POLITECNICO DI MILANO
School of Industrial and Information Engineering Master of Science in Mathematical Engineering

IMPACT OF SENTIMENT ANALYSIS ON AUTOMATIC FINANCIAL TRADING THROUGH REINFORCEMENT LEARNING

Supervisor: Prof. Marcello Restelli

Co-supervisor: Dott. Lorenzo Bisi
Dott. Luca Sabbioni

Candidate:
Paolo Bonetti
893082

Academic Year 2018-2019

A chi c'è sempre stato.

Contents

Ringraziamenti **x**

Sommario **xiii**

Abstract **xv**

1 Introduction **1**

1.1 Outline of Contents . . . 2

2 Sequential Decision Making **4**

2.1 Sequential Decision Making . . . 7

2.2 Markov Decision Processes . . . 10

2.2.1 Problem Definition . . . 10

2.2.2 Return . . . 12

2.2.3 Policy . . . 14

2.2.4 Value Functions . . . 14

3 Reinforcement Learning **20**

3.1 Dynamic Programming . . . 21

3.1.1 Prediction: Policy Evaluation . . . 21

3.1.2 Control: Policy Iteration and Value Iteration . . . 22

3.2 Value-Based Reinforcement Learning . . . 24

3.2.1 Model-free Prediction . . . 25

3.2.2 On-Policy Model-Free Control . . . 28

3.2.3 Off-Policy Model-Free Control . . . 29

3.2.4 Fitted Q-Iteration (FQI) . . . 31

3.3 Policy Search Reinforcement Learning . . . 33

3.3.1 Policy Gradient Methods . . . 34

3.3.2 Trust Region Policy Optimization (TRPO) . . . 37

3.3.3 Proximal Policy Optimization (PPO) 40

4 Natural Language Processing 42

4.1 Traditional Approach: Bag of Words Model 43

4.2 Deep Learning 44

4.2.1 Artificial Neural Networks 45

4.2.2 Feedforward Neural Networks 45

4.2.3 Recurrent Neural Networks 50

4.2.4 Long Short Term Memory (LSTM) 52

4.3 Word Embedding 57

4.3.1 Feedforward Neural Net Language Model (NNLM) 57

4.3.2 Recurrent Neural Net Language Model (RNNLM) 60

4.3.3 Word2vec 62

4.3.4 Global Vectors for Word Representation (GloVe) 67

5 Datasets 70

5.1 S&P 500 Index 70

5.2 Daily Sentiment 74

5.2.1 Twitter Daily Sentiment 74

5.2.2 Reuters Daily Sentiment 76

5.3 Feature Extraction 78

6 Feature Selection 82

6.1 Features and Target: Expanded Dataset 83

6.2 Random Forests 86

6.3 Feature Selection with Random Forests 89

6.3.1 Random Forest Training 90

6.3.2 Random Forest Results 92

6.4 Regression of Sentiment on Residuals 95

6.4.1 The Procedure 95

6.4.2 Training and Results 96

7 RL Models and Results 99

7.1 MDP Models 100

7.2 Algorithms 103

7.2.1 The Procedure 103

7.2.2 Trajectories and Parameters 104

7.2.3 Performance Measures 105

7.3 Results . . . 106
7.3.1 Training . . . 106
7.3.2 Testing . . . 109
7.4 15 Minutes Data . . . 115
7.4.1 Datasets . . . 115
7.4.2 Feature Selection . . . 116
7.4.3 Reinforcement Learning . . . 121

8 Conclusion **123**
8.1 Results . . . 123
8.2 Future Improvements . . . 124

Appendices **125**

A Reinforcement Learning Performances **126**

B 15 Minutes Results **133**
B.1 Feature Selection Results . . . 133
B.2 Reinforcement Learning Results . . . 138

References **145**

List of Figures

2.1 Supervised Learning mainly deals with problems of Regression (2.1a) and Classification (2.1b): the first tries to predict a real value while the latter tries to predict the group a feature belongs to. 5
2.2 Unsupervised Learning addresses the problem to group the input data in classes where they are similar (Clustering, 2.2a) or to project them in lower dimensional spaces (Dimensionality Reduction, 2.2b). 6
2.3 Scheme of the interaction between agent and environment at time t. 8

3.1 Scheme of Policy Iteration algorithm: it is a sequence of a policy evaluation and a greedy policy improvement steps until the optimum is reached. 23

4.1 Scheme of a neuron, the elementary cell of an Artificial Neural Network. 46
4.2 An example of fully connected Feedforward Neural Network with two hidden layers. 48
4.3 An example of Recurrent Neural Network with two hidden layers, three-dimensional data and two-dimensional memory vector. 51
4.4 The unrolled version of a Recurrent Neural Network: at time t it takes as input the memory signal of previous step c_{t-1} and the current datum x_t. Then, hidden layers (represented by the blue circle) combine them producing the current signal, that is elaborated to produce the output $h_t = \hat{f}(x_t)$ and the next memory signal c_t. . . . 52
4.5 The unrolled version of LSTM: at time t it takes as input the memory signal of previous step c_{t-1}, the previous prediction h_{t-1} and current datum x_t. Then, four specific hidden layers combine them producing the current memory signal c_t, and the prediction $h_t = \hat{f}(x_t)$. 53
4.6 The four gates of the LSTM network. 56
4.7 An example of NNLM with $n = 3$ context words. 59

4.8 The network of the RNNLM. 61
4.9 Scheme of Skip-Gram model with $n = 2$. Different colours of arrows are related to different context words. 63
4.10 Examples of regularities applying *Word2vec* algorithms. 66

5.1 Pie Chart of the sector subdivision of the index. 72
5.2 The first five market data. 73
5.3 The curve of the open value of the index in the available data. . . . 74
5.4 First five Twitter sentiment features referring to the two previous days. 76
5.5 An extract of a Reuters Machine Readable News: every news is equipped with a positive, negative and neuter sentiment score. Moreover, the tuple with key *assetName* encodes the asset the news is referring to, which in this case is the *NGEx Resources Inc.* 77
5.6 First five Reuters sentiment features referring to the two previous days. 78
5.7 The first five extracted features. 79
5.8 Five examples of proportional consecutive differences of open values of the index among ten previous days. 80
5.9 Five rows of the dataset with all the available features derived both from market and sentiment data. 81

6.1 One row of the dataset is transformed into nine different samples: each one has a different combination of action and portfolio, leading to a different target reward. 85
6.2 A Decision Tree with *age* and *height* as features and *weight* as target. Each set partitioning the data space is colored as the corresponding leaf node. 87
6.3 An example of Random Forest with $n = 3$ Decision Trees. 89
6.4 The blue line in the figures is the value of the action in each sample, respectively belonging to train and test set. The red points are the corresponding predicted rewards. From their values is clear that they are positive when the action is 1, negative when it is -1 and slightly below 0 when the action is 0. 93
6.5 Confusion matrices show the number of samples whose sign has been correctly predicted and the number of misclassified samples. 94
6.6 Histogram of importance of best 15 features (as ranked in Table 6.2) with standard deviations. 94

7.1 Training curves of PPO and TRPO algorithms. Each point of a curve is the mean train return over 49 days: a value of 0.03 corresponds to a mean gain of the 3%. The data related to the year reported in the legend of each plot are the test set referring to that curve, while the data of other years from 2009 to 2018 are its training set. 107
7.2 Baseline: average daily reward in each year always performing action "1". 111
7.3 Confidence intervals for the mean daily reward computed in 5 different trainings of PPO algorithm for each of the 10 validations and for the final test. 112
7.4 Confidence intervals for the mean daily reward computed in 5 different trainings of TRPO algorithm for each of the 10 validations and for the final test. 113
7.5 Confidence intervals for the mean daily reward computed in 5 different trainings of FQI algorithm for each of the 10 validations and for the final test. 114

B.1 The blue line in the figures represent the value of the action in each sample, respectively belonging to train and test set, which can be 1, -1 or 0. The red points are the corresponding predicted rewards. From their values is clear that they are always negative independently from the action. 134
B.2 Histogram of importances of best 15 features for each of the two Random Forests with related standard deviations. 135
B.3 The blue line in the figures represent the value of the action in each sample, respectively belonging to train and test set, which can be 1, -1 or 0. The red points are the corresponding predicted rewards computed without considering transaction costs. 136
B.4 Histogram of importances of best 15 features with related standard deviations for each of the two Random Forests which consider the reward without transaction costs. 137

B.5 Three different baseline performance scores are reported in these figures: the green points are the best possible average rewards achievable in the four validation sets and in testing, computed by always selecting the best possible action. The red and blue points are the mean rewards obtainable always performing respectively the trivial actions -1 and +1. These two baselines are repeated in the second figure, which focuses only on them, to show their trend which is not clearly understandable in first figure since they have an order of magnitude less than the other baseline. 138
B.6 TRPO 15 minutes mean reward in validation and testing, repeating the same model five times and computing the related confidence intervals. 139
B.7 PPO 15 minutes mean reward in validation and testing, repeating the same model five times and computing the related confidence intervals. 140
B.8 FQI 15 minutes mean reward in validation and testing, repeating the same model five times and computing the related confidence intervals. 141
B.9 TRPO 15 minutes mean reward in validation and testing, without considering transaction costs. 142
B.10 PPO 15 minutes mean reward in validation and testing, without considering transaction costs. 143
B.11 FQI 15 minutes mean reward in validation and testing, without considering transaction costs. 144

List of Tables

4.1 Examples of co-occurrence probabilities and their ratio. 67

5.1 The 10 companies with largest market capitalization. 71

6.1 Parameters tuning: the best parameters and the related Accuracy score. 92
6.2 Feature ranking and performance of the Random Forest. 95
6.3 First Random Forest: same best parameters of Feature Selection and similar performance. 97
6.4 Second Random Forest: best parameters and related performance. . 97

7.1 Parameters selected for the Random Forest Regression in FQI. . . . 105
7.2 FQI daily train and validation average reward for each model and for each validation set. In all iterations from 1 to 10 the value is exactly the same, so only one is reported. 109
7.3 Best parameters and the related Cross-Validation performance; train and test accuracy scores of the final Random Forest Regressions. . . 118
7.4 Regression of sentiment on residuals: accuracy score of the first Random Forests on the two datasets and related performance of the second Random Forest which tries to predict the residuals through sentiment principal components. 119
7.5 Target reward computed without fees: best parameters and the related Cross-Validation performance; train and test accuracy scores of the final Random Forest Regressions. 120

A.1 PPO Validation and Test reward performance. 127
A.2 PPO Validation and Test return performance. 128
A.3 TRPO Validation and Test reward performance. 129
A.4 TRPO Validation and Test return performance. 130
A.5 FQI Validation and Test reward performance. 131

A.6 FQI Validation and Test return performance. 132

B.1 Feature ranking and performance of the two Random Forests trained using Reuters or Twitter fifteen minutes datasets. 135
B.2 Feature ranking and performance of the two Random Forests trained using Reuters or Twitter fifteen minutes datasets computing the target reward without considering transaction costs. 137

Ringraziamenti

Con questa tesi si conclude un lungo percorso da studente, con molti sacrifici e soddisfazioni, per questo desidero ringraziare tutti quelli che hanno condiviso parti di questo viaggio con me.

Il primo ringraziamento va al Professor Restelli, che mi ha trasmesso la passione per il Machine Learning durante il suo corso e mi ha dato l'opportunità di partecipare a questo progetto, seguendone gli sviluppi con attenzione e supporto. Ringrazio anche Luca e Lorenzo per l'aiuto puntuale e preciso che mi hanno fornito, mostrando grande serietà nel loro lavoro ma anche capacità di sdrammatizzare quando necessario. Un ringraziamento infine a Banca IMI per l'opportunità concessami e a CGnal, in particolare nella figura di Mattia Pedrini, per aver condiviso con me il loro progetto.

Vorrei poi ringraziare i miei genitori, che mi hanno dato la libertà e il sostegno di intraprendere la mia strada, anche quando prevedeva di attraversare mezza Europa in auto.

Grazie di cuore a Fabiola, per gli otto anni trascorsi insieme, per tutte le esperienze che abbiamo vissuto, per avermi reso quello che sono e per tutto l'aiuto con questa tesi, non ce l'avrei fatta senza. Grazie anche alla sua famglia, che mi ha accolto come un figlio.

Grazie a tutti gli amici di sempre: Povvo e Giulia che sono sopravvissuti a due anni di convivenza con me, Pampe e Vitto con cui anche quando non ci vediamo per mesi rimane la stessa complicità, Fede che è l'unica a non dare buca quando organizzamo le vacanze, Marty che mi sopporta dall'asilo ed è la compagna di banco che tutti vorrebbero, Verdi e i nostri tentativi di corsa che finiscono quasi sempre con un aperitivo. Grazie anche a chi ha condiviso questi anni universitari

con me, voglio citare Anna che mi ha sopportato quasi ogni giorno (e soprattuto nella preparazione di ARF), Chiara, Luca, Carolina, Martina, Federico, Paola.

Grazie infine a tutte le persone che non ho nominato ma che hanno fatto parte di questo viaggio, perchè ciascuno è stato importante con il suo contributo.

Sommario

Il *sentimento del mercato* (market sentiment) è un indice della fiducia media degli investitori sul mercato finanziario. Questo indice viene stimato ed utilizzato in quanto strettamente legato alla predizione dell'andamento dei prezzi degli strumenti finanziari: ci si aspetta che, se la prevalenza degli investitori è fiduciosa, le quotazioni azionarie avranno una tendenza al rialzo; se viceversa la maggioranza degli investitori è pessimista, il trend sarà al ribasso. Tuttavia è in generale complesso estrarre un indice di sentiment che sia robusto ed in linea con quello che accadrà sul mercato, in quanto questo dipende da molti fattori, perciò svariati metodi di calcolo sono presenti in letteratura. Con lo sviluppo delle Reti Neurali e dell'Elaborazione del Linguaggio Naturale, uno dei possibili stimatori del sentimento del mercato è basato su metodi di *Sentiment Analysis*. Esso consiste nell'estrazione di un indice di sentiment a partire da news o testi riguardanti le compagnie quotate o il mercato finanziario stesso.

In questo progetto vengono analizzati indici di sentiment estratti da news di Reuters e tweets provenienti da Twitter riguardanti le compagnie che compongono l'indice *S&P 500*. In particolare, nella prima parte della tesi si effettua un'ampia analisi delle principali tecniche di Reinforcement Learning e di Natural Language Processing. Successivamente, dopo un'attenta analisi degli indici di sentiment disponibili e delle serie storiche dell'*S&P 500*, il presente lavoro si focalizza in primo luogo sulla determinazione dell'efficacia di questi indici di sentiment nella predizione del trend dell'*S&P 500* mediante l'utilizzo di metodi di Supervised Learning, con l'obiettivo finale di addestrare un agente di Reinforcement Learning in grado di fare trading generando un profitto.

Parole Chiave: Machine Learning, Reinforcement Learning, Elaborazione del Linguaggio Naturale, Reti Neurali, Sentiment Analysis, Trading Automatico

Abstract

Market sentiment is an index of investors attitude with respect to the financial market. An estimate of this index can be useful to traders since it is correlated with the trend of stock prices: if the majority of investors has a positive sentiment, stock prices will probably have an increasing trend; on the other hand, if the common feeling of investors is negative, a downward trend is expected. However, it is complex to design a robust estimator of the market sentiment able to predict the market trend, since it depends on many different factors, therefore several estimators are applied in the literature. In the last decade, with the growth of the applications based on Artificial Neural Networks and of Natural Language Processing algorithms, one of the methods introduced to estimate the market sentiment is the application of *Sentiment Analysis* methods. In particular, they are exploited to extract a sentiment index based on news and documents concerning listed companies or the financial market in general.
This project analyses sentiment values extracted from Reuters news and from tweets of Twitter regarding the companies belonging to the *S&P 500* index. Specifically, in the first part of this thesis an overview on main Reinforcement Learning models and algorithms and on Natural Language Processing techniques is proposed. Then, after an extensive analysis of the available sentiment features and of the historical series of the *S&P 500*, the focus of the present work is firstly on the determination of the efficacy of these sentiment features on the prediction of the trend of the *S&P 500* index through Supervised Learning methods, with the final purpose to train a Reinforcement Learning agent able to profitably trade on the U.S. Stock Market.

Parole Chiave: Machine Learning, Reinforcement Learning, Natural Language Processing, Artificial Neural Network, Sentiment Analysis, Automatic Trading

Chapter 1

Introduction

Every second, on average, 6000 tweets are published on Twitter, which corresponds to 500 million tweets per day. Reuters provides one of the most advanced services for the analysis of news, where each news is already encoded with a sentiment signal. Therefore a reasonable idea is to elaborate the information coming from news and tweets related to the U.S. Financial Market to try to predict the future trend of the Market itself. It is indeed intuitive to state that if the overall sentiment of people or news is positive, stock prices will increase since people are confident and they will buy on the Market, while if the sentiment is negative, stock prices will decrease because people will tend to sell their stocks.
In the last two decades, Machine Learning has experienced exponential growth thanks to the improvement of the computing infrastructures, which made possible both to store large amounts of data and to use them to train algorithms with improved computational speed. Moreover, Machine Learning techniques are the most logical tools for the extraction of information from texts and large amounts of tweets, which can be useful for predicting Market trends.
Despite the enthusiasm due to the many outstanding results of the application of Machine Learning algorithms on real problems, it is necessary to provide statistical evidence on the effectiveness of the application of such algorithms on the specific problem. In particular, two questions arise regarding the application of Machine Learning to exploit sentiment features from tweets or Reuters news with the purpose of predicting the Market trend. The first question is about the cause-effect relationship between sentiment and Market trend: is the sentiment from Twitter or news the cause of a future change in the trend of a Market index or is the trend of the index affecting the sentiment of tweets and news? The other question is whether the selected sentiment values are consistent estimators of the real Market sentiment, which is a random variable that depends on many factors. Indeed, it may be possible that the selected sources or the extraction

process produce an estimate that is not relevant for the prediction.

This thesis investigates the importance of peculiar sentiment features based on tweets and Reuters news referring to a given U.S. Market index, the *S&P 500*. Specifically, the present work is developed on top of a project of Banca IMI led by CGnal consultants. The original project produces a daily signal that suggests traders to buy or sell financial instruments based on the *S&P 500* index, using as features the sentiment signals extracted from tweets and Reuters Machine Readable News through the application of Supervised Learning algorithms. In this work, the significance of such features on the prediction of the *S&P 500* index is investigated, mainly through the application of Reinforcement Learning algorithms. This choice is due to the fact that, if the performance of the algorithms is satisfactory, Reinforcement Learning allows trading stocks automatically, designing an automatic trader able to buy and sell stocks profitably. This thesis can be useful for exploring a different approach with respect to the one adopted in the main project, studying the significance of the available sentiment features from another point of view and designing automatic traders. In particular, this work compares the results obtained by applying Reinforcement Learning algorithms to models that have as features historical series of the index, the available sentiment features or both, in order to conclude whether sentiment features add informativeness in predicting the index or the same results can be produced just by observing its historical series.

1.1 Outline of Contents

The structure of the present thesis is explained in this section, which describes the topics covered in each chapter. In particular, they can be divided into two groups: Chapter 2, 3 and 4 give a theoretical overview of the state of the art of Reinforcement Learning and Natural Language Processing, also introducing Artificial Neural Networks, while Chapter 5, 6 and 7 show applications of the described techniques on the available data.

Chapter 2 introduces Machine Learning in general, defining its three main subfields: Supervised, Unsupervised and Reinforcement Learning. Then, the focus is on Reinforcement Learning models: Sequential Decision Processes, in general, are introduced and Markov Decision Processes (MDPs) in particular are largely discussed.

After the definitions of the processes that allow modeling problems in terms of MDPs, Chapter 3 introduces the main algorithms designed to solve them. In particular, algorithms based on Dynamic Programming are firstly explained, then Reinforcement Learning algorithms are discussed, which describe both Value Search and Policy Search methods, with a particular focus on the three algorithms applied in the thesis: TRPO,

PPO and FQI.

Chapter 4 theoretically introduces Word Embedding procedures (especially explaining *Word2vec and GloVe* algorithms), which are methods designed to transform words into vectors, so that they become understandable by computers. Furthermore, in this chapter, Artificial Neural Networks are described: they are a powerful Supervised Learning technique exploited by some Word Embedding algorithms and they are also useful for understanding some applications performed in the following chapters.

Chapter 5 introduces the available datasets: historical series of the *S&P 500* index and daily sentiment from Twitter and Reuters. Some additional features are also extracted starting from the available ones.

In Chapter 6 Supervised Learning algorithms, specifically Random Forest Regression, are applied to explore the importance of the available features for reward prediction. In particular, two procedures are followed: a Random Forest that ranks all the available features and another Random Forest-based procedure more focused on the sentiment features.

In Chapter 7 three models are described as MDPs, the three selected Reinforcement Learning algorithms are applied to them and testing results are shown and commented on. Additional performance measures regarding the testing of these algorithms can be found in Appendix A. Moreover, Section 7.4 presents an overview of the application of all the techniques applied in Chapter 5, 6 and in previous sections of Chapter 7 on datasets with data available every fifteen minutes, whose main results are reported in Appendix B.

Finally, in Chapter 8, the main conclusions drawn through the thesis are summarized and some possible improvements are discussed.

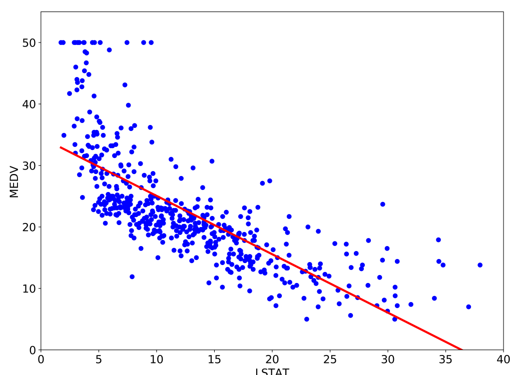
Chapter 2

Sequential Decision Making

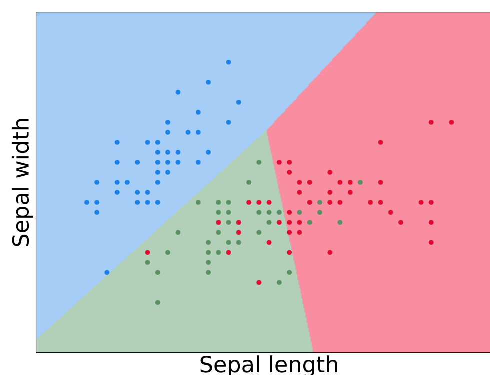
Machine Learning (ML) is a branch of computer science and a sub-field of Artificial Intelligence, whose aim is to extract information from data that can be used in order to make decisions on new data. In particular, ML can be divided into three main sub-fields:

- **Supervised Learning**: given a dataset $D = \{(x_i, t_i) \mid i = 1, 2, ..., n\}$, called **training set**, where each element (*sample*) is a tuple (x_i, t_i), of which x_i is the vector of input features and t_i is the related target output, the goal in Supervised Learning is to estimate the unknown model (basically a function) that produces the output t_i from its related input x_i. Estimating this function is useful in order to produce the correct output given a new set of inputs (the **test set**).
  The main problems of Supervised Learning can be divided into three groups:

  1. **Classification**, where the target is one of K discrete classes and the goal is to assign each input $x_i \in \mathbb{R}^m$ to a class. An example of Classification between two classes can be to decide if a human being is healthy or sick given some medical tests, while a multi-class problem can be to decide the nationality of a person given a picture of her face.
  2. **Regression**, where the output is a continuous number $t_i \in \mathbb{R}$ and the goal is to learn a mapping from the input $x_i \in \mathbb{R}^m$ to the target t_i. Applications of Regression can be predict Stock Market prices given some context features, predict the age of a person given her picture, or predict the value of a house knowing its location and dimension.
  3. **Probability Estimation**, where the target is a probability distribution over some possible events and the goal is to learn a mapping able to associate each input $x_i \in \mathbb{R}^m$ to a probability distribution over some possible events. For example, if the purpose is to predict the next word in a sentence given the n previous ones, it outputs a probability distribution over all words.



(a) An example of Regression: taking the percentage of lower status population as input, the algorithm predicts the median value of houses in thousands of dollars.

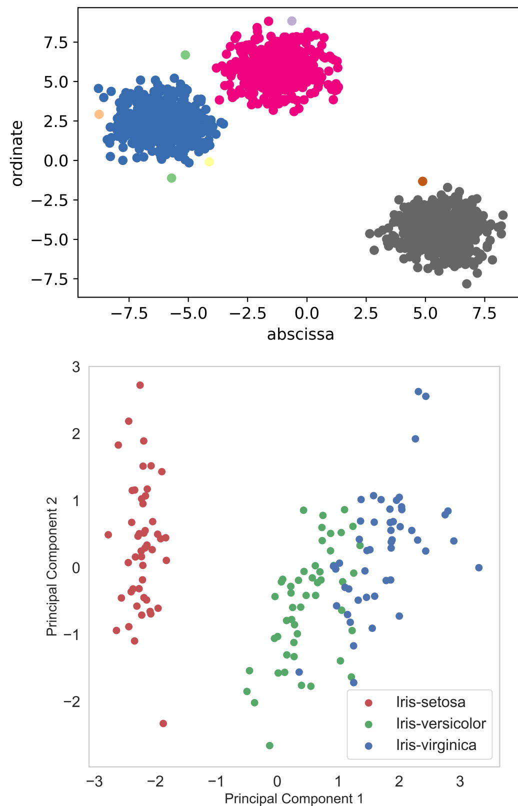


(b) An example of Classification: given the length and the width of the sepal of iris flowers, the algorithm classifies each flower in one of three different species.

Figure 2.1: Supervised Learning mainly deals with problems of Regression (2.1a) and Classification (2.1b): the first tries to predict a real value while the latter tries to predict the group a feature belongs to.

- **Unsupervised Learning**: given a training set $D = \{x_i \mid i = 1, 2, ..., n\}$, made only by a set of inputs $x_i \in \mathbb{R}^m$, the goal of Unsupervised Learning is to compute an efficient representation of the inputs. This kind of approach is called *unsupervised* since there is no target, the purpose is to learn something about the data and the relationships among them. This is done in two main directions:

  1. **Clustering**, where data are grouped by the algorithm in order to maximize the similarity among data in the same group and to minimize it among data in different groups. It is important to notice that this approach is very different from Classification, since here the groups are not given by a target but they are chosen by the algorithm exploiting the properties of the inputs. There are some examples in medicine, where Clustering algorithms can group images of tissues from health and sick people in order to find some unknown features determining the illness, or in web advertising, where algorithms can divide customers in different groups of people that probably like the same products.
  2. **Dimensionality Reduction**, that maps the input data into a lower dimensional space, through a transformation that may be linear or not depending on the method. The most famous example is *Principal Component Analysis (PCA)*, that projects the data in the directions that maximize their variance.



(a) An example of Clustering: some points form three bubbles in the Cartesian plane and a ten-groups clustering algorithm identifies clearly the three circles and other seven single points that can be considered outliers.

(b) An example of Dimensionality Reduction: applying PCA to the dataset already used in 2.1b where the inputs are now four (width and length of sepals and petals of the flowers), data are projected in two dimensions keeping the 96% of the original variance.

Figure 2.2: Unsupervised Learning addresses the problem to group the input data in classes where they are similar (Clustering, 2.2a) or to project them in lower dimensional spaces (Dimensionality Reduction, 2.2b).

- **Reinforcement Learning**: it aims to find a function which outputs the action leading to the best cumulative reward that an agent can take when observing a dynamic environment (usually modeled as a Markov Decision Process, which will be widely discussed in Section 2.1). In particular, the training set $D = \{(s_i, a_i, s_i', r_i) \mid i = 1, 2, ..., n\}$ is composed of 4-tuples where s_i represents the current state, a_i the chosen action, s_i' the next state and r_i the associated reward. Therefore, the idea of Reinforcement Learning is to learn an optimal policy $\pi^*(s)$ that associates to each state the optimal action to be performed on it. Basically, algorithms in this field learn the best actions that an agent can perform given the current state in order to maximize the long term reward. Some examples can be an algorithm that tells a robot how to move in a room in order to reach a certain position, an algorithm that trades actions on the Stock Market, or an algorithm able to win a chess game against a human.

More details and algorithms on Supervised and Unsupervised Learning can be found

in [7] and [47]. In this chapter and in Chapter 3 the focus is on Reinforcement Learning, that is the core field of study of this thesis ([42] is the reference book regarding RL). In particular, in Section 2.1, Sequential Decision Making (SDM) in general is presented. Then, in Section 2.2, the focus is on Markov Decision Processes, that are a particular and widely used class of SDM models. Finally, algorithms designed to solve known and unknown processes modeled as MDPs will be discussed in next chapter.

2.1 Sequential Decision Making

The general problem studied by Reinforcement Learning is how to select actions in order to maximize a numerical reward. The learner must decide the actions to perform based on the current state, so it has to try many actions in different states in order to decide which ones lead to better rewards. Moreover, actions do not affect only the next immediate reward, but they also determine the next state, so they influence all subsequent rewards and it may even happen that an action producing a little or negative immediate reward is the best one because it leads to a state where it is possible to gain a huge future reward. These two important aspects, long term consequences of actions (and the focus on the cumulative rewards, not only on the immediate one) and the exploration of many different states by performing different actions, make this kind of decision making problem challenging and difficult to optimize.

Another main aspect of these kind of problems, that is not in common with Supervised or Unsupervised Learning, is the trade-off between **exploration** and **exploitation**. In order to discover actions that lead to a better reward it is necessary to *explore* different state-action combinations, although trying actions never performed before may lead to some loss; on the other hand, actions tried in the past that produced a good reward might be preferred, so what is already known should be *exploited*. Many RL algorithms address this problem by trying many different actions and repeating them many times in order to have a reliable estimation of the expected reward of each action, then after many iterations they progressively become less explorative and start to focus on actions that appears to lead to best cumulative reward.

The basic structure of a Sequential Decision Making Problem is shown in Figure 2.3: at each step t the learner, called **agent**, receives from the outside (the **environment**) the observation o_t of current dynamics of the process. Then, the agent performs an action a_t and it finally observes the reward r_{t+1} and next observation o_{t+1}. The basic idea is that the most important features of the problem are summarized inside the observation, so that the agent is able to perform an action among a set of available actions A by observing o_t and knowing what happened in past iterations. Then it observes the imme-

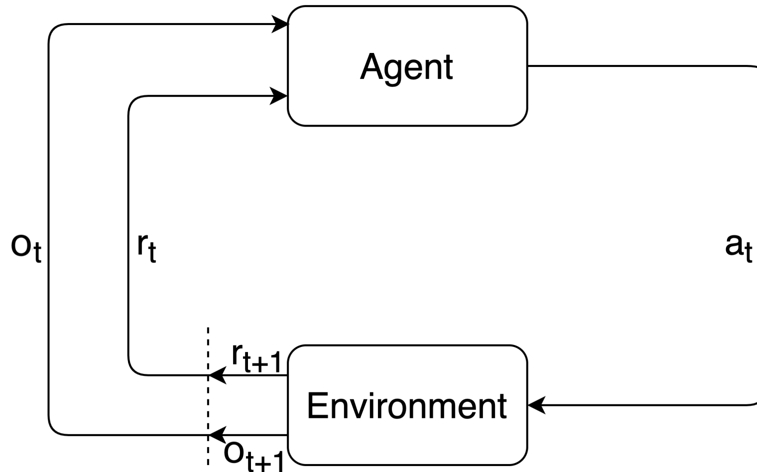


Figure 2.3: Scheme of the interaction between agent and environment at time t.

diate reward due to the performed action, remembering that is not sufficient to decide whether the selected action is good or not, since it will influence all future observations and rewards.

In this setting the **history** (h_t) is defined as the sequence of observation-action-reward at each timestep up to t: $h_t = \{a_1, o_1, r_1, ..., a_t, o_t, r_t\}$. This set is important since the next rewards and observations may depend on what happened before. Indeed, the agent will accordingly select next action and the environment will consequently produce next observation and reward based on the history h_t. The **state** s_t is the information used to determine what will happen next, so it is formally a function of the history:

$$s_t = f(h_t). \tag{2.1}$$

In general, the environment has a private representation of the state s_t^e, that is what it uses to produce the next tuple observation-reward. On the other hand, the agent has another representation of current state s_t^a, that is the information it uses to select the next action based on the history. Moreover, the environment can be **deterministic** or **stochastic**. In the first case the action performed by the agent univocally determines the next state and the reward, while in a stochastic environment the action does not uniquely determines the next state and reward, which may be different performing two different times the same action in the same state.

Summing up, in a general Sequential Decision Making Problem, a_t, o_t, r_t are exchanged between agent and environment at each iteration. They are added to the history of

the episode and both agent and environment have a private function that elaborates the history producing their state. The agent uses its own (s_t^a) to decide next action to perform, while the environment uses it (s_t^e) to elaborate next observation and reward. In order to simplify the problem, many Reinforcement Learning applications are designed using the hypothesis of **fully observable environment**, which means that the agent directly observes the environment state, so that:

$$o_t = s_t^a = s_t^e. \tag{2.2}$$

This is the hypothesis that will be considered from now on in this thesis, since the discussion of main aspects of RL is made easier, because all that is needed to make decisions is explicitly given to the agent. However, in some problems it is not possible to impose the environment to be fully observable: they are called **Partially Observable** Problems and their complexity is that, since the agent does not know exactly the current state, it must consider a probability distribution among all possible states at each iteration, called **belief** [24].

In conclusion of this overview of Sequential Decision Making, that is the basic structure of a Reinforcement Learning Problem, there are some examples that show real world applications which can be modeled as SDM problems:

- *Rubik's cube*: it has a finite number of possible configurations (states) and each action, that is a move of a side of the cube, leads to a deterministic state.
- *Chess*: it is deterministic and finite but the number of possible states is very huge, so in practice it is not possible to explore every state-action pair.
- *Blackjack*: it is still a game with a finite number of states but the transition from one state to the next one is stochastic, since it is not known what will be the next card.
- *Pole balancing*: it is a deterministic problem but the possible states (that can be modeled with position, velocity, angle of the pole, derivative of the angle) are infinite, since position, velocity and angle are continuous variables.

The examples above are all problems where RL algorithms are useful. In fact the dynamics of the environment are unknown (next card of a deck), difficult to be solved exactly (the position of the pole is a system of partial differential equations) or the model of the environment is too complex to be explicated (the game tree of chess is 10^{123}, it is not possible to enumerate all possible solutions in order to find the best one). The power of RL algorithms is that they address this kind of problems by exploring some possible solution and maximizing the expected value of the cumulative reward without the need of the full knowledge of the problem.

2.2 Markov Decision Processes

In this section specific SDM problems called Markov Decision Processes (MDPs) are explained [4]. They are a subset of Sequential Decision Making problems that have some properties making them simpler. Moreover the majority of RL problems are modeled as MDPs, so they are very important and all applications presented in this thesis are modeled in this way.

2.2.1 Problem Definition

In an SDM problem the state signal s_t may contain all the information needed to choose the action to perform a_t. This kind of state is said to have the *Markov property*, which intuitively means that "the future is independent from the past given the present". Hence, in order to make the next decision and to elaborate next state and the reward, it is enough to use the information given by the actual state, without looking back to past states.

Definition 2.2.1 (Markov Property)**.** A stochastic process X_t is said to be **Markovian** if:

$$\mathbb{P}(X_{t+1} = j | X_t = i, X_{t-1} = k_{t-1}, ..., X_1 = k_1, X_0 = k_0) = \mathbb{P}(X_{t+1} = j | X_t = i). \quad (2.3)$$

The formal Definition 2.2.1 can be applied to SDM problems considering the state signal to be a stochastic process S_t with a certain probability distribution defined over all states to be in that state given the history, so if Markov property holds:

$$\mathbb{P}(S_{t+1} = s' | A_t, S_t, S_{t-1}, A_{t-1}, , ..., S_1, A_1, S_0, A_0) = \mathbb{P}(S_{t+1} = s' | S_t, A_t). \quad (2.4)$$

This means that a state signal has the Markov property if it is enough to know last state and the action performed in order to know the probability distribution over all states about what will be the next one.
Moreover, the probabilities $\mathbb{P}(S_{t+1} | S_t, A_t)$ are called **transition probabilities**. A common assumption made on RL environments is that the transition probabilities are **stationary**.

Definition 2.2.2 (Stationary Transition Probabilities)**.** Transition probabilities in a Markovian stochastic process X_t are said to be stationary if they are time invariant. In this case:

$$\mathbb{P}(X_{t+1} = j | X_t = i) = \mathbb{P}(X_1 = j | X_0 = i) = p_{i,j}. \quad (2.5)$$

The Markov property in RL is particularly important mainly for two reasons. Firstly, it is a mathematical property that simplifies real world situations, assuming they are fully described by last state, but this simplification is usually reasonable and it allows to formulate many theoretical results. The second reason is a memory issue: for a long episode it is much faster and memory efficient to store and use the last state signal without the need of all previous ones, which is possible if Markov property holds.

A Reinforcement Learning problem that satisfies the Markov property and it is fully observable is called **Markov Decision Process (MDP)** and it is formalized in Definition 2.2.3 [34].

Definition 2.2.3 (Markov Decision Process)**.** A Markov Decision Process (MDP) is a Markovian stochastic process defined by the tuple $< S, A, P, R, \gamma, \mu >$, where:

- S is the (finite) set of all possible states;
- A is the (finite) set of all possible actions;
- P is the stationary transition probability matrix defining $\mathbb{P}(s'|s, a)$;
- R is the reward function $R(s, a) = \mathbb{E}[r|s, a]$. The reward in a state-action pair (s, a) is defined as the expected value of the random variable r since in general the reward can be stochastic, meaning that performing the same action in the same state does not always lead to the same reward;
- γ is the discount factor $\gamma \in [0, 1]$;
- μ is the probability distribution over all states modeling the probability of a state to be the initial one (in the finite case μ is the set of initial probabilities $\mu_i = P(s_0 = i),\ \forall i \in S$).

Remark. In the continuous case (or more generally when the MDP is not finite) the transition matrix is generalized by a probability density function $t(s, a, s')$ and the initial probability by another probability density function $g(s)$, such that:

$$\int_{S'} t(s, a, s')ds' = \mathbb{P}(s_{t+1} \in S'|s_t = s, a_t = a), \tag{2.6}$$

which denotes the probability that next state is in the region S' given the state-action pair (s,a);

$$\int_{S_0} g(s)ds = \mathbb{P}(s_0 \in S_0), \tag{2.7}$$

defining the probability that the initial state is in the region S_0.

Summing up, the MDP is defined by: all the states and actions that the agent can perform; all transition probabilities to move in a state given the previous state-action pair; the reward function that associates to each tuple state-action the expected immediate reward; the probability distribution that associates to each state the probability to start an episode in it; the discount factor, which represents the idea that immediate reward is better than waiting for it, whose importance is widely discussed in next subsection. Finally, if the sets of states and actions are finite the process is called **finite** MDP.

2.2.2 Return

After the definition of the problem, it is necessary to precisely define what is the goal of an agent. As already discussed, it is not enough to consider as goal the next scalar reward. For example, there can exist an action with a little or negative immediate reward leading to a state where it is possible to gain a huge reward at next iteration. The easiest idea is to consider as goal for the agent the maximization of the cumulative reward, that can be expressed as the sum of all the rewards received in an episode. In RL there exist different reward functions used to measure the cumulative reward:

- **total reward**, that is exactly the sum of all rewards received during the episode:

$$V = \sum_{i=0}^{\infty} r_{i+1}; \tag{2.8}$$

- **average reward**, that is the mean reward received in the episode:

$$V = \lim_{n \to \infty} \frac{r_1 + ... + r_n}{n}; \tag{2.9}$$

- **discounted reward**, that is the sum of the rewards discounted by the discount factor:

$$V = \sum_{i=0}^{\infty} \gamma^i r_{i+1}, \tag{2.10}$$

  which has the advantage to be bounded by $\frac{r_{max}}{1-\gamma}$ if the reward function $R(s, a)$ is bounded, so that there are no divergence issues in the sum when an episode is infinite.

Among the possible cumulative reward functions, the most widely used in RL is the discounted reward, since it considers all the rewards and applies the discount factor γ to solve divergences of a sum in a more efficient way than the asymptotic mean. Moreover, it is important to consider the discount factor not only for mathematical convenience

but also for several intuitive reasons. First of all, the transition between a state and another is a stochastic process, so it is logical that the certain reward gained at the current iteration has more value than the stochastic one that may be possible to gain some iterations later, because there is more uncertainty on it. Another important reason is that, for example in finance, immediate rewards can lead to more interest than delayed ones, so it is particularly valuable in this field. More generally, animals prefer immediate reward rather than a delayed one: having something valuable as soon as possible is always preferred. These considerations highlight the importance of discounting the cumulative reward as done in Equation 2.10. However, when the MDP is finite and an episode that it generates is certainly finite too, it is also possible to use the undiscounted cumulative reward of Equation 2.8. This is done when the purpose is to consider all rewards to have the same importance. On the other hand, in applications where there can be infinitely long episodes and there is no discount factor, the cumulative reward is modeled as in Equation 2.9.

Taking inspiration from the discounted total reward, it is possible to introduce a key element in RL, the **return**.

Definition 2.2.4 (Return)**.** The return v_t is the total discounted reward from time-step t:

$$v_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}. \tag{2.11}$$

The return is important in evaluating the value of performing a certain action in a state of an MDP. The idea is that the value of the action taken at time t are all the rewards received from t to the end of the sequence, since the action influences all the next iterations. Moreover, the sum is discounted since the value of receiving a reward r after $k+1$ iterations is less than receiving it immediately, in particular it is considered to be $\gamma^k r$. By definition, the cumulative discounted reward is exactly the return from iteration 0, v_0, so the return is a generalization of the cumulative discounted reward that can be computed at any step. Indeed, the return is often computed during the learning process because it is possible to evaluate a different return in each time-step, that is more informative than using just one cumulative discounted reward to extract information from an entire sequence sampled following the MDP (this sequence is called **episode**).
It is important to notice that, because of the stochasticity of the process, the purpose of the agent is not simply to maximize the return v_t, but the **expected return** $\mathbb{E}[v_t|s_t = s]$ because, as already explained, the return v_t is a random variable since the model is a stochastic process, so next states and returns may differ repeating the same actions multiple times.

Summing up, starting from the idea of considering as goal of the agent the cumulative reward computed as sum of rewards it is often better to discount it. The discounted cumulative reward also leads to the introduction of the return, that is almost the same computation truncated from a certain time-step to the end of an episode. The return is important to describe the value of each state, in particular the expected return is used to take into account the stochasticity of the process, as will be explained in Section 2.2.4.

2.2.3 Policy

Once the problem has been defined in terms of MDPs and the goal of the agent as (discounted) cumulative reward, it remains to formalize what is meant by solution of the MDP. In ML the "result" is a **policy** function, that at any given state, outputs the action to perform.

Definition 2.2.5 (Policy)**.** A policy is a function $\pi : S \times A \to [0, 1]$ such that, for each state $s \in S$, it is a probability distribution over all possible actions:

$$\pi(a|s) = \mathbb{P}(a|s), \ \forall a \in A. \tag{2.12}$$

Remark. If the action space A of an MDP is continuous, in any state s, the policy $\pi(s)$ is generalized as a probability density function on the action space.

In a MDP it is possible to exclusively consider **stationary** and **Markovian** policies (depending only on the current state and action) since it is demonstrated that it always exists an optimal policy with these characteristics. Moreover, it can be **stochastic** or **deterministic** depending on the problem, but it will be shown that there always exists an optimal deterministic policy. Finally, a **trajectory** is a sequence of state-action pairs $\{s_0, a_0, s_1, a_1, ..., s_n, a_n\}$ obtained following the policy.

2.2.4 Value Functions

Many RL algorithms are focused on estimating how good is to be in a state. This can be done by knowing the expected cumulative reward (return) from that state to the end of the episode. Since the future rewards depend on the actions that the agent will perform in next states, the value of a state can be computed only with respect to a particular policy to follow when choosing actions. Therefore the idea of **value functions** is that it is possible to define the value of each state of the MDP given a policy π. This is logical, since the value of a state is not absolute but it depends on the action that is performed on it.

Definition 2.2.6 (State-Value Function)**.** The **state-value function** $V^\pi(s)$ of an MDP is the expected return from state s following policy π:

$$V^\pi(s) = \mathbb{E}_\pi[v_t|s_t = s]. \tag{2.13}$$

Another possible approach is to evaluate the value of each action in each state with respect to a policy π. In this way it is easier to understand what is the value of each action in each state, which can be useful when the algorithm has to choose the actions that are better to perform.

Definition 2.2.7 (Action-Value Function)**.** The **action-value function** $Q^\pi(s,a)$ of an MDP is the expected return from state s performing action a and then following policy π:

$$Q^\pi(s,a) = \mathbb{E}_\pi[v_t|s_t = s, a_t = a]. \tag{2.14}$$

State-value function and action-value function can be estimated from experience by producing many episodes following the policy, computing all returns and applying the law of large numbers that ensures the empirical mean to converge toward the expected value. On the other hand this kind of approach becomes impractical when the number of states and actions is huge.
Action-value function and state-value function satisfy particular recursive relationships called **Bellman equations** that are used in some RL algorithms.

Theorem 2.2.1 (Bellman Expectation Equation for State-Value Function)**.** *The state-value function can be decomposed into immediate reward plus discounted value of next state:*

$$\begin{aligned} V^\pi(s) &= \mathbb{E}_\pi[r_{t+1} + \gamma V^\pi(s_{t+1})|s_t = s] \\ &= \sum_{a \in A} \pi(a|s) \left(R(s,a) + \gamma \sum_{s' \in S} P(s'|s,a) V^\pi(s') \right). \end{aligned} \tag{2.15}$$

Remark. In a continuous MDP the sums in Equation 2.15 become integrals and both the policy $\pi(a|s)$ and the transition probabilities $P(s'|s,a)$ become probability density functions, as explained in previous Remarks 2.2.1, 2.2.3. In the following, the focus is on finite MDPs, therefore the theory is exposed using sums, keeping in mind that the same results can be reformulated in the continuous case.

In the first equivalence of Equation 2.15 the return at time t is rewritten as the immediate reward r_{t+1} plus the discounted value of the next state. In the second equivalence the expected value is computed explicitly: it is the sum over all actions of the probability to play a certain action in the current state $\pi(a|s)$ of the immediate reward $R(s,a)$ plus the discounted value of next state, that is computed iteratively using value functions.
A similar result holds for the action-value function.

Theorem 2.2.2 (Bellman Expectation Equation for Action-Value Function)**.** *The action-value function can be decomposed as immediate reward plus the discounted value of next*

state-action pair:

$$
\begin{aligned}
Q^\pi(s,a) &= \mathbb{E}_\pi[r_{t+1} + \gamma Q^\pi(s_{t+1}, a_{t+1}) | s_t = s, a_t = a] \\
&= R(s,a) + \gamma \sum_{s' \in S} P(s'|s,a) V^\pi(s') \\
&= R(s,a) + \gamma \sum_{s' \in S} P(s'|s,a) \sum_{a' \in A} \pi(a'|s') Q^\pi(s',a').
\end{aligned}
\tag{2.16}
$$

As in Theorem 2.2.1, the expected value is decomposed into the sum of immediate reward $R(s,a)$ plus the discounted value of next state, that is the sum over all states of their values weighted by transition probabilities. It is important to notice that, since the action is a parameter of the function, there is not the sum over probabilities of taking the action, so the equation is easier than the one for state-value function. Finally, this can be rewritten substituting the values $V^\pi(s')$ with the action-value function, considering that the value of the state s' is the sum over all possible actions $a' \in A$ of the action-value function $Q^\pi(s',a')$ weighted by the probability of taking that action in s', which is $\pi(a'|s')$.

Bellman equations are important in MDPs because they introduce a recursive relationship between value functions, allowing to express the value of a state in terms of the value of other states. This suggests the possibility to use iterative algorithms to compute these values.

It is possible to introduce two operators for the two Bellman equations that are useful to deduce some properties.

Definition 2.2.8 (Bellman Operator for State-Value Function)**.** Bellman operator for the state-value function is the linear operator $T^\pi : \mathbb{R}^{|S|} \to \mathbb{R}^{|S|}$ such that:

$$
(T^\pi V^\pi)(s) = \sum_{a \in A} \pi(a|s) \left(R(s,a) + \gamma \sum_{s' \in S} P(s'|s,a) V^\pi(s') \right). \tag{2.17}
$$

Definition 2.2.9 (Bellman Operator for Action-Value Function)**.** Bellman operator for the action-value function is the linear operator $T^\pi : \mathbb{R}^{|S| \times |A|} \to \mathbb{R}^{|S| \times |A|}$ such that:

$$
(T^\pi Q^\pi)(s,a) = R(s,a) + \gamma \sum_{s' \in S} P(s'|s,a) \sum_{a' \in A} \pi(a'|s') Q^\pi(s',a'). \tag{2.18}
$$

Bellman operators map value functions in value functions following Bellman equation recursive relationships, so it is possible to compactly rewrite the Bellman equations as:

$$
T^\pi V^\pi = V^\pi; \tag{2.19}
$$

$$T^\pi Q^\pi = Q^\pi. \tag{2.20}$$

These equations imply that the value functions are **fixed points** of Bellman operators. Moreover, they are linked with **linear** Equations 2.19 and 2.20 that, knowing the MDP, can be solved to find the value functions explicitly. Finally, if $\gamma \in (0, 1)$ then T^π can be proved to be a **contraction** with respect to the maximum norm.

Optimal Value Functions
It is possible to repeat all steps shown for value functions in a particular state-value function and a particular action-value function.

Definition 2.2.10 (Optimal State-Value Function)**.** The **optimal state-value function** $V^*(s)$ is the maximum state-value function over all policies:

$$V^*(s) = \max_\pi V^\pi(s). \tag{2.21}$$

Definition 2.2.11 (Optimal Action-Value Function)**.** The **optimal action-value function** $Q^*(s, a)$ is the maximum action-value function over all policies:

$$Q^*(s, a) = \max_\pi Q^\pi(s, a). \tag{2.22}$$

These two value functions are particularly important since they follow the best possible policy and they evaluate the best possible value of each state in the MDP. This is the reason why they are called optimal and, once the optimal value function and the policy followed by this function are known, the MDP can be considered solved.
Value functions also define a partial ordering over policies, since:

$$V^\pi(s) \geq V^{\pi'}(s) \; \forall s \in S \implies \pi \geq \pi'.$$

Starting from this consideration it is possible to define the **optimal policy** (π^*) as the policy that is always better or equal than any other policy. The optimal policy is not unique in an MDP but all optimal policies share the same (optimal) value functions $V^*(s),\ Q^*(s, a)$.
The properties of optimal policies are shown in the following theorem.

Theorem 2.2.3. *For any Markov Decision Process:*

- *There always exists at least one optimal policy π^* that is better or equal than any other policy: $\pi^* \geq \pi \; \forall \pi$;*
- *All optimal policies share the optimal state-value function: $V^{\pi^*}(s) = V^*(s)$;*
- *All optimal policies share the optimal action-value function: $Q^{\pi^*}(s, a) = Q^*(s, a)$;*

- *There always exists a* ***deterministic*** *optimal policy.*

In particular, a deterministic optimal policy can be found by maximizing over $Q^*(s,a)$*:*

$$\pi^*(a|s) = \begin{cases} 1 & if \ a = \arg\max_{a \in A} Q^*(s,a) \\ 0 & otherwise \end{cases}. \tag{2.23}$$

This is a very powerful result because it shows that there always exists a deterministic optimal policy, that is not only easier to perform but it is also easy to extract from optimal action-value function just by selecting in each state the action providing the most value. This result is applied in a large group of algorithms that focus their attention on the optimization of the action-value function, knowing that, having the best action-value function, it is straightforward to extract an optimal deterministic policy called **greedy policy**.

As in Theorems 2.2.1 and 2.2.2 optimal value functions can be expressed through specific Bellman equations.

Theorem 2.2.4 (Bellman Optimality Equation for State-Value Function)**.** *Bellman optimality equation for the optimal state-value function is:*

$$\begin{aligned} V^*(s) &= \max_{a \in A} Q^*(s,a) \\ &= \max_{a \in A} \left\{ R(s,a) + \gamma \sum_{s' \in S} P(s'|s,a) V^*(s') \right\}. \end{aligned} \tag{2.24}$$

In the first equivalence of the theorem the best value associated to a state is trivially equated to the maximum of the optimal action-value function in that state. Then the optimal action-value function is substituted by the immediate reward plus the sum over all states of the value of next one weighted by the transition probability of moving in that state, as already done in Theorem 2.2.2.

Theorem 2.2.5 (Bellman Optimality Equation for Action-Value Function)**.** *Bellman optimality equation for the optimal action-value function is:*

$$\begin{aligned} Q^*(s,a) &= R(s,a) + \gamma \sum_{s' \in S} P(s'|s,a) V^*(s') \\ &= R(s,a) + \gamma \sum_{s' \in S} P(s'|s,a) \max_{a' \in A} Q^*(s',a'). \end{aligned} \tag{2.25}$$

The two equivalences are derived through the same substitutions done for state-value function in Bellman optimality Equation 2.2.4.
It is important to notice that both Bellman optimality equations contain a maximization,

leading them to be **nonlinear**.
As done in Definitions 2.2.8 and 2.2.9 it is useful to introduce the Bellman optimality operators.

Definition 2.2.12 (Bellman Optimality Operator for State-Value Function)**.** The Bellman optimality operator for the optimal state-value function is $T^* : \mathbb{R}^{|S|} \to \mathbb{R}^{|S|}$ such that:

$$(T^* V^*)(s) = \max_{a \in A} \left(R(s,a) + \gamma \sum_{s' \in S} P(s'|s,a) V^*(s') \right). \tag{2.26}$$

Definition 2.2.13 (Bellman Optimality Operator for Action-Value Function)**.** The Bellman optimality operator for the optimal action-value function is $T^* : \mathbb{R}^{|S| \times |A|} \to \mathbb{R}^{|S| \times |A|}$ such that:

$$(T^* Q^*)(s,a) = R(s,a) + \gamma \sum_{s' \in S} P(s'|s,a) \max_{a' \in A} Q^*(s',a'). \tag{2.27}$$

These operators are **nonlinear**, hence solving the optimal Bellman equations is complex and does not exist a closed form solution as in Equations 2.19 and 2.20. On the other hand, $V^*(s)$ (respectively $Q^*(s,a)$) is still a **fixed point** of its operator T^*. Moreover, T^* is again a contraction with respect to the infinity norm.

All Bellman operators discussed in this section are contractions and value functions are their fixed points. Therefore it is possible to apply the *Banach-Caccioppoli fixed point theorem* [3] to show that each Bellman operator admits a unique fixed point (that is its related value function). Moreover the theorem states that considering a generic vector $f \in R^{|S|}$ and a policy π:

$$\lim_{k \to \infty} (T^\pi)^k f = V^\pi, \tag{2.28}$$

$$\lim_{k \to \infty} (T^*)^k f = V^*. \tag{2.29}$$

This result shows that, starting with an arbitrary vector that assigns a value to each state, it is enough to apply many times the Bellman (optimality) operator in order to learn the state-value function related to the policy π. The same result also holds for (optimal) action-value functions.

Summing up, in this section all the components of a Reinforcement Learning problem have been discussed. The problem has been formalized as an MDP, the goal of the agent has been explained in terms of (discounted) cumulative reward and the result of an algorithm as (optimal) policy has been discussed. Finally, value functions have been largely presented since they are the starting point of many algorithms. It remains to introduce algorithms designed to solve these problems, which are shown in next chapter.

Chapter 3

Reinforcement Learning

Algorithms designed for solving MDPs can be divided into two main categories. Algorithms for **prediction** take as input the MDP $< S, A, P, R, \gamma, \mu >$ and a policy π and their output is an estimate of the value function V^{π}. Basically, prediction algorithms are designed with the idea of evaluating a given policy. On the other hand algorithms designed for **control** take as input the MDP $< S, A, P, R, \gamma, \mu >$ and they produce as output an approximation of the optimal value function V^* and the optimal policy π^*. The purpose of control algorithms is therefore to "solve" the MDP.

The most intuitive idea to find the best policy is a **brute force** approach. Indeed it is sufficient to enumerate all deterministic policies (knowing that there always exists an optimal deterministic policy from Theorem 2.2.3), then it is possible to use Bellman equations to evaluate each policy and the best one is the optimal deterministic policy. The problem of this naive approach is that the number of policies is exponential ($\#Policies = |A|^{|S|}$), so it becomes impractical for non-trivial problems and more complex algorithms are necessary.

In this chapter many different algorithms for prediction and control are introduced. In Section 3.1 the first solving algorithms based on Dynamic Programming are shown. They are useful but not much applicable in real problems, since they assume full knowledge of the model, so in Sections 3.2 and 3.3 Reinforcement Learning algorithms are explained: they address prediction and control problems without assuming the full knowledge of the MDP.

3.1 Dynamic Programming

The algorithms presented in this section are generally called **Dynamic Programming (DP)**. The term indicates a group of algorithms that exploit the sequential nature of the problem (*Dynamic*) trying to optimize a program, in this case a policy (*Programming*), by breaking it down into a collection of simpler subproblems. It is a very general approach to solve problems that can be applied if they have two main properties:

- problems must have an optimal substructure, so the optimal solution can be decomposed into subproblems (MDPs satisfy this property since Bellman equations give recursive decomposition);
- problems must have overlapping subproblems, so that they recur many times and it is possible to store and reuse the same solutions (value functons store and reuses solutions).

The main drawback of this approach, which makes DP algorithms impossible to be applied in many cases, is that it requires full knowledge of the MDP, a strong hypothesis since knowing all the transition probabilities and the reward function is difficult in MDPs modeling real problems and sometimes it is impossible, because the number of states and/or actions may be infinite or the dynamics of the process may be unknown. Sometimes it is even impossible to store them in memory due to the magnitude of the number of states. However, the importance of DP approach is that it is a starting point for many RL algorithms.
As discussed at the end of Chapter 2, value functions are the unique fixed point of their related Bellman operators, moreover it is easy to extract an optimal policy from an optimal value function as shown in Equation 2.23, therefore the focus of these algorithms for prediction and control is on estimating and improving value functions.

3.1.1 Prediction: Policy Evaluation

For a given policy π, the **Policy Evaluation** algorithm computes an approximation of the state-value function V^π, that is an evaluation of the policy as needed. Recalling that Bellman equations for the state-value function shown in Equation 2.15 are a system of $|S|$ linear equations with $|S|$ variables $V^\pi(s)$, it is enough to solve the system (with complexity $O(|S|^3)$) to find the state-value function. The solution can be expressed in matrix notation:

$$V^\pi = (I - \gamma P^\pi)^{-1} R^\pi. \tag{3.1}$$

To avoid the computations needed to solve a system of linear equations, that are complex with a huge number of states, **Iterative Policy Evaluation** algorithm is proposed. This algorithm iteratively applies the contraction property of Bellman operator in order to

find its fixed point V^π. Recalling Equation 2.28, starting with arbitrary values assigned to $V(s)$, it is possible to update the value of each state toward the correct $V^\pi(s)$ by applying Bellman operator:

$$V_{k+1}(s) \leftarrow \sum_{a \in A} \pi(a|s) \left[R(s,a) + \gamma \sum_{s' \in S} P(s'|s,a) V_k(s') \right]. \tag{3.2}$$

Basically, the update exploits the fixed-point theorem to produce, starting from arbitrary values, a sequence that converges to the unique fixed point V^π of Bellman operator. Schematically:

$$V_0 \to T^\pi V_0 = V_1 \to (T^\pi)^2 V_0 = V_2 \to ... \to (T^\pi)^k V_0 = V_k \to ... \to \lim_{k \to \infty} (T^\pi)^k V_0 = V^\pi. \tag{3.3}$$

This sequence is ensured to be convergent to V^π, so the algorithm formally converges at the limit. In practice it is typically stopped when the quantity $\max_{s \in S} |V_{k+1}(s) - V_k(s)|$ is sufficiently small, meaning that two consecutive updates are sufficiently similar to say that the approximate result is near to the fixed point V^π.

3.1.2 Control: Policy Iteration and Value Iteration

The objective of control is to find an optimal policy. Considering a deterministic policy π the main step to improve it toward an optimal policy π^* with Dynamic Programming is called **policy improvement**. In a state s it is possible to improve the current policy by choosing an action $a \neq \pi(s)$ that is greedily better, namely it is the action that maximizes the action-value function in that state:

$$\pi'(s) = \arg\max_{a \in A} Q^\pi(s,a). \tag{3.4}$$

Applying this greedy update, the value of each state improves, since

$$Q^\pi(s, \pi'(s)) = \max_{a \in A} Q^\pi(s,a) \geq Q^\pi(s, \pi(s)) = V^\pi(s), \tag{3.5}$$

and the following theorem holds:

Theorem 3.1.1. *Let π, π' be two deterministic policies such that*

$$Q^\pi(s, \pi'(s)) \geq V^\pi(s), \ \forall s \in S, \tag{3.6}$$

then the policy π' is not worse than π:

$$V^{\pi'}(s) \geq V^\pi(s), \ \forall s \in S. \tag{3.7}$$

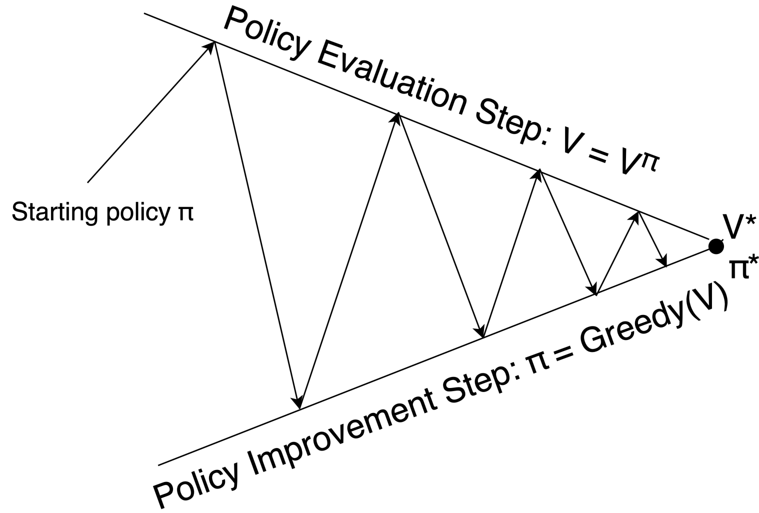


Figure 3.1: Scheme of Policy Iteration algorithm: it is a sequence of a policy evaluation and a greedy policy improvement steps until the optimum is reached.

The theorem ensures that the greedily updated policy can not be worse than the current one, hence it can be an improvement or have the same value ($V^{\pi} = V^{\pi'}$), that means

$$Q^{\pi}(s, \pi'(s)) = \max_{a \in A} Q^{\pi}(s, a) = Q^{\pi}(s, \pi(s)) = V^{\pi}(s), \tag{3.8}$$

which is exactly the Bellman optimality equation, implying that an optimal policy has been found:

$$V^{\pi}(s) = V^{\pi'}(s) = V^{*}(s) \; \forall s \in S. \tag{3.9}$$

This kind of reasoning produces the following sequence:

$$\pi_0 \rightarrow V^{\pi_0} \rightarrow \pi_1 \rightarrow V^{\pi_1} \rightarrow ... \rightarrow \pi^* \rightarrow V^* \rightarrow \pi^*.$$

Summarizing, it is possible to design an algorithm following these theoretical results where each iteration is made of two steps as shown in Figure 3.1: the first is the *evaluation* of the current policy that can be done performing a prediction algorithm (for example Policy Evaluation algorithm presented in Subsection 3.1.1); the second is the *improvement* of the policy (for example the greedy policy improvement), generating $\pi' \geq \pi$. This prediction algorithm is called **Policy Iteration**.

The main drawback of Policy Iteration algorithm is that at each iteration there is a policy evaluation step that is ensured to converge only at the limit, so it may be very slow to approach the correct value function. This is the reason why **Value Iteration** algorithm has been introduced. The idea of Value Iteration is to exploit the properties

of the optimal Bellman operator in order to update the value function. Indeed, as shown in Equation 2.29, applying many times the optimal Bellman operator to any vector in $\mathbb{R}^{|S|}$ updates it toward the optimal value function and, at the limit, it will converge to the unique fixed point that is exactly the optimal value-function $V^*(s)$.
The update rule of the algorithm is therefore:

$$V_{k+1}(s) = \max_{a \in A} \left(R(s,a) + \gamma \sum_{s' \in S} P(s'|s,a) V_k(s') \right) = T^* V_k(s), \tag{3.10}$$

and the scheme of Value Iteration is:

$$V_0 \rightarrow V_1 \rightarrow V_2 \rightarrow ... \rightarrow V^*. \tag{3.11}$$

In contrast to Policy Iteration, there is no explicit policy during the updates and intermediate value functions may be vectors of $\mathbb{R}^{|S|}$ not corresponding to any policy of the MDP. They are simply recursive applications of the optimal Bellman operators that ensure the convergence to the optimal value functions (their fixed points).

In conclusion, all the algorithms presented in this section are based on *state-value* functions and they are exponentially faster than the brute force approach. On the other hand, their complexity is polynomial in the number of states, so in practice they can be applied to problems with a few millions of states at most.
Exploiting Bellman equations it is also possible to apply Linear Programming (LP) rather than Dynamic Programming [14]. The Linear Programming approach consists in an optimization problem whose constraints are $V \geq T^*(V)$ in order to update the value function toward the optimal. It is also possible to exploit Dual Linear Programming for an explicit interpretation of the policy updates. In this thesis LP methods are not explained in detail because they become impractical much sooner than DP methods.

3.2 Value-Based Reinforcement Learning

As explained in previous section, Dynamic Programming algorithms designed to solve MDPs modeling complex real world problems are not applicable in practice because they require the full knowledge of the MDP. Reinforcement Learning (RL) algorithms address the problems of *prediction* and *control* of an **unknown** MDP. They learn directly from episodes of experience, estimating the value function or the policy through the optimization of an approximation of the expected return. RL algorithms try to find the optimal policy sampling episodes made of sequences action-state-reward. Since

they exploit these sequences with many different approaches, it is useful to classify RL algorithms in macro-categories that summarize their main characteristics.

- **Model-Free vs. Model-Based**: Model-Based algorithms are related to DP, indeed they use an initial number of iterations to estimate the parameters of the MDP, then they perform prediction or control on the estimated MDP using DP. Model-Free algorithms use directly the samples to approximate and update value functions and policies. Since the estimate of the MDP could be inaccurate and lead to learn something different from the real problem, most of RL algorithms are *Model-Free*.

- **Value-Based vs. Policy-Based vs. Actor-Critic**: The other main difference between RL algorithms is about the entity the algorithms try to optimize to learn. Value-Based algorithms use samples to estimate the (optimal) value functions, from which is easy, at the end, to extract a deterministic policy as done in Equation 2.23. Policy-Based algorithms directly learn the policy, without storing information about value functions. Finally, Actor-Critic methods use both value functions and the explicit policy in the learning process.

Moreover, RL algorithms present other peculiar characteristics.

- **On-Policy vs. Off-Policy**: an on-policy algorithm uses the same policy to learn the optimal one and to generate episodes, while off-policy methods use a different policy, called *behavioral*, to generate the episodes.

- **Online vs. Offline**: an online algorithm updates its parameters (value functions and/or policy) during the generation of data, while offline algorithms need a full static dataset to perform the learning.

- **Episodic vs Non-Episodic**: some algorithms need a full episode to perform an update, while others are able to learn from incomplete episodes (and they are the only ones applicable to MDPs where episodes can be infinite).

In this section many *Value-Based* algorithms with different characteristics are explained. In particular, Subsection 3.2.1 focuses on prediction with RL, Subsections 3.2.2 and 3.2.3 present algorithms for control respectively on-policy and off-policy. Section 3.3, on the other hand, will focus on *Policy-Based* algorithms.

3.2.1 Model-free Prediction

The two methods presented in this section are Value-Based methods designed to evaluate a given policy π.

Monte-Carlo Policy Evaluation (MC)
Monte-Carlo Policy Evaluation uses the empirical mean return to estimate the expected return, that is exactly the Definition 2.2.6 of value function. It is a model-free, online episodic algorithm. It has two variants: **First Visit**, which for every episode evaluates the return of a state as the return from the first time state s is visited to the end of the episode; **Every Visit**, which computes a different return every time state s is visited.
Basically at iteration k, after sampling a full episode $(s_1, a_1, r_2, ..., s_T)$ following the policy π, the algorithm computes the returns using First Visit or Every Visit variants. Then, each new sample v_t of the return related to each state s_t visited during the episode is added to its value applying an incremental update of the average:

$$V(s_t) \leftarrow V(s_t) + \frac{1}{N(s_t)}(v_t - V(s_t)), \tag{3.12}$$

where $N(s_t)$ is the number of returns available for state s_t.
Once many episodes have been performed, the average of the returns related to any state s_t is an estimate of $V^\pi(s_t)$.
First Visit MC is an unbiased estimator of the expected return (basically the value function), so the law of large numbers ensures that it converges to the exact expected return when the number of episodes is huge. On the other hand, it needs many episodes to compute a consistent estimate of the value function, since it produces only one return for each state visited in an episode. Every Visit MC is instead a biased estimator, but it needs less iterations to produce a reliable estimate of the value function. The main drawback of Monte-Carlo approach is that it is episodic so it needs a full episode to perform an update, therefore it may be slow if the MDP produces long episodes and it is even not applicable when the MDP produces infinite episodes.

Temporal Difference Policy Evaluation (TD)
This is an online Model-Free approach that is able to learn from incomplete episodes, speeding up the updates of the value function with respect to MC approach and allowing to use the method also for infinite episodes.
Starting from the MC update rule, TD substitutes $N(s_t)$ with a generic parameter α and the sample of the return v_t with an estimate $r_{t+1} + \gamma V(s_{t+1})$, that is the immediate reward plus the discounted value of next state. In this way the update rule becomes:

$$V(s_t) \leftarrow V(s_t) + \alpha(r_{t+1} + \gamma V(s_{t+1}) - V(s_t)). \tag{3.13}$$

With the approximation introduced on the sample return, it can be immediately computed at time $t + 1$ of an episode, without waiting until it ends.
TD is faster than MC, it produces a biased estimator of $V^\pi(s_t)$ but it has much lower

variance, since the complete return depends on many random action-state-reward tuples produced during the episode, while TD update depends only on one random action, the following random state and the random immediate reward. It can be proved that MC algorithm converges to the minimum mean-squared error solution, while TD converges to the solution producing the maximum likelihood MDP estimation.

Variants

It is possible to introduce some variants to TD that consider a different approximation of the sample return.

- **TD(n)** estimates the return as the discounted sum of next n rewards plus the estimated value of the n-th state:

  $$v_t^{(n)} = r_{t+1} + \gamma r_{t+2} + ... + \gamma^{n-1} r_{t+n} + \gamma^n V(s_{t+n}). \tag{3.14}$$

  Then the update is, as usual in TD:

  $$V(s_t) \leftarrow V(s_t) + \alpha(v_t^{(n)} - V(s_t)). \tag{3.15}$$

  This approach allows to compute a less biased return for an increasing n, but it needs n following steps to be computed, which increases the variance.

- **Average TD(n)** consists in approximating the return with different n-step returns $(v_t^{(n_1)}, v_t^{(n_2)}, ...)$ averaging them. For example,

  $$v_t^{(n)} = \frac{1}{2} v_t^{(n_1)} + \frac{1}{2} v_t^{(n_2)}. \tag{3.16}$$

- **TD(λ)** computes an approximation of the sample return v_t^λ combining all n-step returns $v_t^{(n)}$ using weights $\lambda^{n-1}(1-\lambda)$. The estimated return becomes:

  $$v_t^\lambda = (1-\lambda) \sum_{n=1}^{\infty} \lambda^{n-1} v_t^{(n)}, \tag{3.17}$$

  with usual update rule for the state-value function:

  $$V(s_t) \leftarrow V(s_t) + \alpha \left(v_t^\lambda - V(s_t) \right). \tag{3.18}$$

  With this kind of estimate, weights of returns are decreased polynomially: immediate reward has weight $1-\lambda$, the next one $(1-\lambda)\lambda$, the third $(1-\lambda)\lambda^2$, and so on until the end of the episode. In this **forward** view of the TD(λ), the advantage of the immediate possibility to update the value function of current state is lost, since the update needs all the rewards until the end of the episode. This is the reason why TD(λ) algorithm has a **backward** view formulation, which is the algorithm

applied in practice. This kind of update takes into account the number of times a state has been visited in the past iterations, with an exponential decreasing weight depending on how long ago the state was visited last. The value that stores this information is called **eligibility trace**, and it is:

$$e_t(s) = \begin{cases} \gamma\lambda e_{t-1}(s) & if\ s \neq s_t \\ 1 & s = s_t \end{cases}. \tag{3.19}$$

The resulting estimated returns are used in the usual update rule:

$$V(s_t) \leftarrow V(s_t) + \alpha e_t(s_t)\left(r_{t+1} + \gamma V(s_{t+1}) - V(s_t)\right). \tag{3.20}$$

In conclusion, algorithms for prediction are building blocks for control, since they are able to evaluate a policy. TD(λ) is the summary of all algorithms presented, since with $\lambda = 0$ it becomes equivalent to TD(0) while with $\lambda = 1$ it is equal to Monte-Carlo value function estimate. All these algorithms are based on the idea of updating the state-value function in order to produce a more and more accurate estimate of the value of a given policy π in any state $V^\pi(s)$. These algorithms can be similarly applied to estimate the action-value function $Q^\pi(s,a)$.

3.2.2 On-Policy Model-Free Control

In this section are described some Reinforcement Learning algorithms aimed to find the optimal policy π^*. In particular, they are Model-Free algorithms and they take inspiration on DP combining it with algorithms for prediction. They are all *on-policy* algorithms, so they learn and update a policy π directly sampling episodes following policy π itself.

On-Policy Monte-Carlo Learning

Policy Iteration algorithm, explained in Section 3.1.2, is made of a policy evaluation step of current policy (estimating V^π) and of a policy improvement step (greedy).

This kind of reasoning can be repeated in this framework by computing the action-value function following Monte-Carlo Policy Evaluation explained in Section 3.2.1 for the policy evaluation step. It is necessary to estimate the action-value function because it is not enough to use the state-value function for a greedy improvement of the policy without knowing the MDP. In fact the greedy rule applied in policy improvement step is:

$$\pi'(s) = \arg\max_{a \in A} Q(s,a). \tag{3.21}$$

Iterating the two steps until two consecutive updates are sufficiently similar, thanks to the uniqueness of fixed points of Bellman optimal operators, it is possible to assert

that the results found are good approximations of $Q^*(s, a)$ and π^*.
A more efficient update rule to ensure continual exploration is to try all actions in every state with non-zero probability, selecting with probability $1 - \epsilon$ the best greedy action and with probability ϵ a random action:

$$\pi(s, a) = \begin{cases} \frac{\epsilon}{m} + 1 - \epsilon & if\ a = \arg\max_{a \in A} Q(s, a) \\ \frac{\epsilon}{m} & otherwise \end{cases} . \tag{3.22}$$

It is possible to prove that this **ϵ-greedy** policy update is better or equal than the previous one, so that the policy improvement step following this rule is consistent and it is preferable to the deterministic greedy update, which explores too few actions by always selecting only one of them.

On-Policy Temporal-Difference Learning
TD for prediction has many advantages with respect to Monte-Carlo, being online, with less variance and applicable to infinite sequences. Therefore the natural idea that follows also for control is to use TD instead of MC in the policy evaluation step of the algorithm. This approach is called **SARSA** algorithm, its policy evaluation step is done estimating the action-value function $Q(s, a)$ with the TD algorithm for prediction, that has update rule:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left(r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \right) . \tag{3.23}$$

Then the policy improvement step can be again the ϵ-greedy policy improvement shown in MC control. By properly choosing the learning rate α, SARSA algorithm can be shown to be convergent to the optimal $Q^*(s, a)$.

Variant
As done for prediction, it is possible to update the action-value function using v_t^λ in a **forward** view or to apply eligibility traces to the past in a **backward** view. The algorithm that applies one of the two options (usually backward is performed since it can be updated immediately) is called **SARSA**(λ).

3.2.3 Off-Policy Model-Free Control

The following algorithms are again *model-free* and designed in order to estimate the optimal policy. They are *off-policy*, so they learn and update a **target policy** π using episodes sampled following a different **behavior policy** $\bar{\pi}$. This approach is inspired by how animals learn: they keep in mind the experience generated from old policies or observing actions taken by other animals. This is very different compared to on-policy approaches discussed before, where the policy is updated and the generated experience is completely discarded when starting a new policy evaluation step.

Importance Sampling for Off-Policy Monte-Carlo and SARSA

Importance Sampling is a general statistical technique for estimating properties of a random variable following a certain distribution $x \sim P$ having only samples from another random variable following a different distribution $x \sim Q$.

The rule for estimating the expected value is:

$$\mathbb{E}_{x \sim P}[f(x)] = \mathbb{E}_{x \sim Q}\left[\frac{P(x)}{Q(x)} f(x)\right]. \tag{3.24}$$

Importance Sampling can be applied to **Off-Policy Monte-Carlo** to evaluate current policy π, producing sample returns following policy $\bar{\pi}$. In order to produce an estimate of the return for the policy π to be evaluated, Importance Sampling correction must be considered:

$$v_t^{\mu} = \frac{\pi(a_T|s_T)}{\bar{\pi}(a_T|s_T)} \frac{\pi(a_{T-1}|s_{T-1})}{\bar{\pi}(a_{T-1}|s_{T-1})} ... \frac{\pi(a_t|s_t)}{\bar{\pi}(a_t|s_t)} v_t, \tag{3.25}$$

where v_t is the return computed following policy $\bar{\pi}$. Then the usual action-value function update can be performed:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(v_t^{\mu} - Q(s_t, a_t)). \tag{3.26}$$

Finally, the policy improvement step can be computed with the usual ϵ-greedy update.

It is also possible to apply Importance Sampling to build an **Off-Policy SARSA** algorithm, using the TD update of action-value function with Importance Sampling correction:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left(r_{t+1} + \gamma \frac{\pi(a_t|s_t)}{\bar{\pi}(a_t|s_t)} Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \right). \tag{3.27}$$

MC Off-Policy algorithm may increase the variance by iterating the estimate of rewards from t to the end of the episode. Moreover it can not be performed if $\bar{\pi}$ is zero where π is not zero. Therefore, to avoid this situation, the target policy is assumed by the algorithm to be absolutely continuous with respect to the behavioral. The off-policy version of SARSA has much lower variance since it uses only one Importance Sampling correction and policies only need to be similar over one step. Both approaches have the advantage, with respect to their on-policy version, to use the same episodes in every iteration for estimating the value function, since they are generated by a different fixed policy. This is computationally more efficient and similar to biological process of learning, since humans does not learn from zero at any moment but they use what happened before in their experience to infer how to behave in a new situation.

Q-Learning
All the algorithms for RL control explained so far take inspiration from Policy Iteration approach in Dynamic Programming. In Section 3.1.2 has been presented also the Value Iteration algorithm for DP control. Hence, Q-learning is an off-policy RL algorithm that aims to learn the optimal policy from experience sampled by a *behavior* policy $\bar{\pi}$, estimating the action-value function $Q^*(s,a)$ in the same fashion of Value-Iteration algorithm. Indeed, at each iteration, it updates the value of the action-value function in a state s_t performing action a_t applying Bellman optimal operator. The resulting update rule is:

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha \left(r_{t+1} + \gamma \max_{a' \in A} Q(s_{t+1},a') - Q(s_t,a_t) \right). \tag{3.28}$$

In this way, the behavioral policy can be random, or ϵ-greedy, since the update is not performed by looking at the action that this policy will perform in next state. This is the main difference between SARSA and Q-learning: the latter may be seen as an Off-Policy version of SARSA that uses the behavioral policy to generate samples but it updates the value functions independently from the policy. The algorithm does not have a target policy improvement step but it only updates the action value function. This is not a problem, since at the end of the algorithm it is straightforward to extract a deterministic policy from the action-value function by picking the best action in each state as done in Equation 2.23.

The importance of Dynamic Programming is now clear: by taking the idea from Monte-Carlo algorithms (that in practice are difficult to apply) of generating episodes to overcome the fact that the MDP is unknown, TD learning applies the same update scheme of Iterative Policy Evaluation, while SARSA applies an update similar to Policy Iteration and Q-learning applies an update in Value Iteration fashion.
In next subsection a particularly important algorithm is discussed. It is one of the three algorithms applied to the environments designed in this thesis. It is presented now since it is an off-policy algorithm as the ones of this section and it is based on an idea similar to Q-Learning.

3.2.4 Fitted Q-Iteration (FQI)

FQI [16] is a model-free, off-policy and offline algorithm. It is designed in order to learn a good approximation of the optimal action-value function $Q^*(s,a)$ by exploiting Value Iteration idea as Q-learning does. The innovative approach of FQI consists in the application of Supervised Learning techniques in doing this.
Since the algorithm is offline, it considers a full dataset F containing the information get from experience. In particular, each row of the dataset represents an interaction with

the environment, and it consists in a 4-tuple containing the current state, the action performed, the immediate reward and the next state:

$$F = \left\{ (s_t^i, a_t^i, r_{t+1}^i, s_{t+1}^i) \mid i = 1, 2, ..., \#F \right\}. \tag{3.29}$$

In FQI the agent is not directly interacting with the environment generating samples and updating a policy or a value function that can be improved toward the optimal, but the starting point is the dataset of 4-tuples F that represents all the experience the agent has collected that is exploited to infer an estimate of the optimal action-value function $Q^*(s, a)$. In complex real problems the possible combinations of actions and states is huge (or infinite in a continuous problem), hence the dataset does not contain all possibilities. To overcome this issue FQI applies a Supervised Learning regression algorithm trained on the 4-tuples dataset with the action-value function as the target function that must be learned. Once the regressor is trained, it can predict the value of any state-action pair, so that it fully estimates the action-value function needed.
Specifically, at each iteration of the algorithm, the horizon in which the optimization of the action-value function is performed increases of a step. At first iteration it estimates $Q_1^*(s, a)$, the action-value function that is optimal only with respect to the next step, i.e. the immediate reward. An approximation of this function can be done by applying the regressor to a training set with as input the current state and the action performed (s_t, a_t) and as target the immediate reward (r_{t+1}). In a general N-th iteration, the estimated function will be $Q_N^*(s, a)$, the optimal action-value function with respect to N steps (that, with sufficiently large N, will be a good estimate of $Q^*(s, a)$). At this step the training set will be once again the couple (s_t, a_t) of current state and action, while the target will be iteratively computed as the immediate reward plus the discounted best value over $N - 1$ steps of next state. Indeed, the first step is univocally determined by the action while the following $N - 1$ steps have optimal value given by $Q_{N-1}^*(s_{t+1}, a_{t+1})$, that is the function that was approximated in previous iteration. The FQI procedure can be summarized as shown in Algorithm 1.

Among all the value-based algorithms presented in this chapter, FQI will be applied to the environments designed in this thesis since the dataset F needed by the algorithm as input will be computed and explored in the Feature Selection presented in Chapter 6. Moreover, FQI estimates the action-value function of unseen state-action couples, which is crucial in the MDPs presented in this thesis, since the state will be made of Stock Market values and sentiment signals, that are continuous variables, so all states can not be explored directly, a required condition for convergence for SARSA or Q-Learning algorithms.

Algorithm 1: Fitted Q-Iteration

Inputs: set of 4-tuples $F = \{s_t^i, a_t^i, r_{t+1}^i, s_{t+1}^i\}_{i=1:\#F}$ and a regression algorithm.

initialization:

$N \leftarrow 0$

$Q_N(s,a) \leftarrow 0 \ \forall s \in S, \forall a \in A$

Iterations:

while *stopping condition not reached* **do**

$N \leftarrow N + 1$

Build the training set $TS = \{(i^k, o^k)\}_{k=1:\#F}$ such that:

$i^k = (s_t^k, a_t^k)$,

$o^k = r_{t+1}^k + \gamma \max_{a \in A} Q_{N-1}(s_{t+1}^k, a)$.

Use the regression algorithm trained on TS to learn the function $Q_N(s,a)$.

end

3.3 Policy Search Reinforcement Learning

All the algorithms considered so far in Reinforcement Learning are designed to learn the value function of a given policy for prediction or to improve it in control. These methods are called **Value Search (or Value Based)** methods. In this section, instead, methods learn directly the policy, without the use of value functions, but improving directly the parametrized policy with respect to a certain loss function. These methods are called **Policy Search (or Policy Based)** methods.
In particular, the policy depends on a parameter vector $\theta \in \mathbb{R}^{d'}$, so $\pi(a|s,\theta)$ is the probability to perform action a in state s with parameter θ. The policy in general can be parametrized in any way, so it is possible to assume the policy to be any function that, given state, action and parameters, assigns the probability of choosing that action. In particular, in a discrete MDP, the natural idea is to give a preference expressed by a preference function $h(s,a,\theta) \in \mathbb{R}$ to every state-action pair, and to apply the softmax distribution:

$$\pi(a|s,\theta) = \frac{\exp(h(s,a,\theta))}{\sum_{a' \in A} \exp(h(s,a',\theta))}. \tag{3.30}$$

In this way the result is the probability distribution of performing action a in state s that respects the preferences expressed by the preference function. The preferences may be parametrized as any function h: a common choice is a Deep Neural Network (discussed in Section 4.2) where the parameter vector θ are the weights of the network, but it can also be a simple linear function.

The main advantage of this approach is that the optimization performed by the algorithm can be done directly to the policy parameters. This allows to improve the policy directly, without using value functions and greedy policies, that lead to an indirect optimization of the policy through value functions producing more variance. Intuitively, an update in Policy Search aims to maximize the performance of the policy. This can be done using the gradient ascent with respect to a certain performance measure $J(\theta)$ to update the parameters toward an optimal policy:

$$\theta_{t+1} \leftarrow \theta_t + \alpha \widehat{\nabla J}(\theta_t), \tag{3.31}$$

where $\widehat{\nabla J}(\theta_t)$ is a stochastic estimator of the gradient of the performance measure with respect to θ_t, which is introduced because it is usually impossible to compute exactly the gradient.
Policy Search methods that perform this kind of update are called **Policy Gradient Methods**, since they focus on the gradient ascent for updating the policy.
Another advantage of Policy Search is that, updating the parameters of the (differentiable) parametrized policy, it makes the changes on action probabilities a smooth function. This is not guaranteed in value search algorithms with ϵ-greedy policy, where the action probabilities may change a lot for an arbitrary small difference of the action-value function estimate. Indeed, when the best action has a value slightly better than another, a small variation of the action-value function may lead to change the best action. This is a concrete drawback of Value Based methods, since the action-value function is always an approximation of the real one, so this situation can happen.
In next Subsection 3.3.1 Policy Gradient approach is described in more detail. Then in Subsections 3.3.2 and 3.3.3 the focus is on two important Policy Search methods, that are widely applied this thesis.

3.3.1 Policy Gradient Methods

The main step in Policy Gradient Methods is the update of the parameters, already shown in Equation 3.31. The core of the update rule is to compute the gradient of a performance measure $J(\theta)$, since the aim of the methods is to find the parameter vector that maximizes it. This can be done exploiting a strong result that is the starting point of many Policy Gradient Methods. First of all, a natural measure of performance of a policy is the expected return following the parametrized policy, that is exactly the Definition 2.2.6 of state-value function. So it is logical to consider:

$$J(\theta) = V_{\pi_\theta}(s_0), \tag{3.32}$$

assuming s_0 to be the common initial state of every episode. It is now possible to enunciate the **Policy Gradient Theorem**.

Theorem 3.3.1 (Policy Gradient Theorem)**.** *Defining the performance measure $J(\theta)$ as in Equation 3.32, its gradient is directly proportional to the following quantity:*

$$\nabla J(\theta) \propto \mathbb{E}_{\pi_\theta} \left[\sum_{a \in A} Q_{\pi_\theta}(S_t, a) \nabla_\theta \pi_\theta(a|S_t, \theta) \right], \tag{3.33}$$

considering S_t the random variable expressing the current state.

The Policy Gradient Theorem provides an analytical expression of the gradient of performance needed in the gradient ascent update. Policy Gradient algorithms provide ways to estimate this expression (an expected value) shown in Equation 3.33 with information from experience (coming from sampling episodes).

REINFORCE

Recalling that the first approach used to estimate the return in ML has been Monte-Carlo, the first Policy Gradient algorithm introduced in this section is REINFORCE [44], that is **Monte-Carlo Policy Gradient**. Starting from Policy Gradient Theorem 3.33, it is possible to derive:

$$\nabla J(\theta) \propto \mathbb{E}_{\pi_\theta} \left[Q_{\pi_\theta}(S_t, A_t) \frac{\nabla_\theta \pi(A_t|S_t, \theta)}{\pi(A_t|S_t, \theta)} \right] = \mathbb{E}_{\pi_\theta} \left[G_t \frac{\nabla_\theta \pi(A_t|S_t, \theta)}{\pi(A_t|S_t, \theta)} \right], \tag{3.34}$$

where A_t and S_t are random variables representing the state-action pair and G_t is the random variable of the return from them. In REINFORCE algorithm the expression inside the second expected value of Equation 3.34 can be computed in an episode and it is considered a sample estimate of the expected value itself. In practice an iteration of the algorithm consists in generating an episode following the current policy $\pi(a|s, \theta)$, then for each step t of the episode it updates the policy parameters from the rule derived above:

$$\theta \leftarrow \theta + \alpha \gamma^t v_t \frac{\nabla_\theta \pi(a_t|s_t, \theta)}{\pi(a_t|s_t, \theta)}, \tag{3.35}$$

where v_t is the return of the episode from step t to the end as introduced in Definition 2.2.4.

This algorithm is Monte-Carlo because it needs the return from the current step to the end of the episode to perform the updates, so it can only update parameters after the exploration of a complete episode. This means that REINFORCE presents all the issues of Monte-Carlo approaches: it is unbiased but it has a huge variance and it may learn slowly; it can only be performed on episodic MDPs (with a sufficiently small number of steps in the episodes to be efficient); it needs to wait until the end of the episode to update the policy. These are the reasons why some improvements to REINFORCE can be useful.

Variants

- *REINFORCE with baseline* [33]: it is possible to subtract a baseline term to the sample return in the update rule. This variation gives more importance to returns that are greater than a baseline value $b(s_t)$ assigned to a state s_t. In this way it is not considered just the value of the return in an episode but it is scaled vith respect to the baseline. The update rule at step t becomes:

$$\theta \leftarrow \theta + \alpha\gamma^t(v_t - b(s_t))\frac{\nabla_\theta \pi(a_t|s_t,\theta)}{\pi(a_t|s_t,\theta)}. \tag{3.36}$$

  A natural choice for the baseline may be an estimate of the state-value function of current policy $V^\pi(s_t)$, that can be computed with one of the Value Search methods discussed in Subsection 3.2.1. This procedure is logical but not the most efficient in order to minimize the variance, therefore more complex and efficient baselines are introduced in [33].

- *Actor-Critic Policy Gradient* [13]: although REINFORCE with baseline computes a value function, it is considered a Policy Search algorithm since the value function is not optimized but it is only used as a fixed parameter in the update rule. In this variant both the policy and the value function are actively used, so the method can be classifies as an Actor-Critic approach. The idea is to improve the performance of the algorithm by substituting the return v_t with an estimate of it as done with SARSA algorithm (3.23), so that the policy can be updated during the episodes. This estimate reduces the variance and makes the algorithm applicable to infinite or continuous MDPs. The update rule approximating the return at step t of the episode becomes:

$$\theta \leftarrow \theta + \alpha\left(r_{t+1} + \gamma V^{\pi_\theta}(s_{t+1}) - V^{\pi_\theta}(s_t)\right)\frac{\nabla_\theta \pi(a_t|s_t,\theta)}{\pi(a_t|s_t,\theta)}. \tag{3.37}$$

  As done with SARSA(λ), it is possible to use all the other approximations discussed for TD algorithm: n-step return (3.14), forward view of λ approximation (3.17) or backward view with eligibility traces (3.19).

Summing up, Policy Gradient methods are the most intuitive Policy Search group of methods that exploit the gradient of the parametrized policy and its analytical expression given by Policy Gradient Theorem, estimating the expected value appearing in the formula with the usual MC or TD approaches. This kind of algorithms are the starting point of more advanced Policy Search methods that have the advantage of optimize the policy directly, without passing through the optimization of the parameter vector θ as done with Policy Gradient approach.

3.3.2 Trust Region Policy Optimization (TRPO)

The algorithm explained in this section, called TRPO [39], updates the parameters of the policy optimizing the performance measure $J(\pi_\theta)$, meanwhile constraining the current policy and its update to be close.
To simplify the notation, it is useful to introduce the **advantage function**.

Definition 3.3.1. The advantage function $A^\pi(s, a)$ is the difference between the action-value function computed in (s, a) and the state-value function evaluated in s:

$$A^\pi(s, a) = Q^\pi(s, a) - V^\pi(s). \tag{3.38}$$

This function represents the advantage of performing action a in state s (and then follow policy π) with respect to the expected return of the state following policy π.
The starting point of the algorithm is inspired by a theoretical result (introduced in [25]).

Theorem 3.3.2. *Defining the performance measure $J(\pi)$ as in Equation 3.32, it is possible to express the expected return of another policy $\tilde{\pi}$ in terms of the advantage over current policy π:*

$$J(\tilde{\pi}) = J(\pi) + \mathbb{E}_{\tilde{\pi}}\left[\sum_{t=0}^{\infty} \gamma^t A_\pi(s_t, a_t)\right]. \tag{3.39}$$

Basically the theorem states that the performance of a new policy $\tilde{\pi}$ is equal to the performance of current policy π plus the expected value with respect to the new policy of the discounted advantage over π. It follows that, to have an improved policy, is enough to update it so that the discounted expected value of the advantage is positive.
In Equation 3.39, the function $J(\tilde{\pi})$ depends on an expected value with respect to the new policy $\tilde{\pi}$, that in practice is unknown. Therefore TRPO introduces a first order local approximation $L(\tilde{\pi})$ of $J(\tilde{\pi})$ as follows:

$$L_\pi(\tilde{\pi}) = J(\pi) + \sum_{s \in S} \rho_\pi(s) \sum_{a \in A} \tilde{\pi}(a|s) A_\pi(s, a), \tag{3.40}$$

where $\rho_\pi(s)$ is the discounted sum of probabilities of visiting state s in each time-step.
The problem of Equation 3.40 is that the function $L_\pi(\tilde{\pi})$ is an approximation of the expected return $J(\tilde{\pi})$, so there is no guarantee that a policy $\tilde{\pi}$ that improves it will also improve the real expected return. Anyway, this can be proved to be true at first order, so a sufficiently small update of the policy $\pi \to \tilde{\pi}$ that improves $L_\pi(\tilde{\pi})$ improves also $J(\tilde{\pi})$.
Hence the core idea of the algorithm is to update the policy π_{new} so that it is not too far from the previous one π_{old} and, meanwhile, it improves the value of $L_{\pi_{old}}(\pi_{new})$ as

much as possible.
The first way to perform this kind of update is to find the greedy policy that maximizes the approximated performance measure:

$$\pi_{best} = \arg\max_{\pi_{best}} L_{\pi_{old}}(\pi_{best}), \tag{3.41}$$

and then to update the policy ensuring that it is does not changes a lot:

$$\pi_{new}(a|s) = (1-\alpha)\pi_{old}(a|s) + \alpha\pi_{best}(a|s), \tag{3.42}$$

with $\alpha \in (0,1)$ and remembering that π_{new} is the updated policy and π_{old} was the previous one. This procedure guarantees the new real expected reward $J(\pi_{new})$ to be better than the previous one minus a small constant (if α is set equal to a small value). This means that in most cases the new policy improves the real expected return, while in the worst case scenario the new policy slightly worsens the previous one.
The drawback of this update is that the new policy can only be of the form described in Equation 3.42, limiting the policy in a restricted class, while it is preferable by an update rule to be applicable to any policy. A better solution is therefore to constrain two consecutive policies to be close enough with respect to a measure of distance between them. The distance chosen in TRPO is the total variation divergence:

$$D_{TV}^{MAX}(\pi, \tilde{\pi}) = \max_{s \in S}\left(\sum_{a \in A} |\pi(a|s) - \tilde{\pi}(a|s)|\right). \tag{3.43}$$

Intuitively, this is the largest possible difference between the probabilities that two policies assign to the actions in the same state. Keeping the distance between two consecutive policies close enough $D_{TV}^{MAX}(\pi_{old}, \pi_{new}) \leq \alpha$, it is possible to prove that the same bound of the update rule 3.42 holds.
It is now necessary to find a way to implement the optimization of the approximated performance as in Equation 3.41, keeping the total variation divergence small, under an arbitrary parametrization θ of the policy $\pi = \pi_\theta$. A way to efficiently design the algorithm is to transform it into a constraint optimization problem, approximating the maximum in the total variations divergence with the mean, in order to decrease the number of constraints. Therefore an update becomes:

$$\begin{aligned} \max_{\theta} \quad & L_{\pi_{\theta old}}(\pi_\theta) \\ \text{s.t.} \quad & D_{KL}^{mean}(\pi_{\theta old}, \pi_\theta) \leq \delta, \end{aligned} \tag{3.44}$$

with δ hyperparameter of the algorithm.
Finally, it is useful to equivalently reformulate this constrained optimization problem in terms of expected values, expanding the elements of the Problem 3.44:

$$
\begin{aligned}
\max_{\theta} \quad & \mathbb{E}_{s,a}\left[\frac{\pi_\theta(a|s)}{\pi_{\theta old}(a|s)} A^{\pi_{\theta old}}(s,a)\right] \\
\text{s.t.} \quad & \mathbb{E}_s\left[D_{KL}(\pi_{\theta old}(\cdot|s)||\pi_\theta(\cdot|s))\right] \leq \delta,
\end{aligned}
\tag{3.45}
$$

or:

$$
\begin{aligned}
\max_{\theta} \quad & \mathbb{E}_{t}\left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta old}(a_t|s_t)} A^{\pi_{\theta old}}(s_t,a_t)\right] \\
\text{s.t.} \quad & \mathbb{E}_t\left[D_{KL}(\pi_{\theta old}(\cdot|s_t)||\pi_\theta(\cdot|s_t))\right] \leq \delta.
\end{aligned}
\tag{3.46}
$$

In the first formulation (3.45) the expected value is computed with respect to the probability of being in a state s and performing action a in it, while in the second (3.46) the expectation is equivalently computed with respect to the time. It is possible to estimate the two constrained optimization problems through experience: after sampling a trajectory of the MDP following the current policy $\pi_{\theta old}$, the action-value function and the expected values needed are estimated from samples, making the algorithm applicable in real problems (for example with Monte-Carlo simulations).
It is possible to summarize the procedure as shown in Algorithm 2.

Algorithm 2: Trust Region Policy Optimization

Inputs: Number T of samples to collect in an iteration
initialization:
$\pi_{\theta old}(s,a)$ as any policy
Iterations:
while *stopping condition not reached* **do**
  Simulate the policy $\pi_{\theta old}$ for T steps generating a trajectory:
    $C = \{s_0, a_0, s_1, a_1, ..., s_{T-1}, a_{T-1}, s_T\}$.
  For each state-action pair (s_t, a_t) in the set C estimate $A^{\pi_{\theta old}}(s_t, a_t)$
   computing $Q^{\pi_{\theta old}}(s_t, a_t)$ and $V^{\pi_{\theta old}}(s_t)$ as discounted sum of future reward
   along the trajectory.
  Averaging over samples, build the objective function and the constraint of
   the problem formulated in 3.46 (or 3.45).
  Approximately solve the problem with conjugate gradient algorithm to
   update the policy.
end

In conclusion, TRPO is a Policy Search RL algorithm that updates directly the policy. It is based on the maximization of an approximation of the expect return through a constrained optimization problem that forces two consecutive policies to be sufficiently similar. The strength of this algorithm with respect to Policy Gradient methods is that

the bound δ of Equation 3.44 is directly in the policy space and not in the space of the parameters of the policy as the hyperparameter α in Policy Gradient update (Equation 3.31). This is a powerful advantage not only because it is easier to tune the parameter and give it an interpretation, but mostly because it directly controls the update of the policy.

3.3.3 Proximal Policy Optimization (PPO)

Proximal Policy Optimization algorithm [38] is another Policy Search approach that focuses on simplifying the computations with respect to the nonlinear optimization problem of TRPO, keeping the core idea of proximality between the current policy and its update.
Starting from TRPO formulation 3.46, it is possible to apply the Lagrangian multipliers in order to move the constraints inside the optimization, transforming it into a Policy Gradient problem. However, in TRPO hard constraints were preferred, since it is difficult to choose Lagrangian multipliers that are well performing in different updates of the problem during the learning. The idea of PPO is to start from the maximization of the same approximated performance measure of TRPO:

$$L^{TRPO}(\pi_\theta) = \mathbb{E}_t \left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta old}(a_t|s_t)} A^{\pi_{\theta old}}(s_t, a_t) \right], \quad (3.47)$$

that without the constraint has the problem to lead to excessively large updates of the policy. To overcome this issue, the approximated performance measure is modified in a way that penalizes updates far from the current policy. In particular, calling:

$$r_t(\pi_\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta old}(a_t|s_t)}, \quad (3.48)$$

the proposed performance measure is:

$$L^{PPO}(\pi_\theta) = \mathbb{E}_t \left[\min \left(r_t(\pi_\theta) A^{\pi_{\theta old}}(s_t, a_t), clip \left(r_t(\pi_\theta), 1-\epsilon, 1+\epsilon \right) A^{\pi_{\theta old}}(s_t, a_t) \right) \right], \quad (3.49)$$

with ϵ hyperparameter usually set equal to 0.2.
The two terms inside the minimum are respectively the same term of TRPO and its "clipped" version that constraints the probability ratio between the policy and its update to be in the range $[1-\epsilon, 1+\epsilon]$, so that they are not much different. Taking the minimum, the measure is a pessimistic choice: it allows to select much different policies when their advantage is worse than the clipped version, while it denies huge update steps when their performance is better. In this way, the final performance measure is a lower bound of $L^{TRPO}(\pi_\theta)$, incorporating in a smart way the idea that in TRPO was expressed by the constraints.

Finally, PPO algorithm can be implemented as shown in Algorithm 2, substituting the problem formulation of Equation 3.46 with the objective function of Equation 3.49, so that at each iteration the algorithm produces T samples and optimizes them using the performance measure proposed by PPO.

Summing up, PPO is a Policy Search approach similar to TRPO, that faces the need of proximality between a policy and its update in a way more similar to Policy Gradient. It does not have constraints, but it optimizes directly the performance measure L^{PPO}, that is designed to update the policy parameters toward the direction of the best improvement of the estimated return, meanwhile penalizing too optimistic updates. PPO has the same stability of TRPO with an easier implementation, keeping a comparable overall performance.

In conclusion, after the introduction of the problems addressed by RL in terms of MDPs, this chapter explains many different Reinforcement Learning solving algorithms. In particular, Dynamic Programming is firstly introduced, since it is a starting point for many successive algorithms. Then, the focus is on RL algorithms: they are divided into Value Search methods, where FQI is the most appealing one, and Policy Search methods, where, starting from Policy Gradient, TRPO and PPO are the two best performing algorithms. For this reason, in Chapter 7, FQI, PPO and TRPO are the three algorithm chosen to be applied on the models designed in this thesis.

Chapter 4

Natural Language Processing

Natural Language Processing (NLP) is a text mining approach that aims to extract information from a text. The main question in NLP is: how can a computer extract information from a string representing a sentence?
A language is made by humans in order to interact efficiently with each other, so it is very difficult for a computer to interpret it. There are many levels of difficulty in understanding a sentence.

- **Morphological**: the first problem is how to encode every word of a language in order to be understandable by a computer. This is already very challenging, since there exist some words that can assume different roles in the sentences (*play* can be both a noun or a verb), there exist some words that have more than one meaning (*bank* is both the financial institution and the land alongside a lake) and there exist some words that are synonymous or have a similar meaning, pointing out the necessity to keep track of similarities between words.
- **Syntactic**: a sentence must follow syntactic rules in order to have sense, so an algorithm able to understand a sentence must understand the syntactic rules of a language.
- **Semantic**: a word can not be evaluated without the semantic context, otherwise it may even happen to understand the opposite of a sentence (not considering the *not* before a verb makes the meaning of a sentence to be exactly the opposite).
- **Contextual**: a sentence with the same meaning can be formulated in very different ways depending on the context (the same person in an official document will use different words than on Twitter) and on the person.

Because of all these problems it is immediately clear that there is no optimum algorithm to extract information from a text, but there exist some well performing algorithms that

are widely used in the literature and they are presented in the following sections. In particular, computers only understand numbers, not characters or sentences, so the first step to make a computer understanding sentences is to map each word of a text to a real-valued vector in a certain vector space, in order to be understandable by a computer. This kind of techniques are called **Word Embeddings** and they are based on the idea to map each word to one vector, such that the information about the morphology, the syntax, the semantics and the context can be taken into account by using the norm between vectors defined in the vector space used.

4.1 Traditional Approach: Bag of Words Model

A popular and simple group of methods is called *bag-of-words model*. The *bag* term is adopted because any information about the structure of the text is discarded and the algorithm only focuses on the words appearing in a text, without considering any relationship among them. So, a text can be considered a "bag" where it is possible to check if some important words are contained or not in order to extract information. The easiest and most intuitive bag-of-words approach to the problem of transforming words of a sentence into vectors is to use **one-hot encoding**. The basic idea is that each word is equal to itself and different to all the others. Hence each word can be represented with a sparse vector that has the same dimension N of the dictionary of the language considered and it is composed with all 0s except for one cell that is 1, which is always in the same position for the same word and in different positions for different words.
A variant to this model is to use the number of occurrences of a word into the considered sentence instead of 1, in order to give more importance to words more frequently appearing.

Example 4.1.1. In this example the following sentences are considered:

- *"Trump speaks to journalists in Pennsylvania"*;
- "*The President addresses the reporters in Philadelphia*".

A one-hot encoding of the most significant words can be:

$$
\begin{aligned}
speaks &= [0010...0000] \\
addresses &= [0000...0010] \\
Trump &= [0000...0100] \\
President &= [0001...0000] \\
Pennsylvania &= [1000...0000] \\
Philadelphia &= [0100...0000].
\end{aligned}
\tag{4.1}
$$

Every word is different, so the 1s are always in different positions. This example shows the main problem of this approach: the words *speak* and *address*, *Trump* and *President*, *Pennsylvania* and *Philadelphia* are highly related each other, in fact the two sentences have a very similar meaning, but using a one-hot encoding, since every word in the two sentences is different, will lead to the conclusion that the two sentences share no similarity, which is wrong.

Starting from the considerations made in Example 4.1.1, two huge drawbacks of this kind of approach are clear:

- the first one is that the representation of each word is very large dimensional and it creates a huge sparse dataset with memory and time complexity issues;
- the second problem is that in this representation every word is considered independently from the others, loosing all information about syntax, semantics and context (two consecutive words are not considered linked in any way). Moreover, also most of the morphological information is lost (two synonymous are considered completely independent exactly as two antonyms).

These two problems make the most intuitive approach not very useful for representing efficiently a sentence in order to extract information. This is the reason why more difficult approaches are discussed in this chapter. In particular, in Section 4.2 some complex Supervised Learning techniques called **Artificial Neural Networks** are explained. They are significant because they constitute a building block for the most important algorithms in this field, that are explained in Section 4.3. In particular, the most famous algorithm in NLP is shown in Subsection 4.3.3: it is called **Word2vec** [28] and it has been developed by researchers from Google in 2013. It is a Deep Learning model that represents words in dense and lower dimensional vectors and it is able to keep track of the semantic meaning between words from the same field. On top of this algorithm other approaches based on this idea of similarity between words have been developed. Among these, In Subsection 4.3.4, **GloVe** algorithm [32] is introduced: it is an algorithm developed at Stanford University and it is the approach adopted in this thesis for encoding words as vectors.

4.2 Deep Learning

Many algorithms that are shown in Section 4.3 are based on Deep Learning algorithms, therefore to explain them it is necessary to give an overview of Artificial Neural Networks and Deep Learning (an introductory description can be found in [27]). Moreover, Artificial Neural Networks need to be discussed since they are also applied many times in next chapters of this thesis.

4.2.1 Artificial Neural Networks

Artificial Neural Networks (ANN) are very powerful Machine Learning techniques. The idea behind them takes inspiration on the model of the biological neural networks of animals, which are composed by many elementary cells that interact with each other through the propagation of signals that in the end are summarized into an output signal of the network.
ANN are outperforming methods in Supervised, Unsupervised and Reinforcement Learning (the description of ANNs and applications on ML problems can be found in many books, examples are [7], [35] and [42]). Their power is that they can learn much more functions than traditional Machine Learning algorithms and, at the same time, they require less specific information about the context of the problem they are applied on, since they do not need any additional information than the data. Their main cost is computational, since they can be very complicated networks, requiring many computations to be properly tuned. Moreover, since they are completely based on data, they need data to be as much general as possible to avoid *overfitting*. For all these reasons, ANN were not used a lot in Machine Learning when the computing infrastructures were not much powerful, while they are very exploited nowadays. In next subsections some different groups of Artificial Neural Network are explained. In particular in Subsection 4.2.2 Feedforward Neural Networks are shown and in Subsection 4.2.3 Recurrent Neural Networks are presented. Finally in Subsection 4.2.4 a particular kind of Recurrent Neural Network called LSTM, that focuses on the concept of memory, is described.

4.2.2 Feedforward Neural Networks

Feedforward Neural Networks (FFNN) are the simplest form of ANN, which allow the signal to propagate only from input to output, without the possibility to propagate backward.
An Artificial Neural Network in general is made of many elementary cells called **neurons** or **nodes**, that are combined into subgroups called **layers**.
A neuron can be outlined as shown in Figure 4.1. It is a predictor of the form:

$$g(x) = \sigma(w^{\intercal} x), \tag{4.2}$$

which is the same structure of a simple algorithm used for Regression, called *Perceptron* [36]. This predictor takes a vector of inputs x, a vector of weights w and a bias b and it applies them on a function $\sigma : \mathbb{R} \to \mathbb{R}$, called **activation function**, producing a real number (the output signal of the neuron) as a result. The intuition behind a neuron is that it takes the signals from some previous neurons as inputs and it elaborates them using the activation function in order to produce its output signal. It is important to point out that in Equation 4.2 the vectors x and w are supposed to have first element

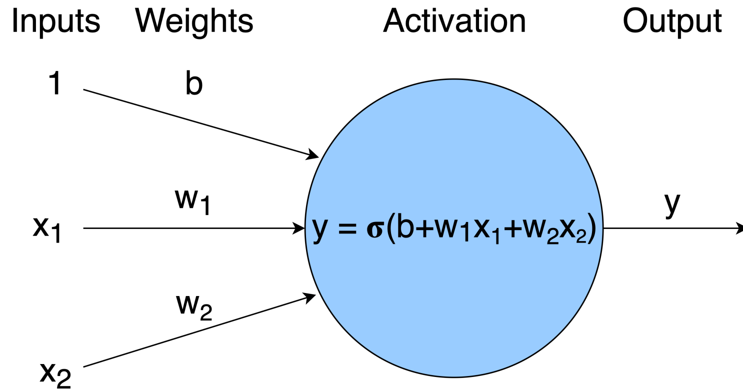


Figure 4.1: Scheme of a neuron, the elementary cell of an Artificial Neural Network.

respectively equal to 1 and bias b, so that in the computations the bias term is always incorporated in the scalar product $w^\intercal x$.

Since the activation function takes as input the weighted sum of signals, its main role is to introduce non-linearity to the network, so that it can model complex non-linear patterns in the data. Hence, the activation function of a neuron is always a nonlinear function, otherwise the network can be shrunk into a Perceptron. Typically the activation function is chosen among the following functions.

- **Sigmoid Activation Function:**

$$\sigma(z) = \frac{1}{1+e^{-z}}. \tag{4.3}$$

It is used in particular when the aim is to predict a probability, since its output is in the interval $[0, 1]$. Its main drawback is the *vanishing gradient*: when the absolute value of the input is a big number the derivative saturates toward 0, preventing the update of weights and the entire learning process.

- **Hyperbolic Tangent (Tanh) Activation Function:**

$$\sigma(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}. \tag{4.4}$$

The shape of the curve is similar to the *sigmoid* function but its image set is $[-1, 1]$, making it is mainly used in Classification between two classes. It is centered in 0, making it usually preferred to the *sigmoid* but it still presents the vanishing gradient issue that saturates the derivative of large inputs to zero.

- **Rectified Linear Unit (ReLU) Activation Function:**

$$\sigma(z) = \max(0, z) = \begin{cases} z & if\ z \geq 0 \\ 0 & if\ z < 0 \end{cases}. \tag{4.5}$$

It is easy to differentiate, its derivative is monotonic and it does not have the problem of *tanh* and *sigmoid* with large numbers. It also deletes some signals (all negative ones), reducing the variance and the risk of overfitting. The main problem of *ReLU* are *dying neurons*: it does not learn properly data with negative inputs, since they are immediately turned to zero.

- **Softmax Activation Function:** it is a multi-dimensional function, assigning a value in the range $[0, 1]$ to each element i of its m-dimensional output vector as follows:

$$\sigma(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{m} e^{z_j}}. \tag{4.6}$$

  By definition, the sum of the elements of its output is exactly 1. This is the reason why this activation function is usually used in the output layer when the aim of the network is to produce a probability distribution over m possible categories.

The choice of the activation function in a neuron is based on the problem faced and on the kind of output expected from the network. In hidden layers the most commonly used activation functions are the *ReLU* and *tanh*. Then the output layer is usually a *tanh* if the problem is a Classification between two classes, a *softmax* if it is a Classification among K different classes, a *ReLU* if the network performs a Regression.

A Feedforward Neural Network is a network made of many neurons, that in the end is able to evaluate a function $f : \mathbb{R}^d \to \mathbb{R}^n$, where d is the dimension of a datum of the dataset (the number of features) and n is the required dimension of the output.
A FFNN can be represented with a direct acyclic graph as in Figure 4.2, since there are no loops and the only allowed direction is the forward one.

It is possible to collect nodes into different **layers**, that are of three main types depending on their location. The first layer is called **input layer**, it is the group of nodes that take as input signal the data. In particular, the input of the node i of the input layer is the i-th feature of the datum considered. Then, there can be one or more **hidden layers**, groups of nodes taking as input the signals from the nodes of the layer before. Finally, the last layer is the **output layer** that takes as input the signals from the last hidden layer and produces as output the result of the ANN: it is made of n nodes and the ordered outputs of each one of them produces the n-dimensional vector required.
In general it is not necessary that every node is connected to all the nodes of the layer before or that it sends its signal to all the nodes of the following layer. If two nodes i, j are connected their edge is associated to a parameter $w_{i,j} \in \mathbb{R}$, that is a parameter of the network representing the weight given to the signal sent from i to j. This weight is used by node j in prediction, as expressed by Equation 4.2. Hence, given a node j and

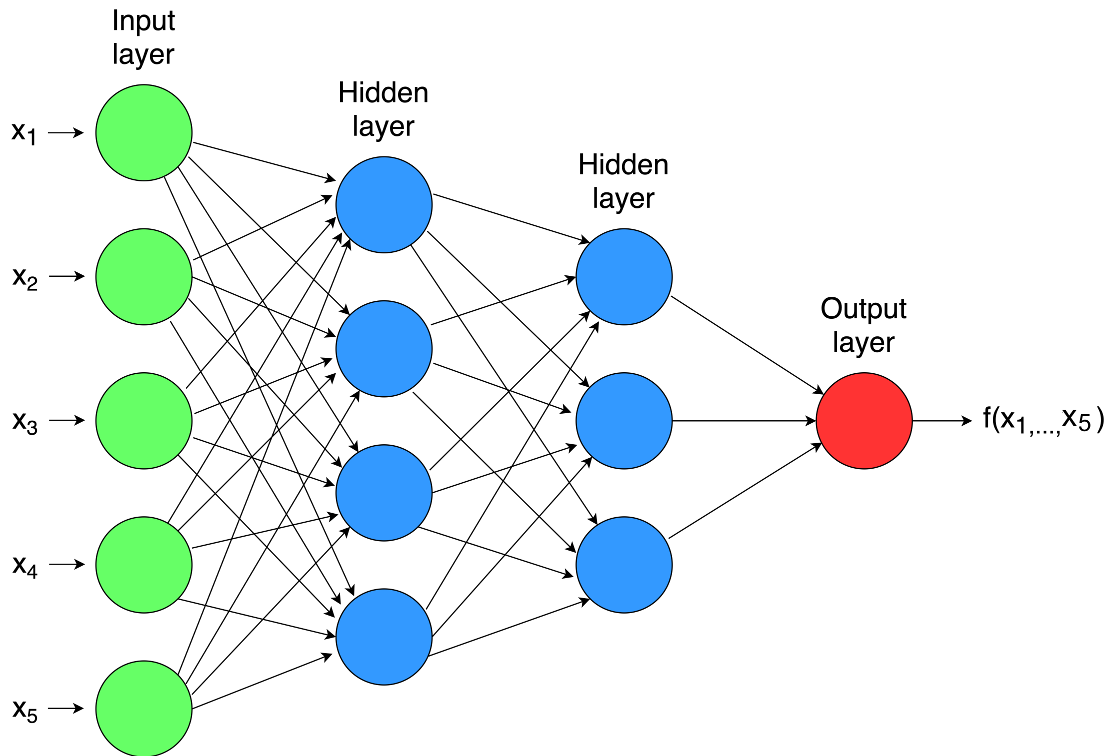


Figure 4.2: An example of fully connected Feedforward Neural Network with two hidden layers.

indicating with $w(j)$ the vector of all the weights associated with the incoming signals in the node and with $v(j)$ the vector of all the incoming signals in the neuron, the function evaluated by the Feedforward Neural Network is computed as follows:

1. every node in the input layer produces a signal $v_i = x_i$, where x_i is the i-th component of the input datum;
2. every node in the hidden layers produces a signal $v_j = \sigma(w(j)^\intercal v(j))$;
3. the k-th node in the output layer produces $f_k = v_k = \sigma(w(k)^\intercal v(k))$, the k-th component of the n-dimensional function f.

Summing up, a FFNN takes as input a d-dimensional datum x, it propagates forward the signal of the datum through the nodes of the network and it produces as output an n-dimensional vector that computes $f(x)$.

It is possible to prove that this kind of network is very significant, in fact it can be the representation of many different functions f, as suggested by the following result.

Theorem 4.2.1. *For every $d \in \mathbb{N}$ there exist a Feedforward Neural Network with $d+1$ nodes in the input layer, one hidden layer and one node in the output layer, so that the network contains all the functions of the form $f : \{-1, +1\}^d \to \{-1, +1\}$.*

Remark. It is possible to prove that the number of nodes needed to design a network able to represent every function $f : \{-1, +1\}^d \to \{-1, +1\}$ is exponential in the dimension d of the input.

Remark. The theorem can be extended to all functions $f : [-1, 1]^d \to [-1, 1]$ that are Lipschitz continuous.

These theoretical results prove that the space of the functions representable using a Feedforward Neural Network is huge, so this kind of network provides a wide hypothesis space when it is used by Machine Learning algorithms. On the other hand, the cost of using just one layer is to have an exponential number of nodes in it. In practice, it is often preferred to have a huge number of hidden layers each with a relatively small number of nodes, since this kind of approach seems to be able to represent a greater number of functions. This approach is called **Deep Learning** and these kind of networks are called **Deep Neural Networks**.

After the explanation of the structure of the network and its usage, it remains to show how to apply and train Feedforward Neural Networks in Supervised Learning algorithms. Remembering that a FFNN is able to evaluate different functions with different weights, the training of a Regression or Classification algorithm is based on tuning the weights of the network starting with some random or user-defined values, with the purpose of predicting the target of the training data using as input their features.
In particular, in a step of the training procedure, a random datum (or a mini-batch of data) is selected to compute the loss between the prediction and the real value of the target. Then, the weights of the network are updated in order to minimize that loss through the application of the Stochastic Gradient Descent algorithm:

$$w_{i,j} \leftarrow w_{i,j} - \alpha_t \frac{\partial \ell_{x_k}(W)}{\partial w_{i,j}}, \tag{4.7}$$

where the stochasticity is due to the random choice of the sample (or the mini-batch) on which evaluate the loss. In Equation 4.7, x_k is the randomly selected datum, α_t is the time-dependent update coefficient of the Stochastic Gradient algorithm, W is the set of all weights of the network and $\ell_{x_k}(W)$ is the loss evaluated in the datum and depending on the weights. Intuitively, each weight $w_{i,j}$ is updated in the opposite direction of the gradient of the loss, which represents the direction of maximum growth of the error.

The application of the Stochastic Gradient Descent algorithm to train the network is called **Backpropagation** ([37], [19] for a more extensive overview). Its name derives from the fact that, to compute the update of Equation 4.7, it is necessary to compute the partial derivative of the loss with respect to each weight. The computation of this derivatives is mainly based on the chain rule used to compute the derivative of a composite function: starting from the nodes in the output layer it is possible to deduce the value of the derivative of the loss with respect to the weights of nodes in the last hidden layer; then it is possible to compute the gradient of the loss with respect to the penultimate hidden layer; this can be *(back)propagated* until the first layer, so that the gradient of the loss with respect to any weight of the network is computed and they can be updated.

Summing up, with Backpropagation and the application of Equation 4.7 it is possible to use the train set to update the weights of a network toward a (local) minimum of the loss function, producing a network able to predict the target of data in the test set. It is also important to notice that FFNN are a very general approach, which can be applied in very different Supervised and Unsupervised Learning problems, with a relatively small number of choices to make: the design of the network and the choice of the activation functions.

4.2.3 Recurrent Neural Networks

The other main category of ANN is called Recurrent Neural Networks (RNN) and their peculiarity is that they allow loops in their architecture (it is possible to find a detailed discussion on RNN in [8]). They are important because, through the creation of loops between nodes, it is possible to introduce a sort of memory of the signal produced in previous iterations. Once again the idea behind RNN is inspired by biological neural networks, since the brains of animals do not start from scratch every second but they have memory, so that for example a human being can understand the last page of a book because he remembers all the others, not needing to restart to read from the beginning. This idea of memory is crucial in an ANN when data are not a plain dataset of rows but a sentence, an image or a time series, where data are not just different samples but they have a temporal or sequential order among them.
The basic structure of a RNN starts from the structure of a Feedforward Neural Network and adds to the input of next iteration not only the datum x_t but also an output signal from previous iteration c_{t-1}. In this way all next iterations have a signal related to previous ones, keeping track of the context while training the network.

From Figure 4.3 it is possible to better understand the structure of a RNN: the input layer, as usual, is made of one neuron for each feature $\{x_1, ..., x_d\}$, plus a certain number m of nodes $\{c_1^{t-1}, ..., c_m^{t-1}\}$ that are the memory terms deriving from the previous

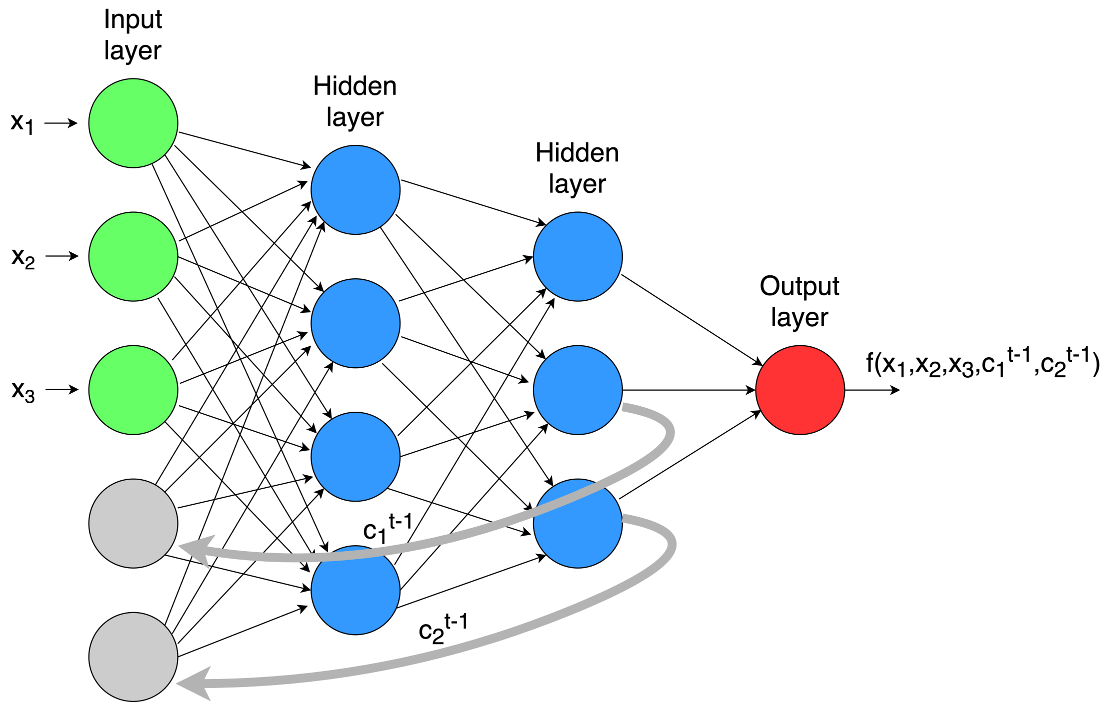


Figure 4.3: An example of Recurrent Neural Network with two hidden layers, three-dimensional data and two-dimensional memory vector.

iteration. In the network all nodes propagate their signal forward, combining the input datum x_t with the memory signal c_{t-1}. Moreover some nodes not only propagate their signal to nodes of next layer but they also loop it to the additional nodes of the input layer, producing memory terms $\{c_1^t, ..., c_m^t\}$ that are used in next iteration. In this way the output signal at time t is $h_t = \hat{f}(x_1, ..., x_d, c_1^{t-1}, ..., c_m^{t-1})$, which is the prediction of the function $f(x_t)$ obtained combining previous memory signals with current features. Summing up, Recurrent Neural Networks process one sequential input datum at time, keeping track of information about the history of all past elements of the sequence through loops.

In the learning process exploiting training data, the update of weights and the evaluation of the overall loss can be still computed applying Backpropagation. Indeed the network can be seen in its ”unrolled” version as in Figure 4.4, which makes the network a sequence of the same Feedforward Neural Network repeated as many times as the dimension of the dataset. The inputs of the t-th network are the datum x_t and the memory signal of the previous network c_{t-1}. In this way the Backpropagation algorithm can be applied to update weights because the network becomes a Feedforward Neural Network. However, for a big number of data the network becomes huge and it can be proved that

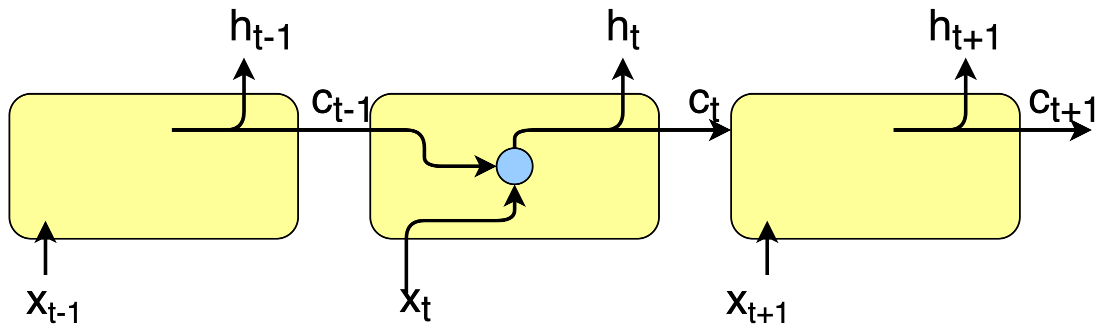


Figure 4.4: The unrolled version of a Recurrent Neural Network: at time t it takes as input the memory signal of previous step c_{t-1} and the current datum x_t. Then, hidden layers (represented by the blue circle) combine them producing the current signal, that is elaborated to produce the output $h_t = \hat{f}(x_t)$ and the next memory signal c_t.

the gradient with respect to weights becomes null [5]. Intuitively, this is due to the fact that the activation function used in nodes that form loops is usually a *tanh*, which has the advantage not to make the memory signal diverging since it is kept in $[-1, 1]$ at any iteration. On the other hand, when the network becomes deep due to many time-steps, the input of the *tanh* activation becomes huge, saturating the derivative toward 0. This is the reason why the main limitation of RNNs is long term memory: they are able to properly update their weights when the sequence of the dataset is short, while they are not efficient in doing so when the sequence is huge.

To conclude, this kind of network is necessary when data are sequential, but it can only deal with dependencies that are short term, since its structure does not allow to keep track of the information for many iterations, both for problems due to the Back-propagation and for the architecture of the network. To solve this issue, when long term memory is important, it is necessary another kind of approach that explicitly keeps track of the memory in a specific vector, so that it does not influence the update of weights. An algorithm based on this concept is explained in next subsection.

4.2.4 Long Short Term Memory (LSTM)

LSTM [20] is a specific Recurrent Neural Network since it is still based on loops to represent the memory. Thinking about it in the unrolled version as done in Figure 4.4 for RNN, the structure of a LSTM can be seen as the repeating of the same network that takes as input the datum at time t and a memory vector computed in iteration $t-1$. The particularity of LSTM is the peculiar way of combining the features with the

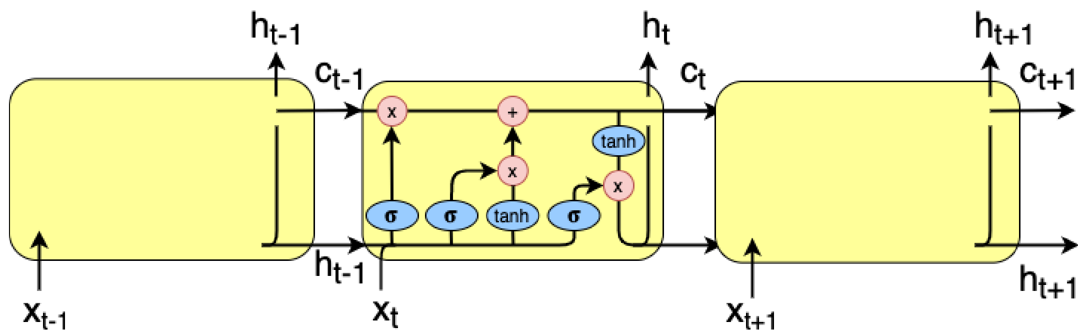


Figure 4.5: The unrolled version of LSTM: at time t it takes as input the memory signal of previous step c_{t-1}, the previous prediction h_{t-1} and current datum x_t. Then, four specific hidden layers combine them producing the current memory signal c_t, and the prediction $h_t = \hat{f}(x_t)$.

memory, since it uses specific hidden units in order to be able to remember inputs for long time when necessary.
In standard RNN presented in Subsection 4.2.3 the network has a single (or multiple) hidden layer combining the features of the datum with the memory terms using a user-defined activation function, while LSTM is made of four specific layers as shown in the unrolled version of LSTM in Figure 4.5. In particular, the line running on top of the diagram is called **cell state** and it is the part of the network that propagates the memory information from the previous network to the next one. Since in the current iteration the memory can be enriched by information from the current computations, the four layers of the LSTM determine what kind of information will be discarded or added to the memory signal. Specifically, the four layers of the LSTM are called **gates** and they perform four different tasks. The four gates of LSTM are shown in Figure 4.6 and they are precisely explained in the following list.

1. **Forget Gate Layer**: it is the first layer and it determines how much memory signal coming from the past will be discarded. It takes as input the current datum x_t and the output of previous network h_{t-1}. Then it applies them on the *sigmoid* function in order to generate a number in $[0, 1]$ for each node of the layer, whose size is equal to the dimension of the memory vector c_{t-1}. In this way, if the output of the *sigmoid* is near to 0 that memory component will be almost fully forgotten, while if it is near to 1 that component of the memory will be mostly remembered. The output of this gate is therefore the vector $f_t \in \mathbb{R}^m$ such that:

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f), \tag{4.8}$$

where W_f are the weights of the layer and b_f is the bias term. The output f_t

is successively multiplied component-wise by the memory signal c_{t-1}, in order to remember the percentage of each component of h_{t-1} as determined by f_t. An example where the prediction is the next word in a sentence makes more clear the reason why some components of the memory are discarded depending on the new input: if the datum x_t is a verb, the old predicate kept in memory becomes not important anymore and it can be discarded.

2. **Input Gate Layer** (first part): this layer and the following are responsible of the new information to store in memory. In particular this layer is made of nodes with *sigmoid* activation function, so that it determines for each component of the memory how much information to add (since the *sigmoid* outputs a value in $[0, 1]$ as explained for the Forget Gate Layer). For example, if x_t is a new male subject and the previous one was a female, it is probable that the network will give a value near to 1 to the component responsible for the gender, in order to save in memory the change of information about it. The value of the output vector of the layer is the vector $i_t \in \mathbb{R}^m$ computed as:

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i). \tag{4.9}$$

   This vector has exactly the same function of f_t but it is focused on the new information.

3. **Input Gate Layer** (second part): this layer determines the value of new information to save in memory. When forgetting a percentage of memory it is enough to decide how much information to forget for each component, while for adding information it is not enough to decide how much information to add for each component. Indeed it is necessary to non-linearly scale and compose the input signal making it compatible with the memory vector c_t. This is the role of this layer, that computes the vector $\tilde{c}_t \in R^m$ of value:

$$\tilde{c}_t = \tanh(W_c \cdot [h_{t-1}, x_t] + b_c). \tag{4.10}$$

   As usual, the activation function is applied to the weighted sum of components of the input plus is a bias term. The activation function of these layer is a *tanh*, that is responsible to determine the value of the signal computing a continuous value in the range $[-1, 1]$, ensuring this signal to be comparable to the one already stored in c_t.
   The output i_t of the *sigmoid* Input Gate Layer is combined with the one produced by this layer $\tilde{c}_t$ with a component-wise multiplication, producing for each component the magnitude of the new information weighted by how much it is taken into account.

Finally, the resulting m-dimensional vector is added to the memory signal from previous iteration, which has already being filtered by Forget Gate Layer. The result of this procedure is the new memory signal:

$$c_t = f_t \cdot c_{t-1} + i_t \cdot \tilde{c}_t, \tag{4.11}$$

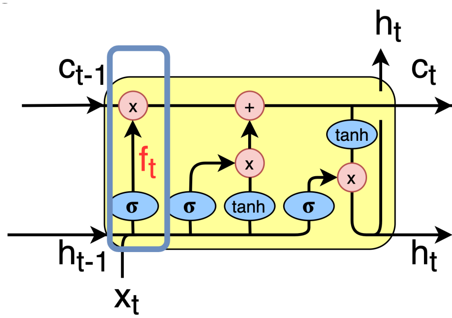
that is the information that will be propagated to the next iteration.

4. **Output Gate Layer**: it is the last layer, the one responsible of the output of this step of the network. Using the cell state c_t, that is the updated memory (so it contains both information about the memory and information of the current input), it is non-linearly scaled in $[-1, 1]$ by applying a *tanh* activation function. Then, a *sigmoid* function is applied to the vector made of previous output and input, producing an m-dimensional vector responsible of choosing what components of c_t are important for the output. Therefore:

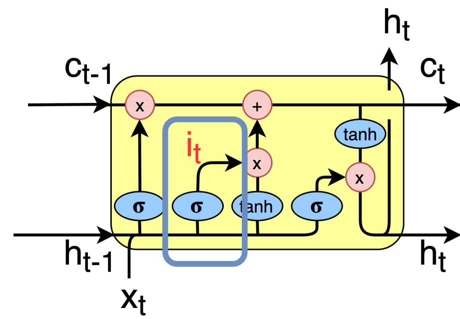
$$\begin{aligned} o_t &= \sigma(W_o[h_{t-1}, x_t] + b_o), \\ h_t &= o_t \cdot \tanh(c_t). \end{aligned} \tag{4.12}$$

For example, if the input is a subject and last output was a comma, the current output could be information about a predicate, underlying that it is probable that next word of the sentence will be a predicate.

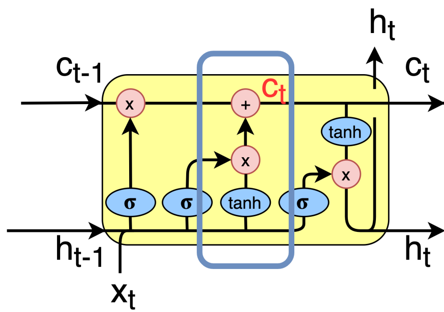
Summing up, LSTM is a particular kind of RNN with four layers that are responsible to update the memory signal c_t and to produce the output h_t. The core idea of LSTM is that the memory is not implicitly propagated through loops in hidden layers but it is directly propagated through a specific memory cell, while hidden layers are only responsible for updates. LSTM is a widely applied algorithm for sequential data, since it has not many backpropagation problems as basic RNN and it is capable to keep track of long term memory if needed. Many variants of LSTM have been proposed in literature (Gated Recurrent Unit [12], Depth-Gated LSTM [46]) but they produce almost the same empirical performances, so they will not be part of this essay.
Now that both Feedforward Neural Networks and Recurrent Neural Networks (with focus on LSTM) have been largely discussed, it is finally possible to introduce efficient Word Embedding algorithms such as *Word2vec* and *GloVe*.



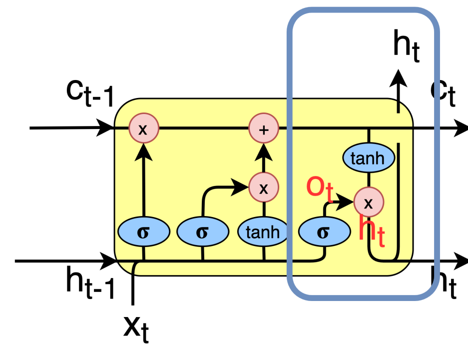
(a) Forget Gate Layer.



(b) Input Gate Layer (first part).



(c) Input Gate Layer (second part).



(d) Output Gate Layer.

Figure 4.6: The four gates of the LSTM network.

4.3 Word Embedding

As already discussed in Section 4.1, the most intuitive approach to map words into vectors is a one-hot encoding approach, that has two main problems related to the sparsity and huge dimension of each vector (*curse of dimensionality*) and to the lack of consideration to similarity and context between words.
Any technique mapping a word from a huge dimensional vector space, where it is represented in one-hot encoding fashion, into a dense lower dimensional vector space is called **Word Embedding**, since it *embeds* the vector in a smaller dimension.
An efficient Word Embedding typically maps a word w from a vocabulary V (with dimension $|V| \geq 10^6$)) in a much smaller vector space (with dimension usually in the range $[100, 500]$). The huge decrease of dimension of the vector space solves the curse of dimensionality issue. Also the problem regarding the lack of similarity can be solved with a proper mapping. In particular, a *continuous representation* rather than a one-hot encoding is preferable, since it allows the introduction of a distance between words, so that the more they are similar the more they will be close in the vector space.

Example 4.3.1. The embedding of a word maps it from one-hot encoding to a much smaller continuous vector:

$$house = [0\ 0\ 0\ ...\ 0\ 1\ 0\ ...\ 0\ 0\ 0] \in \mathbb{R}^{10^6} \Rightarrow house = [0.12\ 0.22\ ...\ -0.78] \in \mathbb{R}^{100}.$$

In Subsections 4.3.1 and 4.3.2 are introduced two Word Embedding algorithms respectively based on Feedforward Neural Networks and Recurrent Neural Networks. Then, in Subsections 4.3.3 and 4.3.4 the two most important Word Embedding algorithms presented in this thesis are discussed: *Word2vec* and *GloVe*.

4.3.1 Feedforward Neural Net Language Model (NNLM)

The first Word Embedding algorithm based on FFNN discussed in this thesis is called **Feedforward Neural Network Language Model (NNLM)** [6]. As shown in Figure 4.7, it consists of four specific layers.

1. An **input layer** that, in the embedding of the t-th word w_t, takes as input the n previous words $(w_{t-n}, ..., w_{t-2}, w_{t-1})$ in one-hot encoding fashion. The dimension of this layer is the number n of previous samples considered (a common choice is $n = 10$) times the dimension of each word, that in one-hot encoding is the dimension of the dictionary of that language $|V| \approx 10^6$.

2. A **projection layer** that maps every word from one-hot encoding (dimension $\approx 10^6$) into a smaller dimension m (typically $500 < m < 2000$) using a shared projection matrix $U \in \mathbb{R}^{|V| \times m}$. The dimension of this layer is the number n of the context words considered times the dimension m of each projected word.

3. A nonlinear, typically with *tanh* activation function, **hidden layer**. It is responsible of non-linear combinations between features and it has dimension h (usually the number of nodes of the hidden layer is $500 < h < 1000$).

4. An **output layer** that, applying the *softmax* activation function, outputs $|V|$ probabilities, one for each word in the dictionary of that language. They are a probability distribution among all words of what will be the t-th word w_t, based on the context of n previous words.

The NNLM approach focuses its attention on predicting the next word w_t given the contest of n previous words. Once it is trained over an appropriately huge dataset of sentences it implicitly performs a Word Embedding. Indeed, in the projection layer each word is projected into a lower dimensional vector through the shared matrix U that, once optimized, reduces the dimension of each word to m. In particular the mapping of any word w can be performed by computing $w^{\intercal}U$ that, since the word is encoded in one-hot encoding, is equivalent to select the i-th row of the matrix U, where i is the position of the 1 in the vector w.
The main problem of this approach is in the training complexity: the projection computes $n \times m$ operations, the hidden layer computes $n \times m \times h$ operations and the output layer computes $h \times |V|$ operations. Total complexity Q is:

$$Q = n \times m + n \times m \times h + h \times |V|, \tag{4.13}$$

where the projection step provides the minor effort on it.
To conclude, NNLM provides a major contribution on building an efficient Feedforward Neural Network starting from words in one-hot encoding fashion. On the other hand, it has too much computational complexity due to its final purpose of predicting the next word, that is not the aim of Word Embedding. Hence, using this model just for obtaining the m-dimensional word vectors is inefficient. The fact that the embedding is implicit and it is not the main purpose of the network suggests that it is possible to design a more efficient network.

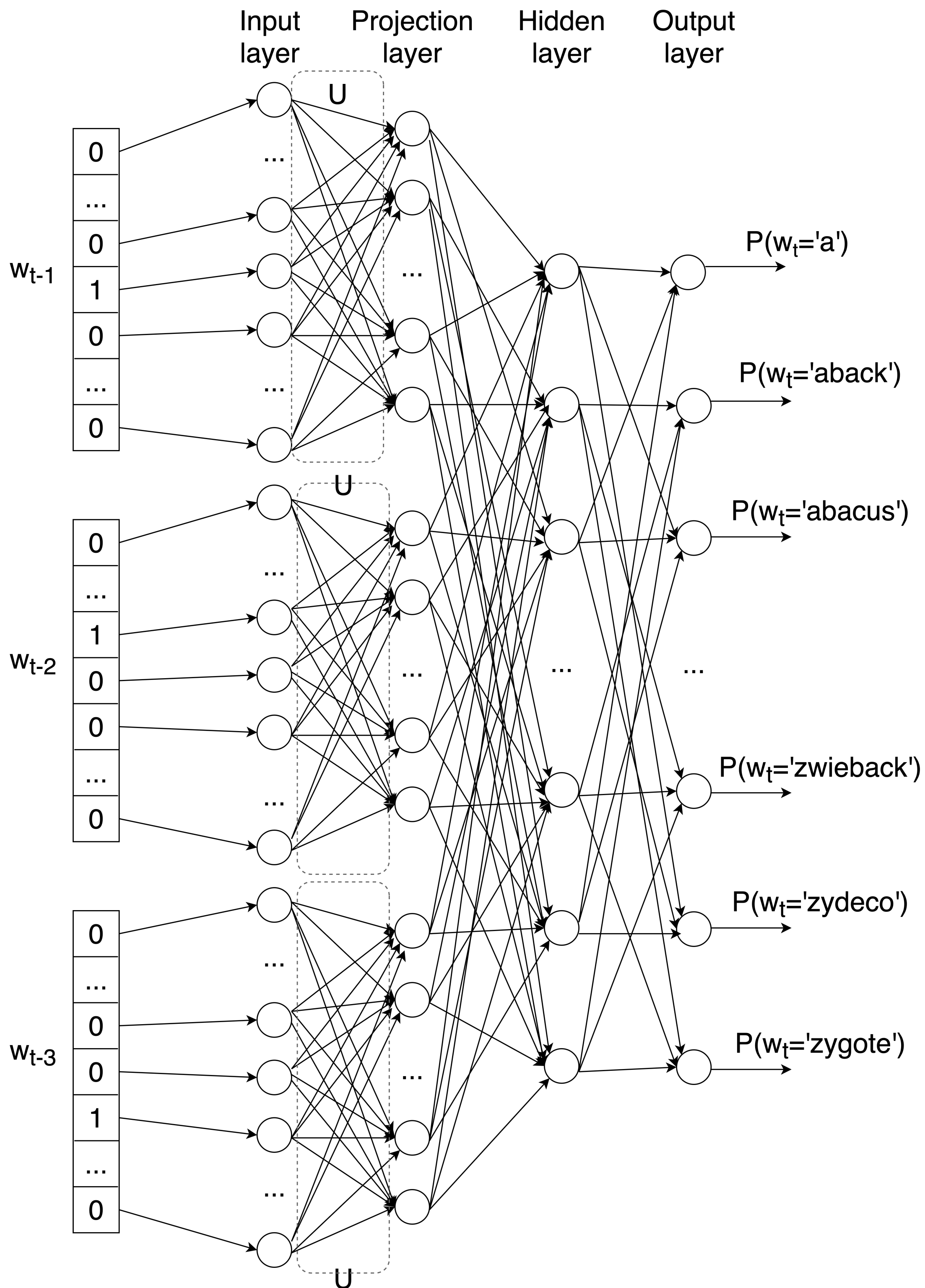


Figure 4.7: An example of NNLM with $n = 3$ context words.

4.3.2 Recurrent Neural Net Language Model (RNNLM)

Recurrent Neural Networks are designed in order to keep in memory the previous information, as largely discussed in Subsection 4.2.3. They have been introduced in Word Embedding [29] to keep in memory the important information from previous words instead of choosing a fixed number n of context words as in NNLM. RNN can store temporary information about the context for an arbitrary long time (although it is very difficult to keep long term information as explained in Subsection 4.2.3). In this kind of approach, as shown in Figure 4.8, the network is designed as follows:

1. an **input layer** takes as input the previous word w_{t-1} and the memory signal h_{t-1} coming from the neurons in the hidden layer whose output is in loop;
2. then an **hidden layer**, also called *context layer*, is made of m nodes (usually $m \in [30, 500]$) with *sigmoid* activation function. It elaborates the information from the context together with the new input w_{t-1}, producing the current signal;
3. finally, an **output layer** takes as input the updated signal producing a probability distribution (through a *softmax* activation function) over all words in the vocabulary. Therefore the output is the probability distribution of next word given the previous one and the context (coming from the memory loop).

Remark. Unlike NNLM, RNNLM has no projection layer, so there is not an explicit Word Embedding, that is done inside the hidden layer depending on weights. Hence it is not possible to recover a matrix to apply to any other word in order to embed it into a smallest vector space.

The number of computations in the hidden layer is $m \times m$ and in the output layer it is $m \times |V|$, therefore the total complexity Q of the algorithm is:

$$Q = m \times m + m \times |V|. \tag{4.14}$$

As in NNLM, most of the complexity comes from the hidden layer.
In conclusion, this approach improves NNLM because it designs a simpler Neural Network, easy to implement and train, with a smaller complexity. Moreover, RNN are specifically designed for sequences, so they do not need context words thanks to their recursive structure. On the other hand, with this approach there is no explicit embedding of the words through a matrix, so it is more complex to extract a Word Embedding rule to apply to any word. Finally, the algorithm is again not efficient for Word Embedding, since most of the complexity comes from the hidden layer, that is used for prediction purposes.

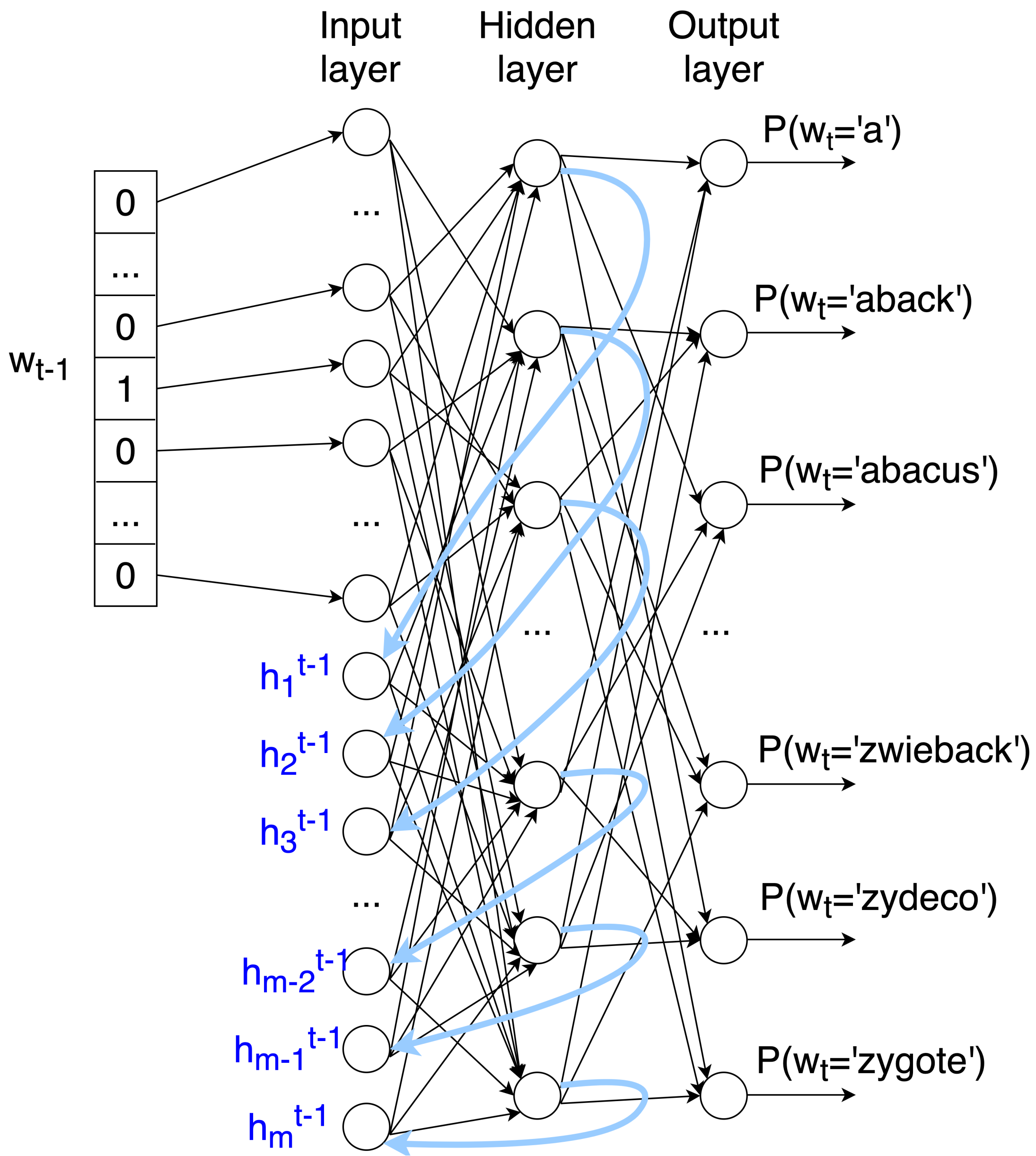


Figure 4.8: The network of the RNNLM.

4.3.3 Word2vec

Word2vec is an efficient Word Embedding tool proposed by researchers from Google in 2013 [28]. It is composed by two different algorithms that focus on the context in order to efficiently learn the embedding of a word based on the words around it in a sentence. This approach aims to produce a *distributed representation* of words by the train of a proper Feedforward Neural Network.
Starting from the FFNN designed in the NNLM, the main purpose of this approach is to minimize the computational complexity producing an efficient representation of words. The core idea is to achieve better performance by simplifying the network and train it on more data. In particular most of the complexity in both the NNLM and the RNNLM is due to the non-linear hidden layer, so *Word2vec* algorithms design networks without hidden layers, leading to a speedup of thousand times.
This approach proposes two different algorithms to perform Word Embedding.

1. **Continuous Bag-of-Words Model (CBOW)**: this network is similar to the Feedforward NNLM shown in Figure 4.7 but the non-linear hidden layer is removed and the projection layer is shared for all words. Moreover, the context words used as input are not just n previous words but there are also n following words. Therefore, choosing $n = 2$, the network that predicts the word w_t consists of $w_{t-2},\ w_{t-1}$ and also $w_{t+2},\ w_{t+1}$. Both the projection matrix U and the projection layer are shared, so each component is projected into the same node and basically averaged with the components in same position of all context words. This is the reason why the model is called *Bag-of-Words*, since it is not considered the order among them but they are just considered a "bag" of context words. Finally, the output is the probability distribution over all words of the vocabulary in order to predict the current word w_t.
   The complexity of projection layer is the number of context words $N = 2n$ multiplied by the dimension m of the projected word vectors, while the computations to perform to output the probability distribution are equivalent to the number m of nodes in projection layer repeated one time for each word in vocabulary V. Therefore the total complexity Q is:

$$Q = N \times m + m \times |V|. \tag{4.15}$$

   This equation shows that the term producing a huge complexity in NNLM and RNNLM is no longer present, making the algorithm much faster. At the same time, the network is still equipped with a projection matrix U as NNLM so, after a proper training of the network, $U \in \mathbb{R}^{|V| \times m}$ is able to project any word of the vocabulary V into a much smaller dimensional space, performing the required word embedding.

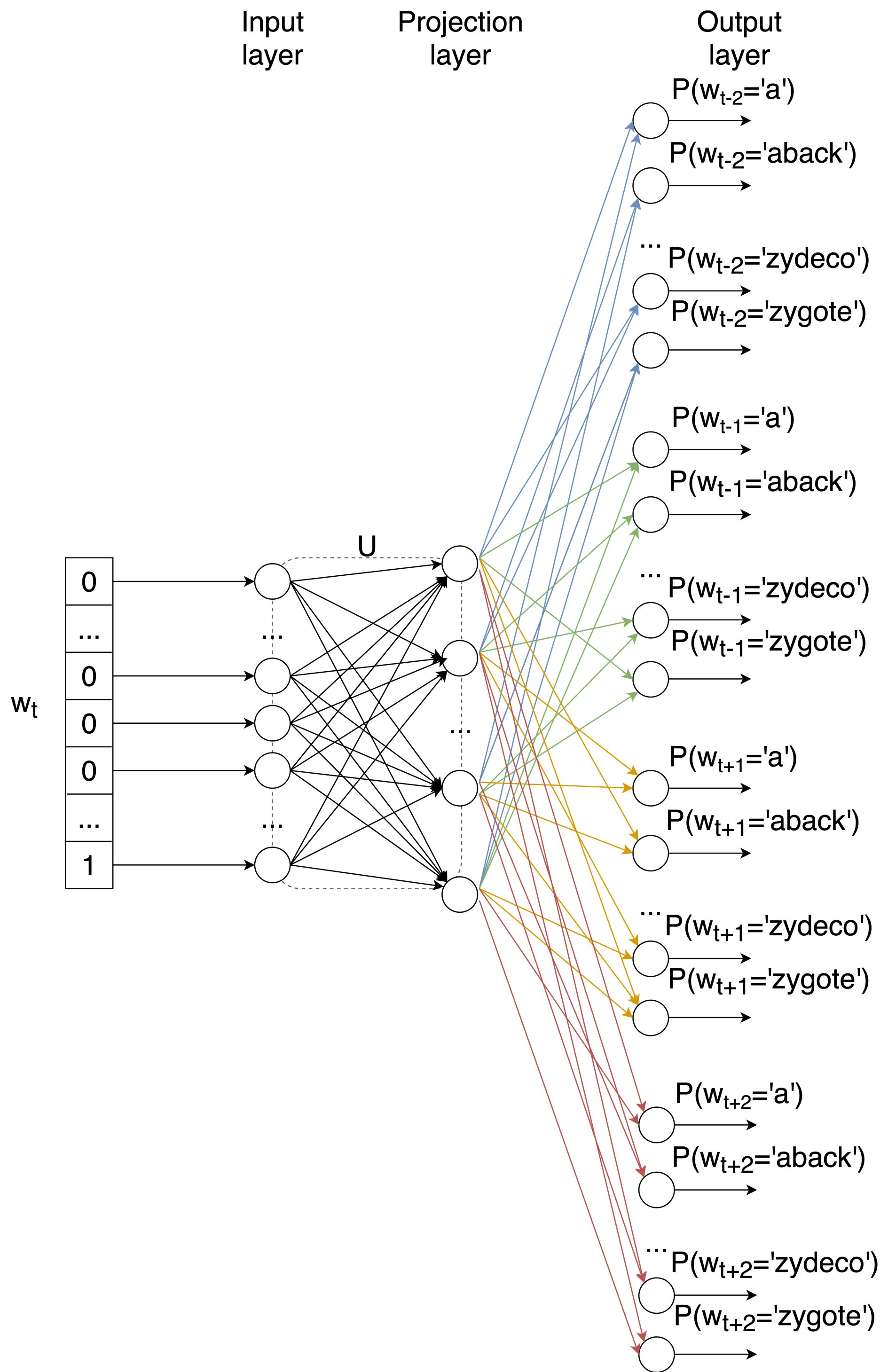


Figure 4.9: Scheme of Skip-Gram model with $n = 2$. Different colours of arrows are related to different context words.

2. **Continuous Skip-gram Model**: it is the complementary network of CBOW. Indeed, taking as input the current word w_t in one-hot encoding fashion, it is trained to predict the words around it. Considering as before $n = 2$, two words before and two words later are the context, so the network predicts $\{w_{t-2}, w_{t-1}, w_{t+1}, w_{t+2}\}$ based on w_t. In particular, as shown in Figure 4.9, the network is made of an input layer that only receives the current word w_t; then in a projection layer the vector is projected in a lower dimensional vector space performing the Word Embedding; finally an output layer outputs a probability distribution among all words in the vocabulary V for each context word.
   The training complexity requires m operations for the projection, then to output a probability distribution on each of the $N = 2n$ context words are needed $m \times |V|$ operations as usual. Therefore the total complexity Q is:

$$Q = m + N \times m \times |V|. \tag{4.16}$$

Summing up, both the approaches are based on Feedforward Neural Networks, they take inspiration from the NNLM presented in Subsection 4.3.1 removing its hidden layer. In this way the two algorithms simplify the network in order to be faster to train (the training speed is between 100000 and 5 millions of words per second). On the other hand, the quality of word vectors significantly improves when using a huge dataset, so the quality is also better than NNLM (that are too complex to be trained on millions of data) despite of the semplification of the network.
The quality and speedness of this Word Embeddings can be shown through some examples. In particular in Example 4.3.2 complexity of the algorithms is explored training them on a real dataset. Then the other Examples provide some evidences that the distributed representation of words obtained by these algorithms has remarkable regularities, since some intuitive semantic similarities are respected by word vectors, suggesting that the Word Embedding is consistent. To visualize these regularities, in the word vector space is applied a PCA keeping the first two components.

Example 4.3.2. Training the networks presented in this chapter on a Google news corpus, containing six billion words and restricting the vocaboulary to one million most frequent words, time for training has been:

- **2 days** on 140 cores for CBOW with projection space of dimension $m = 1000$;
- **2.5 days** on 140 cores for skip-gram with projection space of dimension $m = 1000$;
- **14 days** for NNLM on 180 cores with projection space only of dimension $m = 100$;
- RNNLM was discarded in such a huge training since it was already performing very poorly with easier datasets and weeks of training time.

It is clear from this example that the two proposed algorithms in *Word2vec* have a similar complexity, while NNLM and RNNLM have a similar theoretical approach but they are not possible to perform in practice.

Example 4.3.3. Simple operations between word vectors provide good results. As shown in Figure 4.10a word vectors implicitly encode properly gender and number of words:

- *man* is similar to *woman* as *king* is similar to *queen*. Also *man* is similar to *king* as *woman* is similar to *queen*. This is caught in the vector space:

$$man - woman \simeq king - queen; \tag{4.17}$$

- *man* is similar to *men* as *king* is similar to *kings*. Also *man* is similar to *king* as *men* is similar to *kings*. In the vector space:

$$man - men \simeq king - kings. \tag{4.18}$$

Example 4.3.4. As shown in the PCA plot of Figure 4.10b, another example of regularity in word vectors are names of countries and the name of their capitals. Therefore, for example:

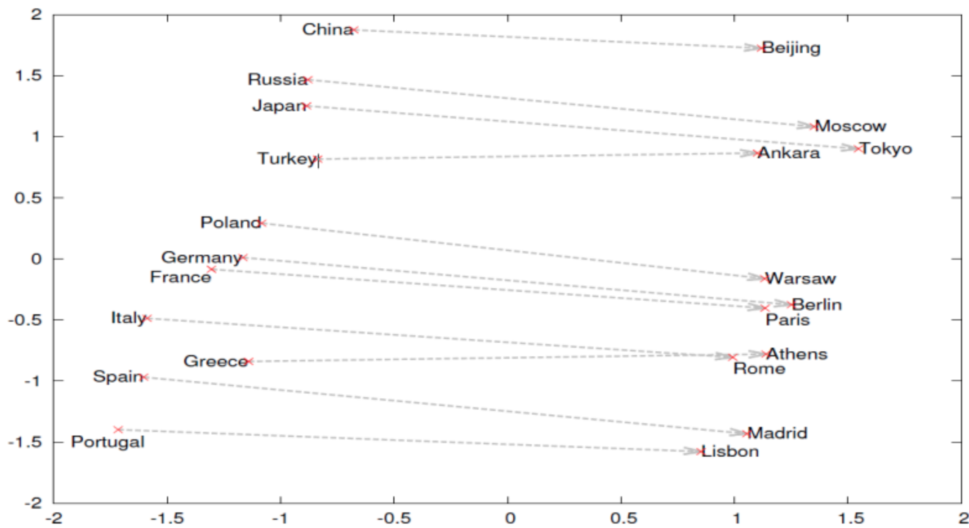
$$Italy - Rome \simeq France - Paris. \tag{4.19}$$

Not all pairs country-capital are exactly represented, but the overall result is satisfactory.

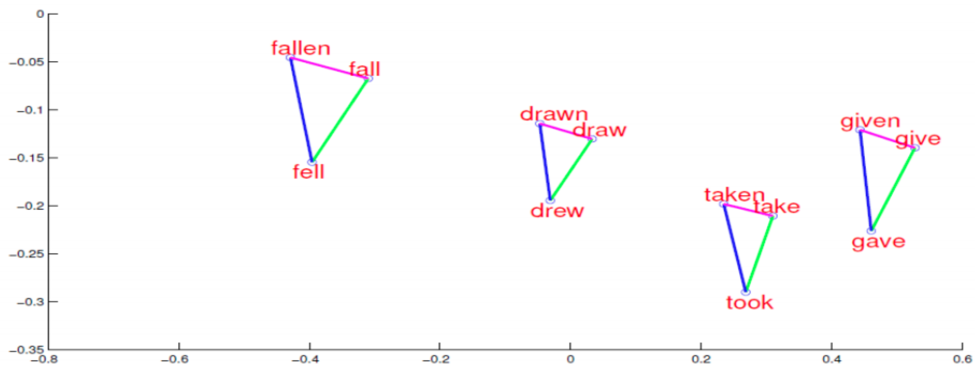
Example 4.3.5. As shown in the PCA plot of Figure 4.10c, the paradigms of irregular verbs have a clear common pattern.

All the examples discussed above show that the proposed algorithms are not only fast to train but also well performing, since they recognize many regularities among words. In conclusion, these two algorithms show that it is possible to extract good word vectors with very simple networks, making them important and widely used tasks to perform Word Embedding on sentences. The word vectors produced by *Word2vec* tool can successively be used in many NLP approaches, like translation of sentences into other languages or sentiment extraction.

(a) Example of gender and number regularities.



(b) Example of regularities between countries and their capitals.



(c) Examples of regularities between irregular verbs.

Figure 4.10: Examples of regularities applying *Word2vec* algorithms.

4.3.4 Global Vectors for Word Representation (GloVe)

The examples about *Word2vec* empirically show some syntactic regularities that are not determined by some kind of constraint chosen in the model. This regularity is implicitly gained by the methods due the optimization of the networks, showing the robustness of the technique. On the other hand in *Word2vec* there is no certainty about the regularity among all words of the vocabulary. *GloVe* algorithm [32] is an Unsupervised Learning method where the model is explicitly designed to exploit regularities, making them emerge in the word vectors. In practice, GloVe algorithm is designed so that the difference between two word vectors can represent as much as possible the semantic difference between them. If, for example, $man - woman$ should represent the gender difference between the two words (they have the same meaning despite of the gender), this difference vector is constrained by the model to be as much as possible equal to $king - queen$, $brother - sister$, $mum - dad$ and so on.
A more sophisticated kind of context is introduced in this setting: calling $X_{i,j}$ the number of times word j occurs in the context of word i and X_i the total number of times any word appears in the context of word i, it is possible to introduce a probability:

$$p_{i,j} = p(j|i) = \frac{X_{i,j}}{X_i}, \tag{4.20}$$

that is an estimate of the probability that word j appears in the context of word i. The elements $X_{i,j}$ are called **word-word co-occurrence counts** and they are stored in the matrix X, while the probabilities $p_{i,j}$ are called **co-occurrence probabilities**. Considering a practical example it is possible to understand how to proper use these information to extract meaningful context signals.

	$k = solid$	$k = gas$	$k = water$	$k = fashion$
$p(k\|ice)$	1.9×10^{-4}	6.6×10^{-5}	3.0×10^{-3}	1.7×10^{-5}
$p(k\|steam)$	2.2×10^{-5}	7.8×10^{-4}	2.2×10^{-3}	1.8×10^{-5}
$p(k\|ice)/p(k\|steam)$	8.9	8.5×10^{-2}	1.36	0.96

Table 4.1: Examples of co-occurrence probabilities and their ratio.

Example 4.3.6. Considering $i = ice$ and $j = steam$, it is expected that a word like $k = solid$ will often be in the context of i and much less in the context of j, while a word $k = gas$ is expected to be much more in the context of j than in the one of i. On the other hand, a word like $k = water$ should be almost equally linked to i and j, while a word $k = fashion$ should be almost never related to them. As shown in Table 4.1 (performed on a six billion words train document) the ratio between the co-occurrence probabilities $p_{i,k}$ and $p_{j,k}$ shows better and easier than raw probabilities what is expected

to happen. Indeed a ratio much larger than 1 means that k appears much more in the context of i, a ratio much smaller than 1 means that k is much more frequent in the context of word j and a ratio near to 1 shows that k is almost equally frequent in the two contexts.

Inspired from the Example above, the main idea of *GloVe* algorithm is to use ratios of co-occurrence probabilities to train the Word Embedding. Hence the general model F is:

$$F(w_i, w_j, w_k) = \frac{p_{i,k}}{p_{j,k}}. \tag{4.21}$$

The estimates of co-occurrence probabilities $p_{i,k}$ and $p_{j,k}$ can be derived from the train document. Therefore knowing the function F that maps any triple of word vectors $w_i, w_j, w_k \in \mathbb{R}^m$ to their co-occurrence probabilities ratio, enables to extract the word vectors, so to perform the Word Embedding.
It remains to estimate the function F, that in principle can be any function but it is possible to select a unique choice by constraining it to have some properties.

- F must encode the information of $\frac{P_{i,k}}{P_{j,k}}$, that can be done using word vector differences. In this way Equation 4.21 is modified to:

$$F(w_i - w_j, w_k) = \frac{p_{i,k}}{p_{j,k}}. \tag{4.22}$$

- It has a vector input but a scalar output, so the most natural way to output a real number is performing a scalar product:

$$F((w_i - w_j)^\intercal w_k) = \frac{p_{i,k}}{p_{j,k}}. \tag{4.23}$$

- The function must be invariant to exchanges between a word and a context word, which leads to the following relation:

$$w_i^\intercal w_k + b_i + b_k = \log(X_{i,k}), \tag{4.24}$$

  where b_i, b_k are bias terms and $X_{i,k}$ is the word-word co-occurrence count between word w_i and context word w_k.

Summing up, the general model of Equation 4.21 has been drastically simplified in Equation 4.24. The objective of the training of *GloVe* algorithm is therefore to learn word vectors such that their dot product is equal to the logarithm of their co-occurrence count plus bias terms. In the algorithm the learning is performed through the optimization of a specific Weighted Least Squares Regression where the loss function is given by:

$$J = \sum_{i,k=1}^{|V|} f(X_{i,k})(w_i^\intercal w_k + b_i + b_k - log(X_{i,k}))^2. \tag{4.25}$$

It is basically a Least Square problem that aims to learn the parameters w_i, w_k, b_i, b_k best fitting the model of Equation 4.24 minimizing the sum of squares, weighted by a proper weighting function $f(X_{i,j})$. This function is introduced to give more importance to co-occurrences between words that appear frequently, without overweighting both rare words that are noisy and give less information and very frequent words that may obscure the others. Among many possible weighting function, the class of function adopted in *GloVe* algorithm are the continuous functions that start from 0, reach 1 in a certain value x_{max} and they have value 1 from that point on:

$$f(x) = \begin{cases} (x/x_{max})^{\alpha} & if\ x < x_{max} \\ 1 & otherwise \end{cases}. \tag{4.26}$$

In conclusion, *GloVe* is a different approach with respect to all Neural Network based approaches shown before in this chapter, it is based on co-occurrences of words in a text, and after a proper Weighted Least Squares optimization, it encodes the word vectors as resulting parameters. *GloVe* uses both global and local statistics to make the Word Embedding and not only the local statistics of n near words as *Word2vec* approach, giving more generality to the word vectors obtained. Moreover, because of the nature of the optimization, *GloVe* word vectors perform well on word analogy tasks as like as *Word2vec*, but this is explicitly imposed by the optimization and it is not the result of hidden computations like in *Word2vec* approach.
Despite the strong differences between the two approaches, both *GloVe* and *Word2vec* are well performing and widely used tools for Word Embedding, so they are performed as a starting point for almost any Natural Language Processing algorithm to convert words into vectors in a way understandable by a computer.

Chapter 5

Datasets

In this chapter the datasets used in this thesis are introduced. In particular this work compares the performance of Reinforcement Learning algorithms trained with the historical series of a financial instrument with the same algorithms trained adding sentiment features to the training set. Indeed the final purpose is to investigate, through Supervised and Reinforcement Learning, whether the sentiment features add some information to the historical series for the prediction of the value of a financial index. In particular, in Section 5.1 the *S&P 500* index and its historical series are presented. Then, in Sections 5.2 and 5.3, the sentiment dataset is respectively discussed and enriched with some features extracted from the date and the historical series, leading to the final dataset shown in Figure 5.9 that is the basis of all the applications in this work.

5.1 S&P 500 Index

This section presents the *S&P 500* index and discusses the reasons why its historical series are adopted in this thesis as representative of the trend of the U.S. Stock Market, focusing on the shape of its curve to explain the first critical issues in dealing with these data.
The *Standard & Poor's 500 (S&P 500)* [21] is a Stock Market index composed by a basket of the 500 U.S. companies with largest market capitalization, whose importance in the index is weighted by this parameter. Market capitalization is indeed computed as the stock price multiplied by the number of stocks on the Market so, basically, these are the 500 most important American companies on the Stock Market. Therefore, the *S&P 500* index is considered one of the best indicators of the overall performance of the American Stock Market, reporting the risk and the return of the most important companies in it. The companies in the index have about the 80% of the total market capitalization in the American stock market, so the *S&P 500* captures most of the

market, making it a very reliable index.
A committee selects the companies to be included in the index, which is updated every March, June, September and December of every year. To be in the *S&P 500* index a company must have some precise requisites [23]:

- it must be a U.S. company;
- its market capitalization must be at least $6.1 *billions*;
- at least the 50% of its stocks must be available to the public;
- it must have produced positive earnings in the last year;
- its stock price must be at least $1 per share.

The best companies (hence the most influential ones) in the index in September 2019 are reported in Table 5.1 [22]. Also the composition of the index in terms of economic sectors reflects the Market, as reported in Figure 5.1 [21], with Information Technology that is by far the leading sector, followed by Health Care and Finance.

Company	Market Cap
Microsoft	$1428 billion
Apple	$1400 billion
Amazon	$1035 billion
Alphabet	$1017 billion
Facebook	$605 billion
Berkshire Hathaway	$555 billion
Visa	$449 billion
JPMorgan	$423 billion
Johnson & Johnson	$398 billion
Walmart	$330 billion

Table 5.1: The 10 companies with largest market capitalization.

The value of the *S&P 500* index is the historical series of prices used in this thesis. It has been adopted for different reasons: firstly, although it is not possible to directly invest in the *S&P 500* there exist some financial instruments able to replicate its trend, so using the value of the index as if it were the price of a stock market on which it is possible to invest can be practically performed on the Stock Market. Moreover, the *S&P 500* is chosen as reference price in this thesis because the purpose is not only to directly invest but it is also to explore the meaningfulness of the sentiment on predicting the trend of the U.S. Market and, as already explained, the *S&P 500* index is a reliable

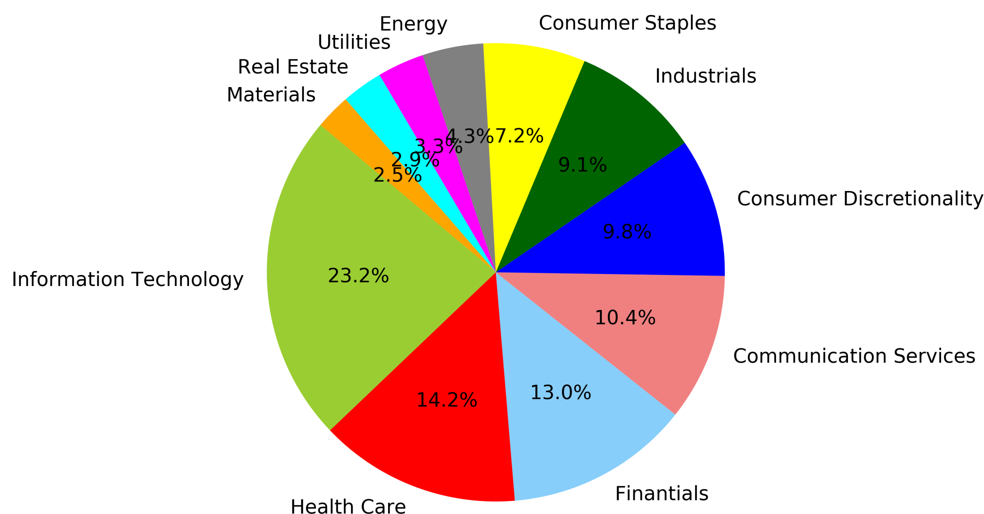


Figure 5.1: Pie Chart of the sector subdivision of the index.

index to capture it with good approximation. Finally, this index has been chosen because it is the index adopted by applications already performed in the main project this thesis is based on, therefore it is logical to use the same dataset already available and applied in the project.
The Market dataset available is therefore the historical series of daily values of the *S&P 500* index from *2009-02-02* to *2019-09-26*, for a total amount of 2682 data. The historical daily series of the index is available since 1980, but the composition and the value of the index are very different from actual ones and the sentiment features described in following sections are only available since *2009-02-02*, therefore also the historical series of the index is used from that date. As shown in Figure 5.2 the dataset is made of seven features.

- *referenceDate*: it is the date corresponding to each datum, which are all opening days of the U.S. stock market in the eleven considered years.
- *ID*: a unique ID identifying each datum (so each working day) is added to the features to uniquely represent each row of the dataset with an integer number.
- *close*: it is the closing value of the index, which is the value of the *S&P 500* at 4 p.m., the time when stock trading stops.
- *high*: it is the highest value the index assumes during the opening hours of the stock market, i.e. the highest value registered from 9.30 a.m. to 4.00 p.m.
- *low*: it is the lowest value the *S&P 500* assumes while the U.S. Stock Market is

	referenceDate	ID	close	high	low	open	volume
0	2009-02-02	0	825.440002	830.780029	812.869995	823.090027	5673270000
1	2009-02-03	1	838.510010	842.599976	821.979980	825.690002	5886310000
2	2009-02-04	2	832.229980	851.849976	829.179993	837.770020	6420450000
3	2009-02-05	3	845.849976	850.549988	819.909973	831.750000	6624030000
4	2009-02-06	4	868.599976	870.750000	845.419983	846.090027	6484100000

Figure 5.2: The first five market data.

open.

- *open*: it is the opening value of the index, which is the value of the *S&P 500* at 9.30 a.m., the time when stock exchanges start.
- *volume*: it is the total amount of shares of all companies belonging to the index traded during the considered day.

The scatterplot of Figure 5.3 shows the shape of the trend of the open value of the index during the period considered in the dataset. The curve is almost monotonically increasing, hence it is immediately clear that there may be an issue complicating a non-trivial learning: between two consecutive days it is probable that the index is increasing, therefore the agent may learn the trivial policy to always buy stocks, performing always the same action. Moreover, the value of the index in 2009 is around 800, while in 2019 it is around 3000. These values are very different and they make impossible to keep the value of the index or the difference between two consecutive values as features, because they have very different magnitudes. Hence, as explained in next sections, the value chosen to balance the dataset is the percentage difference between two consecutive days, so that the percentage does not take into account the values, scaling the data.

In conclusion, the *S&P 500* index is the chosen historical series, it is a consistent representation of the trend of the U.S. Stock Market and its historical series is adopted in this thesis to represent the Market value. On the other hand, it presents a clear increasing pattern, that may cover the other relations between the index and the features, preventing the learning of more complex relations.

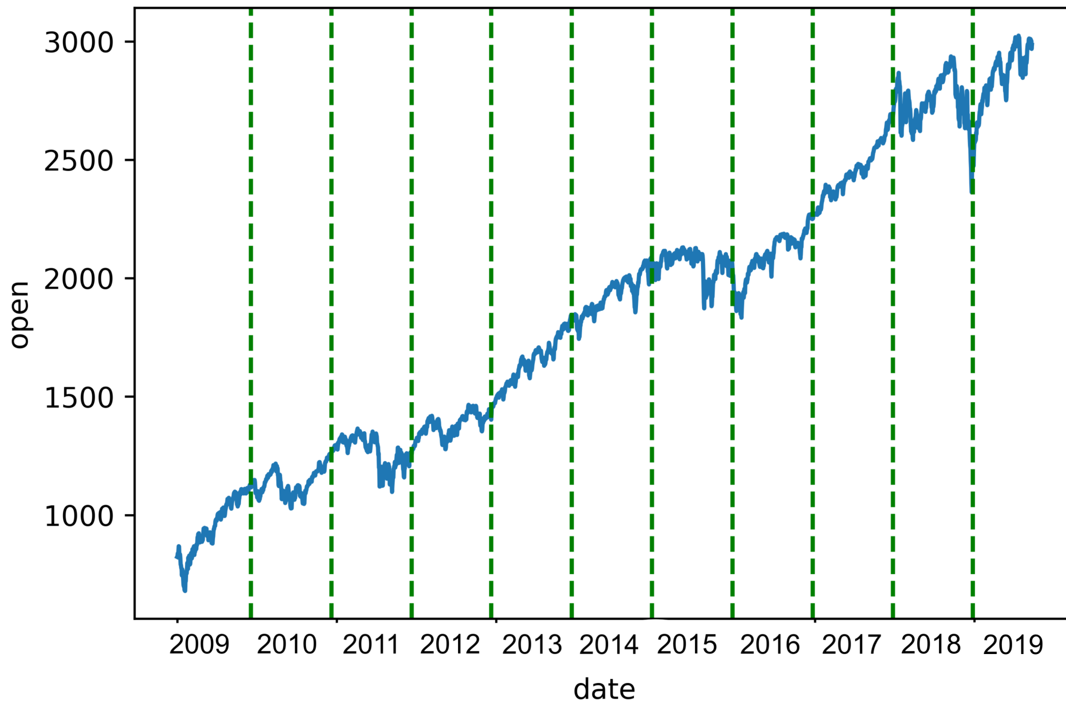


Figure 5.3: The curve of the open value of the index in the available data.

5.2 Daily Sentiment

In this section the datasets regarding the sentiment features are shown. The *S&P 500* historical series presented in Section 5.1 and the sentiment features extracted in this section are the starting point of all the features and targets of any algorithm applied in this thesis based on daily data.
In particular, two daily sentiment signals are available from the project this work is based on, which are the focus of the analysis that is performed in next chapters. These sentiment signals are based on sentences and news coming from two different sources, tweets from Twitter and Reuters news, and they have been extracted as explained more in detail in the following subsections.

5.2.1 Twitter Daily Sentiment

Every day approximately 500 millions of tweets are sent by people coming from all over the world with all possible kind of contents. Therefore Twitter is a very frequented virtual public place where also investors and companies share their messages or opinions.

This is the reason why a sentiment extracted from the tweets published in a day regarding the American Stock Market may be a relevant signal about the overall sentiment of the components of the Market, suggesting if the Market will perform well or badly the next day. On the other hand, it is also possible that the sentiment of tweets is not a predictive signal of what will happen on the Market but a reaction signal to what already happened, making it not informative for predicting the future values. This is one of the reasons why this work is focused on finding whether the sentiment is significant for predicting the Market trend or not.
In this framework, the tweets considered to extract a daily sentiment signal are the ones that present as *hashtag* or *cashtag* at least one of the assets in the *S&P 500*.
Then, to each selected tweet is associated a sentiment feature extracted through the application of a Deep Artificial Neural Network. In particular, Word Embedding has been performed applying *GloVe* algorithm (explained in Subsection 4.3.4). Then, an Artificial Neural Network is designed and trained, with the aim to extract a sentiment score from the word vectors. The hidden layers of the network are composed by two Feedforward hidden layers (introduced in Subsection 4.2.2) responsible for combining information from the word vectors extracted from the tweet, followed by two LSTM (explained in Subsection 4.2.4) able to exploit the sequentiality of the words in a sentence. Finally, the output layer is made of a single fully connected node that maps all the signals produced by the last LSTM layer into a single real number in the interval $[0, 1]$, that is the sentiment associated to the considered tweet. The training both of GloVe algorithm and the Neural Network is performed using a set of sentences, labeled with a sentiment value, made of 1.583.691 Tweets described in [18].
Once the trained Artificial Neural Network is exploited to associate a sentiment to any of the selected tweets regarding the *S&P 500* companies, it is necessary to summarise them, producing a daily sentiment feature. Hence, for each company composing the index and for every hour of a day, the tweets referring to that company are averaged, producing the hourly sentiment score associated to each company. When there are no tweets in an hour referring to a company a **forward filling** procedure is applied: the last hourly sentiment value referring to that company is selected until a new tweet about the company appears. A possible drawback of *forward filling* arises when there are no tweets related to a company for a long period of time. In this case there is no information about the actual sentiment of that company but the procedure still represents it with an obsolete one, which may be substantially different.
The next step is to aggregate all hourly sentiments referring to the same hour of the same day through a weighted average of hourly sentiment values of each company, weighted by the market capitalization of the company, in a fashion similar to the one that is applied by Standard & Poor's to compute the *S&P 500* index. In this way a sentiment signal from tweets over the index is produced for every hour of the day, so it only remains to

	Date	keras_d01	keras_d02
0	2017-04-03	0.690246	0.684221
1	2017-04-04	0.685419	0.690246
2	2017-04-05	0.681478	0.685419
3	2017-04-06	0.682438	0.681478
4	2017-04-07	0.693694	0.682438

Figure 5.4: First five Twitter sentiment features referring to the two previous days.

average the 24 hourly signals of a day to compute the estimated Twitter sentiment value of that day over the *S&P 500* index (that is considered to be a good approximation of the sentiment over the U.S. Stock Market).
In conclusion, in the project this thesis is based on, a sentiment signal for each day has been extracted from Twitter as explained above and the features selected to perform algorithms are the sentiment of one day before and the one referring to two days before the current date, as reported in Figure 5.4. Hence these are the two features initially available in this work referring on tweets.
Unfortunately, this features are available only from *2017-04-03*, for a total amount of about 500 data, that are not enough to significantly perform Machine Learning algorithms. For this reason Twitter daily sentiment is not used in the algorithms applied in this thesis, that therefore focuses on Reuters sentiment explained in Subsection 5.2.2.

5.2.2 Reuters Daily Sentiment

Reuters Machine Readable News are financial news delivered by Reuters together with some features that can be applied by algorithms. In particular, as shown in Figure 5.5, each news is equipped with three signals which represent the probability for each news to be positive, negative or neutral. Therefore, exactly as explained for Twitter sentiment in Subsection 5.2.1, it is possible to extract a daily Reuters daily sentiment signal.
In particular, all news of a certain hour referring to a company are collected and the sentiment signal referring to each news is computed as the difference between the positive and the negative sentiment signal of the news. Then, the hourly average sentiment of each company is calculated, the hourly mean sentiment over the *S&P 500* index weighted by the market capitalization of each company is evaluated, and finally the daily sentiment over the index is produced averaging the 24 hourly signals.

```
{'_id': ObjectId('5c6ecff7aa07ad2f280f3b78'),
 'analytics': {'newsItem': {'marketCommentary': False,
   'wordCount': 5897,
   'exchangeAction': 'UNDEFINED',
   'companyCount': 1,
   'sentenceCount': 235,
   'headlineTag': '',
   'bodySize': 41088},
  'systemVersion': 'TS:40060099',
  'analyticsScores': [{'sentimentNeutral': 0.361112,
    'assetClass': 'CMPNY',
    'sentimentClass': -1,
    'noveltyCounts': [{'window': '12H', 'itemCount': 0},
     {'window': '24H', 'itemCount': 0},
     {'window': '3D', 'itemCount': 0},
     {'window': '5D', 'itemCount': 0},
     {'window': '7D', 'itemCount': 0}],
    'sentimentNegative': 0.364482,
    'assetId': '4295861027',
    'firstMentionSentence': 1,
    'priceTargetIndicator': 'UNDEFINED',
    'relevance': 1.0,
    'sentimentWordCount': 861,
    'volumeCounts': [{'window': '12H', 'itemCount': 0},
     {'window': '24H', 'itemCount': 0},
     {'window': '3D', 'itemCount': 0},
     {'window': '5D', 'itemCount': 0},
     {'window': '7D', 'itemCount': 0}],
    'sentimentPositive': 0.274406,
    'assetName': 'NGEx Resources Inc',
    'assetCodes': ['P:4295861027', 'R:NGQ.TO'],
    'brokerAction': 'UNDEFINED',
    'linkedIds': []}]},...
```

Figure 5.5: An extract of a Reuters Machine Readable News: every news is equipped with a positive, negative and neuter sentiment score. Moreover, the tuple with key *assetName* encodes the asset the news is referring to, which in this case is the *NGEx Resources Inc.*

	referenceDate	reutersSentimentRelevance_d01	reutersSentimentRelevance_d02
0	2009-02-02	0.010516	-0.002345
1	2009-02-03	0.039298	0.010516
2	2009-02-04	0.024336	0.039298
3	2009-02-05	0.028467	0.024336
4	2009-02-06	0.042225	0.028467

Figure 5.6: First five Reuters sentiment features referring to the two previous days.

This daily signal extracted from Reuters Machine Readable News can be considered the Reuters sentiment referred to that day and, exactly as Twitter sentiment, in the considered project the sentiment values referred to the two previous days are selected as features, ad shown in Figure 5.6, therefore they are the two features initially available in this work referring on Reuters news. Moreover, the Reuters data are available from *2009-02-02*, making the dataset much larger than Twitter-based dataset (the number of available days is 2682). For this reason, as already anticipated, the daily sentiment features utilized in this thesis are only Reuters sentiment features, which allow to consider more than ten years of daily data.

In conclusion, the data available at the beginning of this work are the *S&P 500* index historical series (shown in Figure 5.2) for what concerns U.S. Market prices and the daily sentiment of Reuters Machine Readable News of the two previous days (shown in Figure 5.6) as regards sentiment features. Also daily sentiment from tweets is at disposal, but it is discarded due to the short time window of its availability. The daily data considered are therefore from *2009-02-02* to *2019-09-26*, for a total amount of 2682 data made of five market and two sentiment features.

5.3 Feature Extraction

The first purpose of this thesis is to evaluate the relevance of the sentiment features for predicting the next value of the Stock Market. It may happen that sentiment signals are correlated with other features not taken into account in the dataset, making the sentiment to seem relevant because of the correlation with other significant features and not for its own importance. Therefore, in this section, some basic features are generated from the data available, so that their significance can not be implicitly included

	referenceDate	Monday	Tuesday	Wednesday	Thursay	Friday	January	February	March	April
0	2009-02-02	1	0	0	0	0	0	1	0	0
1	2009-02-03	0	1	0	0	0	0	1	0	0
2	2009-02-04	0	0	1	0	0	0	1	0	0
3	2009-02-05	0	0	0	1	0	0	1	0	0
4	2009-02-06	0	0	0	0	1	0	1	0	0

May	June	July	August	September	October	November	December	NextPayment	PreviousPayment
0	0	0	0	0	0	0	0	0.540984	0.459016
0	0	0	0	0	0	0	0	0.524590	0.475410
0	0	0	0	0	0	0	0	0.508197	0.491803
0	0	0	0	0	0	0	0	0.491803	0.508197
0	0	0	0	0	0	0	0	0.475410	0.524590

Figure 5.7: The first five extracted features.

in the sentiment importance. Specifically, some features are extracted from the date of each day. Moreover, the historical series of values of the index are revised, as already anticipated, to consider relative variations between consecutive days, instead of raw values.

Firstly, the date is considered: the day of the week and the month of the year are extracted and encoded in a one-hot encoding fashion. Moreover, since in the most important investment fund based on the *S&P 500* index (the *SPDR S&P 500 trust ETF* [26]) coupon detachment happens on every Friday of the third week of March, June, September and December of every year, two features are added to represent the distance from last coupon detachment (*PreviousPayment*) and the distance to next coupon detachment (*NextPayment*). They are represented by a real number in the interval $[0, 1]$, so that the more the value of the feature is next to 0, the nearer is that event, the more the value is next to 1, the further is the event. In Figure 5.7 a sample of this extracted features is reported.

The other elaboration of the dataset is made on the historical series of the *S&P 500* index. Indeed, the open values of the index plotted in Figure 5.3 are in a too wide range and should be normalized. The chosen procedure is to consider the proportional differences between two consecutive days as features. In fact, the highest and lowest values of the index are not much reliable in a daily analysis because they always happen at different moments of the day, that are unknown a priori, hence it is much more complex to build a model based on them. Also the *close* value of the index is not the best choice

	referenceDate	R0	R1	R2	R3	R4	R5	R6	R7	R8	R9
10	2009-02-17	-0.018394	0.004868	0.003021	-0.045520	-0.001578	0.026179	0.017241	-0.007186	0.014630	0.003159
11	2009-02-18	-0.033655	-0.018394	0.004868	0.003021	-0.045520	-0.001578	0.026179	0.017241	-0.007186	0.014630
12	2009-02-19	-0.003982	-0.033655	-0.018394	0.004868	0.003021	-0.045520	-0.001578	0.026179	0.017241	-0.007186
13	2009-02-20	-0.015281	-0.003982	-0.033655	-0.018394	0.004868	0.003021	-0.045520	-0.001578	0.026179	0.017241
14	2009-02-23	-0.003377	-0.015281	-0.003982	-0.033655	-0.018394	0.004868	0.003021	-0.045520	-0.001578	0.026179

Figure 5.8: Five examples of proportional consecutive differences of open values of the index among ten previous days.

for a daily-based strategy, because it is only available at the end of the day, when the market closes and it is not possible to trade until the open of the next day, when the index may have changed value. Therefore, the features based on the daily historical series of the *S&P 500* adopted in this thesis are the proportional differences between two consecutive days within a certain time window of previous days. Specifically, the lag is set to be equal to 10, so that many previous day are considered, in order to give to Machine Learning algorithms the possibility to understand the trend, without going back too much, since far events are not such relevant as close ones.
Summing up, the features extracted from the historical series of the index are the percentage differences between the open values of two consecutive days among the ten days before, as shown in Figure 5.8 (for a better representation, the first ten data are not reported in the figure, since they have missing values corresponding to the previous days not present in the dataset, that are filled with 0s).

In conclusion, the complete dataset is made of daily data from *2009-02-02* to *2019-26-09* for a total amount 2682 rows. Each sample of the dataset is made of 31 features (plus the date, that is a string univocally identifying each datum) as reported in Figure 5.9: 10 of them derive from the historical series as percentage differences between two consecutive values of the *open* of the *S&P 500* index in the ten previous days; 19 features are extracted from the date representing the day, month and distance from coupon detachments; finally, 2 features are the daily sentiment obtained by Reuters news of the two previous days.

	referenceDate	Monday	Tuesday	Wednesday	Thursay	Friday	January	February	March	April	May	June	July	August
10	2009-02-17	0	1	0	0	0	0	1	0	0	0	0	0	0
11	2009-02-18	0	0	1	0	0	0	1	0	0	0	0	0	0
12	2009-02-19	0	0	0	1	0	0	1	0	0	0	0	0	0
13	2009-02-20	0	0	0	0	1	0	1	0	0	0	0	0	0
14	2009-02-23	1	0	0	0	0	0	1	0	0	0	0	0	0

September	October	November	December	NextPayment	PreviousPayment	reutersSentimentRelevance_d01
0	0	0	0	0.377049	0.622951	0.077040
0	0	0	0	0.360656	0.639344	0.049231
0	0	0	0	0.344262	0.655738	0.052306
0	0	0	0	0.327869	0.672131	0.031278
0	0	0	0	0.311475	0.688525	0.016094

reutersSentimentRelevance_d02	R0	R1	R2	R3	R4	R5	R6	R7	R8	R9
0.057927	-0.018394	0.004868	0.003021	-0.045520	-0.001578	0.026179	0.017241	-0.007186	0.014630	0.003159
0.077040	-0.033655	-0.018394	0.004868	0.003021	-0.045520	-0.001578	0.026179	0.017241	-0.007186	0.014630
0.049231	-0.003982	-0.033655	-0.018394	0.004868	0.003021	-0.045520	-0.001578	0.026179	0.017241	-0.007186
0.052306	-0.015281	-0.003982	-0.033655	-0.018394	0.004868	0.003021	-0.045520	-0.001578	0.026179	0.017241
0.012508	-0.003377	-0.015281	-0.003982	-0.033655	-0.018394	0.004868	0.003021	-0.045520	-0.001578	0.026179

Figure 5.9: Five rows of the dataset with all the available features derived both from market and sentiment data.

Chapter 6

Feature Selection

In this chapter Supervised Learning methods are applied to the dataset presented in Chapter 5 in order to understand the importance on the prediction of the market trend (represented by the *S&P 500* index) of the available features, with a special focus on the sentiment. The procedures applied in this chapter are essentially two.

- The first is a **Feature Selection**, so that the Reinforcement Learning algorithms will be trained considering the features that result to be relevant in this chapter. Moreover, the tuning and the performance of the Regression performed for selecting the features will already give some interesting results for the evaluation of the sentiment.
- The other procedure is specifically focused on the sentiment features, allowing to draw a first conclusion on their importance. In particular, it consists on performing a first Regression without the sentiment features and a second Regression algorithm with only the sentiment, having as target the residuals of the first Regression. In this way it is possible to evaluate if the sentiment features are able to extract something more than what already extracted from the other features about the reward.

In particular, since the final purpose is to apply Reinforcement Learning algorithms, the dataset is firstly elaborated in Section 6.1 so that each row is equipped with a target value that allows to perform a Supervised Learning Feature Selection considering as target the *reward* of the agent, making it robust for choosing the features relevant in the RL approach. Before any application, in Section 6.2 Random Forests are theoretically presented. They are a Supervised Learning method particularly useful for Feature Selection because they naturally rank features in the learning process, producing a hierarchy over them. Then, in Section 6.3, a Random Forest is tuned and trained, producing the first relevant results of this work. Finally, in Section 6.4, the second procedure is performed

using again Random Forests to explore more specifically the influence of the sentiment features over the learning of the target.

6.1 Features and Target: Expanded Dataset

Starting from the complete dataset presented in previous chapter and shown in Figure 5.9, the Supervised Learning algorithms presented in this chapter require an additional elaboration. In particular, they need the addition of a target value to each datum to be able to perform a Regression. Specifically, the purpose of all the approaches of this chapter is to extract information useful for the application of Reinforcement Learning techniques, therefore the target of interest is the *reward*.
The Markov Decision Processes modeled in this work will be described in Chapter 7, but it is possible to anticipate the essential elements that compose the reward of a step of the MDP. In fact, the possible **actions** that the agent can take at the beginning of each day are three: *buy*, *sell* or *do nothing*, encoded as $\{1, -1, 0\}$. Intuitively, if the agent predicts an increase of the value of the *S&P 500* index it should choose to buy (*long position*), with the idea of earn by selling the stocks that have increased their value in next days. If it predicts a decrease of the value it should sell (*short position*), which consists in sell stocks not owned with the idea to buy them later when they have a decreased value. Introducing the short position, it is given the unrealistic possibility to sell stock without owning them, which is illegal in the U.S. Stock Market, but it can be performed with a more complex procedure which consists in borrowing them, so this action is not completely impossible to perform in practice. Finally, if the prediction is that the index will be almost constant the agent should do nothing.
The other element that influences the reward is the action taken the day before, that is the actual **portfolio**, i.e. what the agent has in possession in current day before taking an action. Indeed, it can decide to keep the same position or to change its position, which implies transaction costs. If, for example, yesterday the agent with no open position ($portfolio = 0$) chose to buy (1) and today it chooses to sell (-1), yesterday it paid a fee for the transaction to bring its portfolio from 0 to 1, then today it has to pay a fee for selling the stocks it owns (from 1 to 0) plus a fee for the short selling (from 0 to -1), with a total change of its portfolio from 1 to -1.
With the ideas of action and portfolio, it is finally possible to define the reward of a step of the MDP. In fact, given the features, the action taken and the current position given by the portfolio, the reward is:

$$r_t = a_t \cdot \frac{(open_{t+1} - open_t)}{open_t} - |a_t - a_{t-1}| \cdot fees. \tag{6.1}$$

Basically, the first member of the equation is the percentage profit, computed as percentage increase of the open value of the index between current and next day, multiplied

by the action taken today a_t. In this way it is a positive profit if the correct action has been chosen by the agent, negative if the market moves in the direction opposite to the prediction, null if the chosen action is to do nothing. Then, the reward is computed subtracting to the profit the transaction fees due to the change of position between current portfolio (that is previous action a_{t-1}) and the chosen action (a_t). A modeling choice is to consider the fees a constant value estimated as \$7 for each \$100.000 of stocks bought, that can be refined as an improvement of current work by considering them proportional to the asset or to the amount of money invested. Therefore in the unitary portfolio considered in this framework the constant is $fees = 7 \times 10^{-5}$. The fees are not paid if $a_{t-1} = a_t$ because there is no change of position, they are paid once if there is only one change of position (from -1 to 0, from 0 to 1 or viceversa) and they are paid twice if there are two changes of position (from 1 to -1 or from -1 to 1), as expressed by $|a_t - a_{t-1}|$.

Summing up, every row of the dataset is a day and it is made of 31 available features. To evaluate their importance on the prediction of the reward of that day, it is necessary to add it as target, remembering that it depends on the action taken that day and on the portfolio inherited from previous day. Hence every day, depending on the combination of action and portfolio, there can be 9 different rewards as target. To have all possible scenarios all rows of the dataset need to be replicated nine times when performing Supervised Learning approaches that have the reward as target, so that every repetition of the same row (so the same day) has one of the nine possible combinations of action and portfolio as features and the corresponding reward as target. The same day repeated nine times with different action-portfolio combinations are actually 9 different samples, and action and portfolio in Supervised Learning are two additional features, that are included in the feature selection as all the others, since they influence the target (i.e. the reward). Therefore, the dataset used to perform the feature selection of Section 6.3 is made of 33 features and it is composed by 24129 samples (the 2682 samples of the original dataset repeated nine times, minus the last datum on which there is no next open value so it is not possible to compute the reward). An example of the nine samples extracted from the same day is reported in Figure 6.1, where it is possible to see that all 31 original features are the same in the same day, but the different combinations of action and portfolio lead to different target rewards.

referenceDate	Monday	Tuesday	Wednesday	Thursay	Friday	January	February	March	April	May	June	July	August	September
2009-02-17	0	1	0	0	0	0	1	0	0	0	0	0	0	0
2009-02-17	0	1	0	0	0	0	1	0	0	0	0	0	0	0
2009-02-17	0	1	0	0	0	0	1	0	0	0	0	0	0	0
2009-02-17	0	1	0	0	0	0	1	0	0	0	0	0	0	0
2009-02-17	0	1	0	0	0	0	1	0	0	0	0	0	0	0
2009-02-17	0	1	0	0	0	0	1	0	0	0	0	0	0	0
2009-02-17	0	1	0	0	0	0	1	0	0	0	0	0	0	0
2009-02-17	0	1	0	0	0	0	1	0	0	0	0	0	0	0
2009-02-17	0	1	0	0	0	0	1	0	0	0	0	0	0	0

October	November	December	NextPayment	PreviousPayment	reutersSentimentRelevance_d01	reutersSentimentRelevance_d02
0	0	0	0.377049	0.622951	0.07704	0.057927
0	0	0	0.377049	0.622951	0.07704	0.057927
0	0	0	0.377049	0.622951	0.07704	0.057927
0	0	0	0.377049	0.622951	0.07704	0.057927
0	0	0	0.377049	0.622951	0.07704	0.057927
0	0	0	0.377049	0.622951	0.07704	0.057927
0	0	0	0.377049	0.622951	0.07704	0.057927
0	0	0	0.377049	0.622951	0.07704	0.057927
0	0	0	0.377049	0.622951	0.07704	0.057927

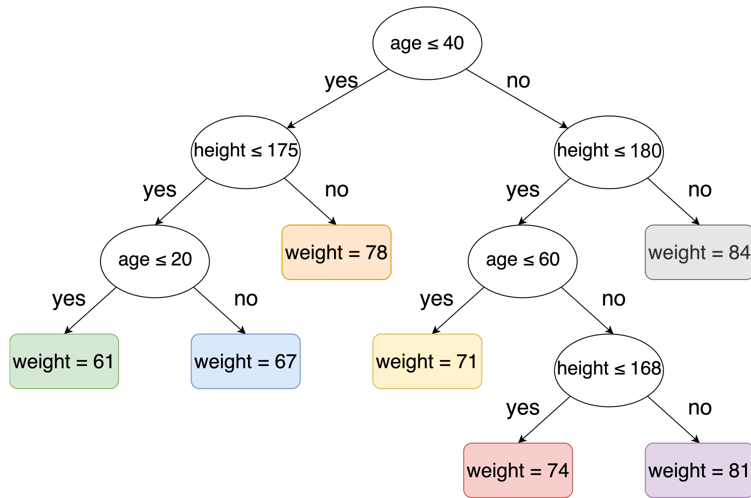
R0	R1	R2	R3	R4	R5	R6	R7	R8	R9	portfolio	action	reward
-0.018394	0.004868	0.003021	-0.04552	-0.001578	0.026179	0.017241	-0.007186	0.01463	0.003159	-1	-1	33.654594
-0.018394	0.004868	0.003021	-0.04552	-0.001578	0.026179	0.017241	-0.007186	0.01463	0.003159	-1	0	-0.070000
-0.018394	0.004868	0.003021	-0.04552	-0.001578	0.026179	0.017241	-0.007186	0.01463	0.003159	-1	1	-33.794594
-0.018394	0.004868	0.003021	-0.04552	-0.001578	0.026179	0.017241	-0.007186	0.01463	0.003159	0	-1	33.584594
-0.018394	0.004868	0.003021	-0.04552	-0.001578	0.026179	0.017241	-0.007186	0.01463	0.003159	0	0	-0.000000
-0.018394	0.004868	0.003021	-0.04552	-0.001578	0.026179	0.017241	-0.007186	0.01463	0.003159	0	1	-33.724594
-0.018394	0.004868	0.003021	-0.04552	-0.001578	0.026179	0.017241	-0.007186	0.01463	0.003159	1	-1	33.514594
-0.018394	0.004868	0.003021	-0.04552	-0.001578	0.026179	0.017241	-0.007186	0.01463	0.003159	1	0	-0.070000
-0.018394	0.004868	0.003021	-0.04552	-0.001578	0.026179	0.017241	-0.007186	0.01463	0.003159	1	1	-33.654594

Figure 6.1: One row of the dataset is transformed into nine different samples: each one has a different combination of action and portfolio, leading to a different target reward.

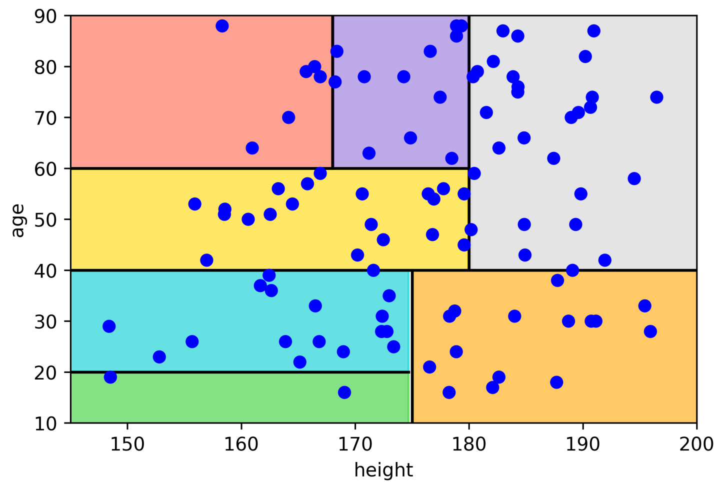
6.2 Random Forests

The Regression algorithm adopted to perform the Feature Selection are **Random Forests** [9]. In this section they are theoretically explained, clearing also the reason why they have been chosen.
A Random Forest is a combination of tree predictors, hence it is firstly necessary to introduce **Decision Trees** [10], [47]. A Decision Tree is a Supervised Learning method based on a tree model that in Regression problems predicts the value of the target y_i, given a datum $x_i \in \mathbb{R}^d$. Denoting with X the data space, a Decision Tree iteratively produces axis-parallel hyperplanes to recursively split the data space partitioning it, until the points inside each set of the partition are relatively homogeneous in terms of target y_i. Once the tree is trained, X is partitioned collecting train data into subgroups. It is then possible to assign each test point to one subgroup, so to a leaf of the tree, and its target can be predicted as mean value of the targets of the train data belonging to the same set. Summing up, as shown in Figure 6.2, a Decision Tree consists of internal nodes that split the data depending on the value of a feature selected for that node, and leaf nodes that represent a set of the partition of X and they are labeled with the predicted value of the target of data in that set, computed in regression as the mean of train targets.
As said, at each internal node, one attribute is selected to split training samples in two subgroups, maximizing a measure of similarity of the targets among data in the same subgroup and minimizing the similarity between them in different subgroups. Therefore, it is necessary to select one splitting method able to choose the attribute on which the splitting is based on and to determine the value of the selected attribute, that is the threshold for the split. In Classification there are different methods to perform the splits, like *information gain* or *information gain ratio*, that are based on *entropy*, or the *Gini index*, that always produces binary splits. In Regression the *mean squared error* or the *standard deviation* are the common indices of impurity used to determine the split, so the best feature and its best threshold are selected as the ones that minimize the chosen impurity measure. Independently from the split procedure adopted, they are all focused on maximizing the purity among data so that in the same set their target values are as much similar as possible. Clearly it is always possible to produce a Regression Decision Tree that predicts exactly all the training target values by splitting until each leaf is made by a single datum, but this is an evident example of *overfitting*, that will perform very poorly in testing. To overcome this issue, the tree must be **pruned**:

- it is possible to adopt a *prepruning* approach so that the partitioning is stopped if a certain depth of the tree is reached, if data in a group have a standard deviation smaller than a chosen tolerance, or if the number of data in a subgroup is smaller than a certain minimum number;



(a) An example of Decision Tree.



(b) The induced partition on data space.

Figure 6.2: A Decision Tree with *age* and *height* as features and *weight* as target. Each set partitioning the data space is colored as the corresponding leaf node.

- another possibility is a *postpruning* approach, that once the tree is built starts from last splits and removes them if there is no statistical evidence that they increase the performance on the evaluation of the target.

In conclusion, a Decision Tree iteratively splits data into subgroups maximizing the similarity among their target inside each subgroup, paying attention to splits, because too many subgroups lead to overfitting. At the end, leaf nodes of the tree predict the target value associated to each set of the produced partition of the data space, which is

computed as the mean of the target values of the training data in the set, assigning that value to each test datum belonging to that group.

A **Random Forest** Regression algorithm is a *Model Ensemble* method consisting in a large number of unpruned Decision Trees with a random selection of features at each split. The idea is to use many weak learners as much uncorrelated as possible, that together form a strong learner.
The correlation between Decision Trees (hence the variance of the model) is reduced in two ways:

- each Decision Tree in the forest has training set composed by the same number of data N of the original training set extracted with **bootstrap** technique, which consists in sampling with replacement N data from the original set;
- at each split of a node, the feature on which the split is based on can be selected only between m variables, randomly selected from the d available features.

Summing up, all Decision Trees of the Random Forest are trained in this way: the training set is extracted with bootstrap, at each split only some features can be selected and the trees are unpruned. Each one of them is a weak regressor, since its training is not optimized and it will probably overfit, but they all together become a strong learner, since their learning procedure, which singularly is not optimal, reduces the correlation among trees, decreasing the variance. Moreover, since there is no pruning and at each split a smaller number of features is available, Decision Trees in the Random Forest are much faster to train. Finally, the prediction of the target of a test datum is computed in the Random Forest Regression as the average value of the predictions of each Decision Tree as shown in Figure 6.3.

A possible variant is called **Extremely Randomized Trees** [17], where randomness is exploited a bit more. In particular, a third random procedure is performed to decrease the variance: as in Random Forests, a random subset of m features is available at each node to perform the split, but instead of looking for the best threshold, some values are randomly chosen for each available feature and the best of these randomly-generated thresholds is used to perform the split.

Random Forests also present two peculiar characteristics:

- The first is that it is possible to compute a validation error directly through the training set. Exploiting the fact that not all data appear in all datasets, the **Out of Bag** approach estimates the value of the target averaging only the output of trees corresponding to bootstrap samples where does not appear the selected

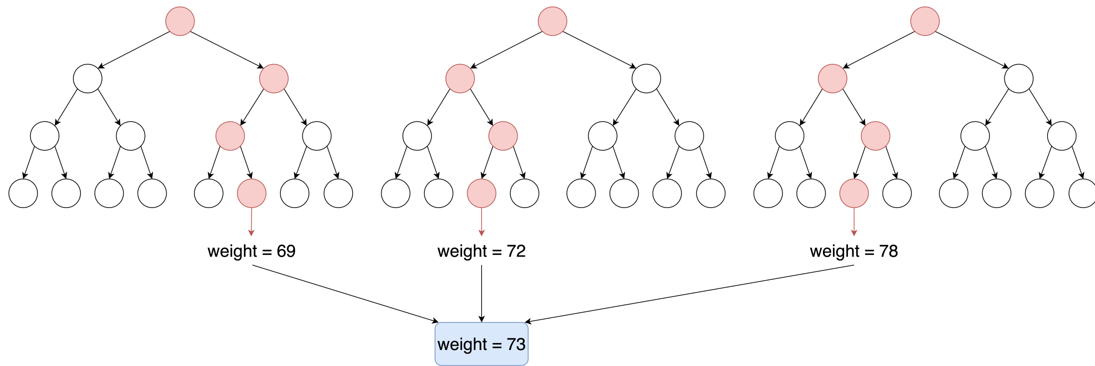


Figure 6.3: An example of Random Forest with $n = 3$ Decision Trees.

datum. Repeating this for all data in the training set allows to measure the performance of the learner in a more robust way than training error without the use of a Cross-Validation set.

- Another important property of Random Forests, that makes them widely applied to perform Feature Selection, is that they produce an estimate of the importance of the features in predicting the target. Indeed Decision Tree methods calculate their splits by mathematically determining which split will most effectively distinguish groups of data with similar target. Therefore in Random Forests the importance of a feature is measured as the sum of the improvements (in terms of *Gini index, information gain* or *mean squared error*) in any node in which the feature is used to split, weighted by the percentage of training data reaching that node.

In conclusion, to train a Random Forest is necessary to set the *number of trees*, to decide the *percentage of features available at each split* and the *minimum number of data required in a node to be splittable*. These parameters can be tuned using a Cross-Validation approach as described in Subsection 6.3.1. They are a relatively small number of hyperparameters and they are enough to be able to train the Regression model and to extract the feature importances, as done in Subsection 6.3.2.

6.3 Feature Selection with Random Forests

In this section the first Supervised Learning approach is performed: it is a Random Forest Regression implemented to find the importance of features on the prediction of the reward. The resulting relevant features will be used to design the MDPs on which RL algorithms are performed. Moreover, the tuning of the Random Forest parameters performed in this section gives information about the features and it is also useful for

tuning the parameters of the Regression method inside the FQI algorithm that will be applied in next chapter.

6.3.1 Random Forest Training

In this subsection all technical details on the Random Forest Regression performed for Feature Selection over the dataset explained in Section 6.1 are reported. In particular the chosen method are *Extremely Randomized Trees*, implemented in the Scikit-Learn [31] tool called *ExtraTreesRegressor* [40].
The best values of the three parameters to tune in modeling a Random Forest (or its variant) are obtained optimizing over a grid of reasonable values.

- The **number of trees** ('n_estimators') should be as large as possible. However, the following set of values is tested to empirically confirm this idea:

  $$n_estimators \in \{50, 100, 200, 500, 1000, 2000, 3000, 4000, 5000\}. \tag{6.2}$$

  The default number of trees in the Scikit-learn tool is set to 100, but since the number of data available in the dataset is not very large it is possible to have more trees without slowing down too much the algorithm. On the other hand, the maximum selected number of trees is 5000, because the more the number of trees increases the slower and more complex becomes the Regression.

- The **percentage of features** ('max_features') to randomly consider when splitting is default set to 1, meaning that all features are considered. As already explained, considering a smaller percentage reduces the correlation between the Decision Trees of the Random Forest, therefore also smaller percentages are tried:

  $$max_features \in \{0.5, 0.6, 0.7, 0.8, 0.9, 1\}. \tag{6.3}$$

  Percentages under the 50% are not considered, so that a consistent number of features is still available at any node.

- Finally, the **minimum number of samples** ('min_samples_split') necessary to split a node is by default set equal to 2, since Random Forests are made of unpruned Decision Trees. However, different splits are tried, because a larger value speeds up the algorithm reducing meanwhile the overfitting of each tree. In particular:

  $$min_samples_split \in \{2, 5, 10, 15, 20, 50, 100, 200, 400, 800, 1000, 2000, 5000, 10000\}. \tag{6.4}$$

  The total number of training data in each iteration of the Cross-Validation is about 20000, hence the maximum value of the minimum data necessary for splits is set

to 10000, about half of the training set. In this way a completely unpruned tree, an extremely pruned tree and many pruning in-between these two extremes are considered, giving to the Cross-Validation procedure a consistent variety of values among which choose the best performing one.

To find the best parameters a **5-Folds Cross-Validation Grid Search** is performed. Namely, on every combination of the possible values considered for the three parameters, a 5-Folds Cross-Validation is performed on data from 2009-02-02 to 2018-12-31, not using the data of 2019, that are exclusively used for testing. In practice, at first iteration data from 2011 to 2018 are used for training and data from 2009 to 2010 are used for measuring the performance; then data from 2009 to 2010 and from 2013 to 2018 are used for training and data from 2011 to 2012 are used for computing the performance. This is repeated for all the data, producing five performance scores of a Random Forest Regression with a certain combination of parameters, that are averaged to compute the cross-validation performance of the forest with that parameters combination. This computation is iterated for all the combinations of parameters in the grid and the combination leading to highest score is selected.
Although the Random Forest is performed for Regression, the measure selected for evaluating the performance is not the usual *R-Squared* but the **Accuracy**, that is the percentage of data in which the predicted reward has the same sign of the real one. This choice is logical from the Reinforcement Learning point of view: if the predicted reward has the same sign of the real reward, the agent will probably correctly choose the action producing that reward if it is positive or it will not perform that action if the resulting reward is negative.

The parameters leading to the best performance score among all the possible combinations are reported in Table 6.1 and they are chosen to perform the Feature Selection. From the best parameters it is possible to confirm that the greatest available number of trees is selected by the grid search, confirming the fact that the higher is the number of trees the more efficient is the learning performance.
Moreover, it is already possible to guess the **poor significance of the features on predicting the reward**. Indeed the best percentage of features is the smallest available and, most importantly, the minimum number of data needed for a split is chosen as large as possible. This means that the selected combination of parameters leads basically to a single split: it is best to only see randomly the 50% of features and when the number of data in a node is half the dimension of the dataset the splitting is stopped, which can happen after only one split if it halves the data. Therefore the Random Forest trained with the selected parameters is made of many trees very much pruned, that basically split the data once and stop without exploring particular patterns among features to do that, and it works better than a forest exploring complex patterns among features

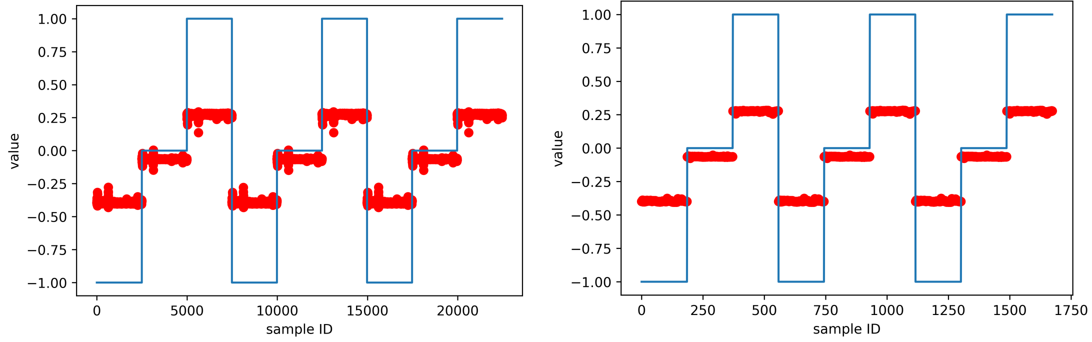
Table 6.1: Parameters tuning: the best parameters and the related Accuracy score.

Parameter	Value
'n_estimators'	5000
'max_features'	0.5
'min_samples_split'	10000
CV-Accuracy	0.59420
CV-Standard Deviation	0.01190

to produce accurate subgroups, meaning that there are no complex efficient patterns among features able to predict the reward, but it is better to do few random splits.
Finally, the produced Cross-Validation Accuracy score is satisfactory, because it is significantly different from the random guess (50%), it is robust since the standard deviation between the scores produced by the five different validation sets is low and it is also greater than the average score produced by this kind of Regression focused on predicting the reward, that practitioners consider it to be about the 55%.

6.3.2 Random Forest Results

After the tuning of hyper-parameters performed in previous section it is possible to train the Random Forest Regression with the best parameters on all data from 2009-02-02 to 2018-12-31. Then, the data from 2019-01-01 to 2019-09-26 are used as test set, producing the Accuracy test scores reported in Table 6.2, that is satisfactory and consistent with respect to the performance obtained in Cross-Validation. Its value is also greater than the training performance, which is not a usual behavior but its reason will be clear by the end of this section. Moreover, train and test confusion matrices are reported in Figure 6.5.
Then, as explained in Section 6.2, it is possible to extract the importance of each feature directly from the Random Forest. In particular, the percentage importance of the best fifteen features together with the related standard deviation is shown in Figure 6.6 and this percentage importance is reported in Table 6.2. From these results it is possible to conclude that the action has the most relevant part on the prediction performed by the Random Forest. The percentage differences of prices are slightly important while the sentiment and all the other features extracted from the date are not relevant. Remembering that the value of the index is mostly increasing (as shown in Figure 5.2), the Random Forest predicts a positive reward when the action is 1, a negative reward when it is -1 and a slightly negative reward when the action is 0. Indeed, the performed Random Forest splits the data only depending on the action and computes the value of the output as the mean reward of the group of samples having that action as feature.



(a) Train action and predicted reward. (b) Test action and predicted reward.

Figure 6.4: The blue line in the figures is the value of the action in each sample, respectively belonging to train and test set. The red points are the corresponding predicted rewards. From their values is clear that they are positive when the action is 1, negative when it is -1 and slightly below 0 when the action is 0.

This pattern is clearly shown in Figure 6.4. Moreover, this is confirmed by the confusion matrix: the signs of the majority of positive and negative rewards are correctly classified, while for almost all null rewards (when both portfolio and action are 0) the predictor considers them negative, leading to the misclassification of almost all of them. All these evidences are in accordance and they show that the predictor is not able to sufficiently exploit the historical series of previous days, the sentiment signal or any other extracted feature to retrieve a more complex pattern predicting the reward, but it only refers to the action assuming the value of the index to always be increasing. This explains the best parameters values, the low importance of the features and the good performance of the Random Forest in terms of Accuracy. Indeed the algorithm only focuses on the action, therefore a more complex structure of the trees is not necessary, leading to a number of samples in a node to be half the dataset. Moreover, the other features do not add informativity to the prediction and when the action is not available to split the historical series are preferred to the sentiment. Finally, the good performance score depends on the fact that the trivial pattern followed in the Regression leads many times to the prediction of the correct sign and the performance in testing is a little better than in training only because the percentage of days when the index is actually increasing is slightly greater.

In conclusion, the Random Forest Regression performed for predicting the reward focuses almost completely on the action, it has a good performance but this is due to the clear increasing pattern of the index and it is a first evidence that the features are not much relevant in predicting the value of the index. In particular, there is evidence

to discard the extracted features representing the day, the month or the distance from coupon detachment and in the MDPs modeled in the following chapter they will not be considered.

		Prediction		
		-1	0	1
True Value	-1	9197	0	3358
	0	2496	2	0
	1	3280	0	4131

(a) The train confusion matrix.

		Prediction		
		-1	0	1
True Value	-1	704	0	229
	0	186	0	0
	1	226	0	329

(b) The test confusion matrix.

Figure 6.5: Confusion matrices show the number of samples whose sign has been correctly predicted and the number of misclassified samples.

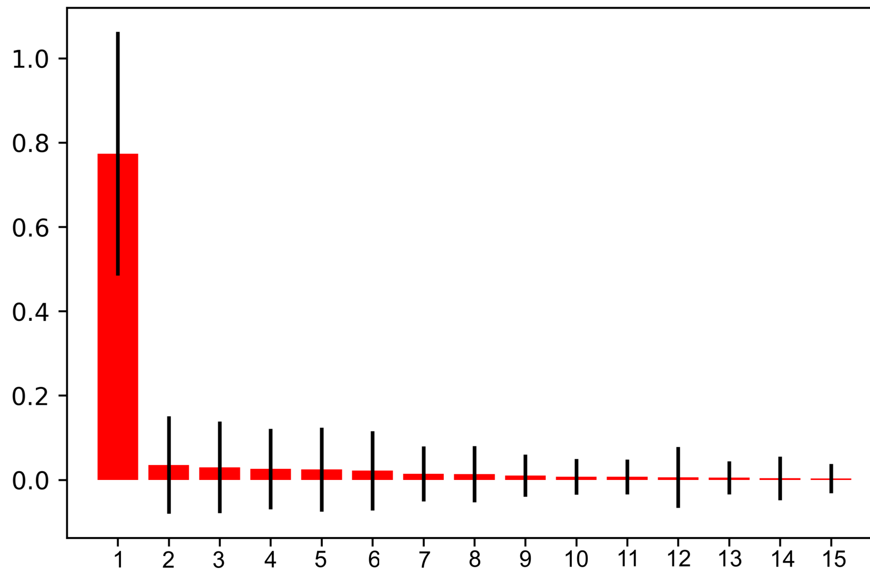


Figure 6.6: Histogram of importance of best 15 features (as ranked in Table 6.2) with standard deviations.

Table 6.2: Feature ranking and performance of the Random Forest.

Feature	Importance
action	77.40%
R1	3.52%
R2	2.97%
R7	2.59%
R0	2.44%
R8	2.18%
R4	1.44%
R6	1.36%
R3	0.99%
R5	0.75%
R9	0.71%
portfolio	0.59%
sentiment_d01	0.49%
PreviousPayment	0.36%
sentiment_d02	0.32%
Train Accuracy (2009-2018)	0.59420
Test Accuracy (2019)	0.61708

6.4 Regression of Sentiment on Residuals

Similarly to Feature Selection, the purpose of the second Supervised Learning approach explained in this section is to investigate the significance of the features. In particular, this procedure focuses on the importance of the sentiment features in predicting the reward. In Subsection 6.4.1 the procedure followed by this approach is described, that is later applied in Subsection 6.4.2.

6.4.1 The Procedure

The chosen Supervised Learning algorithm is again Random Forest Regression. As explained in the discussion on the results of Feature Selection in Subsection 6.3.2, the Random Forest is designed and trained for predicting the reward based on all available features and has a good Accuracy score, but this is almost only depending on the increasing trend of the *S&P 500* index and exclusively the action feature is considered when choosing the sign of the output. Therefore this approach tries to understand the significance of the sentiment features after filtering the Regression from this most rele-

vant pattern between action and reward.
In practice, the procedure consists in training a Random Forest as done in Section 6.3, without using the two sentiment features. Then, another Random Forest is optimized and performed: the features of this second Random Forest are only the two sentiment features of the two days before, while the target are the residuals of the first Random Forest, that are the differences between real and predicted values of the target.
In this way, the first Random Forest regression predicts the value of the reward based on all the features except the sentiment. Then, in the second Random Forest, the sentiment is used to predict the residual of the first Regression so that, if this second Random Forest has a positive performance score, the sentiment is significant in explaining something more about the reward than what already explained by all the other features in the first Random Forest. Hence, a good performance score of the second Random Forest is an evidence of the informativity of the sentiment features, while a bad performance means that the informativity of the reward is not more than all the other features.
The performance score considered for the first Random Forest is the *Accuracy*, in the same fashion of what done in Feature Selection, because the target is again the reward, so the prediction is accurate if the predicted sign is the same of the reward itself. In the second Random Forest, on the other hand, the target are the residuals, real numbers representing the prediction errors, therefore their importance is on their value and not on their sign, which leads to the choice of the *R-Squared* as performance measure, that is the classical score for Regression.

6.4.2 Training and Results

The first Random Forest is performed exactly like the one described in Subsection 6.3.1, with the same best parameters found in that subsection and the same training set. This is reasonable since all the features and the technique are exactly the same except the removal of the sentiment features, that are shown in Subsection 6.3.2 not to be relevant for the complete Random Forest, so there is no need to perform a new parameters tuning.
The parameters of the Regression and the results are reported in Table 6.3. As expected, the Accuracy scores both in training and testing are almost equal to those found in Feature Selection, because as said the only difference in the Random Forest is the absence of sentiment features, that are proved not to be important for this Regression.

In the second Random Forest parameters need to be tuned. The approach adopted for parameters tuning is again a 5-Folds Cross-Validation using training data from 2009 to 2018. As said, the features are the two sentiment signals of the two days before, while the target is computed subtracting the predicted value of the reward produced by the

Table 6.3: First Random Forest: same best parameters of Feature Selection and similar performance.

Parameter	Value
‘n_estimators’	5000
‘max_features’	0.5
‘min_samples_split’	10000
Train Accuracy (2009-2018)	0.59412
Test Accuracy (2019)	0.61707

Table 6.4: Second Random Forest: best parameters and related performance.

Parameter	Value
‘n_estimators’	5000
‘max_features’	1
‘min_samples_split’	10000
CV-R^2	-2.6×10^{-7}
Train R^2 (2009-2018)	-5.9×10^{-8}
Test R^2 (2019)	-7.4×10^{-7}

first Random Forest to its real value. Since there are only two features it is better to let them available at any split, so there is no need to tune the parameter ‘max_features’, representing the percentage of features randomly available at each split. Moreover, the number of trees of the forest should be as big as possible, as theoretically said and empirically shown in the first parameters tuning, so it is set equal to 5000, like in the first Random Forest. Therefore, the only parameter that should be tuned is the minimum number of data needed in a node to perform a split.
As reported in Table 6.4, the best parameter found is again equal to 10000, the maximum possible value in the grid. However, as possible to deduce from the results reported in the table, the *R-squared* values obtained both in training and testing are negative, meaning that the prediction through this second Random Forest is worse than assigning to each predicted value the mean of train target (i.e. the train residuals). In practice, the results obtained with this second Random Forest underline the fact that **the sentiment features does not add informativity to the interpretation of the reward**, which is a second evidence in accordance to what found in Feature Selection that the features and in particular the sentiment features are not relevant in predicting the Market trend represented by the *S&P 500* index.

In conclusion, in this chapter Feature Selection over the available features brought to the decision that for RL algorithms it is enough to consider historical series and sentiment features, since there is no statistical evidence about the importance of the extracted features in predicting the reward. Moreover, the main issue on daily data has emerged: the clearly increasing trend of the *S&P 500* index overrides any other relation between the index and the features. The Supervised Learning approaches presented in this chapter also produce two evidences about the low importance of the sentiment features in predicting the reward, since they are not considered relevant features in the Feature Selection and they are not meaningful on predicting the residuals. The Reuters sentiment features are used anyway in Reinforcement Learning algorithms of next chapter, to perform an exhaustive analysis of their behavior.

Chapter 7

RL Models and Results

In this chapter three Reinforcement Learning environments modeled as Markov Decision Processes (introduced in Section 2.2) are explained. The features considered in the models are chosen starting from the complete dataset shown in Section 5.3 accordingly to the results of Feature Selection (performed in Section 6.3). They are three different models because each one considers different features, as justified in detail in Section 7.1. After the explanation of the models, in Section 7.2 the three algorithms applied to each model, already introduced in Chapter 3, are recalled and discussed specifically for the models available. Moreover, in Section 7.2, technical details about training, validation and testing procedures are addressed for each algorithm, extensively explaining all the passages performed in the learning procedures. Then, in Section 7.3, the results of the application of the three considered algorithms on the three designed MDPs are shown. In particular the performance in train, validation and testing is illustrated, together with some variability measures computed to make statistically robust the conclusions over the obtained results.
Finally, in Section 7.4, all the procedures applied to daily data are repeated on fifteen minutes data.

Models and algorithms designed in this chapter have been implemented using *Python3.7.2* [43] as programming language. In particular, the algorithms have been performed through a customized version of *OpenAI Baselines* [15], which is a set of implementations of RL algorithms and models. Specifically, the models have been implemented as classes with methods required by Baselines algorithms. These algorithms already include PPO and TRPO, while FQI is not present in Baselines but it has been implemented in a similar fashion. Moreover, some modifications have been performed on the algorithms already present in Baselines, as a customized training and testing procedure.

7.1 MDP Models

In the discussion on Reinforcement Learning (Chapter 2), the choice to model problems as Markov Decision Processes has been justified. They have also been defined (Definition 2.2.3) as a tuple of six elements, that are described in this section specifically for the problem addressed in this thesis.
In particular, as done with Supervised Learning approaches in Chapter 6, the aim of this work is to investigate the importance of sentiment features in predicting the Market trend represented by the *S&P 500* index, compared with the capability to learn this trend through its historical series. Therefore three different MDPs have been designed to perform this analysis. These three environments are the same MDP except for the *state*, which is different in each model. Specifically, the three processes are described through the definition of their elements.

- The **state** is the only element which is different in the three models. Indeed in a fully observable process the state is equal to the observation, which is different in the three models to compare the performance of the features. In particular:

    1. in the first MDP the state is made of eleven variables, that are the ten percentage differences of open values of ten previous days and the current portfolio (which corresponds to the action performed the day before);
    2. in the second MDP the state is made of three variables, which are the two daily Reuters sentiment features of the two previous days and the current portfolio;
    3. in the third MDP the state is made by all the thirteen variables of the previous processes, i.e. the ten percentage differences of open values, the two Reuters sentiment signals and the portfolio.

  This choice is reasonable since RL algorithms applied in the first model learn the optimal policy knowing only features related to historical series of the index, in the second case they learn knowing the sentiment features and in the last one they have at disposal both historical series and sentiment signals. In this way it is possible to compare the performance of the three models, concluding if the sentiment features are more informative than historical series or not and if they all together lead to a better result than singularly. Finally, the portfolio is available in any model so that the agent can evaluate the cost of changing its position in terms of transaction costs.
  It is also important to notice that the elements defining the state in the three models are all continuous values except the portfolio, so in each MDP the set of all states S is uncountable infinite.

- The set of **actions** A is made of three actions: $\{-1, 0, 1\}$, which correspond to sell, do nothing or buy. A complication of the models could be to allow the action to be a continuous value in $[0, 1]$, so that it would represent the percentage of stocks that should be sold or bought (with respect to all the available stocks, i.e. the number of stocks that have value equal to all the money available to invest). The modeling choice made in this work is to consider only the extreme values, meaning that all the amount of the available money is invested. Hence the value of the action could be figured as buy or sell the 100% of the available stocks, or to invest on $1 of stocks, without loss of generality since it is always possible to scale it by any different amount of money.
  Considering the three actions, if the agent predicts with a sufficient confidence that the value of the index will increase it performs action 1, which means it buys stocks with the idea to earn selling them later when they have an increased value. If it predicts that the value of the index will decrease it performs action -1, so that it sells stocks borrowed and not owned with the idea of buying them in the future when they will have a decreased value. Finally, if the agent is not sufficiently confident about what will happen in the future given the observation or it predicts a percentage change of the index value smaller than the fee it has to pay for changing its position, it should perform action 0, which implies to close the position and not invest on the Stock Market that day.

- The conditional **transition probability** $\mathbb{P}(s'|s, a)$ is the probability of moving to state s' given the current state s and the action a performed on it. The only endogenous variable of the state is the portfolio, since it depends on the action, but it is also deterministic given the action. Therefore, the probability to be in state s' is 0 if the value of the portfolio in s' is different from the given action a performed on s. On the other hand, the Reuters sentiment signals and the value of the *S&P 500* index are exogenous variables [11], since the action does not affect their value. Indeed the value of the Market and the sentiment of Reuters news depend on external factors, that the action of the agent can not influence. These variables are stochastic and they determine the transition probability from a state s to s', that is a non-zero unknown probability density in any state s' with portfolio equal to the action a performed on s. In conclusion the transition probability is 0 when the portfolio of state s' does not coincide with the performed action a, otherwise it is a positive probability density depending on many external factors. This makes not possible to properly model the transition probabilities, justifying the choice of performing Reinforcement Learning approaches, since they are applicable on MDPs with unknown transition probabilities.

- The **reward function** $R(s, a) = \mathbb{E}[r|s, a]$, computes the expected value of the

reward given the state s and the action a performed in it. The reward has already been largely discussed in Section 6.1, since it is used as target for Supervised Learning algorithms. As defined in that section, the reward at time t is:

$$r = a_t \cdot \frac{(open_{t+1} - open_t)}{open_t} - |a_t - a_{t-1}| \cdot fees. \tag{7.1}$$

The reward depends on the percentage increase between next and current open values, which is available only at next iteration, making it depending on next state s'. Therefore, the expected value computed by the reward function depends on the transition probabilities $\mathbb{P}(s'|s, a)$, that are unknown since, as already discussed, they depend on external factors. This means that also the reward function is unknown, which is typical in MDPs modeling real problems.

- The **discount factor** γ is set to be equal to 1, making the problems *undiscounted*. This choice is reasonable since there are no convergence issues on the reward, because in a reasonable amount of time the sum of the percentage increase or decrease of the index does not diverge. Indeed this would mean that the index would have become infinitely greater or smaller than the actual value. Moreover, except in rare cases, the percentage change of the index is smaller than the 1%, meaning that the sum of relative daily growth of the *S&P 500*, which mostly determines the reward, is a relative small number, that does not need to be discounted. Also practically, the discount can be interpreted as an index of how much an immediate reward should be preferred by the agent with respect to moving to a state that may lead to a better reward in the future. Since in the models presented in this work the action does not influence the next state (except for the portfolio, but it has a small impact on the daily reward), there is no need to compare long and short term reward, giving the same weight to all rewards through selecting a discount factor equal to 1.
- Finally, the **initial probability** distribution μ can be considered deterministic, since the initial state is univocally determined by the values of its variables, that are known given the date of the day that starts an episode of the MDP.

Summing up, an iteration of one of the three MDPs described in this section consists in the following procedure. In 'day 0', the agent observes the percentage differences of the open values of the index in the ten days before and/or the value of the Reuters sentiment features of the two days before. Then it performs an action: to buy if it expects an increase of the value index, to sell if it expects a decrease of it and it does nothing if its expectation is that the index will be almost constant. Finally, the next day starts next iteration, the agent gets a reward depending on what really happened and on transaction costs and it has to decide the new action to perform.

The three models described in this section have been implemented as classes on which it is possible to generate *trajectories* and apply different Reinforcement Learning algorithms. The choice of algorithms and their implementation are discussed in next section.

7.2 Algorithms

After the definition of the models in previous section, the next step is to describe the algorithms performed and the procedure followed to achieve the results described in Section 7.3. In particular, three algorithms have been performed on each of the three MDPs introduced in Section 7.1.
The first two algorithms are **TRPO** and **PPO**, two Policy Search methods already discussed in Subsections 3.3.2 and 3.3.3. In the state of the art, they are two of the most efficient Policy-Based methods, since they have the advantage with respect to Policy Gradient methods to optimize directly the policy, forcing two consecutive policies to be sufficiently similar, respectively as a constrained optimization problem or directly inside the loss function.
The other macro-category of algorithms are Value-Based algorithms, that focus on reaching the optimal value function and to derive an optimal policy from it. Among all available Value-Based algorithms, **FQI** has been chosen to be performed. It is an offline algorithm which exploits a Regression algorithm to update the value function as discussed in Subsection 3.2.4. Therefore it is a natural consequence to choose FQI in this framework, where in Chapter 6 a Random Forest Regression has already been performed using the reward as target, generating the extended dataset and having the parameters already tuned. Moreover, as already discussed, FQI algorithm is able to predict the action-value functions in state-action pairs not directly explored in the generated trajectories, predicting their value through the prediction of the Regression algorithm. This is not possible with SARSA or Q-Learning algorithms, and it is necessary in the problems of this work, since the set of all possible states S is uncountable infinite, making not possible to investigate all states directly generating episodes.

7.2.1 The Procedure

To perform the three selected algorithms, producing more consistent results, and to optimize their hyperparameters, a *Cross-Validation* approach has been chosen. As done in previous approaches, data from *2009-02-02* to *2018-12-31* are used as training data and data of 2019 are used only for testing purposes. In particular, for every year from 2009 to 2018, each algorithm has been trained on the other nine years and tested on the selected year. This is not a standard approach, since theoretically, when the training set is made of historical series, training data should be prior to validation data that

should be in turn prior to test data. Indeed when, for example, 2009 data are used for validation and data from 2010 to 2018 for training, the algorithm is trained on future data with respect to the ones it predicts, which is not very realistic. However, following the chronological sequentiality of data, results would have been based only on the performance on 2019, making the conclusions not much robust, since the available sample size is not sufficiently large. Therefore, a Cross-Validation approach has been adopted to train and validate data from 2009 to 2018, knowing that the most reliable result is the final testing, where all data from 2009 to 2018 are used for training and the available data of 2019 are used for testing.
In addition, in order to take into account the stochasticity inside the optimization of the algorithms, making the conclusions on the results more consistent, each training of the algorithms is repeated five times. In this way, five optimal policies trained on the same model using the same data are available, allowing to compute confidence intervals of the performance of the learned policies on validation or test data, taking into account the stochasticity of the learning process.

7.2.2 Trajectories and Parameters

Another issue to solve is how to sample trajectories starting from the models. Indeed, an episode is theoretically infinite: every day, the agent performs its action, changing or keeping its position, and in principle this can continue forever. However, the trajectories that are possible to sample are only the working days available in the training set. Therefore, the choice is to sample trajectories of 60 days, where the first 10 days are used to generate the state, since some of its variables are the ten percentage differences of prices of the ten previous days, then in 49 days it is possible to perform actions and get the related reward, while the last day is only used to generate the last reward. In conclusion, each trajectory sampled from the MDPs is made of 49 iterations, corresponding to 49 consecutive working days of performed actions and rewards depending on them.

The last issue about the algorithms is the parameters tuning. First of all, in PPO and in TRPO the policy is parametrized by default as a fully connected Feedforward Neural Network that takes as input the state, elaborates its signal through two fully connected hidden layers, each made of 64 nodes with *tanh* activation function, and it outputs the action to perform. The other parameter to tune in TRPO and PPO are the number of iterations of the training process of the algorithm and the number of trajectories to sample at each iteration, that are set respectively to be 1000 iterations in PPO, 3000 iterations in TRPO and 2500 trajectories for both. Since the trajectories start from a random day of the training set, the choice of 2500 trajectories is to have in expectation one trajectory starting from each day at each iteration (remembering that

Table 7.1: Parameters selected for the Random Forest Regression in FQI.

Parameter	Value
'n_estimators'	5000
'max_features'	0.5
'min_samples_split'	10000

the training data are approximately 2500). The number of iterations, on the other hand, is empirically set, since from the learning curves of the algorithms reported in Figure 7.1 it is clear that the chosen values are reasonable to suppose the convergence of the training procedures of the algorithms.
In FQI algorithm the policy is not parametrized, since it is derived from the action-value function as described in Equation 2.23. Hence, the only parameters to tune are the choice of the Regression algorithm and its parameters that, for continuity reasons with respect to Chapter 6, is selected to be a Random Forest Regression with the same parameters already tuned in that chapter and reported again in Table 7.1. Indeed, although the Random Forest Regression of FQI performs a different prediction than the one trained for Feature Selection, since it uses the same features but the target is no more the one-step reward but the cumulative reward represented by the action-value function, it is still possible to use the best parameters already found for Supervised Learning approaches as a good approximation of the best parameters for the Random Forest Regression inside any iteration of the FQI algorithm. Therefore, the only parameter to tune is the number of iteration of the algorithm for each model, that is selected observing the performance reported in next section on the validation set of the algorithm trained with a number of iteration from 1 to 10 that basically, as explained in the description of the algorithm, means that the action value function is optimized with respect to the cumulative reward from 1 to 10 next days.

7.2.3 Performance Measures

The most intuitive performance measure considered is the average **daily reward**, which is an estimate of the expected reward in a day. Another similar performance measure is the cumulative reward computed in a trajectory, which can be considered as a sort of **return**, assuming that an episode lasts 50 days. This assumption is not restrictive: it implies, with respect to assuming an episode to last forever, that every 50 days the position on the Market of the agent is closed independently from the action it would choose.
In order to make a more robust analysis, also some measures of variability are taken into account. The most common procedures used for measuring the variability are the

variance and the **standard deviation**, both of the reward and the return. Once they are estimated through samples, it is possible to compute confidence intervals of the two performance measures.
Finally, an estimate of the Expected Shortfall (CVaR) [2] is computed as the mean of the 5% worst rewards, together with its standard deviation. This scores allow to evaluate a confidence interval on the expected value of the worst obtained rewards. This is important because the agent is trading stocks using money and, since any Financial Market participant is *risk-averse*, it is important not only to have an overall mean positive reward in a year but also not to loose too much money in days when the performance is poor, otherwise the model is not sufficiently stable to be applied in real world contexts, since after a couple of huge losses people would stop to trust on the efficiency of the agent.

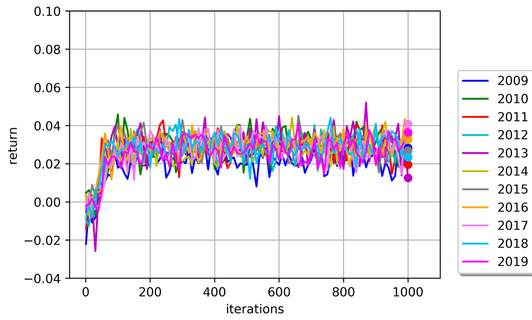
In conclusion, TRPO, PPO and FQI are the algorithms performed on the MDPs modeled in this work. The only parameter tuned using the validation shown in next section is the number of iterations of FQI algorithm, while the other parameters have been naturally deduced by the nature of the problem and by the analysis already performed. Finally, not only the mean return is evaluated as performance score, but also an accurate analysis on scores based on the reward is performed, in order to have a more extensive analysis that also considers the worst cases.

7.3 Results

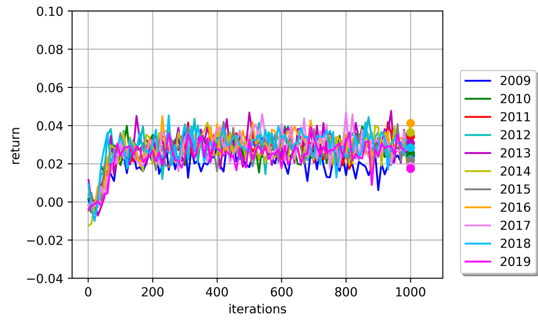
In this section the results of the application of the procedure described in Section 7.2 are shown. The performance of each of the three selected algorithms performed on the three MDPs defined in Section 7.1 and repeated five times is provided. In particular, the training procedure, the parameters tuning, a baseline score, the validation and testing mean and standard deviation scores and a final comparison of performances are presented.

7.3.1 Training

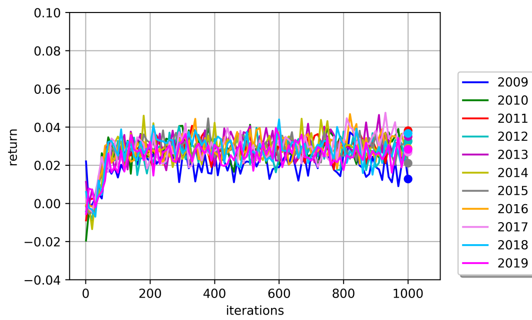
The first step is to train each algorithm on each available model. In Figure 7.1, the curves displaying the performance through the learning process of PPO and TRPO algorithms are reported: for each iteration, the corresponding average training return computed on trajectories of 49 days is plotted. The figures have been drawn using a *TensorFlow* [1] tool called *TensorBoard*, designed to provide visualization during Machine Learning trainings, which is useful to understand the learning process. The training process of the two algorithms is made of 1000 and 3000 iterations respectively for PPO and TRPO. From the curves it is indeed possible to observe that in the end they keep almost the same value for many iterations, so there is no need to further train the algorithms.



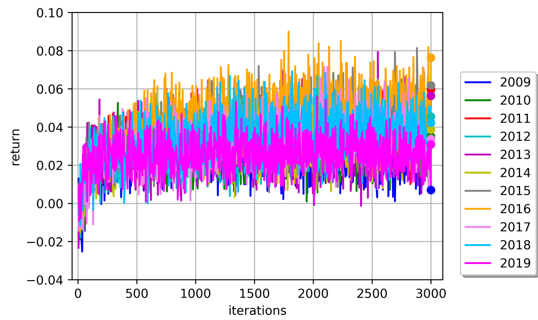
(a) Training curves of PPO algorithm with only sentiment features in the state.



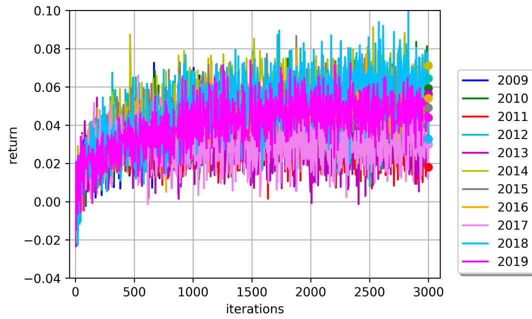
(b) Training curves of PPO algorithm with only Market features in the state.



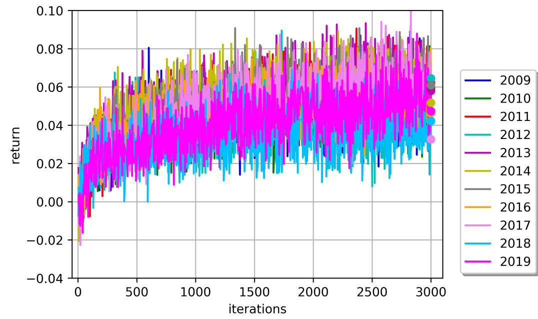
(c) Training curves of PPO algorithm with sentiment and Market features in the state.



(d) Training curves of TRPO algorithm with only sentiment features in the state.



(e) Training curves of TRPO algorithm with only Market features in the state.



(f) Training curves of TRPO algorithm with sentiment and Market features in the state.

Figure 7.1: Training curves of PPO and TRPO algorithms. Each point of a curve is the mean train return over 49 days: a value of 0.03 corresponds to a mean gain of the 3%. The data related to the year reported in the legend of each plot are the test set referring to that curve, while the data of other years from 2009 to 2018 are its training set.

Recalling that, for every year from 2009 to 2018 used as validation set and for 2019 data that are the test set, five repetitions of the training (that has as training set the other available years) are performed to take into account the stochasticity inside the algorithms, in each of the three considered MDPs and for each of the three selected algorithm, eleven different trainings repeated five times for a total amount of **fifty-five** trained models is computed. In each plot of Figure 7.1, only one training curve among the five repetitions of the same algorithm learning on the same training set are reported to make the plots clearer, since the four curves not reported are very similar to the one that is shown.
From the curves in Figures 7.1a, 7.1b and 7.1c it is possible to conclude that the PPO training performance is the same, independently from the features considered as variables of the state. The same conclusion can be drawn from Figures 7.1d, 7.1e and 7.1f concerning TRPO algorithm. Moreover, comparing the training curves of PPO and TRPO algorithms, TRPO seems to perform better than PPO, achieving higher return values in the curves, but it is also much more noisy, so it is necessary a deeper statistical analysis which takes into account the variability of the trained models to properly compare the performances.

It remains to explore the training of FQI algorithm. As already explained, the Regression algorithm chosen for FQI are Random Forests, with parameters reported in Table 7.1. The only parameter left is the number of iterations to perform, which is selected maximizing the validation performance of the training data from 2009 to 2018, iteratively training on nine years, testing on the year not used for training and repeating the procedure five times to take into account the stochasticity of the algorithm, as already done with PPO and TRPO. In FQI this Cross-Validation approach is not only useful to have more results to compare but it is indeed necessary to decide the number of iterations for the final model. Since the learning is slow and in a trajectory there are 49 days, the number of iterations tried is from 1 to 10, which means optimizing the cumulative reward from 1 to 10 future steps. Moreover, as explained in FQI Subsection 3.2.4, the dataset used for the Regression is the extended dataset with the reward as target, that has already been discussed in Section 6.1. The mean daily train and validation rewards averaged over the five repetitions of the same model are computed, so that, for every year, for any of the three MDPs and for all iterations from 1 to 10, one performance score is evaluated. Then, the ten training scores and the ten validation scores are averaged, producing one mean train score and one mean validation score for any model and for each iteration from 1 to 10. As reported in Table 7.2, there is no difference in the performance scores among the ten iterations. Moreover, fixing the training and the validation set there is no difference even among the three different MDPs. This is due to the fact that the algorithm is only learning the trivial pattern to always perform *long*

Table 7.2: FQI daily train and validation average reward for each model and for each validation set. In all iterations from 1 to 10 the value is exactly the same, so only one is reported.

	Sentiment		Prices		Both	
	Train	Validation	Train	Validation	Train	Validation
2009	0.000348	0.001577	0.000348	0.001577	0.000348	0.001577
2010	0.000572	0.000185	0.000572	0.000185	0.000572	0.000185
2011	0.000555	0.000555	0.000555	0.000555	0.000555	0.000555
2012	0.000587	0.000461	0.000587	0.000461	0.000587	0.000461
2013	0.000578	0.000954	0.000578	0.000954	0.000578	0.000954
2014	0.000561	0.000334	0.000561	0.000334	0.000561	0.000334
2015	0.000578	0.000279	0.000578	0.000279	0.000578	0.000279
2016	0.000602	0.000646	0.000602	0.000646	0.000602	0.000646
2017	0.000618	0.000648	0.000618	0.000648	0.000618	0.000648
2018	0.00056	-0.000275	0.00056	-0.000275	0.00056	-0.000275
Average	0.000556	0.000487	.000556	0.000487	0.000556	0.000487

position action (1), independently from the features or from the horizon on which the cumulative reward is optimized, in accordance with the results of the Random Forest Regression performed in Chapter 6. Moreover, the Random Forest performed by the algorithm is composed by many Decision Trees, which provides a noticeable decrease on the stochasticity inside the algorithm, so the action is exactly always the same, without noise that may lead to perform a couple of different actions repeating different times the same training.

7.3.2 Testing

The performance measures described in Subsection 7.2.3 have been applied to the algorithms trained in previous section and they are reported in Appendix A. In particular, in Table A.1 and A.2 the performance measures respectively related to the reward and the return are shown for each model trained with PPO algorithm on the three MDPs. The scores refer to each validation year from 2009 to 2018 and to the final model with all data from 2009 to 2018 as training set and having 2019 data for the test. In Table A.3 and A.4 the same results are reported for the models trained with TRPO algorithm. Finally, in Table A.5 and A.6 they are shown for FQI algorithm.
Recalling that it is also important to take into account the stochasticity inside each algo-

rithm, the training of each model is repeated five times, so that it is possible to compute the average performance score among the five models and a related confidence interval. The average of the five mean rewards obtained repeating the training of the same model five times and the approximated 95% confidence interval are reported in Figures 7.3, 7.4 and 7.5, respectively for models trained with PPO, TRPO and FQI. It is important to notice that, in the plots, the estimated standard deviation is not depending on the variability of the reward due to the different value it assumes in different days (as the standard deviation reported in the tables in appendix) but it is an estimate of the variability of the mean reward depending on different trainings of the same model on same data.
Finally, to compare the results with a *baseline* score, the average daily reward computed performing always the same action (*sell*) is computed for each validation set and for the test set and its values are reported in Figure 7.2.
Remembering that the same validation performance is obtained at any iteration of FQI because the action selected by the agent is to always keep a *long position*, both from the performance measures in Table A.5 and from Figure 7.5 compared with the baseline, it is possible to conclude that models trained with FQI algorithm always perform this action, with absence of variability among different trainings due to the large number of Decision Trees used in the Random Forest inside the algorithm. From Table A.1 and A.3 and from the two Figures 7.3, 7.4 compared with the baseline Figure 7.2, it is possible to observe that also policies optimized with PPO and TRPO algorithms learn to always perform *sell* action, with some exceptions depending on the noise, which are a few in PPO and a more consistent number in TRPO. This explains the training performance, where TRPO seemed to perform better, but this was due to the fact that the algorithm is slower to converge than PPO and it produces more noisy results. Finally, it is also important to notice that the mean of the 5% worst rewards in the tables in Appendix A is a loss never worse than the 3.3%, with a standard deviation never worse than the 0.7%, hence the days the algorithms perform badly bring to an acceptable loss.

In conclusion, all the three algorithms on each validation and on the test set always tend to perform *buy* action independently from the features observed in the state, in accordance with what already shown in the Supervised Learning algorithms performed in Chapter 6. This can be explained through the clearly increasing pattern of the *S&P 500* index but this also leads to the conclusion that **both the historical series and the Reuters sentiment features does not add important information** able to make Machine Learning algorithms capable to predict more complex patterns among them and the reward, so only the clear increasing trend of the index is exploited by the algorithms to optimize their performance.

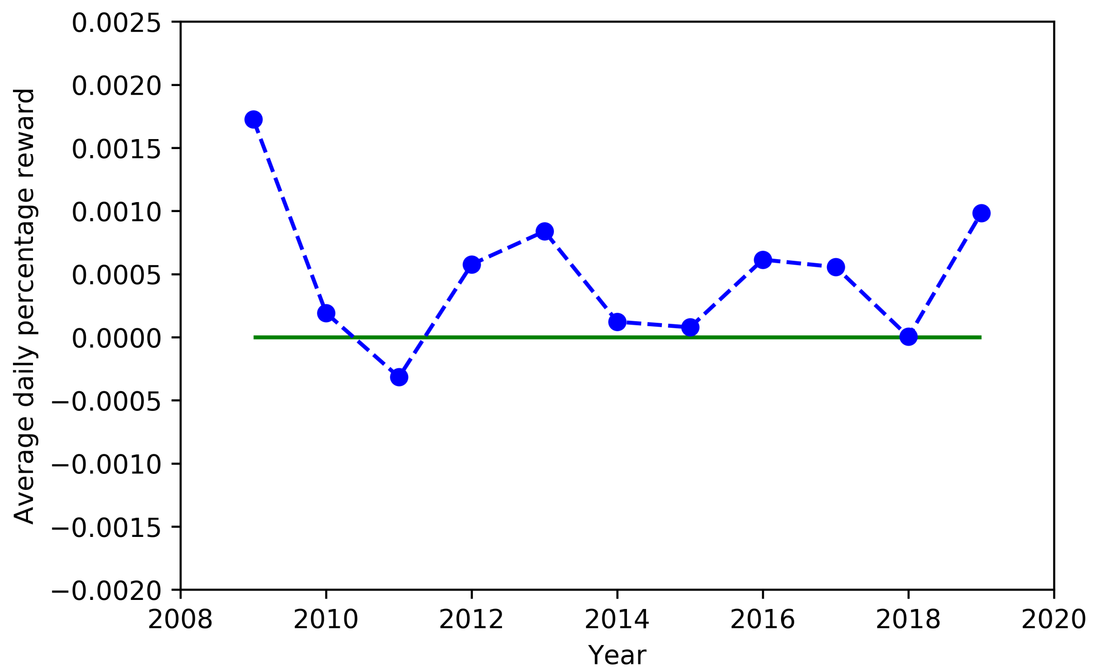
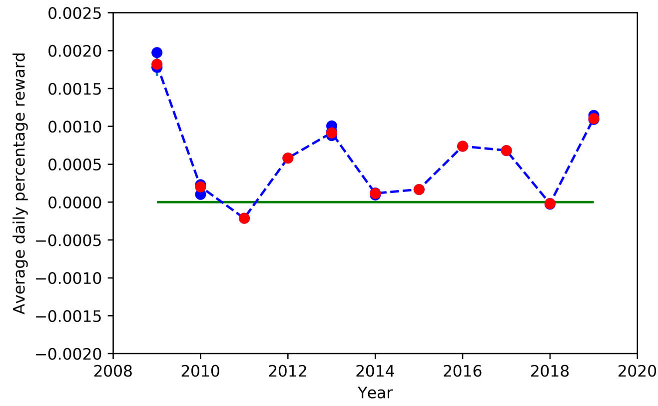
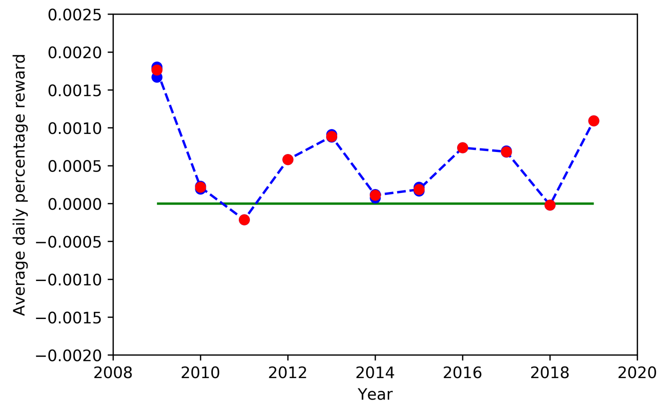


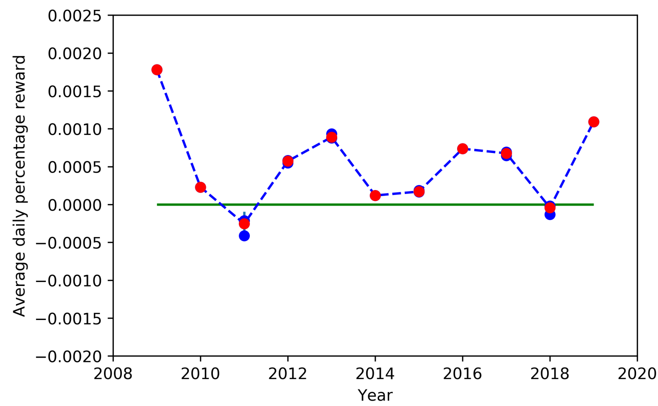
Figure 7.2: Baseline: average daily reward in each year always performing action "1".



(a) Reuters sentiment features.

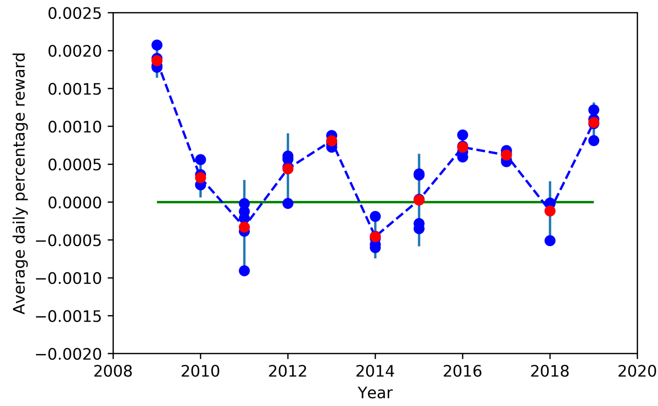


(b) Historical series of the *S&P 500*.

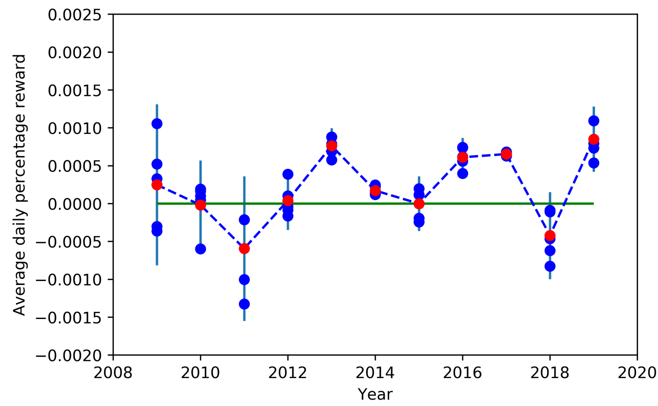


(c) Reuters sentiment and historical series features.

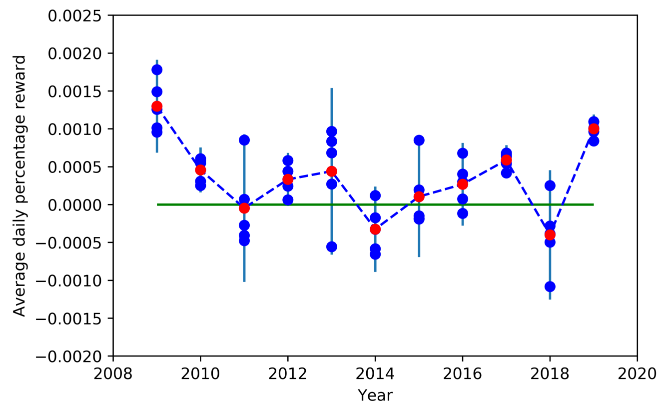
Figure 7.3: Confidence intervals for the mean daily reward computed in 5 different trainings of PPO algorithm for each of the 10 validations and for the final test.



(a) Reuters sentiment features.

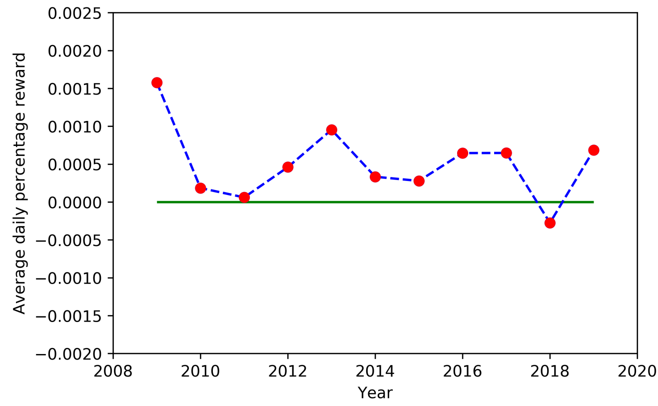


(b) Historical series of the *S&P 500*.

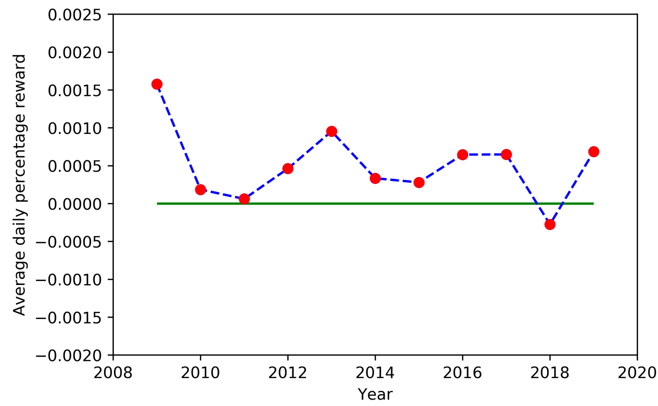


(c) Reuters sentiment and historical series features.

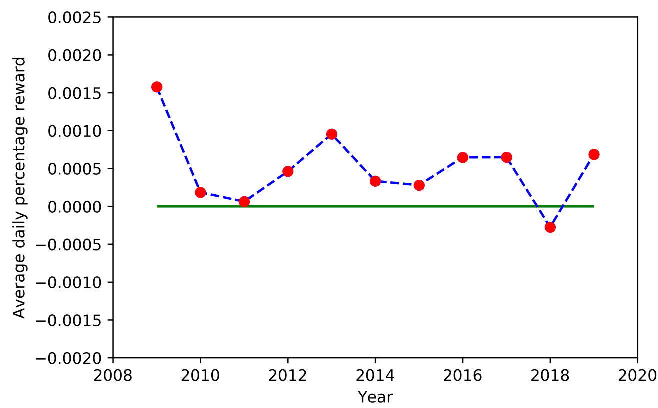
Figure 7.4: Confidence intervals for the mean daily reward computed in 5 different trainings of TRPO algorithm for each of the 10 validations and for the final test.



(a) Reuters sentiment features.



(b) Historical series of the *S&P 500*.



(c) Reuters sentiment and historical series features.

Figure 7.5: Confidence intervals for the mean daily reward computed in 5 different trainings of FQI algorithm for each of the 10 validations and for the final test.

7.4 15 Minutes Data

This last section summarizes the analysis performed applying on a different dataset all the approaches introduced in Chapters 5, 6 and in previous sections of this chapter. In particular, the new available data are sentiment features and historical series of the *S&P 500* index at intervals of fifteen minutes during the opening hours of the U.S. Stock Market.

7.4.1 Datasets

As done in Chapter 5 for daily data, this section introduces and elaborates the available datasets with fifteen minutes features.

The first available fifteen minutes dataset are historical series of *S&P 500* index from *2017-05-02* to *2019-10-09*, for a total amount of 16416 data. They are the 27 records per day, every fifteen minutes from 9.30 a.m. to 4.00 p.m., in the 608 working days of the considered time interval. Features are *open, close, high, low* and *volume* exactly as explained in Section 5.1 for its daily version. Moreover, the *referenceDate* feature includes the date as usual with the addition of the time each datum is referring to.

The other two available datasets are made of sentiment features: the ones extracted from Reuters Machine Readable News and the ones extracted from tweets of Twitter. In particular, it is not possible to simply consider the average sentiment score of last fifteen minutes as a relevant feature, since it is too restrictive to only consider the sentiment of last fifteen minutes ignoring everything that happened before. Therefore, more complex features are extracted from tweets and news applying two procedures, *sentiment trend* and *sentiment shock*, whose description can be found in [41]. In particular, sentiment trend and shocks are computed on a selected *rolling window* using sentiment scores averaged with respect to a chosen *time frame*. For example, if time frame and rolling window are respectively set equal to 1 hour and 12 hours, the sentiment signals used to compute sentiment trends and shocks are the means of the available sentiment scores related to news or tweets available in one hour of the twelve previous hours.
Hence, 168 features are available in each of the two sentiment datasets, which are sentiment trends and shocks obtained combining different time frames and rolling windows:

$$\begin{aligned} time\ frames &= [15M,\ 1H,\ 3H,\ 6H,\ 12H,\ 1D], \\ rolling\ windows &= [1H,\ 3H,\ 6H,\ 12H,\ 1D,\ 3D,\ 7D,\ 15D,\ 30D], \end{aligned}$$

where ‘M’,‘H’ and ‘D’ mean respectively ‘minutes’, ‘hours’ and ‘days’. Moreover, in Reuters dataset, also TF-IDF procedure (further details can be found in [45]) has been

exploited as a variant to evaluate the sentiment in the chosen time frame, so in Reuters dataset the 168 features are repeated two times: one time trends and shocks are evaluated on sentiment signals computed on the chosen time frame with the usual mean of sentiment scores referring to each company belonging to the *S&P 500*, while in the other one sentiment signals of each company are weighted applying a TF-IDF correction which takes into account the importance of the company in the news.
Summing up, two sentiment datasets are available for the analysis of fifteen minutes data from 2017-05-02 to 2019-10-09, one is made of 168 features referring to sentiment from Twitter and the other one is made of 336 features based on sentiment from Reuters news.
Because of the magnitude of the number of sentiment features, a dimensionality reduction technique has been performed. Specifically, *Principal Component Analysis (PCA*, [30]) has been performed on Reuters dataset, keeping the eight principal components explaining at least the 2% of the variability, for a total amount of the 78%. The same has been done for Twitter dataset and such components are again the first eight ones, explaining a total amount of 88% of variability.
In conclusion, the two available sentiment datasets are reduced to have 8 features each, which are respectively the best principal components of the application of PCA on Reuters original dataset and the best principal components resulting on applying PCA to Twitter dataset.

Finally, as done in Section 5.3, some features are extracted from the date: day and month are encoded in one-hot encoding fashion, introducing 17 features, and the *Daytime* feature encodes a continuous value in [0,1] representing the percentage of working hours from the opening of the stock market. Moreover, historical series of the values of the index are modified as done in Section 5.3 for daily data, so that percentage differences of two consecutive open values among the four previous quarters of an hour are the 4 features extracted from historical series of the *S&P 500*.
The complete dataset is therefore made of: 8 principal components of Reuters sentiment, 8 principal components of Twitter sentiment, 4 percentage differences of index opening values, 18 features extracted from date and hour and 1 feature univocally representing each sample containing its date and hour.

7.4.2 Feature Selection

In this section the same procedures explained in Chapter 6 are applied on fifteen minutes data.
Firstly, Random Forest Regression for Feature Selection is performed. It is shown to usually predict a negative reward, which is due to the large importance of the trans-

action costs in fifteen minutes framework. Then, the procedure which considers a first Random Forest Regression without sentiment feature and a second one which tries to learn the residuals exploiting sentiment features confirms the lack of informativity of these sentiment. Finally the two procedures are repeated without considering transaction costs. The aim of this variant is to explore more widely the importance of sentiment features once the strong dependence from the portfolio is removed. The prediction made by Random Forests with this approach still gives little importance to the sentiment. On the other hand it is more interesting since it does not learn a trivial pattern as before, although this is not realistic since transaction costs must be considered in practice.

Expanded Dataset

The first step is to generate the Supervised Learning dataset starting from the available features, recalling that the target of interest is the reward, which is computed as in Equation 6.1. In particular, Twitter and Reuters dataset are considered separately, so that the effectiveness of each of them in predicting the reward can be explored training two different Random Forests. Finally, each of these two datasets for Feature Selection is made as usual of nine repetitions of each sample made of the eight sentiment features, the four index percentage differences and the eighteen extracted features, with different combinations of action and portfolio and the related reward as target, exactly as done in Section 6.1.

Random Forests for Feature Selection

The first Supervised Learning procedure applied on the two described datasets are Random Forests optimizing the *accuracy* of the prediction in order to rank the importance of each feature, as done for daily data in Section 6.3. In particular, one Random Forest Regression is optimized for the dataset with Twitter sentiment principal components and another one is trained for the dataset with Reuters sentiment signals.
Firstly, parameters tuning is performed. The number of Decision Trees in the Random Forest is set to 500 recalling that they should be as many as possible while, on the other hand, a bigger number would slow down the training too much. Then, the percentage of features available at any node and the minimum number of samples needed in a node to perform a split are selected with a 3-Folds Cross-Validation of data from 2017-05-02 to 2018-12-31, choosing among the following alternatives:

$$max_features \in \{0.5, 0.6, 0.7, 0.8, 0.9, 1\}, \tag{7.2}$$

$$min_samples_split \in \{100, 500, 1000, 5000, 10000, 20000, 50000\}. \tag{7.3}$$

Best parameters are reported in Table 7.3. From their value it is possible to deduce, as for daily data, that each Decision Tree in the two Random Forests is splitting only one or two times, using the minimum number of features, therefore it is learning a trivial

Table 7.3: Best parameters and the related Cross-Validation performance; train and test accuracy scores of the final Random Forest Regressions.

	Best Parameters			Accuracy		
	n_estimators	max_features	min_samples_split	CV	Train	Test
Reuters	500	20000	0.6	0.58383	0.58421	0.57789
Twitter	500	50000	0.5	0.58382	0.58382	0.57783

pattern. Specifically, for the Random Forest trained on Twitter data the biggest available number of samples to split and the smallest available percentage of features are the best parameters, while for Reuters dataset a slightly smaller number of samples and bigger percentage of data are chosen. However, performing an ANOVA procedure which compares the average validation accuracy of the best parameters (*max_features*=0.6, *min_samples_split*=20000) with the one with the extreme parameters (*max_features*=0.5, *min_samples_split*=50000) demonstrates that there is no evidence to state that the two means are different. Therefore also for Reuters dataset there is almost no difference on selecting the best parameters obtained by the validation or the two extreme ones. Moreover, another ANOVA procedure has been applied to compare the accuracy score obtained by the best parameters of each of the two Random Forests with 500 Decision Trees with the corresponding Random Forests having the same best parameters and 1000 Decision Trees. Since the ANOVA demonstrates that there is no evidence to state that the Random Forests trained on 1000 Decision Trees has a better validation accuracy score, the choice of selecting 500 Decision Trees is robust.

At this point, Random Forest regression is applied on each of the two datasets with the best parameters just described, using as training set data from 2017-05-02 to 2018-12-31 and as test set data from 2019-01-01 to 2019-10-09, obtaining the accuracy scores shown in Table 7.3. Moreover, in Appendix B are reported the plot of feature importances with their standard deviations, the table ranking the features and the plot comparing actions and predicted rewards, from which it is possible to understand that action, portfolio and percentage differences of the index values are the most important features for the prediction. However they are not much relevant, since in most cases the Random Forests are predicting a trivial slightly negative reward independently from the action. A possible explanation of this behavior is that, considering fifteen minutes differences of the index value, its percentage increase or decrease is not sufficiently large with respect to the fee that the agent must pay to change its position. Therefore the predicted reward is negative in most cases, since the available features are not sufficiently relevant to allow the regressors to learn more complex patterns. Moreover, sentiment features are not considered by the Feature Selection as relevant in predicting the reward, therefore this

Table 7.4: Regression of sentiment on residuals: accuracy score of the first Random Forests on the two datasets and related performance of the second Random Forest which tries to predict the residuals through sentiment principal components.

	Accuracy of first Regression			R-Squared of second Regression		
	CV	Train	Test	CV	Train	Test
Reuters	0.58383	0.58420	0.57787	-0.00000047	-0.00000048	-0.00000112
Twitter	0.58381	0.58382	0.57783	-0.00000074	-0.00000061	-0.00000466

is a first proof that also these fifteen minutes sentiment features do not have a positive impact on the prediction.

Regression of Sentiment on Residuals

The second Supervised Learning procedure, based on Random Forests and introduced in Section 6.4 for daily data, has also been applied to fifteen minutes datasets. The first Random Forests performed on each of the two datasets are tuned with the same best parameters found in Feature Selection. The second Random Forests are made of 500 Decision Trees, while the other two parameters have been optimized with 3-Fold Cross-Validation using the R-Squared as performance measure and using the same grid of values of Feature Selection, shown in Equations 7.2 and 7.3. The best parameters found are again the two extreme values 0.5 and 50000, and the related performances, reported in Table 7.4, are a second evidence that the sentiment features are not adding informativity on the prediction of the reward.

Procedures without Fees

The procedures explained above almost always predict a negative reward, which is largely dependent on the transaction costs, indeed the portfolio is a relevant feature for the prediction. For this reason, to better understand the importance of sentiment features, the Feature Selection and the Random Forest of sentiment on residuals have been also applied to the two available datasets, modified such that the target reward is computed without considering the transaction costs. This is an unrealistic assumption, which is made with the purpose to remove the trivial pattern predicted with transaction costs, in order to explore the importance of sentiment features more on depth.

The best parameters obtained with the already explained 3-Folds Cross-Validation are reported in Table 7.5, together with the accuracy scores of Cross-Validation, training and testing. The number of Decision Trees is again set equal to 500, while the best number of data to split and the percentage of features available at any node found are

Table 7.5: Target reward computed without fees: best parameters and the related Cross-Validation performance; train and test accuracy scores of the final Random Forest Regressions.

	Best Parameters			Accuracy		
	n_estimators	max_features	min_samples_split	CV	Train	Test
Reuters	500	5000	1.0	0.34354	0.37049	0.32712
Twitter	500	10000	0.8	0.34259	0.36324	0.33137

intermediate values among the possibilities (that are still the values of Equations 7.2 and 7.3), from which a non-trivial prediction is expected. Indeed, looking at the plots in Figure B.3 of the Appendix B, the predicted reward is close to 0 when the action is 0 and it often has larger values when the action is 1 or -1. Therefore the Random Forest is not predicting a trivial pattern and it seems to better predict the sign of the Reward. The fact that the accuracy scores are low happens because, without fees, one third of the actions is exactly equal to 0, reducing a lot the accuracy if the prediction is slightly positive or negative. However, considering only the samples where the action is 1 or -1 the test accuracy scores become 0.55839 and 0.54607 respectively for Reuters and Twitter dataset, which are acceptable values. Moreover, the 92.7% of the predicted rewards referring to samples from Reuters dataset having action different from 0 have absolute value larger than 0.003 (the 87.68% for what concerns Reuters dataset), while the 79.58% of samples having action equal to 0 have absolute value smaller than 0.003 (the 84.77% for Reuters). This means that when the action is 1 or -1 the accuracy is a satisfactory score and when the action is 0 most of the predicted values are smaller than the majority of predicted values with the other two actions, so the Regressions are correctly learning the fact that they correspond to small rewards. Finally, from the feature rankings reported in Table B.2 and in Figure B.4 of Appendix B, it is possible to state that the percentage differences of two consecutive values of the *S&P 500* index are the most relevant features and the action is much less relevant than them. This is not positive, because the values of the index affect the magnitude of the prediction but the action should determine the sign, therefore a lack of usage of the action may correspond to a difficulty to choose the best action in Reinforcement Learning framework. It is also important to underline that the sentiment principal components both from Reuters or Twitter data does not seem once again to be relevant for the prediction of the reward.

Also the second Supervised Learning technique has been applied to the datasets with rewards computed without transaction costs: the Random Forest Regression without sentiment features has a performance very similar to the one already explained, while

the second Random Forests predicting the residuals using the eight principal components of sentiment features have again R-Squared scores approximately equal to 0, confirming the lack of information added by the sentiment features on the prediction.

In conclusion, Feature Selection applied to the fifteen minutes datasets through Random Forests almost always predicts trivial negative values of the reward and it is not much promising for the leaning through Reinforcement Learning, while the one applied without considering transaction costs seems to be more promising, since it learns a more complex pattern, although it is an unrealistic model in practice, since transaction costs must be paid.

7.4.3 Reinforcement Learning

The three Reinforcement Learning algorithms applied in previous sections of this chapter (PPO, TRPO, FQI) have also been applied to MDPs designed on fifteen minutes data. In particular, five different MDPs have been designed in this framework. Actions, transition probabilities, reward functions, discount factors and initial probabilities of these five processes are the same described in Section 7.1. The difference is that an episode coincides with one day: each step is every fifteen minutes, from the opening to the close hour of a working day. In the first opening hour the first four values of the index are produced, which are part of the first observation. Therefore, an action is taken every fifteen minutes from one hour after the opening of the Stock Market to fifteen minutes before the closing, for a total amount of 22 steps. Moreover, the variables in the state differentiate the five different MDPs: all of them observe the portfolio; the first process observes only the percentage differences of last four values of the *S&P 500*; the second and third processes observe only the eight principal components respectively of Reuters and Twitter sentiment features; the fourth and fifth models observe both percentage differences of prices and sentiment principal components respectively of Reuters and Twitter sentiment features. In this way it is possible to compare the importance of sentiment features with respect to market prices but also to compare sentiment features extracted from Reuters news with the ones extracted by tweets.

Algorithms have been applied using data from 2017-05-02 to 2019-04-30 as training set and data from 2019-05-01 to 2019-10-09 for testing. In particular, training data have been split in **four** folds: the first with data from 2017-05-02 to 2017-10-31, the second from 2017-11-01 to 2018-04-30, the third from 2018-05-01 to 2018-10-31 and the fourth from 2018-11-01 to 2019-04-30. As described for daily data in Subsection 7.2.1, a 4-Folds Cross-Validation approach has been applied to the training set split in four groups in order to have more robust results and not rely only on the performance of the final test

set. Moreover, each training has been repeated five times, so that the stochasticity of the algorithms is taken into account.
Also the same Feedforward Neural Networks used in daily data has been adopted to parametrize the policy. PPO and TRPO algorithms have been trained using 2000 iterations, sampling 10000 episodes from them at each iteration. On the other hand, the parameters of the Random Forest Regression performed by FQI are set equal to the best parameters found on the datasets in Feature Selection procedures. Finally, the only parameter to tune the number of iteration of FQI, that will be chosen from 1 to 10 depending on the best average reward obtained in Cross-Validation.

Results
PPO, TRPO and FQI average reward in the four validation sets and in the final test set are reported in Appendix B.2. FQI best mean reward in validation is obtained with 1 iteration, which is an evidence of the fact that the agent is not learning a significant policy but a trivial one. This is confirmed by all validation and testing results, which have a mean reward very similar to the trivial baselines of doing the always same action. Moreover there is clearly no significance on the fact that these mean rewards are different from zero. Therefore it is possible to conclude that in all the five environments and independently from the chosen algorithm, the agent does not learn a policy able to earn money by trading.

As already introduced in the Feature Selection Subsection 7.4.2, it is interesting to perform the algorithms also on the MDPs which compute the reward without considering transaction costs. Indeed, from the almost always negative prediction of Random Forests it had already been guessed that the algorithms would have learned a trivial, not profitable policy, while the prediction of the Random Forests considering the reward without transaction costs seemed to be more promising. Therefore, the three algorithms have been performed also on the five MDPs already introduced, without considering transaction costs. The results are reported in Appendix B.2 and they show that the validation and testing performances are not different from the ones obtained considering the transaction costs, therefore it is possible to conclude that also in this framework the agent is not learning a profitable policy.

In conclusion, fifteen minutes data confirm what largely discussed for daily data: the results clearly show that both historical series of the *S&P 500* and sentiment features from Reuters news or tweets are not relevant in the prediction of the trend of the index; moreover, the application of Reinforcement Learning algorithms on them does not produce a policy able to earn money on the U.S. Stock Market investing on the *S&P 500* following the action suggested by the learned policy.

Chapter 8

Conclusion

This final chapter concludes the thesis resuming the main results found and proposes some possible future development to improve the achieved performances.

8.1 Results

Most of the analysis performed in this work uses daily sentiment features from Reuters news and daily historical series of the *S&P 500* index.

The analysis of the *S&P 500* historical series performed in Section 5.1 shows a strongly increasing trend of the index, which could be a criticality in learning more complex pattern between the available features and the index value.

After the elaboration of the available features and the addition in Section 5.3 of some of them, a Feature Selection technique based on Random Forests performed in Section 6.3 shows that the only important feature to predict the reward is the *action*. Analyzing the Feature Selection results it is clear that the predictor is learning the trivial pattern to always predict a reward of the same sign of the observed action.

Another Supervised Learning technique, applied in Section 6.4 to specifically investigate the informativity on the prediction added by the sentiment features, confirms that the sentiment is not providing contribution to the prediction.

Then, three Reinforcement Learning algorithms are applied to three MDPs designed in Section 7.1. In Section 7.3, comparing the resulting average daily rewards with a baseline score obtained always performing *buy* action, it is clear that, except for a few variations due to the noise inside the algorithms, the RL agent is always performing the

trivial action *buy*. This is due to the clearly increasing trend of the *S&P 500*, which overcomes any other weaker pattern, so the agent learns that the index is mainly increasing, implying that the optimal way to earn is to always keep a *long position*.

Finally, in Section 7.4 all the analysis is repeated with fifteen minutes data but, also in that case, there is no evidence that the RL agent is learning a policy to trade on the *S&P 500* profitably.

8.2 Future Improvements

The work presented in this thesis shows the lack of informativity of the available features for the training of a RL agent able to earn trading on the U.S. Stock Market. Therefore future improvements should be focused on the revision and the addition of features.

Since the *S&P 500* is composed by many different companies, the first possible improvement could be to consider the sentiment of each company as a different feature, or to group them into smaller clusters and extract one sentiment feature for each of these clusters. In this way many specific sentiment features would be available as features, trying to improve their informativity.

Another possibility could be to efficiently add different features describing the condition of the U.S. Stock Market, so that more information about the Market could be added to the variables observed by the agent, refining the description of the Market at its disposal when it makes decisions.

Then, the natural focus should be moved to the fifteen minutes data, since it should be easier to predict a short time value rather than the value after one day. The results obtained applying PCA to the numerous available sentiment features lead to a poor performance by the RL agent, therefore a more complex Feature Selection exploring the importance of the Sentiment Features without projecting them applying PCA may lead to more significant results.

Another improvement to the models designed with fifteen minutes data could be review the transaction costs: as shown in the related section, the prediction considering the same costs applied in daily-based models and the one without transaction costs are very different, therefore the fees are a relevant issue in the prediction and more accurate estimates of their value may be crucial for learning a profitable policy in fifteen minutes framework.

Appendices

Appendix A

Reinforcement Learning Performances

The purpose of this appendix is to report the performance measures described in Subsection 7.2.3, applied on the ten models trained in Cross-Validation fashion using data from 2009 to 2018 and for the final model tested on 2019 data. The scores are reported for the eleven models trained with PPO, TRPO and FQI algorithms.
Recalling that returns are computed on 49 consecutive days and the validation or test data available are about 250 in each model, the average return and its standard deviations are not very reliable estimate of the real expected return, since they are basically averaged on five samples, so the daily reward is important and more robust to evaluate the results.

Table A.1: PPO Validation and Test reward performance.

Year	Environment	Mean	Variance (vola)	Stddev	CVaR	Stddev CVaR
2009	Sentiment	0.00197	0.00025	0.01596	-0.03315	0.00497
	Prices	0.00180	0.00025	0.01598	-0.03315	0.00497
	Both	0.00178	0.00025	0.01598	-0.03315	0.00497
2010	Sentiment	0.00022	0.00012	0.01135	-0.02861	0.00449
	Prices	0.00020	0.00012	0.01134	-0.02861	0.00449
	Both	0.00022	0.00012	0.01135	-0.02861	0.00449
2011	Sentiment	-0.00021	0.00019	0.01412	-0.03802	0.01207
	Prices	-0.00021	0.00019	0.01412	-0.03802	0.01207
	Both	-0.00041	0.00019	0.01412	-0.03802	0.01207
2012	Sentiment	0.00057	0.00006	0.00788	-0.01672	0.00358
	Prices	0.00057	0.00006	0.00788	-0.01672	0.00358
	Both	0.00057	0.00006	0.00788	-0.01672	0.00358
2013	Sentiment	0.00087	0.00004	0.00679	-0.01606	0.00397
	Prices	0.00087	0.00004	0.00679	-0.01606	0.00397
	Both	0.00087	0.00004	0.00679	-0.01606	0.00397
2014	Sentiment	0.00011	0.00004	0.00661	-0.01791	0.00328
	Prices	0.00011	0.00004	0.00661	-0.01791	0.00328
	Both	0.00011	0.B0004	0.00661	-0.01791	0.00328
2015	Sentiment	0.00016	0.00008	0.00913	-0.02268	0.00721
	Prices	0.00021	0.00008	0.00910	-0.02268	0.00721
	Both	0.00016	0.00008	0.00913	-0.02268	0.00721
2016	Sentiment	0.00073	0.00005	0.00745	-0.01728	0.00705
	Prices	0.00073	0.00005	0.00745	-0.01728	0.00705
	Both	0.00073	0.00005	0.00745	-0.01728	0.00705
2017	Sentiment	0.00068	0.00001	0.00393	-0.00994	0.00311
	Prices	0.00068	0.00001	0.00393	-0.00994	0.00311
	Both	0.00068	0.00001	0.00393	-0.00994	0.00311
2018	Sentiment	-0.00002	0.00008	0.00926	-0.02574	0.00902
	Prices	-0.00001	0.00008	0.00926	-0.02574	0.00902
	Both	-0.00001	0.00008	0.00926	-0.02574	0.00902
2019	Sentiment	0.00114	0.00003	0.00610	-0.01249	0.00258
	Prices	0.00109	0.00003	0.00614	-0.01249	0.00258
	Both	0.00109	0.00003	0.00614	-0.01249	0.00258

Table A.2: PPO Validation and Test return performance.

Year	Environment	Mean	Stddev
2009	Sentiment	0.08727	0.05035
	Prices	0.08727	0.05035
	Both	0.08727	0.05035
2010	Sentiment	0.01085	0.06281
	Prices	0.00981	0.06187
	Both	0.01122	0.06316
2011	Sentiment	-0.01048	0.02913
	Prices	-0.01048	0.02913
	both	-0.02016	0.02913
2012	Sentiment	0.02841	0.08709
	Prices	0.02841	0.08709
	Both	0.02841	0.08709
2013	Sentiment	0.04308	0.01246
	Prices	0.04308	0.01246
	Both	0.04308	0.01246
2014	Sentiment	0.00586	0.02038
	Prices	0.00586	0.02038
	Both	0.00586	0.02038
2015	Sentiment	0.00819	0.01516
	Prices	0.01058	0.01579
	Both	0.00819	0.01516
2016	Sentiment	0.03609	0.02431
	Prices	0.03609	0.02431
	Both	0.03609	0.02431
2017	Sentiment	0.03337	0.01104
	Prices	0.03337	0.01104
	Both	0.03337	0.01104
2018	Sentiment	-0.00132	0.04236
	Prices	-0.00097	0.04270
	Both	-0.00097	0.04270
2019	Sentiment	0.05613	0.03113
2019	Prices	0.05355	0.03445
2019	Both	0.05355	0.03445

Table A.3: TRPO Validation and Test reward performance.

Year	Environment	Mean	Variance (vola)	Stddev	CVaR	Stddev CVaR
2009	Sentiment	0.00178	0.00025	0.01598	-0.03315	0.00497
	Prices	0.00032	0.00025	0.01607	-0.03721	0.01062
	Both	0.00179	0.00025	0.01598	-0.03346	0.00463
2010	Sentiment	0.00022	0.00012	0.01135	-0.02861	0.00449
	Prices	0.00009	0.00009	0.00990	-0.02302	0.00583
	Both	0.00060	0.00012	0.01134	-0.02839	0.00521
2011	Sentiment	-0.00038	0.00019	0.01412	-0.03654	0.01313
	Prices	-0.00021	0.00019	0.01412	-0.03802	0.01207
	Both	-0.00027	0.00019	0.01412	-0.03802	0.01207
2012	Sentiment	-0.00001	0.00005	0.00754	-0.01743	0.00328
	Prices	-0.00016	0.00006	0.00791	-0.01744	0.00325
	Both	0.00057	0.00006	0.00788	-0.01672	0.00358
2013	Sentiment	0.00087	0.00004	0.00679	-0.01606	0.00397
	Prices	0.00057	0.00003	0.00605	-0.01490	0.00354
	Both	0.00027	0.00004	0.00685	-0.01530	0.00328
2014	Sentiment	-0.00047	0.00003	0.00585	-0.01735	0.00362
	Prices	0.00011	0.00004	0.00661	-0.01791	0.00328
	Both	0.00011	0.B0004	0.00661	-0.01791	0.00328
2015	Sentiment	0.00037	0.00008	0.00912	-0.02058	0.00641
	Prices	0.00011	0.00008	0.00913	-0.02354	0.00671
	Both	0.00019	0.00008	0.00908	-0.02255	0.00735
2016	Sentiment	0.00088	0.00004	0.00654	-0.01336	0.00313
	Prices	0.00074	0.00005	0.00745	-0.01774	0.00675
	Both	0.00030	0.00005	0.00707	-0.01751	0.00687
2017	Sentiment	0.00064	0.00001	0.00393	-0.00994	0.00311
	Prices	0.00064	0.00001	0.00393	-0.00994	0.00311
	Both	0.00066	0.00001	0.00381	-0.00956	0.00343
2018	Sentiment	-0.00001	0.00008	0.00926	-0.02574	0.00902
	Prices	-0.00009	0.00008	0.00926	-0.02574	0.00902
	Both	-0.00049	0.00008	0.00916	-0.02684	0.00823
2019	Sentiment	0.00103	0.00003	0.00580	-0.01249	0.00258
	Prices	0.00073	0.00003	0.00619	-0.01363	0.00240
	Both	0.00096	0.00003	0.00616	-0.01316	0.00214

Table A.4: TRPO Validation and Test return performance.

Year	Environment	Mean	Stddev
2009	Sentiment	0.08727	0.05035
	Prices	0.01607	0.06528
	Both	0.08719	0.07215
2010	Sentiment	0.01122	0.06316
	Prices	0.00487	0.07136
	Both	0.02966	0.02323
2011	Sentiment	-0.01888	0.04999
	Prices	-0.01048	0.02913
	both	-0.01338	0.02935
2012	Sentiment	-0.00088	0.05468
	Prices	-0.00807	0.04445
	Both	0.02841	0.08709
2013	Sentiment	0.04308	0.01246
	Prices	0.02817	0.01185
	Both	0.01326	0.02345
2014	Sentiment	-0.02349	0.02916
	Prices	0.00586	0.02038
	Both	0.00586	0.02038
2015	Sentiment	0.01823	0.00812
	Prices	0.00577	0.01516
	Both	0.00952	0.01339
2016	Sentiment	0.04346	0.03467
	Prices	0.03640	0.03842
	Both	0.01471	0.05059
2017	Sentiment	0.03155	0.01438
	Prices	0.03337	0.01104
	Both	0.03241	0.01650
2018	Sentiment	-0.00097	0.04270
	Prices	-0.00441	0.05908
	Both	-0.02437	0.04461
2019	Sentiment	0.05079	0.02936
2019	Prices	0.03585	0.02038
2019	Both	0.04724	0.03530

Table A.5: FQI Validation and Test reward performance.

Year	Environment	Mean	Variance (vola)	Stddev	CVaR	Stddev CVaR
2009	Sentiment	0.00157	0.00024	0.01557	-0.03252	0.00502
	Prices	0.00157	0.00024	0.01557	-0.03252	0.00502
	Both	0.00157	0.00024	0.01557	-0.03252	0.00502
2010	Sentiment	0.00018	0.00012	0.01147	-0.02861	0.00449
	Prices	0.00018	0.00012	0.01147	-0.02861	0.00449
	Both	0.00018	0.00012	0.01147	-0.02861	0.00449
2011	Sentiment	0.00006	0.00020	0.01432	-0.03803	0.01207
	Prices	0.00006	0.00020	0.01432	-0.03803	0.01207
	Both	0.00006	0.00020	0.01432	-0.03803	0.01207
2012	Sentiment	0.00046	0.00006	0.00794	-0.01724	0.00319
	Prices	0.00046	0.00006	0.00794	-0.01724	0.00319
	Both	0.00046	0.00006	0.00794	-0.01724	0.00319
2013	Sentiment	0.00095	0.00004	0.00674	-0.01606	0.00397
	Prices	0.00095	0.00004	0.00674	-0.01606	0.00397
	Both	0.00095	0.00004	0.00674	-0.01606	0.00397
2014	Sentiment	0.00033	0.00004	0.00679	-0.01791	0.00328
	Prices	0.00033	0.00004	0.00679	-0.01791	0.00328
	Both	0.00033	0.00004	0.00679	-0.01791	0.00328
2015	Sentiment	0.00027	0.00008	0.00900	-0.02205	0.00764
	Prices	0.00027	0.00008	0.00900	-0.02205	0.00764
	Both	0.00027	0.00008	0.00900	-0.02205	0.00764
2016	Sentiment	0.00064	0.00005	0.00737	-0.01728	0.00705
	Prices	0.00064	0.00005	0.00737	-0.01728	0.00705
	Both	0.00064	0.00005	0.00737	-0.01728	0.00705
2017	Sentiment	0.00064	0.00001	0.00390	-0.00994	0.00311
	Prices	0.00064	0.00001	0.00390	-0.00994	0.00311
	Both	0.00064	0.00001	0.00390	-0.00994	0.00311
2018	Sentiment	-0.00027	0.00008	0.00937	-0.02622	0.00868
	Prices	-0.00027	0.00008	0.00937	-0.02622	0.00868
	Both	-0.00027	0.00008	0.00937	-0.02622	0.00868
2019	Sentiment	0.00068	0.00004	0.00682	-0.01444	0.00181
	Prices	0.00068	0.00004	0.00682	-0.01444	0.00181
	Both	0.00068	0.00004	0.00682	-0.01444	0.00181

Table A.6: FQI Validation and Test return performance.

Year	Environment	Mean	Stddev
2009	Sentiment	0.07726	0.06880
	Prices	0.07726	0.06880
	Both	0.07726	0.06880
2010	Sentiment	0.00906	0.07142
	Prices	0.00906	0.07142
	Both	0.00906	0.07142
2011	Sentiment	0.00299	0.09224
	Prices	0.00299	0.09224
	both	0.00299	0.09224
2012	Sentiment	0.02258	0.06081
	Prices	0.02258	0.06081
	Both	0.02258	0.06081
2013	Sentiment	0.04674	0.02493
	Prices	0.04674	0.02493
	Both	0.04674	0.02493
2014	Sentiment	0.01635	0.01994
	Prices	0.01635	0.01994
	Both	0.01635	0.01994
2015	Sentiment	0.01366	0.02119
	Prices	0.01366	0.02119
	Both	0.01366	0.02119
2016	Sentiment	0.03166	0.03962
	Prices	0.03166	0.03962
	Both	0.03166	0.03962
2017	Sentiment	0.03174	0.02045
	Prices	0.03174	0.02045
	Both	0.03174	0.02045
2018	Sentiment	-0.01345	0.05734
	Prices	-0.01345	0.05734
	Both	-0.01345	0.05734
2019	Sentiment	0.03353	0.02841
2019	Prices	0.03353	0.02841
2019	Both	0.03353	0.02841

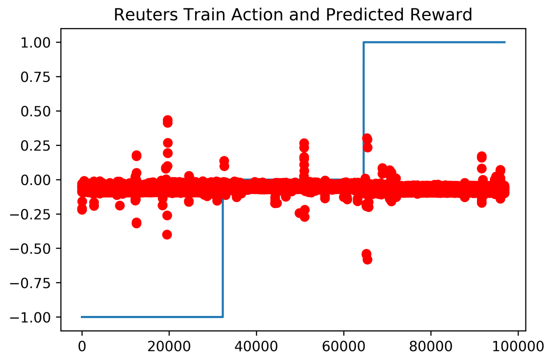
Appendix B

15 Minutes Results

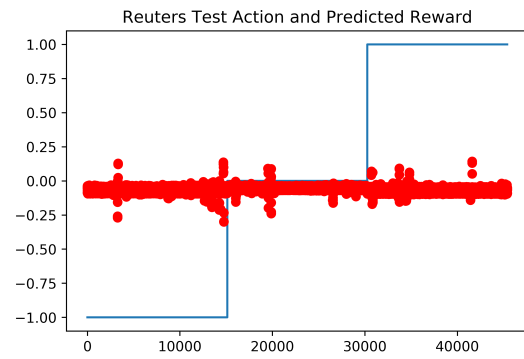
This second appendix collects the results found in fifteen minutes framework. In particular in first section the results of Feature Selection procedure are reported, while in the second one the results of Reinforcement Learning applications are shown.

B.1 Feature Selection Results

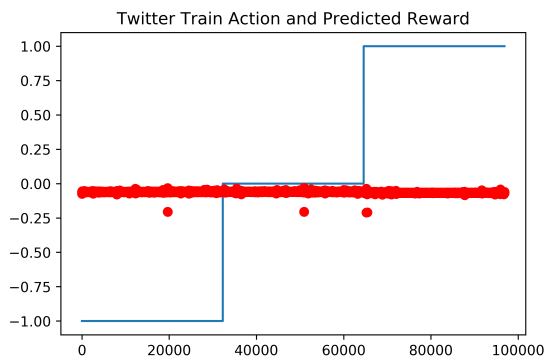
In this section the results of Feature Selection approach on fifteen minutes data described in Section 7.4 are reported. In particular, in Figure B.1 the plots of the predicted train and test rewards on the two datasets are shown, together with the action in order to understand that there is no difference on the prediction depending on the action. Then, Table B.1 presents the best fifteen features based on their percentage importance in the corresponding Random Forests on the prediction of the reward; their importance, together with the corresponding standard deviation, is also shown in Figure B.2. The same figures and tables are repeated for the Random Forest Feature Selection procedures designed considering the reward without transaction costs. The predicted rewards and the corresponding actions are plotted in Figure B.3, while the best fifteen features together with their importances and standard deviations are ranked and plotted in Table B.2 and in Figure B.4.



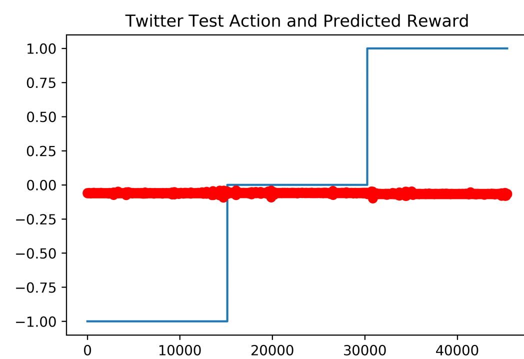
(a) Train action and predicted reward on fifteen minutes Reuters dataset.



(b) Test action and predicted reward on fifteen minutes Reuters dataset.



(c) Train action and predicted reward on fifteen minutes Twitter dataset.

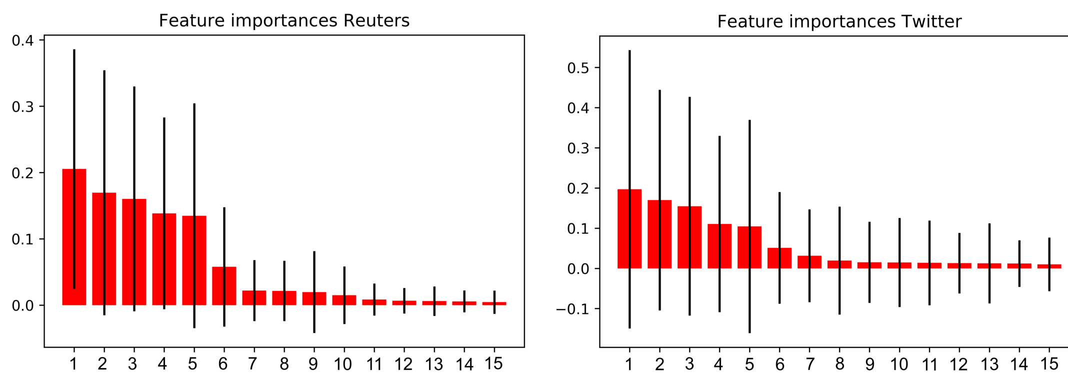


(d) Test action and predicted reward on fifteen minutes Twitter dataset.

Figure B.1: The blue line in the figures represent the value of the action in each sample, respectively belonging to train and test set, which can be 1, -1 or 0. The red points are the corresponding predicted rewards. From their values is clear that they are always negative independently from the action.

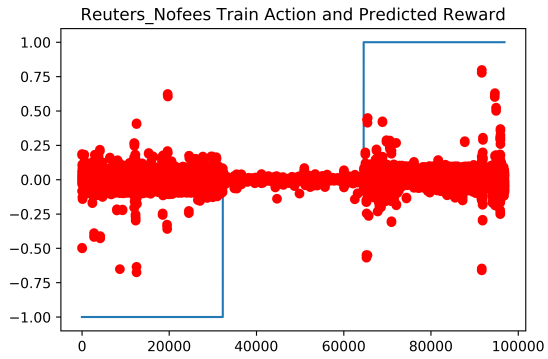
Table B.1: Feature ranking and performance of the two Random Forests trained using Reuters or Twitter fifteen minutes datasets.

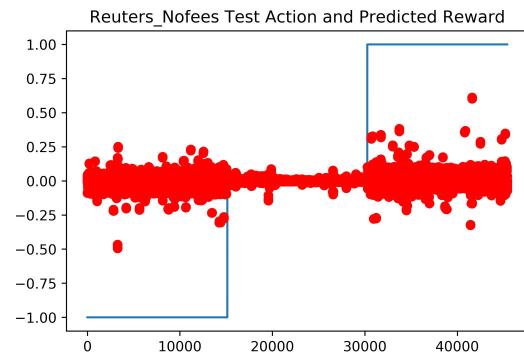
Reuters Feature Selection		Twitter Feature Selection	
Feature	Importance	Feature	Importance
R0	20.54%	action	19.70%
R1	16.96%	R0	17.01%
portfolio	16.04%	R1	15.47%
R2	13.85%	R2	11.04%
action	13.50%	portfolio	10.44%
R3	5.79%	R3	5.11%
December	2.20%	December	3.14%
PC1	2.16%	Wednesday	1.95%
PC8	1.99%	PC7	1.52%
Daytime	1.52%	Tuesday	1.46%
October	0.86%	Thursday	1.38%
April	0.69%	PC1	1.30%
PC7	0.62%	Friday	1.26%
March	0.58%	March	1.20%
PC6	0.46%	PC3	1.01%



(a) Feature importances in Reuter dataset. (b) Feature importances in Twitter dataset.

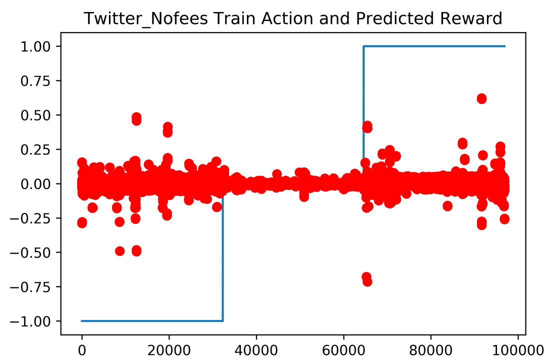
Figure B.2: Histogram of importances of best 15 features for each of the two Random Forests with related standard deviations.

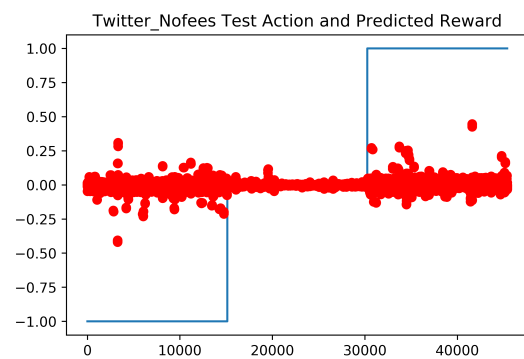




(a) Train action and predicted reward on fifteen minutes Reuters dataset.

(b) Test action and predicted reward on fifteen minutes Reuters dataset.





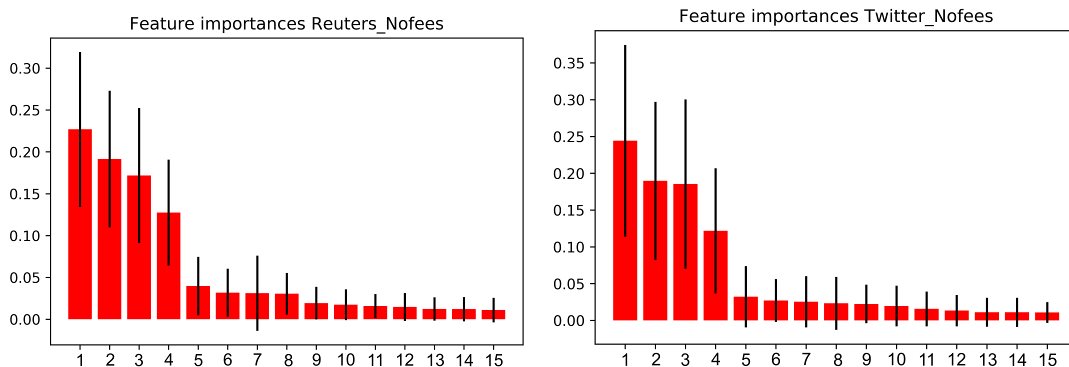
(c) Train action and predicted reward on fifteen minutes Twitter dataset.

(d) Test action and predicted reward on fifteen minutes Twitter dataset.

Figure B.3: The blue line in the figures represent the value of the action in each sample, respectively belonging to train and test set, which can be 1, -1 or 0. The red points are the corresponding predicted rewards computed without considering transaction costs.

Table B.2: Feature ranking and performance of the two Random Forests trained using Reuters or Twitter fifteen minutes datasets computing the target reward without considering transaction costs.

Reuters Feature Selection		Twitter Feature Selection	
Feature	Importance	Feature	Importance
R0	22.69%	R0	24.42%
R2	19.15%	R2	18.97%
R1	17.18%	R1	18.54%
R3	12.75%	R3	12.18%
Daytime	3.98%	Daytime	3.22%
action	3.17%	action	2.71%
PC8	3.12%	PC1	2.54%
PC1	3.05%	PC3	2.33%
December	1.93%	December	2.24%
PC6	1.73%	PC7	1.96%
April	1.58%	PC8	1.57%
PC5	1.48%	October	1.34%
October	1.24%	PC5	1.12%
March	1.21%	PC6	1.11%
PC7	1.11%	April	1.08%



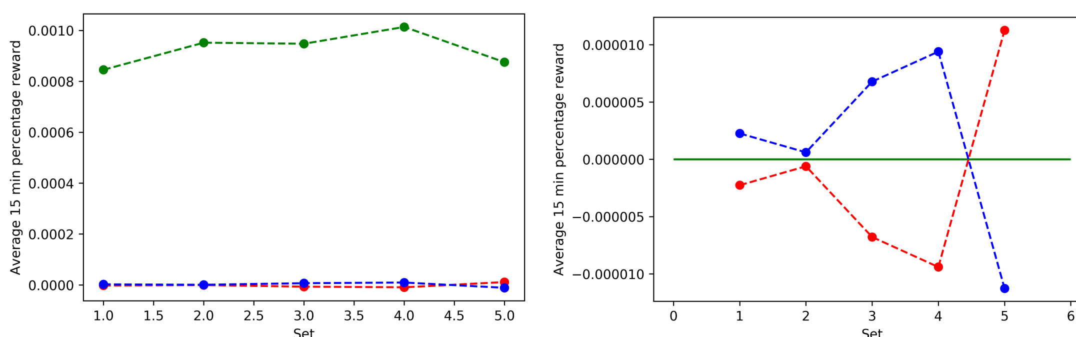
(a) Feature importances in Reuters dataset.

(b) Feature importances in Twitter dataset.

Figure B.4: Histogram of importances of best 15 features with related standard deviations for each of the two Random Forests which consider the reward without transaction costs.

B.2 Reinforcement Learning Results

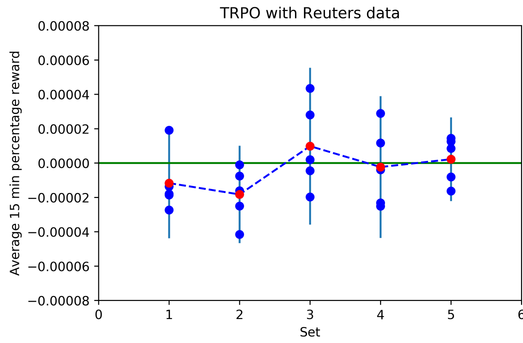
This section reports the results obtained in validation and testing applying PPO, TRPO and FQI to the five MDPs designed in the fifteen minutes framework. In particular, in Figure B.5 some baseline scores are shown to compare them with the results of the algorithms. Then, in Figure B.6, B.7 and B.8 the results respectively of TRPO, PPO and FQI are displayed. From the plots it is clear that the results are almost equal to the trivial baselines and very smaller than the best possible performance, hence the learned policy is not satisfactory to trade and earn on the stock market.
Finally, the validation and testing average fifteen minutes rewards are reported for the models considering the same five MDPs without computing transaction costs in the rewards. TRPO, PPO and FQI scores are respectively plotted in Figure B.9, B.10 and B.11. They are very similar to the ones found with transaction costs, therefore it is possible to conclude that the agent is still not learning a profitable policy.



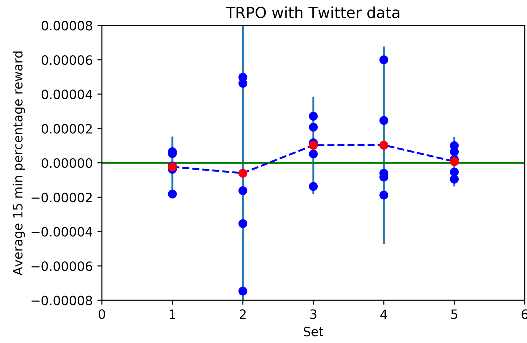
(a) Three different baselines.

(b) Focus on the two trivial baselines.

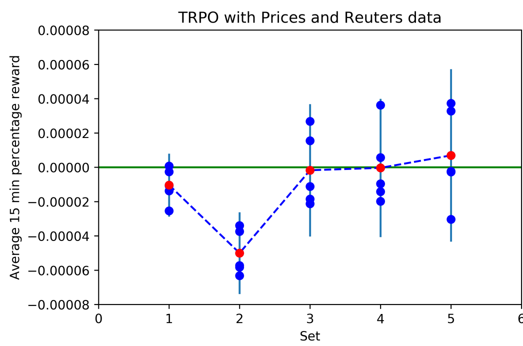
Figure B.5: Three different baseline performance scores are reported in these figures: the green points are the best possible average rewards achievable in the four validation sets and in testing, computed by always selecting the best possible action. The red and blue points are the mean rewards obtainable always performing respectively the trivial actions -1 and +1. These two baselines are repeated in the second figure, which focuses only on them, to show their trend which is not clearly understandable in first figure since they have an order of magnitude less than the other baseline.



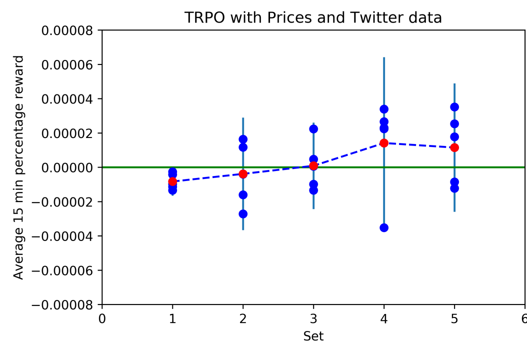
(a) Only Reuters sentiment features principal components in the state.



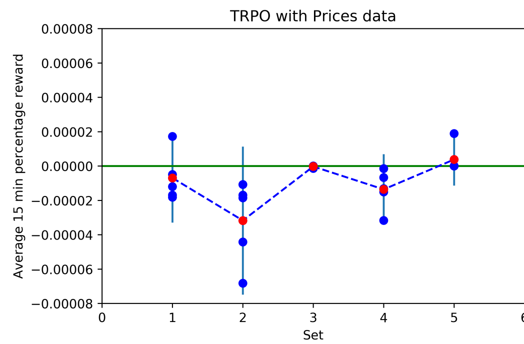
(b) Only Twitter sentiment features principal components in the state.



(c) Reuters sentiment features principal components and percentage differences of index values in the state.

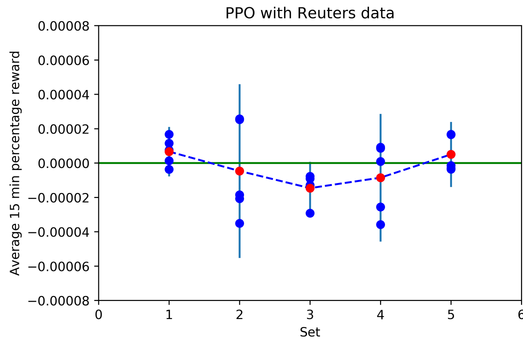


(d) Twitter sentiment features principal components and percentage differences of index values in the state.

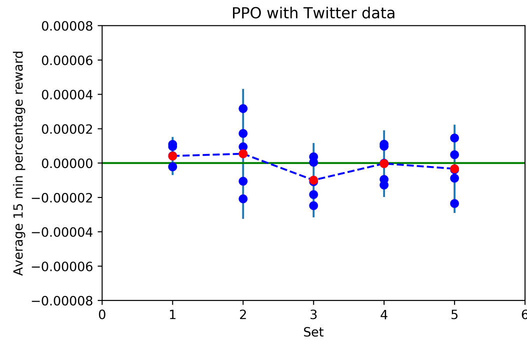


(e) Only percentage differences of index values in the state.

Figure B.6: TRPO 15 minutes mean reward in validation and testing, repeating the same model five times and computing the related confidence intervals.

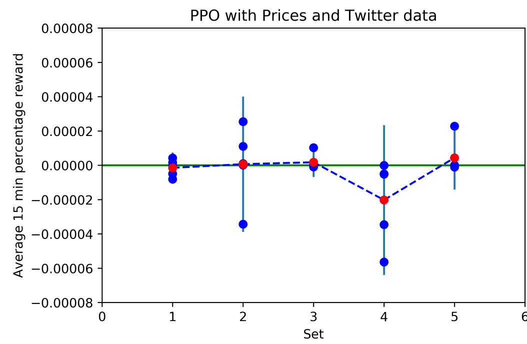


(a) Only Reuters sentiment features principal components in the state.



(b) Only Twitter sentiment features principal components in the state.

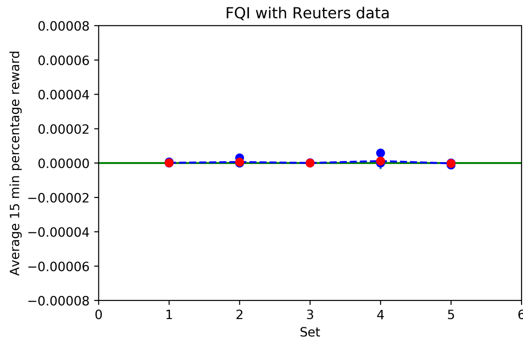
(c) Reuters sentiment features principal components and percentage differences of index values in the state.



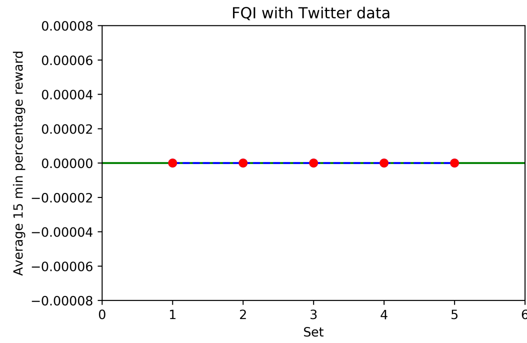
(d) Twitter sentiment features principal components and percentage differences of index values in the state.

(e) Only percentage differences of index values in the state.

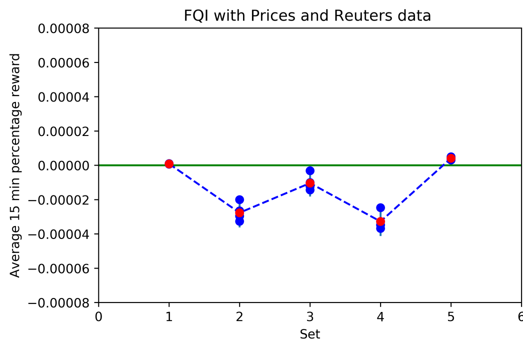
Figure B.7: PPO 15 minutes mean reward in validation and testing, repeating the same model five times and computing the related confidence intervals.



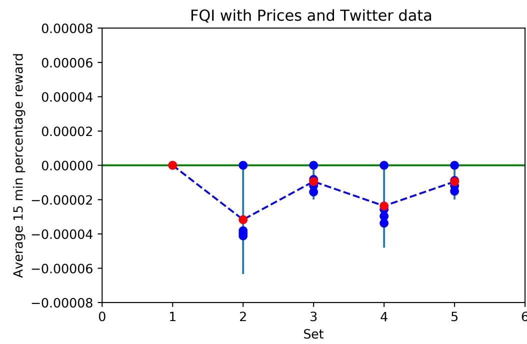
(a) Only Reuters sentiment features principal components in the state.



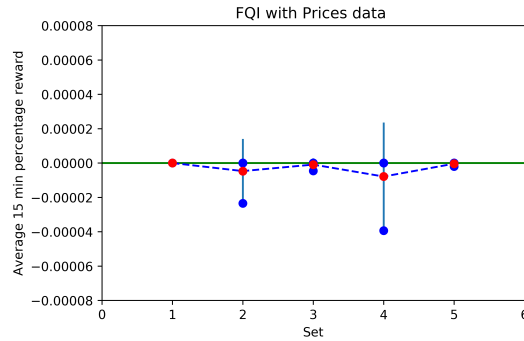
(b) Only Twitter sentiment features principal components in the state.



(c) Reuters sentiment features principal components and percentage differences of index values in the state.

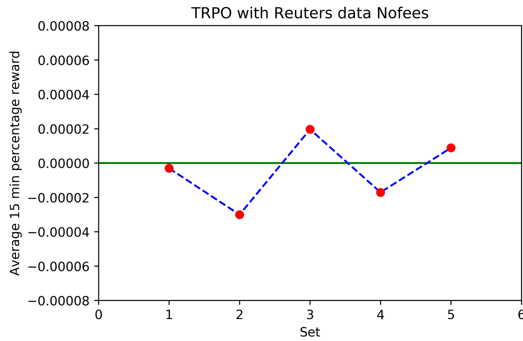


(d) Twitter sentiment features principal components and percentage differences of index values in the state.

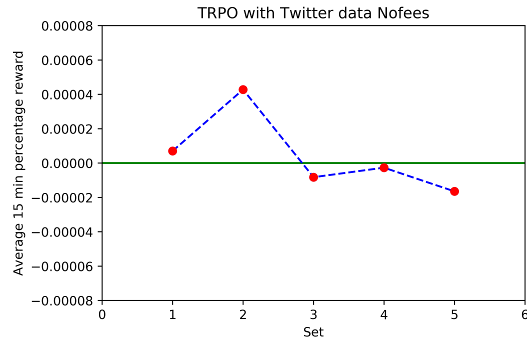


(e) Only percentage differences of index values in the state.

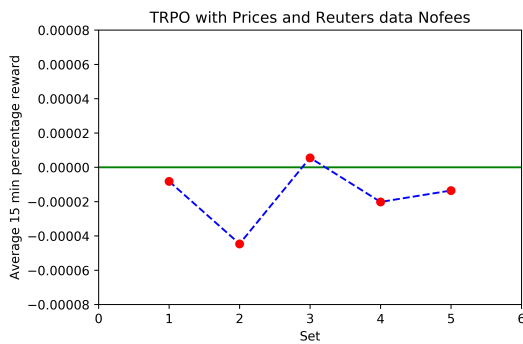
Figure B.8: FQI 15 minutes mean reward in validation and testing, repeating the same model five times and computing the related confidence intervals.



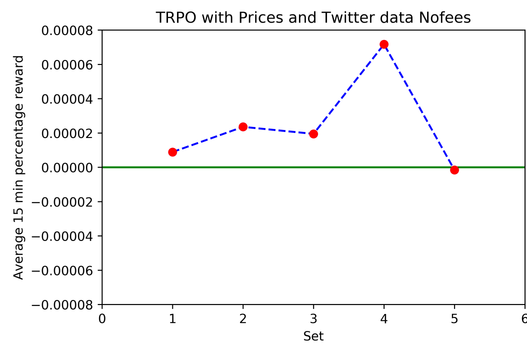
(a) Only Reuters sentiment features principal components in the state.



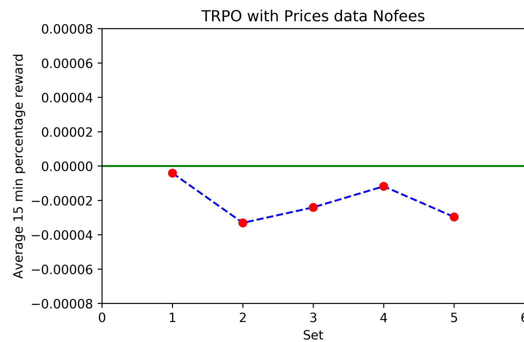
(b) Only Twitter sentiment features principal components in the state.



(c) Reuters sentiment features principal components and percentage differences of index values in the state.

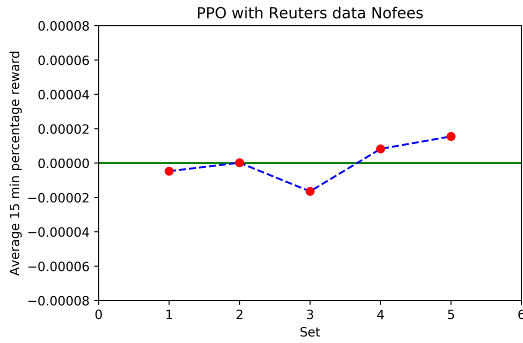


(d) Twitter sentiment features principal components and percentage differences of index values in the state.

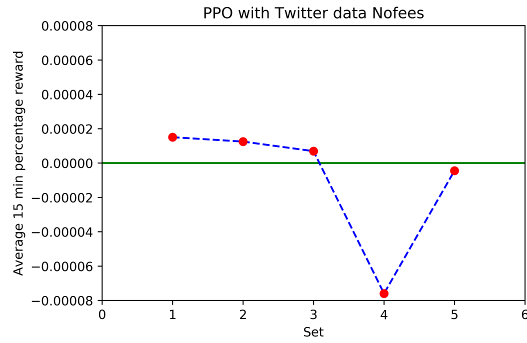


(e) Only percentage differences of index values in the state.

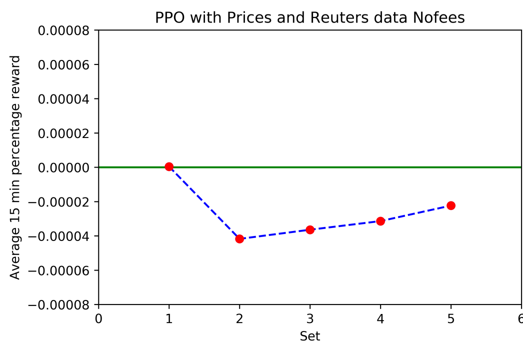
Figure B.9: TRPO 15 minutes mean reward in validation and testing, without considering transaction costs.



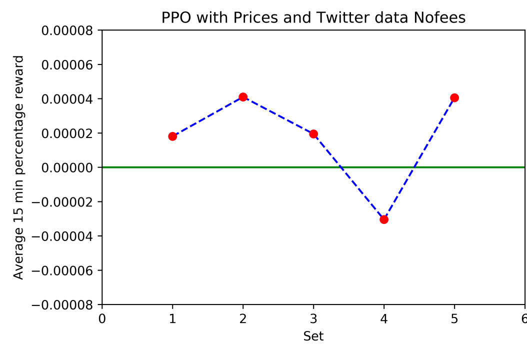
(a) Only Reuters sentiment features principal components in the state.



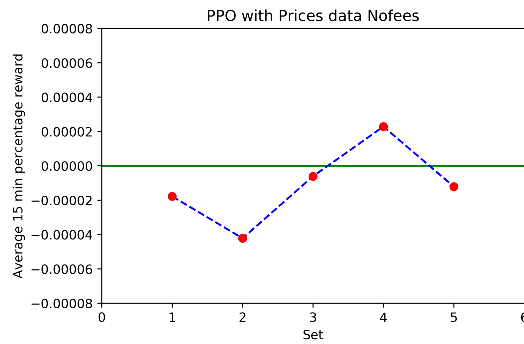
(b) Only Twitter sentiment features principal components in the state.



(c) Reuters sentiment features principal components and percentage differences of index values in the state.

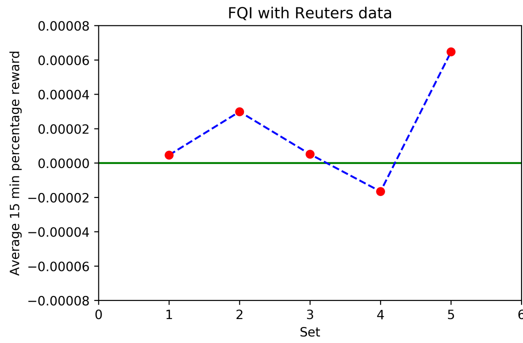


(d) Twitter sentiment features principal components and percentage differences of index values in the state.

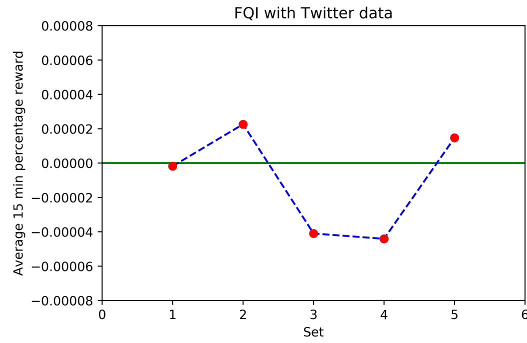


(e) Only percentage differences of index values in the state.

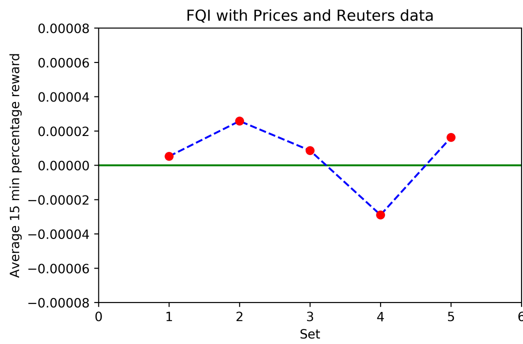
Figure B.10: PPO 15 minutes mean reward in validation and testing, without considering transaction costs.



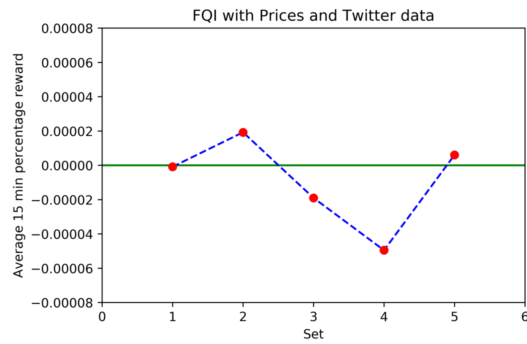
(a) Only Reuters sentiment features principal components in the state.



(b) Only Twitter sentiment features principal components in the state.



(c) Reuters sentiment features principal components and percentage differences of index values in the state.



(d) Twitter sentiment features principal components and percentage differences of index values in the state.

(e) Only percentage differences of index values in the state.

Figure B.11: FQI 15 minutes mean reward in validation and testing, without considering transaction costs.

Bibliography

[1] Martín Abadi, Ashish Agarwal, et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: `https://www.tensorflow.org/`.

[2] Carlo Acerbi and Dirk Tasche. "Expected Shortfall: a natural coherent alternative to Value at Risk". In: *Economic Notes* 31 (2002), pp. 379–388.

[3] Stefan Banach. "Sur les opérations dans les ensembles abstraits et leur application aux équations intégrales". In: *Fundamenta Mathematicae* 3 (1922), pp. 133–181.

[4] Richard Bellman. "A Markovian Decision Process". In: *Journal of Mathematics and Mechanics* 6.5 (1957), pp. 679–684.

[5] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. "Learning Long-Term Dependencies with Gradient Descent is Difficult". In: *IEEE Transactions on Neural Networks* 5 (Feb. 1994), pp. 157–166.

[6] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Jauvin. "A Neural Probabilistic Language Model". In: *Journal of Machine Learning Research* 3 (2003), pp. 1137–1155.

[7] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer-Verlag Berlin, 2006.

[8] Mikael Bodén. "A Guide to Recurrent Neural Networks and Backpropagation". In: *the Dallas project* (Dec. 2001).

[9] Leo Breiman. "Random Forests". In: *Machine Learning* 45 (2001), pp. 5–32.

[10] Leo Breiman, Jerom H. Friedman, Richard A. Olshen, and Charles J. Stone. *Classification and Regression Trees*. Belmont, CA: Wadsworth and Brooks, 1984.

[11] Rohan Chitnis and Tomas Lozano-Perez. "Learning Compact Models for Planning with Exogenous Processes". In: *3rd Conference on Robot Learning (CoRL)* (2019).

[12] Kyunghyun Cho, Bart van Merrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. *Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation.* 2014. arXiv: `1406.1078 [cs.CL]`.

[13] Thomas Degris, Martha White, and Richard S. Sutton. "Off-policy actor-critic". In: *Proceedings of the* 29th *International Conference on Machine Learning* (2012).

[14] F. D'Epenoux. "A probabilistic production and inventory problem". In: *Management Science* 10 (1963), pp. 98–108.

[15] Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, Yuhuai Wu, and Peter Zhokhov. *OpenAI Baselines.* `https://github.com/openai/baselines`. 2017.

[16] Damien Ernst, Pierre Geurts, and Louis Wehenkel. "Tree-Based Batch Mode Reinforcement Learning". In: *Journal of Machine Learning Research* 6 (2005), pp. 503–556.

[17] Pierre Geurts, Damien Ernst, and Louis Wehenkel. "Extremely Randomized Trees". In: *Machine Learning* 63 (2006), pp. 3–42.

[18] Alec Go, Richa Bhayani, and Lei Huang. "Twitter sentiment classification using distant supervision". In: *CS224N Project Report, Stanford* (2009).

[19] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning.* MIT Press, 2016.

[20] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-term Memory". In: *Neural computation* 9 (Dec. 1997), pp. 1735–1780.

[21] S&P Dow Jones Indices. *S&P 500.* URL: `https://us.spindices.com/indices/equity/sp-500` (visited on 03/03/2020).

[22] S&P Dow Jones Indices. *S&P 500 Top 50.* URL: `https://us.spindices.com/indices/equity/sp-500-top-50` (visited on 03/03/2020).

[23] S&P Dow Jones Indices. *S&P U.S. Indices Metodology.* URL: `https://us.spindices.com/documents/methodologies/methodology-sp-us-indices.pdf` (visited on 03/03/2020).

[24] Leslie P. Kaelbling, Michael L. Littman, and Anthony R. Cassandra. "Planning and acting in partially observable stochastic domains". In: *Artificial Intelligence* 101 (1998), pp. 99–134.

[25] Sham Kakade and John Langford. "Approximately Optimal Approximate Reinforcement Learning". In: *Proceedings of the* 19th *International Conference on Machine Learning* (2002).

[26] Carrel Lawrence. *ETFs for the Long Run.* John Wiley & Sons, 2008.

[27] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. "Deep learning". In: *Nature* 521 (2015), pp. 436–444.

[28] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. "Efficient Estimation of Word Representations in Vector Space". In: *Proceedings of Workshop at ICLR* (2013).

[29] Tomas Mikolov, Martin Karafiát, Lukáš Burget, Jan Černocký, and Sanjeev Khudanpur. "Recurrent Neural Network Based Language Model". In: *Proceedings of Interspeech* (2010), pp. 1045–1048.

[30] Karl Pearson. "On Lines and Planes of Closest Fit to Systems of Points in Space". In: *Philosophical Magazine* 2.11 (1901), pp. 559–572.

[31] Fabian Pedregosa, Gael Varoquaux, et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[32] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. "GloVe: Global Vectors for Word Representation". In: *Empirical Methods in Natural Language Processing (EMNLP).* 2014, pp. 1532–1543. URL: `http://www.aclweb.org/anthology/D14-1162`.

[33] Jan Peters and Stefan Schaal. "Policy Gradient Methods for Robotics". In: *IEEE International Conference on Intelligent Robots and Systems* (Nov. 2006), pp. 2219–2225.

[34] Martin L. Puterman. *Markov Decision Processes.* John Wiley and Sons, 1994.

[35] Bryan D. Ripley. *Pattern Recognition and Neural Networks.* Cambridge University Press, 2007.

[36] Frank Rosenblatt. "The Perceptron: A probabilistic model for information storage and organization in the brain". In: *Psychological Review* 65 (1958), pp. 386–408.

[37] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. "Learning representations by back-propagating errors". In: *Nature* 323 (1986), pp. 533–536.

[38] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. "Proximal Policy Optimization Algorithms". In: *arXiv* (2017). arXiv: `1707.06347`.

[39] John Schulman, Sergey Levine, Pieter Abbeel, Michael I. Jordan, and Philipp Moritz. "Trust Region Policy Optimization". In: *Proceedings of the* 31st *International Conference on Machine Learning* (2015).

[40] Scikit-learn. *3.2.4.3.4. sklearn.ensemble.ExtraTreesRegressor*. URL: `https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.ExtraTreesRegressor.html` (visited on 03/03/2020).

[41] Qiang Song, Saud Almahdi, and Steve Y. Yang. "Entropy based measure sentiment analysis in the financial market". In: *IEEE Symposium Series on Computational Intelligence (SSCI)* (2017).

[42] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. edition 2. MIT Press, Cambridge, 2018.

[43] Guido Van Rossum and Fred L. Drake. *Python 3 Reference Manual*. Scotts Valley, CA: CreateSpace, 2009.

[44] Ronald J. Williams. "Simple statistical gradient-following algorithms for connectionist reinforcement learning". In: *Machine Learning* 8 (1992), pp. 229–256.

[45] Ho C. Wu, Robert W. P. Luk, Kam F. Wong, and Kui L. Kwok. "Interpreting TF-IDF Term Weights as Making Relevance Decisions". In: *ACM Transactions on Information Systems* 26.13 (2008).

[46] Kaisheng Yao, Trevor Cohn, Katerina Vylomova, Kevin Duh, and Chris Dyer. *Depth-Gated LSTM*. 2015. arXiv: `1508.03790 [cs.NE]`.

[47] Mohammed J. Zaki and Wagner Meira Jr. *Data Mining and Analysis: Fundamental Concepts and Algorithms*. Cambridge University Press, 2014.