Ca' Foscari
University
of Venice

Master's Degree programme

in

Data Analytics for
Business and
Society

Final Thesis

Exploring the Forex Market with the Reinforcement Learning Agent Deep-Q-Network

Supervisor
Ch. Prof. Marco Corazza

Graduand
Eric Cappellina
Matriculation Number 874846

Academic Year
2022 / 2023

Contents

Introduction

Financial markets are virtual places where both retail and institutional investors seek to grow their capital. Among the various markets available, one stands out: the *Foreign Exchange* (Forex) market, the world's largest. It's where currencies are traded and participants include commercial banks, hedge funds, central banks, businesses, and retail traders who operate daily. Indeed, what's intriguing about Forex is that anyone can participate, from individuals exchanging money while traveling to companies involved in international trade or central banks adjusting interest rates. However, many investors choose to trade Forex with the goal of making a profit over time. This leads to the question: *Is it really possible to be profitable in financial markets?* Economists have attempted to answer this question over the years and among the most prominent theories are the *Efficient Market Hypothesis* (EMH) and the *Adaptive Market Hypothesis* (AMH) developed by Eugene Fama and Andrew Lo, respectively. These theories offer contrasting ideas. EMH posits that the market is perfectly efficient, and investors always behave rationally, implying that all available information is already reflected, making it challenging for traders to be consistently profitable. In contrast, AMH supporters believe that participants do not always act rationally, especially during financial crises when emotions can lead to mistakes. Assuming AMH holds some truth, this research aims to leverage advancements in the field of AI by developing an intelligent artificial agent for trading in the Forex market. Specifically, this study focuses on a machine learning model belonging to the *Reinforcement Learning* (RL) family. The fundamental concept behind RL is that an agent learns from experience in an environment using a reward-punishment system, which makes the model completely different with respect to the supervised learning framework.

The agent studied and implemented in this research is the *Deep-Q-Network* (DQN) and its improvement, the *Double-Deep-Q-Network* (DDQN), which both gained fame for achieving human-expert-level performance in Atari games. The agents are tested over a four-year period, spanning from January 2019 to December 2022, using 30-minute timeframes, yielding positive results and highlighting the potential of this technology.

The structure of this work is divided into three sections: the first section introduces the Forex market and its terminology, followed by an explanation of EMH and AMH, and concludes with an overview of the artificial intelligence field. The second section provides a theoretical explanation of the Reinforcement Learning framework, covering terminology and various solutions to the RL problem, including dynamic programming, Monte Carlo methods, and temporal difference learning. It concludes with a description of the DQN agent and its improvements. The final section serves as the bridge between the previous chapters, presenting the implemented model's structure, the conducted experiments, and their associated results.

Chapter 1

Background

1.1 The Forex Market

The Forex market is where currencies are traded, and it holds the title of being the largest market in the world, with a daily trading volume exceeding $7.5 trillion in April 2022 [27]. This immense size is due to the crucial role of currency exchange in activities like travel, investments, and business operations. Participants in the Forex market include commercial banks, hedge funds, central banks, investment firms, businesses, investors, and retail traders. What sets the Forex market apart from the stock market is that it doesn't have a central exchange for order-taking and transaction execution. Instead, it works as an over-the-counter (OTC) market, enabling direct trading between two parties without a controlling authority. This is made possible by a vast network of computers, allowing traders from anywhere in the world to trade whenever and wherever they choose. As a result, the Forex market operates 24 hours a day, excluding weekends, with four main trading sessions in New York, London, Sydney, and Tokyo.

1.1.1 History

The Forex market has undergone significant transformations over time, resulting in its current structure and functioning. The origins of the Forex market can be traced back to the barter system practiced as early as 6000 BC. In this system, people exchanged goods and services directly, without the use of a standardized medium

of exchange. As civilizations developed and expanded their trading networks, the need for a more efficient means of exchange arose. This led to the emergence of commodity money, such as seashells, beads, and livestock, which were widely accepted as a form of payment.

Over time, the concept of currency evolved, with the introduction of metallic coins in ancient Greece and Rome. These coins, made from precious metals like gold and silver, became widely accepted and facilitated trade across borders. As civilizations interacted through trade routes, the need for foreign currency exchange emerged. Money changers and merchants began offering exchange services, allowing individuals to convert one currency into another.

In 1875, the gold standard monetary system was established, which further influenced the development of the Forex market. Under the gold standard, the value of a country's currency was directly linked to a fixed amount of gold ensuring that each currency had a tangible and universally accepted value. Currencies were freely convertible into gold at a fixed rate, promoting stability in exchange rates. The gold standard facilitated international trade and investment, as it provided a reliable benchmark for currency valuations. However, maintaining the gold standard became increasingly challenging, particularly during periods of economic crisis or war when countries required flexibility in monetary policy. These challenges eventually led to the abandonment of the gold standard and the emergence of alternative exchange rate regimes.

The Forex market as we know it today took shape in the post-World War II era. In 1944, representatives from 44 countries convened in Bretton Woods, New Hampshire, to establish a new international monetary system. The Bretton Woods agreement resulted in fixed exchange rates pegged to the U.S. dollar, with the dollar itself linked to gold. However, the Bretton Woods system faced challenges in the 1960s, as economic imbalances and inflationary pressures increased. In 1971, President Richard Nixon suspended the convertibility of the U.S. dollar into gold, effectively ending the Bretton Woods system. This move led to the era of floating exchange rates, where currencies were allowed to fluctuate based on supply and demand in the foreign exchange market.

Technological advancements played a pivotal role in the transformation of the Forex market. The development of electronic trading platforms and the Internet in the 1990s revolutionized currency trading. These innovations enabled traders to access real-time market data, execute trades electronically, and participate in the global Forex market from anywhere in the world. As a result, the market witnessed a significant increase in trading volumes, with a wide range of participants, including commercial banks, investment firms, hedge funds, businesses, investors, and retail traders.

The Forex market witnessed another significant milestone with the introduction of the euro in 1999. The euro, the single currency of the European Union (EU), was launched electronically and became the official currency of 11 EU member states. This historic event aimed to promote economic integration and facilitate trade among European nations. The introduction of the euro had a profound impact on the Forex market since it eliminated exchange rate fluctuations and exchange costs between the participating countries, making cross-border trade more efficient. The euro also increased the liquidity and depth of the Forex market, as it became one of the most widely traded currencies globally. Moreover, the euro's establishment encouraged further financial integration within the European Union, leading to the creation of a single monetary policy under the European Central Bank (ECB).

1.1.2 Some Terminology

The Forex market, as previously mentioned, centers around the trading of currencies. Unlike other types of securities, currencies inherently possess their own value and cannot be traded individually. Consequently, the Forex market employs the concept of currency pairs to make trading possible. This involves comparing the value of one currency to another so as to determine the *foreign exchange rate* between them. The most widely traded currency pair is EUR/USD, which compares the euro and the US dollar (represented by their respective symbols). The currency on the left (euro) is known as the *base currency*, while the currency on the right (US dollar) is referred to as the *quote currency*. The exchange rate reflects the amount of the quote currency needed to purchase one unit of the base currency.

Essentially, when buying the EUR/USD currency pair (going long), one is acquiring euros and simultaneously selling US dollars. On the other hand, when selling the EUR/USD currency pair (going short), one is selling euros and simultaneously buying US dollars. Since the exchange rate can be seen as the price of the base currency in terms of the quote currency, from now on I will use the term *price* to refer to the exchange rate.

Actually, once the trader has chosen the currency pair to trade on, online brokers show two different prices: the *ask price* and the *bid price*. The former is the price the trader has to pay whenever he goes long, while the latter is the price at which the trader goes short. The ask price is always higher than the bid price and the difference is known as *spread*. This concept is fundamental in the Forex market because ignoring it could affect the trading performance, especially for high-frequency trading strategies. Indeed the spread is the inherent commission a trader pays whenever he enters a trade. To illustrate why the spread is considered a commission, let's consider a scenario where a trader decides to go long on EUR/USD and purchases it at the current ask price of 1,08896. However, the trader changes his/her mind immediately and decides to close the position by going short on EUR/USD at the bid price of 1,08886. As a result, the trader incurs a loss equal to 0,0001, the difference between the bid and ask prices, referred to as the spread. Online brokers provide a service by executing these buy and sell orders on behalf of the trader, and it is reasonable for them to be compensated, in fact, the spread serves as a means of payment to brokers. Depending on the broker, the spread can either be fixed, remaining constant over time, or variable, fluctuating in value. Traders generally prefer trading with lower spreads to minimize their costs. It is worth noting that the most frequently traded currency pairs, such as EUR/USD, typically have the lowest spreads.

Currency pairs exhibit small price movements, typically measured in decimals. To address this characteristic, a unit of measurement known as a *pip* (percentage in point) has been introduced. The pip is the standard measure to indicate price changes in currency pairs, specifically referring to the fourth decimal digit, except for currency pairs involving the Japanese yen (JPY), where the pip represents the

second decimal place. For even finer granularity, the *pipette* is used, corresponding to the movement of the fifth decimal digit (or the third decimal digit in JPY pairs). The pipette essentially represents a fraction of a pip.

In order to account for small price changes in trading, fixed contract sizes called *lots* are used. A lot represents a specific number of units of the base currency. The *standard lot* consists of 100.000 units, the *mini lot* consists of 10.000 units and the *micro lot* consists of 1.000 units. It is important to note that a trader cannot open a position with a different lot size. Lots are essential for calculating the value of 1 pip and, consequently, determining the overall profit or loss of a trade. For example, if a trader's account is denominated in US dollars and he/she opens a position with the standard lot on the EUR/USD currency pair, each pip corresponds to a value of $1. This value is derived by multiplying the standard lot size (100.000) by the pip value (0.0001). By understanding this, the trader can easily calculate the final profit or loss once the position is closed.

The size of trading lots in the forex market directly influences the amount of money required to start trading. Fortunately, online brokers offer the *leverage*, which allows traders to open positions with more money than they actually have. Leverage increases trading power and enables larger positions. However, leveraging can multiply both profits and losses, making it a risky strategy. The maximum leverage offered by online brokers is typically 30. To open a leveraged position, traders must provide a certain amount of money known as *margin*. If the trader's equity falls below a specific margin level, the broker issues a *margin call*. This notifies the trader to add more funds to the trading account or close some open positions. If the trader ignores the margin call and continues to experience losses, the broker will automatically close all open positions. To illustrate this concept, let's consider an example. Imagine a trader with $1000 in their account who decides to use margin trading with a broker that requires a 100% margin level to activate a margin call. The trader chooses to open a position with a mini lot size of $10.000 and a leverage of 20. In this case, the required margin would be $500 (calculated by dividing the contract size by the leverage used). Initially, the trader's equity would be $10.000, consisting of $500 in margin and $9.500 in free margin.

Unfortunately, the trade goes poorly, resulting in a loss of $9.500. Consequently, the equity decreases to $500, and the margin level reaches 100% (since equity and margin are now equal), triggering a margin call from the broker. Once a margin call is issued, the trader cannot open any new positions until the equity exceeds the margin. The trader's options at this point are to deposit more funds, close the position, or hope that the open position reverses in their favor. If the trader ignores the margin call and the open position fails to reverse, the equity will continue to decrease. When it reaches another predetermined level known as the *stop-out level*, determined by the broker, the broker will immediately close the position.

1.2 Trading Strategies

Generally, the primary objective of a trader is to generate profits over an extended period of time. However, it is anything but easy, there are plenty of variables influencing the financial market, especially Forex, and the competition is fierce. Consequently, engaging in trading can be regarded as a profession necessitating prior education and preparation. Professional traders employ various techniques to assist them in making informed decisions and enhance the likelihood of predicting market trends. Essentially, there are two primary approaches. Some traders rely on studying the factors that can impact the market and make decisions based on that analysis, a method known as *fundamental analysis*. Conversely, other traders prefer analyzing price charts using statistical tools to forecast market movements, known as *technical analysis*.

More specifically, fundamental analysis involves examining the factors that could potentially impact a specific security. For instance, in the stock market, it can be beneficial to study a company's balance sheet, financial statements, cash flow, and management, as well as macroeconomic factors such as the overall economic situation and industry-specific conditions related to the traded security. The main idea is to evaluate companies (e.g. discounting future cash flows to the present) and find out if the company evaluation reflects the current price of the share. If the intrinsic value is lower than the current price, indicating overvaluation, it may

be wise to take a short position. Conversely, if the intrinsic value is higher than the current price, indicating undervaluation, it may be beneficial to buy shares. Indeed fundamental analysis assumes that in the short term, a company's value may not align with its actual price, but over the long run, price and value are expected to converge, creating opportunities for profit. Due to the time required for data publication (monthly, quarterly, yearly) and the slower pace of change for companies, fundamental analysis is considered more of an investment activity than trading, as it requires a longer-term perspective.

A clear distinction should be made between fundamental analysis in the stock market and the Forex market. While the previous examples were applicable to the stock market, the Forex market involves currencies rather than businesses, resulting in different variables influencing market dynamics. These variables are related to the macroeconomic field, politics, and import-export activities. In the Forex market, inflation plays a crucial role in determining the value of a country's currency compared to foreign currencies. When a country experiences higher inflation, its national currency tends to lose value relative to foreign currencies. Another closely connected variable is the interest rate. Central banks adjust interest rates to manage inflation. If inflation is too high, they may increase the interest rate, and if inflation is too low, they may decrease it. Raising the interest rate can strengthen the currency because it attracts more foreign investments, resulting in an increase in its value. Conversely, lowering the interest rate can have the opposite effect. It is important to note that these variables should be analyzed in a comparative manner between countries, rather than considering them in an absolute term. Another significant variable is the balance of trade, which reflects import and export activities between countries. Businesses impact the balance of trade, as an increase in exports raises the balance of trade, and vice versa. When the balance of trade is positive, indicating higher exports, the currency's value tends to rise due to increased demand for the national currency from foreign companies. Conversely, a negative balance of trade indicates higher imports, resulting in the national currency's devaluation compared to foreign currencies. Politics and government debt also have an impact on the Forex market. However, the most crucial variable is

investor expectations because market movements are driven by supply and demand. Economic news, for example, can be interpreted negatively, leading to a decline in prices. It is worth noting that expectations are closely tied to future events. Therefore, even if current inflation is rising, if investors anticipate a decrease in the near future, market trends may not align with the current situation.

In contrast, technical analysis involves examining price charts using statistical tools to predict market trends. The most widely used chart is the candlestick chart, which consists of candles representing specific time intervals determined by the trader. These candles display the opening and closing prices through their bodies, while lines extending from the bodies indicate the highest and lowest prices observed during that timeframe. Technical analysis differs significantly from fundamental analysis because its main objective is forecasting price movements rather than determining intrinsic value. Technical traders rely on technical indicators as their primary tools. These indicators involve transforming historical price data into a different format, allowing traders to identify potential trading signals. The main categories of technical indicators are *trend indicators*, *oscillators*, and *volatility indicators*. Trend indicators help determine the direction and strength of a trend (such as moving averages), oscillators assist in identifying trend reversals (like the RSI), and volatility indicators measure the extent of price fluctuations (such as Bollinger Bands). These indicators are often accompanied by trading rules that trigger entry and exit points. Trading rules are based on specific values or conditions of technical indicators, which indicate when to initiate or exit a trade. Basically, technical indicators and trading rules work together to identify optimal entry and exit points in trading. Unlike fundamental analysis, technical analysis focuses more on short-term market movements rather than long-term trends, in fact, the timeframe for analysis can vary from days to hours or even minutes. Technical analysis operates on three main assumptions: price movements follow trends, and identifying them early can be highly profitable; historical patterns tend to repeat themselves, making past data analysis useful in predicting future price behavior; price reflects all relevant information, making the analysis of variables other than price (such as fundamental factors) unnecessary.

1.3 Market Hypothesis

For decades, researchers have debated the profitability of trading in financial markets. Two main theories, the *efficient market hypothesis* (EMH) and the *adaptive market hypothesis* (AMH), aim to address this question. The EMH, developed by Eugene Fama in his 1970 book "*Efficient Capital Markets: A Review of Theory and Empirical Work*" suggests that prices already incorporate all available information due to market efficiency (the efficiency of the market tends to increase as the number of participants in the market grows). As a result, consistently making risk-adjusted excess returns is deemed impossible. According to the EMH, securities are traded at their fair value, making both technical analysis and fundamental analysis ineffective. While there have been traders who have made profits from financial markets, the EMH attributes their success to luck rather than skill. The EMH recommends investing in passive portfolios, which are commonly used as benchmarks, instead of attempting to outperform the market. The EMH encompasses three forms: *weak*, *semi-strong*, and *strong*. The weak form suggests that current prices only reflect past prices, making technical analysis useless. In this case, fundamental analysis becomes the sole means of potential profitability. The semi-strong form proposes that current prices incorporate all publicly available information, making both technical and fundamental analysis ineffective. Only private information can potentially lead to profits. The strong form asserts that current prices reflect all possible information, including both public and private data. Consequently, no information can be used to generate profits, and investing in passive portfolios that track the overall market is considered the only viable approach.

Andrew Lo, an MIT professor, introduced the adaptive market hypothesis in 2004. It is a combination between EMH and behavioral finance. Different from Eugene Fama's theory, AMH takes inspiration from the principles of evolution (competition, adaptation, and natural selection). The main point behind this theory is that participants are not completely rational but during specific conditions like crises or financial bubbles, they make irrational decisions. This is due to cognitive bias such as overreaction or loss aversion. According to AMH, investors adopt a heuristic approach to the market, they aim at satisficing behavior not maximizing it

(not perfectly rational). The theory is characterized by three assumptions: people are motivated by self-interest; people naturally make mistakes; people learn from mistakes. Investors, guided by their experience, employ a trial-and-error approach, adopting strategies that have proven successful in the past and modifying them when they no longer yield desired results. Basically, according to the adaptive market hypothesis, implementing effective trading strategies is feasible, because investors are not completely rational.

1.4 Automated Trading Systems and A.I.

Technical analysis-based trading, as introduced in the previous section, means taking into account specific technical indicators and finding a particular trading rule based on them. More specifically, this means setting some parameters, both for technical indicators and the trading rule. Once the parameters have been chosen, the trading activity can be automated so that a computer can open and close positions according to the specified trading rule. According to the Federal Reserve, more than two-thirds of trades come from automated trading systems in the Forex market. Most of the automated trading systems in circulation rely on technical analysis, however, there are more sophisticated trading systems based on artificial intelligence. Broker platforms have their programming language to create trading strategies inside their platform. The only limit is that these strategies can be only based on technical analysis. An artificial intelligence-based trading system is more difficult to implement since it would require the strategy to be implemented with a programming language external to the broker platform (e.g. C, Java, Python, etc.). Furthermore, to open a position with this kind of system, the programming language should interact with the broker platform through an API (application programming interface). The AI-based trading system, despite being more complex to implement, is more flexible and potentially more profitable than the standard technical analysis trading system.

Generally, automated trading systems have some advantages over traditional ones. For instance, emotions play a relevant role in trading activity. Indeed,

emotional feelings (fear, hesitation, over-optimism) may prevent the trader to stick to the plan. Another strength point is the possibility of backtesting, which means testing the trading strategy on historical data to evaluate it and find out the return expectations. Other advantages are the execution speed, which is higher than traditional trading, and the decrease in errors. On the other hand, a potential drawback may be a technology failure such as a connection loss.

Artificial intelligence-based trading systems usually overperform the ones relying on simple technical analysis. But what is artificial intelligence? Alan Turing, one of the greatest AI pioneers, questioned whether a machine can think and he introduced the so-called *Turing test*, also known as the imitation game, in his 1950 paper "Computer Machinery and Intelligence" to answer this question. This game consists of a human evaluator who judges a conversation between a person and a machine (all the participants are separated from one another). If the evaluator is not able to distinguish the machine from the person, the machine would pass the test. The term *artificial intelligence* was coined for the first time by John McCarthy, who is considered the father of AI, in a 1956 summer program hosted by Marvin Minsky and John McCarty himself at Dartmouth College. Marvin Minsky's definition of AI is "the science of making machines do things that would require intelligence if done by men."

Artificial intelligence systems can be divided into three categories based on their ability to imitate human behavior. The first is *Artificial Narrow Intelligence* (ANI), or weak AI, which is the only available system nowadays. It's programmed to perform a specific task without the ability to go outside the task it is designed to perform. This kind of system, unlike humans, lacks consciousness. The other kind of AI system is *Artificial General Intelligence* (AGI), or strong AI, which is currently unavailable in the real world. It has been used in many science fiction movies where humans interact with self-conscious machines. AGI systems would be able to perform different and general tasks so that it would prevent distinguishing humans from machines (for instance AGI machines would pass the Turing test). The third category is *Artificial Super Intelligence* (ASI) which is a hypothetical system capable of surpassing human intelligence. ASI machines would be far superior to

humans so they may be responsible for humans' extinction from the planet.

The artificial intelligence aim of obtaining human capabilities could not be achieved without *machine learning* which is considered an AI subfield. Machine learning is based on statistical tools and algorithms that are applied to large data sets so as to make a machine capable of performing a specific task. The machine learning field can be divided into three main categories: *supervised learning*, *unsupervised learning*, and *reinforcement learning*. The former deals with a data set composed of input-output pairs. The goal is to find a function that maps input features to the output variable. It is called supervised learning because human intervention is needed to write the label associated with the corresponding inputs. On the other hand, the unsupervised learning framework consists of a data set without labels with the goal of finding an underlying structure in the data. One of the most used unsupervised learning techniques is clustering which tries to find clusters in the data, useful for several fields like marketing, biology, information retrieval, etc. It is called unsupervised because the output labels are not present. The third category is reinforcement learning which differs completely from the previous two. Indeed, the idea is that an agent interacts with an environment in order to maximize a reward function (a learning-through-experience approach). So, if artificial intelligence has the goal of mimicking human cognitive capabilities, machine learning is more like a tool to achieve that goal.

Another AI subfield is *deep learning*, which is considered a machine learning subfield as well. It differs from the latter for the kind of algorithms employed, in fact, deep learning takes inspiration from the human brain, more specifically from neurons. Indeed, artificial neural networks are the most relevant model in this subfield which revolutionized the whole artificial intelligence industry. Their importance is due to the wide range of applications such as computer vision, natural language processing, time series forecasting, etc. Neural networks are very flexible thanks to their architecture composed of layers of neurons connected together. They can be divided into three main categories: *artificial neural network* (ANN) is the standard model, *convolutional neural network* (CNN) is designed to perform computer vision tasks like image recognition and object detection, *recurrent neural*

network (RNN) is used when the data follow a specific order such as language translation in which the order of words matters, stocks prediction where the data is chronologically ordered. Actually, a simple neural network with only one hidden layer falls into the machine learning field, in particular in the supervised learning class. It's only when the architecture becomes huge that neural networks fall into the deep learning field, for instance when the number of hidden layers is greater than two. Generally, the choice of developing a machine learning model or a deep learning one depends on the problem to solve and the amount of data at disposal. Indeed, neural networks need a very large amount of data to be trained on.

Figure 1.1 perfectly describes how artificial intelligence, machine learning, and deep learning are related to each other.

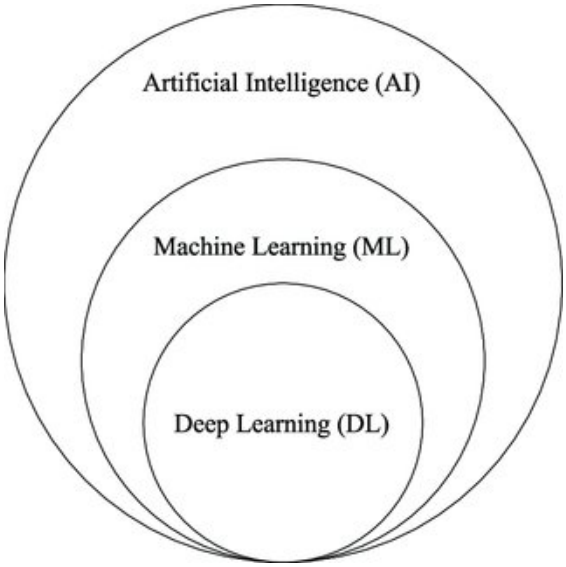

Figure 1.1: Artificial intelligence vs machine learning vs deep learning. Source: Roy Rupali, Towards Data Science, *AI, ML, and DL: How not to get them mixed!*

Chapter 2

Reinforcement Learning and Deep-Q-Network Algorithm

Computer scientists have been working on mimicking human intelligence, or even overcoming it, through computers. The most known forms of machine learning techniques to be developed for succeeding in this purpose are the supervised and unsupervised ones. These two frameworks are used to solve several tasks such as image recognition, recommender systems, object detection, language translation, and many others. By looking at the vast amount of things a machine is able to deal with, the goal of building an intelligent computer seems to be realized. However, if we consider the way by which humans learn, the goal of reproducing a machine with human-like capabilities using supervised or unsupervised learning is far from being achieved. Indeed the supervised learning field requires a data set of observations followed by the true outcomes, the so-called labels, so as to allow the machine to learn from examples. The difference is that superior living beings learn over time through experience, not by examples. Moreover, unsupervised learning's objective is to find the hidden pattern in the data (a hidden structure) which is even further from the concept of human learning.

Reinforcement learning is the third kind of machine learning which is based on this idea: an agent that interacts with an environment in order to reach a certain goal. This approach is the closest to the kind of learning superior living beings experience and it is inspired by biological learning. Two important factors

to consider about the reinforcement learning paradigm are the absence of any instructions and the fact that the actions may affect the near future as well as long-term situations. The former reiterates again the difference from supervised learning, but it also gives us an idea of how powerful this technique can be, just think of all those activities that can not be explained by examples because of the infinite amount of possibilities involved (driving a car, playing chess etc.); without reinforcement learning, they could not be automized in any way. The latter focuses the attention on actions consequences, they not only influence the immediate future but can have an impact in the long run (an investment in the present time may be profitable ten years later). Given its similarity with human behavior, reinforcement learning is applied in several fields such as gaming, robotics, and autonomous cars.

The following sections are dedicated to explaining the theoretical ideas behind this machine learning system as well as the mathematical formulas necessary to develop such a model. All the information is taken from the most relevant book in the reinforcement learning field: "*Reinforcement Learning: An Introduction*" second edition by Sutton and Barto.

2.1 The Reinforcement Learning Framework

As anticipated in the introduction, there is a decision-maker, the *agent*, which interacts with the *environment*, it includes whatever is outside the agent. The interaction is made up of actions computed by the agent and the responses given by the environment; this relationship happens repeatedly. More specifically the procedure is the following: the agent and the environment interacts at each discrete time step $t = 0, 1, 2, 3, \cdots$ At each time step a specific *state* $S_t \in \mathcal{S}$, where $\mathcal{S}$ is the set of possible states, describes the environment and based on the state the agent selects an *action* $A_t \in \mathcal{A}(S_t)$, where $\mathcal{A}(S_t)$ is the set of actions available in state S_t. In the next time step, the agent receives a *reward* $R_{t+1} \subseteq \mathcal{R}$ as a consequence of its previous action, as well as a new environment representation, S_{t+1}. Figure 2.1 explains the whole process. The action is selected on the basis of a *policy* π_t, a mapping from states to actions, given a state it returns the probability of selecting

each available action in state S_t. Reinforcement learning algorithms change the agent policy over time through experience in order to maximize the cumulative future reward.

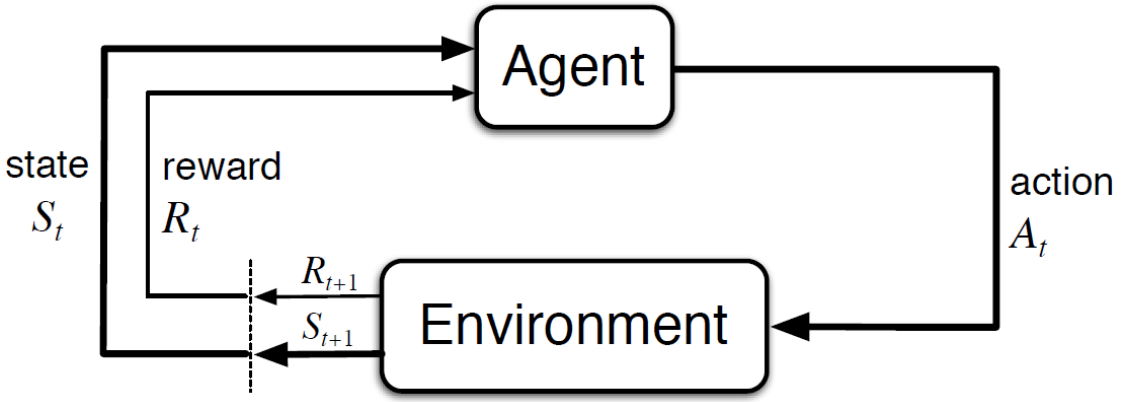

Figure 2.1: The interaction between the agent and environment in the RL framework. Source: Sutton and Barto, *Reinforcement Learning: An Introduction*

2.1.1 Reward Signal

The agent at each time step performs an action and in the following time step, the environment sends back a reward, which is a scalar number. The reward signal informs the agent of the results obtained from the previous action. The purpose is to understand the best-performing actions over time. The agent's main goal is not getting the maximum reward every time but it's to maximize the future cumulative reward. This means that is more desirable to select an action that collects a lower reward in the next time step if it brings the agent into a situation in which the rewards will be higher in the long run. It's worth noting the key importance of the reward signal: through it, the agent will adjust its behavior (its policy). For this reason, it should be set carefully in terms of what we want to achieve, not how we want to achieve the final goal. This distinction is crucial when designing our reinforcement learning framework. For instance, a chess-playing agent should be rewarded only for winning the game and not for taking the opponent's pieces. This mistake will ruin the agent's performance since it will try to take the opponent's pieces at every cost, even at losing the game. The reward is one of the distinctive

elements in RL and its flexibility allows the application of this algorithm in a wide range of domains.

More formally in time step t the future reward is a sequence of rewards after time t denoted as R_{t+1}, R_{t+2}, $R_{t+3}, \cdots$ The agent's goal is to maximize the expected return, where the return is a specific function of the reward sequence. The most simple case is the sum of the future rewards:

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + \cdots + R_T, \tag{2.1}$$

where T is the last time step. Obviously, (2.1) makes sense for those applications in which there is effectively the final time step, this is the case for the episodic tasks (the games usually fall within this family). The problem with this expression arises when the time steps are not in a finite range, this is the case in continuing tasks, for instance, an agent which simulates a robot with a long life span. Indeed the cumulative reward can easily take an infinite value. In order to avoid this issue, the previous formula is modified with:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}, \tag{2.2}$$

where γ, $0 \leq \gamma \leq 1$, is the so called *discount factor*. Basically, it is the present value of future rewards. It's very common in economics for comparing different investments (each of them will return some money in the future and, in order to put them in the same time horizon, they are discounted to the present time; the investment with the highest discounted value is the most convenient). With this mathematical trick, the sum of future rewards converges to a finite number, if $\gamma < 1$. If $\gamma = 0$, the agent is myopic, in the sense that it tries to maximize the next future reward, without caring about the long-term rewards. This choice is not good since maximizing the next reward could prevent the agent from obtaining even higher rewards in the long run. On the other hand, $\gamma = 1$ makes the agent farsighted: it takes into account the future rewards more strongly.

2.1.2 State Signal and Markov Property

The agent-environment interaction could not be possible without the agent's perception of the environment. At each time step the environment sends to the

agent the state signal which is a description of the environment itself. When the agent receives the state signal it chooses an action based on that. Since the state description is responsible for the agent's choice of each action, it should be designed in a proper way. Ideally, it should contain both the current information and past sensations in order to give a complete description of the environment. Past information does not mean including all the environment dynamics that lead to the current time step but a processed measure that takes into account the past sensations of the environment in a compact way. If the state signal retains all the relevant information is said to have the *Markov property*. For instance in a chess game, if the state signal describes the environment with the position of all the pieces in the table, it is a Markovian state since this information is all that matters to choosing the next move in the game, without knowing all the moves that led to the current configuration.

More formally, the probability of obtaining a reward r and a new state s' depends on the whole past history dynamics:

$$\Pr\{R_{t+1} = r, S_{t+1} = s' \mid S_0, A_0, R_1, \ldots, S_{t-1}, A_{t-1}, R_t, S_t, A_t\}. \tag{2.3}$$

However, if the environment is Markov, the one-step dynamics depends only on the current state and action:

$$p(s', r \mid s, a) = \Pr\{R_{t+1} = r, S_{t+1} = s' \mid S_t, A_t\}. \tag{2.4}$$

If and only if (2.3) and (2.4) are equal the environment is said to be Markov. This property is essential in the reinforcement learning domain since it is possible to predict the next rewards by only a function of the current state which is exactly the RL objective. Obviously, not all the applications can be designed in such a way that stick to the Markov property, but starting from this idea it helps to understand even the non-Markov environments.

2.1.3 Policy and Value Functions

Reinforcement learning algorithms are mainly divided into two families based on the approach used to learn an optimal policy: *policy-based* and *value-based* methods.

The latter learns a *value function* that estimates the expected long-term return of being in a particular state and taking a particular action, and from these values, a policy is obtained. Policy-based methods, on the other hand, directly learn the policy that can be used to select actions, rather than estimating the value functions. From now on we will focus on the value-based methods, so it is important to deepen the concept of value functions.

First of all, before explaining the meaning behind value functions, may be useful to revise what a policy is. The policy is the mapping from state to actions, basically, when the agent at time step t receives the state signal from the environment, it selects an action through the policy π_t. There are mainly two kinds of policies in the reinforcement learning field: *deterministic policy* and *stochastic policy*. The former, given the state signal, select an action. The latter, instead of selecting a single action for each state, assigns probabilities to different actions. The agent then samples from this distribution to select one of them. The selected action may vary for the same state due to the inherent randomness introduced by the policy.

Coming back to the value functions, they are functions of a state (or state-action pair) and try to estimate how good it is to start from that state (or from that state and take a specific action). The goodness is expressed in terms of the expected return, so the value functions estimate the expected return starting from the state s (or starting from state s and selecting action a) and following policy π. In fact, the concept of value functions is strictly connected to the concept of policy: value functions are defined with respect to particular policies.

More formally, the value function v_π of state s under policy π is the following:

$$v_\pi(s) \doteq \mathbb{E}_\pi \left[G_t \mid S_t = s \right] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s \right], \text{ for all } s \in \mathcal{S}. \tag{2.5}$$

More specifically, v_π is called *state value function* for policy π and it's important to note that the state value function of a terminal state, if any, is always equal to zero.

In the same manner, we can describe the value of performing action a in state s under a policy π as follows:

$$q_\pi(s, a) \doteq \mathbb{E}_\pi \left[G_t \mid S_t = s, A_t = a \right] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s, A_t = a \right] \tag{2.6}$$

The function q_π is referred to as the *action-value function* for policy π.

Value functions have a very important recursive property that allows the current value function to be expressed in terms of the next time step value function:

$$v_\pi(s) = \sum_a \pi(a \mid s) \sum_{s',r} p\left(s', r \mid s, a\right) \left[r + \gamma v_\pi\left(s'\right)\right]. \tag{2.7}$$

(2.7) is the *Bellman equation* for v_π. Basically, it is an expected value; it looks ahead from one state to its possible successor states, averages over all the possibilities, weighting each by its probability of occurring.

The value functions are useful for their role of ordering the policies in order to find the optimal one. From Sutton and Barto's book:

«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, $\pi \geq \pi'$ if and only if $v_\pi(s) \geq v'_\pi(s)$ for all $s \in \mathcal{S}$. There is always at least one policy that is better than or equal to all other policies. This is an 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, denoted v_*, and defined as

$$v_*(s) = \max_\pi v_\pi(s), \tag{2.8}$$

for all $s \in \mathcal{S}$. Optimal policies also share the same optimal action-value function, denoted q_*, and defined as

$$q_*(s, a) = \max_\pi q_\pi(s, a), \tag{2.9}$$

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

The value function v_*, being the optimal value function, must satisfy (2.7). However, the consistency condition of v_* can be expressed in a different manner, detached from any particular policy:

$$v_*(s) = \max_{a \in \mathcal{A}(s)} \sum_{s',r} p\left(s', r \mid s, a\right) \left[r + \gamma v_*\left(s'\right)\right]. \tag{2.10}$$

(2.10) is the *Bellman optimality equation* and expresses the relationship between the value of a state and the expected return of the best action from that state, under an optimal policy.

The same reasoning can be applied to the state-action value function and its Bellman optimality equation is given by:

$$q_*(s,a) = \sum_{s',r} p\left(s', r \mid s, a\right) \left[r + \gamma \max_{a'} q_*\left(s', a'\right)\right]. \tag{2.11}$$

The Bellman optimality equation has a unique solution that can be achieved with a number of equations equal to the number of states in the environment. Once the optimal value function has been obtained, solving the reinforcement learning problem is straightforward. In fact, the resulting optimal policy would be selecting, in each state, the action that will lead to the next state having the highest value function, the so-called *greedy policy*. A greedy policy refers to a decision-making strategy where an agent always chooses the action that appears to be the most immediately rewarding or promising based on its current estimates. The agent simply selects the action that maximizes its expected immediate reward without considering the potential long-term consequences or exploration of other options Basically, the greedy policy looks one step ahead to compare all the possible actions an agent can perform in a given state, and, on the basis of that, an action is selected. At first, seems like the long-term consequences are not taken into account because the agent chooses the action only by looking at the following time step while the distant future is not considered, but this is not true. The fascinating role of the value function is to give a state its goodness estimate where the goodness is the expected future return starting from that state. This means that inherently the value function takes into account the long-term rewards, it makes the distant future available in the present time. As a consequence, looking at the next time step in order to choose the proper action, means considering the long term as well.

Trying to solve the reinforcement learning problem in this way, with a system of equations, is nearly impossible for real-world tasks. First, a model of the environment's dynamics is needed, which means specifying the probability of obtaining a reward r and a new state s', and it's usually not possible. Second, most of the time the computational resources are insufficient to solve the equations. Lastly, the state signal should have the Markov property which is not always the case. In real-world applications, the reinforcement learning problem is solved using a heuristic approach: instead of finding the optimal policy, a sub-optimal policy is

learned. In the following sections, some of the most famous methods for solving the RL problem are presented.

2.2 Dynamic Programming

As anticipated in the previous paragraph there are multiple ways to solve a reinforcement learning kind of problem. *Dynamic programming* (DP) is one of them. It is a combination of algorithms that try to find the best policy through the value functions estimate. In the real world dynamic programming solution is not used because a full model of the environment is needed. The model allows us to predict the one-step dynamics such as the probability of getting a reward r and a new state s'. Another limitation is the high computational expense which prevents DP to be used in real-world applications. However dynamic programming is necessary because all the other solutions' goal is to solve the reinforcement learning problem with the same effect as DP, without the model and with less computation. The last limitation is the fact that DP is proven to find the best policy for a finite Markov decision process (MDP), but if the state space or action space is continuous, the convergence to an optimal policy can not be proved, indeed a finite MDP environment is assumed.

Dynamic programming makes use of value functions in order to find the optimal policy. The *policy evaluation* algorithm is an iterative process, through which, a value function for a given policy is estimated. Indeed, as we said earlier, the value functions need necessarily a policy to be computed. So the policy evaluation algorithm starts with a random policy π. Each state is assigned a random value function (except for the terminal states which are always assigned a value of 0). Successively the Bellman equation for v_π (2.7) is used to update the value function for each state:

$$v_{k+1}(s) = \sum_a \pi(a \mid s) \sum_{s',r} p\left(s', r \mid s, a\right) \left[r + \gamma v_k\left(s'\right)\right]. \tag{2.12}$$

In particular, v_{k+1} is computed using the next expected reward and the past value function of the next state. It can be proved that as k tends to infinity the estimated

value function tends to the policy value function. We can see how the estimated value functions depend on the estimate of the next state value functions, in other words, the estimate is based on another estimate and this concept is known as *bootstrapping*.

The purpose of computing the value functions for an arbitrary policy is to find better policies. The *policy improvement* algorithm is an iterative process that tries to improve the current policy. Suppose the value function of a given policy has been computed with the policy evaluation algorithm. Now the objective is to find out whether there is a better policy than the starting one. It can be proved that by making a policy greedy with respect to the current value function, the new policy is better than, or as good as, the first one:

$$\pi'(s) = \arg\max_a \sum_{s',r} p\left(s', r \mid s, a\right)\left[r + \gamma v_k\left(s'\right)\right]. \tag{2.13}$$

If the new greedy policy is as good as the previous one, it can be proved that both two policies must be optimal, so the policy improvement algorithm enhances a policy except when it is already optimal.

Until now we have seen how policy evaluation and policy improvement work for just one iteration but it is not enough to find the optimal policy. *The policy iteration algorithm* applies both policy evaluation and policy improvement until optimal policy convergence. The iterative process is the following:

$$\pi_0 \xrightarrow{\text{E}} v_{\pi_0} \xrightarrow{\text{I}} \pi_1 \xrightarrow{\text{E}} v_{\pi_1} \xrightarrow{\text{I}} \pi_2 \xrightarrow{\text{E}} \cdots \xrightarrow{\text{I}} \pi_* \xrightarrow{\text{E}} v_*,$$

where $\xrightarrow{\text{E}}$ denotes a policy evaluation and $\xrightarrow{\text{I}}$ denotes a policy improvement. Basically, the two processes interact with each other until the policy can not be improved, meaning that an optimal policy has been found. The policy iteration algorithm supposes that each process starts only if the other has ended. The problem is that the policy evaluation algorithm converges to v_π only in the limit implying too much computational time. Fortunately, there are variations of such approaches that allow the policy evaluation process not to be ended until the policy improvement starts. For instance, the *value iteration* algorithm provides a single iteration of policy evaluation between each policy improvement.

In general, the term *generalized policy iteration* (GPI) is used to refer to this interaction process between policy evaluation and policy improvement independent of the granularity of the two processes. Figure 2.2 explains the whole procedure. Each of the two lines represents a solution for one of the two processes. Starting from an arbitrary policy the policy evaluation algorithm solves the evaluation problem driving the value function to the upper line, while the policy improvement algorithm solves the problem of improving the policy driving the policy to the bottom line. These two processes compete with each other because the policy improvement makes the value function inconsistent and the policy evaluation makes the policy no more greedy. This competition can be seen in Figure 2.2 since each process drives toward its own line representing its solution and at the same time leaves the other line. On the other hand, the two processes cooperate because each one drives towards the optimal solution where the two lines converge, the point in which the optimal value function is consistent with the optimal policy and the optimal policy is greedy with respect to the optimal value function.

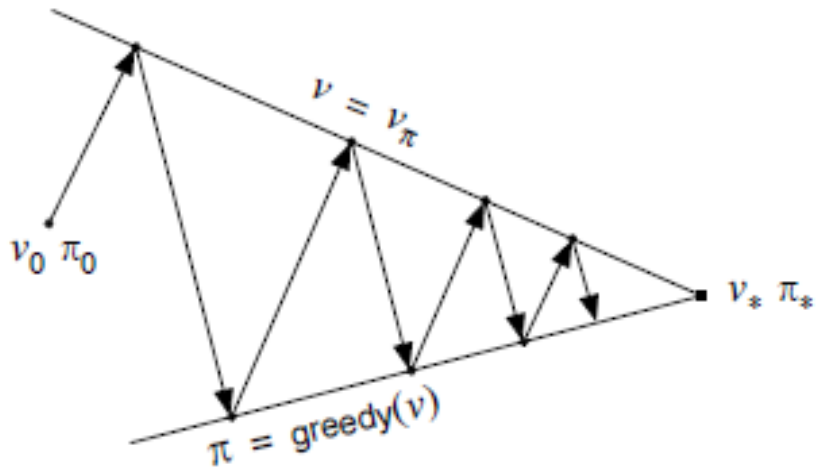

Figure 2.2: The GPI methodology. Source: Sutton and Barto, *Reinforcement Learning: An Introduction*

2.3 Monte Carlo Methods

Dynamic programming is a solution to the reinforcement learning problem but there are some drawdowns that prevent DP to be widely used, especially in real-world cases. Indeed it is not always possible to define a model of the environment's dynamics, furthermore, the computational time and memory can become really huge. *Monte Carlo* methods aim to overcome these issues. The Monte Carlo approach provides a kind of learning by experience, without making explicit a model of the environment (model-free solution). Unlike dynamic programming which seems more like planning, Monte Carlo methods are more similar to superior living beings learning, using experience as the main driver to improvement. On the other hand, they have an important limitation: their application is limited to episodic tasks. This is due to the main idea behind Monte Carlo methods which is estimating value functions by averaging returns. Since the latter is only available at the end of each episode, value functions need an episode to end in order to be updated. As a consequence continuous tasks, given the lack of a final time step and the resulting return, can not be solved with this technique. Another drawdown, linked to the previous limitation, is that the agent learns episode by episode and not in an online fashion.

The Monte Carlo method aims to find the optimal policy by estimating value functions. The approach is the same as dynamic programming and indeed the GPI methodology is used: value function estimation and policy improvement interact with each other until the convergence to an optimal policy is met. The value function estimation algorithm differs from the DP one. Supposing we want to approximate a state value function for an arbitrary policy π, given a set of episodes and passing through s: in an episode, each time a state s is encountered, it is called a visit to s. This state can be visited multiple times within the same episode. The initial visit to it within an episode is referred to as the first visit to s. The *first-visit* Monte Carlo method estimates the value of s by calculating the average of the returns following the first visit to it. On the other hand, the *every-visit* Monte Carlo method calculates the average of the returns following all visits to s. It can be proved that both algorithms converge to v_π if the number of visits to

the state s approaches infinity. However an adjustment needs to be done: since the lack of a model of the environment's dynamics, state value functions are not sufficient to solve the reinforcement learning problem. Indeed, as we have seen in dynamic programming, having the model allows the agent to look one step ahead and choose the action which leads to the best state value function. This can not be done without knowing the probability of obtaining a specific reward and new state, so the state-action value function must be estimated instead. Fortunately, it's the same process as state value functions: in an episode, a state-action pair (s, a) is considered visited if the state s is encountered and the action a is taken. The every-visit Monte Carlo method estimates the value of a state-action pair by calculating the average of the returns following all visits to that pair. The first-visit Monte Carlo method, on the other hand, calculates the average of the returns following the first occurrence of the state-action pair in each episode. Both methods converge to v_π as the number of visits to each state-action pair approaches infinity. This assumption can be a problem because some state-action pairs are never visited in the case of deterministic policies, indeed for each state, the same action is always selected. In order to overcome this issue the adoption of stochastic policies is a possible solution, so that every action has a non-zero probability to be selected. Another idea is the implementation of exploring starts which means starting each episode in a random state-action pair even if can be problematic for several real-world applications and so the first solution seems the best one.

After having estimated the state-action value function for an arbitrary policy, it's time to improve it using the policy improvement procedure. It works, like DP, by making the new policy greedy with respect to the estimated value function. This time, since we are dealing with state-action value functions, a greedy policy means selecting the action that presents the highest value function:

$$\pi(s) = \arg\max_a q(s, a) \tag{2.14}$$

2.4 Temporal-Difference Learning

Both dynamic programming and Monte Carlo methods are able to solve the reinforcement learning problem for a finite Markov decision process. DP solution is based on estimating state value functions (or state-action value functions) and finding an optimal policy. The estimate relies upon another estimate, the next state value (or next state-action pair value), the so-called bootstrapping. DP's main limitation is the need for a model of the environment's dynamics. Monte Carlo methods can be seen as an improvement over dynamic programming since the model is no more needed, indeed this approach enables the agent to learn from experience. Monte Carlo's main drawback is the limited application capability because it's only effective for episodic tasks. *Temporal-difference* learning (TD) is one of the most important improvements in the reinforcement learning field and it's widely used in real-world applications. TD is a combination of dynamic programming and Monte Carlo methods, in fact like DP it estimates value functions using another estimate (it bootstraps), and like Monte Carlo learns from experience without a model of the environment. The resulting algorithm converges to an optimal policy without the drawbacks which characterize the other two approaches.

Temporal-difference learning can be split into prediction and control problems. The former deals with estimating value functions given an arbitrary policy, and the latter improves the policy making it greedy with respect to the current value function. Starting from the prediction problem, it is very similar to the Monte Carlo one but it differs from it because of the target value. If Monte Carlo updates the value functions using the actual return G_t as a target, TD uses the next state (or state-action pair) value and the reward along the way as a target:

$$V(S_t) \longleftarrow V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)]. \tag{2.15}$$

The target in Monte Carlo methods is considered an estimate because the expected value is not known. Instead, a sample return is used as a substitute for the true expected return. On the other hand, in DP methods, the target is estimated not because of the expected values themselves, which are assumed to be provided by a

model of the environment, but because the value of $v_\pi(S_{t+1})$ is unknown. Hence, the current estimate $V(S_{t+1})$ is used in its place. In TD methods, the target is an estimate for both reasons: it samples the expected return and relies on the current estimate V rather than the actual v_π. As a result, TD methods effectively combine the sampling approach of Monte Carlo with the bootstrapping technique of DP. TD and Monte Carlo are called sample backups because their estimates are based on a sample successor state (or state-action pair) differently from DP which needs the complete distribution of all possible successors. This makes both TD and Monte Carlo efficient in terms of memory and computational time.

As anticipated earlier, the TD algorithm is made of two distinct parts interacting with each other: prediction (estimation of the value function for a given policy) and control (policy improvement) problems, as stated in the GPI procedure. The former has two main approaches: *on-policy* and *off-policy*. Both of them rely upon state-action value functions rather than state-value functions because, like Monte Carlo methods, TD is model free. More specifically the on-policy TD updates the state-action value functions using:

$$Q(S_t, a_t) \longleftarrow Q(S_t, a_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, a_{t+1}) - Q(S_t, a_t)], \tag{2.16}$$

which is very similar to (2.15). After each transition from a nonterminal state S_t, this update is performed. If the subsequent state S_{t+1} is a terminal state, then $Q(S_{t+1}; A_{t+1})$ is set to zero. (2.16) incorporates all five elements of a transition (the current state S_t, the action taken A_t, the reward received R_{t+1}, the next state S_{t+1}, and the next action A_{t+1}). The combination of these five elements gives the algorithm its name, *Sarsa*. The control problem for the Sarsa algorithm is just making the policy greedy with respect to the current state-action value function. These two processes keep interacting until an optimal policy is found: it can be proved that Sarsa converges to an optimal policy if all state-action pairs are visited an infinite number of times and the policy converges in the limit to the greedy policy.

The off-policy TD, also known as *Q-learning* (Watkins, 1989) differs from the previous one since, in the prediction problem, it implements the following expression:

Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
  Initialize S
  Repeat (for each step of episode):
    Choose A from S using policy derived from Q (e.g., ϵ-greedy)
    Take action A, observe R, S'
    $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
    $S \leftarrow S'$;
  until S is terminal

Figure 2.3: Q-learning algorithm. Source: Sutton and Barto, *Reinforcement Learning: An Introduction*

$$Q(S_t, a_t) \longleftarrow Q(S_t, a_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, a_t)]. \tag{2.17}$$

(2.17) directly approximates the optimal state-action value function, regardless of the policy being followed. The policy still plays a role as it determines which state-action pairs are visited and updated. If all state-action pairs keep on updating, the state-action value function has been shown to converge.

2.5 Function Approximation

Until now we have assumed finite Markov decision processes, which means environments with a discrete set of states, actions, and rewards. This kind of setup allows value function approximation to be in a tabular form, where each entry corresponds to a state or state-action pair. The issue arises when an environment can't be modeled in such a way but its description consists of an infinite set of states, in this case, it's not possible to approximate value functions in a tabular form. The same reasoning is applied to an infinite set of actions an agent can perform. Even if these sets of states/actions are not continuous but still huge, the computational time, the memory, and the data needed would be a problem to take into consideration. Ideally, an agent should *generalize*: from experiencing a subset of states it should be able to approximate over a much larger subset; indeed in

several real-world tasks, the majority of encountered situations will be entirely novel and have no previous identical experiences. Fortunately, generalization capability can be obtained from the most known kind of machine learning: supervised learning. In the RL field, the term *function approximation* is used to indicate the supervised learning task of learning from examples, taken from a desired function (e.g., value function), in order to approximate the entire function.

As usual, the GPI procedure is adopted so as to find better policies and it is composed of prediction and control problems. The former starts from an arbitrary policy π and estimates the value function v_π for that specific policy. With function approximation, the value function can't be in a tabular form but it's a parametrized function with parameter vector $\mathbf{w} \in \mathbb{R}^n$, so $\hat{v}(s, \mathbf{w})$ is the approximation of state s given the weight vector $\mathbf{w}$. For instance, $\hat{v}$ could be a neural network, and the weight vector $\mathbf{w}$ all the connections weights between neurons. The training data for supervised learning tasks consists of observations with the respective label (the true value). In RL the training data is made of backup examples, more specifically a backup can be represented by the notation $s \longrightarrow v$ where s is the state-backed up and v is the backed-up value or target. This procedure allows applying all the supervised learning algorithms (linear regression, decision tree, neural network, etc.) as function approximators. However, there are some problems with RL that make some algorithms less suitable. For instance, in the supervised learning framework, a static training set is used to train the model over which multiple passes are made, whereas in RL the training data constantly changes due to the online learning of the agent. Another issue is the non-stationarity of target functions in GPI control methods, where the objective often involves learning q_π while the policy π changes. Remarkably, even if the policy remains unchanged, the target values of training examples exhibit nonstationarity when generated through bootstrapping methods such as DP and TD.

The value function approximator must be evaluated, as we do in supervised learning problems, with a metric, for instance, the *root-mean-squared error* (RMSE):

$$RMSE(\mathbf{w}) = \sqrt{\sum_{s \in \mathcal{S}} d(s)[v_\pi(s) - \hat{v}(s, \mathbf{w})]^2} \tag{2.18}$$

where d is a distribution over states and its role is to weight errors based on the importance of each state. In fact, the number of components of $\mathbf{w}$ is smaller than the number of the states, this means that improving approximation for a subset of states results in worse approximation for another subset. The distribution d tries to balance this trade-off. An interesting distribution is an *on-policy distribution* that characterizes how frequently states are encountered when an agent interacts with the environment and selects actions based on a specific policy π. It is called the on-policy distribution because it represents the distribution of backups in on-policy control methods. By minimizing the error over the on-policy distribution, we can allocate the resources for function approximation towards the states that actually occur while following the policy, disregarding those that never happen.

The objective is to find a parameter vector $\mathbf{w}$ for which the RMSE is lower than the RMSE of any other possible parameter vector (global minima). This achievement is occasionally feasible for simple function approximators like linear models but impractical for more complex approximators like neural networks; the latter can only aspire to find local minima.

One of the most used learning methods to approximate the value function is the so-called *gradient descent*. From Sutton and Barto's book: «In gradient-descent methods, the parameter vector is a column vector with a fixed number of real-valued components, $\mathbf{w} = (w_1, w_2, \cdots, w_n)$ and the approximate value function $\hat{v}(s, \mathbf{w})$ is a smooth differentiable function of $\mathbf{w}$ for all $s \in \mathcal{S}$. We will be updating $\mathbf{w}$ at each of a series of discrete time steps, $t = 1, 2, 3, \cdots$, so we will need a notation $\mathbf{w}_t$ for the weight vector at each step.» Assuming that in each training backup example, the true value function $v_\pi(S_t)$ is given and that the states follow the same distribution over which the RMSE is being minimized, the procedure is to reduce the error from the examples. Gradient-descent methods achieve this by making small adjustments to the parameter vector after each example, in a direction that would result in the greatest reduction of the error:

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha \left[v_\pi \left(S_t \right) - \hat{v} \left(S_t, \mathbf{w}_t \right) \right] \nabla \hat{v} \left(S_t, \mathbf{w}_t \right), \tag{2.19}$$

where α is a positive step-size parameter, and $\nabla f(\mathbf{w}_t)$ represents the vector of

partial derivatives with respect to the components of the weight vector:

$$\left(\frac{\partial f(\mathbf{w}_t)}{\partial w_{t,1}}, \frac{\partial f(\mathbf{w}_t)}{\partial w_{t,2}}, \ldots, \frac{\partial f(\mathbf{w}_t)}{\partial w_{t,n}}\right).$$

This vector is the gradient of f with respect to $\mathbf{w}_t$. The weight vector is adjusted in such a way that is proportional to the negative gradient because it refers to the direction in which the error decreases most rapidly.

Unfortunately, the true value $v_\pi(S_t)$ for each training example is not known so (2.19) should be modified with the following expression:

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha[V_t - \hat{v}(S_t, \mathbf{w}_t)]\nabla\hat{v}(S_t, \mathbf{w}_t), \tag{2.20}$$

where V_t is an approximation of the true value $v_\pi(S_t)$ (e.g., $V_t = G_t$ in Monte Carlo methods). The parameter vector $v_\pi(S_t)$ converges if, and only if, the approximator V_t is an unbiased estimate, which is the case for Monte Carlo methods. On the other hand, TD learning estimates are biased so the convergence proof is not met.

It's important to recognize that selecting $\mathbf{w}$ means choosing a specific value function and, as a consequence, any change to the weight vector $\mathbf{w}$ will inevitably change the value function. Since each update will change $\mathbf{w}$ this means that any update to $\mathbf{w}$ will inevitably change the value estimates for many states contrary to the other methods we have seen earlier where updating one state value never impacted the another.

2.6 Neural Networks

Throughout their history, humans have tried to mimic other living beings in order to improve certain activities or even discover new aspects from scratch. There is even a specific word that explains this human behavior of mimicking natural patterns: *biomimicry*. This is the case, for instance, of planes that take inspiration from birds and bats, in fact, aviation pioneers studied the animal world so as to emulate aerodynamics. Intelligence is another example, indeed neuroscientists' studies of the human brain are applied by engineers to build intelligent machines. Before introducing the advancements in the artificial intelligence field, may be useful to explain how the human brain works from a high-level perspective.

Our brain is one of the most complex biological structures. It can be seen as an information processing device composed of nervous cells, the so-called *neurons*. Each neuron can differ in size but the components, shown in Figure 2.4, are the same. The *dendrites*, which resemble tree branches, have the role of receiving electrical signals from other neurons in the form of chemicals called *neurotransmitters*. The body cell, also known as *Soma*, is the area in which the nucleus containing the DNA resides. The body cell is responsible for processing the electrical signals received. The processing involves a weighted aggregation and, if the processed electrical signal is strong enough, it is sent to the next component of the neuron, the *Axon*. This area is covered with a material called *Myolin* which prevents the signal from being degraded. It transports the new signal to the terminal part of the Axon, where *Synapses* have the important role of releasing the electrical signals to other neurons' dendrites so that the whole process is repeated.

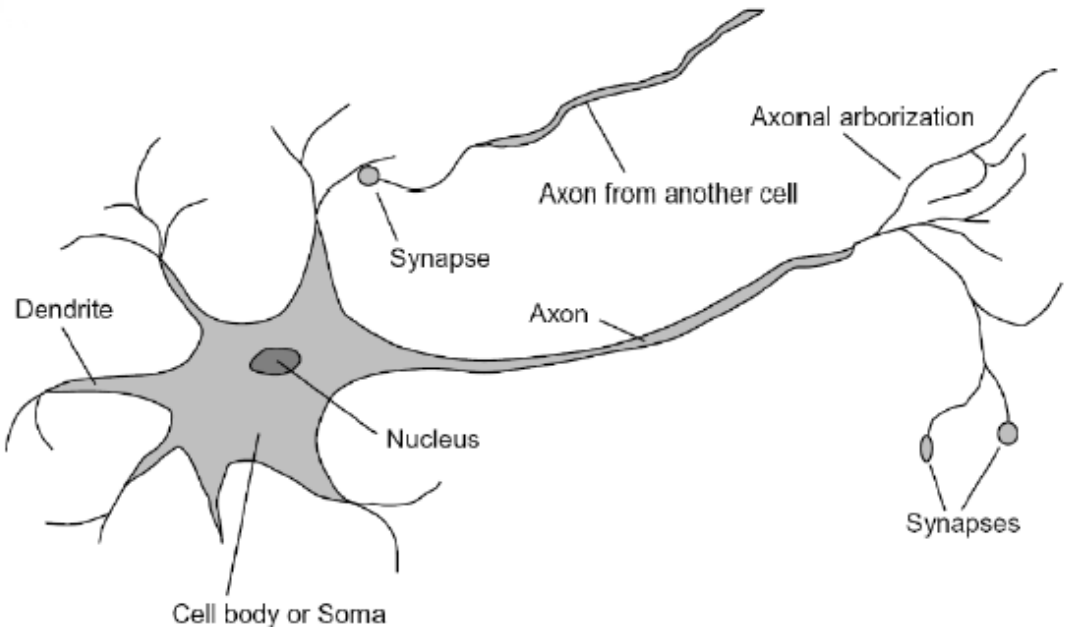

Figure 2.4: Sketch of a human neuron

The human brain is composed of 10^{11} neurons each of which is connected to other 10^4 neurons. The large dimension of the neural network is what makes the brain so powerful, in fact, each neuron is only able to perform simple tasks. More specifically, each neuron works in a binary way, whether it fires or not based on the processed electrical signals. This concept of explaining intellectual abilities

through a large neural network is known as *connectionism*: «Connectionism [$\cdots$] hopes to explain intellectual abilities using [$\cdots$] models of the brain composed of large numbers of units [$\cdots$] together with weights that measure the strength of connections between the units. These weights model the effects of the synapses that link one neuron to another.» [20] The large dimension of the neural network in the human brain could raise doubt about the computational speed, suggesting a low execution speed. This is far from true, indeed neurons work in parallel, making the processing very fast.

2.6.1 Artificial Neuron and Perceptron

Starting from the concepts of the human brain, two researchers, Warren Mc Culloch (neuroscientist) and Walter Pitt (logician) developed the so-called *Artificial Neuron* in their 1943 paper, "*A Logical Calculus of Ideas Immanent in Nervous Activity*" which is the first tentative to mimic the biological neuron in order to build intelligent machines. Figure 2.5 describes the Artificial Neuron structure. Basically, it can be considered as a two-part structure in which the first one, g in Figure 2.5, receives the input data and performs an aggregation while the second structure, f, makes the decision based on the previous computation. The model accepts only boolean input and also the output must be boolean. The inputs can either be *excitatory* or *inhibitory*. The latter means that the input itself is responsible for firing the Artificial Neuron without considering the other inputs. Whereas the former has not had this maximum effect on the neuron so the firing is due to the combination with other inputs. More formally, the aggregation is a simple summation of the inputs:

$$g(x_1, x_2, x_3, \cdots, x_n) = g(\mathbf{x}) = \sum_{i=1}^{n} x_i, \tag{2.21}$$

and then the final decision is based on the aggregated value:

$$y = f(g(\mathbf{x})) = \begin{cases} 1 \text{ if } g(\mathbf{x}) \geq \theta \\ 0 \text{ if } g(\mathbf{x}) < \theta \end{cases} \tag{2.22}$$

where θ is a threshold parameter that must be manually set depending on the specific task. The Artificial Neuron has some visible drawbacks that prevent its

application nowadays. For instance, the inputs are given the same importance and they must be boolean, this means that real numbers can not be fed into the neuron. Another relevant disadvantage is due to the fact that the threshold parameter should be manually set and the model is unable to learn it by itself. The last downside is the inability to learn non-linear functions, for instance, the boolean function XOR (exclusive or).

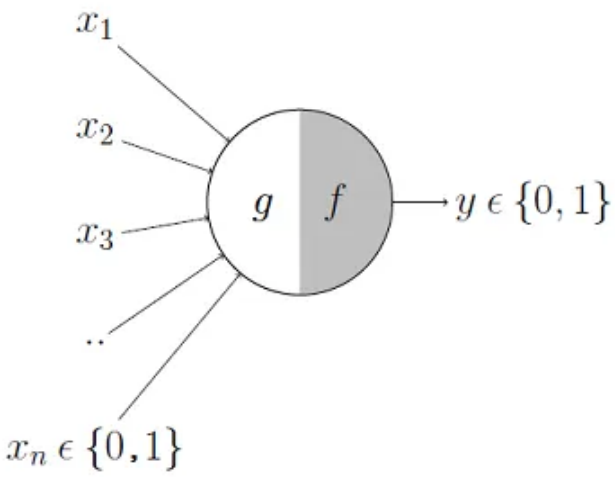

Figure 2.5: McCulloch and Pitt's Artificial Neuron. Source: Akshay L Chandra, Towards Data Science, *McCulloch-Pitts Neuron - Mankind's First Mathematical Model Of A Biological Neuron*

Donald Hebb in his 1949 book "*The Organization of Behavior. A Neuropsychological Theory*" introduced the concept of synaptic plasticity. He describes how neural activities influence the connection between neurons, in particular when two neurons fire together, the connection between them is strengthened. This idea was fundamental in the future developments of artificial intelligence, indeed Frank Rosenblatt, an American psychologist, in his 1958 book "*The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain*", took inspiration both from McCulloch and Pitt's Artificial Neuron and Hebb'rule introducing a new mathematical model: the *Perceptron*. Figure 2.6 shows the Perceptron architecture and it can be seen how it is similar to the MP Artificial Neuron even if the Perceptron is an improvement over it. Such as McCulloch and

Pitt's model, Rosenblatt's Perceptron receives some inputs, but in this case, they may not be boolean inputs, whereas they can take real values. Each input comes with its own weight associated with it, differently from the Artificial Neuron which assumed equal connection strength, as a consequence, the Perceptron performs a weighted aggregation of inputs. On top of the inputs, there is always another input equal to one, and its associated weight is called the *bias*. The final output is the result of a transformation function, a step function, applied to the weighted aggregation. More formally, each input $x_i \subseteq \mathcal{R}$, with $i = 0, \cdots, n$ is associated with a connection weight w_i with $i = 0, \cdots, n$. Notice that $x_0 = 1$ and w_0 is the bias. The weighted aggregation is the following:

$$\sum_{i=0}^{n} w_i x_i = w_0 x_0 + \cdots + w_n x_n. \tag{2.23}$$

The final output is given by the transformation function (step function, which is neither continuous nor differentiable) applied to the weighted aggregation:

$$y = f(g(\mathbf{x})) = \begin{cases} +1 \text{ if } \sum_{i=0}^{n} w_i x_i \geq 0 \\ -1 \text{ or } 0 \text{ if } \sum_{i=0}^{n} w_i x_i \leq 0 \end{cases} \tag{2.24}$$

The Perceptron, like the Artificial Neuron, is a linear classifier which means that it's only able to work when two classes are linearly separable. However, the most relevant improvement over MP Artificial Neuron is the ability to learn by itself. With the word *learning* we mean that the Perceptron adjusts weights so as to make better classifications. This is possible with an algorithm called *Delta Rule* which has the following form:

$$Update = LearningFactor \cdot (DesiredOutput - ActualOutput) \cdot Input \tag{2.25}$$

The magnitude of the weights update depends on the difference between the desired output and the actual output. When the actual output is equal to the desired output, there is no update. If the actual output is lower than the desired output, the update is positive if the input is positive or negative if the input is negative. If the actual output is greater than the desired output, the update is negative if the input is positive or positive if the input is negative. Indeed, the goal is to reduce the difference between the actual output and the desired output. The input

influences the magnitude of the update, in fact, the smaller the input, the smaller the magnitude of the update, conversely, a great input has a bigger effect on weights update. Even the learning rate has an effect on the magnitude of the update: a small learning rate takes many steps in order to converge, while a large learning rate makes the updates more dependent on recent instances. It can be proved that, if the inputs are linearly separable, the Delta Rule terminates the updating of the weights after a finite number of iterations.

The Perceptron is not able to learn the XOR function given its non-linear property. Minsky and Paper in their 1969 book "*Perceptrons: An Introduction to Computational Geometry*" critique the Perceptron underlying the fact that it is not able to learn non-linear functions. Furthermore, in 1973, the UK Science Research Council commissioned the Lighthill Report which was a criticism of the artificial intelligence field for not achieving big results as promised. As a consequence, these critiques led to decreasing funds for artificial intelligence research. This period is known as AI winter (70'-80') which denotes the crisis of the AI field due to the high hype and overinflated promises that were not respected.

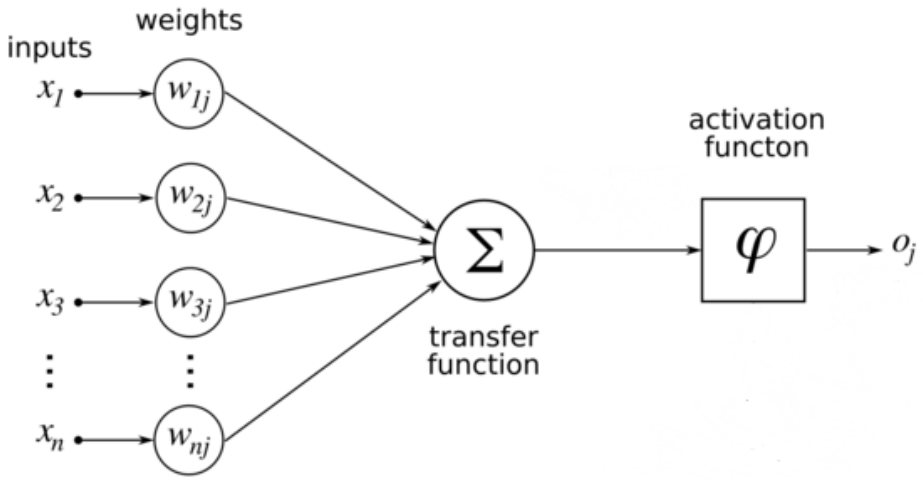

Figure 2.6: Rosenblatt's Perceptron. Source: Mario Emmanuel, Towards Data Science, *The 1958 Perceptron as a tumour classifier*

2.6.2 Multi Layer Perceptron

The Perceptron can be improved by either using a different transformation function or using a different architectural structure of the network. One of the most relevant activation functions is the *sigmoid* which is continuous and differentiable in all its domains. The sigmoid is also bounded between 0 and 1, but the differentiability property is the crucial one since the algorithm from which the network learns is based on partial derivatives. The architectural structure of the network is how neurons can be interconnected to form such a network. One of the most common architectures is the *multi-layer perceptron feedforward artificial neural network*, which is characterized by an input layer composed of neurons responsible for receiving inputs from the external environment. An output layer has the task of releasing the outputs, which are constituted by several neurons equal to the number of outputs. Between the input layer and the output layer, there is a certain number of hidden layers (both the number of hidden layers and the number of neurons for each layer must be set by the user). Generally, each hidden neuron is characterized by an activation function (e.g. sigmoid), while each output neuron is characterized by a linear transformation. All these layers are interconnected with each other and the flow of information is feedforward, from the input layer to the output layer. Each connection has an associated weight which explains the importance of the connection itself. Figure 2.7 depicts an example of multi-layer perceptron artificial neural network architecture.

More formally, an input layer is constituted by $n+1$ number of neurons, with $n \geq 1$, that is as many as the inputs plus 1 for the bias. One or more hidden layers, each composed by $h_l \in \mathbb{N}^+$ of neurons, with $l = 1, \cdots, H$, where H is the number of hidden layers. The number of neurons for each hidden layer is $h_l + 1$ because of the bias term. Supposing that the number of hidden layers $H = 1$, $w_{0;i,j}$ is the weight associated with the arc from the $i-th$ node of the $0-th$ layer to the $j-th$ node of the next layer and the output layer is constituted by 1 neuron, the final output of the MLP is the following:

$$y = a \cdot \sum_{j=0}^{h_1+1} w_{1;j,y} \cdot \frac{1}{1 + \exp\left\{-\sum_{i=0}^{n+1} w_{0;i,j} \cdot x_i\right\}}, \tag{2.26}$$

where a is the linear transformation and $\frac{1}{1+\exp\left\{-\sum_{i=0}^{n+1} w_{0;i,j} \cdot x_i\right\}}$ is the output of the hidden neurons which are the result of the weighted aggregation fed into the activation function, in this case, a sigmoid function.

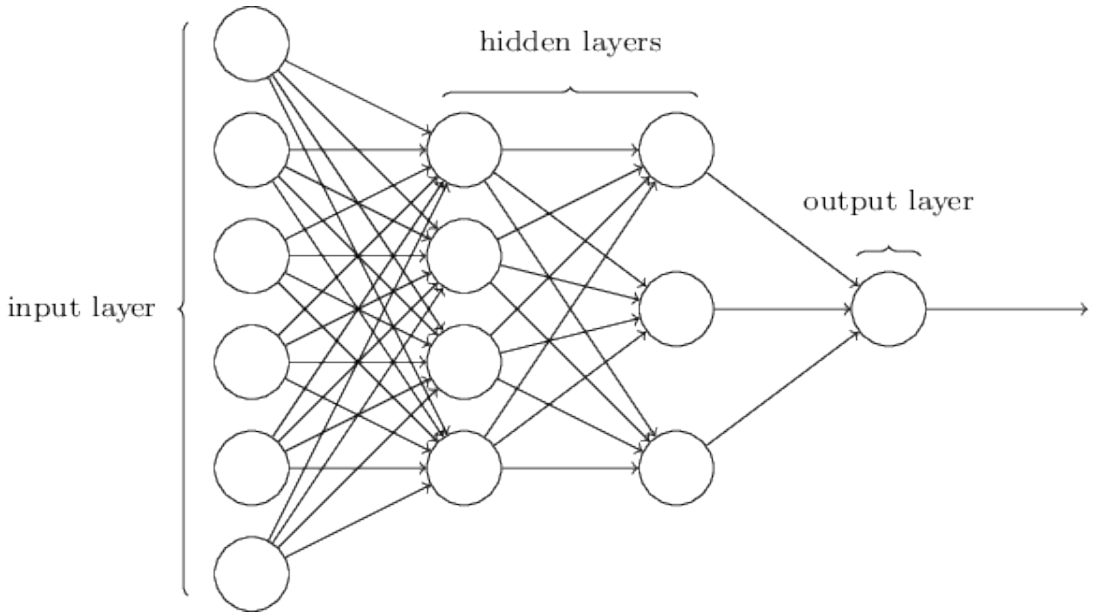

Figure 2.7: Multi Layer Perceptron. Source: Computer Science Wiki, *Multi-layer perceptron (MLP)*

The MLP feedforward ANN is considered a universal function approximator as proved by Cybenko in his 1989 paper "*Approximation by superpositions of a sigmoidal function*": «[The] networks with one internal layer and an arbitrary continuous sigmoidal function can approximate the continuous function with arbitrary precision providing that no constraints are placed on the number of nodes or the size of the weights.». In the same year the Defense Advanced Research Projects Agency (DARPA) published the "*DARPA neural network study final report*" in which highlights the importance of the number of hidden layers for learning non-linear functions. Cybenko focuses on the importance of the sigmoid activation function, while DARPA highlights the importance of the number of hidden layers.

The neural network changes its weights according to the so-called *backpropagation algorithm*, popularized by Rumelhart D.E., Hinton G.E., and Williams R.J. in their 1989 paper "*Learning internal representations by error propagation*" who took inspiration from the Rosenblatt's Delta Rule. The algorithm, explained in Figure

2.8, is an iterative numerical optimization based on the gradient descent method (2.19).

Randomly initialize the vector of weights: **w**
for $k = 1$ to K **do**
  Calculate the error:
  $E = \frac{1}{2}\sum_{i=1}^{m}(y_i - o_i)^2$
  for $i = 1$ to W **do**
    Solve the minimization problem for weight w_i:
    $w_i^{(l+1)} = w_i^{(l)} + \Delta w_i^{(l+1)} - \alpha\frac{\partial E}{\partial w_i}$
  end for
end for

Figure 2.8: Backpropagation algorithm

In the backpropagation algorithm, the error term is computed by comparing the predicted output with the actual label, but it is only applied to a portion of the entire data set. To do this, the data set D is split into three parts, the training set TRAIN, the validation set VALID, and the testing set TEST, such that $TRAIN \cup VALID \cup TEST = D$, $TRAIN \cap VALID = \emptyset$, $TRAIN \cap TEST = \emptyset$, $VALID \cap TEST = \emptyset$, $TRAIN \cap VALID \cap TEST = \emptyset$. The neural network is only trained with the backpropagation algorithm on the training set. The number of iterations of the backpropagation algorithm influences the neural network's performance. Indeed, if the number of iterations is too large, the neural network memorizes the input-output pairs without learning the relationship between them, a problem known as *overfitting*. On the other hand, if the number of iterations is too small, the neural network does not memorize the input-output pairs, but it does not learn their relationship as well, a problem known as *underfitting*. The goal of a neural network is to learn the relationship between input-output pairs so that it has the capability of predicting the unseen data (the data not used for training the network): «[T]he goal of network training is not to learn an exact representation of the training data itself, but rather to build a [$\cdots$] model of the process which generates the data. This is important if the network is to exhibit good generalization, that is, to make good predictions for new inputs.» [0] One way to deal with this problem and find the correct number of iterations is using

the so-called *early stopping* approach. It works by iterating the backpropagation algorithm in the training set and each iteration computes the error in the validation set. The correct number of iterations is the one that minimizes the error in the validation set (as explained in Figure 2.9).

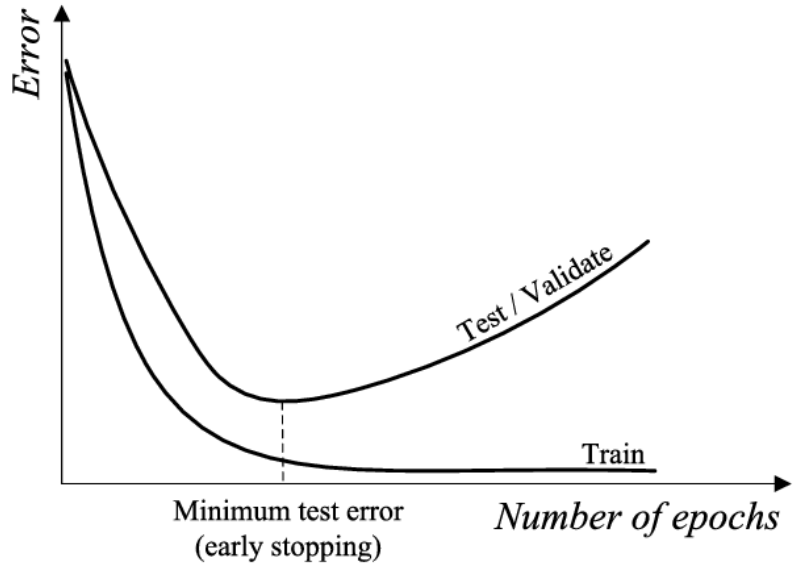

Figure 2.9: Early stopping approach. Source: Mohammad Yawar, Coding Ninjas, *Early Stopping In Deep Learning*

Finally, after setting the weights of the neural network, the performance of the network is evaluated using the testing dataset. This evaluation provides the most reliable result because the network has not encountered the testing data during training. The performance on the training dataset is not informative since the network has already been exposed to that data during training. Similarly, the performance of the validation dataset is not directly comparable to the testing dataset, as the validation dataset is primarily used to optimize the selection of the best set of weights. Therefore, the performance on the validation dataset may be biased and not indicative of the network's true performance on unseen data.

Let w_j, with $j = 1, \cdots, W$, be a weight of the neural network and suppose $\widehat{w}_j(x_0, \cdots, x_n)$ be the estimator of w_j. The error to manage is $\mathbb{E}[\widehat{w}_j(x_0, \cdots, x_n) - w_j]$

and it can be rewritten in the following way:

$$\begin{gathered}\mathbb{E}[\widehat{w}_j(x_0,\cdots,x_n)-w_j]= \\ \mathbb{E}^2\left[\widehat{w}_j\left(x_0,\ldots,x_n\right)\right]-2\mathbb{E}\left[\widehat{w}_j\left(x_0,\ldots,x_n\right)\right]\cdot w_j+w_j^2+ \\ +\mathbb{E}\left[\widehat{w}_j^2\left(x_0,\ldots,x_n\right)\right]-\mathbb{E}^2\left[\widehat{w}_j\left(x_0,\ldots,x_n\right)\right]= \\ \boxed{(\mathbb{E}[\widehat{w}_j(x_0,\cdots,x_n)]-w_j)^2}+\boxed{\mathbb{E}\left[\widehat{w}_j^2\left(x_0,\ldots,x_n\right)\right]-\mathbb{E}^2\left[\widehat{w}_j\left(x_0,\ldots,x_n\right)\right]}= \\ Bias^2([\widehat{w}_j(x_0,\cdots,x_n)],w_j)+Var(\widehat{w}_j(x_0,\cdots,x_n)).\end{gathered} \tag{2.27}$$

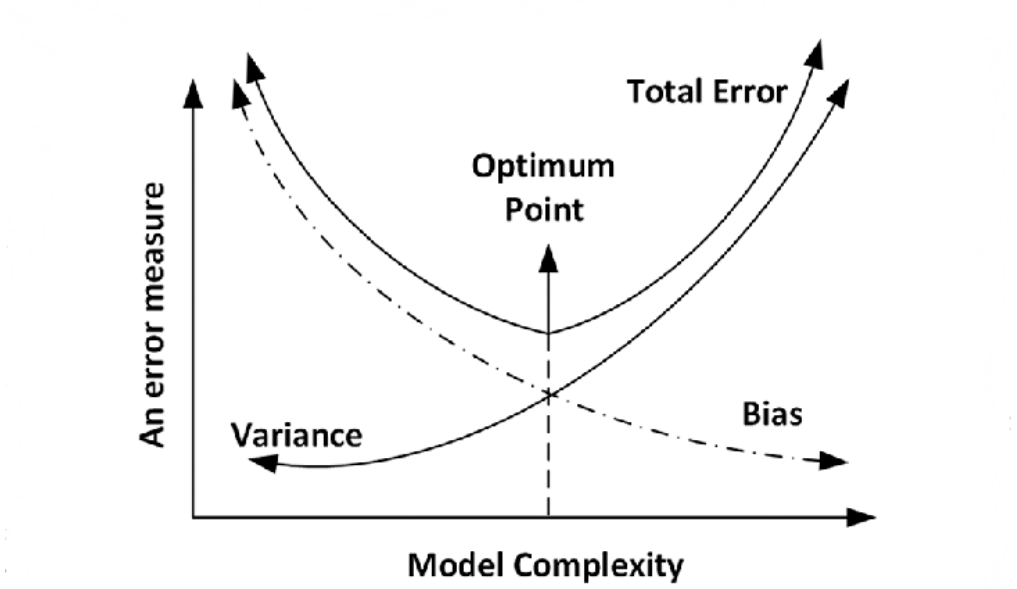

Figure 2.10: Bias-variance trade-off. Source: Gungor Osman Erman and Al-Qadi Imad, *Developing Machine-Learning Models to Predict Airfield Pavement Responses*

The *bias error* of an estimator is attributed to the inherent limitations of its functional form, while the *variance error* quantifies the inaccuracies in estimating the weight value. The concept of *bias-variance tradeoff* can be illustrated in Figure 2.10, which depicts a two-axis chart with model complexity on the x-axis and total error on the y-axis. As model complexity increases, the bias decreases because the model's architecture becomes more capable of capturing the relationship between input-output pairs. However, this leads to an increase in variance since the complex model tends to overly rely on the specific patterns present in the training data, resulting in overfitting. Conversely, reducing model complexity raises the bias because the model becomes less capable of capturing the input-output relationship,

but it reduces the variance since a simpler model generalizes better by avoiding overfitting.

2.7 Deep-Q-Network Algorithm

Mnih et al. in their 2015 paper "*Human-level control through deep reinforcement learning*" introduced a new reinforcement learning algorithm capable of performing at a human level in the Atari 2600 games environment. More specifically, the agent reaches a level similar to a professional human games tester across 49 games receiving as inputs only the pixels and the game score. This is made possible by using a convolutional neural network as a function approximator for the optimal action-value function:

$$Q^*(s,a) = \max_{\pi} \mathbb{E}\left[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots \mid s_t = s, a_t = a, \pi\right]. \tag{2.28}$$

Due to its non-linearity, using a neural network as a function approximator improves flexibility and generalizes the relationship between state-action pairs and their associated values. On the other hand, it may be unstable and even diverge from the actual function. This is mainly due to three reasons: the sequence of observation is highly correlated; small changes to the action value function may have an impact on the policy, hence changing the data distribution; the high correlation between the action values and the target values. All these instabilities are solved or reduced by implementing two ideas. First, a biological-inspired mechanism to randomize the sequence of observations and prevent a change in the data distribution is known as *experience replay*. The second idea is to use an additional function approximation, the so-called *target network*, to approximate the target values with the same weights as the other Q-function approximator, the *online network*, but updated periodically, resulting in a decrease in correlation between action values and the target values.

More formally, $Q(s,a;\theta_i)$ is the parametrized form of the action value function using a CNN, where θ_i are the weights of the network at iteration i. At each time step t, the agent experiences $e_t = (s, a, s', r, T)$, which is a tuple containing the initial state s, the action a taken by the agent, the new state s', the reward r obtained, and a boolean value T indicating whether the new state is a terminal state or not.

Each experience is stored in the data set $D_t = \{e_1, \cdots, e_t\}$ and the CNN is trained on uniformly drawn random samples from the data set: $(s, a, r, s', T) \sim \mathrm{U}(D)$. The following loss function is used to update the network:

$$L_i(\theta_i) = \mathbb{E}_{(s,a,r,s')\sim \mathrm{U}(D)} \left[\left(r + \gamma \max_{a'} Q\left(s', a'; \theta_i^-\right) - Q\left(s, a; \theta_i\right) \right)^2 \right] \tag{2.29}$$

where γ is the discount factor, θ_i are the parameters of the action value function at iteration i, and θ_i^- are the parameters of the network used to update the target values at iteration i. The target network parameters θ_i^- are updated every C number of steps and set equal to the online network. Figure 2.11 explains the whole algorithm in detail. The DQN implementation differs from the algorithm in Figure 2.11, in which the Q-network updates its parameters θ at each environment step. Indeed, in the original algorithm, the Q-network takes a gradient descent step every 4 environment steps. In this way, the training time decreases significantly, since learning steps are computationally expensive compared to forward passes. Another advantage is that, by waiting for a longer time between learning steps, more transitions are stored in the experience replay memory, resulting in a distribution closer to the current policy and preventing the network from overfitting.

Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
for episode 1, M **do** Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ε select a random action a_t
    otherwise select $a_t = \arg\max_a Q(\phi(s_t), a; \theta)$
    Execute action a_t in the emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store experience $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
    Sample random minibatch of experiences $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to the weights θ
    Every C steps reset $\hat{Q} = Q$
  end for
end for

Figure 2.11: Deep-Q-Network algorithm. Source: Van Hasselt et al. *Deep reinforcement learning with double q-learning*

It is common thought that the more training time, the better the performance, but this is not true. Unlike neural networks and Q-learning in the tabular setting

(which improve their performance by increasing the data, resulting in average learning curves that improve stably), the DQN agent can drop its performance after a period of learning. This is known as *catastrophic forgetting* and the introduction of the experience replay and the target network aims at reducing its effect. However, these two elements are not enough to prevent catastrophic forgetting from happening, because one of its causes is inherently contained in the value-based RL methods. Indeed, value-based methods learn a proxy of the policy rather than approximating the policy itself. As a consequence, learning updates may improve the accuracy of the approximator while decreasing the performance of the policy. For instance, suppose that the true Q-values for state s and actions $a1, a2$ is $Q^*(s, a_1) = 2$ and $Q^*(s, a_2) = 3$, so the optimal action would be to select a_2. The function approximator, using parameters θ, estimates $Q^*(s, a_1; \theta) = 0$ and $Q^*(s, a_1; \theta) = 1$, resulting again in choosing a_2. After some updates to the parameters, the function approximator estimates $Q^*(s, a_1; \theta) = 2$ and $Q^*(s, a_1; \theta) = 1$, which are closer to the true Q-values but now the action chosen from the policy is a_1, the non-optimal one.

Mnih et al. tested the algorithm in the Atari 2600 games environment, across 49 games, using the same architecture, hyperparameters, and learning procedure. The results were very promising since the game scores were similar to a human professional game tester. Furthermore, they demonstrated the relevance of each component of the algorithm: the replay memory, the target network, and the deep convolutional neural network. In fact, by disabling them, the agent performance decreased significantly.

2.7.1 Double DQN

Q-learning is one of the most famous reinforcement learning algorithms. Still, it has the problem of overestimating the action values due to the maximum operator in the function used to make updates. These overestimations may be very common in practice and may lead to worse agent performance. The latter aspect may not verify if the overestimations are uniform over the actions (in this way, the choice of the best action is not affected by overestimations), but if they are not uniformly

distributed, the agent performance would get worse. Double-Q-learning (Hado Van Hasselt, 2010) is an improvement over Q-learning since it tries to prevent the overestimations of the action values. Van Hasselt et. al. in their 2015 paper "*Deep Reinforcement Learning with Double Q-Learning*" extended the Double Q-learning, first developed in the tabular case, to the function approximation setting, allowing the application of Double-Q-learning to the DQN algorithm.

Q-learning uses the max operator both to evaluate and select the action, as a result, it's very likely that the action estimate is overestimated. Indeed, the target values in Q-learning are given by the following formula:

$$Y_t^{DQN} = r + \gamma \max_{a'} Q\left(s', a'; \theta_i^-\right) \tag{2.30}$$

The main idea behind Double-Q-learning is to decouple the selection of the best action, the one having the highest action value, from its evaluation. To implement this idea, two different function approximators are used, one to select the action and the other to evaluate it. In the DQN algorithm, since there are already two function approximators (the online network and the target network), seems natural to use the online network to evaluate the greedy policy by choosing the best action, while the target network to estimate its value. As a consequence, 2.30 becomes:

$$Y_t^{DoubleDQN} = r + \gamma Q\left(s', \underset{a}{\text{argmax}} Q\left(s', a'; \boldsymbol{\theta}_i\right); \boldsymbol{\theta}_t^-\right) \tag{2.31}$$

The Double DQN algorithm was tested in the Atari 2600 games setting and was able to achieve higher scores on several games compared to the standard DQN, furthermore, the accuracy of action value estimates increased, proving that DQN overestimations lead to poorer policies.

2.7.2 Prioritized Experience Replay

Experience replay allows the agent to store the environment transitions to learn from them, through past experiences. It also helps reduce correlations between transitions and, as a consequence, learning more stably, without overfitting. The agent uniformly samples the transitions, regardless of their significance, currently present in the experience replay and learns from them by adjusting the weights of the

neural network which is used as a function approximator. But what if there are some transitions more useful than others? Schaul et al. in their 2016 paper "*Prioritized Experience Replay*" studied different ways of sampling from the experience replay, prioritizing some transitions, to speed up learning. More specifically, they prioritize transitions with higher expected learning which is measured by the magnitude of the temporal-difference (TD) error. The idea is that the greater the TD error, the more surprising the transition is.

The algorithm stores the TD errors in the experience replay along with their associated transitions. Then the transitions with the largest TD error are sampled from the replay memory and used to adjust the weights of the function approximator through a Q-learning update. When the transitions are stored for the first time in the replay memory, they don't have a TD error (since the error is only computed when the transitions are sampled), hence they are given the maximum priority in order to guarantee that all transitions are seen at least once.

This algorithm has some issues. First, the TD error updates are only applied to the sampled transitions, as a consequence, the transitions which have a low error when first visited, may be not updated anymore, since the highest priority is given to those with higher error. Another issue, closely connected to the previous one, is that reducing the TD error may take a long time, meaning that always the same transitions are sampled, leading to overfitting. Lastly, this algorithm suffers noisy rewards.

To overcome these issues, the researchers implement a stochastic sampling method balancing between greedy priority sampling and uniform random sampling. Following this idea, the probability of sampling transition i is:

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha}, \tag{2.32}$$

where $p_i > 0$ is the priority of transition i and α measures the strenght of prioritization ($\alpha = 0$ is the uniform random sampling case).

The researchers introduce two variants, both with sampling probability monotonic to the absolute magnitude of the TD error $|\delta|$. The first one is the direct prioritization in which $p_i = |\delta_i| + \epsilon$, where ϵ is a small positive constant that allows transitions with $|\delta| = 0$ to be sampled. The second variant is the indirect rank-based

prioritization in which $p_i = \frac{1}{rank(i)}$, where $rank(i)$ is the rank of the transition i once the replay memory is sorted according to $|\delta_i|$. This last version may be more robust compared to the first one since it's insensitive to outliers.

Input: minibatch k, step-size η, replay period K and size N, exponents α and β, budget T.
Initialize replay memory $\mathcal{H} = \emptyset$, $\Delta = 0$, $p_1 = 1$
Observe S_0 and choose $A_0 \sim \pi_\theta(S_0)$
for $t = 1$ **to** T **do**
  Observe S_t, R_t, γ_t
  Store transition $(S_{t-1}, A_{t-1}, R_t, \gamma_t, S_t)$ in $\mathcal{H}$ with maximal priority $p_t = \max_{i<t} p_i$
  if $t \equiv 0 \mod K$ **then**
    for $j = 1$ **to** k **do**
      Sample transition $j \sim P(j) = p_j^\alpha / \sum_i p_i^\alpha$
      Compute importance-sampling weight $w_j = (N \cdot P(j))^{-\beta} / \max_i w_i$
      Compute TD-error $\delta_j = R_j + \gamma_j Q_{\text{target}}(S_j, \arg\max_a Q(S_j, a)) - Q(S_{j-1}, A_{j-1})$
      Update transition priority $p_j \leftarrow |\delta_j|$
      Accumulate weight-change $\Delta \leftarrow \Delta + w_j \cdot \delta_j \cdot \nabla_\theta Q(S_{j-1}, A_{j-1})$
    end for
    Update weights $\theta \leftarrow \theta + \eta \cdot \Delta$, reset $\Delta = 0$
    From time to time copy weights into target network $\theta_{\text{target}} \leftarrow \theta$
  end if
  Choose action $A_t \sim \pi_\theta(S_t)$
end for

Figure 2.12: Prioritized experience replay algorithm. Source: Schaul et al. *Prioritized experience replay*

The estimation of expected values using uniform random sampling updates relies on these updates being generated from the same distribution as their expected values. However, prioritized replay introduces bias because it alters this distribution leading to a change in the converged solution of the estimates. To address this bias, we can employ *weighted* importance sampling:

$$w_i = (\frac{1}{N} \cdot \frac{1}{P(i)})^\beta, \tag{2.33}$$

which fully compensate for the non-uniform probabilities if $\beta = 1$. By incorporating these weights into the Q-learning updates, using weighted importance sampling instead of ordinary importance sampling, the biased distribution can be corrected. For stability purposes, the weights are normalized.

In typical reinforcement learning scenarios, unbiased updates become more critical towards the end of training when the process is highly non-stationary due to changing policies, state distributions, and bootstrap targets. For this reason, the

coefficient β is corrected over time by defining a schedule that gradually increases the exponent parameter. Specifically, β is linearly increased from its initial value to 1. It is important to note that the choice of this hyperparameter interacts with the prioritization exponent α: increasing both parameters prioritizes more aggressive sampling while simultaneously providing stronger correction for it. The whole stochastic prioritized sampling algorithm is described in Figure 2.12.

The Double DQN agent, trained using the prioritized experience replay memory, overperforms the standard DQN agent on 41 out of 49 Atari games.

Chapter 3

Reinforcement Learning Applied to Forex Trading

So far we have explored the Forex market, its functioning, and the theoretical foundation of reinforcement learning. This section serves as the bridge between these two concepts, aiming to evaluate a reinforcement learning agent's performance in Forex trading. Moreover, this section aims to outline the process of creating the model, highlight the challenges encountered during its development, present the outcomes of the experiments conducted, and conclude with some ultimate reflections and potential areas for enhancement.

3.1 Literature Review

Before starting the explanation of the model, a brief overview of the literature on applying reinforcement learning to the Forex market is provided. While research on this technique is more prevalent in stock trading, I will present the relevant studies I have reviewed and drawn inspiration from.

Carapuco et al. [1] conducted experiments with a Q-learning algorithm, specifically using a Deep Q-Network (DQN) agent, focusing on the EUR/USD currency pair. Their approach involved a unique method of describing the market state using tick data, and they introduced a modified version of the Sortino ratio as a reward signal. Notably, their algorithm is designed as a supervised learning

framework, where the agent undergoes training on a dataset before being tested on an out-of-sample dataset. In the EUR/USD market spanning from 2010 to 2017, Carapuco et al. observed consistent results across 10 tests with varying initial conditions. The algorithm achieved an annual average profit of 16.3 ± 2.8%.

Dempster and Lemans [5] introduced an innovative approach termed *adaptive reinforcement learning*, comprising three key components: a decision-making model, a risk management layer, and an optimization layer. This architecture is structured in layers, with the model layer employing a direct reinforcement method that optimizes the policy directly, more specifically *Recurrent Reinforcement Learning* (RRL), originally introduced by Moody and Saffel in 2001. The second layer enables the customization of risk management by tailoring it to individual preferences, capable of autonomously closing positions even when the model in the first layer suggests otherwise. The third layer, the adaptive component, earns its name by dynamically optimizing the parameters of the risk management layer in real-time. In their experimentation, Dempster-Lemans achieved a notable 26% annual return on the EUR/USD currency pair. This achievement was realized over a two-year span (2000-2001) and using a 1-minute timeframe.

Huang and Chien-Yi [8] adopted an online Q-learning strategy, specifically calling it *Deep-Recurrent-Q-Network* (DRQN). This approach involves the integration of an LSTM network as a function approximator within the conventional DQN algorithm. A notable advancement they introduced includes using a compact replay memory with an extended sampled sequence for the training process. They also came up with an innovative action augmentation technique, which reduces the necessity for random exploration within the context of financial trading. They achieved positive returns across 12 different currency pairs, encompassing both major and cross pairs, even after accounting for transaction costs. Operating within a 15-minute timeframe, their study spanned from 2013 to the conclusion of 2017.

Shavandi and Khedmati [13] proposed an innovative approach to trading, distinct from conventional methods. They drew inspiration from the *fractal market hypothesis*, which states that market behavior is influenced by the collective psychology of investors with varying trading horizons and interpretations of information. To

implement this, they developed a framework using multiagent deep reinforcement learning. This system capitalizes on the combined expertise of multiple agents, each specialized in trading within specific timeframes. Notably, the shortest timeframe agent triggers the actual trading decisions, with input from agents specialized in longer timeframes to help in decision-making. The proposed approach was empirically tested across twenty independent experiments spanning from mid-2012 to mid-2021, using a 5-minute timeframe. The results demonstrated an average annual return of 5.1%.

Tsantekidis et al. [14] introduced an advanced deep reinforcement learning model and an innovative reward-shaping technique centered around price trailing. This approach led to substantial performance enhancements, notably boosting the Sharpe ratio and curtailing maximum drawdown. The researchers evaluated this strategy using an extensive data set encompassing 28 distinct currency pairs from 2017 to mid-2018. Preceding this, a training phase employed data from 2009 to 2016. The outcome of their experiment showcased an annual return of 4.1%, achieved through the use of a DQN (Deep Q-Network) agent.

3.2 Workflow Organization

This part is dedicated to explaining how the project has been implemented from a software point of view. The whole implementation is written in Python code, the most common programming language in the machine learning field, both in industry and in academic research. More specifically, I have preferred to use an object-oriented programming (OOP) workflow, this choice has been driven by the great amount of code needed and the high number of entities interacting with each other. Indeed, such a big project should be written in a clear and understandable manner so as to avoid incurring errors. The whole program has been structured in such a way that 8 classes interact with each other so as to simulate an RL agent trading in the Forex market, as provided in Figure 3.1.

The first one is the *Preprocessor* which is responsible for taking as input the raw EUR/USD data (30-minute timeframe open, high, low, close candles data)

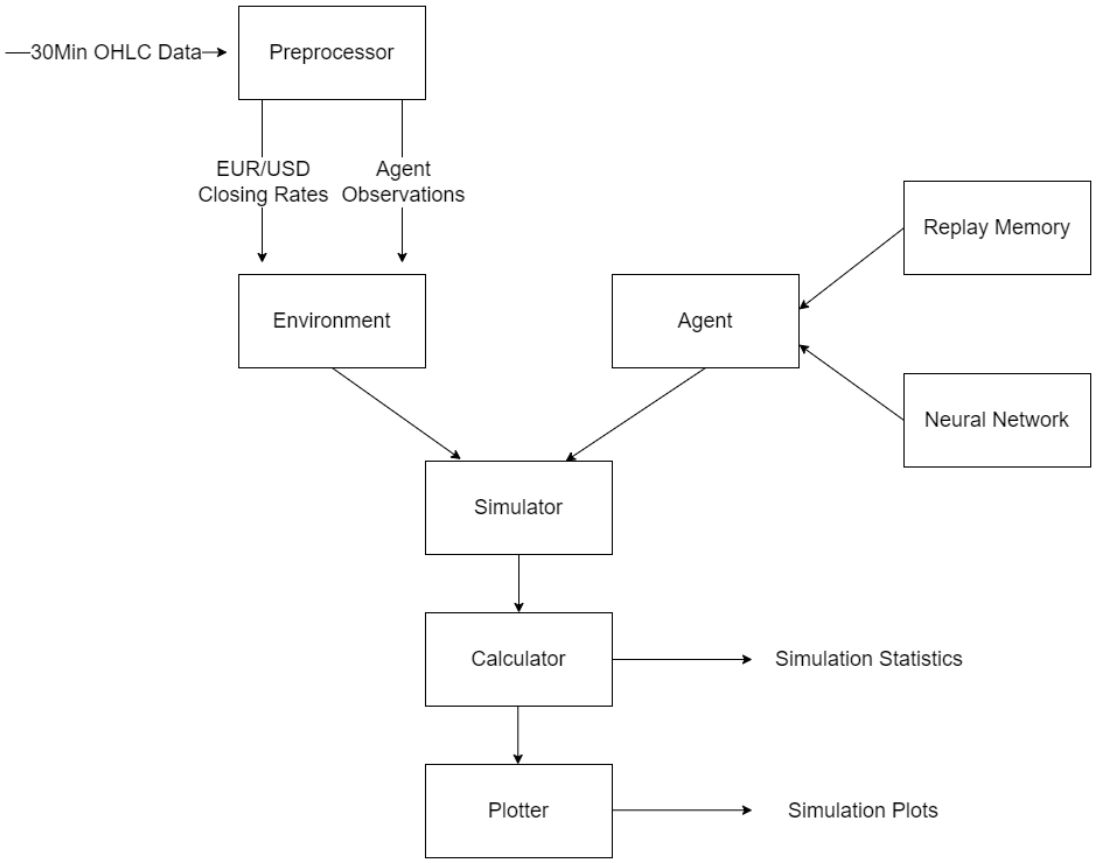

Figure 3.1: Workflow organization flowchart.

and preprocessing it, which means transforming the raw data through feature engineering techniques with the goal of obtaining the state description observable by the agent. Alongside the agent observations, the closing EUR/USD exchange rates are kept in order to compute profit and loss during the simulation.

The second class is the *Agent* which receives as inputs the other two classes: the *Replay Memory* and the *Neural Network*. The former, as introduced in the previous chapter, is the agent's memory in which a certain number of environment transitions are stored. The latter is the Q-function approximator which evaluates the value of each action, for each encountered state.

The fifth class is the *Environment* in which the agent has to operate. Inside this class, the two main functions are the step and the reward functions (note that I used the term function, indeed they are not classes but functions inside the Environment class). The step function takes as input the agent action and returns the next state along with the reward obtained by the agent. As a consequence,

inside the step function, the reward function is called, which is the most important function in the whole project, indeed the agent learns from it.

The sixth class is the *Simulator*, which is responsible for running the simulations, taking as inputs both the environment and the agent. The simulation starts with the environment in its initial state and runs all over the states, with the agent choosing the actions, until the last observation, the moment in which the simulation stops. The Simulator outputs are a series of lists such as the balance over time, the actions performed by the agent and the reward obtained from each step.

All these lists are taken as inputs by the last two classes: the *Calculator* and the *Plotter*. The former has the task of computing all the statistics related to the simulation, for instance, the annual return on investment, the win ratio, the maximum drawdown, etc. The latter is responsible for the visualization part, which includes the balance over time chart, the agent's actions, from which it can be observed how the agent behaved in the simulation, and the reward at each step.

This whole project structure helps in the understanding of each component and how they relate to each other, furthermore it is essential in improving readability and preventing programming mistakes.

3.3 Model Structure

The following section aims to explain in detail all the model components necessary to develop a reinforcement learning agent specifically designed for Forex trading. As introduced previously, the agent will learn to trade using 30-minute timeframe OHLC data, as a consequence, the environment will inform the agent through the state signal every 30 minutes. The same timeframe dictates the time at which the agent is allowed to perform an action.

3.3.1 Environment Components

Let's start by introducing the state signal, which describes the current state of the environment. As explained in the previous chapter, the state signal should be able to represent also the past states, as declared in the Markov hypothesis.

By using the raw OHLC data as a state descriptor is not advisable, since they do not provide valuable information for the agent and could favor an unwanted price correlation problem [14]. More specifically, whenever the agent encounters the lowest price in the data set it will always buy the asset, but this behavior is not correct since the price could further decrease. Instead, starting from the raw OHLC data, a preprocessing phase is necessary to create new features. I have decided to implement the same features as [14]:

$$1) \quad x_{t,1} = \frac{p_c(t) - p_c(t-1)}{p_c(t-1)}, \qquad 4) \quad x_{t,4} = \frac{p_h(t) - p_c(t)}{p_c(t)},$$

$$2) \quad x_{t,2} = \frac{p_h(t) - p_h(t-1)}{p_h(t-1)}, \qquad 5) \quad x_{t,5} = \frac{p_c(t) - p_l(t)}{p_c(t)}.$$

$$3) \quad x_{t,3} = \frac{p_l(t) - p_l(t-1)}{p_l(t-1)},$$

The first three features are the percentage change of the closing price, the highest price and the lowest price in the 30-minute timeframe respectively. The last two features are a measure of volatility. Due to the non-stationarity of the financial data, these features are useful because of their normalizing nature, helping the agent in its learning process. The result of this feature engineering work is that the agent at each time step t will receive from the environment the following vector:

$$s_t^{(n)} = [X_t, X_{t-1}, X_{t-2}, \ldots, X_{t-n+1}],$$

where X_t is the feature vector at time t, X_{t-1} is the feature vector at time $t-1$, and so on:

$$\begin{aligned} X_t &= [x_{t,1}, x_{t,2}, x_{t,3}, x_{t,4}, x_{t,5}] \\ X_{t-1} &= [x_{t-1,1}, x_{t-1,2}, x_{t-1,3}, x_{t-1,4}, x_{t-1,5}] \\ X_{t-2} &= [x_{t-2,1}, x_{t-2,2}, x_{t-2,3}, x_{t-2,4}, x_{t-2,5}] \\ &\vdots \\ X_{t-n+1} &= [x_{t-n+1,1}, x_{t-n+1,2}, x_{t-n+1,3}, x_{t-n+1,4}, x_{t-n+1,5}]. \end{aligned}$$

The parameter n controls the number of the last feature vectors to include in the state signal s_t. In the experiments, I have tried different values so as to study the impact of the n parameter on the agent's performance. Notably, I have chosen 1, 4, 8 and 16 as values for the n parameter.

When the agent processes the state signal, an action is performed. In the academic literature, the typical action space used is discrete: $\mathcal{A}(\mathcal{S}_t) = \{-1, 0, 1\}$, where -1 represents a short position, 0 indicates staying out of the market, and 1 means a long position. I have decided to stick with this action space since it is simple to implement and proven to work with different reinforcement learning models. The only limit with this kind of action space is that the agent will always trade using the entire capital at its disposal, which is not realistic. Regarding the agent's actions is important to specify that I've chosen to force the agent to stay out of the market as its last action. This decision was made to facilitate the computation of all relevant statistics without the agent actively involved in the market.

The agent's evaluation of each action is given by its associated Q-value and, generally, the agent selects the action with the highest value. However, the Q-value depends on the reward function defined by the user. Consequently, the reward is, by far, the most important component of the model and unfortunately is the most difficult to define as well. In the academic literature, most of the implemented reward functions are risk-adjusted measures like Sharpe, Sortino and Calmar ratios, which theoretically allow the agent to find a profitable policy and contemporary limiting the risk. I have decided not to implement one single reward function, but to experiment with employing three different rewards so as to find out which reward best suits the Forex trading environment. The first one is taken from [8] and it is defined as the balance log returns:

$$r_{t+1} = \log(\frac{v_{t+1}}{v_t}) \tag{3.1}$$

where v_t is the balance at time t which comprises both the account safe capital and the unrealized profit. The starting balance $v_0 = 100.000\$$. The balance value satisfies the following recursive relation:

$$v_{t+1} = v_t \cdot [1 + a_t \cdot (c_{t+1} - c_t) - |a_t - a_{t-1}| \cdot \delta] \tag{3.2}$$

where a_t is the action selected by the agent at time t, c_{t+1} and c_t are respectively the price at time $t+1$ and time t. The commissions δ correspond to the bid-ask spread, however, since I've downloaded data in bid price format, I've exclusively applied them when the agent opts for a buying action at time t. It's worth noting that the bid price corresponds to short positions, and because the data is in bid price format, the price matches precisely when the agent goes short. Conversely, when the agent chooses a long position, the bid-format data doesn't accurately represent the actual entry or exit price. In this scenario, the actual price is the asking price, which results from adding the spread to the bid price. Consequently, I've made the decision to apply commissions uniquely when a buy action is taken (a buy action is not only described with $a_t = 1$, but even $a_t = 0$ means buy if $a_{t-1} = -1$). The spread parameter, denoted as δ, is subject to variation depending on the specific broker. Fortunately, Dukascopy Bank, the broker from which I source my data, offers statistics regarding the average spread for each currency pair [26]. Consequently, I have chosen to set $\delta = 0.3$ bp[1], relying on the information provided by Dukascopy Bank.

The other two reward functions start from the previous one and add new components to it, in this sense they should be an improvement over the portfolio log returns reward function. However, designing a more sophisticated reward function does not mean it works better than the simple one, indeed the goal is to find out which one of the three rewards best suits the Forex trading environment. One of these two new rewards is the well-known Sharpe ratio, the most studied reward function in the academic literature, defined as:

$$SR_{t+1} = \frac{\mathbb{E}_L\left(r_{t+1}\right)}{\sqrt{\mathbb{V}\mathrm{ar}_L\left(r_{t+1}\right)}} \tag{3.3}$$

where r_{t+1} is the same as 3.1, $\mathbb{E}$ and $\mathbb{V}$ar are the mean and the variance of the last L returns respectively. This ratio tells the amount of expected return for a unit of risk. I have set $L = 4$, so the reward is computed taking into account the last four returns which, in terms of time, cover a two-hour window.

[1]bp (basis point) in Forex corresponds to the pip. For the majority of currency pairs, a pip is equivalent to 0.0001, except in the case of the Japanese Yen (JPY), where it amounts to 0.01.

The last reward function is an innovative one since I have not found it in the literature. It is defined with the following expression:

$$r_{t+1} = (1 - \beta) \cdot \log(\frac{v_{t+1}}{v_t}) + \beta \cdot \log(\frac{v_c}{v_o}), \tag{3.4}$$

where v_o represents the account balance at the moment a trade is initiated, v_c denotes the account balance at the point when the trade is concluded and β is the weighting factor between the two logarithmic returns. Essentially it is a weighted summation between 3.1 and the logarithmic return of the last closed trade. The parameter β controls the weight of the last closed trade logarithmic return. The idea behind this reward function is that, differently from the previous two, it tells the agent that is important to win the trades. It seems an obvious aspect but actually, the other reward functions do not take into consideration it. Indeed, those rewards are computed based only on one-step logarithmic returns, without taking into account the actual logarithmic return of a trade, once it has been closed. From a metaphorical perspective, it's analogous to a sports manager who prioritizes team training over the actual matches. I have decided to set $\beta = 0.7$ so that the agent gives more importance to winning the trades rather than focusing on one-step logarithmic returns with the risk of losing the trades. Table 3.1 summarizes all the aforementioned environment parameters.

Parameter	**Value**
v_0	100.000
n	1, 4, 8, 16
δ	0.3 bp
L	4
β	0.7

Table 3.1: Environment parameters employed in the model.

3.3.2 Agent Components

Until now, I have discussed the environment settings of the model. It's time to describe the agent components with all the parameters involved, starting from

the agent's policy. The agent will choose a specific action after evaluating each of them: the action with the highest Q-value is selected. However, always selecting the best action is not a good choice, indeed a bit of exploration is needed to find new scenarios that could potentially lead to more rewards in the long run (the so-called *exploration-exploitation tradeoff*). In this sense, the literature seems aligned with the so-called ε-greedy policy which works as follows:

$$a = \begin{cases} \arg\max_a Q\left(s_t, a_t, \boldsymbol{\theta}_t\right) & \text{with probability } 1-\varepsilon \\ a_t & \text{with probability } \varepsilon \end{cases} \tag{3.5}$$

where ε is a parameter that controls the likelihood of choosing the best action or a random action. Basically, ε is the probability of selecting a random action and $1-\varepsilon$ is the probability of going greedy. The random action is sampled using a uniform distribution. At the beginning $\varepsilon = 1$, and as a result, the agent is obliged to select actions randomly. This choice is due to the fact that, at the beginning, the agent knows nothing about the environment and it has to explore it properly to understand its dynamics. As time passes, ε decreases linearly until it reaches a minimum value of 0.1, meaning that the agent will still explore sometimes. The initial and the final ε values (ε_{max} and ε_{min} respectively) have been chosen based on the literature. Conversely, the linear reduction of ε is strongly influenced by the length of the data set, making this parameter specific to the user's preferences and needs. I decided to decrease ε in such a way that the minimum value is reached after, more or less, six months. In my opinion, this is a reasonable time window since it allows the agent to explore more heavily for enough time without impacting too much in the long run since random action in financial trading could be catastrophic.

The agent estimates the Q-value for each action before selecting one of them. The Q-value estimation is a critical component in a reinforcement learning model. In this research, a feedforward neural network is employed as the Q-function approximator, with the input layer containing neurons equal to the state signal's dimension and the output layer having three neurons corresponding to the available actions. Typically, there exists an unspecified number of hidden layers h between the input and output layers, and these layers need to be determined beforehand by the user. The same rationale applies to the number of neurons x within each

hidden layer. In this particular study, two variations of the network are created, one with a single hidden layer and the other with two hidden layers. This is done to investigate whether adding an extra hidden layer could enhance the agent's action estimations and subsequently improve its learning process. Regarding the number of neurons in both cases, with either one or two hidden layers, it is $k = 10$. To explain this choice, it's necessary to revisit the state signal. As previously mentioned, the study involves experimenting with different state signals by adjusting the n parameter, which controls the number of time steps. For instance, when $n = 1$, the state signal becomes a five-element vector, resulting in the neural network's input layer consisting of 5 neurons. Given that the input layer has 5 neurons, following Kolmogorov's representation theorem, the number of neurons in the subsequent layer should be a maximum of 10. I have maintained this configuration of 10 neurons when using different state signals because I aimed to observe how altering the n parameter would influence the agent's learning process. The network weights are initialized according to the guidelines outlined in [8]. Specifically, the weights of the hidden layers follow the initialization scheme proposed by He et al., while the weight matrix of the output layer is initialized with values drawn from a normal distribution with a mean of 0 and a standard deviation of 0.001. Biases are uniformly set to zero.

As detailed in the previous chapter, the agent employs a replay memory buffer to store $N = 480$ environment transitions. Periodically, for every $C = 4$ environment step, the agent randomly samples $batch\_size = 96$ transitions from this buffer to train its neural network. Essentially, it learns from past environmental experiences. It's crucial to remember the existence of a second neural network known as the target network, which is responsible for estimating the Q-values of the next state. These estimated values are used to compute target values, against which the agent evaluates its prediction errors. The target network shares the same structure as the online network and is initialized in the same manner. However, it remains static throughout training and does not undergo weight updates through the backpropagation algorithm. The occasion on which the target network's weights are modified occurs in every $W = 96$ environment step. At this point, the weights

of the online network are copied and updated in the target network.

Many reinforcement learning models employed in financial trading tend to treat reinforcement learning as if it were a supervised problem. Instead of executing the agent in a continuous online manner, they opt for a sliding train-test set approach. In this approach, they initially train the agent using the training set and then evaluate its performance on the testing set. Subsequently, the testing set becomes the new training set, and this cycle repeats dynamically. This approach proves effective, furthermore, it provides a way for configuring various model hyperparameters. On the other hand, I have chosen to adopt the online learning methodology because it aligns better with the inherent nature of reinforcement learning, where an agent learns autonomously by interacting with its environment. Another motivating factor behind this choice is the considerable computational time that the sliding train-test set method would require. Given that I have not employed reinforcement learning in a supervised learning fashion, I lack the option to optimize the hyperparameters. Consequently, I have decided to establish all the previously explained parameters, along with certain other parameters of the neural network such as the learning rate and optimizer, by drawing inspiration from the existing academic literature. Specifically, I have taken hints from [8], as it closely resembles the configuration I am working with. Table 3.2 summarizes all the aforementioned agent's parameters.

Parameter	Value
ε_{max}	1
ε_{min}	0.1
ε_{decr}	1.5625×10^{-4}
x	10
h	1, 2
$learning\_rate$	1×10^{-3}
$optimizer$	RMSprop
N	480
$batch\_size$	96
C	4
W	96

Table 3.2: Agent's parameters employed in the model.

3.3.3 Operative Signal

As previously discussed, at each time step t, the agent selects an action denoted as $a_t \in \mathcal{A}$. Consequently, after the simulation concludes, a list of actions chosen by the agent is generated. However, it is important to note that running the simulation multiple times yields different lists of actions. This variability arises from the initial state of the agent, where the weights of its neural network are randomly initialized. Since the agent's action selection relies on the estimates produced by the neural network, these initial randomizations significantly influence the agent's decision-making process. To mitigate this variability and ensure robust evaluation, the simulation is repeated $K = 50$ times for each set of parameters. As a result of this process, at each time step t there are precisely K actions available, as if multiple simulations were running simultaneously. Consequently, to arrive at a single action choice for each time step, it becomes necessary to employ an aggregation method. [3] proposes the following approach:

$$\overline{a_t} = \frac{\sum_{k=1}^{K} a_{t,k}}{K}. \tag{3.6}$$

In this equation, $a_{t,k}$ represents the k-th action at time step t, K denotes the total number of simulations conducted, and $\overline{a_t}$ is the average action selected at time step

t based on all K actions. Given that the agent's action space $\mathcal{A}$ consists of the values $\{-1, 0, 1\}$, it is necessary to map the average action $\overline{a_t}$ to one of the three values within the agent's action space. Consequently, the next step involves the following decision-making process:

$$a_t = \begin{cases} -1 & \text{if } \overline{a_t} \in [-1, -1/3] \\ 0 & \text{if } \overline{a_t} \in [-1/3, 1/3] \\ 1 & \text{if } \overline{a_t} \in [1/3, 1] \end{cases} \tag{3.7}$$

In this equation, a_t represents the final action choice at time step t and it is determined based on the value of $\overline{a_t}$ ensuring that it falls within the predefined intervals.

3.4 Experiments

Now that the model structure has been defined, the experimental methodology is outlined. Initially, I downloaded the 30-minute EUR/USD data from Dukascopy Bank, a Swiss online financial institution. This data set covers the period from January 2019 to December 2022, encompassing four years of testing. This timeframe is chosen for its suitability in representing different economic scenarios. The year 2019 exhibits relative stability, while the onset of 2020 marks the emergence of the COVID-19 pandemic, introducing substantial uncertainty, especially in the Forex market. Figure 3.2 shows, in the upper plot, the EUR/USD exchange rate in the aforementioned period, while in the lower chart, the 30-minute returns are plotted. Looking at the exchange rate over time, it's very balanced since the first part is characterized by a horizontal movement without a clear trend (perhaps a small allusion to a bearish trend). The agent could find itself having difficulty making a profit in this period, both because of the lack of a strong trend and especially because of the agent's initial exploration. Following this bearish-horizontal period, the exchange rate starts a bullish trend where the agent ideally should go long. Finally, a strong bearish trend begins in May 2021, in which the agent should go short on EUR/USD, eventually followed by a trend reversal. As mentioned earlier,

these four years encompass a variety of situations that the agent must navigate. Therefore, if the agent can generate substantial profit during this period, it would serve as strong evidence of the model's efficacy.

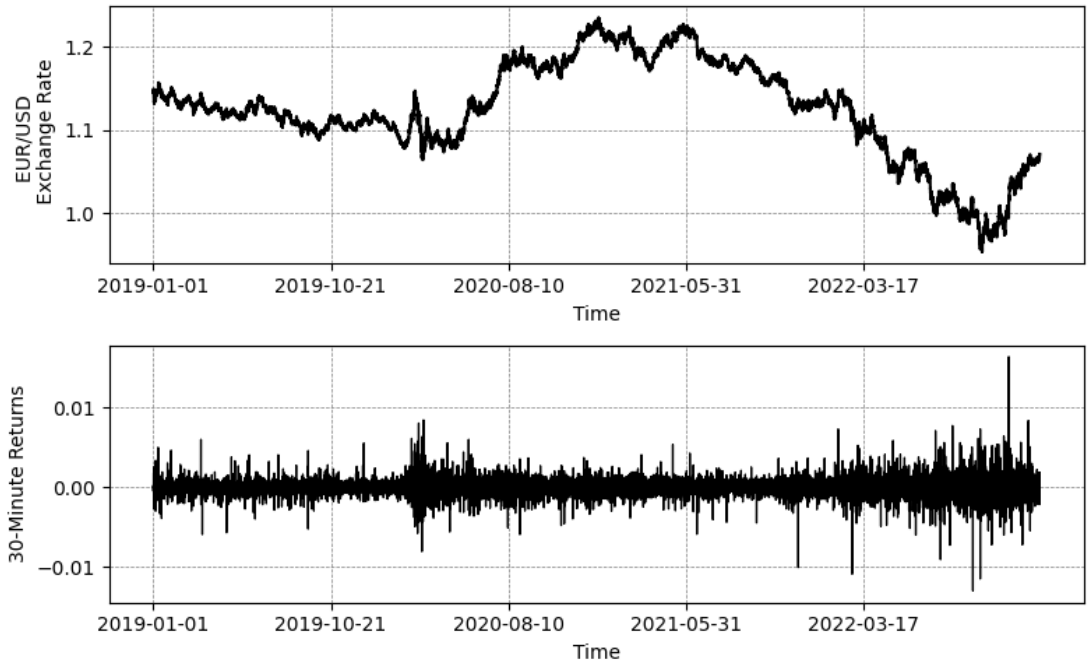

Figure 3.2: The upper chart reproduces the 30-minute timeframe EUR/USD data. The lower chart depicts the one-step returns.

At the end of the simulation, the Calculator class will print out the following statistics:

1. Annualized return (%)
2. $> v_0$ (%)
3. Maximum drawdown (%)
4. Annual trades (#)
5. On position (%)
6. Long trades (%)
7. Gross win ratio (%)
8. Net win ratio (%)

The primary metric of importance is the initial one, as it determines whether the agent managed to generate a profit. Furthermore, it's not only essential to ascertain whether the annualized return is positive or negative, but the magnitude of this value also holds significance. Given the complexity of this model, both in

terms of its implementation and computational requirements, one would expect a notably superior performance compared to standard trading strategies.

Assessing the agent's performance solely based on whether it generated a profit at the conclusion of the testing period is an incomplete measure. Notably, if the agent sustained losses for the majority of the period and only managed to reverse its performance towards profitability at the very end, such an outcome would be considered suboptimal. The second statistical metric, however, provides valuable insights in this context. Specifically, it quantifies the percentage of instances where the agent's balance exceeded its initial value. This metric proves particularly useful for dynamic analysis of the agent's performance over the entire duration of the testing period, offering a more comprehensive perspective beyond just the starting and final balance.

The third metric is a widely employed risk assessment measure within the realm of finance, commonly referred to as the *maximum drawdown*. It is a crucial risk metric in finance that measures the largest peak-to-trough decline in an investment's value or portfolio over a specified period. In other words, it quantifies the maximum loss an investment or trading strategy has incurred from its previous peak value to its lowest point within the chosen time.

The fourth metric pertains to the yearly count of trades initiated by the agent. It's important to note that the agent makes decisions every 30 minutes, potentially resulting in a substantial number of trades. This can pose an issue due to the associated costs, the spread, incurred with each trade. Hence, the ideal scenario involves the agent learning a profitable policy and avoiding opening a high number of trades.

The subsequent metric represents the proportion of time the agent spends in active positions, providing insights into the agent's market engagement.

The sixth metric examines the percentage of long trades carried out by the agent, understanding the agent's trading preferences, whether it favors long or short trades. This metric provides insights into the agent's overall trading behavior.

The last two statistics measure the gross win rate and net win rate. The gross win rate is determined by dividing the number of successful trades, without

accounting for the spread, by the total number of trades. Conversely, the net win rate is calculated in a similar manner, but it considers the impact of the spread when evaluating trade success.

All of the statistics mentioned earlier are typically associated with the operational signal. Nevertheless, I have opted to maintain records of these metrics for each individual simulation conducted. Specifically, for each of the aforementioned metrics, I have computed a 95% confidence interval using bootstrapping. This approach offers insights into the K simulations that were conducted, treating each simulation as if it were the definitive operational signal.

3.5 Benchmark

In order to validate our reinforcement learning model, it should be compared with some sort of benchmark. Commonly, a benchmark is a baseline trading system, possibly one of the most employed ones, so as to compare the results of a newly developed trading system to the standard approach. The rationale behind this procedure is that the baseline trading system is usually very simple to develop rather than the new potential model. Consequently, if the profits are more or less similar, priority is given to the simpler one. This is the case for our reinforcement learning model, indeed it can be considered as a very sophisticated automated trading system. In the literature, the most common benchmarks are the *buy and hold* (B&H) or the *sell and hold* (S&H) strategies. As the names suggest, the former buys the asset and holds it until the end of the period, while the latter is the same approach but it shorts the asset instead of buying it. In this work, I have decided to create a benchmark in a different way: I will test six different trading strategies, the most common ones, and take the most profitable as the final benchmark.

The first trading rule is based on a technical indicator, the *Relative Strength Index* (RSI), developed by J. Welles Wilder Jr. and introduced in his 1978 book, *New Concepts in Technical Trading Systems*. It is computed using the following

formula:

$$RSI = 100 - \frac{100}{1 + RS}$$

$$RS = \frac{\text{Average Gain}}{\text{Average Loss}}$$

where RS, the so-called *relative strength*, is a ratio between average gains and average losses over a specified period (gains are positive price changes while losses are negative price changes). The RSI is a momentum indicator that, like all the other momentum indicators, has the role of finding a trend reversal. From a mathematical point of view, it is a time series with values ranging from 0 to 100. In order to define the RSI indicator, three parameters should be set: the window length, the overbought and oversold levels. The window length is necessary to compute the RSI itself, more specifically it is used inside the RS formula. It is usually set to 14 time periods. The overbought level is the RSI value at which the asset is said to be overbought which means too many people are buying it and it could potentially start reversing its trend. Consequently, when the RSI reaches this level, a short position could be a reasonable trade. On the other hand, the oversold level is the RSI value at which the asset is said to be oversold and it is usually a buy signal. I have set the overbought and oversold levels to 80 and 20 respectively. The trading rule that will trigger a long or short position is the following:

1. If $RSI(t) > OB$ AND $RSI(t-1) \leq OB$:

   $\text{Signal}_{RSI}(t) = SHORT$

2. elif $RSI(t) < OS$ AND $RSI(t-1) \geq OS$:

   $\text{Signal}_{RSI}(t) = LONG$

3. else: $\text{Signal}_{RSI}(t) = \text{Signal}_{RSI}(t-1)$

The second trading system relies again on a technical indicator known as Bollinger Bands (BB), developed by John Bollinger in the 1980s. This indicator belongs to the volatility indicators family it has three main components: the middle, the lower, and the upper bands. The middle band is a simple moving average of

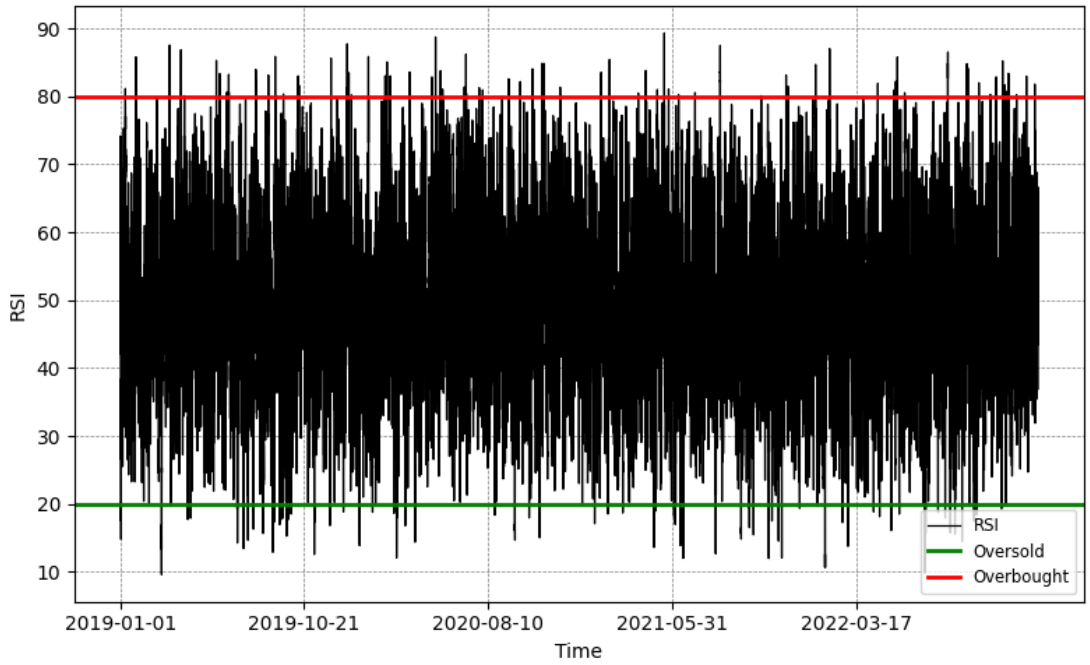

Figure 3.3: RSI on 30-minute timeframe EUR/USD data (January 2019 - December 2022).

the asset price for a given lookback period. The upper and the lower bands are calculated respectively by adding and subtracting a certain number of standard deviations from the middle band. The BB indicator. The trading rule which triggers a long or short position is the following:

1. If $\text{Price}(t) > \text{Bolli}_L(t)$ AND $\text{Price}\ (t-1) \leq \text{Bolli}_L(t-1)$:

   $\text{Signal}_{\text{Boll}}\ (t) = \text{LONG}$

2. elif $\text{Price}(t) < \text{Bolli}_U(t)$ AND $\text{Price}\ (t-1) \geq \text{Bolli}_U(t-1)$:

   $\text{Signal}_{\text{Boll}}\ (t) = \text{SHORT}$

3. else: $\text{Signal}_{Boll}(t) = \text{Signal}_{Boll}(t-1)$

Basically, when the asset price touches the upper band, the asset is said to be overbought and a short position is triggered. Conversely, when the asset price touches the lower band, the asset is said to be oversold, and a long position is triggered. Even the BB indicator needs some parameters to be set a priori, in particular, the lookback period for the simple moving average and the number

of standard deviations to find the upper and lower bands. The standard values of these two parameters are 20 for the lookback period and 2 for the number of standard deviations. In Figure 2 the BB indicator has a lookback period of 960 because by using the standard value of 20 the chart did not make sense. The value 960 has been chosen because the 30-minute timeframe corresponds exactly to 20 days.

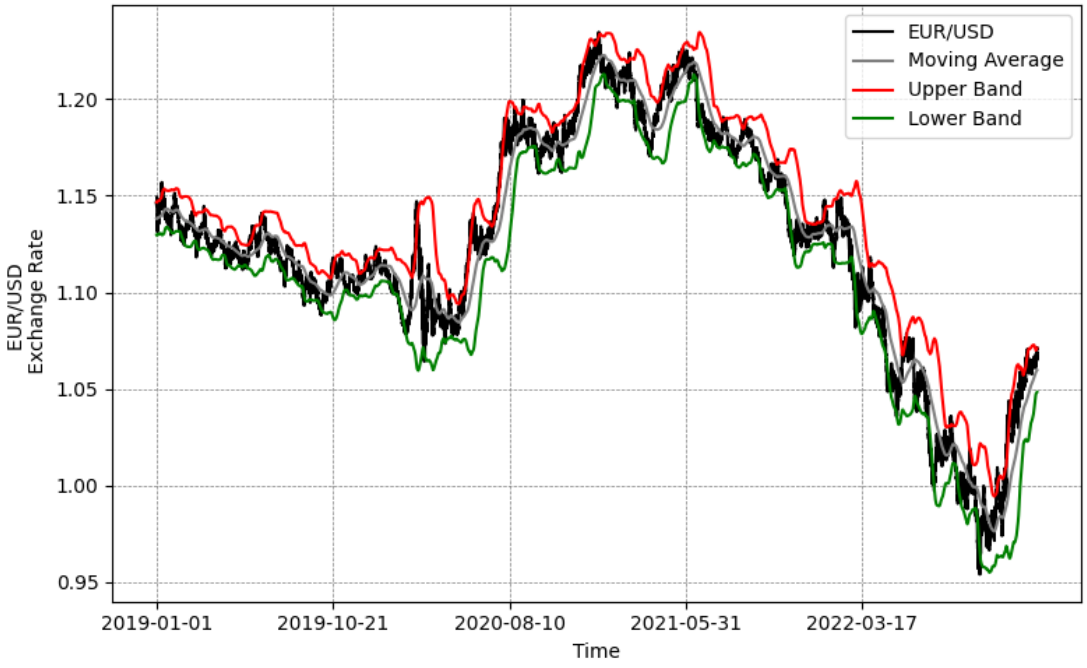

Figure 3.4: Bollinger Bands on 30-minute timeframe EUR/USD data (January 2019 - December 2022).

The third trading system, again employing a technical indicator, is based on the so-called Moving Average Converge Divergence (MACD) indicator, developed by Gerald Appel in the late 1970s. The MACD indicator is one of the trend-following indicators that have the goal of making a profit by following a trend rather than entering in reverse points. This indicator has two components, the MACD line and the signal line. The former is computed by subtracting a long exponential moving average (EMA) from a short EMA (long and short mean the length of the lookback time window). The latter is an EMA of the MACD line. The trading rule from which trading signals are triggered is given by the following:

1. If $MACD_1(t) >$ Bench(t) AND MACD$_1(t-1) \leq$ Bench$(t-1)$:

$\text{Signal}_{\text{MACD}}(t) = \text{LONG}$

2. elif $\text{MACD}_1(t) < \text{Bench}(t)$ AND $MACD_2(t-1) \geq \text{Bench}(t-1)$:

   $\text{Signal}_{\text{MACD}}(t) = \text{SHORT}$

3. else: $\text{Signal}_{MACD}(t) = \text{Signal}_{MACD}(t-1)$

As the trading rule suggests, traders typically look for two main types of signals when using the MACD: a bullish signal, when the MACD line crosses above the signal line, suggesting a potential buy signal; a bearish signal when the MACD line crosses below the signal line, indicating a potential sell signal. The MACD chart is depicted in Figure 3. The parameters to define are three lookback periods: two for the fast and slow EMAs computation and one for the signal line. The standard values are 12 and 26 for the fast EMA and slow EMA respectively, while 9 is the default value for the signal line lookback period. As in the BB case, the standard parameter values did not suit the 30-minute EUR/USD data. Consequently, I have made the same adjustment of the BB indicator: instead of the standard values 12, 26 and 9, I have employed 576, 1248 and 432 respectively.

The fourth trading system takes inspiration from John Bollinger's suggestion of using the BB indicator in conjunction with two or three non-correlated indicators such as RSI and MACD. As a result, I have decided to combine all these three indicators with their respective trading rules. The combination criteria is simple and intuitive, basically, it is a majority voting: starting from the three action signals, one for each trading rule, the final combined signal is given by a majority voting approach at each time step.

The last two trading systems are the most standard trading approaches, buy and hold and sell and hold strategies, explained at the beginning of this section. All these six trading systems are said to be always in the market, since, once the first trade is opened, the action could be long or short, without the possibility of going out of the market.

Table 3.3 presents the results derived from the aforementioned six distinct trading systems during a four-year testing period. Most of these systems yielded negative returns, except for the Bollinger and S&H trading strategies. Notably, the Bollinger

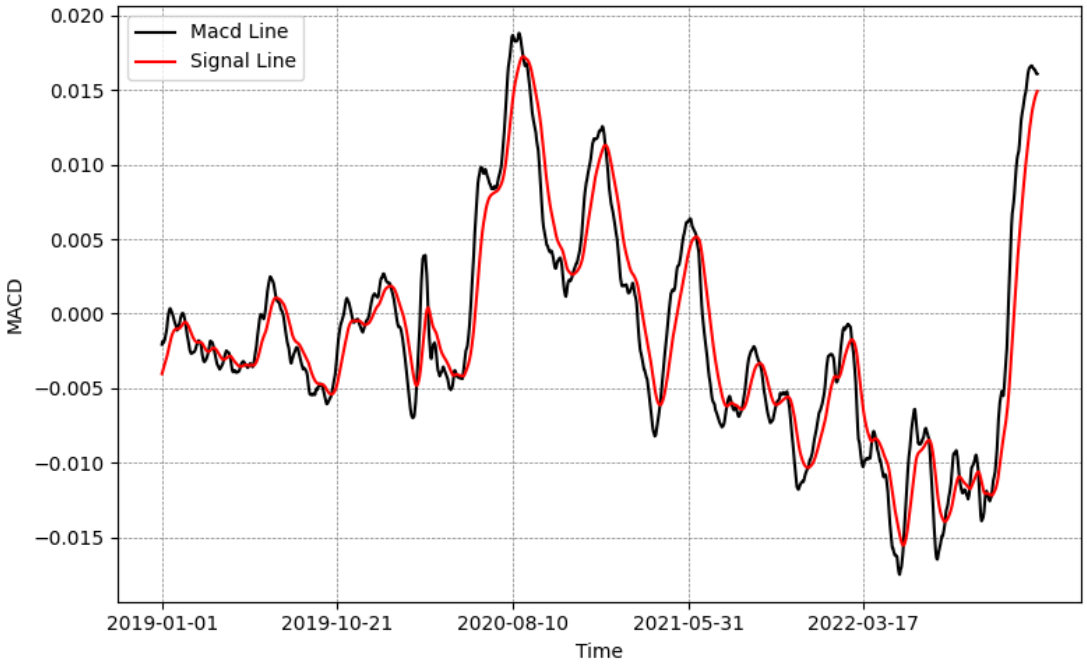

Figure 3.5: MACD on 30-minute timeframe EUR/USD data (January 2019 - December 2022).

strategy outperformed the others with an annual return of 3.78%, maintaining a balance higher than the initial one for 99.61% of the time. Consequently, this performance has been established as the benchmark level, set at a 3.78% annual return. Nevertheless, because the implemented strategy is considerably simpler in comparison to the DQN agent, I have opted to increase the benchmark to a 4% annual return. This adjustment is motivated by the expectation that the agent should not merely achieve a marginally superior performance.

Trading Strategy	Annualized Return (%)	$> v_0$ (%)	Maximum Drawdown (%)	Annual Trades	On Position (%)	Long Trades (%)	Gross Win Ratio (%)	Net Win Ratio (%)
RSI	−2.92	43.53	23.55	19.25	100.0	49.35	41.56	41.56
BOLL	**3.78**	**99.61**	**11.58**	**9.75**	**100.0**	**48.72**	**41.03**	**41.03**
MACD	−4.96	0.37	28.02	13.75	100.0	50.91	43.64	43.64
RSI-BOLL-MACD	−3.28	39.11	26.03	26.25	100.0	49.52	42.86	42.86
B&H	−2.17	32.98	24.97	0.25	100.0	100.0	0.0	0.0
S&H	1.58	65.25	15.95	0.25	100.0	0.0	100.0	100.0

Table 3.3: Comparison between benchmark trading rules outcomes.

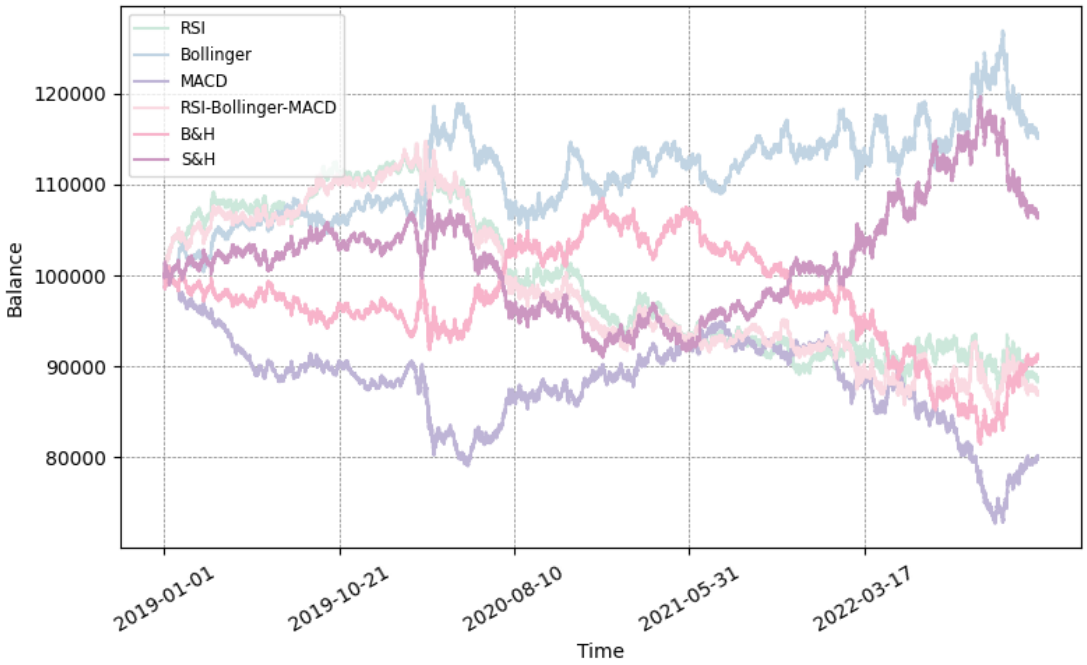

Figure 3.6: Balance over time between the benchmark trading rules.

3.6 Results

In the following section, the results obtained from the conducted experiments are described. As introduced previously, various parameters were adjusted to assess their impact on the agent's performance. Consequently, simulations were conducted for each parameter combination.

Before running the actual simulations, an initial test involved setting $\delta = 0$ (without the spread). The rationale behind this choice was to isolate and test the agent's learning capability by removing the spread. Indeed, if the agent couldn't operate profitably in an environment without spread, it would likely struggle with the addition of costs. In this no-commissions setting, simulations were not carried out for all possible combinations of parameters due to the extensive computational time required. Specifically, for each reward function, only a one-time step feature vector and one hidden layer were tested ($n = 1$ and $h = 1$ respectively). The results are presented in Table 3.4.

Examining the results, it is evident that all three configurations managed to generate a profit at the end of the four-year testing period. The Log Return and

Reward Function	Annualized Return (%)	$> v_0$ (%)	Maximum Drawdown (%)	Annual Trades	On Position (%)	Long Trades (%)	Gross Win Ratio (%)	Net Win Ratio (%)
RESULTS OF 50 SIMULATIONS (95% BOOTSTRAP CONFIDENCE INTERVAL)								
Log Return	[6.19, 9.1]	[60.46, 75.71]	[9.24, 11.21]	[4371, 4450]	[66.51, 67.3]	[49.44, 50.07]	[49.74, 49.97]	[49.74, 49.97]
Sharpe Ratio	[0.36, 1.71]	[44.56, 60.69]	[10.96, 13.03]	[1524, 1555]	[63.69, 69.31]	[49.92, 51.2]	[49.62, 49.92]	[49.62, 49.92]
Modified Log Return	[16.83, 19.28]	[90.02, 93.83]	[9.05, 10.48]	[5027, 5089]	[87.17, 87.78]	[49.85, 50.5]	[49.54, 49.85]	[49.54, 49.85]
RESULTS OF THE OPERATIVE SIGNAL								
Log Return	14.47	84.17	3.21	1923	25.69	49.05	50.1	50.1
Sharpe Ratio	1.12	64.8	10.42	477	55.53	40.4	50.94	50.94
Modified Log Return	23.07	92.32	8.51	4302	68.54	49.92	49.29	49.29

Table 3.4: Outcomes of the simulations conducted without the spread and employing parameters $n = 1$ and $h = 1$. Both the 95% bootstrapped confidence intervals and the operative signal metrics are shown.

Modified Log Return reward functions achieved significant results with annual returns of 14.47% and 23.07%, respectively. The Sharpe Ratio reward obtained a 1.12% annual return, which, while positive, did not meet the high expectations. An interesting observation is that the operational signal performance improved significantly compared to the performance of individual agents (except for the Sharpe Ratio, which performed similarly to single agents). Overall, these results highlight the high potential of the DQN agent and suggest that the agent's learning capability is effective.

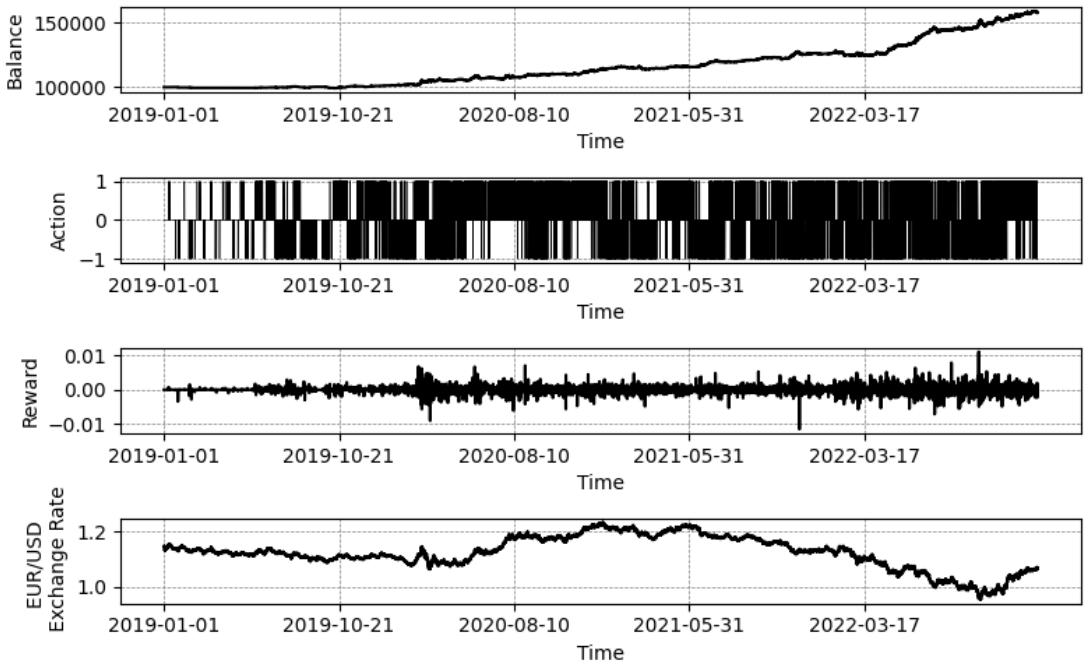

Figure 3.7: Operative signal using Log Return reward function and without spread.

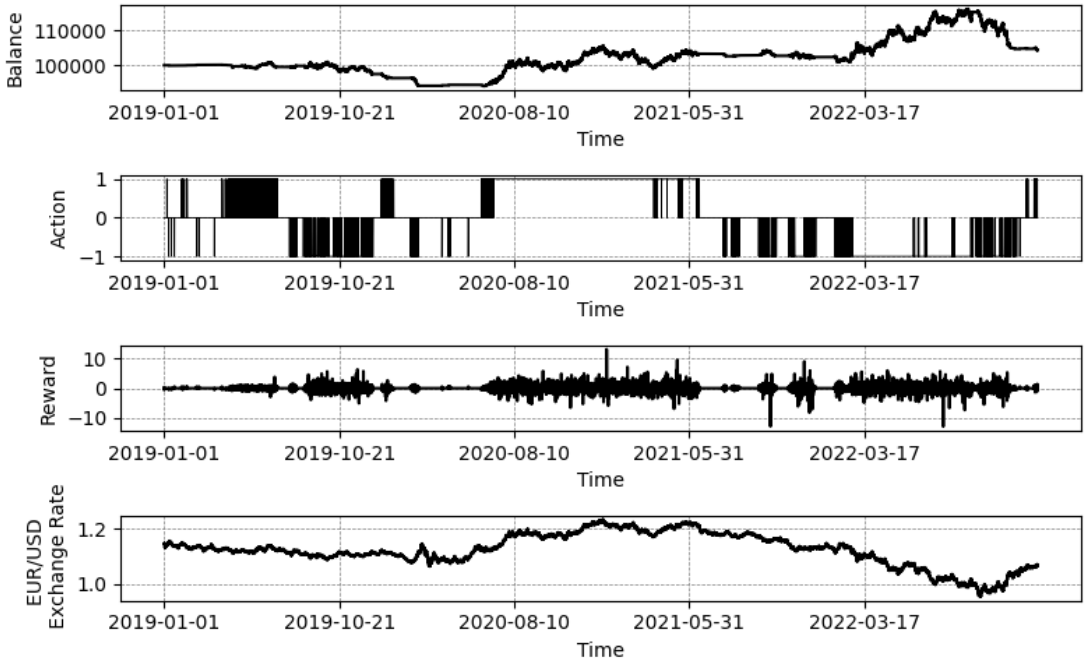

Figure 3.8: Operative signal using Sharpe Ratio reward function and without spread.

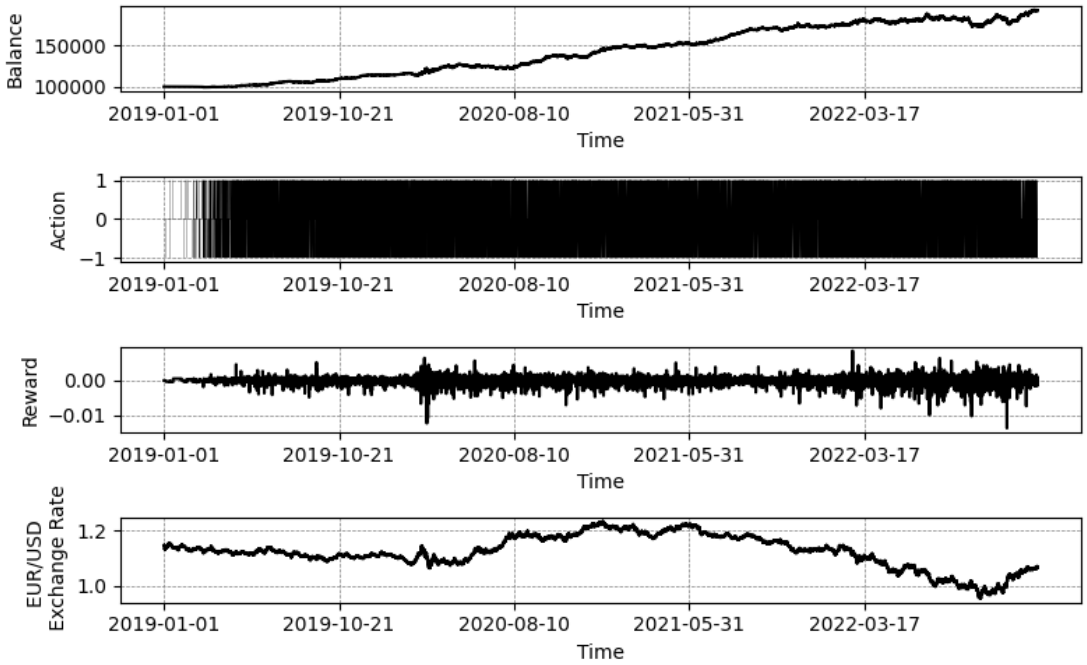

Figure 3.9: Operative signal using the Modified Log Return reward function and without spread.

3.6.1 DQN Results

Now that a profitable policy has been demonstrated in a zero-spread environment, it is time to test the agent in a real-world scenario, where each trade comes with a cost in the form of the bid-ask spread ($\delta = 0.3$ bp).

Log Return Reward Function

Table 3.5 summarizes the experiments performed with the DQN agent and the Log Return reward function for each combination of parameters.

REWARD: LOG RETURN, AGENT: DQN									
n	Hidden Layers	Annualized Return (%)	$> v_0$ (%)	Maximum Drawdown (%)	Annual Trades	On Position (%)	Long Trades (%)	Gross Win Ratio (%)	Net Win Ratio (%)
RESULTS OF 50 SIMULATIONS (95% BOOTSTRAP CONFIDENCE INTERVAL)									
1	1	$[-6.76, -5.18]$	$[0.51, 4.51]$	$[28.51, 32.63]$	$[4229, 4294]$	$[66.16, 67.23]$	$[45.28, 45.75]$	$[49.42, 49.66]$	$[47.17, 47.4]$
4	1	$[-9.02, -7.6]$	$[0.79, 1.91]$	$[35.13, 39.36]$	$[4147, 4210]$	$[64.79, 65.99]$	$[45.3, 45.96]$	$[49.51, 49.73]$	$[47.25, 47.48]$
8	1	$[-9.12, -7.79]$	$[0.94, 2.22]$	$[35.17, 39.44]$	$[4132, 4203]$	$[64.74, 65.79]$	$[45.24, 45.85]$	$[49.46, 49.67]$	$[47.2, 47.42]$
16	1	$[-10.01, -8.74]$	$[1.32, 2.99]$	$[38.51, 42.68]$	$[4101, 4149]$	$[64.19, 65.18]$	$[45.2, 45.91]$	$[49.5, 49.68]$	$[47.2, 47.38]$
1	2	$[-7.43, -6.3]$	$[0.76, 1.54]$	$[30.79, 34.25]$	$[4231, 4309]$	$[66.92, 67.72]$	$[45.32, 45.79]$	$[49.58, 49.75]$	$[47.33, 47.5]$
4	2	$[-9.78, -8.63]$	$[0.66, 1.54]$	$[37.83, 41.49]$	$[4113, 4188]$	$[65.53, 66.4]$	$[45.45, 45.98]$	$[49.65, 49.85]$	$[47.38, 47.56]$
8	2	$[-9.78, -8.56]$	$[0.64, 1.66]$	$[37.25, 41.36]$	$[4107, 4178]$	$[65.15, 66.29]$	$[45.66, 46.19]$	$[49.6, 49.77]$	$[47.36, 47.52]$
16	2	$[-9.88, -8.72]$	$[0.63, 1.56]$	$[37.68, 41.72]$	$[4070, 4134]$	$[65.49, 66.49]$	$[45.13, 45.54]$	$[49.58, 49.76]$	$[47.3, 47.47]$
RESULTS OF THE OPERATIVE SIGNAL									
1	1	2.05	14.65	10.45	1744	28.25	29.43	46.02%	44.3
4	1	-1.01	0.0	12.94	1276	23.44	24.75	48.12	45.96
8	1	-1.12	0.69	9.7	1099	22.09	25.28	49.49	47.37
16	1	-3.61	0.0	14.89	994	21.75	23.66	48.43	46.37
1	2	5.53	39.56	6.19	1512	25.93	27.08	45.95	44.0
4	2	-2.64	6.18	14.76	1063	22.36	23.81	47.49	45.64
8	2	-1.8	3.56	11.54	948	20.85	22.92	47.84	45.52
16	2	-2.0	2.8	10.71	914	22.32	21.4	48.37	46.4

Table 3.5: Outcomes of the simulations conducted employing the DQN agent with the Log Return reward function for each combination of parameters. Both the 95% bootstrapped confidence intervals and the operative signal metrics are shown.

Examining the annualized returns reveals that the majority of the combinations yielded negative results. Positive outcomes were observed only when the state signal was described with $n = 1$, resulting in notable returns of 2.05% and 5.53% with one and two hidden layers, respectively. However, it's important to note that, overall,

the results are predominantly negative.

The addition of the second hidden layer improved the agent's annual return for cases with $n = 1$ and $n = 16$, while it had a detrimental effect in the other two scenarios.

Maintaining a balance higher than the initial one proved challenging for the agent, as three out of eight combinations had this value near or exactly at 0%. Even the two combinations with positive annual returns achieved relatively low percentages (39.56% and 14.55% for the best and second-best combinations, respectively).

Despite the negative performance in terms of annual returns, the maximum drawdown percentage remained relatively moderate, with the worst statistics at 14.89%. This suggests that, although the agent struggled to increase the initial balance, it didn't incur significant losses most of the time.

One positive aspect is the improvement in performance brought by the operative signal compared to the achievements of individual agents. This may be attributed to the operative agent's fewer trades, resulting in reduced total commissions. However, since the operative signal improved the performance also with $\delta = 0$, another explanation may be that, by averaging and aggregating all the actions for each time step, the non-zero actions in the operative agent tend to be the significant ones, and consequently it's more likely to perform profitable trades.

Another noteworthy observation is the balance between long and short positions. In the behavior of individual agents, there's a balanced distribution with 45% in long positions and, consequently, 55% in short positions. However, the operative signal comprises approximately only 25% long trades, indicating that many long trades were not significant. Surprisingly, although the operative signal improved performance in terms of annual return and maximum drawdown, it decreased the win ratio.

In general, the results are unsatisfactory, with only two combinations yielding positive returns, one of which surpasses 4%, making it superior to the benchmark.

Sharpe Ratio Reward Function

Now it's time to evaluate the performance of the DQN agent with the Sharpe Ratio reward function. Table 3.6 summarizes the experiments for each combination of parameters.

REWARD: SHARPE RATIO, AGENT: DQN									
n	Hidden Layers	Annualized Return (%)	$> v_0$ (%)	Maximum Drawdown (%)	Annual Trades	On Position (%)	Long Trades (%)	Gross Win Ratio (%)	Net Win Ratio (%)
RESULTS OF 50 SIMULATIONS (95% BOOTSTRAP CONFIDENCE INTERVAL)									
1	1	$[-2.63, -1.5]$	$[4.89, 11.04]$	$[16.82, 19.43]$	$[1647, 1669]$	$[87.29, 89.37]$	$[46.17, 47.22]$	$[49.71, 50.02]$	$[47.55, 47.85]$
4	1	$[-2.85, -1.49]$	$[5.39, 13.46]$	$[17.5, 20.33]$	$[1658, 1683]$	$[88.16, 90.05]$	$[46.58, 47.64]$	$[49.77, 50.0]$	$[47.68, 47.93]$
8	1	$[-3.78, -2.28]$	$[4.45, 11.22]$	$[18.57, 21.72]$	$[1667, 1689]$	$[88.06, 89.94]$	$[46.42, 47.44]$	$[49.81, 50.06]$	$[47.67, 47.92]$
16	1	$[-2.91, -1.47]$	$[6.33, 14.93]$	$[17.29, 20.43]$	$[1657, 1680]$	$[87.5, 89.81]$	$[46.43, 47.51]$	$[49.78, 50.01]$	$[47.65, 47.9]$
1	2	$[-3.66, -2.22]$	$[3.48, 11.66]$	$[18.67, 21.43]$	$[1632, 1663]$	$[81.4, 84.52]$	$[46.68, 47.65]$	$[49.68, 49.9]$	$[47.5, 47.7]$
4	2	$[-3.67, -2.27]$	$[5.5, 14.44]$	$[17.84, 21.46]$	$[1660, 1691]$	$[84.0, 86.5]$	$[46.8, 48.18]$	$[49.77, 50.04]$	$[47.67, 47.93]$
8	2	$[-4.28, -2.48]$	$[3.66, 12.15]$	$[19.89, 24.15]$	$[1653, 1686]$	$[83.08, 86.42]$	$[46.3, 47.56]$	$[49.72, 50.0]$	$[47.58, 47.84]$
16	2	$[-3.18, -1.9]$	$[3.83, 9.89]$	$[17.28, 20.15]$	$[1672, 1711]$	$[84.74, 87.68]$	$[46.05, 47.29]$	$[49.81, 50.09]$	$[47.71, 47.99]$
RESULTS OF THE OPERATIVE SIGNAL									
1	1	4.44	75.91	9.91	172	78.69	23.03	50.44	48.4
4	1	3.78	87.93	9.01	182	74.86	24.62	49.24	46.49
8	1	3.56	64.55	9.81	240	73.51	13.65	50.52	46.98
16	1	4.44	94.45	8.67	216	76.35	24.07	48.26	46.53
1	2	0.29	28.04	14.26	401	72.19	30.67	51.31	48.88
4	2	1.34	40.15	12.41	367	71.28	31.54	50.68	48.5
8	2	1.06	32.47	12.31	434	72.08	19.56	50.51	48.47
16	2	2.37	34.07	11.32	326	73.61	22.27	51.92	50.23

Table 3.6: Outcomes of the simulations conducted employing the DQN agent with the Sharpe Ratio reward function for each combination of parameters. Both the 95% bootstrapped confidence intervals and the operative signal metrics are shown.

This time, all operative agents ended up with positive returns at the conclusion of the four-year testing period.

Interestingly, the addition of the second layer had a significantly adverse impact on performance across all metrics, including the annual return, the percentage of times the balance exceeded the initial value and the maximum drawdown. Conversely, when employing only one hidden layer, performance remained consistent across various state signals, maintaining stable annual returns and maximum drawdowns.

Once again, the combined operative agent greatly outperformed the performance of individual agents. This is evident when examining the 95% bootstrapped confidence intervals describing the annual return of individual agents, which were consistently negative, meaning significant negative behavior. In contrast, all operative signals managed to achieve a positive annual return. Averaging and aggregating the actions of individual agents resulted in a reduced number of total trades and a lower percentage of long trades.

The standout combination of parameters is undoubtedly the one with $n = 16$ and $h = 1$, which achieved a 4.44% annual return, accompanied by an 8.67% maximum drawdown. Furthermore, this combination maintained a balance higher than the initial one for an impressive 94.45% of the time.

In summary, the results are positive; however, when compared to the benchmark, only two out of the eight combinations yielded an annual return greater than 4%.

Modified Log Return Reward Function

The remaining DQN agent to evaluate is the one employing the Modified Log Return reward function. Table 3.7 summarizes the experiments performed with it for each combination of parameters.

The annual return is positive for each combination of parameters. Only one of them ($n = 1$ and $h = 1$) presents a marginal annual return of 0.74%, just slightly higher than zero.

The parameter n appears to have a significant impact on the agent's performance. Higher values of n yielded improved results, with $n = 8$ and $n = 16$ outperforming $n = 1$ and $n = 4$.

The best combinations are those with $n = 8$, which achieved impressive annual returns of 11.69% and 11.66% with one and two hidden layers, respectively.

The addition of the second layer significantly improved performance only when $n = 1$, while in other cases, it either decreased performance ($n = 4$ and $n = 8$) or showed non-significant improvement ($n = 16$).

Once again, the operative signal demonstrated a remarkable improvement compared to the behavior of individual agents, particularly in terms of annual return

REWARD: MODIFIED LOG RETURN, AGENT: DQN									
n	Hidden Layers	Annualized Return (%)	$> v_0$ (%)	Maximum Drawdown (%)	Annual Trades	On Position (%)	Long Trades (%)	Gross Win Ratio (%)	Net Win Ratio (%)
RESULTS OF 50 SIMULATIONS (95% BOOTSTRAP CONFIDENCE INTERVAL)									
1	1	$[-3.62, -2.2]$	$[12.48, 24.64]$	$[20.66, 24.33]$	$[4939, 5006]$	$[87.29, 87.87]$	$[51.35, 51.85]$	$[50.24, 50.48]$	$[48.01, 48.25]$
4	1	$[-2.94, -1.47]$	$[6.32, 15.68]$	$[19.46, 22.31]$	$[4008, 4051]$	$[88.76, 89.34]$	$[50.57, 50.88]$	$[50.09, 50.27]$	$[47.95, 48.13]$
8	1	$[0.46, 2.17]$	$[19.25, 32.04]$	$[16.22, 18.64]$	$[3692, 3731]$	$[89.54, 90.1]$	$[50.47, 50.7]$	$[49.96, 50.15]$	$[47.81, 47.99]$
16	1	$[-0.19, 1.38]$	$[12.49, 22.92]$	$[17.46, 20.08]$	$[3578, 3607]$	$[90.1, 90.62]$	$[50.49, 50.66]$	$[49.84, 49.99]$	$[47.64, 47.8]$
1	2	$[-3.29, -2.04]$	$[10.58, 22.1]$	$[20.04, 23.01]$	$[5046, 5117]$	$[87.56, 88.17]$	$[51.35, 51.81]$	$[50.32, 50.54]$	$[48.09, 48.3]$
4	2	$[-2.23, -0.65]$	$[9.46, 19.23]$	$[17.83, 21.01]$	$[4115, 4161]$	$[88.71, 89.32]$	$[50.51, 50.79]$	$[50.02, 50.21]$	$[47.86, 48.05]$
8	2	$[-0.08, 1.54]$	$[12.71, 22.69]$	$[17.31, 19.75]$	$[3821, 3868]$	$[89.01, 89.69]$	$[50.47, 50.75]$	$[49.89, 50.07]$	$[47.71, 47.89]$
16	2	$[-1.19, 0.32]$	$[9.3, 20.69]$	$[17.92, 20.75]$	$[3672, 3715]$	$[89.11, 90.08]$	$[50.53, 50.81]$	$[49.99, 50.18]$	$[47.74, 47.92]$
RESULTS OF THE OPERATIVE SIGNAL									
1	1	0.74	79.4	20.79	4145	69.11	54.3	51.76	49.49
4	1	4.64	91.52	12.04	2702	70.68	52.14	51.71	50.0
8	1	11.69	99.58	7.01	2252	71.26	52.24	50.78	49.01
16	1	8.68	76.24	13.62	1970	71.02	52.88	51.78	50.01
1	2	2.8	81.62	16.79	4127	66.94	55.21	51.87	49.7
4	2	4.4	86.7	10.94	2669	69.29	52.24	51.42	49.72
8	2	11.66	83.61	7.04	2242	69.68	51.99	50.74	49.04
16	2	8.8	64.87	11.62	1979	69.6	52.24	50.51	48.96

Table 3.7: Outcomes of the simulations conducted employing the DQN agent with the Modified Log Return reward function for each combination of parameters. Both the 95% bootstrapped confidence intervals and the operative signal metrics are shown.

and the percentage of times the balance exceeded the initial value. Furthermore, this improvement was accompanied by a reduced number of trades and an increase in market inactivity (approximately +19%).

The choice between long and short trades remained well-balanced in both single agents and the operative agent, while gross and net win ratios were both enhanced with the operative signal.

Overall, the results are positive, with the best combination achieving an 11.69% annual return and a 7.01% maximum drawdown. Moreover, this top-performing combination maintained a balance higher than the initial one for an impressive 99.58% of the time. In comparison to the benchmark, six out of eight operative agents achieved an annual return greater than 4%, with four of these six doubling the 4% benchmark or even surpassing it.

Comparison between Rewards Function

The DQN agent's learning capability has demonstrated effectiveness even in a real-world scenario with the inclusion of the spread. However, it's worth noting that the Log Return reward function yielded poor results. This reward function underperformed in comparison to the other two rewards across all combinations of parameters, except for $n = 1, h = 2$, where it outperformed the other two.

The agent using the Modified Log Return reward, which proved to be the best one, consistently outperformed the Sharpe Ratio agent in most combinations of parameters, except for $n = 1$, $h = 1$.

In terms of operative behavior, the Sharpe Ratio agent had a lower number of trades compared to the other two. On average, the Sharpe Ratio agent executed 292 annual trades, while the Log Return and Modified Log Return agents 1194 and 2761 trades, on average, respectively.

A surprising observation regarding the Sharpe Ratio agent is that it improved its performance when transitioning from the no-commission setting to the commissions setting, unlike the other agents. When comparing the $n = 1, h = 1$ setting (the unique combination of parameters tested both with and without spread for each reward function) with and without spread, it's evident that the number of annual trades decreased from 477 to 172 (-64%) for the Sharpe Ratio agent, while the Log Return and Modified Log Return agents experienced a more modest decrease of 9.3% and 3.6%, respectively. As a result, the Sharpe Ratio agent not only incurred substantially lower total commissions compared to the other two agents but also managed to significantly reduce the number of trades when transitioning from a commission-free environment to one with spreads.

Another insightful statistic worth analyzing in conjunction with the number of trades is market activity (the percentage of time the agent was in a position). Interestingly, these two statistics provide insights into the agents' behavioral operativity and they differ significantly among the three agents. Indeed, the Log Return agent had an average market activity of approximately 23% which combined with a high number of annual trades (1194), means that it frequently opened and closed positions quickly. In contrast, the Sharpe Ratio agent achieved a 74% market activity,

combined with a very low number of trades (292), indicating that it remained in the market for extended periods without frequent position changes, thus intelligently reducing spread commissions. Conversely, the Modified Log Return agent had the highest number of trades (2761) and a 70% market activity, suggesting a tendency to switch trade positions (from long to short and vice versa) more frequently.

Another key difference between the three DQN agents is the preference for long trades rather than short trades. In particular, the agent using the Modified Log Return reward function is the most balanced in these terms, with 53% of long trades on average. Conversely, the agents employing the Log Return and the Sharpe Ratio reward functions presented respectively a 25% and a 24% long trades preference.

The last statistic comparison is the win ratio. The Modified Log Return has both the highest gross win ratio (with all values greater than 50%) and net win ratio.

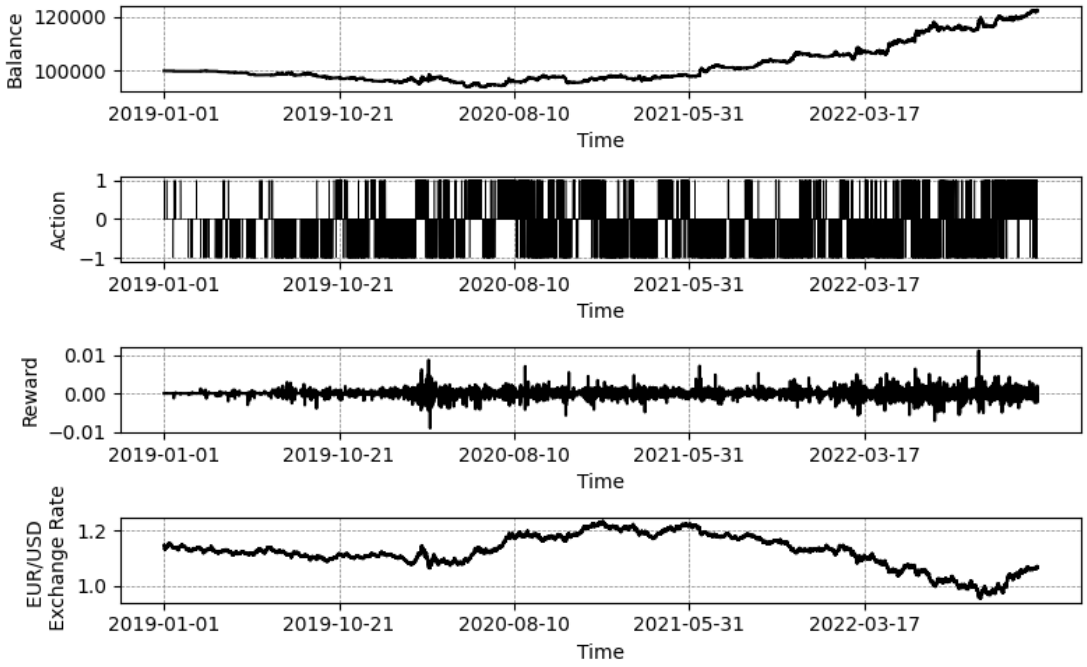

Figure 3.10: Best combination of parameters for the DQN agent with the Log Return reward function.

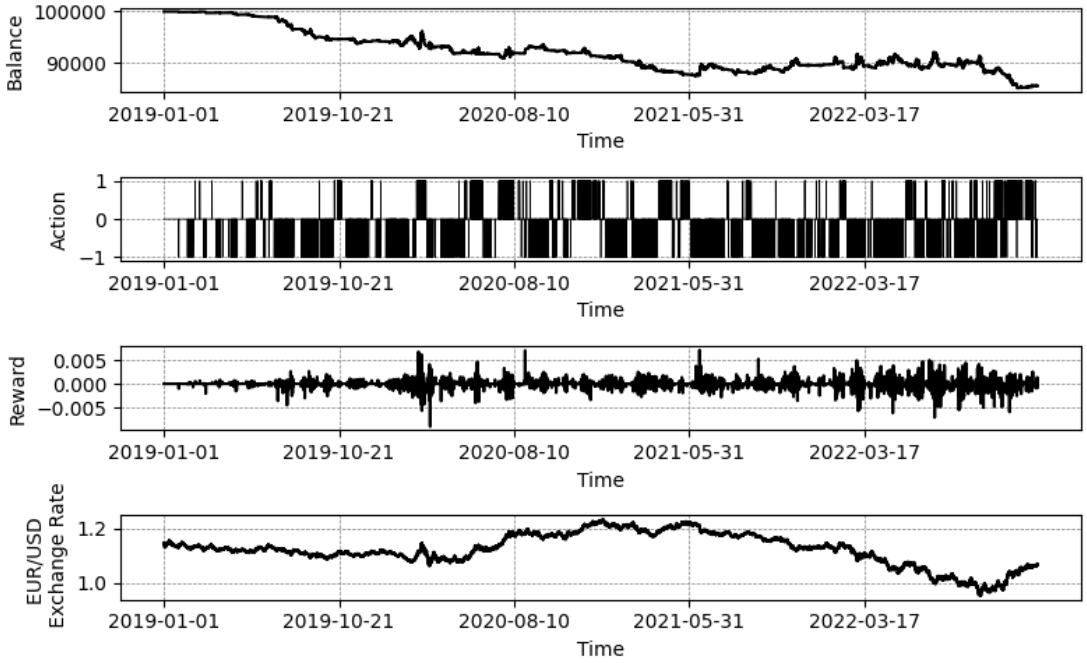

Figure 3.11: Worst combination of parameters for the DQN agent with the Log Return reward function.

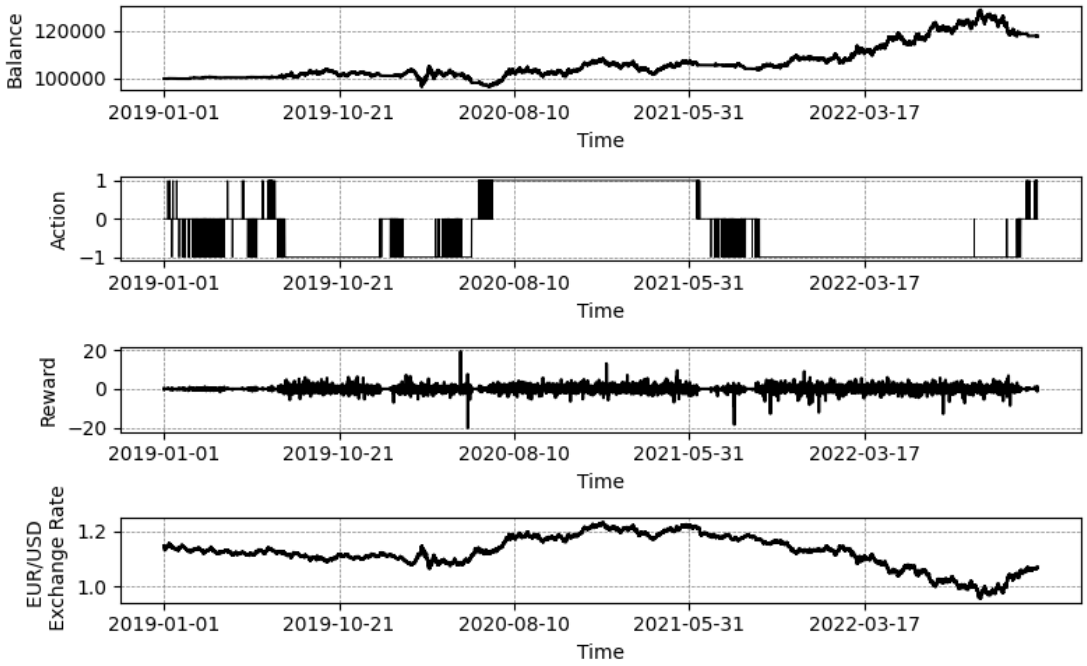

Figure 3.12: Best combination of parameters for the DQN agent with the Sharpe Ratio reward function.

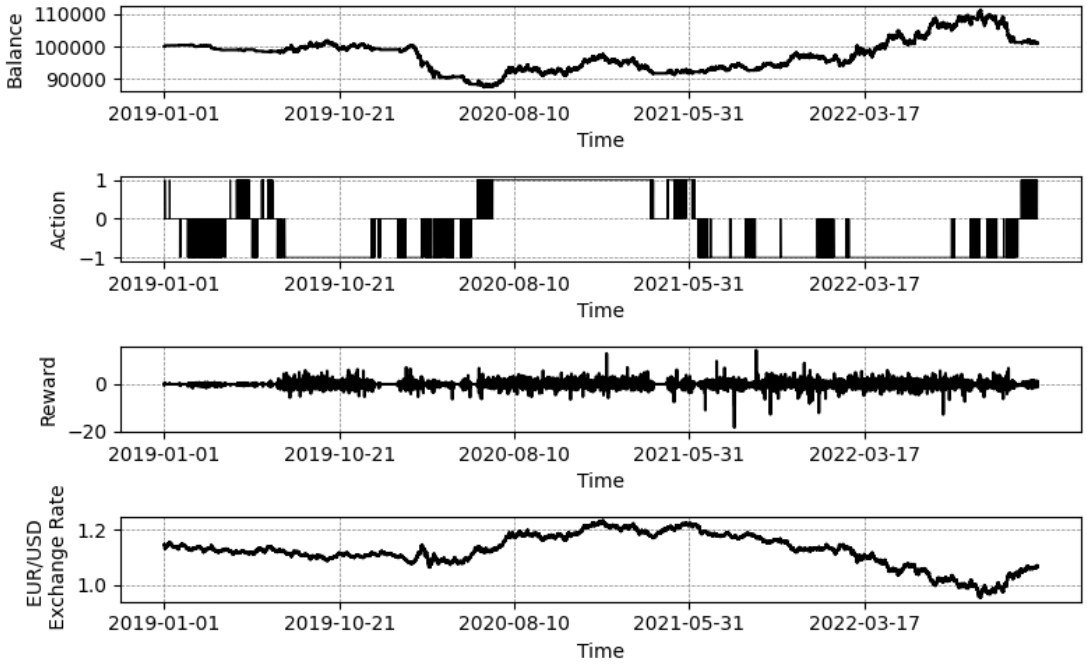

Figure 3.13: Worst combination of parameters for the DQN agent with the Sharpe Ratio reward function.

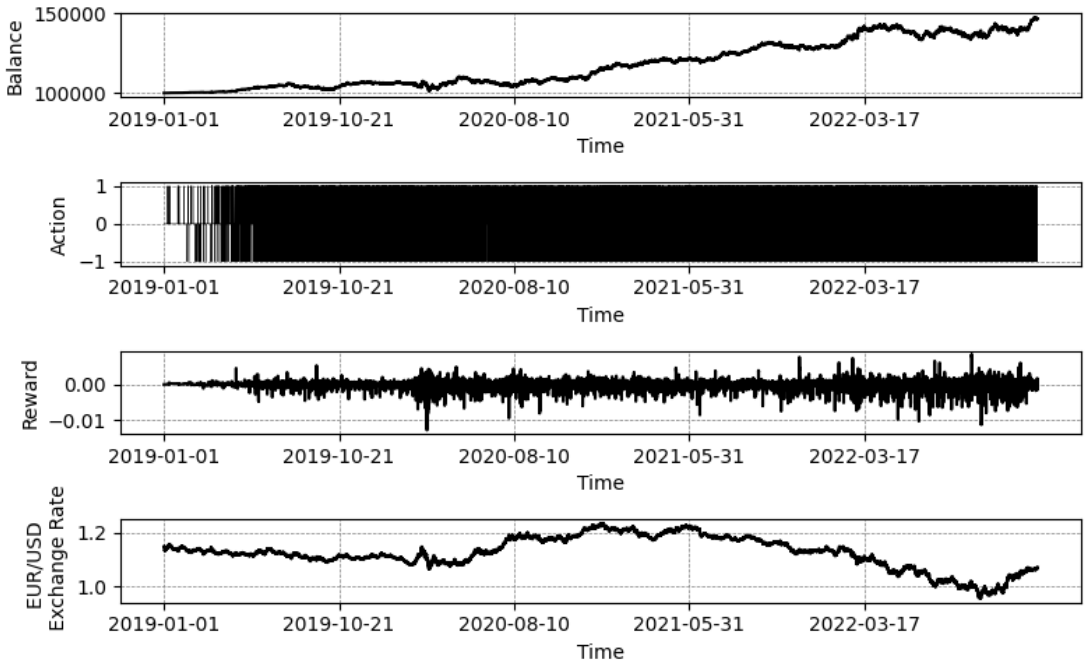

Figure 3.14: Best combination of parameters for the DQN agent with the Modified Log Return reward function.

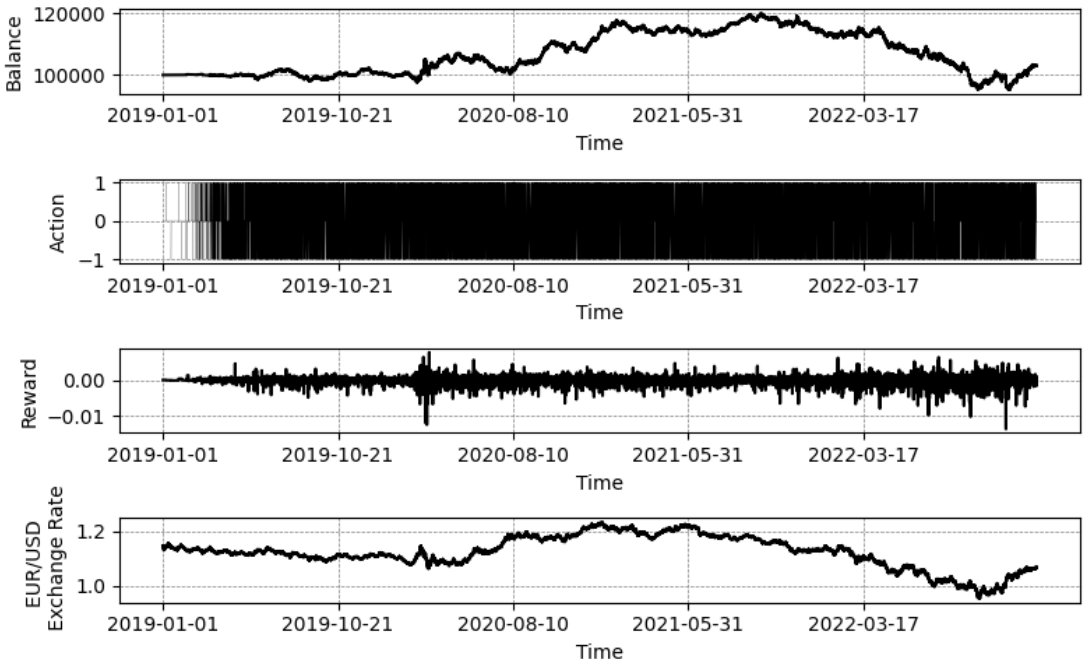

Figure 3.15: Worst combination of parameters for the DQN agent with the Modified Log Return reward function.

3.6.2 DDQN Results

When comparing the results of the Double Deep Q-Network (DDQN) with those of the standard DQN, it becomes evident that the improved agent achieved performance levels that were very similar to those of the basic agent.

REWARD: LOG RETURN, AGENT: DDQN									
n	Hidden Layers	Annualized Return (%)	$> v_0$ (%)	Maximum Drawdown (%)	Annual Trades	On Position (%)	Long Trades (%)	Gross Win Ratio (%)	Net Win Ratio (%)
RESULTS OF 50 SIMULATIONS (95% BOOTSTRAP CONFIDENCE INTERVAL)									
1	1	$[-7.09, -5.4]$	$[0.96, 2.9]$	$[28.69, 33.68]$	$[4023, 4115]$	$[66.53, 67.53]$	$[45.63, 46.23]$	$[49.57, 49.85]$	$[47.37, 47.64]$
4	1	$[-8.48, -6.83]$	$[0.68, 3.08]$	$[32.15, 37.06]$	$[4002, 4074]$	$[65.14, 66.3]$	$[45.62, 46.21]$	$[49.38, 49.64]$	$[47.16, 47.42]$
8	1	$[-8.76, -7.57]$	$[0.7, 1.47]$	$[34.01, 38.21]$	$[3971, 4033]$	$[65.28, 66.29]$	$[45.91, 46.48]$	$[49.56, 49.8]$	$[47.31, 47.55]$
16	1	$[-9.55, -8.42]$	$[0.58, 1.32]$	$[36.96, 40.98]$	$[3935, 3996]$	$[64.73, 65.7]$	$[45.65, 46.23]$	$[49.64, 49.84]$	$[47.35, 47.54]$
1	2	$[-7.54, -6.59]$	$[0.84, 2.13]$	$[30.47, 33.77]$	$[3987, 4058]$	$[67.16, 68.02]$	$[45.71, 46.26]$	$[49.59, 49.83]$	$[47.37, 47.61]$
4	2	$[-9.33, -8.11]$	$[0.79, 1.76]$	$[36.09, 39.86]$	$[3917, 4004]$	$[65.99, 66.92]$	$[45.8, 46.48]$	$[49.7, 49.9]$	$[47.46, 47.64]$
8	2	$[-9.41, -8.37]$	$[0.91, 1.75]$	$[37.46, 40.81]$	$[3930, 4007]$	$[65.6, 66.55]$	$[45.6, 46.26]$	$[49.6, 49.79]$	$[47.35, 47.55]$
16	2	$[-10.23, -9.06]$	$[1.04, 2.32]$	$[39.26, 43.16]$	$[3855, 3926]$	$[65.42, 66.25]$	$[45.5, 46.07]$	$[49.61, 49.78]$	$[47.35, 47.52]$
RESULTS OF THE OPERATIVE SIGNAL									
1	1	1.35	22.61	7.98	1683	27.74	30.05	47.21	45.1
4	1	-0.22	20.3	7.09	1337	24.17	27.9	48.66	46.74
8	1	0.72	14.23	10.56	1136	23.12	27.06	48.95	46.71
16	1	-1.81	3.86	11.65	1022	21.63	23.7	49.34	47.31
1	2	0.03	3.07	8.93	1401	25.82	28.5	47.81	45.98
4	2	0.22	6.18	8.37	1040	22.34	47.91	47.7	45.95
8	2	-2.46	4.72	13.39	976	21.54	24.9	49.06	47.12
16	2	-2.93	3.13	12.71	912	22.23	22.55	48.42	46.4

Table 3.8: Outcomes of the simulations conducted employing the DDQN agent with the Log Return reward function for each combination of parameters. Both the 95% bootstrapped confidence intervals and the operative signal metrics are shown.

Starting with the comparison between the DDQN employing the Log Return reward function and its DQN counterpart, it's worth noting that, while in DDQN there were more instances obtaining positive annual returns, they were predominantly close to zero. The only exception was the parameter combination of $n = 1$ and $h = 1$, which achieved a 1.35% annual return, an outcome that can be considered suboptimal.

Furthermore, the two best parameter combinations, which yielded annual returns of 2.05% and 5.53% with the DQN agent, experienced a significant decline when applied to the DDQN, resulting in annual returns of 1.35% and 0.03%, respectively.

In all other metrics related to the operational behavior of the agent, the DDQN remained consistent with the performance of the DQN.

REWARD: SHARPE RATIO, AGENT: DDQN									
n	Hidden Layers	Annualized Return (%)	$> v_0$ (%)	Maximum Drawdown (%)	Annual Trades	On Position (%)	Long Trades (%)	Gross Win Ratio (%)	Net Win Ratio (%)
RESULTS OF 50 SIMULATIONS (95% BOOTSTRAP CONFIDENCE INTERVAL)									
1	1	$[-2.93, -1.84]$	$[3.98, 11.6]$	$[17.65, 20.29]$	$[1641, 1659]$	$[87.13, 89.1]$	$[46.21, 47.07]$	$[49.76, 50.04]$	$[47.62, 47.89]$
4	1	$[-3.1, -1.87]$	$[5.61, 14.73]$	$[16.96, 19.36]$	$[1653, 1674]$	$[88.01, 90.32]$	$[45.95, 46.96]$	$[49.65, 49.97]$	$[47.51, 47.82]$
8	1	$[-2.93, -1.57]$	$[5.47, 15.26]$	$[17.19, 19.91]$	$[1650, 1671]$	$[87.7, 89.54]$	$[46.52, 47.56]$	$[49.81, 50.05]$	$[47.68, 47.91]$
16	1	$[-2.82, -1.34]$	$[5.45, 11.29]$	$[17.0, 20.04]$	$[1658, 1679]$	$[88.01, 89.78]$	$[46.25, 47.36]$	$[49.61, 49.87]$	$[47.49, 47.73]$
1	2	$[-3.62, -2.39]$	$[3.35, 9.72]$	$[19.06, 22.01]$	$[1630, 1664]$	$[81.9, 84.97]$	$[46.75, 47.81]$	$[49.64, 49.92]$	$[47.48, 47.76]$
4	2	$[-3.82, -2.19]$	$[4.08, 11.88]$	$[19.26, 22.57]$	$[1661, 1687]$	$[85.28, 87.57]$	$[46.6, 47.73]$	$[49.76, 49.98]$	$[47.63, 47.87]$
8	2	$[-4.05, -2.59]$	$[3.61, 9.69]$	$[19.76, 22.87]$	$[1652, 1687]$	$[83.25, 86.58]$	$[46.62, 47.69]$	$[49.65, 49.9]$	$[47.53, 47.8]$
16	2	$[-3.71, -2.48]$	$[4.48, 13.85]$	$[18.86, 21.4]$	$[1661, 1700]$	$[84.85, 87.12]$	$[46.66, 47.65]$	$[49.79, 50.08]$	$[47.65, 47.92]$
RESULTS OF THE OPERATIVE SIGNAL									
1	1	4.19	57.7	9.17	229	78.32	18.82	50.88	48.25
4	1	3.58	66.1	10.81	234	78.89	11.75	54.59	51.82
8	1	4.32	73.51	9.91	223	76.92	27.38	47.03	44.78
16	1	3.89	61.32	8.68	220	77.52	20.32	50.62	48.24
1	2	1.84	41.57	10.09	379	71.9	30.23	52.01	49.97
4	2	1.44	40.99	12.59	298	73.16	21.14	51.85	49.66
8	2	2.9	68.89	11.56	267	73.73	29.68	50.0	48.41
16	2	4.45	73.28	9.9	371	73.08	22.2	50.4	48.11

Table 3.9: Outcomes of the simulations conducted employing the DDQN agent with the Sharpe Ratio reward function for each combination of parameters. Both the 95% bootstrapped confidence intervals and the operative signal metrics are shown.

In the scenario where the Sharpe Ratio reward function was employed, the DDQN demonstrated improvements in annual returns compared to the standard DQN, but this was notably evident only when two hidden layers ($h = 2$) were used. Conversely, when employing a single hidden layer, most of the annual returns saw slight deterioration. Other performance metrics of the DDQN remained consistent with those of the DQN simulations.

An interesting observation is that the combination of parameters $n = 4$ and $h = 1$ exhibited the highest gross win ratio among all the simulations, reaching 54.59%. However, this noteworthy win ratio did not translate into significantly impressive annual returns, as this particular parameter combination achieved a 3.58% annual return, which can be considered good but not exceptionally so. This

example underscores that while a high win ratio is favorable, it is not necessarily a guarantee of superior results. Indeed, parameter combinations with a lower win ratio percentage managed to attain significantly better annual returns.

In the final comparison between the DDQN agent using the Modified Log Return and its basic counterpart, the DQN, the results do not exhibit significant improvements. In this scenario, there are certain parameter combinations that show improvement when employing the DDQN, particularly when the state is represented by $n = 16$. However, there are also cases where the performance deteriorated when compared to the DQN.

Overall, the metrics describing the operational behavior of the DDQN and the DQN remain relatively consistent with each other, indicating that the DDQN does not consistently outperform the DQN when using the Modified Log Return as the reward function.

REWARD: MODIFIED LOG RETURN, AGENT: DDQN									
n	Hidden Layers	Annualized Return (%)	$> v_0$ (%)	Maximum Drawdown (%)	Annual Trades	On Position (%)	Long Trades (%)	Gross Win Ratio (%)	Net Win Ratio (%)
RESULTS OF 50 SIMULATIONS (95% BOOTSTRAP CONFIDENCE INTERVAL)									
1	1	$[-2.68, -1.1]$	$[14.68, 26.73]$	$[19.44, 22.38]$	$[4880,\ 4948]$	$[87.17, 87.82]$	$[51.35, 51.94]$	$[50.26, 50.58]$	$[48.09, 48.39]$
4	1	$[-2.61, -1.25]$	$[4.95, 13.94]$	$[18.47, 21.25]$	$[3966,\ 4000]$	$[88.95, 89.4]$	$[50.58, 50.87]$	$[50.05, 50.24]$	$[47.96, 48.13]$
8	1	$[0.91, 2.16]$	$[15.18, 24.78]$	$[16.2, 18.32]$	$[3671,\ 3714]$	$[89.3, 89.81]$	$[50.48, 50.7]$	$[49.87, 50.06]$	$[47.74, 47.95]$
16	1	$[0.78, 2.57]$	$[18.49, 31.93]$	$[16.3, 18.46]$	$[3529,\ 3564]$	$[89.77, 90.35]$	$[50.45, 50.67]$	$[49.93, 50.11]$	$[47.77, 47.94]$
1	2	$[-3.68, -2.4]$	$[13.01, 25.57]$	$[21.47, 24.56]$	$[4969, 5058]$	$[87.64, 88.16]$	$[51.27, 51.82]$	$[50.31, 50.55]$	$[48.1, 48.34]$
4	2	$[-3.23, -1.66]$	$[3.84, 12.9]$	$[19.92, 23.34]$	$[4029, 4073]$	$[88.27, 89.02]$	$[50.52, 50.81]$	$[50.01, 50.19]$	$[47.84, 48.02]$
8	2	$[-0.03, 1.41]$	$[10.21, 19.8]$	$[17.17, 19.58]$	$[3772, 3813]$	$[88.8, 89.52]$	$[50.32, 50.62]$	$[49.81, 50.02]$	$[47.64, 47.84]$
16	2	$[-0.3, 1.24]$	$[11.44, 21.71]$	$[17.01, 19.45]$	$[3636, 3673]$	$[89.65, 90.32]$	$[50.43, 50.65]$	$[49.93, 50.11]$	$[47.71, 47.89]$
RESULTS OF THE OPERATIVE SIGNAL									
1	1	3.36	93.02	17.22	4160	69.61	54.24	51.58	49.55
4	1	3.83	87.51	11.95	2704	70.71	52.43	51.68	50.06
8	1	11.38	92.26	7.9	2271	71.06	52.55	51.01	49.44
16	1	10.13	93.77	13.78	1973	70.92	52.68	51.64	50.16
1	2	2.03	86.61	19.52	4135	67.93	54.51	51.77	49.62
4	2	3.56	68.04	10.86	2650	69.03	52.18	51.07	49.4
8	2	10.49	89.45	6.77	2261	69.55	51.66	50.28	48.5
16	2	10.66	84.39	11.27	1984	70.21	52.67	51.55	49.82

Table 3.10: Outcomes of the simulations conducted employing the DDQN agent with the Modified Log Return reward function for each combination of parameters. Both the 95% bootstrapped confidence intervals and the operative signal metrics are shown.

3.7 Future Work

The aforementioned results of the model can be considered satisfactory, demonstrating the potential of the RL framework. It's worth noting that the agent was able to profit in the Forex market using only raw OHLC candle data as inputs. It should be emphasized that the Forex market is influenced by a wide range of external factors, including interest rates, inflation levels, import-export activities, international politics, and more. It's perhaps the most influenced market globally, presenting a significant challenge that the agent successfully overcame. However, the model is far from perfect, and there are several aspects that can be improved upon.

Starting with the action space, the implemented model employed a discrete action space with only three options available for the agent: buy, sell, or stay out of the market. In reality, traders often manage their capital based on perceived risks. Two potential solutions to this problem include expanding the action space while keeping it discrete (e.g., $A = \{-1, -0.5, 0, 0.5, 1\}$) or employing a continuous action space, although this would require modifying the DQN agent, as it's not designed for continuous action spaces.

Regarding the environment space, an improvement could involve incorporating technical analysis indicators into the state signal to provide the agent with a more comprehensive view of its environment.

Another enhancement could be the introduction of a risk management technique, ideally within the reward function. This would enable the agent to learn risk management autonomously, without external intervention.

Concerning the agent itself, one possible improvement is replacing the feedforward neural network with a recurrent neural network, which is better suited to consider the temporal order of features.

Furthermore, starting the agent's learning process before the actual testing period could be beneficial, allowing the agent to familiarize itself with the environment prior to trading.

An additional advancement involves a shift in the automated trading system's approach, transitioning from a completely online learning agent to a supervised

learning agent with a sliding training-testing set. While this may seem counterintuitive for the RL setting, it offers advantages such as the ability to evaluate model parameters and select the best-performing ones, as well as reducing the occurrence of random actions, which can be detrimental in financial trading.

Lastly, testing the agent over a longer testing period exceeding four years could provide stronger validation of its capabilities.

Conclusion

The rise of artificial intelligence has disrupted various industries, including the financial sector. This research aimed to leverage advancements in AI by developing an automated trading system and testing its performance in the Forex market. The machine learning algorithm employed in this study falls under the subfield of Reinforcement Learning. The primary objective of this research was to evaluate the Deep-Q-Network (DQN) agent and its enhancement, the Double-Deep-Q-Network (DDQN), within a four-year timeframe, using 30-minute interval EUR/USD OHLC data. Each agent underwent testing with various parameter combinations to assess their impact on performance. The findings were highly promising, particularly when implementing the Modified Log Return reward function, an innovative reward signal that takes into consideration the last closed trade performance. In fact, the DQN agent, when combined with this reward function, achieved positive annual returns in all eight parameter combinations. More notably, six out of eight combinations exceeded a 4% annual return threshold, which served as the benchmark. The most noteworthy achievement was an annual return of 11.69%, which is quite remarkable. Moreover, this result was obtained with the balance being higher than the initial balance in 99.58% of instances, while the maximum drawdown was limited to 7.01%. Surprisingly, the addition of a second hidden layer in the neural network did not enhance model performance. Furthermore, even the DDQN did not demonstrate improved results compared to the basic DQN. Another intriguing observation is that the agent using the Sharpe Ratio performed better when subjected to a commission setting, as opposed to when it was tested without considering the bid-ask spread commission.

Bibliography

[1] João Carapuço, Rui Neves, and Nuno Horta. "Reinforcement learning applied to Forex trading". In: *Applied Soft Computing* 73 (2018), pp. 783–794.

[2] Jerry Coakley, Michele Marzano, and John Nankervis. "How profitable are FX technical trading rules?" In: *International Review of Financial Analysis* 45 (2016), pp. 273–282.

[3] Marco Corazza and Andrea Sangalli. "Q-Learning and SARSA: a comparison between two intelligent stochastic control approaches for financial trading". In: *University Ca'Foscari of Venice, Dept. of Economics Research Paper Series No* 15 (2015).

[4] George Cybenko. "Approximation by superpositions of a sigmoidal function". In: *Mathematics of control, signals and systems* 2.4 (1989), pp. 303–314.

[5] Michael AH Dempster and Vasco Leemans. "An automated FX trading system using adaptive reinforcement learning". In: *Expert systems with applications* 30.3 (2006), pp. 543–552.

[6] Hado Hasselt. "Double Q-learning". In: *Advances in neural information processing systems* 23 (2010).

[7] Po-Hsuan Hsu, Mark P Taylor, and Zigan Wang. "Technical trading: Is it still beating the foreign exchange market?" In: *Journal of International Economics* 102 (2016), pp. 188–208.

[8] Chien Yi Huang. "Financial trading as a game: A deep reinforcement learning approach". In: *arXiv preprint arXiv:1807.02787* (2018).

[9] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". In: *nature* 518.7540 (2015), pp. 529–533.

[10] Murat Ozturk, Ismail Hakki Toroslu, and Guven Fidan. "Heuristic based trading system on Forex data using technical indicator rules". In: *Applied Soft Computing* 43 (2016), pp. 170–186.

[11] Melrose Roderick, James MacGlashan, and Stefanie Tellex. "Implementing the deep q-network". In: *arXiv preprint arXiv:1711.07478* (2017).

[12] Tom Schaul et al. "Prioritized experience replay". In: (2015).

[13] Ali Shavandi and Majid Khedmati. "A multi-agent deep reinforcement learning framework for algorithmic trading in financial markets". In: *Expert Systems with Applications* 208 (2022), p. 118124.

[14] Avraam Tsantekidis et al. "Price trailing for financial trading using deep reinforcement learning". In: *IEEE Transactions on neural networks and learning systems* 32.7 (2020), pp. 2837–2846.

[15] Viliam Vajda. "Could a trader using only "old" technical indicator be successful at the Forex market?" In: *Procedia Economics and Finance* 15 (2014), pp. 318–325.

[16] Hado Van Hasselt et al. "Deep reinforcement learning and the deadly triad". In: *arXiv preprint arXiv:1812.02648* (2018).

[17] Christopher JCH Watkins and Peter Dayan. "Q-learning". In: *Machine learning* 8 (1992), pp. 279–292.

[18] Nima Zarrabi, Stuart Snaith, and Jerry Coakley. "FX technical trading rules can be profitable sometimes!" In: *International Review of Financial Analysis* 49 (2017), pp. 113–127.

Sitography

[19] BabyPips. *The Relationship Between Margin and Leverage*. URL: https://www.babypips.com/learn/forex/margin-vs-leverage.

[20] Cameron Buckner and James Garson. *Connectionism*. URL: https://plato.stanford.edu/entries/connectionism/.

[21] Akshay L Chandra. *McCulloch-Pitts Neuron — Mankind's First Mathematical Model Of A Biological Neuron*. URL: https://towardsdatascience.com/mcculloch-pitts-model-5fdf65ac5dd1.

[22] Lucas Downey. *Efficient Market Hypothesis (EMH): Definition and Critique*. URL: https://www.investopedia.com/terms/e/efficientmarkethypothesis.asp.

[23] Huzefa Hamid. *The History of Forex Trading*. URL: https://www.dailyforex.com/forex-articles/2008/04/the-history-of-forex-trading/66.

[24] Daniele Liberto. *Adaptive Market Hypothesis (AMH): Overview, Examples, Criticisms*. URL: https://www.investopedia.com/terms/a/adaptive-market-hypothesis.asp.

[25] Roy Rupali. *AI, ML, and DL: How not to get them mixed!* URL: https://towardsdatascience.com/understanding-the-difference-between-ai-ml-and-dl-cceb63252a6c.

[26] Dukascopy Bank SA. *Average Spreads*. URL: https://www.dukascopy.com/swiss/english/marketwatch/average-spreads/.

[27] Triennial Central Bank Survey. *OTC foreign exchange turnover in April 2022*. URL: https://www.bis.org/statistics/rpfx22_fx.htm.

[28] Wikipedia. *AI winter*. URL: https://en.wikipedia.org/wiki/AI_winter#:~:text=In%20the%20history%20of%20artificial,idea%20of%20a%20nuclear%20winter..