Escola Tècnica Superior d'Enginyeria ETSE-UV

MASTER'S DEGREE IN DATA SCIENCE

Master's Final Project

REINFORCEMENT LEARNING IN STOCK MARKET

AUTHOR :
PABLO CARRERA FLÓREZ DE QUIÑONES

SUPERVISOR(S):
VALERO LAPARRA PÉREZ-MUELAS
JORGE MUÑOZ MARÍ

SEPTEMBER 2020

Abstract

Reinforcement Learning is a very active field that has seen lots of progress in the last decade, however industry is still resistant to incorporate it to the automatization of its decision process. In this work we make a revision of the fundamentals of Reinforcement Learning reaching some state-of-the-art algorithms in order to make its implementation more profitable. As a practical application of this knowledge, we propose a Reinforcement Learning approach to Stock Market from scratch, developing an original agent-environment interface and comparing the performance of different training algorithms. Finally we run a series of experiments to tune our agent and test its performance.

Resumen

El Reinforcement Learning es un campo muy activo que ha sufrido grandes progresos en la última decada, sin embargo la industria todavía se muestra reacia a incorporarlo a la automatización de su proceso de toma de decisiones. En este trabajo hacemos una revisión de los fundamentos del Reinforcement Learning alcanzando el estado del arte en algunos de los algorithmos desarrollados para hacer su implementación más prometedora. Como aplicación práctica de este conocimiento proponemos una aproximacion desde cero al Mercado de Valores usando Reinforcement Learning, desarrollando una interfaz agent-entorno original y comparando el desempeño de diferentes algoritmos de entrenamiento. Finalmente realizamos una serie de experimentos para tunear y testear el desempeño de nuestro agente.

Resum

El Reinforcement Learning és un camp molt actiu que ha patit grans progressos en l'última dècada, no obstant això la indústria encara es mostra poc inclinada a incorporar-ho a l'automatització del seu procés de presa de decisions. En aquest treball fem una revisió dels fonaments del Reinforcement Learning aconseguint l'estat de l'art en alguns dels algoritmes desenvolupats per a fer la seua implementació més aprofitable. Com a aplicació pràctica d'aquest coneixement proposem una aproximació des de zero al Mercat de Valors usant Reinforcement Learning, desenvolupant una interfície agent-entorn original i comparant el resultat de diferents algoritmes d'entrenament. Finalment realitzem una sèrie d'experiments per a tunear i testear els retultats del nostre agent.

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

One of the key steps in the the expansion of Machine Learning to the industry is the change from the data modelling culture to the algorithm modeling culture [1]. This evolution implies to put aside the idea of statistical modelling in the sense of discovering the stochastic data-generating model behind a problem, and focus on the idea of statistical modelling in the sense of the search of best possible algorithm to reproduce or exploit the data. This change of mindset has proven to be very fruitful to solve novel real-world problems arising from massive data sets in very different domains such as finance (portfolio management, prediction of defaults and risk management) [2], law (document classification, prediction of litigation outcome and recommendation of resolutions) [3], medicine (automatic diagnosis, cardiology, radiology and smart electronic health records) [4], pharmaceutical industry (personalized treatment, drug discovery and clinical trial research) [5], physics (particle physics, cosmology, quantum many-body physics, quantum computing and material physics) [6] or sports (player performance, ticketing and strategy) [7].

In this sense, the two big paradigms of Machine Learning, Supervised Learning and Unsupervised Learning have been very active in last decades, in the academy and the industry, by the disruption of Deep Learning. But the third paradigm, Reinforcement Learning, that was lacking from attention, has been specially fruitful in the last years in the the academy, with rising expectations in industry now [8].

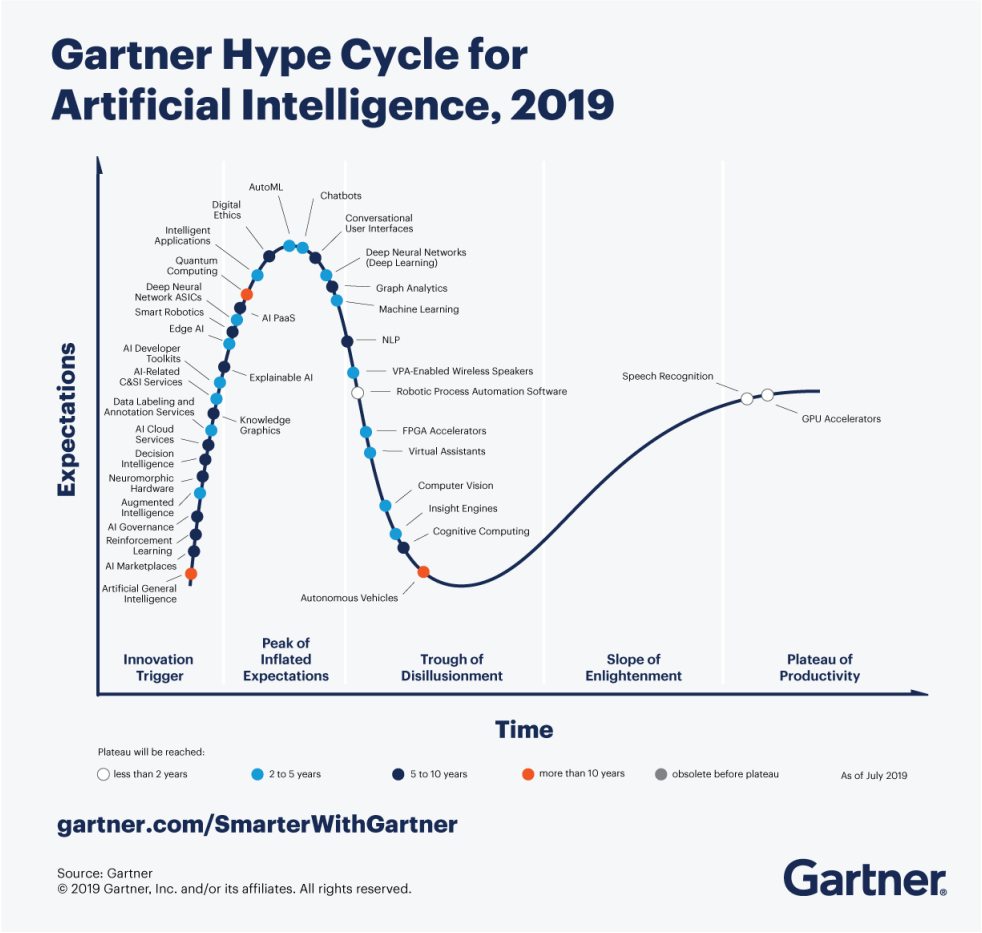

Figure 1: *Gartner Hype Cycle for Artificial Intelligence for 2019. We can see that the expectations on Reinforcement Learning are rising. Source: [8]*

Reinforcement learning consists in learning what to do, that is, how to map situations to actions, so as to maximize a numerical reward signal. In this sense, it is different from supervised learning, since we are not learning from a training set of labeled examples provided by a knowledgeable external supervisor, and from unsupervised learning, since we are not looking for the structure hidden in collections of unlabeled data. More formally, Reinforcement Learning consists of an agent interacting with an environment, this interactions occurs by environment showing an state to the agent and the agent taking an action of the set of possible actions available for this states, then the agent receives a reward indicating how good or bad was its decision. Then the fundamental problem of Reinforcement Learning is the trade-off between long-term rewards versus short-term reward.

In this sense, games provide a very adequate framework to develop the ideas of Reinforcement Learning. In games, the rules mediating the interaction between the player and the game, or between the player and other players tend to be very clear, allowing to get rid of the necessity of modelling the environment and rather focus on learning process [9]. So, Reinforcement Learning has evolved by solving classical games such as Backgammon, with the TD-gammon [10], Go, with AlphaGo [11] [12], or Chess, with AlphaZero [13]. These achievements also were translated to video games, such as the classic ATARI 2600 set of games [14] [15], with the recent achievement of the remaining game of Montezuma Revenge [16] that was still unsolved. Recently, a more generic version of these algorithms was released, muZero [17], begin capable of mastering both, classic games and ATARI games. In order to keep advancing, Reinforcement Learning took more complex problems, illustrated by more sophisticated games such as DOTA 2 [18], or Starcraft II with AlphaStar [19], which have more freedom of choice and involve more complicated long-term strategies. Solving Go and Starcraft II has been considered huge milestones in the history of Reinforcement Learning.

After all these achievements, it is reasonable to assume that Reinforcement Learning algorithms, combined with Deep Learning advances to learn representation of very challenge tasks, will slowly perform better and better in more ambiguous, real-life environments while choosing from an arbitrary number of possible actions, rather than from the limited options of a repeatable video game. In this sense, nowadays Reinforcement Learning has experiencing its jump to the industry with applications in traffic light control [20], bidding and advertising [21], resource management in computer clusters [22], recommendations systems [23], autonomous driving [24] [25] and many more [26].

One field that is of special interest for us is finance, or more concretely, the activity buying and selling financial instruments, also known as trading and explained more in detail in Section 2. The idea of automatizing trading has evolved from the quantitative methods from algorithmic trading to the use of Deep Learning to implement winning trading strategies, and now Reinforcement Learning seems to be getting popular in this field [27]. The aim of this work is then to explore and build an end-to-end application of the ideas of Reinforcement Learning to trading.

This work is structured as follows: in section 2 we introduce, at high level, the role and structure of financial markets, getting more deep in the specific case of the stock market; in section 3 we introduce, in more detail the learning paradigm known as Reinforcement Learning, taking special attention to its fundamentals, to then present some of the most popular state-of-the-art implementations of these ideas; in section 4 we use the ideas introduced in section 3 to develop an environment reproducing a simplified version of the stock market and some agents capable to operate in this environment; in section 5 we show the results of some experiments developed with the environment and the agents developed in section 4; finally, in section 6 we make a summary of this work, highlighting the key insights and proposing some lines of future work. We have published all our code in our GitHub [1]

[1] https://github.com/pablocarreraflorez/TFM-reinforcement-learning

2 Financial Markets

In this section we are providing an introduction to financial markets. First, we are going to define them, to explain their properties and actors and to divide them by the type trade they involve. Then we are going deeper into one of the most popular financial markets, the stock market and the characteristics of the assets involved in it. Finally, we are going to make a operational introduction to the activity known as trading and the different approaches to make profits with it.

2.1 Fundamentals

A financial market, as defined in [28] or [29], is a market in which people exchange, or trade, financial assets. Although the existence of these markets is not a necessary condition for the creation and exchange of a financial asset, since in principle these operations can be realized outside of them, nowadays in most economies financial assets are created and subsequently traded in some type of financial market.

The function of financial assets is to transfer funds from those who have surplus funds to those who need funds to invest in tangible assets, in such a way that they also redistribute the unavoidable risks associated with tangible assets among those seeking and those providing the funds. In this sense, financial markets have three properties:

- First, the interactions of buyers and sellers in a financial market determine the price of the traded asset. That is, the return on a financial asset is not given or fixed by prior in the market.
- Second, financial markets provide a mechanism for an investor to sell a financial asset. That is, they allow investors to get rid of the obligation of keeping an asset by transferring it to another investor.
- Third, financial markets reduce the cost of transacting. The search costs, related to looking for a potential buyer or seller, and information costs, related to the assessment of the financial merits of an asset, are almost totally avoided.

so from these properties surge many ways to classify financial markets:

- Nature of claim: we can distinguish between debt markets, where funds are borrowed and lent, and equity markets, where ownership of securities are issued and subscribed.
- Maturity of claim: we can distinguish between money markets, that allow firms to borrow funds on a short term basis, and capital markets, that allow firms to gain long-term funding to support expansion.
- Seasoning of claim: we can distinguish between primary markets, which deal with newly issued claims, and secondary markets, which deal with financial claims previously issued.
- Seasoning of claim: we can distinguish between cash markets, that trade assets directly, and derivatives instruments markets, that trade financial instruments based on an underlying asset.

2.2 Stock Market

Stocks, also known as equity securities or equities, represent ownership shares in a corporation [29]. Each share of a stock entitles its owner to one vote on any matters of corporate governance that are put to a vote at the corporation's annual meeting and to a share in the financial benefits of ownership when those earning are distributed in the form of dividends. There are two types of stocks, the so called common stocks and the preferred stocks. The key difference between these two forms lies in the degree to which the participate in the distribution of earnings and the priority going to each in the distribution of earnings, typically preferred stockholders are entitled to a fixed dividend before common stockholders may receive dividends. We will center our discussion in common stocks since they are by far more the most common type of equity. The two most important characteristics of common stock as an investment are

- **Residual claim**: this means that stockholders are the last in line of all those who have a claim on the assets and income of the corporation. For a firm not in liquidation, shareholders have claim to the part of operating income left over after interest and taxes have been paid. In a liquidation of the firm's assets the share-holders have a claim to what is left after all other claimants such as the tax authorities, employees, suppliers, bondholders, and other creditors have been paid.
- **Limited liability**: this means that the most shareholders can lose in the event of failure of the corporation is their original investment. Unlike owners of unincorporated businesses, whose creditors can lay claim to the personal assets of the owner, corporate shareholders may at worst have worthless stock. They are not personally liable for the firm's obligations.

Stocks are created by the necessity of companies to raise new capital to achieve growth. Through an initial public offering, or IPO, the shares of the company are sold to institutional and individual investors. This process, also known as floating, transforms a privately held company into a public company. An IPO is underwritten by one or more investment banks, who also arrange for the shares to be listed on one or more stock exchanges. When a company lists its securities on a public exchange, the money paid by the investing public for the newly-issued shares goes directly to the company (primary offering) as well as to any early private investors who opt to sell all or a portion of their holdings (secondary offerings).

Figure 2: *Biggest IPOs of history in terms of total monetary volume. We can see that the economic consequences of this type of operations are noticeable. Source: Statista.*

After the IPO, shares are traded freely in the open market and money passes between public investors, making the stock market a secondary market, but note that the company will not be affected by these movements and is never required to repay the capital to its public investors. Those investors must endure the unpredictable nature of the open market to price and trade their shares. In thi sense, for a common stock its total value, also known as the market capitalization of the stock, is the sum of the price of all the shares. Then, for a common stock investor, the return realized by holding one of these shares come from two sources:

- **Dividend payments**: dividends are distributions made by a corporations to its owners, typically in the form of cash or additional shares of the stock. The payment of dividends is not compulsory, usually young companies do not pay them, but as time goes on and they mature they start doing it.
- **Changes in the price of the stock**: while holding a share, if the price at a future date is higher than the purchase price there is a capital gain, and if the price at a future date is lower than the purchase price there is a capital loss. When a stockholder is calculating the return from holding a stock from the purchase date to a given point in time, it is actually calculating the gain or loss that he will get in case of selling the stock at this time.

There is an additional very popular way to make profits from a stock, known as short selling. Short selling, usually referred as to take a short position or be in short, differs from the traditional operation of buying a stock, usually refereed as to take a long position to be in long, in the fact that, instead of taking profit from the rise of the stock's prices, we will take profits from the fall of the stock's price. The operation of short selling has two steps, in the first step, the short seller borrow a given quantity of shares from a broker and sells them to a third party; then, in the second step, the short seller rebuys the shares from the third party and return them back to the broker. If the price of the stock has fallen between the time of the initial sale and the time of he devolution, the equivalent the investor will have made a profit equal to the of prices.

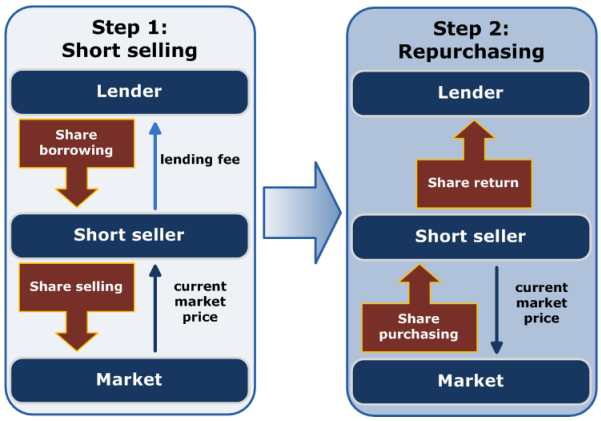

Figure 3: *Schematic representation of short selling. Source: Investorpedia.*

The desire of stockholders to trade their shares has led to the establishment of stock exchanges, organizations which provide marketplaces for trading shares and other derived financial products. Those exchanges can be physical locations, such as the New York Stock Exchange (NYSE), or digital platforms, such as the NASDAQ. The biggest stock exchange in Spain is the Bolsa de Madrid, followed by the Borsa de Barcelona. These markets, according to the properties of the equities they trade are classified as secondary capital markets. Each stock exchange imposes its own listing requirements upon companies that want to be listed on that exchange. Such conditions may include minimum number of shares outstanding, minimum market capitalization, and minimum annual income.

Symbols	Open	Low	Close	High	Volume
ANA	94.400	93.450	93.800	94.400	36596.000
ACX	9.950	9.920	10.045	10.045	240375.000
ACS	35.070	35.020	35.650	35.800	396662.000
AENA	171.050	170.500	170.500	172.750	73393.000
AMS	72.700	72.200	72.800	73.300	148692.000
MTS	15.600	15.510	15.620	15.650	220308.000
SAB	1.038	1.034	1.040	1.046	5437795.000
SAN	3.720	3.685	3.730	3.743	19491991.000
BKIA	1.890	1.885	1.903	1.903	2016010.000
BKT	6.550	6.520	6.532	6.572	564270.000
BBVA	4.995	4.953	4.983	5.004	5624533.000
CABK	2.785	2.763	2.798	2.798	8844742.000
CLNX	38.480	37.840	38.370	38.870	221575.000
CIE	21.180	21.000	21.080	21.380	67281.000
ENG	23.040	22.690	22.740	23.110	504112.000
ENC	3.668	3.612	3.670	3.718	731818.000
ELE	24.200	23.730	23.790	24.200	676458.000
FER	26.550	26.500	26.970	26.970	1125539.000
GRF	31.600	31.290	31.430	31.640	198479.000
IAG	7.300	7.182	7.220	7.362	877608.000
IBE	9.216	9.174	9.180	9.250	6724758.000
ITX	31.520	31.160	31.450	31.710	726006.000
IDR	10.100	10.080	10.180	10.250	150119.000
COL	11.200	11.190	11.360	11.450	198963.000
MAP	2.382	2.360	2.360	2.402	1876403.000
TL5	5.520	5.504	5.660	5.660	292443.000
MEL	7.890	7.850	7.860	7.975	105154.000
MRL	12.650	12.560	12.790	12.790	199986.000
NTGY	22.610	22.400	22.400	22.710	742966.000
REE	18.115	17.860	17.925	18.115	411070.000
REP	14.100	13.930	13.930	14.100	2221149.000
SGRE	15.500	15.500	15.635	15.695	441154.000
TRE	23.400	23.300	23.800	23.800	107676.000
TEF	6.293	6.200	6.227	6.300	7967845.000
VIS	47.900	47.100	47.100	47.900	19412.000

Table 1: *Daily values for prices and volumes for the stocks composing the IBEX 35 as of December 31, 2019. IBEX 35 is the benchmark stock market index of the Bolsa de Madrid, Spain's principal stock exchange. It is a market capitalization weighted index comprising the 35 most liquid Spanish stocks traded in the Madrid Stock Exchange and is reviewed twice annually. Source: own*

Nowadays stock trading is mostly electronic. This has been possible due to the liberalization of the markets and the technological advances for monitoring the markets and executing orders, which also has enforced the competitiveness between markets, lowering the costs and making trading available to the general public and the institutions.

2.3 Trading

As we said before, the globalization of stock market has made it available to both, the particular investor and the institutional investor. When an investor wants to buy or sell a share of common stock, the price and conditions under which the order is to be executed must be communicated to a broker, then the broker arranges the trade and charges a commission to the

investor. These orders can be of many types:

- **Market order**: is a buy or sell order that are to be executed immediately at current market price. A buy order is made at bid price (best offered price) and a sell order is made at ask price (best asked price). These is a gap between both prices known as bid-ask spread.
- **Conditional order**: prices can change between the time a order is made and the time a order is executed, so investors also may place orders specifying prices at which the are willing to buy or sell a share.

All these orders are included in the order book, which constitutes a record of the interest of buyers and sellers in a particular financial instrument, in this case stock shares. Then, a matching engine uses the book to determine which orders can be fully or partially executed. Consequently, the price of a stock fluctuates fundamentally due to the theory of supply and demand. The usual tool for studying these fluctuations are candlestick diagrams, such the one in Figure 4, which represent the open, close, low and high values for the price of a stock in a given time lapse.

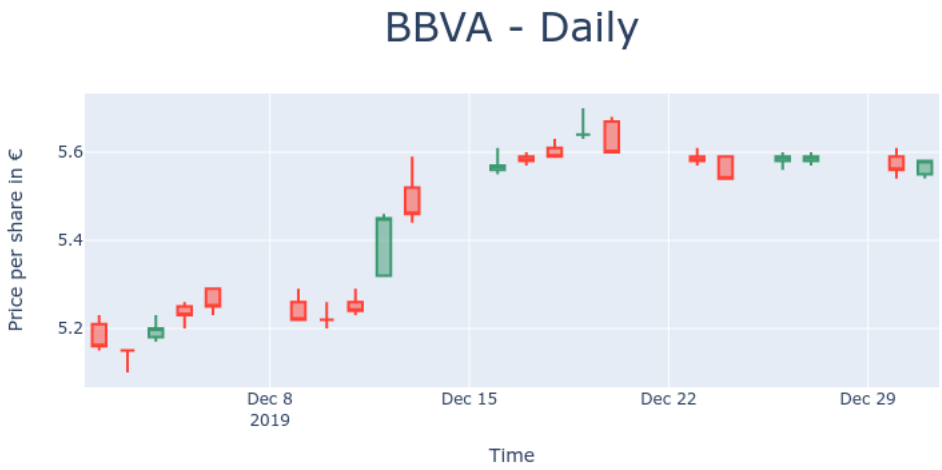

Figure 4: *Example of candlestick diagram for a stock. Data shown correspond to BBVA stock from December 2019. Each box, known as candlestick, contains lots of information. If the box is red it means that the closing price is lower than the opening price, consequently the lower edge of the box denotes the closing price of the stock this day, and the upper box denotes the opening price. If the box is green it means that the closing price is higher than the opening price, consequently the lower edge of the box denotes the opening price of the stock this day, and the upper box denotes the closing price. The lowest whisper, called lower shadow, always denotes the lowest price of the stock this day, and the upper whisper, called upper shadow, always the highest price. Source: own.*

In addition to supply and demand, there are many factors that influence the demand for a particular stock and the fields of fundamental analysis and technical analysis attempt to understand market conditions that lead to price changes in order to make profits. These two fields represent two different approaches to the market:

- **Fundamental analysis**: it consists on the analysis of the financial states of a company in order to determine its value. In this context stands out the idea of value investing [30], that supports that markets may incorrectly price a stock in the short run but the correct price will eventually be reached at some point. Under this assumption, investors can make profits

by buying/selling the wrongly priced stocks and then waiting for the market to reprice the security to sell/buy it and gain profit from the difference.

- **Technical analysis**: it consists on the use use of past market data, such as price and volume, to develop forecasts about the direction of prices [31]. It relies heavily in the use of candlestick diagrams, as the one shown in Figure 4, and techniques of time-series analysis, but it also exploits more advanced techniques from other fields, such as statistics or signal analysis.

These two approaches were a hot topic for many years, but now there is some consensus in the fact that stock market's prices are essentially unpredictable. This is partially based on the efficient-market hypothesis [32], which states that all the available information of an asset is contained in its prices, which makes impossible to "beat the market" consistently since these prices will only react to new, unavailable, information. However, some of the methods deviated in these approaches can still be used to take advantage from other investors in the market and make profits. This idea led to the growth of a new approach to trading that reaped lot of success, algorithmic trading.

Algorithmic trading consists in the application of the ideas provided by fundamental analysis and technical analysis to automate trading strategies using computer programs [33] [34]. These algorithms can handle much more information at the same time, and much faster, than a human trader, so it can implement a set of programmed rules very effectively and with high margin of benefits. Some examples of these trading strategies can be:

- **Trend-following strategies**: some popular strategies follow trends in moving averages, channel breakouts, price level movements, and related technical indicator, so trades are done based on the occurrence of desirable trends. The typical strategy consists in observing the ratio between 50-day and 200-day moving averages in order to sell if it is lower than or to buying if it is higher than 1.
- **Arbitrage opportunities**: some other popular strategies consist in, given two markets, buy a stock at a lower price in one market and simultaneously selling it at a higher price in another market in order to keep the price differential as a risk-free profit. Implementing an algorithm to identify such price differentials and placing the orders efficiently allows profitable opportunities.
- **Mean reversion**: this strategy is based on the concept that the high and low prices of an asset are a temporary phenomenon that revert to their mean value periodically. Then, identifying and defining a price range allows trades to be placed automatically when the price of an asset breaks in and out of its defined range.
- **Volume-weighted and Time-weighted average price**: VWAP and TWAP strategies consist in breaking up a large order and release dynamically determined smaller chunks of the order to the market using stock-specific historical volume profiles in the first case, and using evenly divided time slops, looking for a minimization of the market impact.

It is important to note the capital importance of algorithmic trading in the markets. In this sense, it is estimated that in 2010 about the 80% of the trades were done by machines [29]. It is also important to note that the automation of the trading task lead to a huge increase in the velocity of the operations, even reaching the limit of miliseconds and conforming what was know as High Frequency Trading, which is estimated to comprise about the 60% of the trades [29].

2.4 Machine Learning

Taking into account the trend to change from data modelling culture to the algorithmic modelling culture [1], Machine Learning seems to be the natural evolution of algorithmic trading to keep making progress in the field. As we have seen, Algorithmic Trading involves to introduce a set of rules, or strategies, into a computer, as in traditional programming, and then let the algorithm operate with them. Machine Learning then emerges as the natural way to the application of these strategies more flexible, and even breaking the idea of having to set a strategy at all. Some of the applications of Machine Learning in the last years have been:

- **Stock price forecasting**: the prediction of stock prices is the most classic problem. Despite efficient market hypothesis states that it is not possible to predict stock prices and that stocks behave in random walk manner, technical analysts believe that most information about the stocks are reflected in recent prices and so if trends in the movements are observed then prices can be easily predicted. In addition, stock market's movements are affected by many macro-economical factors such as political events, firms' policies, general economic conditions, commodity price index, bank rates, bank exchange rates, investors' expectations, institutional investors' choices, movements of other stock markets, psychology of investors, ... The literature in this topic is extensive [35], [36], [37], but the publications of machine learning techniques used to forecast stock market movements can be categorised according to the machine learning technique used (ARIMA models, Suport Vector Machines, Neural Networks, Genetic Algorithms, ...), the forecasting timeframe (minutely, hourly, daily, monthly, ...), the input variables used (lagged index data, exchange rates, candlestick values, unemployment rates, ...), and the evaluation techniques employed. It is found that there is some consensus between researchers stressing the importance of stock index forecasting, since it gives an overall picture of economy, and the positive results of the use of Neural Networks because of their ability to learn very nonlinear relations.
- **Portfolio optimization**: another classical problem is the portfolio optimization, which describes the process of selecting the best portfolio of stocks out of the set of all portfolios being considered, according to some objective such as the maximization of expected return or the minimization of financial risk [38]. Some noticeable approaches to this problem can be the development of eigenporfolio theory, which compute orthogonal varying portfolios using Principal Component Analysis [39], or hierarchical risk parity models, which exploits hierarchical clustering algorithms to find clusters of the assets to re-allocate risk over them recursively [40]. These approaches generate portfolios with such a high quantity of assets that make unfeasible its treatment by human traders, but can get acceptable results when managed automatically.
- **Sentiment analysis**: another typical problem is the analysis of external information to make buy and sell decisions, even if this information is not used directly to predict the stock price it is proven to be a useful approach. This information is usually extracted from news websites [41] or social networks [42] and analyzed using techniques from Natural Language Processing, implemented by state-of-the-art Deep Learning techniques [43]. There is some consensus over the fact that this information turns out to be more useful, not by itself, but when used to support other more straightforward approaches to the stock market such of stock price forecasting.

In this sense, Reinforcement Learning constitutes another step forward in the direction of algorithmic culture [1], allowing, in principle, to combine the prediction and the portfolio management task in one integrated step, thereby closely aligning the machine learning problem with the objectives a human investor in a more organic way [27] [44]. At the same time, important constraints, such as transaction costs, market liquidity, and the investor's degree of risk-aversion, can be conveniently taken into account since Reinforcement Learning tries to learn how to interact with the market, constituting a more natural approach to the problem of trading.

3 Reinforcement Learning

In this section we are going to make a operational introduction to the field known as Reinforcement Learning [45]. First we are going to introduce Markov Decision Processes, which are the classical formalization of sequential decision making, where actions influence not just immediate rewards, but also subsequent situations. Then we are going to present the classical solutions to these problems in terms of Dynamic Programming, Monte Carlos methods and its combination in the form of Temporal-Difference methods. Finally, we are going to introduce the use of approximate solutions to these methods, which allow us to use state-of-the-art algorithms, such as Neural Networks, to solve efficiently real-world problems.

3.1 Markov Decision Processes

Now we are going to introduce Markov Decision Processes, or MDPs. MDPs are meant to be a straightforward framing of the problem of learning from interaction to achieve a goal [46]. The learner and decision maker is called the agent. The thing it interacts with, comprising everything outside the agent, is called the environment. These interact continually, the agent selecting actions and the environment responding to these actions and presenting new situations to the agent. The environment also gives rise to rewards, special numerical values that the agent seeks to maximize over time through its choice of actions.

3.1.1 Agent-environment interface

More formally, the agent and environment interact with each other in a sequence of discrete time-steps, $t = 0, 1, 2, 3, ...$ At each time step t, the agent receives some representation of the environment's state, $S_t \in \mathcal{S}$, and on that basis selects an action, $A_t \in \mathcal{A}(S_t)$. One time step later, in part as a consequence of its action, the agent receives a numerical reward, $R_{t+1} \in \mathcal{R}$, and finds itself in a new state S_{t+1}. Then, the MDP and agent give rise to a sequence, or trajectory, that begins like this $S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, ...$

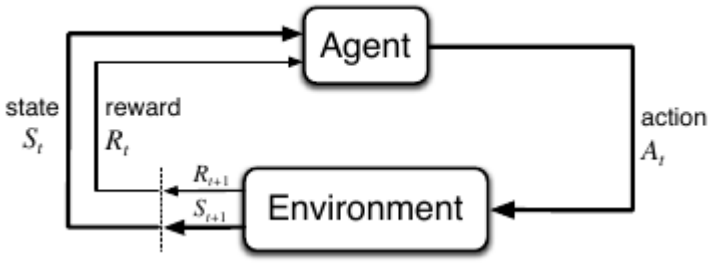

Figure 5: *Schema of the agent-environment interaction in a MDP. Source: [45]*

In a finite MDP, the sets of states $\mathcal{S}$, actions $\mathcal{A}$, and rewards $\mathcal{R}$ all have a finite number of elements. In this case, the random variables R_t and S_t have well defined discrete probability distributions dependent only on the preceding state and action:

$$p(s', r|s, a) \doteq \mathrm{P}(S_t = s', R_t = t|S_{t-1} = s, A_{t-1} = a) \tag{1}$$

so the function $p : \mathcal{S} \times \mathcal{R} \times \mathcal{S} \times \mathcal{A} \longrightarrow [0, 1]$, that defines the dynamics of the MDP, is an ordinary deterministic function of four arguments. And since it involves a conditional probability distribution, it satisfies that

$$\sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p(s', r|s, a) = 1 \tag{2}$$

which implies that the probability of each possible value for S_t and R_t depends only on the immediately preceding state and action, S_{t-1} and A_{t-1}, and not at all on earlier states and actions. This is best viewed a restriction not on the decision process, but on the state, which must include information about all aspects of the past agent-environment interaction that make a difference for the future. If it does, then the state is said to have the Markov property.

3.1.2 Rewards and returns

In reinforcement learning, the goal of the agent is formalized in terms of a special signal, called the reward, passing from the environment to the agent. At each time step t, the reward is a real number $R_t \in \mathbb{R}$. Informally, the agent's goal is to maximize the total amount of reward it receives, that is, maximizing not immediate reward, but cumulative reward in the long run. If we want it to solve a problem for us, we must provide rewards to it in such a way that in maximizing them the agent will also achieve our goals. It is thus critical that the rewards we set up truly indicate what we want to accomplish.

More formally, we seek to maximize the expected return, where the return is defined as some specific function of the reward sequence $G_t \doteq f(R_{t+1}, R_{t+2}, ..., R_T)$, where T is a final time step. In the simplest case the return can be defined as the sum of the rewards,

$$G_t \doteq R_{t+1} + ... + R_T = \sum_{k=t+1}^{T} R_k \tag{3}$$

so this approach makes sense in applications in which there is a natural notion of final time step T, that is, when the agent-environment interaction breaks naturally into subsequences, which are usually called episodes. Each episode ends in a special state called the terminal state S_T, followed by a reset to a standard starting state. Then the next episode begin independently of how the previous one ended. Thus, the episodes can all be considered to end in the same terminal state, with different rewards for the different outcomes. Tasks with episodes of this kind are called episodic tasks.

However, in many cases the agent-environment interaction does not break naturally into identifiable episodes, but goes on continually without limit. Then the above return formulation is problematic for continuing tasks because the final time step would be $T = \infty$ and then also the return. We can solve this problem by introducing the concept of discounting, with this the agent tries to select actions so that the sum of the discounted rewards it receives over the future is maximized. In particular, it chooses A_t to maximize the expected discounted return:

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... = \sum_{k=t+1}^{\infty} \gamma^{k-(t+1)} R_k \tag{4}$$

where $0 \leq \gamma \leq 1$ is called the discount rate. If $\gamma < 1$, the infinite sum has a finite value as long as the reward sequence is bounded, and in the extreme case of $\gamma = 0$ the agent will be concerned only with maximizing immediate rewards. As γ approaches 1, the return objective takes future rewards into account more strongly, that is, the agent becomes more farsighted. In this sense is important to note the following recursive rule

$$G_t = R_{t+1} + \gamma G_{t+1} \tag{5}$$

It is useful to establish one notation that enables us to talk precisely about both cases simultaneously. This can be achieved by considering episode termination to be the entering of a special absorbing state that transitions only to itself and that generates only rewards of zero. Then we get the same return whether we sum over the first T rewards or over the full infinite sequence.Thus, we can define the return, in general, using the convention of omitting episode numbers when they are not needed, as

$$G_t \doteq \sum_{k=t+1}^{T} \gamma^{k-(t+1)} R_k \tag{6}$$

where we include the possibility of $T = \infty$ and $\gamma = 1$, but not both at the same time.

3.1.3 Policies and value functions

Reinforcement learning algorithms usually involve estimating how good it is for the agent to be in a given state, which is defined in terms of the expected return. The rewards the agent can expect to receive in the future depend on what actions it will take, so value functions are defined with respect to particular ways of acting, called policies. More formally, a policy is a mapping $\pi : \mathcal{S} \times \mathcal{A} \longrightarrow [0, 1]$ from states and actions to probabilities of selecting each possible action, if the agent is following policy π at time t, then $\pi_t(a|s)$ is the probability that $A_t = a$ is chosen if we are in $S_t = s$. Like p, π is an ordinary function which defines a probability distribution over a $A(s)$ for $s \in S$. Reinforcement learning methods specify how the agent's policy is changed as a result of its experience.

Then, the value of a state, that is, of being in that state, is associated to a given policy. This is formalized by the definition of the value function v_π for policy π. The value of a state s under a policy π, denoted $v_\pi(s)$, is the expected return when starting in s and following π thereafter

$$v_\pi(s) \doteq \mathbb{E}_\pi \left[G_t | S_t = s \right] \tag{7}$$

for $s \in \mathcal{S}$. Note that the value of the terminal state, if any, is always zero.

Similarly, we define the value of taking action a in state s under a policy π, denoted $q_\pi(s, a)$, as the expected return of taking the action a in the state s following policy π

$$q_\pi(s, a) \doteq \mathbb{E}_\pi \left[G_t | S_t = s, A_t = a \right] \tag{8}$$

and we call q_π the action-value function for policy π. By definition the following relation is satisfied

$$v_\pi(s) \doteq \mathbb{E}_\pi \left[G_t | S_t = s \right] \tag{9}$$

$$\implies v_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \mathbb{E}_\pi \left[G_t | S_t = s, A_t = a \right] \tag{10}$$

$$\implies v_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) q_\pi(s, a) \tag{11}$$

A fundamental property of value functions is that they satisfy recursive relationships similar to that which we have already established for the return

$$v_\pi(s) \doteq \mathbb{E}_\pi \left[G_t | S_t = s \right] \tag{12}$$

$$\implies v_\pi(s) = \mathbb{E}_\pi \left[R_{t+1} + \gamma G_{t+1} | S_t = s \right] \tag{13}$$

$$\implies v_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p(s', r|s, a) \left[r + \gamma \mathbb{E}_\pi \left[G_{t+1} | S_t = s' \right] \right] \tag{14}$$

$$\implies v_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p(s', r|s, a) \left[r + \gamma v_\pi(s') \right] \tag{15}$$

which is known as the Bellman equation [47] and represent the relation between the value of an state and the subsequent states. Then, the value function $v_\pi(s)$ can be seen as the unique solution to the Bellman equation [45]. For the action-value function

$$q_\pi(s, a) \doteq \mathbb{E}_\pi \left[G_t | S_t = s, A_t = a \right] \tag{16}$$

$$\implies q_\pi(s, a) = \mathbb{E}_\pi \left[R_{t+1} + \gamma G_{t+1} | S_t = s, A_t = a \right] \tag{17}$$

$$\implies q_\pi(s, a) = \sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p(s', r|s, a) \left[r + \gamma v_\pi(s') \right] \tag{18}$$

$$\implies q_\pi(s, a) = \sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p(s', r|s, a) \left[r + \sum_{a' \in \mathcal{A}} \pi(a'|s') q_\pi(s', a') \right] \tag{19}$$

we obtain a similar result, which is denoted as the Bellman equation for the action-value function. Again, the action-value function $q_\pi(s, a)$ is the unique solution to the Bellman equation [45].

3.1.4 Optimality

Solving a reinforcement learning task means, roughly, finding a policy that achieves the most reward over the long run. For finite MDPs value functions define a partial ordering over policies, a policy π is defined to be better than or equal to a policy π' if its expected return is greater than or equal to that of π for all states. In other words, $\pi > \pi'$ if and only if $v_\pi(s) > v_{\pi'}(s)$ for all $s \in \mathcal{S}$. There is always at least one policy that is better than or equal to all other policies, the optimal policy. Although there may be more than one, we denote all the optimal policies by π_*. They share the same state-value function, called the optimal state-value function,

$$v_*(s) \doteq \max_\pi v_\pi(s) \tag{20}$$

for all $s \in \mathcal{S}$. Optimal policies also share the same optimal action-value function

$$q_*(s,a) \doteq \max_\pi q_\pi(s,a) \tag{21}$$

for all $s \in \mathcal{S}$ and $a \in \mathcal{A}$.

Because $v_*(s)$ is the value function for a policy, it must satisfy the self-consistency condition given by the Bellman equation for state values, but because it is the optimal one its consistency condition can be written in a special form without reference to any specific policy. Intuitively, the Bellman optimality equation expresses the fact that the value of a state under an optimal policy must equal the expected return for the best action from that state,

$$\begin{aligned}
&v_*(s) = \max_a q_*(s,a) && (22)\\
\implies &v_*(s) = \max_a \mathbb{E}\left[G_t | S_t = s, A_t = a\right] && (23)\\
\implies &v_*(s) = \max_a \mathbb{E}\left[R_{t+1} + \gamma G_{t+1} | S_t = s, A_t = a\right] && (24)\\
\implies &v_*(s) = \max_a \mathbb{E}\left[R_{t+1} + \gamma v_*(S_{t+1}) | S_t = s, A_t = a\right] && (25)\\
\implies &v_*(s) = \max_a \sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p(s', r | s, a)\left[r + \gamma v_*(s')\right] && (26)
\end{aligned}$$

or in terms of the action-value function

$$\begin{aligned}
&q_*(s,a) = \mathbb{E}\left[G_t | S_t = s, A_t = a\right] && (27)\\
\implies &q_*(s,a) = \mathbb{E}\left[R_{t+1} + \gamma G_{t+1} | S_t = s, A_t = a\right] && (28)\\
\implies &q_*(s,a) = \mathbb{E}\left[R_{t+1} + \gamma v_*(S_{t+1}) | S_t = s, A_t = a\right] && (29)\\
\implies &q_*(s,a) = \mathbb{E}\left[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a') | S_t = s, A_t = a\right] && (30)\\
\implies &q_*(s,a) = \sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p(s', r | s, a)\left[r + \gamma \max_{a'} q_*(s', a')\right] && (31)
\end{aligned}$$

so for finite MDPs the Bellman optimality equation for v_*, or q_*, also has a unique solution [45]. If the dynamics p of the environment are known, then in principle one can solve this system of equations using any one of a variety of methods for solving systems of nonlinear equations.

Once one has v_*, it is relatively easy to determine an optimal policy. For each state s, there will be one or more actions at which the maximum is obtained in the Bellman optimality equation. Any policy that assigns nonzero probability only to these actions is an optimal policy. We can also think of this as a one-step search, if we have the optimal value function, v_* , then the actions that appear best after a one-step search will be optimal actions. So, by means of v_* , the optimal

expected long-term return is turned into a quantity that is locally and immediately available for each state. Hence, a one-step-ahead search yields the long-term optimal actions.

Having q_* makes choosing optimal actions even easier, the agent does not even have to do a one-step-ahead search, for any state s, it can simply find any action that maximizes $q_*(s, a)$. The action-value function effectively caches the results of all one-step-ahead searches. It provides the optimal expected long-term return as a value that is locally and immediately available for each state–action pair.

Explicitly solving the Bellman optimality equation provides one route to finding an optimal policy, and thus to solving the reinforcement learning problem. However, this solution is rarely directly useful, it leads to an exhaustive search, looking ahead at all possibilities, computing their probabilities of occurrence and their desirabilities in terms of expected rewards. This solution relies on at least three assumptions that are rarely true in practice: we accurately know the dynamics of the environment, we have enough computational resources to complete the computation of the solution and the system satisfies the Markov property. But for the kinds of tasks in which we are interested, one is generally not able to implement this solution exactly because various combinations of these assumptions are violated. Then many reinforcement learning methods can be understood as approximately solving the Bellman optimality equation, using actual experienced transitions in place of knowledge of the expected transitions.

3.2 Dynamic Programming

The term dynamic programming (DP) refers to a collection of algorithms that can bused to compute optimal policies given a perfect model of the environment as a MDP. Classical DP algorithms are of limited utility in reinforcement learning both because of their assumption of a perfect model and because of their great computational expense, but they are still important theoretically to set the basis of the learning process.

First we consider how to compute the state-value function v_π for an arbitrary policy π, this is called policy evaluation. If the environment's dynamics are completely known, then the Bellman equation is a system of $|\mathcal{S}|$ simultaneous linear equations in $|\mathcal{S}|$ unknowns (the $v_\pi(s)$, $s \in \mathcal{S}$). So in principle, its solution is a straightforward computation, but, for our purposes, iterative solution methods are most suitable. Consider then a sequence of approximate value functions $v_0, v_1, v_2, ...$, each mapping $\mathcal{S}^+$ to $\mathbb{R}$. The initial approximation v_0 is chosen arbitrarily (except that the terminal state, if any, must be given value 0), and each successive approximation is obtained by using the Bellman equation as an update rule

$$v_{k+1}(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p(s', r|s, a) \left[r + \gamma v_k(s') \right] \tag{32}$$

for all $s \in \mathcal{S}$. Each iteration updates the value of every state once to produce the new approximate value function. Clearly, $v_k = v_\pi$ is a fixed point for this update rule because the Bellman equation for v_π assures us of equality in this case. So, it is usually applied until the difference between two consecutive iterations of the value function estimate go beyond a given threshold θ.

Then, we can use the value function for a policy to find better policies, which is called policy improvement. Suppose we have determined the value function v_π for an arbitrary deterministic policy π. For some state s we would like to know whether or not we should change the policy to deterministically choose an action $a \neq \pi(s)$. Consider selecting a in s and thereafter following the existing policy, π. The value of this way of behaving is $q_\pi(s, a)$ and then, the key criterion is to

check whether this is greater than or less than $v_\pi(s)$. If it is greater, that is, if it is better to select a once in s and thereafter follow π than it would be to follow π all the time, then one would expect it to be better still to select a every time s is encountered, and that the new policy would in fact be a better one overall. A natural extension is to consider changes at all states and to all possible actions, selecting at each state the action that appears best according to $q_\pi(s,a)$. In other words, to consider the new greedy policy π' given by

$$\pi'(s) = \text{argmax}_a \left[q_\pi(s,a)\right] \tag{33}$$

$$\implies \pi'(s) = \text{argmax}_a \left[\sum_{r\in\mathcal{R}} \sum_{s'\in\mathcal{S}} p(s',r|s,a)\left[r + v_\pi(s')\right]\right] \tag{34}$$

where argmax_a denotes the value of a at which the expression that follows is maximized (with ties broken arbitrarily). Suppose the new greedy policy π' is as good as, but not better than, the old policy π, then $v_\pi = v_{\pi'}$, and then for all $s \in \mathcal{S}$:

$$v_{\pi'}(s) = \max_a \left[\mathbb{E}\left[R_{t+1} + \gamma v_{\pi'}(S_{t+1})|S_t = s, A_t = a\right]\right] \tag{35}$$

$$\implies v_{\pi'}(s)' = \max_a \left[\sum_{r\in\mathcal{R}} \sum_{s'\in\mathcal{S}} p(s',r|s,a)\left[r + v_\pi(s)\right]\right] \tag{36}$$

but this is the same as the Bellman optimality equation, and therefore, $v_{\pi'}$ must be v_* , and both π and π' must be optimal policies. Policy improvement thus must give us a strictly better policy except when the original policy is already optimal.

Finally, once a policy π has been evaluated to get v_π, and it is used to yield a better policy π', we can then compute $v_{\pi'}$ and improve it again to yield an even better π''. We can thus obtain a sequence of monotonically improving policies and value functions

$$\pi_0 \xrightarrow{E} v_{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} v_{\pi_1} \xrightarrow{I} ... \xrightarrow{I} \pi_* \xrightarrow{E} v_{\pi_*}$$

where $\xrightarrow{E}$ denotes a policy evaluation and $\xrightarrow{I}$ denotes a policy improvement. Each policy is guaranteed to be a strict improvement over the previous one (unless it is already optimal). Because a finite MDP has only a finite number of policies, this process must converge to an optimal policy and optimal value function in a finite number of iterations. This way of finding an optimal policy

is called policy iteration.

Algorithm 1: Policy Iteration

Inputs: θ
Initialize policy function $\pi(s)$ with random values
Initialize value function $V(s, a)$ with random values
Policy Evaluation:
while $\Delta > \theta$ **do**
  $\Delta \longleftarrow 0$
  for *each* s **do**
    $v \longleftarrow V(s)$
    $V(s) \longleftarrow \sum_{s',r} p(s', r|s, \pi(s))[r + \gamma V(s')]$
    $\Delta \longleftarrow \max(\Delta, |v - V(s)|)$
  end
end
Policy Improvement:
$policy\_stable \longleftarrow True$
for *each* s **do**
  $old\_action \longleftarrow \pi(s)$
  $\pi(s) \longleftarrow \text{argmax}_a \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]$
  $\Delta \longleftarrow \max(\Delta, |v - V(s)|)$
  if $\pi(s) \neq old\_action$ **then**
    $policy\_stable \longleftarrow False$
  end
end

However, although we have complete knowledge of the environment, it would not be easy to apply these DP methods to compute the value function. DP methods require the distribution of next events, all of the probabilities must be computed before DP can be applied, and such computations are often complex and error-prone.

3.3 Monte Carlo methods

One solution to the problem of building a model of the environment can be found in Monte Carlo (MC) methods. MC methods are a broad class of algorithms that rely on repeated random sampling to obtain numerical results, the underlying concept is to use randomness to solve problems that might be deterministic in principle. In this sense MC methods, for Reinforcement Learning, require only experience-sample sequences of states, actions, and rewards from actual or simulated interaction with an environment in order to average over them. So, despite a model is required, the model need only generate sample transitions, not the complete probability distributions of all possible transitions that is required for DP.

First, as in DP, suppose we wish to estimate $v_\pi(s)$, the value of a state s under policy π, given a set of episodes obtained by following π and passing through s. Each occurrence of state s in an episode is called a visit to s. Then, the first-visit Monte Carlo methods estimate $v_\pi(s)$ as the average of the returns following first visits to s in each episode, and the every-visit Monte Carlo methods average the returns following all visits to s in each episode. However, since a model of environment is not available, it is particularly useful to estimate action values rather than state values. So we are doing this computation for $q_\pi(s, a)$ taking into account visits to state-action pairs instead of states alone.

Then, after this policy evaluation, policy improvement is done by making the policy π

greedy with respect to the current action value function q_π. For any action-value function q, the corresponding greedy policy is the one that, for each state s deterministically chooses an action with maximal action-value. Then policy improvement then can be done by constructing each π_{k+1} as the greedy policy with respect to q_k.This leads to a series of evaluations and improvements, analogous to the ones in the policy iteration algorithm in DP.

Algorithm 2: Every-visit Monte Carlo

Inputs: M
Initialize policy function $\pi(s)$ with random values
Initialize action-value function $Q(s, a)$ with random values
Initialize $returns(s, a)$ with an empty list
for $episode = 1 : M$ **do**
  Choose s_0, a_0 randomly
  Generate an episode from s_0, a_0 following π: $s_0, a_0, r_1, ..., s_{T-1}, a_{T-1}, r_T$
  $G \longleftarrow 0$
  for $t = T - 1 : 0$ **do**
    $G \longleftarrow \gamma G + r_{t+1}$
    Append G to $returns(s, a)$
    $Q(s_t, a_t) \longleftarrow$ average$(returns(s, a))$
    $\pi(s_t) \longleftarrow \text{argmax}_a Q(s_t, a)$
  end
end

3.4 Temporal-Difference Methods

Temporal-Difference (TD) learning is a combination of Monte Carlo (MC) ideas and Dynamic Programming (DP) ideas. On the one hand, like MC methods, TD methods can learn directly from raw experience without a model of the environment's dynamics. And, on the other hand, like DP, TD methods update estimates based in part on other learned estimates, without waiting for a final outcome.

The problem with MC methods is that the updates of $V(S_t)$, that serve as a estimation of $v(s)$, are done using the expected return G_t as a target, that is,

$$V(S_t) \longleftarrow V(S_t) + \alpha[G_t - V(S_t)]$$

so we have to wait until the end of the episode to compute the value of the returns G_t that allow us to update the values of $V(S_t)$. The idea beyond TD methods is not to wait until the ed of the episode but to make updates in each step, so we can use the recurrence property of the returns to perform a more useful update such as

$$V(S_t) \longleftarrow V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)]$$

which constitute the simplest TD method, the TD(0). This can be complicated including more steps, constituting the so called $T(\lambda)$ methods [45]. With this we get the best of both approaches, we get the online and fully-incremental computational power of DP methods, and the lack of necessity of a model of the environment from MC methods.

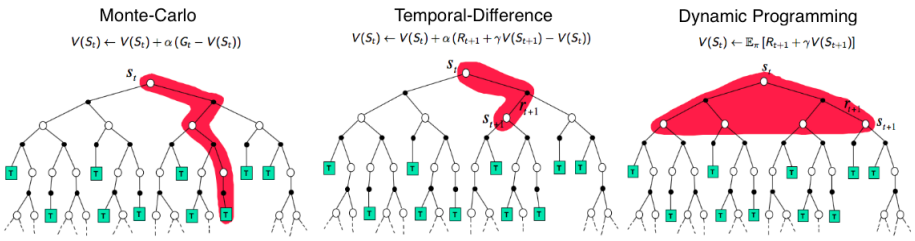

Figure 6: *Backup diagrams comparing Dynamic Programming, Monte Carlo methods and TD methods. Source: https://lilianweng.github.io/lil-log/assets/images/TD_MC_DP_backups.png.*

Moreover, we can extend these one-step TD methods to recover MC methods with the so called n-step TD methods. With one-step TD methods the same time-step determines how often the action can be changed and the time interval over which bootstrapping is done. Sometimes we may want to be able to update the action very fast to take into account anything that has changed, but bootstrapping works best if it is over a length of time in which a significant and recognizable state change has occurred. With one-step TD methods, these time intervals are the same, and so a compromise must be made, so, in this sense n-step methods enable bootstrapping to occur over multiple steps, freeing us from the compromise of using only one time step each time.

3.4.1 Exploration vs. exploitation

The policy evaluation problem for action values is to estimate $q_\pi(s, a)$, the expected return when starting in state s, taking action a, and thereafter following policy π. A state action pair (s, a) is said to be visited in an episode if the state s is visited and the action a is taken in it. The only complication is that many state-action pairs may never be visited. For policy evaluation to work for action-value pairs, we must assure continual exploration.

This problem is formulated more generally in the context of the so called exploration vs. exploitation trade-off, which shows the necessity to balance the exploitation of behaviours that the agent know that provide high return, with the exploration of new behaviors that the agent does not know its returns yet. The common approach to assuring that all state-action pairs are encountered, is to consider policies that are stochastic with a nonzero probability of selecting all action in each state. That is, these policies most of the time will select the best possible action, performing exploitation, but sometimes they will select a random action, performing exploration. There are two approaches to ensuring this:

- **On-policy methods**: which attempt to evaluate or improve the policy what is used to make decisions. In on-policy methods the policy is generally soft, meaning that $\pi(a|s) > 0$ for all $s \in \mathcal{S}, a \in \mathcal{A}$, but gradually shifted closer and closer to a deterministic optimal policy. A example of these policies are ε-greedy policies, meaning that most of the time they choose an action that has maximal estimated action values, but with probability ε they instead select an action at random. That is, all nongreedy actions are given the minimal probability of selection $\varepsilon/|\mathcal{A}|$, and the remaining bulk of probability $1 - \varepsilon + \varepsilon/|\mathcal{A}|$ is given to the greedy action. Using the natural notion of greedy policy for ε-soft policies, one is assured of improvement in every step. But with the drawback that we can only achieve the best policy among the ε-soft policies, but not the best policy overall.
- **Off-policy methods**: which evaluate or improve a policy different from that used to generate the data. The on-policy approach is actually a compromise, it learns action values not for the optimal policy, but for a near-optimal policy that still explores. A more straightforward

approach is to use two policies, one that is learned about and that becomes the optimal policy, the target policy, and one that is used to generate behavior, the behavior policy. In this case we say that learning is from data "off" the target policy, and the overall process is termed off-policy learning. A typical case is when the target policy is made as the deterministic greedy policy with respect to the current estimate of the action-value function given by the behavior policy.

3.4.2 On-policy methods

On the one hand, the most straightforward application of the TD idea in the form of a on-policy method is the SARSA algorithm [48]. Is is defined by the update rule

$$Q(S_t, A_t) \longleftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)\right]$$

which is done after every transition form a nonterminal state S_t. If S_{t+1} is terminal, then $Q(S_{t+1}, A_{t+1})$ is set to zero. This rule uses every element of the quintuple $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$ that make up a transition from one state-action pair to another, which gives rise to the name SARSA for the algorithm. We continually estimate q_π for the policy π and at the same time change π towards greediness with respect to q_π. The detail of this algorithm is shown in Algorithm 3.

Algorithm 3: SARSA

Inputs: M, ε, α
Initialize action-value function $Q(s, a)$ with random values
for *episode* $= 1 : M$ **do**
  Initialize s_t
  for $t = 1 : T$ **do**
    With probability ε select a random a_t, otherwise $a_t = \text{argmax}_a Q(s, a)$
    Execute action a_t and observe reward r_{t+1} and state s_{t+1}
    With probability ε select a random a_{t+1}, otherwise $a_{t+1} = \text{argmax}_a Q(s_{t+1}, a)$
    $Q(s_t, a_t) \longleftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)\right]$
    $s_t \longleftarrow s_{t+1}$
    $a_t \longleftarrow a_{t+1}$
  end
end

3.4.3 Off-policy methods

On the other hand, we can build a off-policy version of this algorithm doing a little change to the update rule. This algorithm is usually known as Q-learning [49] and is defined by the update rule

$$Q(S_t, A_t) \longleftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\right]$$

where the function Q directly approximates q_* the optimal action-values function, independent of the policy being followed, so it constitutes an off-policy algorithm. However, the policy still has

still an effect since it determines which state-action pairs are visited and updated, but all that is required for correct convergence is that all pairs continue to be updated. The detail of this algorithm is shown in Algorithm 4.

Algorithm 4: Q-learning

Inputs: M, ε, α
Initialize action-value function $Q(s, a)$ with random values
for *episode* $= 1 : M$ **do**
  Initialize s_t
  for $t = 1 : T$ **do**
    With probability ε select a random a_t, otherwise $a_t = \text{argmax}_a Q(s, a)$
    Execute action a_t and observe reward r_{t+1} and state s_{t+1}
    $Q(s_t, a_t) \longleftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)\right]$
    $s_t \longleftarrow s_{t+1}$
  end
end

However, the problem of this algorithm is that, since it involves a maximization in the construction of their target policies, it can lead to a significant positive bias in the estimation of Q-vales, hurting performance and potentially leading to suboptimal policies. One way to view the problem is that it is due to using the same samples both to determine the maximizing action and to estimate its value. Te idea of Double Q-learning [50] solve this problem by dividing the samples in two sets and using them to learn two independent estimates, call them $Q_1(a)$ and $Q_2(s)$, each an estimate of the true value $q(a)$ for all $a \in \mathcal{A}$. We could then use one estimate, say Q_1, to determine the maximizing action $A^* = \text{argmax}_a Q_1(a)$, and other, say Q_2, to provide the estimate of its value $Q_2(A^*) = Q_2(\text{argmax}_a Q_1(a))$. This estimate will then be unbiased in the sense that $\mathbb{E}[Q_2(A^*)] = q(A^*)$. We can also repeat the process with the role of the two estimates reversed to yield a second unbiased estimate $Q_1(A^*) = Q_1(\text{argmax}_a Q_2(a))$, so the two approximate value functions are treated completely symmetrically eliminating the bias. The detail of this algorithm is shown in Algorithm 5.

Algorithm 5: Double Q-learning

Inputs: M, ε, α
Initialize action-value function $Q_1(s, a)$ with random values
Initialize action-value function $Q_2(s, a)$ with random values
for *episode* $= 1 : M$ **do**
  Initialize s_t
  for $t = 1 : T$ **do**
    With probability ε select a random a_t, otherwise $a_t = \text{argmax}_a Q(s, a)$
    Execute action a_t and observe reward r_{t+1} and state s_{t+1}
    if $t \ \% \ 2 = 0$ **then**
      $Q_1(s_t, a_t) \longleftarrow Q_1(s_t, a_t) + \alpha \left[r_{t+1} + \gamma Q_2(s_{t+1}, \text{argmax}_a Q_1(s_{t+1}, a)) - Q_1(s_t, a_t)\right]$
    else
      $Q_2(s_t, a_t) \longleftarrow Q_2(s_t, a_t) + \alpha \left[r_{t+1} + \gamma Q_1(s_{t+1}, \text{argmax}_a Q_2(s_{t+1}, a)) - Q_2(s_t, a_t)\right]$
    end
    $s_t \longleftarrow s_{t+1}$
  end
end

3.5 State-of-the-art

All the methods that we have presented are based in a tabular set-up of the data, so in principle they are able to being applied to problems with arbitrarily large state spaces. However, in many of the tasks to which Reinforcement Learning can be useful the state space is combinatorial and enormous, or even continuous and non-easily discretizable. In such cases we cannot expect to find an optimal policy, or an optimal value function, even in the limit of infinite time and data. In addition to this, the other he problem with large state spaces is that almost every state encountered will never have been seen before, so to make sensible decisions in such states it is necessary to generalize from previous encounters with different states that are in some sense similar to the current one. Then, the kind of generalization we require is often called function approximation because it takes examples from a desired function and attempts to generalize from them to construct an approximation of the entire function. In this sense, function approximation is an instance of supervised learning.

3.5.1 Deep Q-learning

One of the most popular ways of formalizing this problem is by the use of Neural Networks for the approximating function due to its ability to learn very complex and nonlinear functions [51]. This fact, united to the advantages of optimality and simplicity given by the Q-learning algorithm, lead to the development of a series of algorithms known as Deep Q-learning (DQL) [14] [15] [52]. In this approach we want to build a function $Q(s, a; \theta)$, that is represented as a parametrized functional form with weights θ, that approximates the action-value function $q(s, a)$. To rank, and improve, approximate solutions given by the weights θ we need to define a cost function $L_(\theta)$. The most straightforward definition of this is a mean-square loss

$$L(\theta) = \mathbb{E}_{s,a\sim\rho(.)}\left[(y_i - Q(s, a; \theta))^2\right] \tag{37}$$

where $\rho(s, a)$ denotes the probability distribution of state-action pairs, which. The most popular methods to get the optimal weights θ_i of the approximate solution are the ones based on stochastic gradient descent. Then, given the update target Y_t for these weights, that can be any approximation of $q(S_t, A_t)$ including the usual SARSA or Q-learning updates, we can apply the general gradient-descent update rule

$$\theta_{t+1} = \theta_t + \alpha\left[Y_t - Q(S_t, A_t; \theta_t)\right]\nabla_{\theta_t} Q(S_t, A_t; \theta_t) \tag{38}$$

There are several ways of building the Q-network to approximate the q function, but the most effective seems to be the one in which there is a separate output unit for each possible action, and only the state representation is an input to the Neural Network [14]. The outputs correspond to the predicted Q-values of the individual actions for the input state. The main advantage of this type of architecture is the ability to compute Q-values for all possible actions in a given state with only a single forward pass through the network. This Q-network can be trained be minimising a sequence of loss functions $L_t(\theta_t)$ that changes at each iteration t as exposed above, where in this case, $y_t = \mathbb{E}_{s'\in\mathcal{E}}\left[r + \gamma \max_{a'} Q(s', a'; \theta_{t-1})|s, a\right]$ is the target for iteration t, and $\mathcal{E}$ is the emulator of the environment we are using to train the network. Then, applying the update rule of Q-learning

$$Y_t = R_{t+1} + \gamma \max_a Q(S_{t+1}, a; \theta_t) \tag{39}$$

and computing the gradient using the technique of backpropagation, we get the original implementation of Deep Q-Learning algorithm [14]. Note that the algorithm is an off-policy method, it learns about the greedy policy in which $a = \text{argmax}_a Q(s, a; \theta)$ while following a behavior distribution that ensures adequate exploration. A detailed version of the algorithm is shown in Algorithm 6. The benchmark to test this algorithm is the known Atari 2600 game environment, in which it achieves good performance, in most case superior than an experimented human player.

Algorithm 6: Deep Q-Learning (DQL)

Inputs: N, M, ε, C
Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights θ
for *episode* $= 1 : M$ **do**
  for $t = 1 : T$ **do**
    With probability ε select a random a_t, otherwise $a_t = \text{argmax}_a Q(s_t, a; \theta)$
    Execute action a_t in emulator and reward r_{t+1} and s_{t+1}
    Store transition $(s_t, a_t, r_{t+1}, s_{t+1})$ in $\mathcal{D}$
    Sample random minibatch $(s_j, a_j, r_{j+1}, s_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_{t+1} & \text{for terminal } s_{t+1} \\ r_{t+1} + \gamma \max_a Q(s_{t+1}, a; \theta) & \text{for non terminal } s_{t+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(s_t, a_t; \theta))^2$
  end
end

3.5.1.1 Experience Replay

In order to perform effectively the training of the Q-network is important to take into account the concept of Experience Replay [53]. In tabular Reinforcement Learning, the update is performed online, in a sample-by-sample manner, that is, every time a new transition is made, the value function is updated. The problem with this is that typically we will need a huge amount of episodes to achieve an optimal, or near optimal policy, in this new set-up with virtually infinite state-action pairs. The reason for this is that, if weights are adjusted for one certain state-action pair, then unpredictable changes also occur at other places in the state-action space. The technique of experience replay consists in the agent storing its experiences at every time-step $e_t = (s_t, a_t, r_t, s_{t+1})$ to build a dataset $\mathcal{D} = \{e_0, e_1, ...\}$ called replay memory. Then, in the training step of the learning process, instead of training with the corresponding sample, we apply updates using samples of experience drawn at random from the pool of stored samples. In this approach each step of experience is potentially used in many weight updates, making data more efficient, and breaking the correlations between samples due to the randomizing of the samples, reducing the variance of the updates.

This approach can be improved by prioritizing the most relevant experiences [54]. This prioritization avoid expensive sweeps over the entire replay memory by focusing only in the the most important subset of experience. Then, the central component of prioritized experience replay is the criterion by which the importance of each transition is measured. The ideal criterion would be the amount the agent can learn form a transition, but since it is not directly accessible we can approximated it by the magnitude δ of the TD update, which can be interpreted as how far the value is from the next-step estimate or how unexpected the transition is. We define the probability of sampling transition i as

$$P(i) = \frac{p_i^\alpha}{\sum_j p_j^\alpha}$$

where p_i is the priority of transition i and α determines how much prioritization is used. There are two main variants for this prioritization, proportional prioritization $p_i = |\delta_i| + \varepsilon$, where ε is a small positive constant that prevents the edge case of transitions not being revisited if this update is zero; and rank-based prioritization $p_i = 1/\text{rank}(i)$, where $\text{rank}(i)$ in the rank of transition i when the replay memory is sorted accordint to $|\delta_i|$. Both variant have been proven to improve the results of DQL algorithms in Atari 2600 game environment.

3.5.1.2 Target networks

Arrived to this point, we can introduce one modification to the method aimed at further improving the stability of the method [15]. The idea is to use a separate network for generating targets y_i in the Q-learning update. More formally, the proposal consist in cloning the Q-network after C updates to obtain a target network Q', and use it for generating the Q-learning targets y_i for the following C updates to Q before cloning it again. This modification makes the algorithm more stable compared to standard online Q-learning, where an update that increases $Q(S_t, A_t)$ often also increases $Q(S_{t+1}, a)$ for all a and hence also increases the target y_t, possibly leading to oscillations or divergences of the policy. Generating the targets using an older set of parameters adds a delay between the time an update to Q is made and the time the update affects the targets y_i, making these oscillations or divergences more unlikely. A detailed version of the algorithm is shown in Algorithm 7. This modification shows a noticeable improvement in the benchmark of Atari 2600 game environment, getting even better results than the original DQL implementation.

Algorithm 7: Deep Q-Learning with target network (DQL')

Inputs: N, M, ε, C
Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function Q with random weights $\theta' = \theta$
for *episode* $= 1 : M$ **do**
  for $t = 1 : T$ **do**
    With probability ε select a random a_t, otherwise $a_t = \text{argmax}_a Q(s_t, a; \theta)$
    Execute action a_t in emulator and observe r_t and s_{t+1}
    Store transition $(s_t, a_t, r_{t+1}, s_{t+1})$ in $\mathcal{D}$
    Sample random minibatch $(s_j, a_j, r_{j+1}, s_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_{t+1} & \text{for terminal } s_{t+1} \\ r_{t+1} + \gamma \max_a Q(s_{t+1}, a; \theta') & \text{for non terminal } s_{t+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(s_t, a_t; \theta))^2$
    Every C steps align networks with $\theta' = \theta$
  end
end

This improvement can be pushed even further using again the idea of Double Q-learning [52]. As we said before, Q-learning is known to overestimate q-values due to its maximization steps, and this is still true in the context of approximate solutions, so the idea is to reduce overestimations by decomposing the maximization operation into action selection and action evaluation. Although not fully decoupled, the target network provides a natural candidate for the second action-value function, without having to introduce additional networks. So, the proposal is to evaluate the greedy policy according to the online network Q, but using the target network Q' to estimate its value,

$$Y_t = R_{t+1} + \gamma Q'(S_{t+1}, \text{argmax}_a Q(S_{t+1}, a; \theta_t); \theta'_t) \tag{40}$$

where the second set of weights θ' can be updated symmetrically by switching the roles of θ and θ'

in the subsequent steps. A detailed version of the algorithm, known as Double Deep Q-Learning (DDQL), is shown in Algorithm 8. This improvement to the DQL algorithm is shown not only to produce more accurate vale estimates but also to lead to better policies, improving the results over the benchmark of Atari 2600 game environment.

Algorithm 8: Double Deep Q-Learning (DDQL)

Inputs: N, M, ε, C
Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function Q with random weights $\theta' = \theta$
for *episode* $= 1 : M$ **do**
  for $t = 1 : T$ **do**
    With probability ε select a random a_t, otherwise $a_t = \text{argmax}_a Q(s_t, a; \theta)$
    Execute action a_t in emulator and observe r_{t+1} and s_{t+1}
    Store transition $(s_t, a_t, r_{t+1}, s_{t+1})$ in $\mathcal{D}$
    Sample random minibatch $(s_j, a_j, r_{j+1}, s_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_{t+1} & \text{for terminal } s_{t+1} \\ r_{t+1} + \gamma Q'(s_{t+1}, \text{argmax}_a Q(s_{t+1}, a; \theta); \theta') & \text{for non terminal } s_{t+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(s_t, a_t; \theta))^2$
    Every C steps change networks with $\theta' \Longleftrightarrow \theta$
  end
end

4 Experimental Design

In this section we are going to use the domain knowledge that we have obtained in Section 2 to present an approach to the Stock Market as a Reinforcement Learning problem with the characteristics presented in Section 3. We divide this approach in three sequential steps. First, we have to build an emulator of the environment, the Stock Market, using the guidelines of OpenAI Gym [55], which is the most extended framework to develop Reinforcement Learning problems. Second, we have to build agents capable of interacting with this environment, using Tensorflow [56], which is one the most extended frameworks to develop Deep Learning applications. Finally, we have to implement the training algorithms aimed to improve the behavior of the agents built in the previous steps, namely, we are implementating, from scratch, the algorithms of Deep Reinforcement Learning in [14] [15] [52].

4.1 Environment

In order to build an agent to operate in Stock Market, first we have to define how the agent should perceive its environment, that is, we have to build a emulator for the Stock Market. In this sense, we need to consider how a human will perform the task. To formulate RL problem, we need to define three things:

- **Observations**: at any time-step we have available the time-series of prices for a stock at any previous time-step, so we are defining each state as a the previous *time_horizon* prices. The other information that a human has is its current positions regarding this stock, that is, flat, long or short, so we are using this information also as a part of the state.

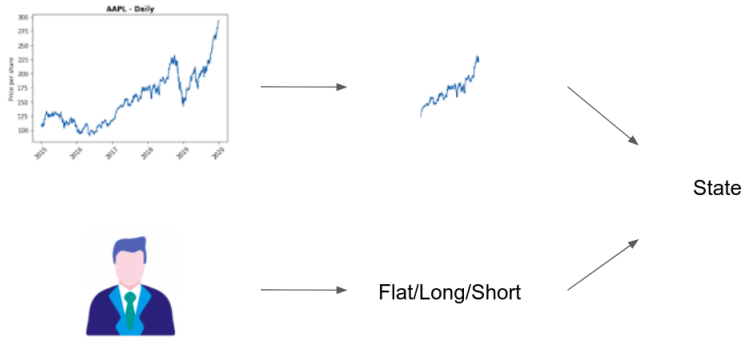

Figure 7: *Schema of the building of a state in our environment. Source: own*

- **Actions**: there are three possible actions at any time-step regarding a stock: hold, buy or sell. So we are going to implement they as follow:
  - *Hold*: do nothing, that is, skip the time-step without buying or selling.
  - *Buy*: open a position, that is, buy a share and save the *entry_price* at this time to compute the profit.
  - *Sell*: close a position, that is, sell a share and save the price at this time to compute the profit.

  However only some combinations of actions and positions are possible. We are going to implement the logic of positions and actions as follows:
  - *Flat*: you can hold to stay *Flat*, buy to become *Long* or sell to become *Short*.
  - *Long*: you can hold to stay *Long* or sell to become *Flat*. Buy to become *Long* will be considered as hold to stay *Long*.
  - *Short*: you can hold to stay *Short* or buy to become *Flat*. Sell to become *Short* will be considered as hold to stay *Short*.

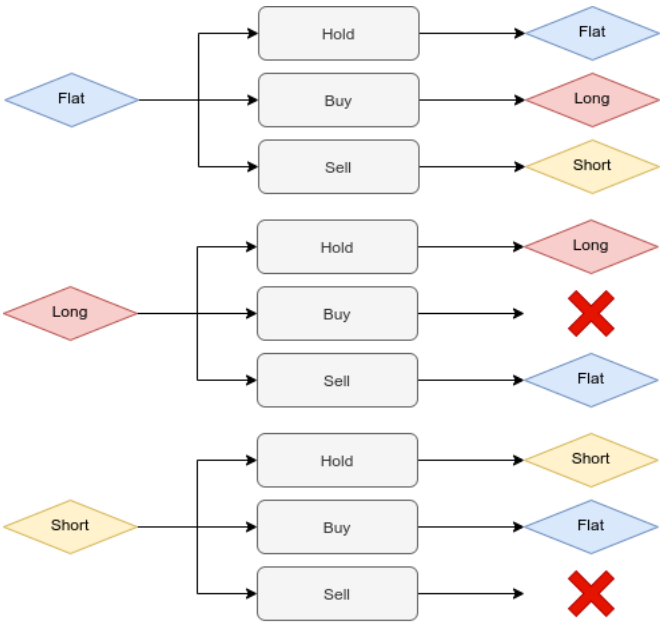

Figure 8: *Schema of the action and position dynamics. Source: own*

- **Rewards**: the usual way to analyze the evolution of prices in Stock Market and build profitable portfolios is not by directly analyzing the prices itself, but the so-called returns. Returns denote the relative change of the price in each time-step, which is a is more robust approach than using the prices themselves because its normalization, which allow to generalize better between different stocks. Due to the casuistry of each position, rewards are computed in different ways depending on the position taken by the agent:
  - *Flat*: if the agent has no position opened, the reward at time-step t is:

$$R_t = 0$$

  - *Long*: if the agent buys at time-step t_{entry} at price P_{entry}, since we expect the price to rise the reward at time-step t is:

$$R_t = \frac{P_t - P_{entry}}{P_{entry}}$$

  - *Short*: if the agent sells at time-step t_{entry} at price P_{entry}, since we expect the price to fall the reward at time-step t is:

$$R_t = \frac{P_{entry} - P_t}{P_{entry}}$$

In our simulator of the environment, we reproduce the brokerage fee, which is the fee that is charged when we buy or sell stocks. The introduction of a fee also incentivize profit that is sustained over time, generating long-term strategies that gain money consistently rather that rapidly gain money using unsustainable strategies based in spurious changes on the price rather than on the its trend over time. This can be implemented as a fixed amount, as a percentage of the operations or as a combination of both ones. We have choose to implement it as a fixed percentage of the operation ϵ, so taking it into account, the rewards after closing a positions are computed as:

  - *Long*: if the agent buys at time-step t_{entry} at price P_{entry}, and sells at time-step t_{exit} at price P_{exit}, the reward at time-step t_{exit} is:

$$R_f = \left(1 + \frac{P_{exit} - P_{entry}}{P_{entry}}\right)(1 - \epsilon) - 1$$

  - *Short*: if the agent sells at time-step t_{entry} at price P_{entry}, and buy at time-step t_{exit} at price P_{exit}, the reward at time-step t_{exit} is:

$$R_f = \left(1 + \frac{P_{entry} - P_{exit}}{P_{entry}}\right)(1 - \epsilon) - 1$$

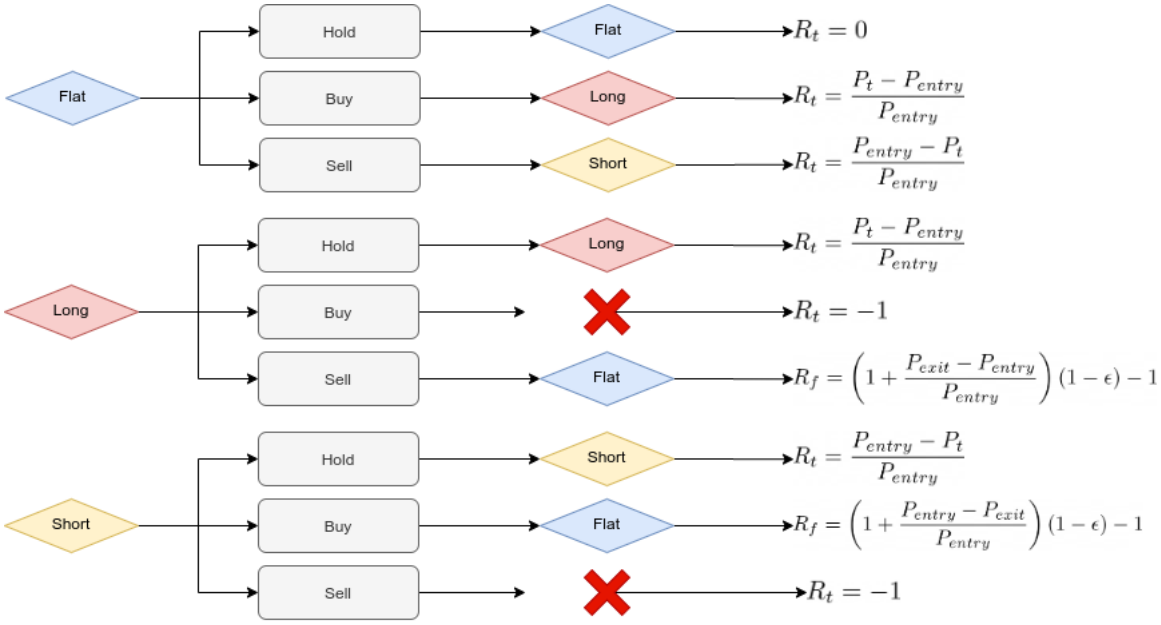

Figure 9: *Schema of the reward dynamics. Source: own*

As we said, an environment comprise everything outside the agent, so its code implementation has to contain all the functionality to allow the agent to interact with it and to learn. We are going to implement this environment according to the guidelines of OpenAI gym [55], one on the most used libraries to develop reinforcement learning applications. The environments are then defined as a python class with the following methods:

- **__init(self)__**: define initial information of the environment such as the observation space or the action space.
- **reset(self)**: reset the environment's state.
- **step(self, action)**: step the environment by one timestep. Returns:
  - *observation*: an environment-specific object representing our observation of the environment.
  - *reward*: amount of reward achieved by the previous action.
  - *done*: whether it's time to reset the environment again.
  - *info*: diagnostic information useful for debugging.
- **render(self, mode='human')**: render one frame of the environment.

So, with this structure in mind we develop an environment based on the historical time-series of prices of a given stock with the following parameters:

- **stock**: name of the stock to track.
- **data_start**: date to start tracking the prices of the stock. We are starting our tracking at 2015/01/01 by default.
- **date_end**: date to finish tracking the prices of the stock. We are finishing our tracking at 2019/12/31 by default.

- **time_horizon**: number of time-steps inside each state. We are experimenting with different values for this parameter.
- **time_skip**: number of time-steps to skip between states. With the introduction of this parameters we lighten the training process and avoid the redundancy of information between states. We are going to use a value o 5, a week in laboral days, for this parameters.
- **fee**: ratio of the operation to pay as a fee to the broker. We are experimenting with different values for this parameter despite using a value o 0.01 by default.

4.2 Agents

Once we have build a simulator for the environment we can start to build agents to interact with it. Following the actual trends, we are going to build our agents using the techniques of Deep Learning, that is, through the use of Neural Networks [57]. Neural Networks consists in framework of algorithms initiated in the 40s [58], as a emulation of the human neurons, and popularized lately by the discovering of the algorithm of backpropagation [59]. This algorithm implement the training step of the network in a sequential layer-by-layer fashion, allowing the process to scale very efficiently and, consequently, expanding its applications to the industry.

They key idea behind a Neuron is to build a unit of computation that performs a operation on its inputs, usually coming from other Neurons. The concatenation of these units, that can be performed in parallel or in sequence, builds the structure known as Neural Network. The training of the weights, associated with the union of neurons, of the Neural Network is performed using the backpropagation algorithm [59], as a layer-by-layer application of gradient descent. This gradient descent can be improved in many ways, such as adding momentum or adapting the learning rate through the learning process, usually combined and constituting the well-know ADAM optimizer [60]. All these features, are implemented in all the state-of-the-art libraries for Deep Learning such as Tensorflow [56] of PyTorch [61]. In our case, we are using Tensorflow since it is the most extended framework nowadays.

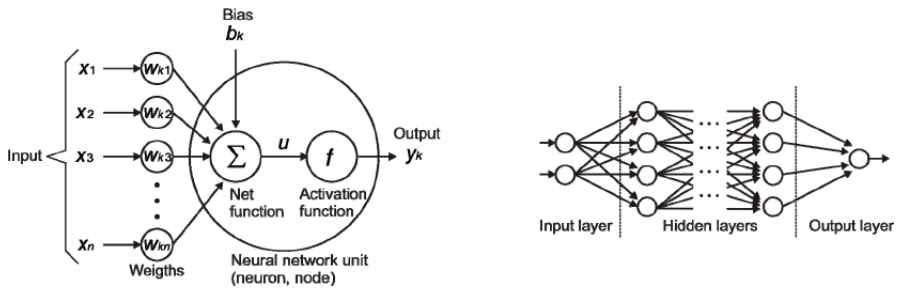

Figure 10: *Functional schema of a neuron and a neuronal network. Source: [58]*

One on the bigger revolutions in the field of Neural Networks were Recurrent Neuronal Networks, in which, in addition to the connections between neurons in adjacent layers, the neurons of the same layer also share connections. Concretely, in Recurrent Neural Networks, each neuron shares a connection with the next neuron in the same layer, producing a recurrent pattern that makes these networks ideal to the treatment of sequential data such as time-series. In this sense the most extender architecture is the one of Long Short-Term Memory [62], usually known as LSTM, which tries to avoid the problem of long-term dependencies with the use of a memory flow combined with the habitual flow between the neurons.

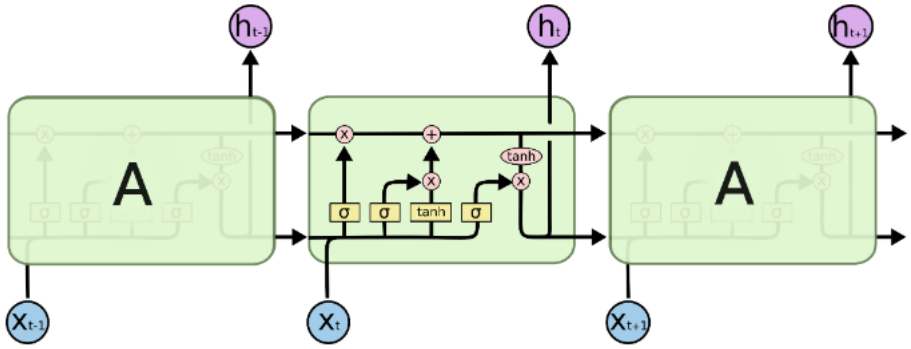

Figure 11: *Detail of a LSTM unit. We can see that there are two flows between the units in the same layer, the lower usual flow for the recurrent units, and the upper flow known as memory. Source: https://colah.github.io/posts/2015-08-Understanding-LSTMs/*

In this sense, we are developing our agents using two networks: the first approach will be to use a classic Feedforward Neural Networks with two hidden layers for the prices and one additional joined with the information of the position; the second approach will be a Recurrent Neural Network with LSTM units with, again two hidden layers for the prices and one additional joined with the information of the position. The architectures of these networks are showed in more detail in Section 5.

4.3 Algorithms

Once we have defined our agent, we need to decide how we are going to train it. In this sense, we are going to use the three more extended versions of Deep Q-learning, the original approach [14], the approach with a target network [15], and the approach of Double Q-learning [52]. In Figures 12, 13 and 14 we detail the updates involved by those algorithms.

```
for state_t, action_t, reward_t, state_t_next, done_check in minibatch:
    # Compute prediction
    y_t = self.q_network.predict(state_t)

    # Apply update
    if done_check:
        y_t[0][action_t] = reward_t
    else:
        y_t[0][action_t] = reward_t + self.gamma * np.max(self.q_network.predict(state_t_next))

    # Perform gradient descent in minibatch
    self.q_network.fit(state_t, y_t, epochs = 1, batch_size = self.batch_size, verbose = 0)
```

Figure 12: *Detail of the update of Deep Q-Learning. Source: own.*

```
for state_t, action_t, reward_t, state_t_next, done_check in minibatch:
    # Compute prediction
    y_t = self.q_network.predict(state_t)

    # Apply update
    if done_check:
        y_t[0][action_t] = reward_t
    else:
        y_t[0][action_t] = reward_t + self.gamma * np.max(self.q_network_target.predict(state_t_next))

    # Perform gradient descent
    self.q_network.fit(state_t, y_t, epochs = 1, batch_size = self.batch_size, verbose = 0)

    # Align if necessary
    align += 1
    if (align % steps_to_align) == 0:
        self.q_network_target.set_weights(self.q_network.get_weights())
```

Figure 13: *Detail of the update of Deep Q-Learning with target network. Source: own.*

```
for state_t, action_t, reward_t, state_t_next, done_check in minibatch:
    # Compute prediction
    y_t = self.q_network.predict(state_t)

    # Apply update
    if done_check:
        y_t[0][action_t] = reward_t
    else:
        action_pred = np.argmax(self.q_network.predict(state_t_next))
        y_t[0][action_t] = reward_t + self.gamma * self.q_network_target.predict(state_t_next)[0][action_pred]

    # Perform gradient descent in minibatch
    self.q_network.fit(state_t, y_t, epochs = 1, batch_size = self.batch_size, verbose = 0)

    # Change networks if necessary
    change += 1
    if (change % steps_to_change) == 0:
        # Create temporal network
        temp = tf.keras.models.clone_model(self.q_network)
        temp.set_weights(self.q_network.get_weights())

        # Change weights
        self.q_network.set_weights(self.q_network_target.get_weights())
        self.q_network_target.set_weights(temp.get_weights())
```

Figure 14: *Detail of the update of Double Deep Q-Learning. Source: own.*

5 Results

In this section we are going to use the environment and agents built in Section 4 to find the best combination of architectures and parameters to solve our Reinforcement Learning approach to Stock Market. In this sense, we are going to run the following experiments:

- **Feedforward network with a given stock**: in this experiment we chose one stock, namely AAPl (Apple), to train the agent, and three stocks to test the performance of the agent, namely, AAPL (Apple), GE (General Electric) and CAT (Caterpillar), in order to see if the knowledge obtained by the agent can generalize to other stocks. We chose a Feedforward Neural Network to implement our agent and compare the performance of the three classic methods of Deep Q-learning [14] [15] [52] for different values for *time_horizon*, namely 8, 16 and 32.
- **Recurrent network with a given stock**: in this experiment we chose one stock, namely AAPl (Apple), to train the agent, and three stocks to test the performance of the agent, namely, AAPL (Apple), GE (General Electric) and CAT (Caterpillar), in order to see if the knowledge obtained by the agent can generalize to other stocks. We chose a Recurrent Neural Network to implement our agent and compare the performance of the three classic methods of Deep Q-learning [14] [15] [52] for different values for *time_horizon*, namely 8, 16 and 32.
- **Training with a random stocks**: in this experiment we do not chose one stock to train the agent, but use random stocks from S&P500. Then we use three stocks to test the performance of the agent, namely, AAPL (Apple), GE (General Electric) and CAT (Caterpillar), in order to see if the knowledge obtained by the agent can generalize to other stocks. We chose a Feedforward Neural Network and a Recurrent Neural Network to implement our agent and compare the performance of DDQL [52] with *time_horizon* $= 32$, with the results of Experiment 1 and 2.

5.1 Feedforward network with a given stock

The first experiment that we are going to run, as we said before, consists in analyzing the adequacy of using a Feedforward Neural Network for our agent. In this sense, we are going to perform three independent analysis, based in the length of the time window used to build the

states. With this, we want to study independently the choices of the training algorithm and the *time_horizon* parameter, looking for possible interactions between both choices.

5.1.1 time_horizon = 8

The first run of the experiment uses a choice of *time_horizon* = 8. This choice, united to the value of the position of the agent, requires a specific architecture of the Neuronal Network, shown in Figure 15, which joins he information of both inputs, one with a previous encoding by two feedforward layers and other with any modification into one last feedforward layer.

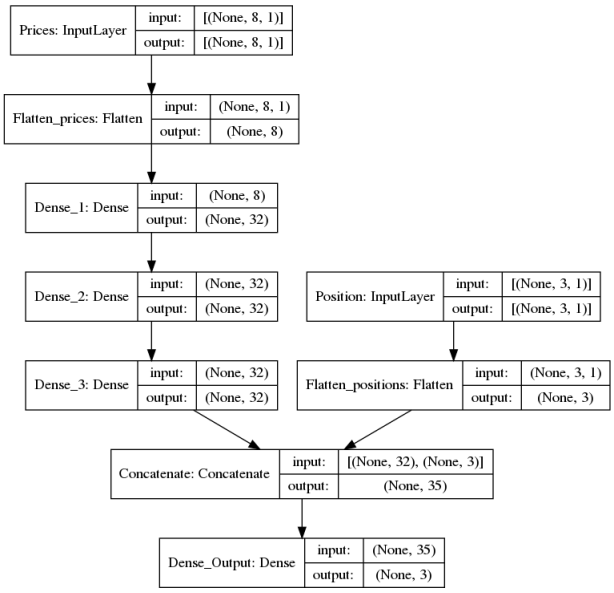

Figure 15: *Architecture of the Feedforward Neural Network for a choice of time_horizon = 8. Source: own*

With this architecture we can build our agent and start to operate. In the following figures we show the training process with the AAPL stock and the test of this trained agent in this stock, and two additional stocks, GE y CAT, to study the capacity of generalization of this training process. In Figure 16 we show and agent trained by the classical Deep Q-learning algorithm [14]; in Figure 17 we show and agent trained by the Deep Q-learning algorithm with target network [15]; and, in Figure 18 we show and agent trained by the Double Deep Q-learning algorithm [52].

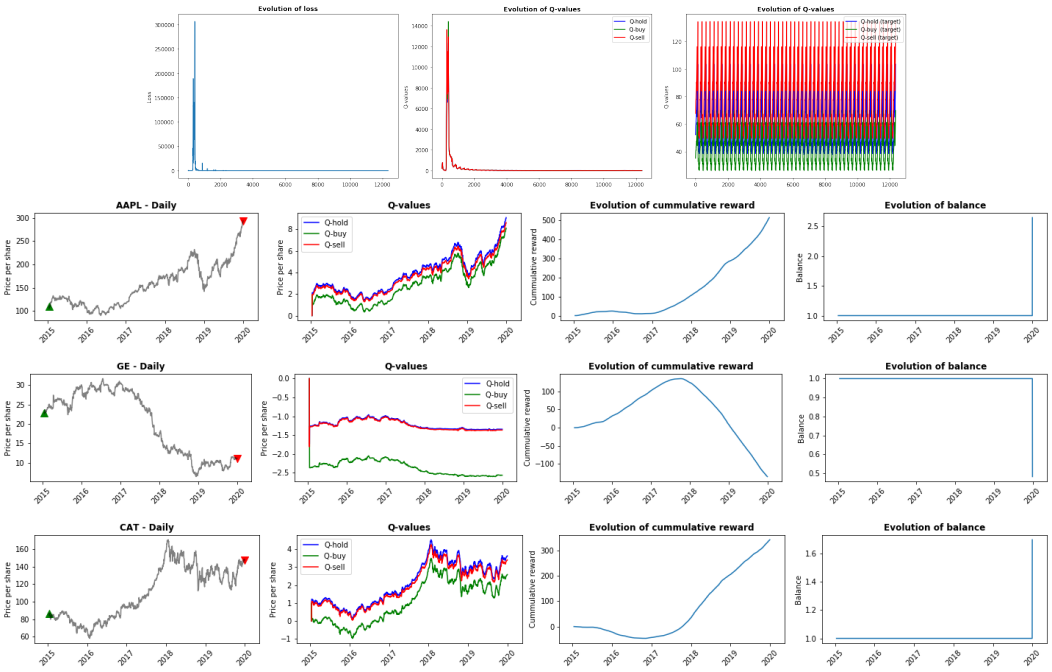

Figure 16: *Diagrams of experiment 1. The first row shows the evolution of the loss and Q-values for a agent trained with the classic Deep Q-learning algorithm [14]. The next rows show the application of the trained agent to different stocks. Train with time_horizon = 8. Source: own*

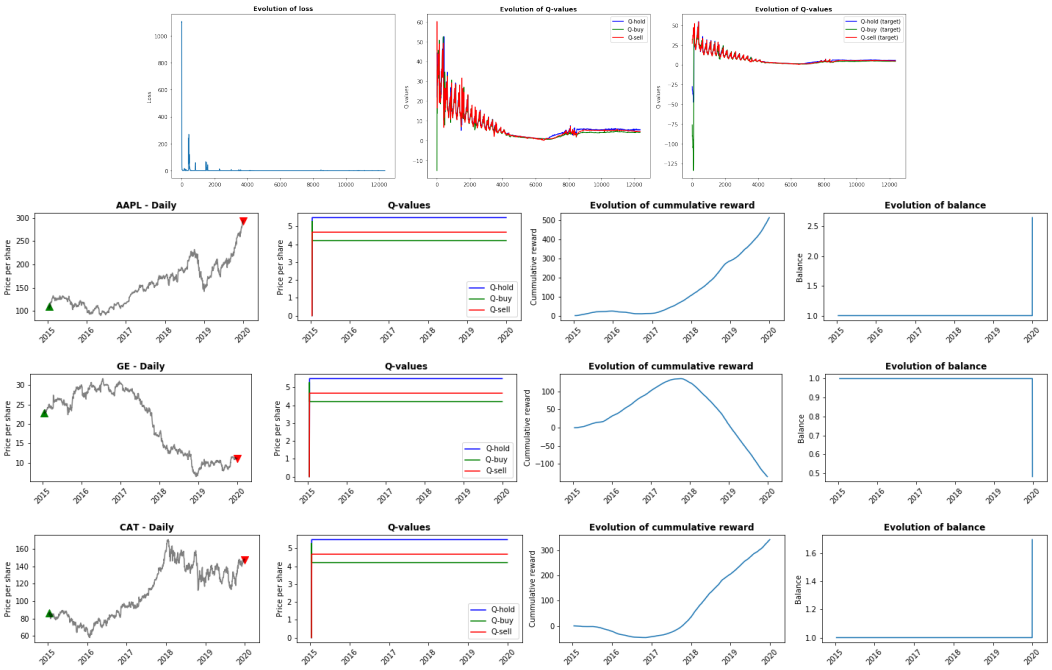

Figure 17: *Diagrams of experiment 1. The first row shows the evolution of the loss and Q-values for a agent trained with the Deep Q-learning algorithm with target network [15]. The next rows show the application of the trained agent to different stocks. Train with time_horizon = 8. Source: own*

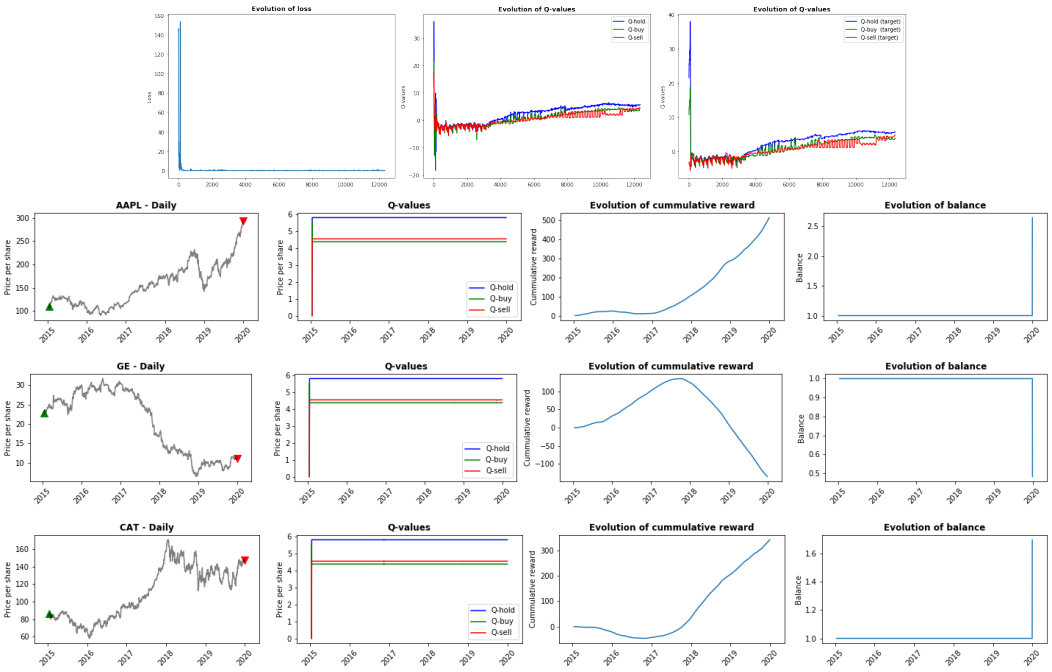

Figure 18: *Diagrams of experiment 1. The first row shows the evolution of the loss and Q-values for a agent trained with the Double Deep Q-learning algorithm [52]. The next rows show the application of the trained agent to different stocks. Train with time_horizon = 8. Source: own*

In these figures we can see many features that will appear constantly in the following runs of the experiment:

- The learning process is very fast, in less than 5 episodes the agent achieves a plateau in the reduction of the loss, which indicates that the training algorithm is very efficient despite its online definition. This is produced, as noted in [14], by the use of experience replay, which allows the agent to skip the redundancy of the subsequent states, by choosing randomly them from a buffer of states, and consequently, to avoid getting stuck in the learning process. This is common to the three algorithms
- The tuning of Q-values is a bit slower in the last two algorithms [15] [50]. This is produced by the use of a target network that does not update in all steps, slowing the velocity of convergence.
- The use of more sophisticated algorithms traduce in a better separation of the Q-values in the tests, we can see that the gaps between the hold, buy and sell lines is bigger in [15] respect to [14], and even bigger in [50].
- The agent does not generalized very well, we can see that, despite the change of the trend in the two last time-series, it does not produce the adequate behavior.

5.1.2 time_horizon = 16

The second run of the experiment uses a choice of *time_horizon* = 16. This choice, united to the value of the position of the agent, requires a specific architecture of the Neuronal Network, shown in Figure 19, which joins he information of both inputs, one with a previous encoding by two feedforward layers and other with any modification into one last feedforward layer.

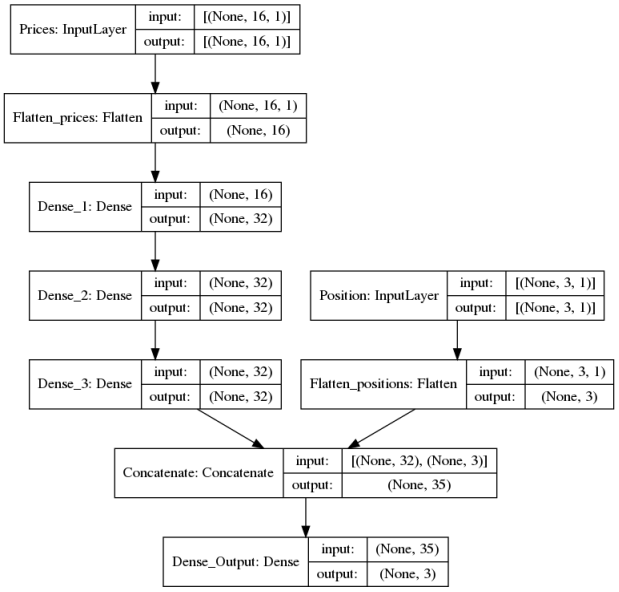

Figure 19: *Architecture of the Feedforward Neural Network for a choice of time_horizon = 16. Source: own*

With this architecture we can build our agent and start to operate. In the following figures we show the training process with the AAPL stock and the test of this trained agent in this stock, and two additional stocks, GE y CAT, to study the capacity of generalization of this training process. In Figure 20 we show and agent trained by the classical Deep Q-learning algorithm [14]; in Figure 21 we show and agent trained by the Deep Q-learning algorithm with target network [15]; and, in Figure 22 we show and agent trained by the Double Deep Q-learning algorithm [52].

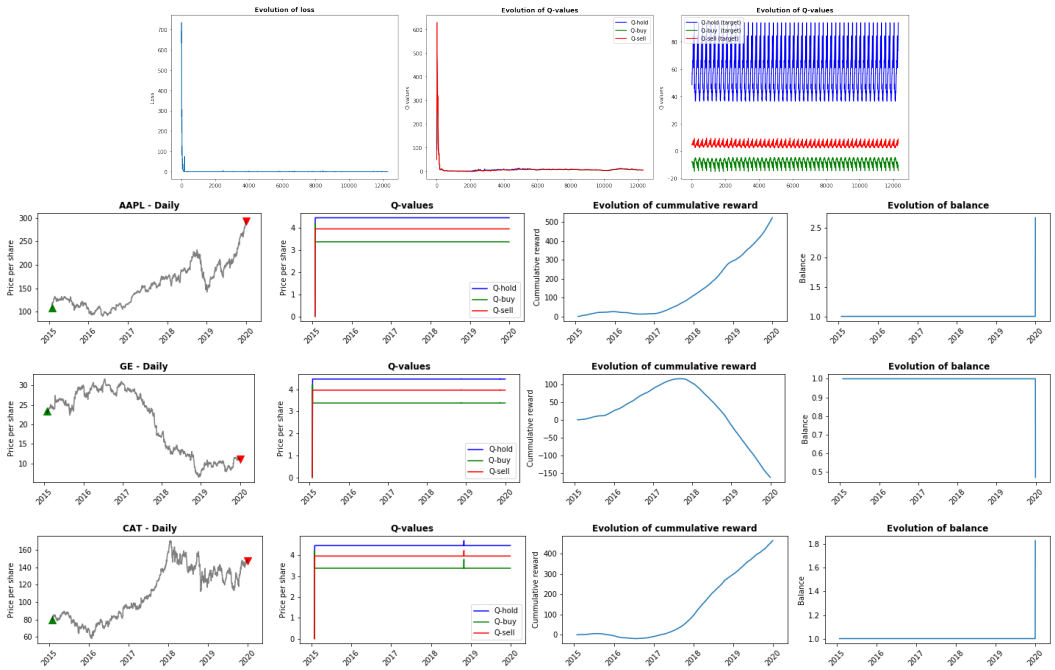

Figure 20: *Diagrams of experiment 1. The first row shows the evolution of the loss and Q-values for a agent trained with the classic Deep Q-learning algorithm [14]. The next rows show the application of the trained agent to different stocks. Train with time_horizon = 16. Source: own*

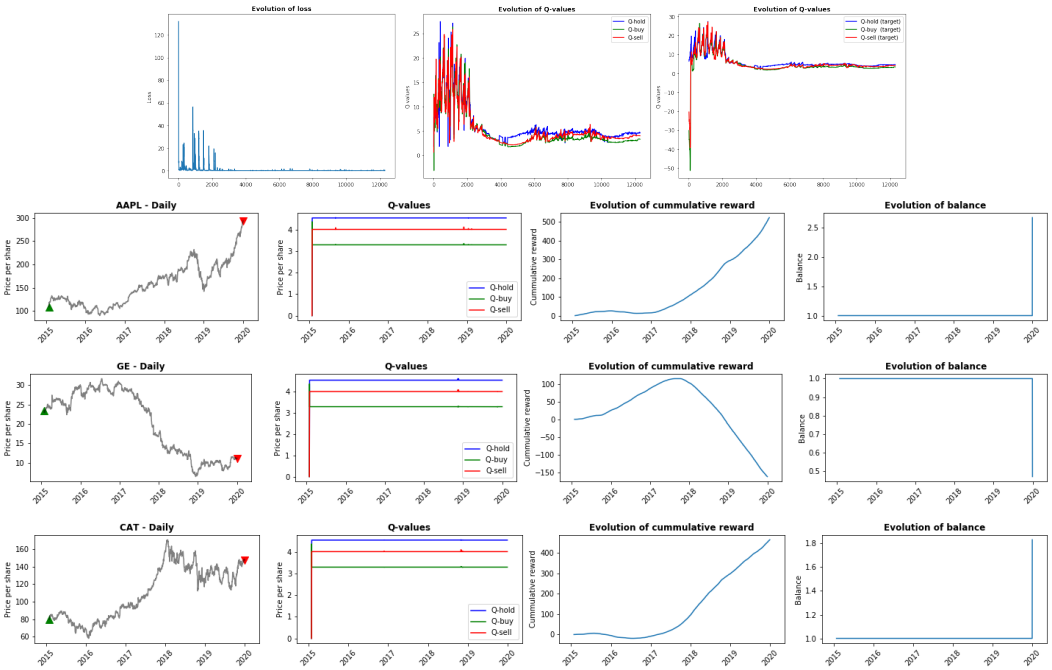

Figure 21: *Diagrams of experiment 1. The first row shows the evolution of the loss and Q-values for a agent trained with the Deep Q-learning algorithm with target network [15]. The next rows show the application of the trained agent to different stocks. Train with time_horizon = 16. Source: own*

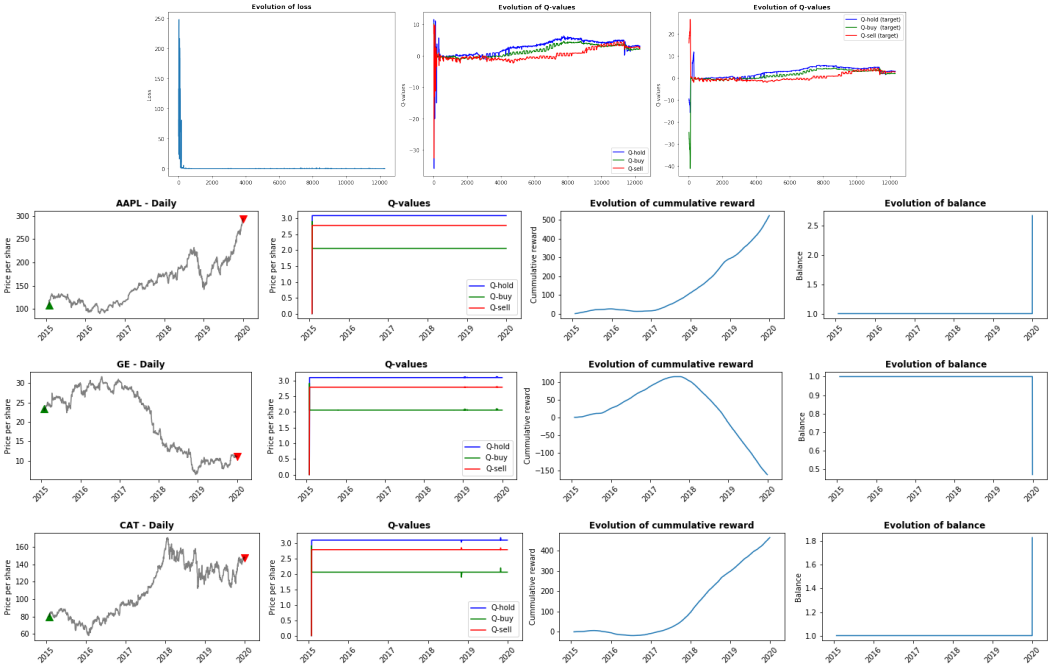

Figure 22: *Diagrams of experiment 1. The first row shows the evolution of the loss and Q-values for a agent trained with the Double Deep Q-learning algorithm [52]. The next rows show the application of the trained agent to different stocks. Train with time_horizon = 16. Source: own*

In these figures we can see more or less the features that in the previous runs of the experi-

ment, plus an additional fact:

- The learning process is very fast by the use of experience replay
- The tuning of Q-values is a bit slower in the last two algorithms due to the use of a target network that does not update in all steps.
- The use of more sophisticated algorithms traduce in a better separation of the Q-values in the tests.
- The agent does not generalized very well.
- We can see some lag between the trend in the time-series of prices and the time-series of rewards. The bigger width of the states, implemented by the parameter *time_horizon*, creates a time separation between the time the agent receives the information and the time that the agent can experiment a reward from using it.

5.1.3 time_horizon = 32

The third run of the experiment uses a choice of *time_horizon* = 32. This choice, united to the value of the position of the agent, requires a specific architecture of the Neuronal Network, shown in Figure 23, which joins he information of both inputs, one with a previous encoding by two feedforward layers and other with any modification into one last feedforward layer.

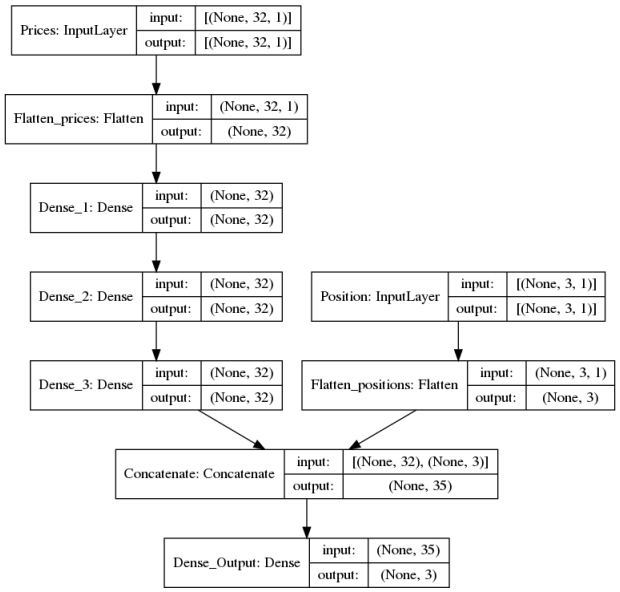

Figure 23: *Architecture of the Feedforward Neural Network for a choice of time_horizon* = 32. *Source: own*

With this architecture we can build our agent and start to operate. In the following figures we show the training process with the AAPL stock and the test of this trained agent in this stock, and two additional stocks, GE y CAT, to study the capacity of generalization of this training process. In Figure 24 we show and agent trained by the classical Deep Q-learning algorithm [14]; in Figure 25 we show and agent trained by the Deep Q-learning algorithm with target network [15]; and, in Figure 26 we show and agent trained by the Double Deep Q-learning algorithm [52].

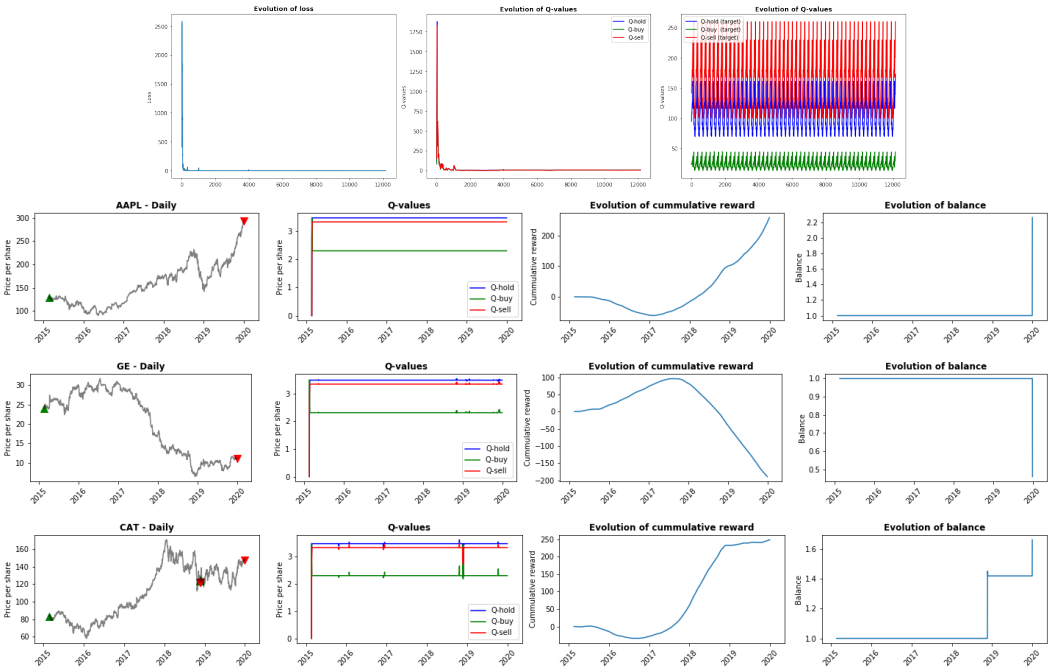

Figure 24: *Diagrams of experiment 1. The first row shows the evolution of the loss and Q-values for a agent trained with the classic Deep Q-learning algorithm [14]. The next rows show the application of the trained agent to different stocks. Train with time_horizon = 32. Source: own*

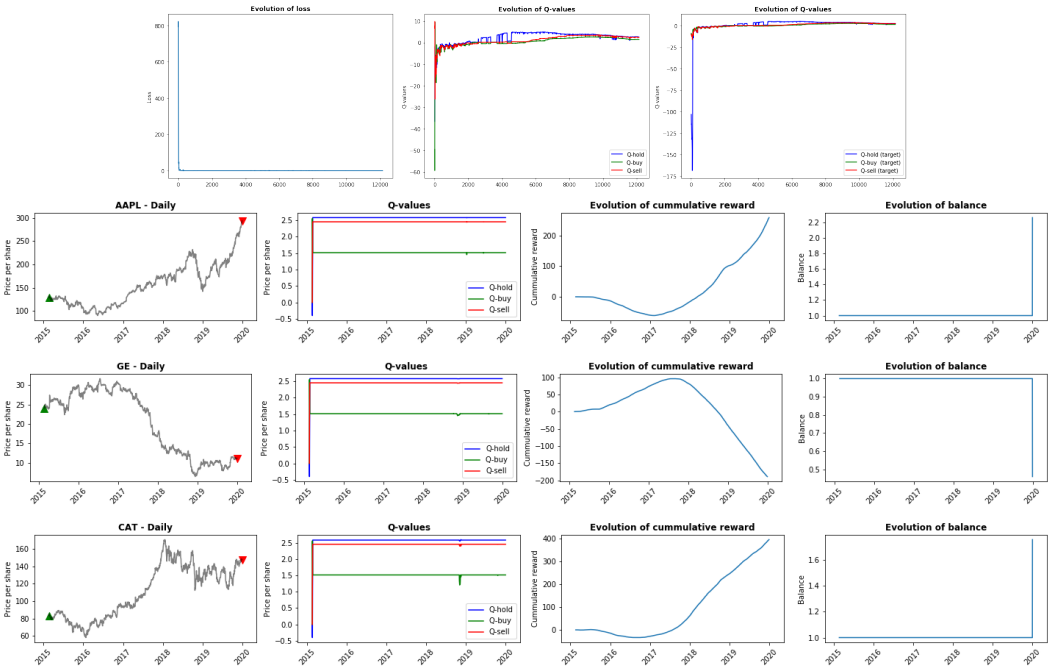

Figure 25: *Diagrams of experiment 1. The first row shows the evolution of the loss and Q-values for a agent trained with the Deep Q-learning algorithm with target network [15]. The next rows show the application of the trained agent to different stocks. Train with time_horizon = 32. Source: own*

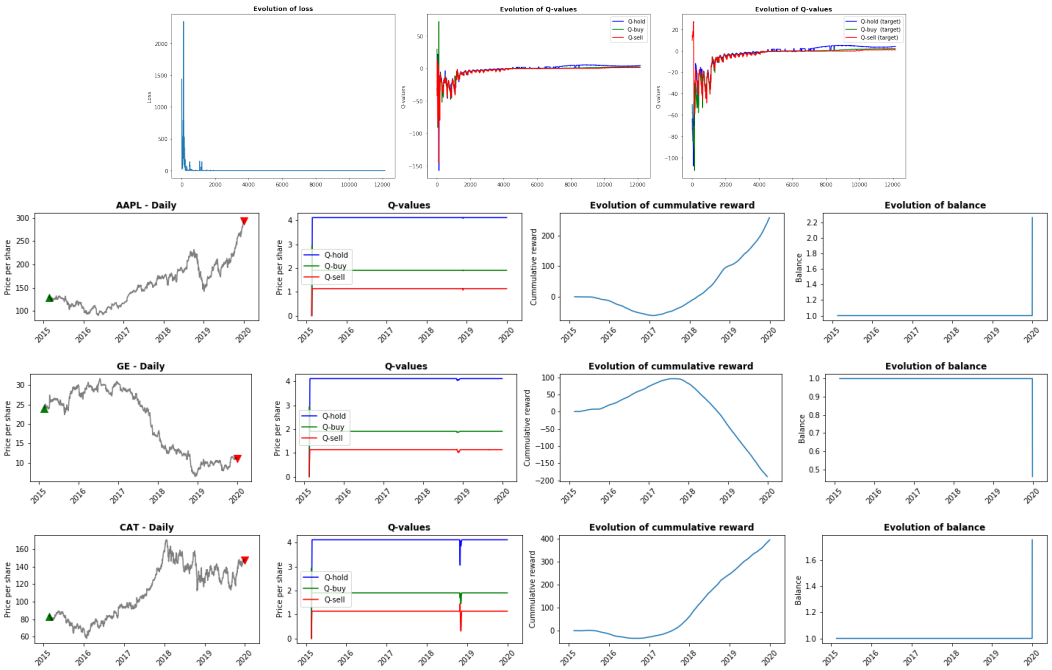

Figure 26: *Diagrams of experiment 1. The first row shows the evolution of the loss and Q-values for a agent trained with the Double Deep Q-learning algorithm [52]. The next rows show the application of the trained agent to different stocks. Train with time_horizon = 32. Source: own*

In these figures we can see more or less the features that in the previous runs of the experiment:

- The learning process is very fast by the use of experience replay
- The tuning of Q-values is a bit slower in the last two algorithms due to the use of a target network that does not update in all steps.
- The use of more sophisticated algorithms traduce in a better separation of the Q-values in the tests.
- The agent does not generalized very well.
- The lag between the time-series of prices and the rewards is bigger.

5.1.4 Comments

As we said before, the first row of diagram in each run of the experiment shows the evolution of the learning process. So, on the one hand, looking at the evolution of the loss we can see that it converges very fast to a plateau, and, when observed carefully we can see that the more complex algorithms achieve values of the loss slightly lower. Also, looking at the evolution of the Q-values we can see that they reach more stable values. When comparing the diagrams for the different values of *time_horizon*, we can see that the higher this value is, the better are the results in loss and Q-values. This is trivial since the increase of this parameter also increases the information available to the agent in order to take a decision.

And, on the other hand, when looking at the tests we can see that the use of more complex algorithms make the Q-values of the actions to be more separated, that is, the decisions are better

in the sense that the margin that marks the best decision is bigger. The same effect can be seen when, using the same algorithm, we increase the value of *time_horizon*. However, the increase of this parameter induces a lag between the instant when the agent receives the information of a time-step, and the instant in which the action triggered by this information receives a reward. So, this parameter has to be tuned carefully in order to be enough flexible to detect important changes of the trend of the time-series of prices. In this sense, we are going to be conservative and the are going to choose 32 as the best value for the *time_horizon* parameter, since is when the lag starts to be noticeable but still conserves an important quantity of information. Finally, is important to note that the agent generalizes very bad when we change the stock and the actions are not taken in an optimal way, this is specially noticeable in the GE stock. This can arise from the particular choice of the training stock, or from the fact that we use only one stock to train the agent, producing an unintentional overfitting.

5.2 Recurrent network with a given stock

The second experiment that we are going to run, as we said before, consists in analyzing the adequacy of using a Recurrent neural network for our agent and its improvements from the use of a Feedforward Neural Network. In this sense, we are going to perform three independent analysis, based in the length of the time window used to build the states. With this, we want to study independently the choices of the training algorithm and the *time_horizon* parameter, looking for possible interactions between both choices.

5.2.1 time_horizon = 8

The first run of the experiment uses a choice of *time_horizon* = 8. This choice, united to the value of the position of the agent, requires a specific architecture of the Neuronal Network, shown in Figure 27, which joins he information of both inputs, one with a previous encoding by one Bidirectional LSTM and one simple LSTM layer, and other with any modification into one last feedforward layer.

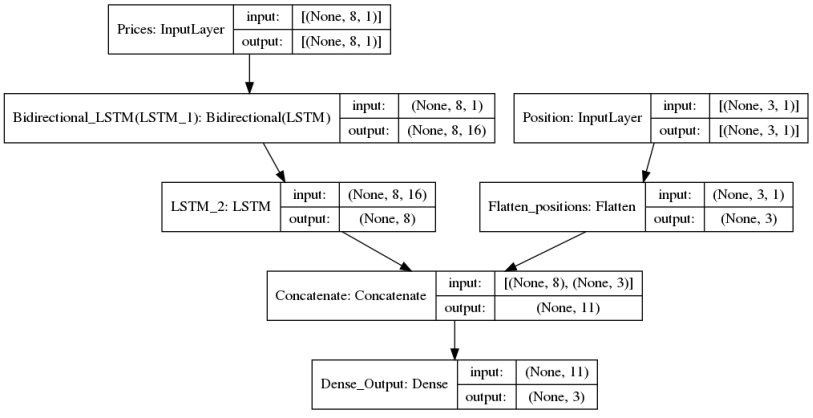

Figure 27: *Architecture of the LSTM Neural Network for a choice of time_horizon* = 8. *Source: own*

With this architecture we can build our agent and start to operate. In the following figures we show the training process with the AAPL stock and the test of this trained agent in this stock, and two additional stocks, GE y CAT, to study the capacity of generalization of this training process. In Figure 28 we show and agent trained by the classical Deep Q-learning algorithm [14];

in Figure 29 we show and agent trained by the Deep Q-learning algorithm with target network [15]; and, in Figure 30 we show and agent trained by the Double Deep Q-learning algorithm [52].

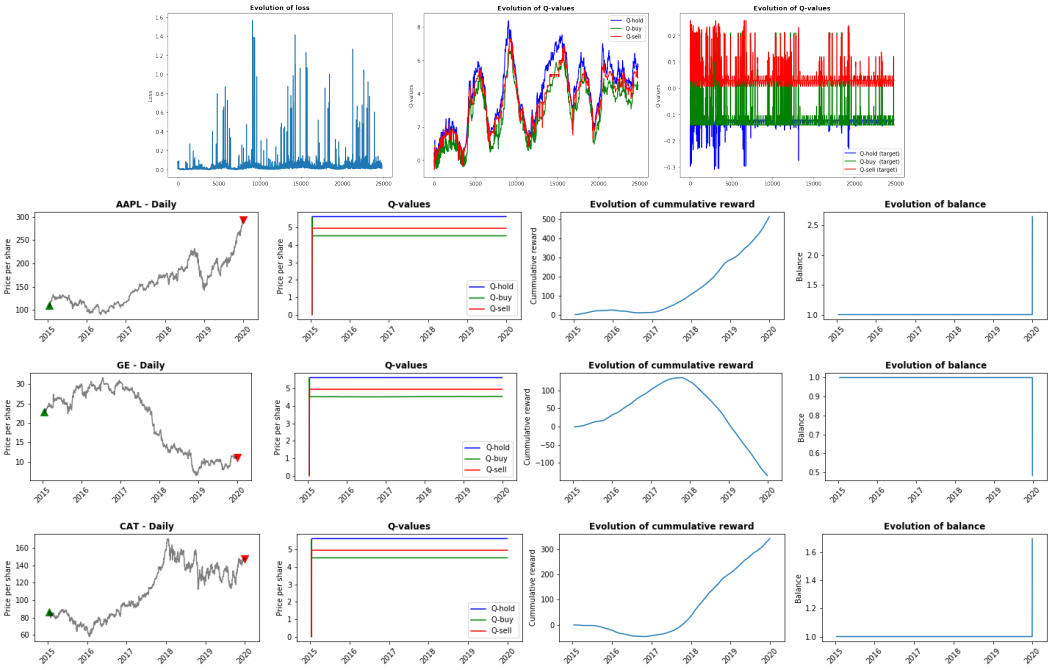

Figure 28: *Diagrams of experiment 2. The first row shows the evolution of the loss and Q-values for a agent trained with the classic Deep Q-learning algorithm [14]. The next rows show the application of the trained agent to different stocks. Train with time_horizon = 8. Source: own*

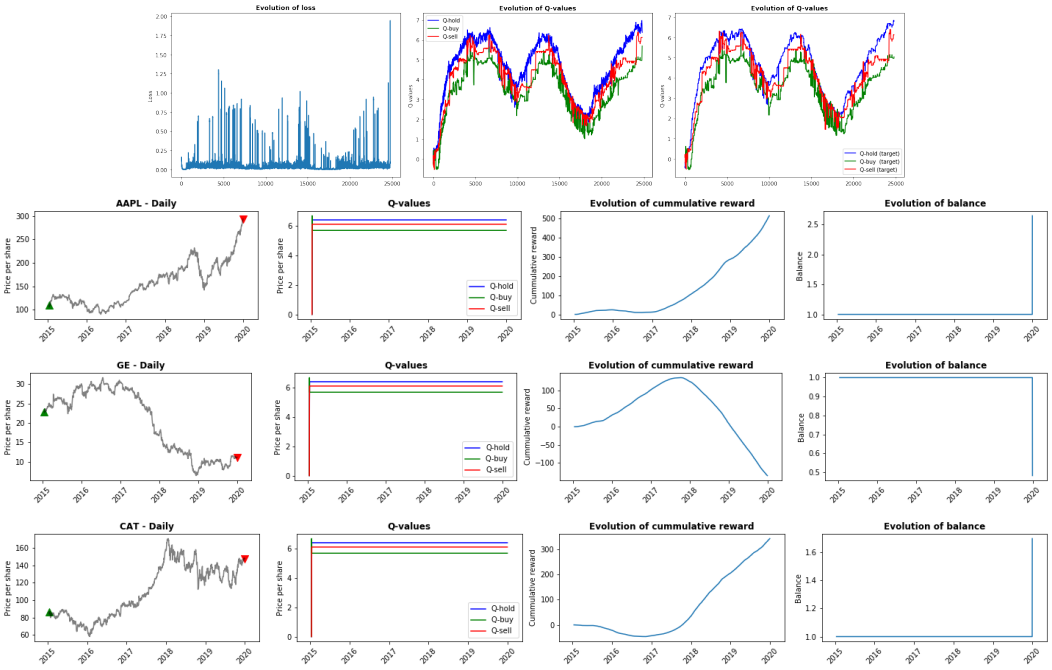

Figure 29: *Diagrams of experiment 2. The first row shows the evolution of the loss and Q-values for a agent trained with the Deep Q-learning algorithm with target network [15]. The next rows show the application of the trained agent to different stocks. Train with time_horizon = 8. Source: own*

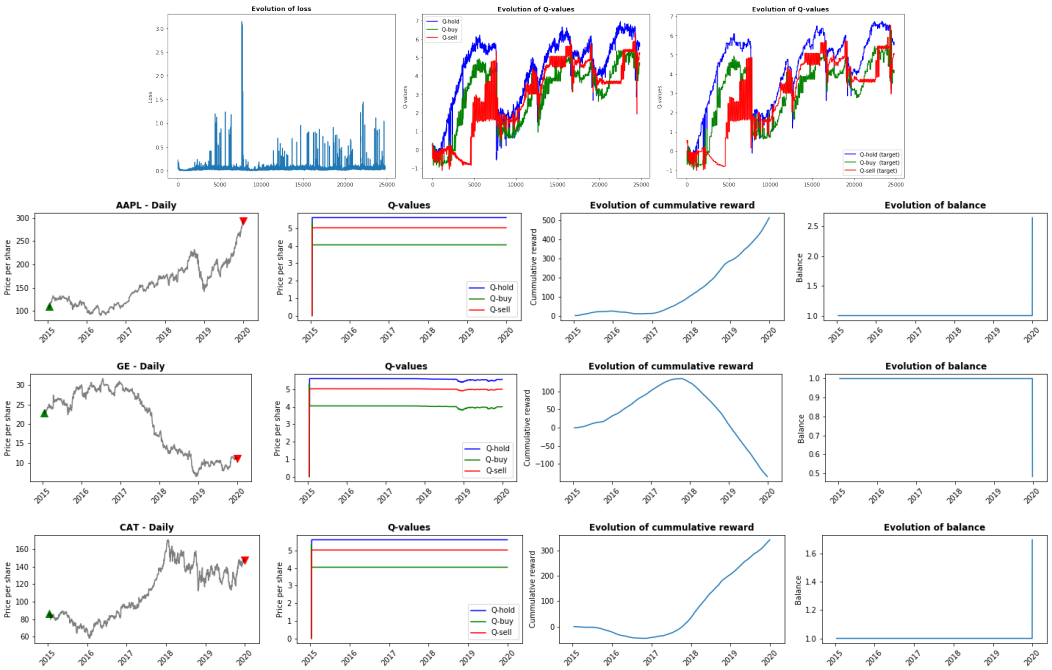

Figure 30: *Diagrams of experiment 2. The first row shows the evolution of the loss and Q-values for an agent trained with the Double Deep Q-learning algorithm [52]. The next rows show the application of the trained agent to different stocks. Train with time_horizon = 8. Source: own*

In these figures we can see many features that will appear constantly in the following runs of the experiment:

- The learning process is even faster than with the Feedforward Neural Network, practically since the first step the algorithm achieves a plateau in the loss and the Q-values. Despite its spiky aspect, when looking at the scale we can see that it reach a level lower than the best ones achieved in the Experiment1. This is common to the three algorithms
- The use of more sophisticated algorithms traduce in a better separation of the Q-values in the tests. As in the Experiment 1, we can see that the gaps between the hold, buy and sell lines is bigger in [15] respect to [14], and even bigger in [50].
- The agent does not generalized very well, we can see that, as in the Experiment 1, despite the change of the trend in the two last time-series, it does not produce the adequate behavior.

5.2.2 time_horizon = 16

The second run of the experiment uses a choice of *time_horizon* = 16. This choice, united to the value of the position of the agent, requires a specific architecture of the Neuronal Network, shown in Figure 31, which joins he information of both inputs, one with a previous encoding by one Bidirectional LSTM and one simple LSTM layer, and other with any modification into one last feedforward layer.

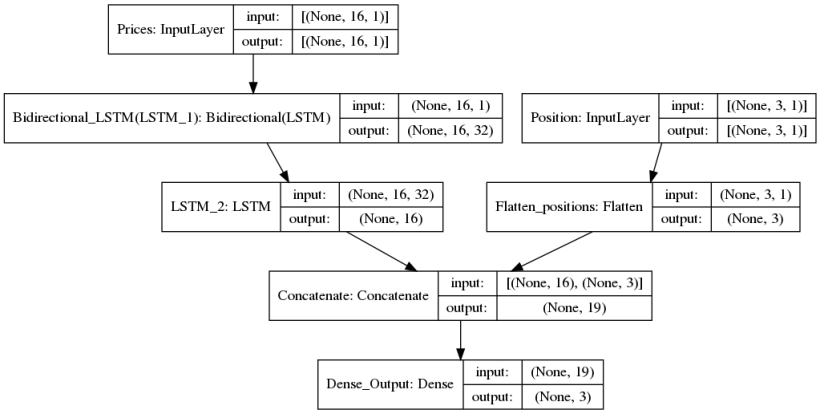

Figure 31: *Architecture of the LSTM Neural Network for a choice of time_horizon* $= 16$. *Source: own*

With this architecture we can build our agent and start to operate. In the following figures we show the training process with the AAPL stock and the test of this trained agent in this stock, and two additional stocks, GE y CAT, to study the capacity of generalization of this training process. In Figure 32 we show and agent trained by the classical Deep Q-learning algorithm [14]; in Figure 33 we show and agent trained by the Deep Q-learning algorithm with target network [15]; and, in Figure 34 we show and agent trained by the Double Deep Q-learning algorithm [52].

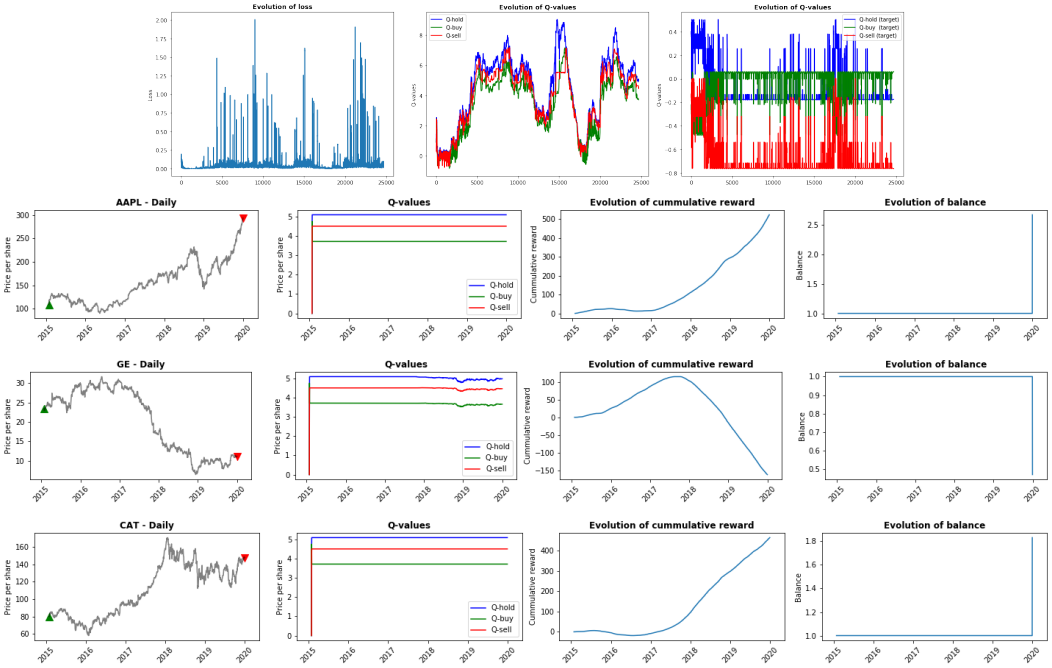

Figure 32: *Diagrams of experiment 2. The first row shows the evolution of the loss and Q-values for a agent trained with the classic Deep Q-learning algorithm [14]. The next rows show the application of the trained agent to different stocks. Train with time_horizon* $= 16$. *Source: own*

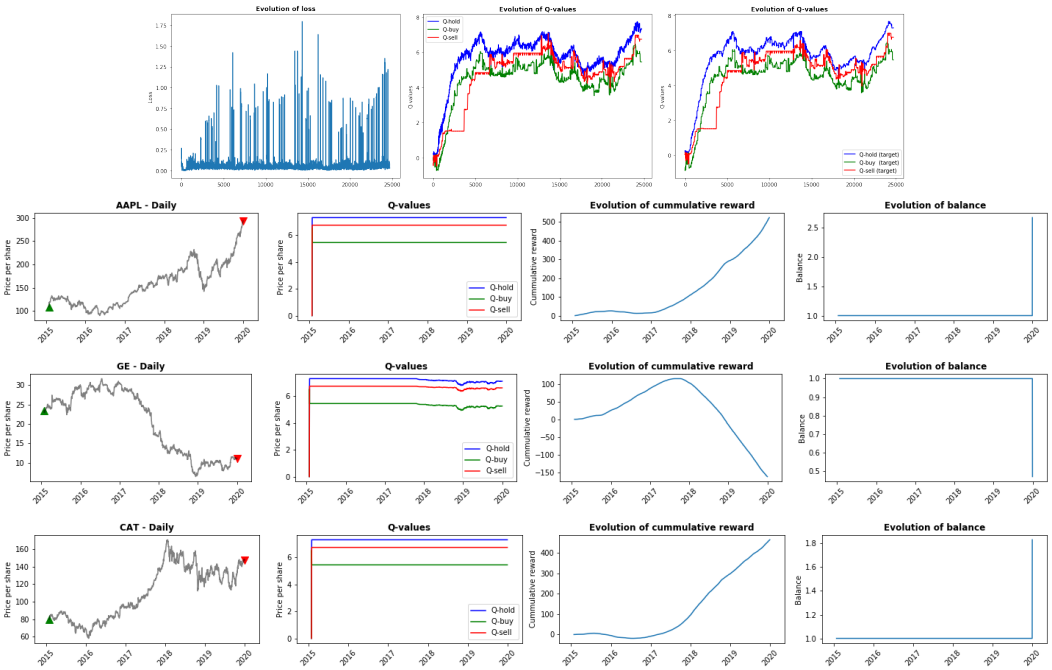

Figure 33: *Diagrams of experiment 2. The first row shows the evolution of the loss and Q-values for a agent trained with the Deep Q-learning algorithm with target network [15]. The next rows show the application of the trained agent to different stocks. Train with time_horizon = 16. Source: own*

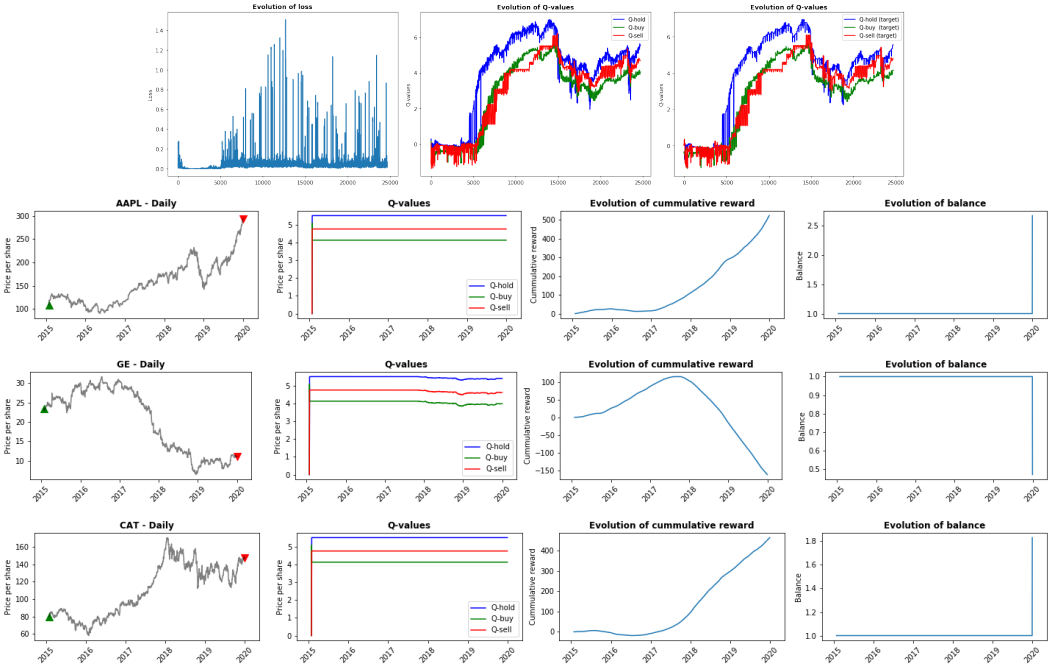

Figure 34: *Diagrams of experiment 2. The first row shows the evolution of the loss and Q-values for a agent trained with the Double Deep Q-learning algorithm [52]. The next rows show the application of the trained agent to different stocks. Train with time_horizon = 16. Source: own*

In these figures we can see more or less the features that in the previous runs of the experi-

ment:

- The learning process is even faster than with the Feedforward Neural Network, practically since the first step the algorithm achieves a plateau in the loss and the Q-values.
- The use of more sophisticated algorithms traduce in a better separation of the Q-values in the tests.
- The agent does not generalized very well, as in the Experiment 1.

5.2.3 time_horizon = 32

The third run of the experiment uses a choice of *time_horizon* = 32. This choice, united to the value of the position of the agent, requires a specific architecture of the Neuronal Network, shown in Figure 35, which joins he information of both inputs, one with a previous encoding by one Bidirectional LSTM and one simple LSTM layer, and other with any modification into one last feedforward layer.

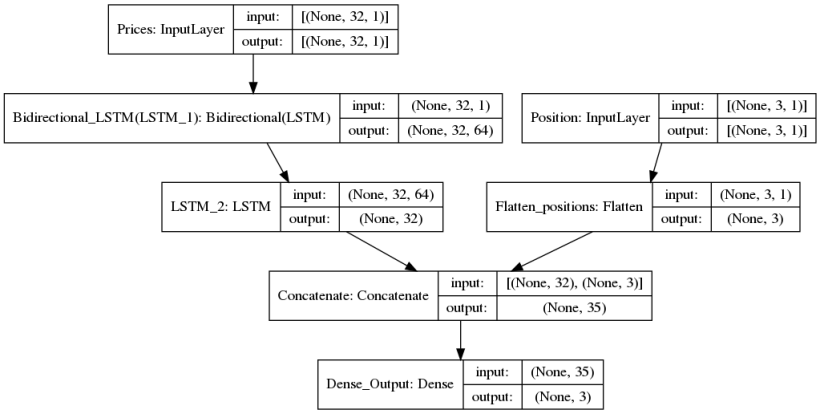

Figure 35: *Architecture of the LSTM Neural Network for a choice of time_horizon* = 32. *Source: own*

With this architecture we can build our agent and start to operate. In the following figures we show the training process with the AAPL stock and the test of this trained agent in this stock, and two additional stocks, GE y CAT, to study the capacity of generalization of this training process. In Figure 36 we show and agent trained by the classical Deep Q-learning algorithm [14]; in Figure 37 we show and agent trained by the Deep Q-learning algorithm with target network [15]; and, in Figure 38 we show and agent trained by the Double Deep Q-learning algorithm [52].

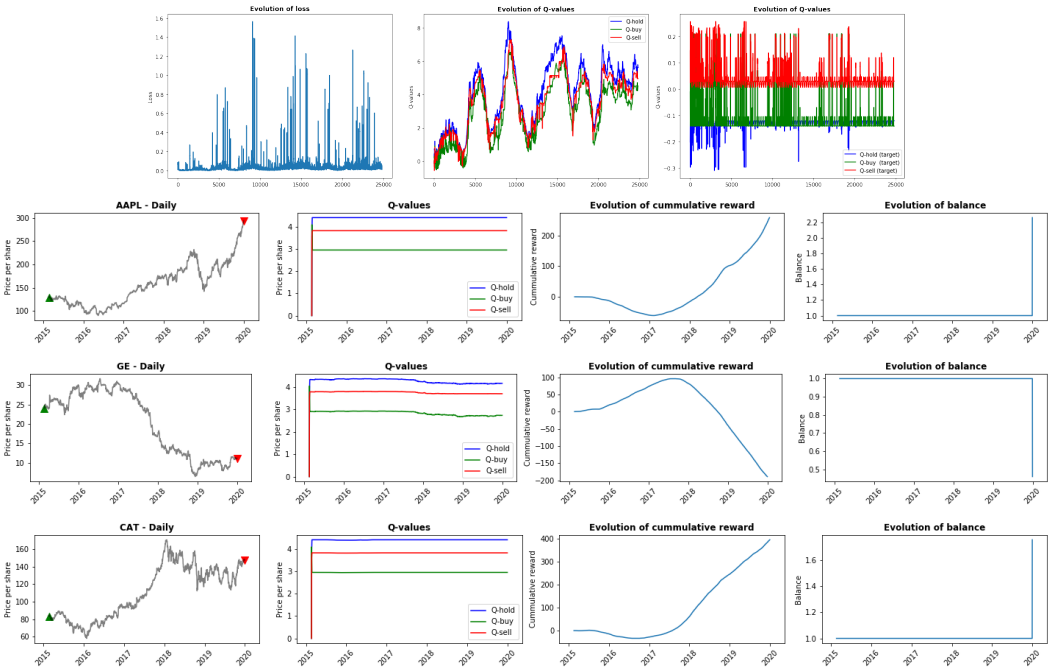

Figure 36: *Diagrams of experiment 2. The first row shows the evolution of the loss and Q-values for a agent trained with the classic Deep Q-learning algorithm [14]. The next rows show the application of the trained agent to different stocks. Train with time_horizon = 32. Source: own*

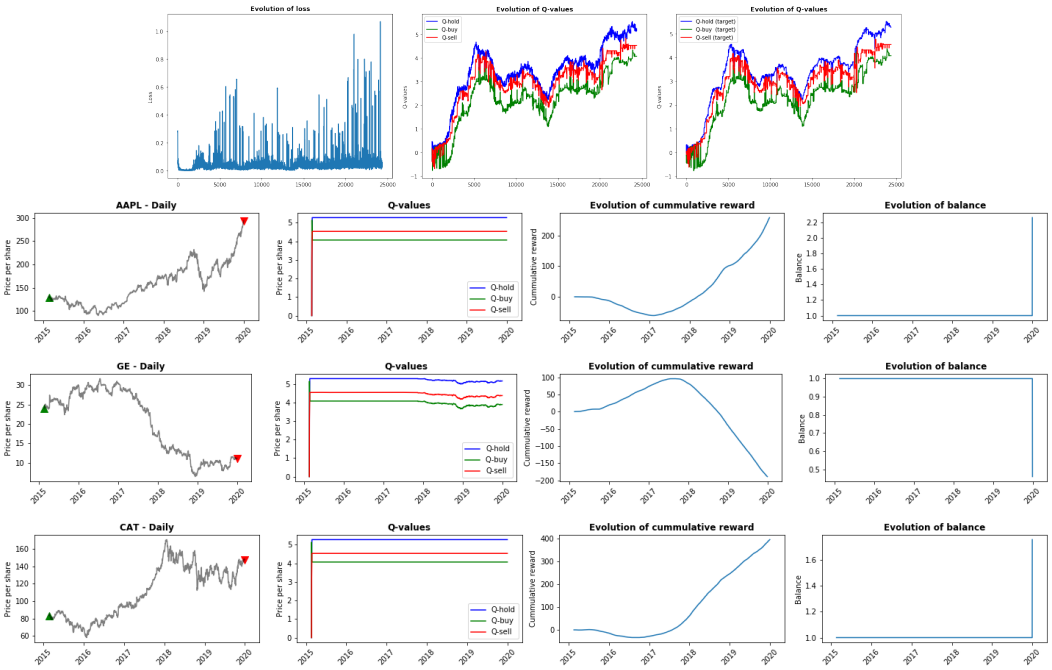

Figure 37: *Diagrams of experiment 2. The first row shows the evolution of the loss and Q-values for a agent trained with the Deep Q-learning algorithm with target network [15]. The next rows show the application of the trained agent to different stocks. Train with time_horizon = 32. Source: own*

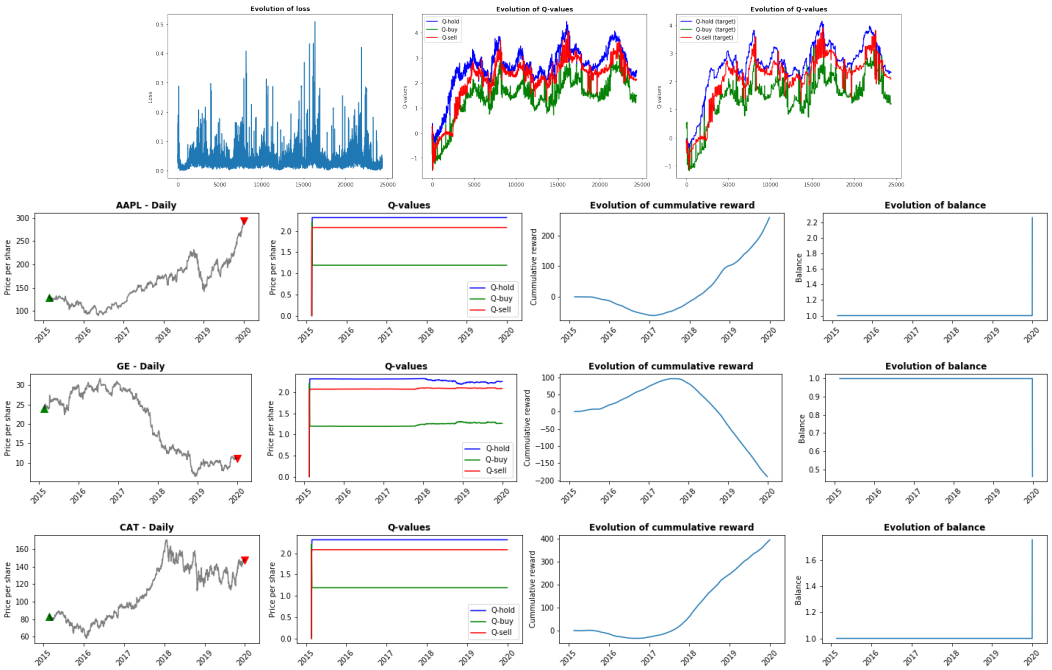

Figure 38: *Diagrams of experiment 2. The first row shows the evolution of the loss and Q-values for a agent trained with the Double Deep Q-learning algorithm [52]. The next rows show the application of the trained agent to different stocks. Train with time_horizon = 32. Source: own*

In these figures we can see more or less the features that in the previous runs of the experiment:

- The learning process is even faster than with the Feedforward Neural Network, practically since the first step the algorithm achieves a plateau in the loss and the Q-values.
- The use of more sophisticated algorithms traduce in a better separation of the Q-values in the tests.
- The agent does not generalized very well, as in the Experiment 1.

5.2.4 Comments

As we said before, the first row of diagram in each run of the experiment shows the evolution of the learning process. So, on the one hand, looking at the evolution of the loss we can see that it converges very fast to a plateau, since the first interaction it reaches optimal values that fluctuate very little, creating a plateau. This is common for the three algorithms, but looking carefully we can see that the more complex algorithms achieve lower values of the loss. Also, looking at the evolution of the Q-values we can see that they reach more stable values since the initial steps. When comparing the diagrams for the different values of *time_horizon*, we can see that the higher this value is, the better are the results in loss and Q-values. This is trivial since the increase of this parameter also increases the information available to the agent in order to take a decision,as in the Experiment 1.

And, on the other hand, when looking at the tests we can see that the use of more complex algorithms make the Q-values of the actions to be more separated, as in Experiment 1, that is, the decisions are better in the sense that the margin that marks the best decision is bigger. The same

effect can be seen when, using the same algorithm, we increase the value of *time_horizon*. However, as observed in Experiment 1, the increase of this parameter induces a lag between the instant when the agent receives the information of a time-steps. In this sense, we are going to be conservative and the are going to choose 32, again, as the best value for the *time_horizon* parameter, since is when the lag starts to be noticeable but still conserves an important quantity of information. Finally, is important to note that the agent still generalizes bad when we change the stock. This can arise from the particular choice of the training stock, or from the fact that we use only one stock to train the agent, producing an unintentional overfitting.

5.3 Training with random stocks

In this experiment we are going to propose a solution to the overfitting observed in Experiments 1 and 2. The idea is to introduce some variance in the training data by the use of different stocks to train the agent. With this approach, we expect that the buffer of states composing the memory for experience turns to be more diversificated. This will make that the agent get experience with different types of trends in the prices, and, consequently, to be able to generalize better, being capable of taking adequate decision for any stock in the tests. In order to avoid the posible bias of taking conclusion by looking only at 3 stocks, we are going to evaluate the balance achieved by the agent over all available stocks of S&P500 index and computing the mean value and the standard deviation.

5.3.1 Feedforward Neural Network

The first run of the experiment uses a choice of *time_horizon* $= 32$, as we said before. This choice, united to the value of the position of the agent, requires a specific architecture of the Neuronal Network, shown in Figure 23. Before training directly with random stocks, we are going to see how the performance of the agent changes depending of the stock it is trained with. In this sense, we are training our agent with the same stocks as before, namely AAPL, GE and CA in order to have a benchmark to compare with the performance of the new agent.

Training stock	**Balance**
AAPL	1.648 ± 1.013
GE	0.330 ± 1.004
CAT	1.221 ± 1.032
Random	1.653 ± 1.018

Table 2: *Mean and standard deviation of the balance of the agent through all the available stocks of S&P500. The architecture of the agent in this case is a Feedforward Neural Network with a time_horizon = 32.*

In Table 2 we show the results of the experiment and we can see that, effectively, the use of random stocks in the training step seems to be very fruitful, it produce the best results. A balance of 1.653 means a 65.3% of benefits in the five years that we using to test the agent, which is a very good profit. We can see that AAPL also produces good results, but since its picked arbitrarily we can totally confide in it despite its good results.

5.3.2 Recurrent Neural Network

The second run of the experiment uses a choice of *time_horizon* = 32, as we said before. This choice, united to the value of the position of the agent, requires a specific architecture of the Neuronal Network, shown in Figure 35. Before training directly with random stocks, we are going to see how the performance of the agent changes depending of the stock it is trained with, as in the previous case. In this sense, we are training our agent with the same stocks as before, namely AAPL, GE and CA in order to have a benchmark to compare with the performance of the new agent.

Training stock	**Balance**
AAPL	1.653 ± 1.004
GE	0.326 ± 1.003
CAT	1.653 ± 1.004
Random	1.670 ± 1.105

Table 3: *Mean and standard deviation of the balance of the agent through all the available stocks of S&P500. The architecture of the agent in this case is a Recurrent Neural Network with a time_horizon = 32.*

In Table 3 we show the results of the experiment and we can see that, also in this case, the use of random stocks in the training step seems to be very fruitful and it produces, in mean terms, a secure win with the investment.

5.3.3 Comments

In this experiment we have seen that, in both cases, when using a Feedforward Neural Network and a Recurrent Neural Network, the use of random stocks in the training step produce very good results, the margins of benefit of out agent are considerably high. The APPL stock also seems to produce good results, but since it is picked arbitrarely we cannot confide in it to operate independetly despite its profits. The use of random stocks to trian the agent is the best way to avoid the possible overfitting and to improve the generalization capabilities of our agent.

5.4 Future work

As a final comment to this work, we propose several ways to continue the work developed in these pages. There are three clear ways to improve the work done here:

- **Environment**: the first way to continue our work is to improve the environment, that is, the simulator of the environment. The environment can be improved by using more desagregated data, that is, hourly or minutely data, which is not usually public but provided by the online brokers on demand. This can be pushed even further, with little modifications in code, by using online data. Also, more realistic fees can be introduced since in general there is a fixed quantity to pay and an additional quantity relative to the stock or the amount of shares in the transaction. Finally, we can introduce more diverse data into the environment to improve the information that the agent has at the moment of taking a decision, that is, economic information about the current situation of the world or sentiment from news regarding that stock for example.

- **Agent**: the second way to continue our work is to improve the agent. The most obvious way to make this improvement is to make it able to operate with quantities, that is, making it able to not only buy and sell, but to choose the quantity of shares to buy and sell. This feature involves to introduce a continuous set of actions, instead of the discrete one that we are using in our work and, consequently, to change the training algorithms. Another implication of the introduction of quantities is the finiteness of the number of shares of each stock is the management of the prices associated with the offer and demand of each stock, with this feature we could also use our agent-environment interface to study the stock market from a realistic and quantitave approach.

- **Algorithms**: the third way to continue our work is to push the training algorithm more towards to the state of the art, in this sense, we can introduce the use of dueling networks [63], noisy networks [64] or a combination of all advances at the same time using the Rainbow model [65]. In the case we chose to introduce continuous actions we also need to use a different set of algorithms, capables of dealing with this requirement, some of the state of the art algorithms to develop this idea are deterministic policy gradient algorithms [66] [67], trust region policy optimization algorithms, [68], proximal policy optimization algorithms [69] or actor-critic algorithms [70].

6 Conclusions

The main goal of this work was to approach ourselves to Reinforcement Learning not only in a theoretical way, but also in a practical way. In order to achieve the first step we have pursued an extensive revision of the literature of the field, reviewing its fundamentals and reaching some state of the art techniques to make it usable to real world projects. Then, we put into practice this knowledge by developing an agent to operate with data from stock market.

In order to elaborate this agent we studied the fundamentals of the stock market and the rules to operate in it. With this knowledge we built an environment simulating the stock market and a generic agent to train with this environment. Then to develop the training process we implement some state of the art Deep Q-learning algorithms using Tensorflow.

Finally, we run a series of experiment to tune the hyperparameters of our agent in order to make it able to achieve the best possible performance for its architecture. In this sense, we conclude that the best architecture is a Recurrent Neural Network with bidirectional LSTM units. Regarding the hyperameters, we conclude that the best time horizon for the agent is 32 days, because it allows the agent to use much information to take decisions without lagging too much the rewards; and the best fee is 0.01, because it makes the environment more realistic and rewards long-term investments rather than short-term buys and sells. Finally, regarding the training data, we conclude that using random stocks to train the agent produce better and more general results than using only one stock.

References

[1] Leo Breiman. "Statistical Modeling: The Two Cultures (with comments and a rejoinder by the author)". In: *Statist. Sci.* 16 (2001), pp. 199–231. DOI: 10.1214/ss/1009213726.

[2] Marcos Lopez de Prado. *Advances in Financial Machine Learning*. Wiley, 2018. ISBN: 9781119482086.

[3] Harry Surden. "Machine learning and law". In: *Wash. L. Rev.* 89 (2014).

[4] Rahul C. Deo. "Machine learning in medicine". In: *Circulation* 132.20 (2015), pp. 1920–1930. DOI: 10.1161/CIRCULATIONAHA.115.001593.

[5] Sean Ekins. "The Next Era: Deep Learning in Pharmaceutical Research". In: *Pharmaceutical Research* 33 (2016), pp. 2594–2603. DOI: 10.1007/s11095-016-2029-7.

[6] Giuseppe Carleo et al. "Machine learning and the physical sciences". In: *Reviews of Modern Physics* 91 (4 2019). DOI: 10.1103/RevModPhys.91.045002.

[7] Rory Bunker and Teo Susnjak. *The Application of Machine Learning Techniques for Predicting Results in Team Sport: A Review*. 2019. arXiv: 1912.11762 [cs.LG].

[8] Kenneth Brant, Jim Hare, and Svetlana Sicular. *Hype Cycle for Artificial Intelligence, 2019*. Technical Report G00369840. Gartner Research, 2019. URL: https://www.gartner.com/en/documents/3953603/hype-cycle-for-artificial-intelligence-2019.

[9] Kun Shao et al. *A Survey of Deep Reinforcement Learning in Video Games*. 2019. arXiv: 1912.10944 [cs.MA].

[10] Gerald Tesauro. "Temporal Difference Learning and TD-Gammon". In: 38.3 (1995), pp. 58–68. DOI: 10.1145/203330.203343.

[11] David Silver et al. "Mastering the game of Go with deep neural networks and tree search". In: *Nature* 529 (2016), pp. 484–489. DOI: 10.1038/nature16961.

[12] David Silver et al. "Mastering the game of Go without human knowledge". In: *Nature* 550 (2017), pp. 354–359. DOI: 10.1038/nature24270.

[13] David Silver et al. *Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm*. 2017. arXiv: 1712.01815 [cs.AI].

[14] Volodymyr Mnih et al. *Playing Atari with Deep Reinforcement Learning*. 2013. arXiv: 1312.5602 [cs.LG].

[15] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". In: *Nature* 518 (2015), pp. 529–533. DOI: 10.1038/nature14236.

[16] Adrien Ecoffet et al. *Go-Explore: a New Approach for Hard-Exploration Problems*. 2019. arXiv: 1901.10995 [cs.LG].

[17] Julian Schrittwieser et al. *Mastering Atari, Go, Chess and Shogi by Planning with a Learned Model*. 2019. arXiv: 1911.08265 [cs.LG].

[18] Christopher Berner et al. *Dota 2 with Large Scale Deep Reinforcement Learning*. 2019. arXiv: 1912.06680 [cs.LG].

[19] Oriol Vinyals et al. "Grandmaster level in StarCraft II using multi-agent reinforcement learning". In: *Nature* 575 (2019), pp. 350–354. DOI: 10.1038/s41586-019-1724-z.

[20] Hua Wei et al. "IntelliLight: A Reinforcement Learning Approach for Intelligent Traffic Light Control". In: *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. ACM. 2018, pp. 2496–2505.

[21] Junqi Jin et al. *Real-Time Bidding with Multi-Agent Reinforcement Learning in Display Advertising*. 2018. arXiv: 1802.09756 [stat.ML].

[22] Hongzi Mao et al. "Resource management with deep reinforcement learning". In: *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*. 2016, pp. 50–56.

[23] Xiangyu Zhao et al. *Deep reinforcement learning for search, recommendation, and online advertising: a survey*. 2018. arXiv: 1812.07127 [cs.IR].

[24] Lex Fridman, Jack Terwilliger, and Benedikt Jenik. *DeepTraffic: Crowdsourced Hyperparameter Tuning of Deep Reinforcement Learning Systems for Multi-Agent Dense Traffic Navigation*. 2018. arXiv: 1801.02805 [cs.NE].

[25] Sen Wang, Daoyuan Jia, and Xinshuo Weng. *Deep Reinforcement Learning for Autonomous Driving*. 2018. arXiv: 1811.11329 [cs.CV].

[26] Yuxi Li. *Deep Reinforcement Learning*. 2018. arXiv: 1810.06339 [cs.LG].

[27] Thomas G. Fischer. *Reinforcement Learning in Financial Markets - a survey*. FAU Discussion Papers in Economics 12/2018. 2018. URL: http://hdl.handle.net/10419/183139.

[28] Frank J. Fabozzi, Franco P. Modigliani, and Frank J. Jones. *Foundations of Financial Markets and Institutions*. 4th. Pearson, 2013. ISBN: 9781292021775.

[29] Zvi Bodie, Alex Kane, and Alan Marcus. *Investments*. McGraw-Hill Education, 2018. ISBN: 9781259277177.

[30] Benjamin Graham and Jason Zweig. *The Intelligent Investor (Revised 1973 edition)*. Harper Business, 2003. ISBN: 9780060752613.

[31] John J. Murphy. *Technical Analysis of the Financial Markets: A Comprehensive Guide to Trading Methods and Applications*. New York Institute of Finance, 1999. ISBN: 9780735200661.

[32] Eugene F. Fama. "Efficient Capital Markets: A Review of Theory and Empirical Work". In: *The Journal of Finance* 25 (1970), pp. 383–417. DOI: 10.2307/2325486.

[33] Ernie Chan. *Quantitative Trading: How to Build Your Own Algorithmic Trading Business*. Wiley Trading. Wiley, 2009. ISBN: 9780470466261.

[34] Ernie Chan. *Algorithmic Trading: Winning Strategies and Their Rationale*. Wiley Trading. Wiley, 2013. ISBN: 9781118460146.

[35] Bjoern Krollner, Bruce Vanstone, and Gavin Finnie. "Financial time series forecasting with machine learning techniques: A survey". In: *Proceedings of the 18th European Symposium on Artificial Neural Networks (ESANN 2010)* (2010), pp. 25–30.

[36] Dev Shah, Haruna Isah, and Farhana Zulkernine. "Stock Market Analysis: A Review and Taxonomy of Prediction Techniques". In: *International Journal of Financial Studies* 7 (2019), pp. 383–417. DOI: 10.3390/ijfs7020026.

[37] Dongdong Lv et al. "An Empirical Study of Machine Learning Algorithms for Stock Daily Trading Strategy". In: *Mathematical Problems in Engineering* 2019 (2019).

[38] Harry Max Markowitz. "Portfolio Selection". In: *The Journal of Finance* 7 (1952), pp. 77–91. DOI: 10.2307/2975974.

[39] Marco Avellaneda and Jeong-Hyun Lee. "Statistical arbitrage in the US equities market". In: *Quantitative Finance* 10 (2010), pp. 761–782. DOI: 10.1080/14697680903124632.

[40] Marcos Lopez de Prado. "Building Diversified Portfolios that Outperform Out-of-Sample". In: *Journal of Portfolio Management* 10 (2016), pp. 761–782. DOI: 110.3905/jpm.2016.42.4.059.

[41] Xiaodong Li et al. "News impact on stock price return via sentiment analysis". In: *Knowledge-Based Systems* 69 (2014), pp. 14–23.

[42] Venkata Sasank Pagolu et al. "Sentiment analysis of Twitter data for predicting stock market movements". In: *2016 international conference on signal processing, communication, power and embedded system (SCOPES)*. IEEE. 2016, pp. 1345–1350.

[43] Lei Zhang, Shuai Wang, and Bing Liu. "Deep learning for sentiment analysis: A survey". In: *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 8.4 (2018).

[44] Terry Lingze Meng and Matloob Khushi. "Reinforcement Learning in Financial Markets". In: *Data* 4 (2019). DOI: 10.3390/data4030110.

[45] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. 2nd. Adaptive Computation and Machine Learning. MIT Press, 2018. ISBN: 9780262039246.

[46] Richard E. Bellman. “A Markovian Decision Process”. In: *Indiana University Mathematics Journal* 6 (1957), pp. 679–684.

[47] Richard E. Bellman. *Dynamic Programming*. Adaptive Computation and Machine Learning. Princeton University Press, 1957. ISBN: 9780691146683.

[48] Gavin A. Rummery and Mahesan Niranjan. *On-line Q-learning using connectionist systems*. Technical Report CUED/F-INFENG/TR 166. Engineering Department, Cambridge University, 1994.

[49] Christopher J. C. H. Watkins and Peter Dayan. “Q-learning”. In: *Machine Learning* 8 (1992), pp. 279–292. DOI: 10.1007/BF00992698.

[50] Hado van Hasselt. “Double Q-learning”. In: *Advances in Neural Information Processing Systems 23*. 2010, pp. 2613–2621.

[51] Martin Riedmiller. “Neural Fitted q Iteration – First Experiences with a Data Efficient Neural Reinforcement Learning Method”. In: *Proceedings of the 16th European Conference on Machine Learning*. ECML’05. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 317–328. ISBN: 3540292438. DOI: 10.1007/11564096_32.

[52] Hado van Hasselt, Arthur Guez, and David Silver. *Deep Reinforcement Learning with Double Q-learning*. 2015. arXiv: 1509.06461 [cs.LG].

[53] Long-Ji Lin. “Self-Improving Reactive Agents Based on Reinforcement Learning, Planning and Teaching”. In: *Machine Learning* 8 (1992), pp. 293–321. DOI: 10.1007/BF00992699.

[54] Tom Schaul et al. *Prioritized Experience Replay*. 2015. arXiv: 1511.05952 [cs.LG].

[55] Greg Brockman et al. *OpenAI Gym*. 2016. arXiv: 1606.01540 [cs.LG]. URL: https://github.com/openai/gym.

[56] Martí Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://github.com/tensorflow/tensorflow.

[57] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. URL: http://www.deeplearningbook.org.

[58] Warren S. McCulloch and Walter Pitts. “A logical calculus of the ideas immanent in nervous activity”. In: *The bulletin of mathematical biophysics* 5 (1943), pp. 115–133. DOI: 10.1007/BF02478259.

[59] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. “Learning representations by back-propagating errors”. In: *Nature* 323 (1986), pp. 533–536. DOI: 10.1038/323533a0.

[60] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2014. arXiv: 1412.6980 [cs.LG].

[61] Adam Paszke et al. “Automatic differentiation in PyTorch”. In: *NIPS-W*. 2017.

[62] Sepp Hochreiter and Jürgen Schmidhuber. “Long Short-Term Memory”. In: *Neural Computation* 9.8 (1997), pp. 1735–1780. DOI: 10.1162/neco.1997.9.8.1735.

[63] Ziyu Wang et al. *Dueling Network Architectures for Deep Reinforcement Learning*. 2015. arXiv: 1511.06581 [cs.LG].

[64] Meire Fortunato et al. *Noisy Networks for Exploration*. 2017. arXiv: 1706.10295 [cs.LG].

[65] Matteo Hessel et al. *Rainbow: Combining Improvements in Deep Reinforcement Learning*. 2017. arXiv: 1710.02298 [cs.AI].

[66] David Silver et al. “Deterministic policy gradient algorithms”. In: 2014.

[67] Timothy P. Lillicrap et al. *Continuous control with deep reinforcement learning*. 2015. arXiv: 1509.02971 [cs.LG].

[68] John Schulman et al. *Trust Region Policy Optimization*. 2015. arXiv: 1502.05477 [cs.LG].

[69] John Schulman et al. *Proximal Policy Optimization Algorithms*. 2017. arXiv: 1707.06347 [cs.LG].

[70] Volodymyr Mnih et al. *Asynchronous Methods for Deep Reinforcement Learning*. 2016. arXiv: 1602.01783 [cs.LG].

[71] Marc G. Bellemare, Will Dabney, and Rémi Munos. *A Distributional Perspective on Reinforcement Learning*. 2017. arXiv: 1707.06887 [cs.LG].