p_{data}) and unacceptable ones. In the same section, we analyzed the inner dynamics of a Generative Adversarial Network and some common problems that can slow down the training process and lead to a sub-optimal final configuration.

One of the most difficult problems experienced with standard GANs arises when the data generating process and the generator distribution have disjointed support. In this case, the Jensen-Shannon divergence becomes constant and doesn't provide precise information about the distance. An excellent alternative is provided by the Wasserstein measure, which is employed in a more efficient model, called WGAN. This method can efficiently manage disjointed distributions, but it's necessary to enforce the L-Lipschitz condition on the Critic. The standard approach is based on clipping the parameters after each gradient ascent update. This simple technique guarantees the L-Lipschitz condition, but it's necessary to use very small clipping factors, and this can lead to a slower conversion. For this reason, it's normally necessary to repeat the training of the Critic a fixed number of times (such as five) before each single generator training step.

In the next chapter, we are going to introduce another probabilistic generative neural model, based on a particular kind of neural network, called the Restricted Boltzmann Machine.

13
Deep Belief Networks

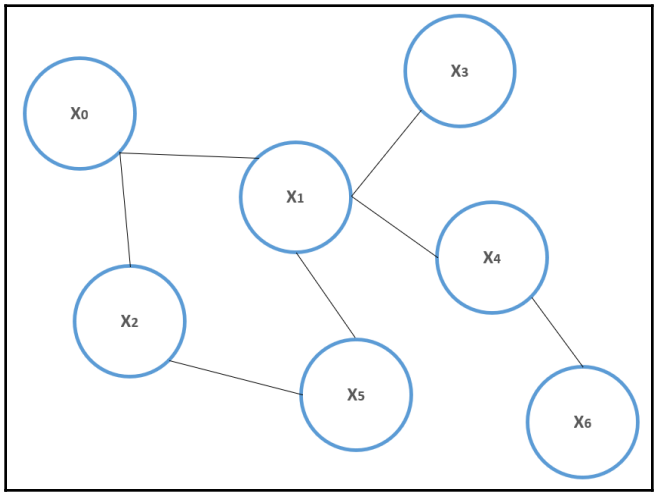
In this chapter, we are going to present two probabilistic generative models that employ a set of latent variables to represent a specific data generation process. **Restricted Boltzmann Machines** (**RBMs**), proposed in 1986, are the building blocks of a more complex model, called a **Deep Belief Network** (**DBN**), which is capable of capturing complex relationships among features at different levels (in a way not dissimilar to a deep convolutional network). Both models can be used in unsupervised and supervised scenarios as preprocessors or, as is usual with DBN, fine-tuning the parameters using a standard backpropagation algorithm.

In particular, we will discuss:

- **Markov random fields** (**MRF**)
- RBM
- **Contrastive Divergence** (**CD-k**) algorithm
- DBN with supervised and unsupervised examples

MRF

Let's consider a set of random variables, x_i, organized in an undirected graph, $G=(V, E)$, as shown in the following diagram:



Example of a probabilistic undirected graph

Two random variables, a and b, are conditionally independent given the random variable, c if:

$$P(a, b|c) = P(a|c)P(b|c)$$

Now, consider the graph again; if all generic couples of subsets of variables S_i and S_j are conditionally independent given a separating subset, S_k (so that all connections between variables belonging to S_i to variables belonging to S_j pass through S_k), the graph is called a **Markov random field** (**MRF**).

Given *G=(V, E)*, a subset containing vertices such that every couple is adjacent is called a **clique** (the set of all cliques is often denoted as *cl(G)*). For example, consider the graph shown previously; (x_0, x_1) is a clique and if x_0 and x_5 were connected, (x_0, x_1, x_5) would be a clique. A **maximal clique** is a clique that cannot be expanded by adding new vertices. A particular family of MRF is made up of all those graphs whose joint probability distribution can be factorized as:

$$p(\bar{x}) = \alpha \prod_{i \in cl(G)} \rho_i(\bar{x})$$

In this case, α is the normalizing constant and the product is extended to the set of all maximal cliques. According to the **Hammersley–Clifford** theorem (for further information, please refer to *Proof of Hammersley-Clifford Theorem*, *Cheung S., University of Kentucky, 2008*), if the joint probability density function is strictly positive, the MRF can be factorized and all the ϱ_i functions are strictly positive too. Hence *p(x)*, after some straightforward manipulations based on the properties of logarithms, can be rewritten as a **Gibbs** (or **Boltzmann**) distribution:

$$p(\bar{x}) = \alpha\, e^{log \prod_{i \in cl(G)} \rho_i(\bar{x})} = \alpha\, e^{\sum_{i \in cl(G)} log\ \rho_i(\bar{x})} = \frac{1}{Z} e^{-E(\bar{x})}$$

The term *E(x)* is called energy, as it derives from the first application of such a distribution in statistical physics. *1/Z* is now the normalizing constant employing the standard notation. In our scenarios, we always consider graphs containing observed (x_i) and latent variables (h_j). Therefore, it's useful to express the joint probability as:

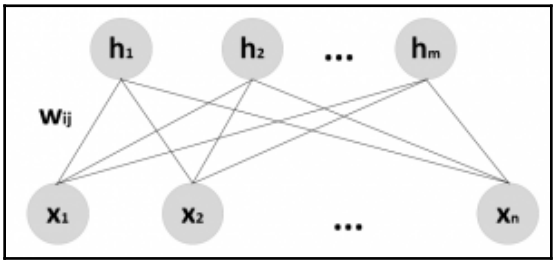
$$p(\bar{x}, \bar{h}) = \frac{1}{Z} e^{-E(\bar{x}, \bar{h})}$$

Whenever it's necessary to marginalize to obtain *p(x)*, we can simply sum over h_j:

$$p(\bar{x}) = \sum_j \frac{1}{Z} e^{-E(\bar{x}, \bar{h}_j)}$$

RBMs

A RBM (originally called **Harmonium**) is a neural model proposed by Smolensky (in *Information processing in dynamical systems: Foundations of harmony theory, Smolensky P., Parallel Distributed Processing, Vol 1, The MIT Press*) that is made up of a layer of input (observable) neurons and a layer of hidden (latent) neurons. A generic structure is shown in the following diagram:



Structure of Restricted Boltzmann Machine

As the undirected graph is bipartite (there are no connections between neurons belonging to the same layer), the underlying probabilistic structure is MRF. In the original model (even if this is not a restriction), all the neurons are assumed to be Bernoulli-distributed *(x_i, $h_i = \{0, 1\}$)*, with a bias, b_i (for the observed units) and c_j (for the latent neurons). The resulting energy function is:

$$E(\bar{x}, \bar{h}) = -\sum_i \sum_j w_{ij} \bar{x}_i \bar{h}_j - \sum_i \bar{b}_i \bar{x}_i - \sum_j \bar{c}_j \bar{h}_j$$

A RBM is a probabilistic generative model that can learn a data-generating process, p_{data}, which is represented by the observed units but exploits the presence of the latent variables in order to model all the internal relationships. If we summarized all the parameters in a single vector, $\theta = \{w_{ij}, b_i, c_j\}$, the Gibbs distribution becomes:

$$p(\bar{x}, \bar{h}; \bar{\theta}) = \frac{1}{Z} e^{-E(\bar{x}, \bar{h}; \bar{\theta})} = \frac{e^{-E(\bar{x}, \bar{h}; \bar{\theta})}}{\sum_i \sum_j e^{-E(\bar{x}_i, \bar{h}_j; \bar{\theta})}}$$

The training goal of a RBM is to maximize the log-likelihood with respect to an input distribution. Hence, the first step is determining $L(\theta; x)$ after the marginalization of the previous expression:

$$L(\bar{\theta}; \bar{x}) = log\, p(\bar{x}; \bar{\theta}) = log \sum_{\bar{h}} \frac{1}{Z} e^{-E(\bar{x}, \bar{h}; \bar{\theta})} = log \sum_{\bar{h}} e^{-E(\bar{x}, \bar{h}; \bar{\theta})} - log \sum_{\bar{x}} \sum_{\bar{h}} \frac{1}{Z} e^{-E(\bar{x}, \bar{h}; \bar{\theta})}$$

As we need to maximize the log-likelihood, it's useful to compute the gradient with respect to θ:

$$\nabla_{\bar{\theta}} L(\bar{\theta}; \bar{x}) = \nabla_{\bar{\theta}} log \sum_{\bar{h}} e^{-E(\bar{x}, \bar{h}; \bar{\theta})} - \nabla_{\bar{\theta}} log \sum_{\bar{x}} \sum_{\bar{h}} \frac{1}{Z} e^{-E(\bar{x}, \bar{h}; \bar{\theta})}$$

Applying the chain rule of derivatives, we get:

$$\nabla_{\bar{\theta}} L(\bar{\theta}; \bar{x}) = -\sum_{\bar{h}} \frac{e^{-E(\bar{x}, \bar{h}; \bar{\theta})}}{\sum_{\bar{h}} e^{-E(\bar{x}, \bar{h}; \bar{\theta})}} \nabla_{\bar{\theta}} E(\bar{x}, \bar{h}; \bar{\theta}) + \sum_{\bar{x}} \sum_{\bar{h}} \frac{e^{-E(\bar{x}, \bar{h}; \bar{\theta})}}{\sum_{\bar{x}} \sum_{\bar{h}} e^{-E(\bar{x}, \bar{h}; \bar{\theta})}} \nabla_{\bar{\theta}} E(\bar{x}, \bar{h}; \bar{\theta})$$

Using the conditional and joint probability equalities, the previous expression becomes:

$$\nabla_{\bar{\theta}} L(\bar{\theta}; \bar{x}) = -\sum_{\bar{h}} p(\bar{h}|\bar{x}; \bar{\theta}) \nabla_{\bar{\theta}} E(\bar{x}, \bar{h}^{(p)}; \bar{\theta}) + \sum_{\bar{x}} \sum_{\bar{h}} p(\bar{x}, \bar{h}; \bar{\theta}) \nabla_{\bar{\theta}} E(\bar{x}, \bar{h}; \bar{\theta})$$

Considering the full joint probability, after some tedious manipulations (which we omit), it's possible to derive the following expressions ($\sigma(\bullet)$ is the sigmoid function):

$$\begin{cases} p\left(\bar{h}_j = 1|\bar{x}\right) = \sigma\left(\sum_i w_{ij} \bar{x}_i + c_j\right) \\ p\left(\bar{x}_i = 1|\bar{h}\right) = \sigma\left(\sum_j w_{ij} \bar{h}_j + b_i\right) \end{cases}$$

At this point, we can compute the gradient of the log-likelihood with respect to each single parameter, w_{ij}, b_i, and c_j. Starting with w_{ij}, and considering that $\nabla_{w_{ij}} E(x, h; \theta) = -x_i h_j$, we get:

$$\nabla_{w_{ij}} L(\bar{\theta}; \bar{x}) = \sum_{\bar{h}} p(\bar{h}|\bar{x}; \bar{\theta}) \bar{x}_i \bar{h}_j - \sum_{\bar{x}} \sum_{\bar{h}} p(\bar{x}, \bar{h}; \bar{\theta}) \bar{x}_i \bar{h}_j$$

The expression can be rewritten as:

$$\nabla_{w_{ij}} L(\bar{\theta}; \bar{x}) = \sum_{\bar{h}} p(\bar{h}|\bar{x}; \bar{\theta}) \bar{x}_i \bar{h}_j - \sum_{\bar{x}} p(\bar{x}; \bar{\theta}) \sum_{\bar{h}} p(\bar{h}|\bar{x}; \bar{\theta}) \bar{x}_i \bar{h}_j$$

Now, considering that all the units are Bernoulli-distributed, and isolating only the j^{th} hidden unit, it's possible to apply the simplification:

$$\sum_{\bar{h}} p(\bar{h}|\bar{x};\bar{\theta})\bar{x}_i\bar{h}_j = \sum_{\bar{h}} \prod_i p(\bar{h}_i|\bar{x};\bar{\theta})\bar{x}_i\bar{h}_j =$$
$$= \sum_{\bar{h}_j} p(\bar{h}_j|\bar{x};\bar{\theta})\bar{x}_i\bar{h}_j \sum_{\bar{h}_{i\neq j}} p(\bar{h}_i|\bar{x};\bar{\theta}) = \sum_{\bar{h}_j} p(\bar{h}_j|\bar{x};\bar{\theta})\bar{x}_i\bar{h}_j = p(\bar{h}_j = 1|\bar{x})\bar{x}_i$$

Therefore, the gradient becomes:

$$\nabla_{w_{ij}} L(\bar{\theta};\bar{x}) = p(\bar{h}_j = 1|\bar{x})\bar{x}_i - \sum_{\bar{x}} p(\bar{x};\bar{\theta})p(\bar{h}_j = 1|\bar{x})\bar{x}_i$$

Analogously, we can derive the gradient of L with respect to b_i and c_j:

$$\begin{cases} \nabla_{b_i} L(\bar{\theta};\bar{x}) = \bar{x}_i - \sum_{\bar{x}} p(\bar{x};\bar{\theta})\bar{x}_i \\ \nabla_{c_j} L(\bar{\theta};\bar{x}) = p(\bar{h}_j = 1|\bar{x}) - \sum_{\bar{x}} p(\bar{x};\bar{\theta})p(\bar{h}_j = 1|\bar{x}) \end{cases}$$

Hence, the first term of every gradient is very easy to compute, while the second one requires summing over all observed values. As this operation is impracticable, the only feasible alternative is an approximation based on sampling, using a method such as Gibbs sampling (for further information, see `Chapter 4`, *Bayesian Networks and Hidden Markov Models*). However, as this algorithm samples from the conditionals $p(x|h)$ and $p(h|x)$, rather than from the full joint distribution $p(x, h)$, it requires the associated Markov chain to reach its stationary distribution, π, in order to provide valid samples. As we don't know how many sampling steps are required to reach π, Gibbs sampling can also be an unfeasible solution because of its potentially high computational cost.

In order to solve this problem, Hinton proposed (in *A Practical Guide to Training Restricted Boltzmann Machines, Hinton G., Dept. Computer Science, University of Toronto*) an alternative algorithm called **CD-k**. The idea is very simple but extremely effective: instead of waiting for the Markov chain to reach the stationary distribution, we sample a fixed number of times starting from a training sample at $t=0$ $x^{(0)}$ and computing $h^{(1)}$ by sampling from $p(h^{(1)}|x^{(0)})$. Then, the hidden vector is employed to sample the reconstruction, $x^{(2)}$, from $p(x^{(2)}|h^{(1)})$. This procedure can be repeated any number of times, but in practice, a single sampling step is normally enough to ensure quite good accuracy. At this point, the gradient of the log-likelihood is approximated as (considering t steps):

$$\nabla_{\bar{\theta}} L(\bar{\theta}; \bar{x}) \approx -\sum_{p} p(\bar{h}^{(p)} | \bar{x}^{(0)}; \bar{\theta}) \nabla_{\bar{\theta}} E(\bar{x}^{(0)}, \bar{h}^{(p)}; \bar{\theta}) + \sum_{p} p(\bar{h}^{(p)} | \bar{x}^{(t)}; \bar{\theta}) \nabla_{\bar{\theta}} E(\bar{x}^{(t)}, \bar{h}^{(q)}; \bar{\theta})$$

The single gradients with respect to w_{ij}, b_i, and c_j can be easily obtained considering the preceding procedure. The term *contrastive* derives from the approximation of the gradient of L computed at $x^{(0)}$ with a weighted difference between a term called the *positive gradient* and another defined as the *negative gradient*. This approach is analogous to the approximation of a derivative with this incremental ratio:

$$\frac{\partial L}{\partial x} \approx \frac{1}{2h} L(x + h) - \frac{1}{2h} L(x - h) \quad if \quad h \rightarrow 0$$

The complete RBM training algorithm, based on a single-step CD-k is (assuming that there are M training samples):

1. Set the number, N_h, of hidden units
2. Set a number of epochs, N_e
3. Set a `learning_rate` η (for example, $\eta = 0.01$)
4. For $e=1$ to N_e:
    1. Set $\Delta w = 0$, $\Delta b = 0$, and $\Delta c = 0$
    2. For $i=1$ to M:
        1. Sample $h^{(i)}$ from $p(h | x^{(i)})$
        2. Sample a reconstruction $x^{(i+1)}$ from $p(x^{(i+1)} | h^{(i)})$
        3. Accumulate the updates for weights and biases:
            1. Δw += $p(h = 1 | x^{(i)}) x^{(i)} - p(h = 1 | x^{(i+1)}) x^{(i+1)}$ (as outer product)
            2. Δb += $x^{(i)} - x^{(i+1)}$
            3. Δc += $p(h = 1 | x^{(i)}) - p(h = 1 | x^{(i+1)})$
    3. Update weights and biases:
        1. w += $\eta \Delta w$
        2. b += $\eta \Delta b$
        3. c += $\eta \Delta c$

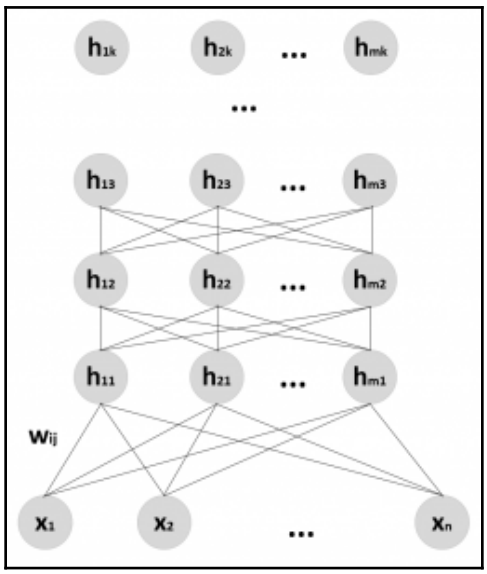
The outer product between two vectors is defined as:

$$\bar{a} \otimes \bar{b} = \begin{pmatrix} a_1 \\ \vdots \\ a_n \end{pmatrix} \cdot \begin{pmatrix} b_1 & \cdots & b_m \end{pmatrix} = \begin{pmatrix} a_1 b_1 & \cdots & a_1 b_m \\ \vdots & \ddots & \vdots \\ a_n b_1 & \cdots & a_n b_m \end{pmatrix}$$

If vector *a* has an *(n, 1)* shape and *b* has an *(m, 1)* shape, the result is a matrix with a *(n, m)* shape.

DBNs

A Belief or Bayesian network is a concept already explored in `Chapter 4`, *Bayesian Networks and Hidden Markov Models*. In this particular case, we are going to consider Belief Networks where there are visible and latent variables, organized into homogeneous layers. The first layer always contains the input (visible) units, while all the remaining ones are latent. Hence, a DBN can be structured as a stack of RBMs, where each hidden layer is also the visible one of the subsequent RBM, as shown in the following diagram (the number of units can be different for each layer):



Structure of a generic Deep Belief Network

The learning procedure is usually greedy and step-wise (as proposed in *A fast learning algorithm for deep belief nets*, *Hinton G. E.*, *Osindero S.*, *Teh Y. W.*, *Neural Computation*, *18*/*7*). The first RBM is trained with the dataset and optimized to reconstruct the original distribution using the CD-k algorithm. At this point, the internal (hidden) representations are employed as input for the next RBM, and so on until all the blocks are fully trained. In this way, the DBN is forced to create subsequent internal representations of the dataset that can be used for different purposes. Of course, when the model is trained, it's possible to infer from the recognition (inverse) model sampling from the hidden layers and compute the activation probability as (x represents a generic cause):

$$p\left(\bar{x}_i = 1|\bar{h}\right) = \sigma\left(\sum_j w_{ij}\bar{h}_j + b_i\right)$$

As a DBN is always a generative process, in an unsupervised scenario, it can perform a component analysis/dimensionality reduction with an approach that is based on the idea of creating a chain of sub-processes, which are able to rebuild an internal representation. While a single RBM focuses on a single hidden layer and hence cannot learn sub-features, a DBN greedily learns how to represent each sub-feature vector using a refined hidden distribution. The concept behind this process is not very different from a cascade of convolutional layers, with the main difference that in this case, the learning procedure is greedy. Another distinction with methods such as PCA is that we don't know exactly how the internal representation is built. As the latent variables are optimized by maximizing the log-likelihood, there are possibly many optimal points but we cannot easily impose constraints on them. However, DBNs show very powerful properties in different scenarios, even if their computational cost is normally considerably higher than other methods. One of the main problems (common to the majority of deep learning methods) concerns the right choice of hidden units in every layer. As they represent latent variables, their number is a crucial factor for the success of a training procedure. The right choice is not immediate, because it's necessary to know the complexity of the data-generating process, however, as a rule of thumb, I suggest starting with a couple of layers containing 32/64 units and proceeding to increase the number of hidden neurons and the layers until the desired accuracy is reached (in the same way, I suggest starting with a small learning rate, for example, 0.01 -, increasing it if necessary).

As the first RBM is responsible for reconstructing the original dataset, it's very useful to monitor the log-likelihood (or the error) after each epoch in order to understand whether the process is learning correctly (decreasing error) or it's saturating the capacity. It's clear that an initial bad reconstruction leads to subsequently worse representations. As the learning process is greedy, in an unsupervised task there's no way to improve the performance of lower layers when the previous training steps are finished therefore, I always suggest tuning up the parameters so that the first reconstruction is very accurate. Of course, all the considerations about overfitting are still valid, so, it's also important to monitor the generalization ability with validation samples. However, in a component analysis, we assume we're working with a distribution that is representative of the underlying data-generating process, so the risk of finding before-seen features should be minimal.

In a supervised scenario, there are generally two options whose first step is always a greedy training of the DBN. However, the first approach performs a subsequent refinement using a standard algorithm, such as backpropagation (considering the whole architecture as a single deep network), while the second one uses the last internal representation as the input of a separate classifier. It goes without saying that the first method has many more degrees of freedom because it works with a pre-trained network whose weights can be adjusted until the validation accuracy reaches its maximum value. In this case, the first greedy step works with the same assumption that has been empirically confirmed by observing the internal behavior of deep models (similar to convolutional networks). The first layers learn how to detect low-level features, while all the subsequent ones increase the details. Therefore, the backpropagation step presumably starts from a point that is already quite close to the optimum and can converge more quickly. Conversely, the second approach is analogous to applying the kernel trick to a standard **Support Vector Machine** (**SVM**). In fact, the external classifier is generally a very simple one (such as a logistic regression or an SVM) and the increased accuracy is normally due to an improved linear separability obtained by projecting the original samples onto a sub-space (often higher-dimensional) where they can be easily classified. In general, this method yields worse performance than the first one because there's no way to tune up the parameters once the DBN is trained. Therefore, when the final projections are not suitable for a linear classification, it's necessary to employ more complex models and the resulting computational cost can be very high without a proportional performance gain. As deep learning is generally based on the concept of end-to-end learning, training the whole network can be useful to implicitly include the pre-processing steps in the complete structure, which becomes a black box that associates input samples with specific outcomes. On the other hand, whenever an explicit pipeline is requested, greedy-training the DBN and employing a separate classifier could be a more suitable solution.

Example of unsupervised DBN in Python

In this example, we are going to use a Python library freely available on GitHub (`https://github.com/albertbup/deep-belief-network`) that allows working with supervised and unsupervised DBN using NumPy (CPU-only) or Tensorflow (CPU or GPU support) with the standard Scikit-Learn interface. Our goal is to create a lower-dimensional representation of a subset of the `mnist` dataset (as the training process can be quite slow, we'll limit it to `400` samples). The first step is loading (using the Keras helper function), shuffling, and normalizing the dataset:

```
import numpy as np

from keras.datasets import mnist
from sklearn.utils import shuffle

(X_train, Y_train), (_, _) = mnist.load_data()
X_train, Y_train = shuffle(X_train, Y_train, random_state=1000)

nb_samples = 400

width = X_train.shape[1]
height = X_train.shape[2]

X = X_train[0:nb_samples].reshape((nb_samples, width *
height)).astype(np.float32) / 255.0
Y = Y_train[0:nb_samples]
```

At this point, we can create an instance of the the `UnsupervisedDBN` class, setting three layers with respectively `512`, `256`, and `64` sigmoid units (as we want to bind the values between `0` and `1`). The learning rate, η (`learning_rate_rbm`), is set equal to `0.05`, the batch size (`batch_size`) to `64`, and the number of epochs for each RBM (`n_epochs_rbm`) to `100`. The default value for the number of CD-k steps is `1`, but it's possible to change it using the `contrastive_divergence_iter` parameter:

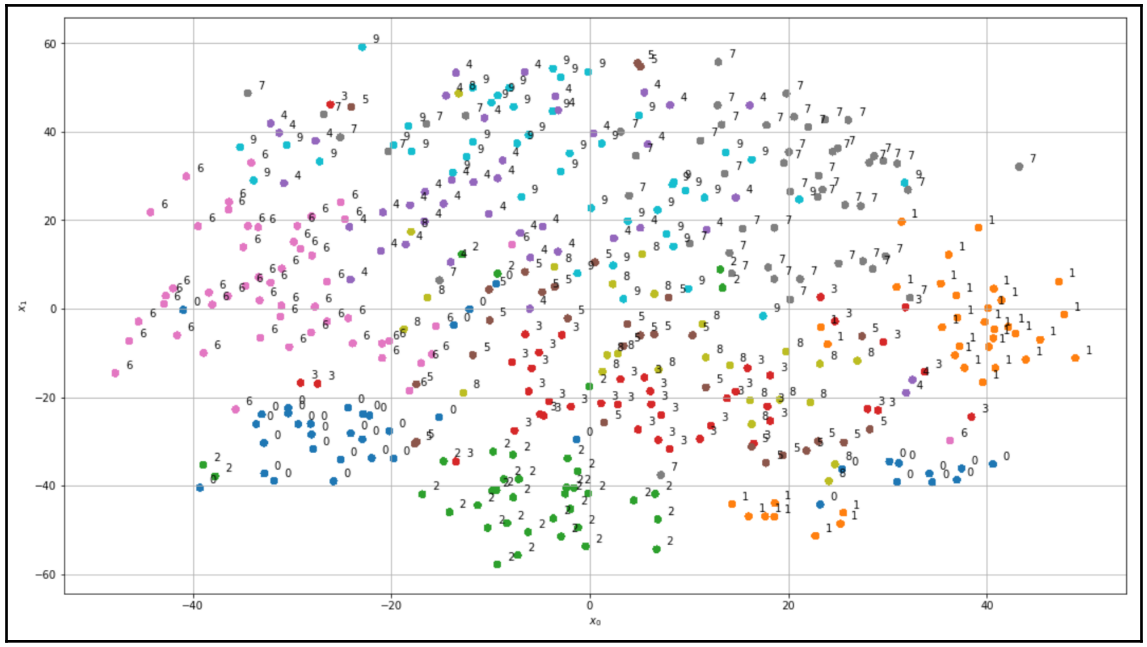
```
from dbn.tensorflow import UnsupervisedDBN

unsupervised_dbn = UnsupervisedDBN(hidden_layers_structure=[512, 256, 64],
                                   learning_rate_rbm=0.05,
                                   n_epochs_rbm=100,
                                   batch_size=64,
                                   activation_function='sigmoid')

X_dbn = unsupervised_dbn.fit_transform(X)

[START] Pre-training step:
```

```
>> Epoch 1 finished       RBM Reconstruction error 55.562027
>> Epoch 2 finished       RBM Reconstruction error 53.663380

...

>> Epoch 99 finished       RBM Reconstruction error 5.169244
>> Epoch 100 finished     RBM Reconstruction error 5.130809
[END] Pre-training step
```

Once the training process is complete, the `X_dbn` array contains the values sampled from the last hidden layer. Unfortunately, this library doesn't implement an inverse transformation method, but we can use the t-SNE algorithm to project the distribution onto a bidimensional space:

```
from sklearn.manifold import TSNE

tsne = TSNE(n_components=2, perplexity=20, random_state=1000)
X_tsne = tsne.fit_transform(X_dbn)
```

The corresponding plot is shown in the following graph:



t-SNE plot of the last DBN hidden layer distribution (64-dimensional)

As you can see, even if there are still a few anomalies, the hidden low-dimensional representation is globally coherent with the original dataset because the group containing the same digits is organized in compact clusters that preserve some geometrical properties. For example, the group containing the digits representing a **1** is very close to the one containing the images of 7s, as well as the groups of 3s and 8s. This result confirms that a DBN can be successfully employed as a preprocessing layer for classification purposes, but in this case, rather than reducing the dimensionality, it's often preferable to increase it, in order to exploit the redundancy to use a simpler linear classifier (to better understand this concept, think about augmenting a dataset with polynomial features). I invite you to test this ability by preprocessing the whole MNIST dataset and then classifying it using a logistic regression, comparing the results with a direct approach.

The library can be installed using the `pip install git+git://github.com/albertbup/deep-belief-network.git` command (NumPy or Tensorflow CPU) or `pip install git+git://github.com/albertbup/deep-belief-network.git@master_gpu` (Tensorflow GPU). In both cases, the commands will also install Tensorflow and other dependencies that are often present in common Python distributions (such as Anaconda); therefore, in order to limit the installation only to the DBN component, it's necessary to add the `--no-deps` attribute to the `pip` command. For further information, please refer to the GitHub page.

Example of Supervised DBN with Python

In this example, we are going to employ the KDD Cup '99 dataset (provided by Scikit-Learn), which contains the logs generated by an intrusion detection system exposed to normal and dangerous network activities. We are focusing only on the `smtp` sub-dataset, which is the smallest one, because, as explained before, the training process can be very long. This dataset is not extremely complex and it can be successfully classified with simpler methods; however, the example has only a didactic purpose and can be useful for understanding how to work with this kind of data.

The first step is to load the dataset, encode the labels (which are strings), and standardize the values:

```
from sklearn.datasets import fetch_kddcup99
from sklearn.preprocessing import LabelEncoder, StandardScaler

kddcup = fetch_kddcup99(subset='smtp', shuffle=True, random_state=1000)
```

```
ss = StandardScaler()
X = ss.fit_transform(kddcup['data']).astype(np.float32)

le = LabelEncoder()
Y = le.fit_transform(kddcup['target']).astype(np.float32)
```

At this point, we can create train and test sets:

```
from sklearn.model_selection import train_test_split

X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.25,
random_state=1000)
```

The model is based on an instance of the `SupervisedDBNClassification` class, which implements the backpropagation method. The parameters are very similar to the unsupervised case, but now we can also specify the **stochastic gradient descent** (**SGD**) learning rate (`learning_rate`), the number of backpropagation epochs (`n_iter_backprop`), and an optional dropout (`dropout_p`). The algorithm performs an initial greedy training (whose computational cost is normally higher than the SGD phase), followed by a fine-tuning:

```
from dbn.tensorflow import SupervisedDBNClassification

classifier = SupervisedDBNClassification(hidden_layers_structure=[64, 64],
                                         learning_rate_rbm=0.001,
                                         learning_rate=0.01,
                                         n_epochs_rbm=20,
                                         n_iter_backprop=150,
                                         batch_size=256,
                                         activation_function='relu',
                                         dropout_p=0.25)

classifier.fit(X_train, Y_train)

[START] Pre-training step:
>> Epoch 1 finished       RBM Reconstruction error 2.478997
>> Epoch 2 finished       RBM Reconstruction error 2.459004

...

>> Epoch 147 finished       ANN training loss 0.006651
>> Epoch 148 finished     ANN training loss 0.006631
>> Epoch 149 finished     ANN training loss 0.006612
[END] Fine tuning step

SupervisedDBNClassification(batch_size=256, dropout_p=0.25,
              idx_to_label_map={0: 1.0, 1: 0.0, 2: 2.0},
```

```
                l2_regularization=1.0,
                label_to_idx_map={0.0: 1, 1.0: 0, 2.0: 2},
                learning_rate=0.01, n_iter_backprop=150, verbose=True)
```

Once the training process is finished, we can evaluate performance on the test set:

```
from sklearn.metrics.classification import accuracy_score

Y_pred = classifier.predict(X_test)
print(accuracy_score(Y_test, Y_pred))
1.0
```

The validation accuracy is `1.0` (there are no misclassifications), but this is really a simple dataset that needs only a few minutes of training. I invite you to test the performance of a DBN in the classification of the MNIST/Fashion MNIST dataset, comparing the results with the one obtained using a deep convolutional network. In this case, it's important to monitor the reconstruction error of each RBM, trying to minimize it before running the backpropagation phase. At the end of this exercise, you should be able to answer this question: which is preferable, an end-to-end or a preprocessing-based approach?

When running these experiments, where there's an intensive use of sampling, I always suggest setting the random seed (and keeping it constant) in NumPy in order to guarantee reproducibility (using the `np.random.seed(...)` command). As this library also works with Tensorflow, it's necessary to repeat the operation using the `tf.set_random_seed(...)` command.

Summary

In this chapter, we presented the MRF as the underlying structure of an RBM. An MRF is represented as an undirected graph whose vertices are random variables. In particular, for our purposes, we considered MRFs whose joint probability can be expressed as a product of the positive functions of each random variable. The most common distribution, based on an exponential, is called the Gibbs (or Boltzmann) distribution and it is particularly suitable for our problems because the logarithm cancels the exponential, yielding simpler expressions.

An RBM is a simple bipartite, undirected graph, made up of visible and latent variables, with connections only between different groups. The goal of this model is to learn a probability distribution, thanks to the presence of hidden units that can model the unknown relationships. Unfortunately, the log-likelihood, although very simple, cannot be easily optimized because the normalization term requires summing over all the input values. For this reason, Hinton proposed an alternative algorithm, called CD-k, which outputs an approximation of the gradient of the log-likelihood based on a fixed number (normally 1) of Gibbs sampling steps.

Stacking multiple RBMs allows modeling DBNs, where the hidden layer of each block is also the visible layer of the following one. DBN can be trained using a greedy approach, maximizing the log-likelihood of each RBM in sequence. In an unsupervised scenario, a DBN is able to extract the features of a data-generating process in a hierarchical way, and therefore the application includes component analysis and dimensionality reduction. In a supervised scenario, a DBN can be greedily pre-trained and fine-tuned using the backpropagation algorithm (considering the whole network) or sometimes using a preprocessing step in a pipeline where the classifier is generally a very simple model (such as a logistic regression).

In the next chapter, `Chapter 14`, *Introduction to Reinforcement Learning*, we are going to introduce the concept of reinforcement learning, discussing the most important elements of systems that can autonomously learn to play a game or allow a robot to walk, jump, and perform tasks that are extremely difficult to model and control using classic methods.

14
Introduction to Reinforcement Learning

In this chapter, we are going to introduce the fundamental concepts of **Reinforcement Learning** (**RL**), which is a set of approaches that allows an agent to learn how to behave in an unknown environment, thanks to the rewards that are provided after each possible action. RL has been studied for decades, but it has reached a very high maturity level in the last few years when it became possible to employ deep learning models together with standard (and often simple) algorithms in order to solve extremely complex problems (such as learning how to play an Atari game perfectly).

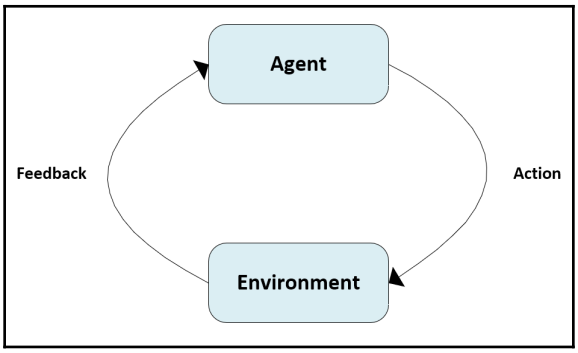
In particular, we will discuss:

- The concepts of environment, agent, policy, and reward
- The concept of the **Markov Decision Process** (**MDP**)
- The policy iteration algorithm
- The value iteration algorithm
- The TD(0) algorithm

Reinforcement Learning fundamentals

Imagine that you want to learn to ride a bike and ask a friend for advice. They explain how the gears work, how to release the brake and a few other technical details. In the end, you ask the secret to keeping balanced. What kind of answer do you expect? In an imaginary supervised world, you should be able to perfectly quantify your actions and correct the errors by comparing the outcomes with precise reference values. In the real world, you have no idea about the quantities underlying your actions and, above all, you will never know what the right value is. Increasing the level of abstraction, the scenario we're considering can be described as: a generic **agent** performs actions inside an **environment** and receives **feedback** that is somehow proportional to the competence of its actions.

According to this **feedback**, the **agent** can correct its actions in order to reach a specific goal. This basic schema is represented in the following diagram:



Basic RL schema

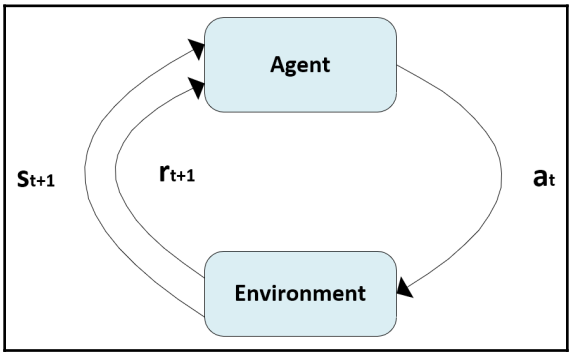
Returning to our initial example, when you ride a bike for the first time and try to keep your balance, you will notice that the wrong movement causes an increase in the slope, which in turn increases the horizontal component of the gravity force, pushing the bike laterally. As the vertical component is compensated, the result is a rotation that ends when the bike falls down completely. However, as you can use your legs to control the balance, when the bike starts falling, thanks to Newton's third law, the force on the leg increases and your brain understands that it's necessary to make a movement in the opposite direction. Even if this problem can be easily expressed in terms of physical laws, nobody learns to ride a bike by computing forces and momentums. This is one of the main concepts of RL: an agent must always make its choices considering a piece of information, usually defined as a *reward*, that represents the response, provided by the environment. If the action is correct, the reward will be positive, otherwise, it will be negative. After receiving a reward, an agent can fine-tune the strategy, called *policy*, in order to maximize the expected future reward. For example, after a few rides, you will be able to slightly move your body so as to keep the balance while turning, but probably, in the beginning, you needed to extend your leg to avoid falling down. Hence, your initial policy suggested a wrong action, which received repeated negative rewards and so your brain corrected it by increasing the probability of choosing another action. The implicit hypothesis that underlies this approach is that an agent is always *rational*, meaning that its goal is to maximize the expected return of its actions (nobody would like to fall down just to feel a different emotion).

Before discussing the single components of an RL system, it's necessary to add a couple of fundamental assumptions. The first one is that an agent can repeat the experiences an infinite number of times. In other words, we assume that it's possible to learn a valid policy (possibly the optimal one) only if we have enough time. Clearly, this is unacceptable in the animal world and we all know that many experiences are extremely dangerous; however, this assumption is necessary to prove the convergence of some algorithms. Indeed, sub-optimal policies sometimes can be learned very quickly, but it's necessary to iterate many times to reach the optimal one. In real artificial systems, we always stop the learning process after a finite number of iterations, but it's almost impossible to find valid solutions if some experiences prevent the agent from continuing to interact with the environment. As many tasks have final states (either positive or negative), we assume that the agent can play any number of *episodes* (somewhat analogous to the epochs of supervised learning), exploiting the experience previously learned.

The second assumption is a little bit more technical and it's usually known as the *Markov property*. When the agent interacts with the environment, it observes a sequence of states. Even if it can seem like an oxymoron, we assume that each state is stateful. We can explain this concept with a simple example; suppose that you're filling a tank and every five seconds you measure the level. Imagine that at $t = 0$, the level $L = 10$ and the water is flowing in. What do you expect at $t = 1$? Obviously, $L > 10$. In other words, without external unknown causes, we assume that a state contains the previous history, so that the sequence, even if discretized, represents a continuous evolution where no jumps are allowed. When an RL task satisfies this property, it's called a Markov Decision Process and it's very easy to employ simple algorithms to evaluate the actions. Luckily, the majority of natural events can be modeled as MDPs (when you're walking toward a door, every step in the right direction must decrease the distance), but there are some games that are implicitly stateless. For example, if you want to employ an RL algorithm to learn how to guess the outcome of a probabilistic sequence of independent events (such as tossing a coin), the result could be dramatically wrong. The reason is clear: any state is independent of the previous ones and every attempt to build up a history is a failure. Therefore, if you observe a sequence of *0, 0, 0, 0, ...* you are not justified in increasing the value of betting on 0 unless, after considering the likelihood of the events, you suppose that the coin is loaded. However, if there's no reason to do so, the process isn't an MDP and every episode (event) is completely independent. All the assumptions that we, either implicitly or explicitly, make are based on this fundamental concept, so pay attention when evaluating new, unusual scenarios because you may discover that the employment of a specific algorithm isn't theoretically justified.

Environment

The **environment** is the entity where the **agent** has to reach its goals. For our purposes, a generic environment is a system that receives an input action, a_t (we use the index t because this is a natural time process), and outputs a tuple composed by a state, s_{t+1}, and a reward, r_{t+1}. These two elements are the only pieces of information provided to the agent to make its next decision. If we are working with an MDP and the sets of possible actions, A, and states, S, are discrete and finite, the problem is a defined finite MDP (in many continuous cases, it's possible to treat the problem as a finite MDP by discretizing the spaces). If there are final states, the task is called *episodic* and, in general, the goal is to reach a positive final state in the shortest amount of time or maximize a score. The schema of the cyclic interaction between agent an environment is shown in the following diagram:



Agent-environment interaction schema

A very important feature of an environment is its internal nature. It can be either *deterministic* or *stochastic*. A deterministic environment is characterized by a function that associates each possible action, a_t, in a specific state, s_t, to a well-defined successor, s_{t+1}, with a precise reward, r_{t+1}:

$$s_{t+1}, r_{t+1} = f(s_t, a_t) \;\; where \;\; a_t \in A \;\; and \;\; s_t, s_{t+1} \in S$$

Conversely, a stochastic environment is characterized by a transition probability between the current state, s_t, and a set of possible successors, s^i_{t+1}, given an action, a_t:

$$T(s_t, s^i_{t+1}, a_t) = \begin{pmatrix} p(s_t, s^1_{t+1}, a_t) & \cdots & p(s_t, s^i_{t+1}, a_t) \end{pmatrix}$$

If a state, s_i, has a transitional probability, $T(s_i, s_i, a_t) = 1 \; \forall \; a_t \in A$, the state is defined as *absorbing*. In general, all ending states in episodic tasks are modeled as absorbing ones, to avoid any further transition. When an episode is not limited to a fixed number of steps, the only criterion to determine its end is to check whether the agent has reached an absorbing state.

As we don't know which state will be the successor, it's necessary to consider the expected value of all possible rewards considering the initial state, s_t, and the action, a_t:

$$E\left[r_{t+1}^{i}; s_t, a_t\right]$$

In general, it's easier to manage stochastic environments because they can be immediately converted into deterministic ones by setting all probabilities to zero except the one corresponding to the actual successor (for example, T(•) = (0, 0, ..., 1, ..., 0)). In the same way, the expected return can be set equal to r_{t+1}. The knowledge of T(•), as well as $E[r^{i}_{t+1}]$, is necessary to employ some specific algorithms, but it can become problematic when finding a suitable model for the environment requires an extremely complex analysis. In all those cases, model-free methods can be employed and, therefore, the environment is considered as a black-box, whose output at time, t (subsequent to an action performed by the agent, a_{t-1}), is the only available piece of information for the evaluation of a policy.

Rewards

We have seen that rewards (sometimes negative rewards are called *penalties*, but it's preferable to use a standardized notation) are the only feedback provided by the environment after each action. However, there are two different approaches to the use of rewards. The first one is the strategy of a very short-sighted agent and consists in taking into account only the reward just received. The main problem with this approach is clearly the inability to consider longer sequences that can lead to a very high reward. For example, an agent has to traverse a few states with negative reward (for example, -0.1), but after them, they arrive at a state with a very positive reward (for example, +5.0). A short-sighted agent couldn't find out the best policy because it will simply try to avoid the immediate negative rewards. On the other side, it's better to suppose that a single reward contains a part of the future rewards that will be obtained following the same policy. This concept can be expressed by introducing a *discounted reward*, which is defined as:

$$R_t = \sum_{i=0}^{\infty} \gamma^i r_{i+t+1} = r_{t+1} + \gamma r_{t+2} + \ldots + \gamma^k r_{k+t+1} + \ldots$$

In the previous expression, we are assuming an infinite horizon with a discount factor, γ, which is a real number bounded between 0 and 1 (not included). When γ = 0, the agent is extremely short-sighted, because of $R_t = r_{t+1}$, but when $\gamma \to 1$, the current reward takes into account the future contributions discounted in a way that is inversely proportional to the time-step. In this way, very close rewards will have a higher weight than very distant ones. If the absolute value of all rewards is limited by a maximum immediate absolute reward, $|r_i| \leq |r_{max}|$, the previous expression will be always bounded. In fact, considering the properties of a geometric series, we get:

$$|R_t| = \left|\sum_{i=0}^{\infty} \gamma^i r_{i+t+1}\right| \leqslant \sum_{i=0}^{\infty} \gamma^i \left|r_{i+t+1}\right| \leq |r_{max}| \sum_{i=0}^{\infty} \gamma^i = \frac{|r_{max}|}{1-\gamma}$$

Clearly, the right choice of γ is a crucial factor in many problems and cannot be easily generalized. As in many other similar cases, I suggest testing different values, picking the one that minimizes the convergence speed while yielding a quasi-optimal policy. Of course, if the tasks are episodic with length, $T(e_i)$, the discounted reward becomes:

$$R_t = \sum_{i=0}^{T(e_i)-t-1} \gamma^i r_{i+t+1} = r_{t+1} + \gamma r_{t+2} + \ldots + \gamma^k r_{k+t+1} + \ldots + \gamma^{T(e_i)} r_{T(e_i)+t+1}$$

Checkerboard environment in Python

We are going to consider an example based on a checkerboard environment representing a tunnel. The goal of the agent is to reach the ending state (lower-right corner), avoiding 10 wells that are negative absorbing states. The rewards are:

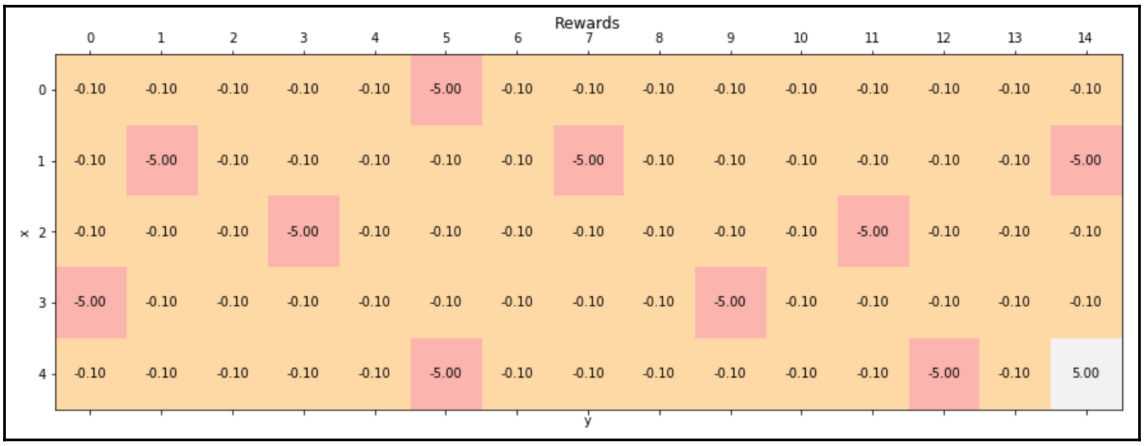
- **Ending state**: +5.0
- **Wells**: -5.0
- **All other states**: -0.1

Selecting a small negative reward for all non-terminal states is helpful to force the agent to move forward until the maximum (final) reward has been achieved. Let's start modeling an environment that has a 5 × 15 matrix:

```
import numpy as np

width = 15
```

```
height = 5

y_final = width - 1
x_final = height - 1

y_wells = [0, 1, 3, 5, 5, 7, 9, 11, 12, 14]
x_wells = [3, 1, 2, 0, 4, 1, 3, 2, 4, 1]

standard_reward = -0.1
tunnel_rewards = np.ones(shape=(height, width)) * standard_reward

for x_well, y_well in zip(x_wells, y_wells):
    tunnel_rewards[x_well, y_well] = -5.0

tunnel_rewards[x_final, y_final] = 5.0
```

The graphical representation of the environment (in terms of rewards) is shown in the following chart:



Rewards in the tunnel environment

The agent is allowed to move in four directions: up, down, left, and right. Clearly, in this case, the environment is deterministic because every action moves the agent to a predefined cell. We assume that whenever an action is forbidden (such as trying to move on the left when the agent is in the first column), the successor state is the same one (with the corresponding reward).

Policy

A *policy* is formally a deterministic or stochastic law that the agent follows in order to maximize its return. Conventionally, all policies are denoted with the letter π. A *deterministic policy* is usually a function of the current state that outputs a precise action:

$$a_{t+1} = \pi(s_t)$$

A *stochastic policy*, analogously to environments, outputs the probability of each action (in this case, we are assuming we work with a finite MPD):

$$\pi(s_t) = \begin{pmatrix} p(a_{t+1} = a^{(1)}) & \cdots & p(a_{t+1} = a^{(n)}) \end{pmatrix}$$

However, contrary to the environment, an agent must always pick a specific action, transforming any stochastic policy into a deterministic sequence of choices. In general, a policy where $\pi(s, a) > 0 \; \forall \, a \in A$, is called *soft* and it's often very useful during the training process because it allows a more flexible modeling without the premature selection of a suboptimal action. Instead, when $\pi(s, a_i) = 0 \; \forall \, i \neq j$ and $\pi(s, a_j) = 1$, the policy is also defined as *hard*. This transformation can be performed in many ways, but the most common one is to define a policy that is greedy with respect to a value (we're going to discuss this concept in the next section). This means that, at every step, the policy will select the action that maximizes the value of the successor state. Obviously, this is a very rational approach, which could be too pragmatic. In fact, when the values of some states don't change, a greedy policy will always force the agent to perform the same actions.

Such a problem is known as the *exploration-exploitation dilemma* and arises when it would be better to allow the agent to evaluate alternative strategies that could appear initially to be suboptimal. In other words, we want the agent to explore the environment before starting to exploit the policy, to know whether the policy is really the best one or if there are hidden alternatives. To solve this problem, it's possible to employ an *ε-greedy policy*, where the value, ε, is called the *exploration factor* and represents a probability. In this case, the policy will pick a random action with probability ε and a greedy one with probability $1 - \varepsilon$. In general, at the beginning of the training process, ε is kept very close to 1.0 to incentivize the exploration and it's progressively decreased when the policy becomes more stable. In many Deep RL applications, this approach is fundamental, in particular, when there are no models of the environment. The reason is that greedy policies can be initially wrong and it's necessary to allow the agent to explore many possible state and action sequences before forcing a deterministic decision.

Policy iteration

In this section, we are going to analyze a strategy to find an optimal policy based on a complete knowledge of the environment (in terms of transition probability and expected returns). The first step is to define a method that can be employed to build a greedy policy. Let's suppose we're working with a finite MDP and a generic policy, π; we can define the intrinsic value of a state, s_t, as the expected discounted return obtained by the agent starting from s_t and following the stochastic policy, π:

$$V(s_t;\pi) = E_\pi\left[R_t; s_t\right] = E_\pi\left[\sum_{i=0}^{\infty}\gamma^i r_{i+t+1}; s_t\right]$$

In this case, we are assuming that, as the agent will follow π, state s_a is more useful than s_b if the expected return starting from s_a is greater than the one obtained starting from s_b. Unfortunately, trying to directly find the value of each state using the previous definition is almost impossible when $\gamma > 0$. However, this a problem that can be solved using Dynamic Programming (for further information, please refer to *Dynamic Programming and Markov Process*, *Ronald A. Howard*, The MIT Press), which allows us to solve the problem iteratively.

In particular, we need to turn the previous formula into a *Bellman equation*:

$$V(s_t;\pi) = E_\pi\left[\sum_{i=0}^{\infty}\gamma^i r_{i+t+1}; s_t\right] = E_\pi\left[r_{t+1} + \gamma\sum_{i=0}^{\infty}\gamma^i r_{i+t+2}; s_t\right] = E_\pi\left[r_{t+1}; s_t\right] + \gamma E_\pi\left[\sum_{i=0}^{\infty}\gamma^i r_{i+t+2}; s_t\right]$$

The first term on the right-hand side can be expressed as:

$$E_\pi\left[r_{t+1}; s_t\right] = \sum_{a_k}\pi(s_t)\sum_{s_k}T(s_t, s_k; a_k)\; E\left[r_{t+1}; s_k, a_k\right]$$

In other words, it is the weighted average of all expected returns considering that the agent is state, s_t, and evaluates all possible actions and the consequent state transitions. For the second term, we need a small trick. Let's suppose we start from s_{t+1}, so that the expected value corresponds to $V(s_{t+1};\pi)$; however, as the sum starts from s_t, we need to consider all possible transitions starting from s_t. In this case, we can rewrite the term as:

$$E_\pi\left[\sum_{i=0}^{\infty}\gamma^i r_{i+t+2}; s_t\right] = \sum_{a_k}\pi(s_t)\sum_{s_k}T(s_t; s_k; a_k)\; V(s_k;\pi)$$

Again, the first terms take into account all possible transitions starting from s_t (and ending in s_{t+1}), while the second one is the value of each ending state. Therefore the complete expression becomes:

$$V(s_t;\pi) = \sum_{a_k} \pi(s_t) \sum_{s_k} T(s_t;s_k;a_k) \left[E\left[r_{t+1};s_k,a_k\right] + \gamma V(s_k;\pi) \right]$$

For a deterministic policy, instead, the formula is:

$$V(s_t;\pi) = \sum_{s_k} T(s_t,s_k;\pi(s_t)) \left[E\left[r_{t+1};s_k,\pi(s_t)\right] + \gamma V(s_k;\pi) \right]$$

The previous equations are particular cases of a generic discrete *Bellman equation* for a finite MDP that can be expressed as a vectorial operator, L_π, applied to the value vector:

$$L_\pi V = R + \gamma T V$$

It's easy to prove that there exists a unique fixed point that corresponds to $V(s;\pi)$, so $L\pi\ V(s;\pi) = V(s;\pi)$. However, in order to solve the system, we need to consider all equations at the same time because, both on the left-hand and on the right-hand side of the *Bellman equation*, there is the $V(\bullet;\pi)$ term. Is it possible to transform the problem into an iterative procedure, so that a previous computation can be exploited for the following one? The answer is yes and it's the consequence of an important property of L_π. Let's consider the infinity norm of the difference between two value vectors computed at time *t* and *t+1*:

$$\left\| L_\pi V^{(t+1)} - L_\pi V^{(t)} \right\|_\infty = \left\| R + \gamma T V^{(t+1)} - R - \gamma T V^{(t)} \right\|_\infty = \gamma \left\| T V^{(t+1)} - T V^{(t)} \right\|_\infty \leqslant \gamma \left\| V^{(t+1)} - V^{(t)} \right\|_\infty$$

As the discount factor $\gamma \in [0, 1[$, the Bellman operator, L_π, is a γ-contraction that reduces the distance between the arguments by a factor of γ (they get more and more similar). The *Banach Fixed-Point Theorem* states that a contraction, $L: D \rightarrow D$, on a metric space, D, admits a unique fixed point, $d^* \in D$, that can be found by repeatedly applying the contraction to any $d^{(0)} \in D$.

Hence, we know about the existence of a unique fixed point, $V(s;\pi)$, that is the goal of our research. If we now consider a generic starting point, $V^{(t)}$, and we compute the norm of the difference with $V(s;\pi)$, we obtain:

$$\left\|V^{(t)} - V(s;\pi)\right\|_\infty = \left\|L_\pi V^{(t-1)} - L_\pi V(s;\pi)\right\|_\infty \leqslant \gamma\left\|V^{(t-1)} - V(s;\pi)\right\|_\infty$$

Repeating this procedure iteratively until $t = 0$, we get:

$$\gamma\left\|V^{(t-1)} - V(s;\pi)\right\|_\infty \leqslant \gamma^2\left\|V^{(t-2)} - V(s;\pi)\right\|_\infty \leqslant \ldots \leqslant \gamma^{t+1}\left\|V^{(0)} - V(s;\pi)\right\|_\infty$$

The term $\gamma^{t+1} \rightarrow 0$, while continuing the iterations over the distance between $V^{(t)}$ and $V(s;\pi)$, gets smaller and smaller, authorizing us to employ the iterative approach instead of the one-shot closed method. Hence, the *Bellman equation* becomes:

$$V^{(i+1)}(s_t) = \sum_{a_k}\pi(s_t)\sum_{s_k}T(s_t, s_k; a_k)\left[E\left[r_{t+1}; s_k, a_k\right] + \gamma V^{(i)}(s_k)\right]$$

This formula allows us to find the value for each state (the step is formally called *policy evaluation*), but, of course, it requires a policy. At the first step, we can randomly select the actions because we don't have any other piece of information, but after a complete evaluation cycle, we can start defining a greedy policy with respect to the values. In order to achieve this goal, we need to introduce a very important concept in RL, the *Q function* (which must not be confused with the *Q function* defined in the EM algorithm), which is defined as the expected discounted return obtained by an agent starting from the state, s_t, and selecting a specific action, a_t:

$$Q(s_t, a_t;\pi) = E_\pi\left[R_t; s_t, a_t\right] = E_\pi\left[\sum_{i=0}^{\infty}\gamma^i r_{i+t+1}; s_t, a_t\right]$$

The definition is very similar to $V(s;\pi)$, but, in this case, we include the action, a_t, as a variable. Clearly, it's possible to define a Bellman equation for $Q(s, a;\pi)$ by simply removing the policy/action summation:

$$Q(s_t, a_t;\pi) = \sum_{s_k}T(s_t, s_k; a_t)\left[E\left[r_{t+1}; s_k, a_t\right] + \gamma V(s_k;\pi)\right]$$

Sutton and Barto (in *Reinforcement Learning*, *Sutton R. S., Barto A. G.*, The MIT Press) proved a simple but very important theorem (called the *Policy improvement theorem*), which states that given the deterministic policies, π_1 and π_2, if $Q(s, \pi_2(s); \pi_2) \geq V(s; \pi_1) \; \forall \; s \in S$, then π_2 is better than or equal to π_1. The proof is very compact and can be found in their book, however, the result can be understood intuitively. If we consider a sequence of states, $s_1 \rightarrow s_2 \rightarrow ... \rightarrow s_n$ and $\pi_2(s_i) = \pi_1(s_i) \; \forall \; i < m < n$, while $\pi_2(s_i) \geq \pi_1(s_i) \; \forall \; i \geq m$, the policy, π_2, is at least equal to π_1 and it's become better if at least an inequality is strict. Conversely, if $Q(s, \pi_2(s); \pi_2) \geq V(s; \pi_1)$, this means that $\pi_2(s) \geq \pi_1(s)$ and, again, $Q(s, \pi_2(s); \pi_2) > V(s; \pi_1)$ if there's at least a state, s_i, where $\pi_2(s_i) > \pi_1(s_i)$. Hence, after a complete policy evaluation cycle, we are authorized to define a new greedy policy as:

$$\pi^{(k+1)}(s_t) = argmax_{a_t} Q(s_t, a_t; \pi^{(k)})$$

This step is called *policy improvement* and its goal is to set the action associated with each state as the one that leads to the transition to the successor state with the maximum value. It's not difficult to understand that an optimal policy will remain stable when $V^{(t)} \rightarrow V(s; \pi)$. In fact, when $t \rightarrow \infty$, the Q function will converge to a stable fixed point determined by $V(s; \pi)$ and the argmax(•) will always select the same actions. However, if we start with a random policy, in general, a single policy evaluation cycle isn't enough to assure the convergence. Therefore, after a policy improvement step, it's often necessary to repeat the evaluation and continue alternating the two phases until the policy becomes stable (that's why the algorithm is called policy iteration). In general, the convergence is quite fast, but the actual speed depends on the nature of the problem, the number of states and actions, and the consistency of the rewards.

The complete policy iteration algorithm (as proposed by Sutton and Barto) is:

1. Set an initial deterministic random policy $\pi(s)$
2. Set the initial value array $V(s) = 0 \; \forall \; s \in S$
3. Set a tolerance threshold Thr (for example, Thr = 0.0001)
4. Set a maximum number of iterations N_{iter}
5. Set a counter $e = 0$

1. While $e < N_{iter}$:
    1. e += 1
    2. Do:
        1. Set $V_{old}(s) = V(s) \; \forall \; s \in S$
        2. Perform a Policy Evaluation step reading the current value from $V_{old}(s)$ and updating $V(s)$
    3. While $Avg(|V(s) - V_{old}(s)|) > Thr$
    4. Set $\pi_{old}(s) = \pi(s) \; \forall \; s \in S$
    5. Perform a policy improvement step
    6. If $\pi_{old}(s) == \pi(s)$:
        1. Break
2. Output the final deterministic policy $\pi(s)$

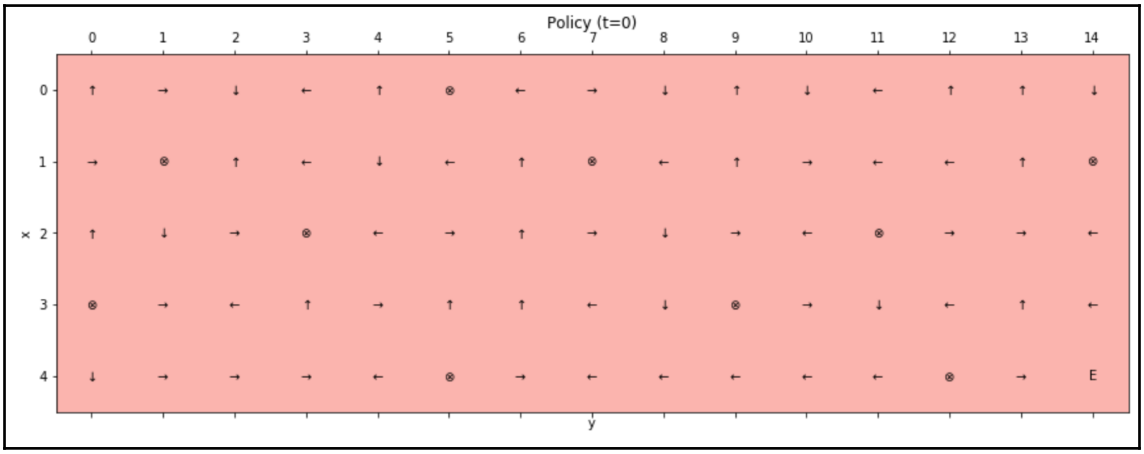
In this case, as we have a full knowledge of the environment, there's no need for an exploration phase. The policy is always exploited as it's built to be greedy to the real value (obtained when $t \rightarrow \infty$).

Policy iteration in the checkerboard environment

We want to apply the policy iteration algorithm in order to find an optimal policy for the tunnel environment. Let's start by defining a random initial policy and a value matrix with all values (except the terminal states) equal to 0:

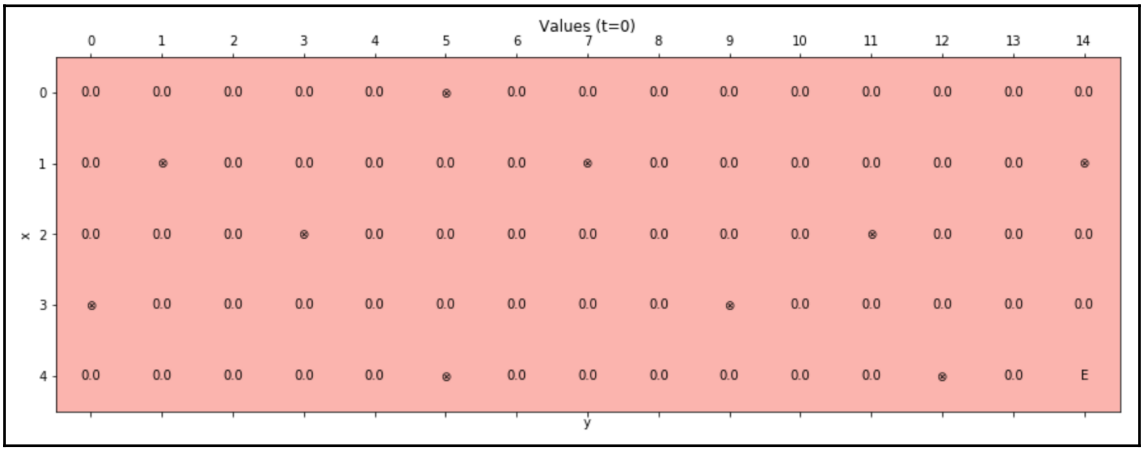
```
import numpy as np

nb_actions = 4

policy = np.random.randint(0, nb_actions, size=(height,
width)).astype(np.uint8)
tunnel_values = np.zeros(shape=(height, width))
```

The initial random policy (t=0) is shown in the following chart:



Initial (t=0) random policy

The states denoted with ⊗ represent the wells, while the final positive one is represented by the capital letter *E*. Hence, the initial value matrix (t=0) is:



Initial (t=0) value matrix

At this point, we need to define the functions to perform the policy evaluation and improvement steps. As the environment is deterministic, the processes are slightly simpler because the generic transition probability, $T(s_i, s_j; a_k)$, is equal to 1 for the only possible successor and 0 otherwise. In the same way, the policy is deterministic and only a single action is taken into account. The policy evaluation step is performed, freezing the current values and updating the whole matrix, $V^{(t+1)}$, with $V^{(t)}$; however, it's also possible to use the new values immediately. I invite the reader to test both strategies in order to find the fastest way. In this example, we are employing a discount factor, $\gamma = 0.9$ (it goes without saying that an interesting exercise consists of testing different values and comparing the result of the evaluation process and the final behavior):

```
import numpy as np

gamma = 0.9

def policy_evaluation():
    old_tunnel_values = tunnel_values.copy()
    for i in range(height):
        for j in range(width):
            action = policy[i, j]
            if action == 0:
                if i == 0:
                    x = 0
                else:
                    x = i - 1
                y = j
            elif action == 1:
                if j == width - 1:
                    y = width - 1
                else:
                    y = j + 1
                x = i
            elif action == 2:
                if i == height - 1:
                    x = height - 1
                else:
                    x = i + 1
                y = j
            else:
                if j == 0:
                    y = 0
                else:
                    y = j - 1
                x = i
            reward = tunnel_rewards[x, y]
            tunnel_values[i, j] = reward + (gamma * old_tunnel_values[x,
```

```
y])

def is_final(x, y):
    if (x, y) in zip(x_wells, y_wells) or (x, y) == (x_final, y_final):
        return True
    return False

def policy_improvement():
    for i in range(height):
        for j in range(width):
            if is_final(i, j):
                continue
            values = np.zeros(shape=(nb_actions, ))
            values[0] = (tunnel_rewards[i - 1, j] + (gamma *
tunnel_values[i - 1, j])) if i > 0 else -np.inf
            values[1] = (tunnel_rewards[i, j + 1] + (gamma *
tunnel_values[i, j + 1])) if j < width - 1 else -np.inf
            values[2] = (tunnel_rewards[i + 1, j] + (gamma *
tunnel_values[i + 1, j])) if i < height - 1 else -np.inf
            values[3] = (tunnel_rewards[i, j - 1] + (gamma *
tunnel_values[i, j - 1])) if j > 0 else -np.inf
            policy[i, j] = np.argmax(values).astype(np.uint8)
```

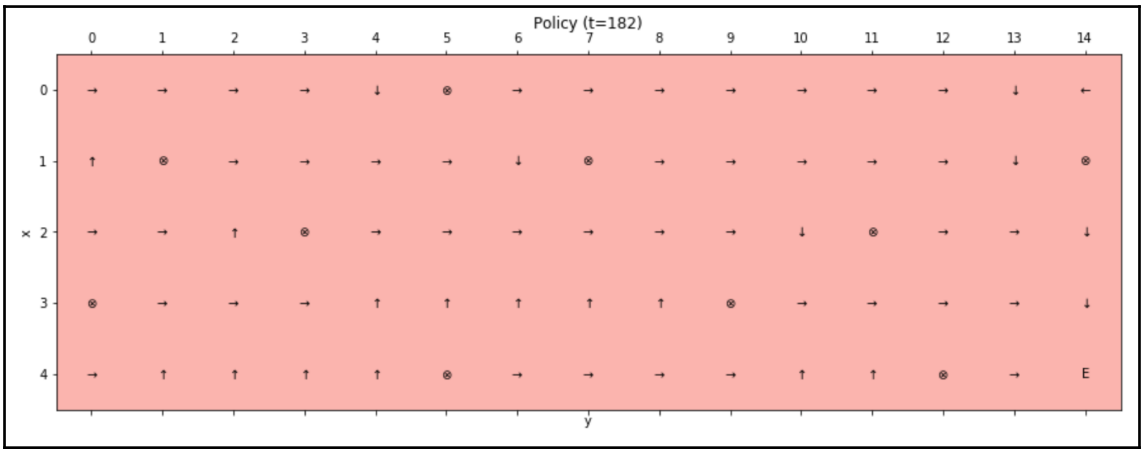
Once the functions have been defined, we start the policy iteration cycle (with a maximum number of epochs, N_{iter} = 100,000, and a tolerance threshold equal to 10^{-5}):

```
import numpy as np

nb_max_epochs = 100000
tolerance = 1e-5

e = 0

while e < nb_max_epochs:
    e += 1
    old_tunnel_values = tunnel_values.copy()
    policy_evaluation()
    if np.mean(np.abs(tunnel_values - old_tunnel_values)) < tolerance:
        old_policy = policy.copy()
        policy_improvement()

        if np.sum(policy - old_policy) == 0:
            break
```

At the end of the process (in this case, the algorithm converged after 182 iterations, but this value can change with different initial policies), the value matrix is:

Final value matrix

Analyzing the values, it's possible to see how the algorithm discovered that they are an implicit function of the distance between a cell and the ending state. Moreover, the policy always avoids the wells because the maximum value is always found in an adjacent state. It's easy to verify this behavior by plotting the final policy:



Final policy

Picking a random initial state, the agent will always reach the ending one, avoiding the wells and confirming the optimality of the policy iteration algorithm.

Value iteration

An alternative approach to policy iteration is provided by the *value iteration* algorithm. The main assumption is based on the empirical observation that the policy evaluation step converges rather quickly and it's reasonable to stop the process after a fixed number of steps (normally 1). In fact, policy iteration can be imagined like a game where the first player tries to find the correct values considering a stable policy, while the other one creates a new policy that is greedy with respect to the new values. Clearly, the second step compromises the validity of the previous evaluation, forcing the first player to repeat the process. However, as the Bellman equation uses a single fixed point, the algorithm converges to a solution characterized by the fact that the policy doesn't change anymore and, consequently, the evaluation becomes stable. This process can be simplified by removing the policy improvement step and continuing the evaluation in a greedy fashion. Formally, each step is based on the following update rule:

$$V^{(i+1)}(s_t) = max_{a_k} \sum_{s_k} T(s_t, s_k; a_k) \left[E\left[r_{t+1}; s_k, a_k \right] + \gamma V^{(i)}(s_k) \right]$$

Now the iteration doesn't consider the policy anymore (assuming implicitly that it will be greedy with respect to the values), and selects $V^{(t+1)}$ as the maximum possible value among all $V^{(t)}(a_t)$. In other words, value iteration anticipates the choice that is made by the policy improvement step by selecting the value that corresponds to the action that is likely ($p \rightarrow 1$) to be selected. It's not difficult to extend the convergence proof presented in the previous section to this case, therefore, $V^{(\infty)} \rightarrow V^{(opt)}$, as well as policy iteration does. However, the average number of iterations is normally smaller because we are starting with a random policy that can contrast the value iteration process.

When the values become stable, the optimal greedy policy is simply obtained as:

$$\pi^{(opt)}(s_t) = argmax_{a_t} Q^{(opt)}(s_t, a_t)$$

This step is formally equivalent to a policy improvement iteration, which, however, is done only once at the end of the process.

The complete value iteration algorithm (as proposed by Sutton and Barto) is:

1. Set the initial value array, $V(s) = 0 \; \forall \; s \in S$
2. Set a tolerance threshold, *Thr*, (for example, *Thr = 0.0001*)
3. Set a maximum number of iteration, N_{iter}
4. Set a counter, $e = 0$
5. While $e < N_{iter}$:
    1. e += *1*
    2. Do:
        1. Set $V_{old}(s) = V(s) \; \forall \; s \in S$
        2. Perform a value evaluation step reading the current value from $V_{old}(s)$ and updating $V(s)$
    3. While $Avg(|V(s) - V_{old}(s)|) > Thr$
6. Output the final deterministic policy $\pi(s) = argmax_a \; Q(s, a)$

Value iteration in the checkerboard environment

To test this algorithm, we need to set an initial value matrix with all values equal to 0 (they can be also randomly chosen but, as we don't have any prior information on the final configuration, every initial choice is probabilistically equivalent):

```
import numpy as np

tunnel_values = np.zeros(shape=(height, width))
```

At this point, we can define the two functions to perform the value evaluation and the final policy selection (the function `is_final()` is the one defined in the previous example):

```
import numpy as np

def value_evaluation():
    old_tunnel_values = tunnel_values.copy()
    for i in range(height):
        for j in range(width):
            rewards = np.zeros(shape=(nb_actions, ))
            old_values = np.zeros(shape=(nb_actions, ))
            for k in range(nb_actions):
                if k == 0:
                    if i == 0:
                        x = 0
                    else:
```

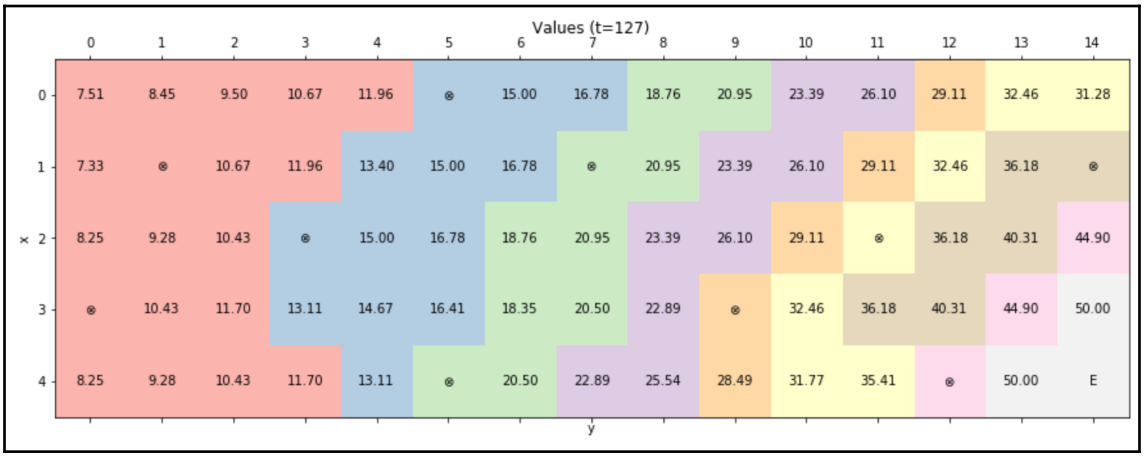
```
                        x = i - 1
                    y = j

                elif k == 1:
                    if j == width - 1:
                        y = width - 1
                    else:
                        y = j + 1
                    x = i

                elif k == 2:
                    if i == height - 1:
                        x = height - 1
                    else:
                        x = i + 1
                    y = j

                else:
                    if j == 0:
                        y = 0
                    else:
                        y = j - 1
                    x = i
                rewards[k] = tunnel_rewards[x, y]
                old_values[k] = old_tunnel_values[x, y]
            new_values = np.zeros(shape=(nb_actions, ))
            for k in range(nb_actions):
                new_values[k] = rewards[k] + (gamma * old_values[k])
            tunnel_values[i, j] = np.max(new_values)

def policy_selection():
    policy = np.zeros(shape=(height, width)).astype(np.uint8)
    for i in range(height):
        for j in range(width):
            if is_final(i, j):
                continue
            values = np.zeros(shape=(nb_actions, ))
            values[0] = (tunnel_rewards[i - 1, j] + (gamma *
tunnel_values[i - 1, j])) if i > 0 else -np.inf
            values[1] = (tunnel_rewards[i, j + 1] + (gamma *
tunnel_values[i, j + 1])) if j < width - 1 else -np.inf
            values[2] = (tunnel_rewards[i + 1, j] + (gamma *
tunnel_values[i + 1, j])) if i < height - 1 else -np.inf
            values[3] = (tunnel_rewards[i, j - 1] + (gamma *
tunnel_values[i, j - 1])) if j > 0 else -np.inf
            policy[i, j] = np.argmax(values).astype(np.uint8)
    return policy
```

The main differences are in the `value_evaluation()` function, which now has to consider all possible successor states and select the value corresponding to the action that leads to the state with the highest value. Instead, the `policy_selection()` function is equivalent to `policy_improvement()`, but, as it is invoked only once, it outputs directly to the final optimal policy.

At this point, we can run a training cycle (assuming the same constants as before):

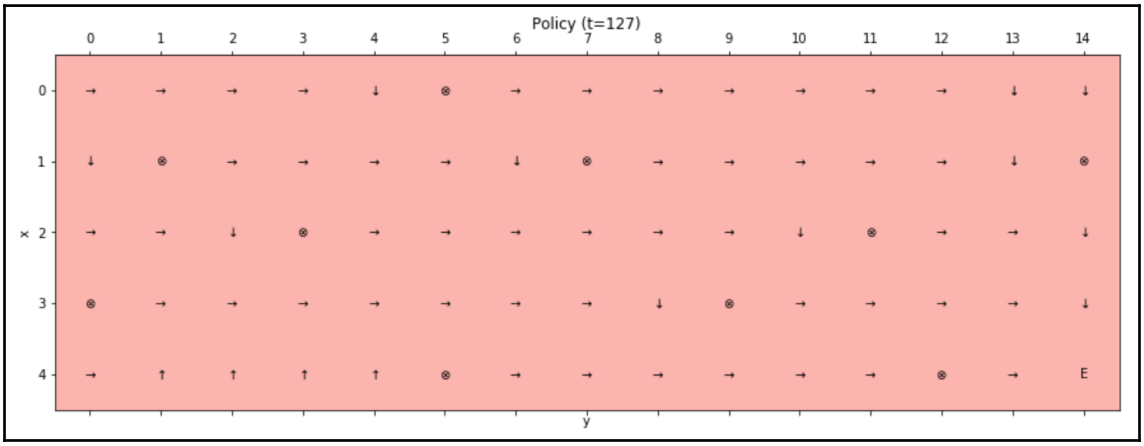
```
import numpy as np

e = 0

policy = None

while e < nb_max_epochs:
    e += 1
    old_tunnel_values = tunnel_values.copy()
    value_evaluation()
    if np.mean(np.abs(tunnel_values - old_tunnel_values)) < tolerance:
        policy = policy_selection()
        break
```

The final value configuration (after 127 iterations) is shown in the following chart:



Final value matrix

As in the previous example, the final value configuration is a function of the distance between each state and the ending one, but, in this case, the choice of $\gamma = 0.9$ isn't optimal. In fact, the wells close to the final state aren't considered very dangerous anymore. Plotting the final policy can help us understand the behavior:



Final policy

As expected, the wells that are far from the target are avoided, but the two that are close to the final state are accepted as reasonable penalties. This happens because the value iteration algorithm is very greedy with respect to the value and the discount factor, $\gamma < 1.0$; the effect of negative states can be compensated for by the final reward. In many scenarios, these states are absorbing, therefore their implicit reward is $+\infty$ or $-\infty$, meaning that no other actions can change the final value. I invite the reader to repeat the example with different discount factors (remember that an agent with $\gamma \to 1$ is very short-sighted and will avoid any obstacle, even reducing the efficiency of the policy) and change the values of the final states. Moreover, the reader should be able to answer the question: What is the agent's behavior when the standard reward (whose default value is *-0.1*) is increased or decreased?

TD(0) algorithm

One of the problems with Dynamic Programming algorithms is the need for a full knowledge of the environment in terms of states and transition probabilities. Unfortunately, there are many cases where these pieces of information are unknown before the direct experience. In particular, the states can be discovered by letting the agent explore the environment, but the transition probabilities require us to count the number of transitions to a certain state and this is often impossible.

Moreover, an environment with absorbing states can prevent visiting many states if the agent has learned a good initial policy. For example, in a game, which can be described as an episodic MDP, the agent discovers the environment while learning how to move forward without ending in a negative absorbing state.

A general solution to these problems is provided by a different evaluation strategy, called **Temporal Difference** (**TD**) RL. In this case, we start with an empty value matrix and we let the agent follow a greedy policy with respect to the value (but the initial one, which is generally random). Once the agent observes a transition, $s_i \rightarrow s_j$, due to an action, a_i, with a reward, r_{ij}, it updates the estimation of $V(s_i)$. The process is structured in episodes (which is the most natural way) and ends when a maximum number of steps have been done or a terminal state is met. In particular, the TD(0) algorithm updates the value according to the rule:

$$V^{(t+1)}(s_i) = V^{(t)}(s_i) + \alpha \left(r_{ij} + \gamma V^{(t)}(s_j) - V^{(t)}(s_i) \right)$$

The constant, α, is bound between 0 and 1 and acts as a learning rate. Each update considers a variation with respect to the current value, $V^{(t)}(s_i)$, which is proportional to the difference between the actual return and the previous estimation. The term $r_{ij} + \gamma V^{(t)}(s_j)$ is analogous to the one employed in the previous methods and represents the expected value given the current return and the discounted value starting from the successor state. However, as $V^{(t)}(s_j)$ is an estimation, the process is based on a bootstrap from the previous values. In other words, we start from an estimation to determine the next one, which should be closer to the stable fixed point. Indeed, TD(0) is the simplest example of a family of TD algorithms that are based on a sequence (usually called backup) that can be generalized as (considering k steps):

$$R_t^k = r_{t+1} + \gamma^2 r_{t+2} + \ldots + \gamma^{k-1} r_{k-1} + \gamma^k V^{(t)}(s_{t+k})$$

As we're using a single reward to approximate the expected discounted return, TD(0) is usually called a one-step TD method (or one-step backup). A more complex algorithm can be built considering more subsequent rewards or alternative strategies. We're going to analyze a generic variant called TD(λ) in `Chapter 15`, *Advanced Policy Estimation Algorithms* and explain why this algorithm corresponds to a choice of $\lambda = 0$.

TD(0) has been proven to converge, even if the proof (which can be found for a model-based approach in *Convergence of Model-Based Temporal Difference Learning for Control*, *Van Hasselt H.*, *Wiering M. A.*, *Proceedings of the 2007 IEEE Symposium on Approximate Dynamic Programming and Reinforcement Learning* (ADPRL 2007)) is more complex because it's necessary to consider the evolution of the Markov Process. In fact, in this case, we are approximating the expected discounted return with both a truncated estimation and a bootstrap value, $V(s_j)$, which is initially (and for a large number of iterations) unstable. However, assuming the convergence for $t \to \infty$, we get:

$$V^{(\infty)}(s_i) = V^{(\infty)}(s_i) + \alpha\left(r_{ij} + \gamma V^{(\infty)}(s_j) - V^{(\infty)}(s_i)\right) \;\Rightarrow\; V^{(\infty)}(s_i) = r_{ij} + \gamma V^{(\infty)}(s_j)$$

The last formula expresses the value of the state, s_i, assuming that the greedy optimal policy forces the agent to perform the action that causes the transition to s_j. Of course, at this point, it's natural to ask under which conditions the algorithm converges. In fact, we are considering episodic tasks and the estimation, $V^{(\infty)}(s_i)$, can be correct only if the agent performs a transition to s_i an infinite number of times, selecting all possible actions an infinite number of times. Such a condition is often expressed by saying that the policy must be **Greedy in the Limit with Infinite Explorations** (**GLIE**). In other words, the real greediness is achieved only as an asymptotic state when the agent is able to explore the environment without limitations for an unlimited number of episodes.

This is probably the most important limitation of TD RL, because, in real-life scenarios, some states can be very unlikely and, hence, the estimation can never accumulate the experience needed to converge to the actual value. We are going to analyze some methods to solve this problem in `Chapter 15`, *Advanced Policy Estimation Algorithms*, but, in our example, we employ a random start. In other words, as the policy is greedy and could always avoid some states, we force the agent to start each episode in a random nonterminal cell. In this way, we allow a deep exploration even with a greedy policy. Whenever this approach is not feasible (because, for example, the environment dynamics are not controllable), the exploration-exploitation dilemma can be solved only by employing an ε-greedy policy, which selects a fraction of suboptimal (or even wrong) actions. In this way, it's possible to observe a higher number of transitions paying the price of a slower convergence.

However, as pointed out by Sutton and Barto, TD(0) converges to the maximum-likelihood estimation of the value function determined by the MDP, finding the implicit transition probabilities of the model. Therefore, if the number of observations is high enough, TD(0) can quickly find an optimal policy, but, at the same time, it's also more sensitive to biased estimations if some couple's state-action are never experienced (or experienced very seldom). In our example, we don't know which the initial state is, hence selecting a fixed starting point yields a policy that is extremely rigid and almost completely unable to manage noisy situations. For example, if the starting point is changed to an adjacent (but never explored) cell, the algorithm could fail to find the optimal path to the positive terminal state. On the other hand, if we know that the dynamics are well-defined, TD(0) will force the agent to select the actions that are most likely to produce the optimal result given the current knowledge of the environment. If the dynamics are partially stochastic, the advantage of an ε-greedy policy can be understood considering a sequence of episodes where the agent experiences the same transitions and the corresponding values are increased proportionally. If, for example, the environment changes one transition after many experiences, the agent has to face a brand new experience when the policy is already almost stable. The correction requires many episodes and, as this random change has a very low probability, it's possible that the agent will never learn the correct behavior. Instead, by selecting a few random actions, the probability of encountering a similar state (or even the same one) increases (think about a game where the state is represented by a screenshot) and the algorithm can become more robust with respect to very unlikely transitions.

The complete TD(0) algorithm is:

1. Set an initial deterministic random policy, $\pi(s)$
2. Set the initial value array, $V(s) = 0 \; \forall \, s \in S$
3. Set the number of episodes, $N_{episodes}$
4. Set a maximum number of steps per episode, N_{max}
5. Set a constant, α (for example, $\alpha = 0.1$)
6. Set a constant, γ (for example, $\gamma = 0.9$)
7. Set a counter, $e = 0$
8. For $i = 1$ to $N_{episodes}$:
    1. Observe the initial state, s_i
    2. While s_j is non-terminal and $e < N_{max}$:
        1. $e \mathrel{+}= 1$
        2. Select the action, $a_t = \pi(s_i)$

   2. Observe the transition, $(a_i, s_i) \rightarrow (s_j, r_{ij})$
   3. Update the value function for the state, s_i
   4. Set $s_i = s_j$
2. Update the policy to be greedy with respect to the value function, $\pi(s) = argmax_a Q(s, a)$

TD(0) in the checkerboard environment

At this point, we can test the TD(0) algorithm on the checkerboard environment. The first step is to define an initial random policy and a value matrix with all elements equal to 0:

```
import numpy as np

policy = np.random.randint(0, nb_actions, size=(height,
width)).astype(np.uint8)
tunnel_values = np.zeros(shape=(height, width))
```

As we want to select a random starting point at the beginning of each episode, we need to define a helper function that must exclude the terminal states (all the constants are the same as previously defined):

```
import numpy as np

xy_grid = np.meshgrid(np.arange(0, height), np.arange(0, width),
sparse=False)
xy_grid = np.array(xy_grid).T.reshape(-1, 2)

xy_final = list(zip(x_wells, y_wells))
xy_final.append([x_final, y_final])

xy_start = []

for x, y in xy_grid:
    if (x, y) not in xy_final:
        xy_start.append([x, y])
xy_start = np.array(xy_start)

def starting_point():
    xy = np.squeeze(xy_start[np.random.randint(0, xy_start.shape[0],
size=1)])
    return xy[0], xy[1]
```

Now we can implement the function to evaluate a single episode (setting the maximum number of steps equal to 500 and the constant to $\alpha = 0.25$):

```
max_steps = 1000
alpha = 0.25

def episode():
    (i, j) = starting_point()
    x = y = 0
    e = 0
    while e < max_steps:
        e += 1
        action = policy[i, j]
        if action == 0:
            if i == 0:
                x = 0
            else:
                x = i - 1
            y = j
        elif action == 1:
            if j == width - 1:
                y = width - 1
            else:
                y = j + 1
            x = i
        elif action == 2:
            if i == height - 1:
                x = height - 1
            else:
                x = i + 1
            y = j
        else:
            if j == 0:
                y = 0
            else:
                y = j - 1
            x = i
        reward = tunnel_rewards[x, y]
        tunnel_values[i, j] += alpha * (reward + (gamma * tunnel_values[x,
y]) - tunnel_values[i, j])
        if is_final(x, y):
            break
        else:
            i = x
            j = y
```

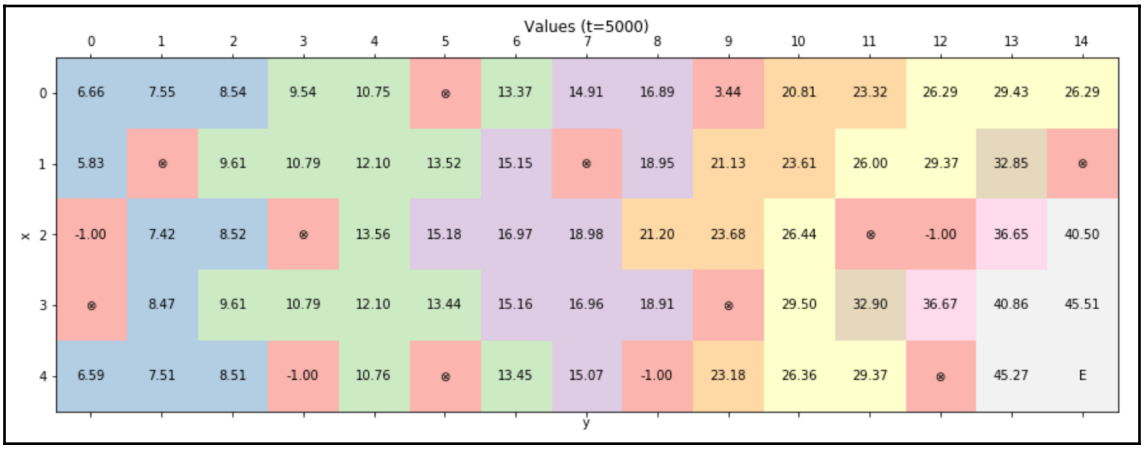
The function to determine the greedy policy with respect to the values is the same as already implemented in the previous examples; however, we report it to guarantee the consistency of the example:

```
def policy_selection():
    for i in range(height):
        for j in range(width):
            if is_final(i, j):
                continue
            values = np.zeros(shape=(nb_actions, ))
            values[0] = (tunnel_rewards[i - 1, j] + (gamma *
tunnel_values[i - 1, j])) if i > 0 else -np.inf
            values[1] = (tunnel_rewards[i, j + 1] + (gamma *
tunnel_values[i, j + 1])) if j < width - 1 else -np.inf
            values[2] = (tunnel_rewards[i + 1, j] + (gamma *
tunnel_values[i + 1, j])) if i < height - 1 else -np.inf
            values[3] = (tunnel_rewards[i, j - 1] + (gamma *
tunnel_values[i, j - 1])) if j > 0 else -np.inf
            policy[i, j] = np.argmax(values).astype(np.uint8)
```

At this point, we can start a training cycle with 5,000 episodes:

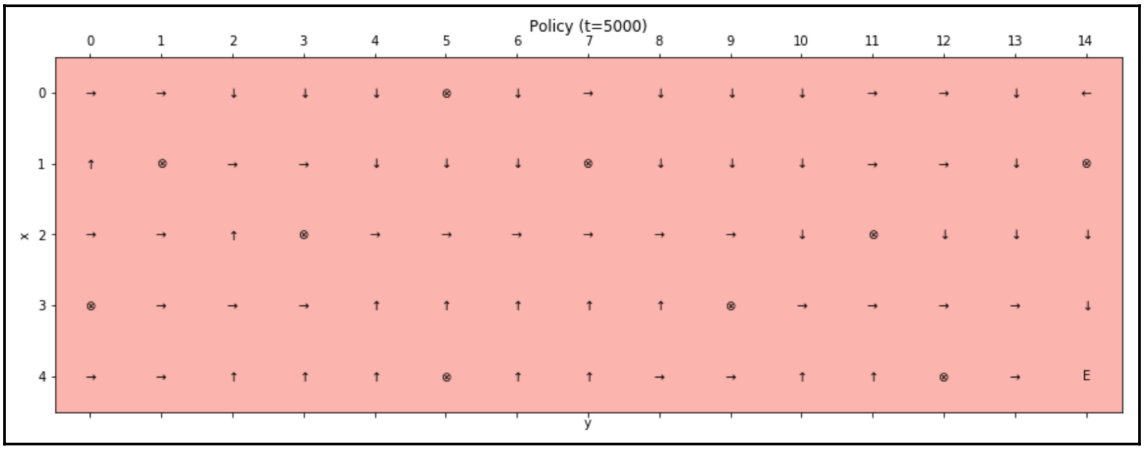
```
n_episodes = 5000

for _ in range(n_episodes):
    episode()
    policy_selection()
```

The final value matrix is shown in the following chart:



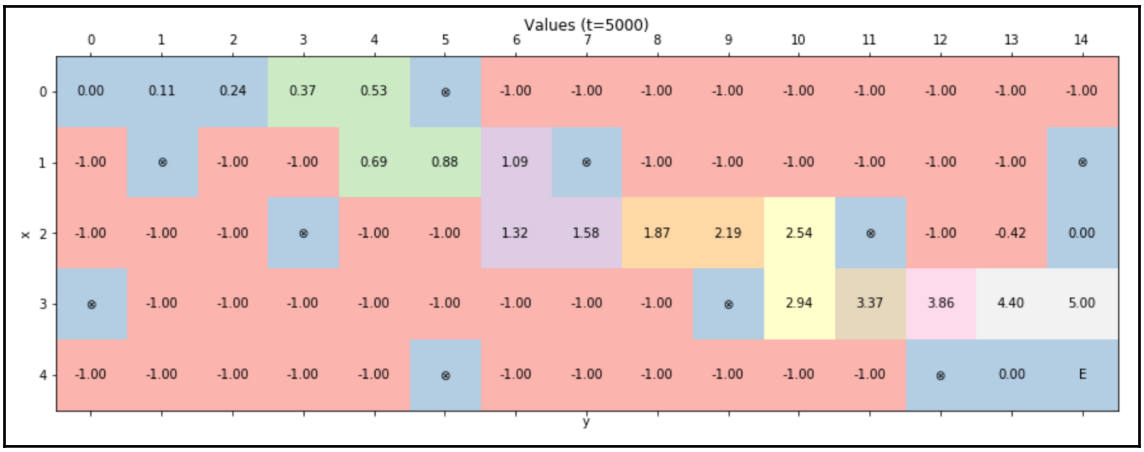
Final value matrix with random starts

Like in the previous examples, the final values are inversely proportional to the distance from the final positive state. Let's analyze the resulting policy to understand whether the algorithm converged to a consistent solution:



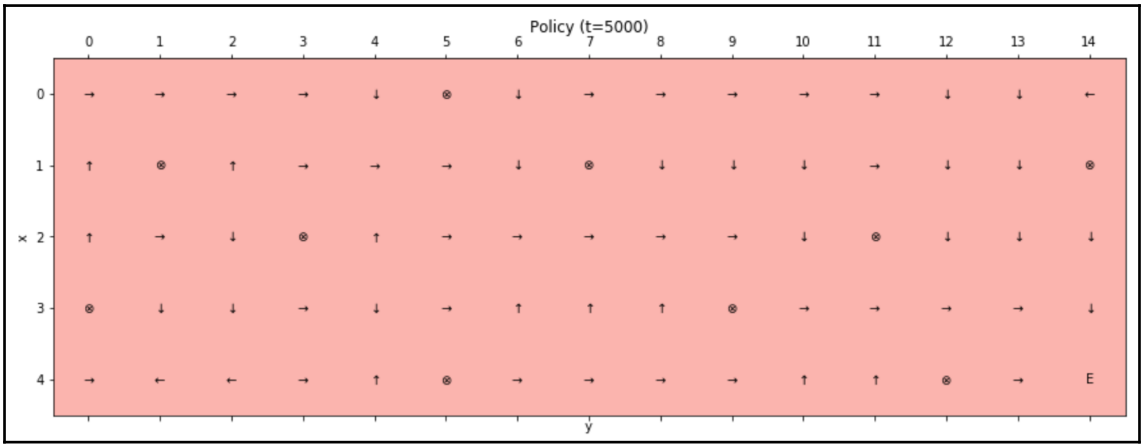
Final policy with random starts

As can be seen, the random choice of the starting state is allowed to find the best path independently from the initial condition. To better understand the advantage of this strategy, let's plot the final value matrix when the initial state is fixed to the cell *(0, 0)*, corresponding to the upper-left corner:



Final value matrix with a fixed initial state (0, 0)

Without any further analysis, it's possible to see that many states have never been visited or visited only a few times, and the resulting policy is therefore extremely greedy with respect to the specific initial state. The blocks containing values equal to -1.0 indicate states where the agent often has to pick a random action because there's no difference in the values, hence it can be extremely difficult to solve the environment with a different initial state. The resulting policy confirms this analysis:



Final policy with a fixed initial state (0, 0)

As it's possible to see, the agent is able to reach the final state only when the initial point allows us to cross the trajectory starting from *(0, 0)*. In all these cases, it's possible to recover the optimal policy, even if the paths longer than the ones obtained in the previous example. Instead, states such as *(0, 4)* are clearly situations where there's a *loss of policy*. In other words, the agent acts without any knowledge or awareness and the probability of success converges to 0. As an exercise, I invite the reader to test this algorithm with different starting points (for example, a set of fixed ones) and higher α values. The goal is also to answer these questions: Is it possible to speed up the learning process? Is it necessary to start from all possible states in order to obtain a global optimal policy?

Summary

In this chapter, we introduced the most important RL concepts, focusing on the mathematical structure of an environment as a Markov Decision Process, and on the different kinds of policy and how they can be derived from the expected reward obtained by an agent. In particular, we defined the value of a state as the expected future reward considering a sequence discounted by a factor, γ. In the same way, we introduced the concept of the Q function, which is the value of an action when the agent is in a specific state.

These concepts directly employed the policy iteration algorithm, which is based on a Dynamic Programming approach assuming complete knowledge of the environment. The task is split into two stages; during the first one, the agent evaluates all the states given the current policy, while in the second one, the policy is updated in order to be greedy with respect to the new value function. In this way, the agent is forced to always pick the action that leads to a transition that maximizes the obtained value.

We also analyzed a variant, called value iteration, that performs a single evaluation and selects the policy in a greedy manner. The main difference from the previous approach is that now the agent immediately selects the highest value assuming that the result of this process is equivalent to a policy iteration. Indeed, it's easy to prove that, after infinite transitions, both algorithms converge on the optimal value function.

The last algorithm is called TD(0) and it's based on a model-free approach. In fact, in many cases, it's difficult to know all the transition probabilities and, sometimes, even all possible states are unknown. This method is based on the Temporal Difference evaluation, which is performed directly while interacting with the environment. If the agent can visit all the states an infinite number of times (clearly, this is only a theoretical condition), the algorithm has been proven to converge to the optimal value function more quickly than other methods.

In the next chapter, `Chapter 15`, *Advanced Policy Estimation Algorithms* we'll continue the discussion of RL algorithms, introducing some more advanced methods that can be immediately implemented using Deep Convolutional Networks.

15
Advanced Policy Estimation Algorithms

In this chapter, we will continue our exploration of the world of **Reinforcement Learning** (**RL**), focusing our attention on complex algorithms that can be employed to solve difficult problems. As this is still the introductory part of RL (the whole topic is extremely large), the structure of the chapter is based on many practical examples that can be used as a basis to work on more complex scenarios.

The topics that will be discussed in this chapter are:

- TD(λ) algorithm
- Action-Critic TD(0)
- SARSA
- Q-learning
- Q-learning with a simple visual input and a neural network

TD(λ) algorithm

In the previous chapter, we introduced the temporal difference strategy, and we discussed a simple example called TD(0). In the case of TD(0), the discounted reward is approximated by using a one-step backup. Hence, if the agent performs an action a_t in the state s_t, and the transition to the state s_{t+1} is observed, the approximation becomes the following:

$$R_t^1 = r_{t+1} + \gamma V(s_{t+1})$$

If the task is episodic (as in many real-life scenarios) and has *T(e$_i$)* steps, the complete backup for the episode e_i is as follows:

$$R_t^{T(e_i)} = R_t = r_{t+1} + \gamma^2 r_{t+2} + \ldots + \gamma^{T(e_i)-1} r_{T(e_i)-1} + \gamma^{T(e_i)} V^{(t)}\left(s_{t+T(e_i)}\right)$$

The previous expression ends when the MDP process reaches an absorbing state; therefore, R_t is the actual value of the discounted reward. The difference between TD(0) and this choice is clear: in the first case, we can update the value function after each transition, whereas with a complete backup, we need to wait for the end of the episode. We can say that this method (which is called Monte Carlo, because it's based on the idea of averaging the overall reward of an entire sequence) is exactly the opposite of TD(0); therefore, it's reasonable to think about an intermediate solution, based on *k*-step backups. In particular, our goal is to find an online algorithm that can exploit the backups once they are available.

Let's imagine a sequence of four steps. The agent is in the first state and observes a transition; at this point, only a one-step backup is possible, and it's a good idea to update the value function in order to improve the convergence speed. After the second transition, the agent can use a two-step backup; however, it can also consider the first one-step backup in addition to the newer, longer one. So, we have two approximations:

$$\begin{cases} R_t^1 = r_{t+1} + \gamma V(s_{t+1}) \\ R_t^2 = r_{t+1} + \gamma r_{t+2} + \gamma^2 V(s_{t+2}) \end{cases}$$

Which of the preceding is the most reliable? Obviously, the second one depends on the first one (in particular, when the value function is almost stabilized), and so on until the end of the episode. Hence, the most common strategy is to employ a weighted average that assigns a different level of importance to each backup (assuming the longest backup has *k* steps):

$$\widetilde{R_t} = \lambda_1 R_t^1 + \lambda_2 R_t^2 + \ldots + \lambda_k R_t^k \quad and \quad \sum_i \lambda_i = 1$$

Watkins (in *Learning from Delayed Rewards*, *Watkins C.I.C.H.*, *Ph.D. Thesis*, *University of Cambridge*, *1989*) proved that this approach (with or without averaging) has the fundamental property of reducing the absolute error of the expected R_t^k, with respect to the optimal value function, *V(s; π)*. In fact, he proved that the following inequality holds:

$$max_{s^*} \left| E_\pi \left[R_t^k ; s^* \right] - V(s^*; \pi) \right| \leqslant \gamma^k max_s \left| V(s) - V(s; \pi) \right|$$

As γ is bounded between 0 and 1, the right-hand side is always smaller than the maximum absolute error *V(t) - V(s;π)*, where *V(s)* is the value of a state during an episode. Therefore, the expected discounted return of a *k*-step backup (or of a combination of different backups) yields a more accurate estimation of the optimal value function if the policy is chosen to be greedy with respect to it. This is not surprising, as a longer backup incorporates more actual returns, but the importance of this theorem resides in its validity when an average of different *k*-step backups are employed. In other words, it provides us with the mathematical proof that an intuitive approach actually converges, and it can also effectively improve both the convergence speed and the final accuracy.

However, managing *k* coefficients is generally problematic, and in many cases, useless. The main idea behind TD(λ) is to employ a single factor, λ, that can be tuned in order to meet specific requirements. The theoretical analysis (or *forward view*, as referred to by Sutton and Barto) is based, in a general case, on an exponentially decaying average. If we consider a geometric series with λ bounded between 0 and 1 (exclusive), we get:

$$\sum_{i=0}^{\infty}\lambda^i = \frac{1}{1-\lambda} \Rightarrow (1-\lambda)\sum_{i=0}^{\infty}\lambda^i = 1 \Rightarrow \sum_{i=0}^{\infty}(1-\lambda)\lambda^i = 1$$

Hence, we can consider the averaged discounted return $R_t^{(\lambda)}$ with infinite backups as:

$$R_t^{(\lambda)} = \sum_{i=1}^{\infty}(1-\lambda)\lambda^{i-1}R_t^i = (1-\lambda)\sum_{i=1}^{\infty}\lambda^{i-1}R_t^i$$

Before defining the finite case, it's helpful to understand how $R_t^{(\lambda)}$ was built. As λ is bounded between 0 and 1, the factors decay proportionally to λ, so the first backup has the highest impact, and all of the subsequent ones have smaller and smaller influences on the estimation. This means that, in general, we are assuming that the estimation of R_t has more importance to the *immediate* backups (which become more and more precise), and we exploit the longer ones only to improve the estimated value. Now, it should be clear that λ = 0 is equivalent to TD(0), because only the one-step backup remains in the sum (remember that $0^0 = 1$), while higher values involve all of the remaining backups. Let's now consider an episode e_i whose length is $T(e_i)$.

Conventionally, if the agent reached an absorbing state at $t = T(e_i)$, all of the remaining $t+i$ returns are equal to R_t (this is straightforward, as all of the possible rewards have already been collected); therefore, we can truncate $R_t^{(\lambda)}$:

$$R_t^{(\lambda)} = \sum_{i=1}^{T(e_i)-t-1} (1-\lambda)\lambda^{i-1} R_t^i + \lambda^{T(e_i)-t-1} R_t$$

The first term of the previous expression involves all of the non-terminal states, while the second is equal to R_t discounted proportionally to the distance between the first time step and the final state. Again, if $\lambda = 0$, we obtain TD(0), but we are now also authorized to consider $\lambda = 1$ (because the sum is always extended to a finite number of elements). When $\lambda = 1$, we obtain $R_t^{(\lambda)} = R_t$, which means that we need to wait until the end of the episode to get the actual discounted reward. As explained previously, this method is normally not a first-choice solution, because when the episodes are very long, the agent selects the actions with a value function that is not up to date in the majority of cases. Therefore, *TD(λ)* is normally employed with λ values less than 1, in order to obtain the advantage of an online update, together with a correction based on the new states. To achieve this goal without looking at the future (we want to update *V(s)* as soon as new pieces of information are available), we need to introduce the concept of *eligibility trace e(s)* (sometimes, in the context of computational neuroscience, *e(s)* is also called *stimulus trace*).

An eligibility trace for a state s is a function of time that returns the weight (greater than 0) of the specific state. Let's imagine a sequence, $s_1, s_2, ..., s_n$, and consider a state, s_i. After a backup $V(s_i)$ is updated, the agent continues its exploration. When is a new update of s_i (given longer backups) important? If s_i is not visited anymore, the effect of longer backups must be smaller and smaller, and s_i is said to not be eligible for changes in *V(s)*. This is a consequence of the previous assumption that shorter backups must generally have higher importance. So, if s_i is an initial state (or is immediately after the initial state) and the agent moves to other states, the effect of s_i must decay. Conversely, if s_i is revisited, it means that the previous estimation of $V(s_i)$ is probably wrong, and hence s_i is eligible for a change. (To better understand this concept, imagine a sequence, $s_1, s_2, s_1, ...$ It's clear that when the agent is in s_1, as well as in s_2, it cannot select the right action; therefore, it's necessary to reevaluate *V(s)* until the agent is able to move forward.)

The most common strategy (which is also discussed in *Reinforcement Learning*, *Sutton R. S.*, *Barto A. G.*, *The MIT Press*) is to define the eligibility traces in a recursive fashion. After each time step, $e_t(s)$ decays by a factor equal to $\gamma\lambda$ (to meet the requirement imposed by the forward view); but, when the state s is revisited, $e_t(s)$ is also increased by 1 ($e_t(s) = \gamma\lambda e_{t-1}(s) + 1$). In this way, we impose a jump in the trend of $e(s)$ whenever we desire to emphasize its impact. However, as $e(s)$ decays independently of the jumps, the states that are visited and revisited later have a lower impact than the ones that are revisited very soon. The reason for this choice is very intuitive: the importance of a state revisited after a long sequence is clearly lower than the importance of a state that is revisited after a few steps. In fact, the estimation of R_t is obviously wrong if the agent moves back and forth between two states at the beginning of the episode, but the error becomes less significant when the agent revisits a state after having explored other areas. For example, a policy can allow an initial phase in order to reach a partial goal, and then it can force the agent to move back to reach a terminal state.

Exploiting the eligibility traces, $TD(\lambda)$ can achieve a very fast convergence in more complex environments, with a trade-off between a one-step TD method and a Monte Carlo one (which is normally avoided). At this point, the reader might wonder if we are sure about the convergence, and luckily, the answer is positive. Dayan proved (in *The convergence of TD (λ) for General λ*, *Dayan P.*, *Machine Learning 8*, *3–4/1992*) that $TD(\lambda)$ converges for a generic λ with only a few specific assumptions and the fundamental condition that the policy is GLIE. The proof is very technical, and it's beyond the scope of this book; however, the most important assumptions (which are generally met) are:

- The **Markov Decision Process** (**MDP**) has absorbing states (in other words, all of the episodes end in a finite number of steps).
- All of the transition probabilities are not-null (all states can be visited an infinite number of times).

The first condition is obvious, the absence of absorbing states yields infinite explorations, which are not compatible with a TD method (sometimes it's possible to prematurely end an episode, but this can either be unacceptable (in some contexts) or a sub-optimal choice (in many others)). Moreover, Sutton and Barto (in the aforementioned book) proved that TD(λ) is equivalent to employing the weighted average of discounted return approximations, but without the constraint of looking ahead in the future (which is clearly impossible).

The complete TD(λ) algorithm (with an optional forced termination of the episode) is:

1. Set an initial deterministic random policy, $\pi(s)$
2. Set the initial value array, $V(s) = 0 \; \forall \, s \in S$

3. Set the initial eligibility trace array, $e(s) = 0 \; \forall \; s \in S$
4. Set the number of episodes, $N_{episodes}$
5. Set a maximum number of steps per episode, N_{max}
6. Set a constant, α ($\alpha = 0.1$)
7. Set a constant, γ ($\gamma = 0.9$)
8. Set a constant, λ ($\lambda = 0.5$)
9. Set a counter, $e = 0$
10. For $i = 1$ to $N_{episodes}$:
    1. Create an empty state list, L
    2. Observe the initial state, s_i, and append s_i to L
    3. While s_j is non-terminal and $e < N_{max:}$
        1. $e \mathrel{+}= 1$
        2. Select the action, $a_t = \pi(s_i)$
        3. Observe the transition, $(a_t, s_i) \rightarrow (s_j, r_{ij})$
        4. Compute the TD error as $TD_{error} = r_{ij} + \gamma V(s_j) - V(s_i)$
        5. Increment the eligibility trace, $e(s_i) \mathrel{+}= 1.0$
        6. For s in L:
            1. Update the value, $V(s) \mathrel{+}= \alpha \cdot TD_{error} \cdot e(s)$
            2. Update the eligibility trace, $e(s) \mathrel{*}= \gamma\lambda$
        7. Set $s_i = s_j$
        8. Append s_j to L
    4. Update the policy to be greedy with respect to the value function, $\pi(s) = argmax_a \, Q(s, a)$

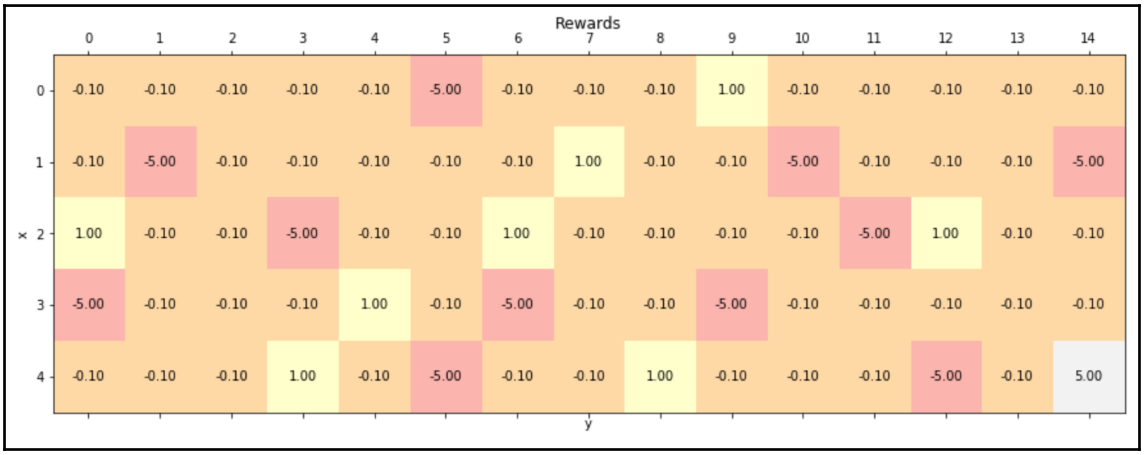
The reader can better understand the logic of this algorithm by considering the TD error and its back-propagation. Even if this is only a comparison, it's possible to imagine the behavior of TD(λ) as similar to the **Stochastic Gradient Descent** (**SGD**) algorithms employed to train a neural network. In fact, the error is propagated to the previous states (analogous to the lower layers of an MLP) and affects them proportionally to their importance, which is defined by their eligibility traces. Hence, a state with a higher eligibility trace can be considered more responsible for the error; therefore, the corresponding value must be corrected proportionally. This isn't a formal explanation, but it can simplify comprehension of the dynamics without an excessive loss of rigor.

TD(λ) in a more complex Checkerboard environment

At this point, we want to test the TD(λ) algorithm with a slightly more complex tunnel environment. In fact, together with the absorbing states, we will also consider some intermediate positive states, which can be imagined as *checkpoints*. An agent should learn the optimal path from any cell to the final state, trying to pass through the highest number of checkpoints possible. Let's start by defining the new structure:

```
import numpy as np

width = 15
height = 5

y_final = width - 1
x_final = height - 1

y_wells = [0, 1, 3, 5, 5, 6, 7, 9, 10, 11, 12, 14]
x_wells = [3, 1, 2, 0, 4, 3, 1, 3, 1, 2, 4, 1]

y_prizes = [0, 3, 4, 6, 7, 8, 9, 12]
x_prizes = [2, 4, 3, 2, 1, 4, 0, 2]

standard_reward = -0.1
tunnel_rewards = np.ones(shape=(height, width)) * standard_reward

def init_tunnel_rewards():
    for x_well, y_well in zip(x_wells, y_wells):
        tunnel_rewards[x_well, y_well] = -5.0

    for x_prize, y_prize in zip(x_prizes, y_prizes):
        tunnel_rewards[x_prize, y_prize] = 1.0

    tunnel_rewards[x_final, y_final] = 5.0

init_tunnel_rewards()
```

The reward structure is shown in the following diagram:



Reward schema in the new tunnel environment

At this point, we can proceed to initialize all of the constants (in particular, we have chosen $\lambda = 0.6$, which is an intermediate solution that guarantees an awareness close to a Monte Carlo method, without compromising the learning speed):

```
import numpy as np

nb_actions = 4
max_steps = 1000
alpha = 0.25
lambd = 0.6
gamma = 0.95

tunnel_values = np.zeros(shape=(height, width))
eligibility_traces = np.zeros(shape=(height, width))
policy = np.random.randint(0, nb_actions, size=(height,
width)).astype(np.uint8)
```

As in Python, the keyword `lambda` is reserved; we used the truncated expression `lambd` to declare the constant.

As we want to start from a random cell, we need to repeat the same procedure presented in the previous chapter; but, in this case, we are also including the checkpoint states:

```
import numpy as np

xy_grid = np.meshgrid(np.arange(0, height), np.arange(0, width),
sparse=False)
xy_grid = np.array(xy_grid).T.reshape(-1, 2)

xy_final = list(zip(x_wells, y_wells)) + list(zip(x_prizes, y_prizes))
xy_final.append([x_final, y_final])

xy_start = []

for x, y in xy_grid:
    if (x, y) not in xy_final:
        xy_start.append([x, y])
xy_start = np.array(xy_start)

def starting_point():
    xy = np.squeeze(xy_start[np.random.randint(0, xy_start.shape[0],
size=1)])
    return xy[0], xy[1]
```

We can now define the `episode()` function, which implements a complete TD(λ) cycle. As we don't want the agent to roam around trying to pass through the checkpoints an infinite number of times, we have decided to reduce the reward during the exploration, to incentivize the agent to pass through only the necessary checkpoints—trying, at the same time, to reach the final state as soon as possible:

```
import numpy as np

def is_final(x, y):
    if (x, y) in zip(x_wells, y_wells) or (x, y) == (x_final, y_final):
        return True
    return False

def episode():
    (i, j) = starting_point()
    x = y = 0
    e = 0
    state_history = [(i, j)]
    init_tunnel_rewards()
    total_reward = 0.0
    while e < max_steps:
        e += 1
        action = policy[i, j]
```

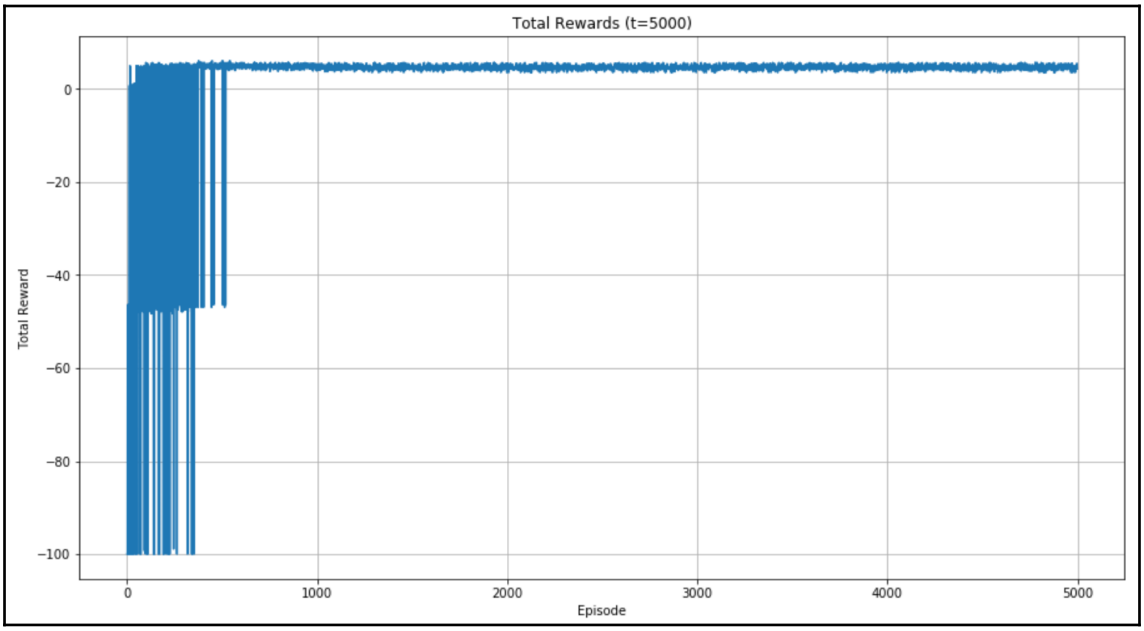
```
            if action == 0:
                if i == 0:
                    x = 0
                else:
                    x = i - 1
                y = j
            elif action == 1:
                if j == width - 1:
                    y = width - 1
                else:
                    y = j + 1
                x = i
            elif action == 2:
                if i == height - 1:
                    x = height - 1
                else:
                    x = i + 1
                y = j
            else:
                if j == 0:
                    y = 0
                else:
                    y = j - 1
                x = i
            reward = tunnel_rewards[x, y]
            total_reward += reward
            td_error = reward + (gamma * tunnel_values[x, y]) -
tunnel_values[i, j]
            eligibility_traces[i, j] += 1.0
            for sx, sy in state_history:
                tunnel_values[sx, sy] += (alpha * td_error *
eligibility_traces[sx, sy])
                eligibility_traces[sx, sy] *= (gamma * lambd)
            if is_final(x, y):
                break
            else:
                i = x
                j = y
                state_history.append([x, y])
                tunnel_rewards[x_prizes, y_prizes] *= 0.85
    return total_reward
```

```
def policy_selection():
    for i in range(height):
        for j in range(width):
            if is_final(i, j):
                continue
            values = np.zeros(shape=(nb_actions, ))
            values[0] = (tunnel_rewards[i - 1, j] + (gamma *
tunnel_values[i - 1, j])) if i > 0 else -np.inf
            values[1] = (tunnel_rewards[i, j + 1] + (gamma *
tunnel_values[i, j + 1])) if j < width - 1 else -np.inf
            values[2] = (tunnel_rewards[i + 1, j] + (gamma *
tunnel_values[i + 1, j])) if i < height - 1 else -np.inf
            values[3] = (tunnel_rewards[i, j - 1] + (gamma *
tunnel_values[i, j - 1])) if j > 0 else -np.inf
            policy[i, j] = np.argmax(values).astype(np.uint8)
```

The `is_final()` and `policy_selection()` functions are the same ones defined in the previous chapter, and need no explanation. Even if it's not really necessary, we have decided to implement a forced termination after a number of steps, equal to `max_steps`. This is helpful at the beginning because as the policy is not ε-greedy, the agent can remain stuck in a looping exploration that never ends. We can now train the model for a fixed number of episodes (alternatively, it's possible to stop the process when the value array doesn't change anymore):

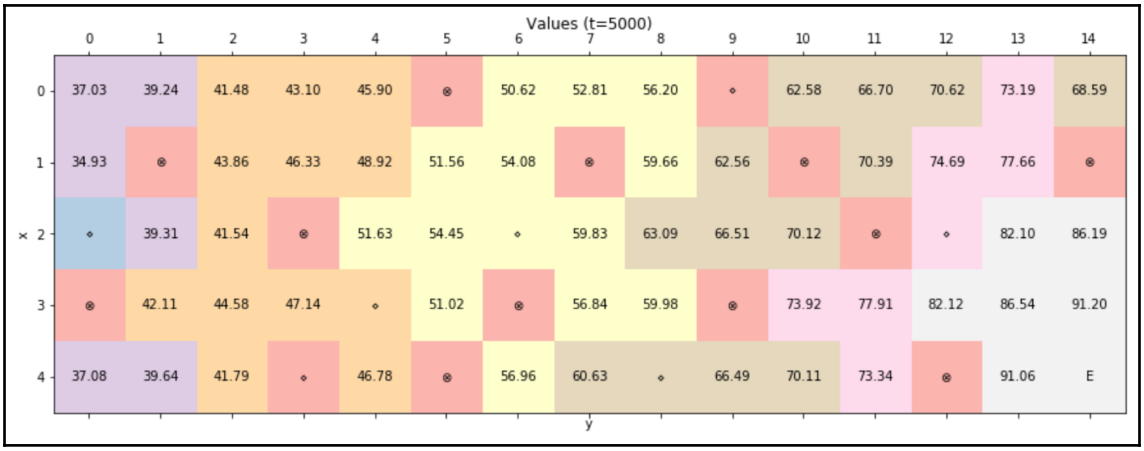
```
n_episodes = 5000

total_rewards = []

for _ in range(n_episodes):
    e_reward = episode()
    total_rewards.append(e_reward)
    policy_selection()
```

The `episode()` function returns the total rewards; therefore, it's useful to check how the agent learning process evolved:



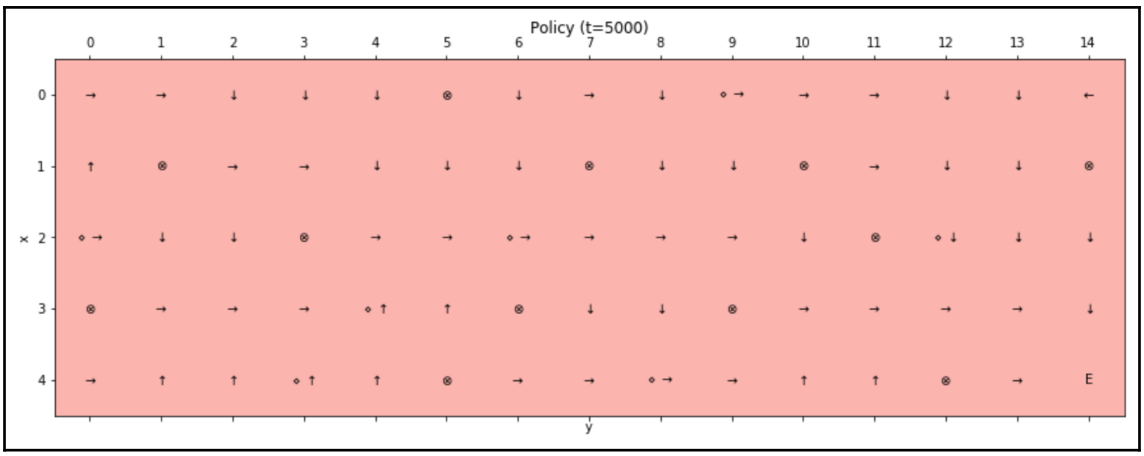
Total rewards achieved by the agent

At the beginning (for about 500 episodes), the agent employs an unacceptable policy that yields very negative total rewards. However, in about 1,000 iterations, the algorithm reaches an optimal policy that is only slightly improved by the following episodes. The oscillations are due to the different starting points; however, the total rewards are never negative, and as the checkpoint weights decay, this is a positive signal, indicating that the agent reaches the final positive state. To have a confirmation of this hypothesis, we can plot the learned value function:



Final value matrix

The values are coherent with our initial analysis; in fact, they tend to be higher when the cell is close to a checkpoint, but, at the same time, the global configuration (considering a policy greedy with respect to *V(s)*) forces the agent to reach the ending state whose surrounding values are the highest. The last step is checking the actual policy, with a particular focus on the checkpoints:



Final policy

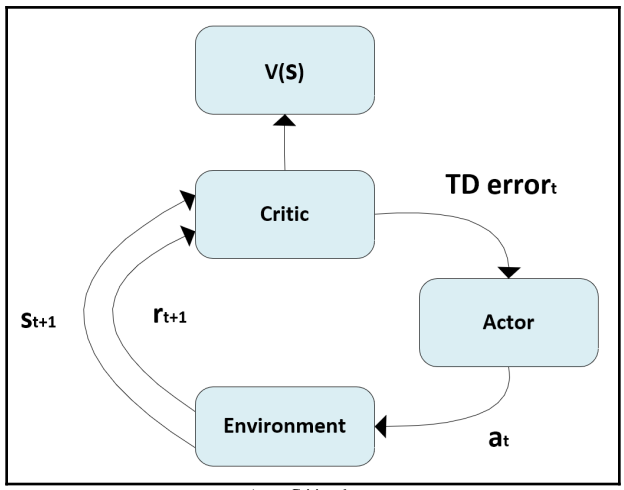
As it's possible to observe, the agent tries to pass through the checkpoints, but when it's close to the final state, it (correctly) prefers to end the episode as soon as possible. I invite the reader to repeat the experiment using different values for the constant λ, and changing the environment dynamics for the checkpoints. What happens if their values remain the same? Is it possible to improve the policy with a higher λ?

It's important to remember that, as we are extensively using random values, successive experiments can yield different results due to different initial conditions. However, the algorithm should always converge to an optimal policy when the number of episodes is high enough.

Actor-Critic TD(0) in the checkerboard environment

In this example, we want to employ an alternative algorithm called *Actor-Critic*, together with TD(0). In this method, the agent is split into two components, a Critic, which is responsible for evaluating the quality of the value estimation, and an actor, which selects and performs an action. As pointed out by Dayan (in *Theoretical Neuroscience*, *Dayan P.*, *Abbott L. F.*, *The MIT Press*), the dynamics of an Actor-Critic approach are similar to the interleaving policy evaluation and policy improvement steps. In fact, the knowledge of the Critic is obtained through an iterative process, and its initial evaluations are normally sub-optimal.

The structural schema is shown in the following diagram:



Actor-Critic schema

In this particular case, it's preferable to employ a ε-greedy soft policy, based on the softmax function. The model stores a matrix (or an approximating function) called *policy importance*, where each entry $p_i(s, a)$ is a value representing the preference for a specific action in a certain state. The actual stochastic policy is obtained by applying the softmax with a simple trick to increase the numerical stability when the exponentials become very large:

$$\pi(s,a) = \frac{e^{p_i(s,a)}}{\sum_{a_k} e^{p_i(s,a_k)}} = \frac{e^{-max_a p_i(s,a)}\ e^{p_i(s,a)}}{e^{-max_a p_i(s,a)}\ \sum_{a_k} e^{p_i(s,a_k)}} = \frac{e^{p_i(s,a)-max_a p_i(s,a)}}{\sum_{a_k} e^{p_i(s,a_k)-max_a p_i(s,a)}}$$

After performing the action a in the state s_i and observing the transition to the state s_j with a reward r_{ij}, the Critic evaluates the TD error:

$$TD_{error} = r_{ij} + \gamma V(s_j) - V(s_i)$$

If $V(s_i) < r_{ij} + \gamma V(s_j)$, the transition is considered positive, because the value is increasing. Conversely, when $V(s_i) > r_{ij} + \gamma V(s_j)$, the Critic evaluates the action as negative, because the previous value was higher than the new estimation. A more general approach is based on the concept of *advantage*, which is defined as:

$$A(s,a;\pi) = Q(s,a;\pi) - V(s;\pi)$$

Normally, one of the terms from the previous expression can be approximated. In our case, we cannot compute the Q function directly; hence, we approximate it with the term $r_{ij} + \gamma V(s_j)$. It's clear that the role of the advantage is analogous to the one of the TD error (which is an approximation) and must represent the confirmation that an action in a certain state is a good or bad choice. An analysis of all **advantage Actor-Critic** (**A3C**) algorithms (in other words, improvements of the standard *policy gradient* algorithm) is beyond the scope of this book. However, the reader can find some helpful pieces of information in *High-Dimensional Continuous Control Using Generalized Advantage Estimation*, *Schulman J., Moritz P., Levine S., Jordan M. I., Abbeel P., ICLR 2016.*

Of course, an Actor-Critic correction is not sufficient. To improve the policy, it's necessary to employ a standard algorithm (such as TD(0), TD(λ), or least square regression, which can be implemented using a neural network) in order to learn the correct value function, $V(s)$. As for many other algorithms, this process can converge only after a sufficiently high number of iterations, which must be exploited to visit the states many times, experimenting with all possible actions.

Hence, with a TD(0) approach, the first step after evaluating the TD error is updating *V(s)* using the rule defined in the previous chapter:

$$V(s_i) = V(s_i) + \alpha\left(r_{ij} + \gamma V(s_j) - V(s_i)\right) = V(s_i) + \alpha TD_{error}$$

The second step is more pragmatic; in fact, the main role of the Critic is actually to criticize every action, deciding when it's better to increase or decrease the probability of selecting it again in a certain state. This goal can be achieved by simply updating the policy importance:

$$p_i(s_i, a) = p_i(s_i, a) + \rho TD_{error}$$

The role of the learning rate ρ is extremely important; in fact, incorrect values (in other words, values that are too high) can yield initial wrong corrections that may compromise the convergence. It's essential to not forget that the value function is almost completely unknown at the beginning, and therefore the Critic has no chance to increase the right probability with awareness. For this reason, I always suggest to start with very small value ($\rho = 0.001$) and increase it only if the convergence speed of the algorithm is effectively improved.

As the policy is based on the softmax function, after a Critic update, the values will always be renormalized, resulting in an actual probability distribution. After an adequately large number of iterations, with the right choice of both ρ and γ, the model is able to learn both a stochastic policy and a value function. Therefore, it's possible to employ the trained agent by always selecting the action with the highest probability (which corresponds to an implicitly greedy behavior):

$$\pi(s) = argmax_a \pi(s, a)$$

Let's now apply this algorithm to the tunnel environment. The first step is defining the constants (as we are looking for a long sighted agent, we are setting the discount factor $\gamma = 0.99$):

```
import numpy as np

tunnel_values = np.zeros(shape=(height, width))

gamma = 0.99
alpha = 0.25
rho = 0.001
```

At this point, we need to define the policy importance array, and a function to generate the softmax policy:

```
import numpy as np

nb_actions = 4

policy_importances = np.zeros(shape=(height, width, nb_actions))

def get_softmax_policy():
    softmax_policy = policy_importances - np.amax(policy_importances,
axis=2, keepdims=True)
    return np.exp(softmax_policy) / np.sum(np.exp(softmax_policy), axis=2,
keepdims=True)
```

The functions needed to implement a single training step are very straightforward, and the reader should already be familiar with their structure:

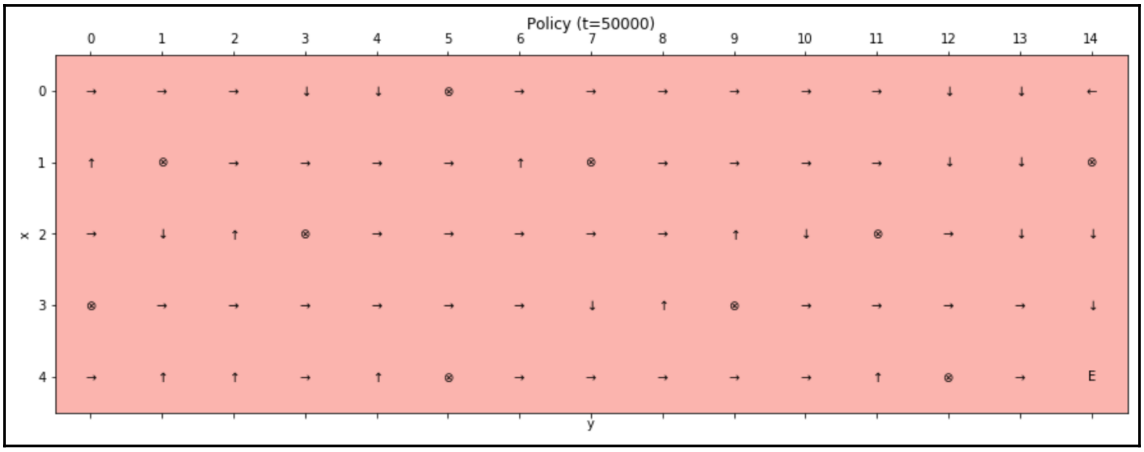
```
import numpy as np

def select_action(epsilon, i, j):
    if np.random.uniform(0.0, 1.0) < epsilon:
        return np.random.randint(0, nb_actions)
    policy = get_softmax_policy()
    return np.argmax(policy[i, j])

def action_critic_episode(epsilon):
    (i, j) = starting_point()
    x = y = 0
    e = 0
    while e < max_steps:
        e += 1
        action = select_action(epsilon, i, j)
        if action == 0:
            if i == 0:
                x = 0
            else:
                x = i - 1
            y = j
        elif action == 1:
            if j == width - 1:
                y = width - 1
            else:
                y = j + 1
            x = i
        elif action == 2:
            if i == height - 1:
```

```
                x = height - 1
            else:
                x = i + 1
            y = j
        else:
            if j == 0:
                y = 0
            else:
                y = j - 1
            x = i
        reward = tunnel_rewards[x, y]
        td_error = reward + (gamma * tunnel_values[x, y]) -
tunnel_values[i, j]
        tunnel_values[i, j] += (alpha * td_error)
        policy_importances[i, j, action] += (rho * td_error)
        if is_final(x, y):
            break
        else:
            i = x
            j = y
```

At this point, we can train the model with 50,000 iterations, and 30,000 explorative ones (with a linear decay of the exploration factor):

```
n_episodes = 50000
n_exploration = 30000

for t in range(n_episodes):
    epsilon = 0.0
    if t <= n_exploration:
        epsilon = 1.0 - (float(t) / float(n_exploration))
    action_critic_episode(epsilon)
```

The resulting greedy policy is shown in the following figure:



Final greedy policy

The final greedy policy is consistent with the objective, and the agent always reaches the final positive state by avoiding the wells. This kind of algorithm can appear more complex than necessary; however, in complex situations, it turns out to be extremely effective. In fact, the learning process can be dramatically improved, thanks to the fast corrections performed by the Critic. Moreover, the author has noticed that the Actor-Critic is more robust to wrong (or noisy) evaluations. As the policy is learned separately, the effect of small variations in *V(s)* cannot easily change the probabilities $\pi(s, a)$ (in particular, when an action is generally much *stronger* than the others). On the other hand, as discussed previously, it's necessary to avoid a premature convergence in order to let the algorithm modify the importance/probabilities, without an excessive number of iterations. The right trade-off can be found only after a complete analysis of each specific scenario, and unfortunately, there are no general rules that work in every case. My suggestion is to test various configurations, starting with small values (and, for example, a discount factor of $\gamma \in [0.7, 0.9]$), evaluating the total reward achieved after the same exploration period.

Complex deep learning models (such as asynchronous A3C; see *Asynchronous Methods for Deep Reinforcement Learning*, *Mnih V., Puigdomènech Badia A., Mirza M., Graves A., Lillicrap T. P., Harley T., Silver D., Kavukcuoglu K., arXiv:1602.01783 [cs.LG]* for further information) are based on a single network that outputs both the softmax policy (whose actions are generally proportional to their probability) and the value. Instead of employing an explicitly ε-greedy soft policy, it's possible to add a *maximum-entropy constraint* to the global cost function:

$$max\ H(\pi) = -max \sum_i \pi(s, a_i)\ log\ \pi(s, a_i) = min \sum_i \pi(s, a_i)\ log\ \pi(s, a_i)$$

As the entropy is at the maximum when all of the actions have the same probability, this constraint (with an appropriate weight) forces the algorithm to increase the exploration probability until an action becomes dominant and there's no more need to avoid a greedy selection. This is a sound and easy way to employ an *adaptive ε-greedy policy*, because as the model works with each state separately, the states where the uncertainty is very low can become greedy; it's possible to automatically keep a high entropy whenever it's necessary to continue the exploration, in order to maximize the reward.

The effect of double correction, together with a maximum-entropy constraint, improves the convergence speed of the model, encourages the exploration during the initial iterations, and yields very high final accuracy. I invite the reader to implement this variant with other scenarios and algorithms. In particular, at the end of this chapter, we are going to experiment with an algorithm based on a neural network. As the example is pretty simple, I suggest using Tensorflow to create a small network based on the Actor-Critic approach. The reader can employ a *mean squared error* loss for the value and softmax cross entropy for the policy. Once the models work successfully with our toy examples, it will be possible to start working with more complex scenarios (like the ones proposed in OpenAI Gym at `https://gym.openai.com/`).

SARSA algorithm

SARSA (whose name is derived from the sequence *state-action-reward-state-action*) is a natural extension of TD(0) to the estimation of the *Q* function. Its standard formulation (which is sometimes called one-step SARSA, or SARSA(0), for the same reasons explained in the previous chapter) is based on a single next reward, r_{t+1}, which is obtained by executing the action a_t in the state s_t. The temporal difference computation is based on the following update rule:

$$Q(s_t, a_t; \pi) = Q(s_t, a_t; \pi) + \alpha \left(r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}; \pi) - Q(s_t, a_t; \pi) \right)$$

The equation is equivalent to TD(0), and if the policy is chosen to be GLIE, it has been proven (in *Convergence Results for Single-Step On-Policy Reinforcement-Learning Algorithms*, *Singh S., Jaakkola T., Littman M. L., Szepesvári C., Machine Learning, 39/2000*) that SARSA converges to an optimal policy, $\pi^{opt}(s)$, with the probability 1, when all couples (state, action) are experienced an infinite number of times. This means that if the policy is updated to be greedy with respect to the current value function induced by *Q*, it holds that:

$$p\left(\lim_{k \to \infty} \pi^{(k)}(s) = \pi^{opt}(s) \right) = 1 \quad \forall\, s \in S$$

The same result is valid for the *Q* function. In particular, the most important conditions required by the proof are:

- The learning rate, $\alpha \in [0, 1]$, with the constraints $\Sigma\alpha = \infty$ and $\Sigma\alpha^2 < \infty$
- The variance of the rewards must be finite

The first condition is particularly important when α is a function of the state and the time step; however, in many cases, it is a constant bounded between 0 and 1, and hence, $\Sigma\alpha^2 = \infty$. A common way to solve this problem (above all when a large number of iterations are required) is to let the learning rate decay (in other words, exponentially) during the training process. Instead, to mitigate the effect of very large rewards, it's possible to clip them in a suitable range ($[-1, 1]$). In many cases, it's not necessary to employ these strategies, but in more complex scenarios, they can become crucial in order to ensure the convergence of the algorithm. Moreover, as pointed out in the previous chapter, these kinds of algorithms need a long exploration phase before starting to stabilize the policy. The most common strategy is to employ a ε-greedy policy, with a temporal decay of the exploration factor. During the first iterations, the agent must explore without caring about the returns of the actions. In this way, it's possible to assess the actual values before the beginning of a final refining phase characterized by a purely greedy exploration, based on a more precise approximation of $V(s)$.

The complete SARSA(0) algorithm (with an optional forced termination of the episode) is:

1. Set an initial deterministic random policy, $\pi(s)$
2. Set the initial value array, $Q(s, a) = 0 \; \forall \, s \in S$ and $\forall \, a \in A$
3. Set the number of episodes, $N_{episodes}$
4. Set a maximum number of steps per episode, N_{max}
5. Set a constant, α ($\alpha = 0.1$)
6. Set a constant, γ ($\gamma = 0.9$)
7. Set an initial exploration factor, $\varepsilon^{(0)}$ ($\varepsilon^{(0)} = 1.0$)
8. Define a policy to let the exploration factor ε decay (linear or exponential)
9. Set a counter, $e = 0$
10. For $i = 1$ to $N_{episodes}$:
    1. Observe the initial state, s_i
    2. While s_j is non-terminal and $e < N_{max}$:
        1. e += 1
        2. Select the action, $a_t = \pi(s_i)$, with an exploration factor $\varepsilon^{(e)}$
        3. Observe the transition, $(a_t, s_i) \rightarrow (s_j, r_{ij})$

4. Select the action, $a_{t+1} = \pi(s_j)$, with an exploration factor $\varepsilon^{(e)}$
5. Update the $Q(s_t, a_t)$ function (if s_j is terminal, set $Q(s_{t+1}, a_{t+1}) = 0$)
6. Set $s_i = s_j$

The concept of eligibility trace can also be extended to SARSA (and other TD methods); however, that is beyond the scope of this book. A reader who is interested can find all of the algorithms (together with their mathematical formulations) in *Sutton R. S., Barto A. G., Reinforcement Learning, A Bradford Book*.

SARSA in the checkerboard environment

We can now test the SARSA algorithm in the original tunnel environment (all of the elements that are not redefined are the same as the previous chapter). The first step is defining the *Q(s, a)* array and the constants employed in the training process:

```
import numpy as np

nb_actions = 4

Q = np.zeros(shape=(height, width, nb_actions))

x_start = 0
y_start = 0

max_steps = 2000
alpha = 0.25
```

As we want to employ a ε-greedy policy, we can set the starting point to `(0, 0)`, forcing the agent to reach the positive final state. We can now define the functions needed to perform a training step:

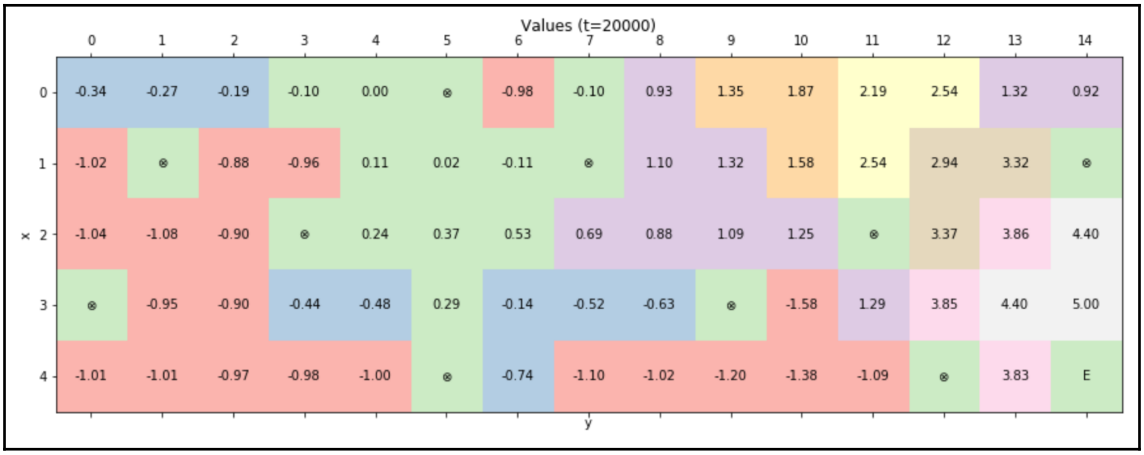
```
import numpy as np

def is_final(x, y):
    if (x, y) in zip(x_wells, y_wells) or (x, y) == (x_final, y_final):
        return True
    return False

def select_action(epsilon, i, j):
    if np.random.uniform(0.0, 1.0) < epsilon:
        return np.random.randint(0, nb_actions)
```

```
    return np.argmax(Q[i, j])

def sarsa_step(epsilon):
    e = 0
    i = x_start
    j = y_start
    while e < max_steps:
        e += 1
        action = select_action(epsilon, i, j)
        if action == 0:
            if i == 0:
                x = 0
            else:
                x = i - 1
            y = j
        elif action == 1:
            if j == width - 1:
                y = width - 1
            else:
                y = j + 1
            x = i
        elif action == 2:
            if i == height - 1:
                x = height - 1
            else:
                x = i + 1
            y = j
        else:
            if j == 0:
                y = 0
            else:
                y = j - 1
            x = i
        action_n = select_action(epsilon, x, y)
        reward = tunnel_rewards[x, y]
        if is_final(x, y):
            Q[i, j, action] += alpha * (reward - Q[i, j, action])
            break
        else:
            Q[i, j, action] += alpha * (reward + (gamma * Q[x, y,
action_n]) - Q[i, j, action])
            i = x
            j = y
```

The `select_action()` function has been designed to select a random action with the probability ε, and a greedy one with respect to $Q(s, a)$, with the probability $1 - \varepsilon$.
The `sarsa_step()` function is straightforward, and executes a complete episode updating the $Q(s, a)$ (that's why this is an online algorithm). At this point, it's possible to train the model for 20,000 episodes and employ a linear decay for ε during the first 15,000 episodes (when t > 15,000, ε is set equal to 0 in order to employ a purely greedy policy):

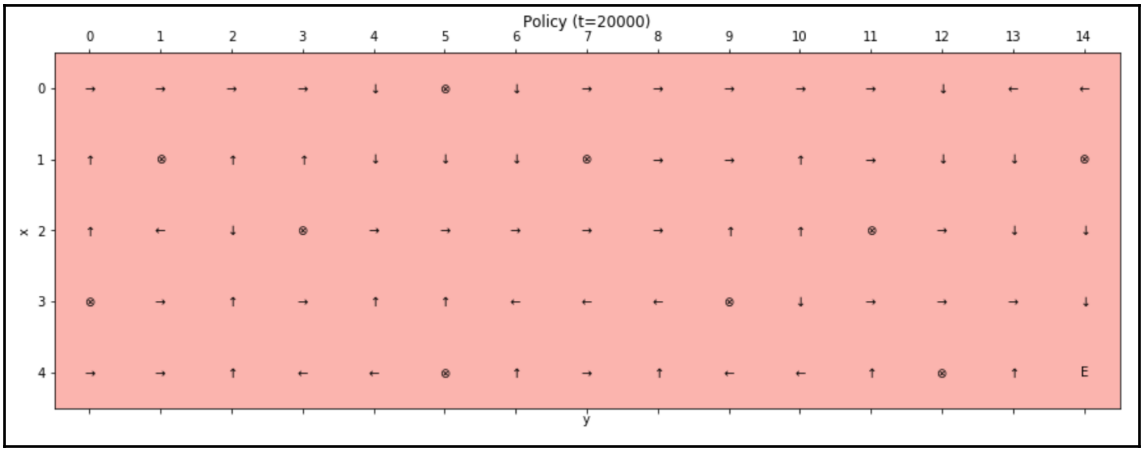
```
n_episodes = 20000
n_exploration = 15000

for t in range(n_episodes):
    epsilon = 0.0
    if t <= n_exploration:
        epsilon = 1.0 - (float(t) / float(n_exploration))
    sarsa_step(epsilon)
```

As usual, let's check the learned values (considering that the policy is greedy, we're going to plot $V(s) = max_a Q(s, a)$):



Final value matrix (as $V(s) = max_a Q(s, a)$)

As expected, the Q function has been learned in a consistent way, and we can get a confirmation plotting the resulting policy:



Final policy

The policy is coherent with the initial objective, and the agent avoids all negative absorbing states, always trying to move towards the final positive state. However, some paths seem longer than expected. As an exercise, I invite the reader to retrain the model for a larger number of iterations, adjusting the exploration period. Moreover, *is it possible to improve the model by increasing (or decreasing) the discount factor γ?* Remember that $\gamma \rightarrow 0$ leads to a short-sighted agent, which is able to select actions only considering the immediate reward, while $\gamma \rightarrow 1$ forces the agent to take into account a larger number of future rewards. This particular example is based on a long environment, because the agent always starts from *(0, 0)* and must reach the farthest point; therefore, all intermediate states have less importance, and it's helpful to look at the future to pick the optimal actions. Using random starts can surely improve the policy for all initial states, but it's interesting to investigate how different γ values can affect the decisions; hence, I suggest repeating the experiment in order to evaluate the various configurations and increase awareness about the different factors that are involved in a TD algorithm.

Q-learning

This algorithm was proposed by Watkins (in *Learning from delayed rewards*, *Watkins C.I.C.H., Ph.D. Thesis, University of Cambridge, 1989*; and further analyzed in *Watkins C.I.C.H., Dayan P., Technical Note Q-Learning, Machine Learning 8, 1992*) as a more efficient alternative to SARSA. The main feature of *Q-learning* is that the TD update rule is immediately greedy with respect to the $Q(s_{t+1}, a)$ function:

$$Q(s_t, a_t; \pi) = Q(s_t, a_t; \pi) + \alpha \left(r_{t+1} + \gamma \, max_a Q(s_{t+1}, a; \pi) - Q(s_t, a_t; \pi) \right)$$

The key idea is to compare the current $Q(s_t, a_t)$ value with the maximum Q value achievable when the agent is in the successor state. In fact, as the policy must be GLIE, the convergence speed can be increased by avoiding wrong estimations due to the selection of a Q value that won't be associated with the final action. By choosing the maximum Q value, the algorithm will move towards the optimal solution faster than SARSA, and also, the convergence proof is less restrictive. In fact, Watkins and Dayan (in the aforementioned papers) proved that, if $|r_i| < R$, the learning rate $\alpha \in [0, 1[$ (in this case, α must be always smaller than 1) with the same constraints imposed for SARSA ($\Sigma\alpha = \infty$ and $\Sigma\alpha^2 < \infty$), then the estimated Q function converges with probability 1 to the optimal one:

$$p\left(\lim_{k\to\infty} Q^{(k)}(s,a) = Q^{opt}(s,a)\right) = 1 \quad \forall\, s \in S \;\; and \;\; \forall\, a \in A$$

As discussed for SARSA, the conditions on the rewards and the learning rate can be managed by employing a clipping function and a temporal decay, respectively. In almost all deep Q-learning applications, these are extremely important factors to guarantee the convergence; therefore, I invite the reader to consider them whenever the training process isn't able to converge to an acceptable solution.

The complete Q-learning algorithm (with an optional forced termination of the episode) is:

1. Set an initial deterministic random policy, $\pi(s)$
2. Set the initial value array, $Q(s, a) = 0\ \forall\, s \in S$ and $\forall\, a \in A$
3. Set the number of episodes, $N_{episodes}$
4. Set a maximum number of steps per episode, N_{max}
5. Set a constant, α ($\alpha = 0.1$)
6. Set a constant, γ ($\gamma = 0.9$)
7. Set an initial exploration factor, $\varepsilon^{(0)}$ ($\varepsilon^{(0)} = 1.0$)
8. Define a policy to let the exploration factor ε decay (linear or exponential)
9. Set a counter, $e = 0$
10. For $i = 1$ to $N_{episodes}$:
    1. Observe the initial state, s_i
    2. While s_j is non-terminal and $e < N_{max}$:
        1. e += 1
        2. Select the action, $a_t = \pi(s_i)$, with an exploration factor $\varepsilon^{(e)}$
        3. Observe the transition $(a_t, s_i) \rightarrow (s_j, r_{ij})$
        4. Select the action, $a_{t+1} = \pi(s_j)$, with an exploration factor $\varepsilon^{(e)}$

5. Update the $Q(s_t, a_t)$ function (if s_j is terminal, set $Q(s_{t+1}, a_{t+1}) = 0$) using $max_a\ Q(s_{t+1}, a)$
6. Set $s_i = s_j$

Q-learning in the checkerboard environment

Let's repeat the previous experiment with the Q-learning algorithm. As all of the constants are the same (as well as the choice of a ε-greedy policy and the starting point set to *(0, 0)*), we can directly define the function that implements the training for a single episode:

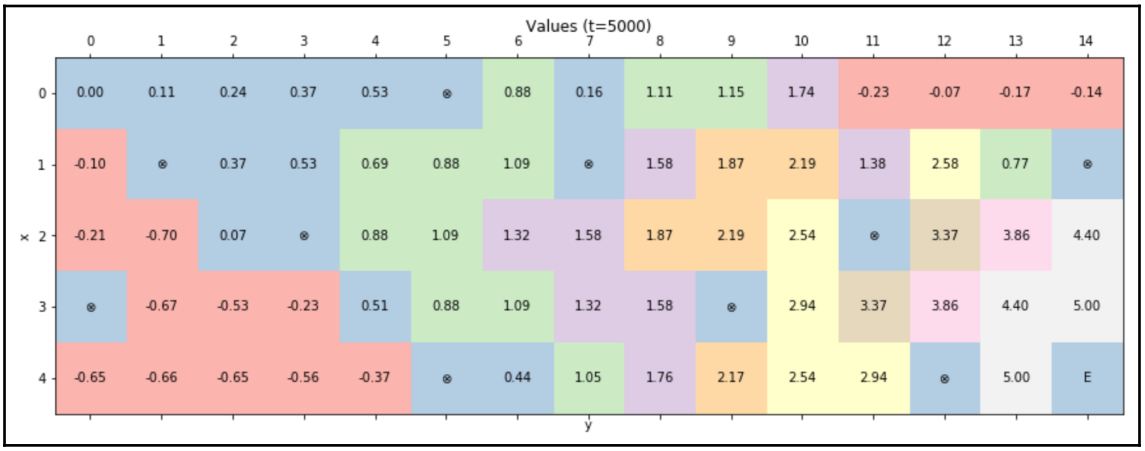
```
import numpy as np

def q_step(epsilon):
    e = 0
    i = x_start
    j = y_start
    while e < max_steps:
        e += 1
        action = select_action(epsilon, i, j)
        if action == 0:
            if i == 0:
                x = 0
            else:
                x = i - 1
            y = j
        elif action == 1:
            if j == width - 1:
                y = width - 1
            else:
                y = j + 1
            x = i
        elif action == 2:
            if i == height - 1:
                x = height - 1
            else:
                x = i + 1
            y = j
        else:
            if j == 0:
                y = 0
            else:
                y = j - 1
            x = i
        reward = tunnel_rewards[x, y]
```

```
            if is_final(x, y):
                Q[i, j, action] += alpha * (reward - Q[i, j, action])
                break
            else:
                Q[i, j, action] += alpha * (reward + (gamma * np.max(Q[x, y]))
- Q[i, j, action])
                i = x
                j = y
```

We can now train the model for 5,000 iterations, with 3,500 explorative ones:

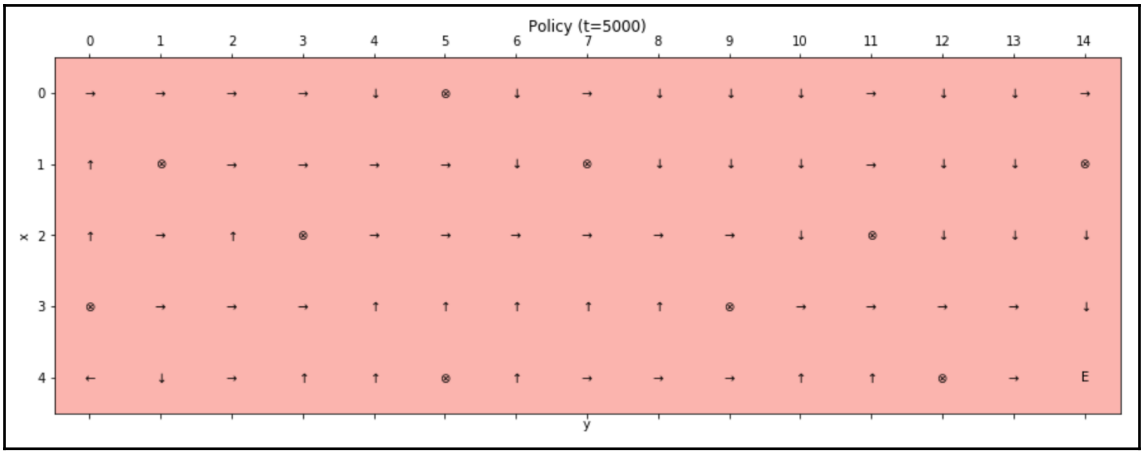
```
n_episodes = 5000
n_exploration = 3500

for t in range(n_episodes):
    epsilon = 0.0
    if t <= n_exploration:
        epsilon = 1.0 - (float(t) / float(n_exploration))
    q_step(epsilon)
```

The resulting value matrix (defined as in the SARSA experiment) is:



Final value matrix

Again, the learned *Q* function (and obviously, also the greedy *V(s)*) is coherent with the initial objective (in particular, considering the starting point set to *(0, 0)*), and the resulting policy can immediately confirm this result:



Final policy

The behavior of Q-learning is not very different from SARSA (even if the convergence is faster), and some initial states are not perfectly managed. This is a consequence of our choice; therefore, I invite the reader to repeat the exercise using random starts and comparing the training speed of Q-learning and SARSA.

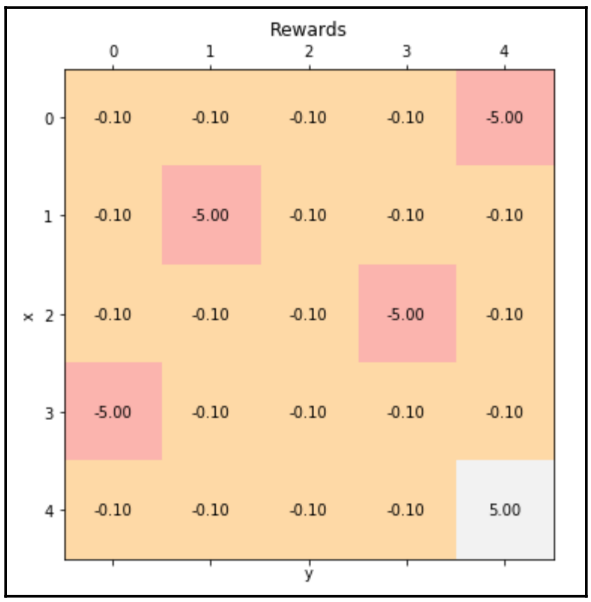
Q-learning using a neural network

Now, we want to test the Q-learning algorithm using a smaller checkerboard environment and a neural network (with Keras). The main difference from the previous examples is that now, the state is represented by a screenshot of the current configuration; hence, the model has to learn how to associate a value with each input image and action. This isn't actual deep Q-learning (which is based on Deep Convolutional Networks, and requires more complex environments that we cannot discuss in this book), but it shows how such a model can learn an optimal policy with the same input provided to a human being. In order to reduce the training time, we are considering a square checkerboard environment, with four negative absorbing states and a positive final one:

```
import numpy as np

width = 5
height = 5
nb_actions = 4

y_final = width - 1
x_final = height - 1
```

```
y_wells = [0, 1, 3, 4]
x_wells = [3, 1, 2, 0]

standard_reward = -0.1
tunnel_rewards = np.ones(shape=(height, width)) * standard_reward

for x_well, y_well in zip(x_wells, y_wells):
    tunnel_rewards[x_well, y_well] = -5.0

tunnel_rewards[x_final, y_final] = 5.0
```

A graphical representation of the rewards is shown in the following figure:



Rewards in the smaller checkerboard environment

As we want to provide the network with a graphical input, we need to define a function to create a matrix representing the tunnel:

```
import numpy as np

def reset_tunnel():
    tunnel = np.zeros(shape=(height, width), dtype=np.float32)

    for x_well, y_well in zip(x_wells, y_wells):
        tunnel[x_well, y_well] = -1.0
```

```
    tunnel[x_final, y_final] = 0.5
    return tunnel
```

The `reset_tunnel()` function sets all values equal to 0, except for (which is marked with `-1`) and the final state (defined by `0.5`). The position of the agent (defined with the value `1`) is directly managed by the training function. At this point, we can create and compile our neural network. As the problem is not very complex, we are employing an MLP:

```
from keras.models import Sequential
from keras.layers import Dense, Activation

model = Sequential()

model.add(Dense(8, input_dim=width * height))
model.add(Activation('tanh'))
model.add(Dense(4))
model.add(Activation('tanh'))

model.add(Dense(nb_actions))
model.add(Activation('linear'))

model.compile(optimizer='rmsprop',
              loss='mse')
```

The input is a flattened array, while the output is the *Q* function (all of the values corresponding to each action). The network is trained using RMSprop and a mean squared error loss function (our goal is to reduce the MSE between the actual value and the prediction). In order to train and query the network, it's helpful to create two dedicated functions:

```
import numpy as np

def train(state, q_value):
    model.train_on_batch(np.expand_dims(state.flatten(), axis=0),
np.expand_dims(q_value, axis=0))

def get_Q_value(state):
    return model.predict(np.expand_dims(state.flatten(), axis=0))[0]

def select_action_neural_network(epsilon, state):
    Q_value = get_Q_value(state)
    if np.random.uniform(0.0, 1.0) < epsilon:
        return Q_value, np.random.randint(0, nb_actions)
    return Q_value, np.argmax(Q_value)
```

The behavior of these functions is straightforward. The only element that may be new to the reader is the use of the `train_on_batch()` method. Contrary to `fit()`, this function allows us to perform a single training step, given a batch of input-output couples (in our case, we always have a single couple). As our goal is finding an optimal path to the final state, starting from every possible cell, we are going to employ random starts:

```
import numpy as np

xy_grid = np.meshgrid(np.arange(0, height), np.arange(0, width),
sparse=False)
xy_grid = np.array(xy_grid).T.reshape(-1, 2)

xy_final = list(zip(x_wells, y_wells))
xy_final.append([x_final, y_final])

xy_start = []

for x, y in xy_grid:
    if (x, y) not in xy_final:
        xy_start.append([x, y])
xy_start = np.array(xy_start)

def starting_point():
    xy = np.squeeze(xy_start[np.random.randint(0, xy_start.shape[0],
size=1)])
    return xy[0], xy[1]
```

Now, we can define the functions needed to perform a single training step:

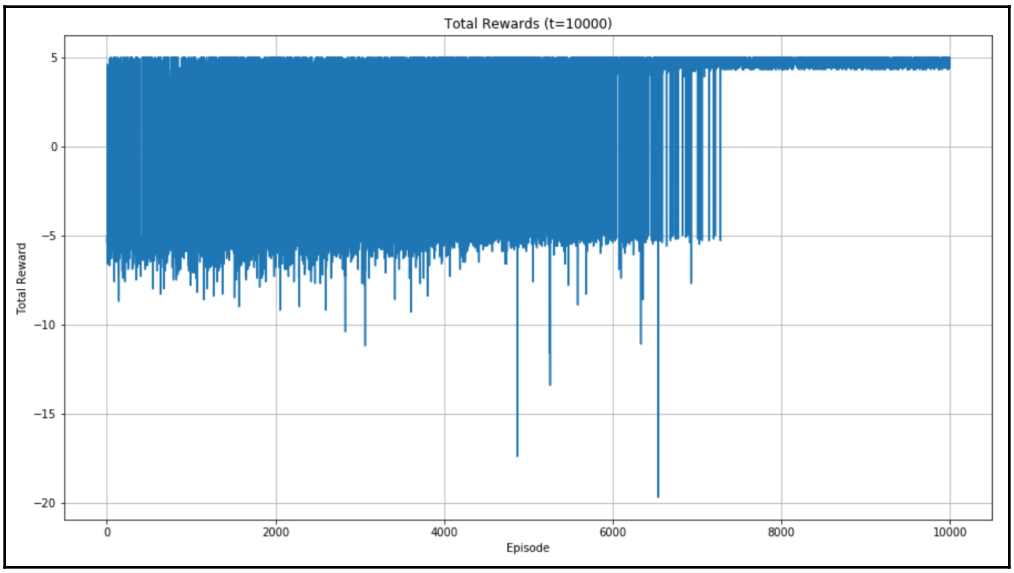
```
import numpy as np

def is_final(x, y):
    if (x, y) in zip(x_wells, y_wells) or (x, y) == (x_final, y_final):
        return True
    return False

def q_step_neural_network(epsilon, initial_state):
    e = 0
    total_reward = 0.0
    (i, j) = starting_point()
    prev_value = 0.0
    tunnel = initial_state.copy()
    tunnel[i, j] = 1.0
    while e < max_steps:
        e += 1
        q_value, action = select_action_neural_network(epsilon, tunnel)
        if action == 0:
```

```
            if i == 0:
                x = 0
            else:
                x = i - 1
            y = j
        elif action == 1:
            if j == width - 1:
                y = width - 1
            else:
                y = j + 1
            x = i
        elif action == 2:
            if i == height - 1:
                x = height - 1
            else:
                x = i + 1
            y = j
        else:
            if j == 0:
                y = 0
            else:
                y = j - 1
            x = i
        reward = tunnel_rewards[x, y]
        total_reward += reward
        tunnel_n = tunnel.copy()
        tunnel_n[i, j] = prev_value
        tunnel_n[x, y] = 1.0
        prev_value = tunnel[x, y]
        if is_final(x, y):
            q_value[action] = reward
            train(tunnel, q_value)
            break
        else:
            q_value[action] = reward + (gamma *
np.max(get_Q_value(tunnel_n)))
            train(tunnel, q_value)
            i = x
            j = y
            tunnel = tunnel_n.copy()
    return total_reward
```

The `q_step_neural_network()` function is very similar to the one defined in the previous example. The only difference is the management of the visual state. Every time there's a transition, the value `1.0` (denoting the agent) is moved from the old position to the new one, and the value of the previous cell is reset to its default (saved in the `prev_value` variable). Another secondary difference is the absence of α because there's already a learning rate set in the SGD algorithm, and it doesn't make sense to add another parameter to the model. We can now train the model for 10,000 iterations, with 7,500 explorative ones:

```
n_episodes = 10000
n_exploration = 7500

total_rewards = []

for t in range(n_episodes):
    tunnel = reset_tunnel()
    epsilon = 0.0
    if t <= n_exploration:
        epsilon = 1.0 - (float(t) / float(n_exploration))
    t_reward= q_step_neural_network(epsilon, tunnel)
    total_rewards.append(t_reward)
```

When the training process has finished, we can analyze the total rewards, in order to understand whether the network has successfully learned the *Q* functions:



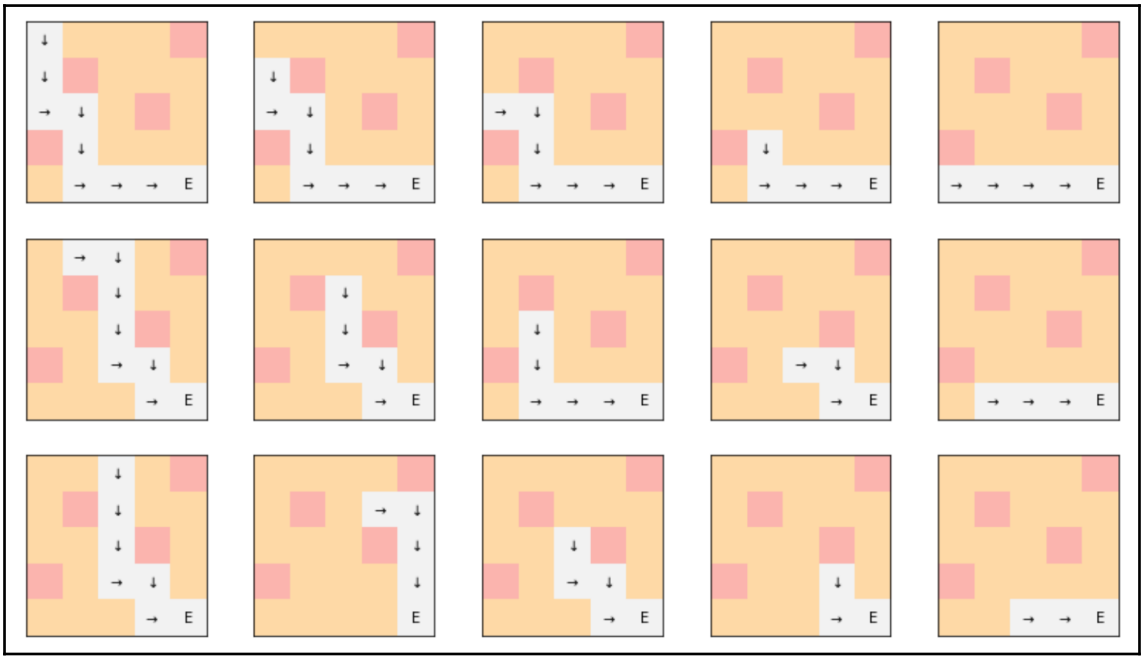
Total rewards obtained by the neural network Q-learning algorithm

It's clear that the model is working well, because after the exploration period, the total reward becomes stationary around `4`, with small oscillations due to the different path lengths (however, the final plot can be different because of the internal random state employed by Keras). To see a confirmation, let's generate the trajectories for all of the possible initial states, using the greedy policy (equivalent to $\varepsilon = 0$):

```
import numpy as np

trajectories = []
tunnels_c = []

for i, j in xy_start:
    tunnel = reset_tunnel()

    prev_value = 0.0

    trajectory = [[i, j, -1]]

    tunnel_c = tunnel.copy()
    tunnel[i, j] = 1.0
    tunnel_c[i, j] = 1.0

    final = False
    e = 0

    while not final and e < max_steps:
        e += 1

        q_value = get_Q_value(tunnel)
        action = np.argmax(q_value)

        if action == 0:
            if i == 0:
                x = 0
            else:
                x = i - 1
            y = j

        elif action == 1:
            if j == width - 1:
                y = width - 1
            else:
                y = j + 1
            x = i

        elif action == 2:
            if i == height - 1:
```

```
                x = height - 1
            else:
                x = i + 1
            y = j

        else:
            if j == 0:
                y = 0
            else:
                y = j - 1
            x = i

        trajectory[e - 1][2] = action
        trajectory.append([x, y, -1])

        tunnel[i, j] = prev_value

        prev_value = tunnel[x, y]

        tunnel[x, y] = 1.0
        tunnel_c[x, y] = 1.0

        i = x
        j = y

        final = is_final(x, y)
    trajectories.append(np.array(trajectory))
    tunnels_c.append(tunnel_c)
trajectories = np.array(trajectories)
```

Twelve random trajectories are shown in the following figure:



Twelve trajectories generated using the greedy policy

The agent always follows the optimal policy, independent from the initial state, and never ends up in a well. Even if the example is quite simple, it's helpful to introduce the reader to the concept of deep Q-learning (for further details, the reader can check the introductory paper, *Deep Reinforcement Learning*: *An Overview*, *Li Y.*, *arXiv:1701.07274 [cs.LG]*).

In a general case, the environment can be a more complex game (like Atari or Sega), and the number of possible actions is very limited. Moreover, there's no possibility to employ random starts, but it's generally a good practice to skip a number of initial frames, in order to avoid a bias to the estimator. Clearly, the network must be more complex (involving convolutions to better learn the geometric dependencies), and the number of iterations must be extremely large. Many other tricks and specific algorithms can be employed in order to speed up the convergence, but due to a lack of space, they are beyond the scope of this book.

However, the general process and its logic are almost the same, and it's not difficult to understand why some strategies are preferable, and how the accuracy can be improved. As an exercise, I invite the reader to create more complex environments, with or without checkpoints and stochastic rewards. It's not surprising to see how the model will be able to easily learn the dynamics with a sufficiently large number of episodes. Moreover, as suggested in the Actor-Critic section, it's a good idea to use Tensorflow to implement such a model, comparing the performances with Q-learning.

Summary

In this chapter, we presented the natural evolution of TD(0), based on an average of backups with different lengths. The algorithm, called TD(λ), is extremely powerful, and it assures a faster convergence than TD(0), with only a few (non-restrictive) conditions. We also showed how to implement the Actor-Critic method with TD(0), in order to learn about both a stochastic policy and a value function.

In further sections, we discussed two methods based on the estimation of the *Q* function: SARSA and Q-learning. They are very similar, but the latter has a greedy approach, and its performance (in particular, the training speed) results in it being superior to SARSA. The Q-learning algorithm is one of the most important models for the latest developments. In fact, it was the first RL approach employed with a Deep Convolutional Network to solve complex environments (like Atari games). For this reason, we also presented a simple example, based on an MLP that processes a visual input and outputs the *Q* values for each action.

The world of RL is extremely fascinating, and hundreds of researchers work every day to improve algorithms and solve more and more complex problems. I invite the reader to check the references in order to find useful resources that can be exploited to obtain a deeper understanding of the models and their developments. Moreover, I suggest reading the blog posts written by the Google DeepMind team, which is one of the pioneers in the field of deep RL. I also suggest searching for the papers freely available on *arXiv*.

I'm happy to end this book with this topic, because I believe that RL can provide new and more powerful tools that will dramatically change our lives!

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

Feature Engineering Made Easy
Sinan Ozdemir, Divya Susarla

ISBN: 978-1-78728-760-0

- Identify and leverage different feature types
- Clean features in data to improve predictive power
- Understand why and how to perform feature selection, and model error analysis
- Leverage domain knowledge to construct new features
- Deliver features based on mathematical insights
- Use machine-learning algorithms to construct features
- Master feature engineering and optimization
- Harness feature engineering for real-world applications through a structured case study

Machine Learning Solutions
Jalaj Thanaki

ISBN: 978-1-78839-004-0

- Select the right algorithm to derive the best solution in ML domains
- Perform predictive analysis efficiently using ML algorithms
- Predict stock prices using the stock index value
- Perform customer analytics for an e-commerce platform
- Build recommendation engines for various domains
- Build NLP applications for the health domain
- Build language generation applications using different NLP techniques
- Build computer vision applications such as facial emotion recognition

Leave a review - let other readers know what you think

Please share your thoughts on this book with others by leaving a review on the site that you bought it from. If you purchased the book from Amazon, please leave us an honest review on this book's Amazon page. This is vital so that other potential readers can see and use your unbiased opinion to make purchasing decisions, we can understand what our customers think about our products, and our authors can see your feedback on the title that they have worked with Packt to create. It will only take a few minutes of your time, but is valuable to other potential customers, our authors, and Packt. Thank you!

Index

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z