UiO : Department of Mathematics
University of Oslo

Trading Financial Markets Using Reinforcement Learning

Application and Analysis

Fábio Rodrigues Pereira

Master's Thesis, Spring 2022

Abstract

Stock market forecasting has long piqued the curiosity of academics and professionals. However, because of the markets' chaotic dynamics, increased volatility, unstable liquidity, and periodic flash crashes, conducting this kind of investigation is challenging while implementing algorithmic trading. To address these issues, the current research presents a feedback–learning tool based on reinforcement learning methods, namely function approximation reinforcement learning: SARSA, Q–Learning, and Greedy–GQ. We examined the reinforcement learning theory, from the fundamental to the most sophisticated, engineered the control agents from scratch, and ultimately validated statistically reliable analyzed findings. Numerous scenarios were provided in which reinforcement learning agents acted in the market while trading a future contract of the São Paulo stock exchange in Brazil (B3's mini–index). Our best panels demonstrated solid final cumulative earnings of over 150% in a period of one year.

Acknowledgements

This thesis would not have been completed without the support of several people:

- Many thanks to Prof. Dr. Salvador Ortiz–Latorre, who reviewed my multiple drafts and acted as an outstanding mentor, and thesis supervisor, giving guidance while perfectly balancing wisdom and humor.
- I am grateful to the University of Oslo for giving me the extraordinary opportunity to study in a magnificent environment filled with incredibly intelligent and delightful individuals.
- An even special and affectionate word of thanks to my family that deserves an infinite amount of appreciation and love:
    - ☐ To my Grandpa Osvaldo Rodrigues, who unfortunately passed away during this COVID-19 pandemic. I wanted to express my profound gratitude to him for his lessons in life and for assuring me, in a night's dream after his death, that my decisions and life are on the right track. This affirmation comforted me and boosted my motivation to complete this thesis.
    - ☐ To my Granny Benedita de Oliveira Rodrigues, who, along with my parents and Grandpa, raised me with her love and care, and still teaches me life lessons. Without her significant contributions, I would not be where I am today.
    - ☐ To my Daddy Joaquim Laercio Pereira and Mommy Rita de Cássia Rodrigues Pereira, who have both committed their lives to supporting and providing for me and my siblings. I would not have overcome my greatest challenges in life without their love, sacrifices, and hard work.
    - ☐ To my siblings Patricia Rodrigues Pereira and Vitor Rodrigues Pereira for loving and supporting me and always giving me great advices.
    - ☐ To my love, Aurora Villemo Krogstad, for always being there for me, cheering me up on through tough times, and celebrating in my accomplishments. "*O seu sorriso vale mais que um diamante. Se você vier comigo, aí nóis vamo adiante. Com a cabeça erguida e mantendo a fé em Deus. O seu dia mais feliz vai ser o mesmo que o meu*" (Dias de luta, dias de glória – Charlie Brown Jr. – 2006)

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

We launch our research by outlining the current state of artificial intelligence (AI) in the world, followed by an examination of the distinctions among AI's sub–levels. While determining reinforcement learning (RL) as our AI central topic, we bring in its simplest structure, how it differed from other systems, and why it is rarely employed in the financial industry, in spite of its unquestionable promises of earnings. Moreover, a compelling theoretical economics controversy between efficiency and rationality on the one hand and inefficiency and irrationality on the other, could not be ignored. Two key hypotheses are evaluated for this purpose: the efficient market hypothesis and the adaptive markets hypothesis. At the end, we briefly explain why RL may carry out the same, if not better, degree of achievement than conventional stochastic optimal controls. In addition, we also hand out the research methodologies, historical context, and thesis structure.

1.1 Motivation and justification

In 2022, we stand facing the transformation of an emphatic world, giving rise to a fundamentally new human way of life, working, and interacting with one another, our planet, and the universe. The World Economic Forum refers to this new cycle as the fourth industrial revolution (4IR), a phase that has never been experienced before in human history. In a nutshell, the first industrial revolution (1760–1840) mechanized the generation of water and steam power. The second (1870–1914) saw the mass generation of electric power. The third period (the 1950s) witnessed the introduction of automated production via breakthroughs in computers and electronics. The fourth (2016–now) can be characterized by a series of rapidly evolving and well–publicized developments in a variety of subsets of technological domains, including nanotechnology, biotechnology, advanced digital production technologies, human–machine interfaces, and artificial intelligence. With regard to these achievements in the 4IR, AI has created machines that learn unknown complex intellectual tasks in real–time and respond faster and more efficiently than humans. Earlier, these duties were exclusively performed by extremely skilled humans after years of training, experience, and practice. These valuable learning and responding machine systems are already widely applied in uncountable different expertise, and, as we will see, they could not be missing in the field of finance.

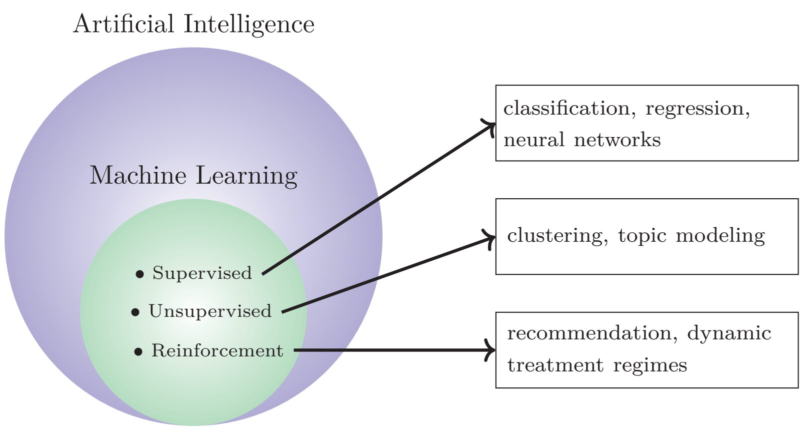
Artificial Intelligence
Machine Learning
• Supervised
• Unsupervised
• Reinforcement
classification, regression, neural networks
clustering, topic modeling
recommendation, dynamic treatment regimes

Figure 1.1: AI and ML diagrams

AI problems are commonly separated into supervised, unsupervised, and RL. The first set of problems includes classification, regression, and neural networks, while the second encloses compression, clustering, and topic modeling. On the one hand, the supervised models learn from a training set with labeled and correct outcomes. On the other hand, the unsupervised models typically infer from hidden structures discovered from within collected data. RL is different from supervised or unsupervised machine learning (ML) techniques because it neither receives any knowledgeable external information nor tries to find hidden structures; instead, it only maximizes a reward measure. Essentially, it employs a trial–and–error cumulative reward method to optimize the expectation of a utility function over time. This is known as a sequential decision problem where the agent must learn an optimal behavior in an unknown environment through reinforcements (feedback). Fig. 1.2 illustrates a simple structure for an RL system.

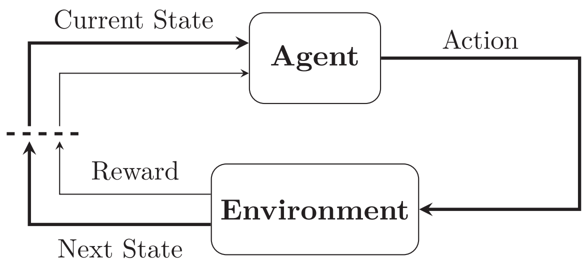

Figure 1.2: RL's basic framework

The motivation for this master thesis resides precisely in the point that RL methods remain largely unexplored by the financial trading sector. Increased volatility, unstable liquidity, and periodic flash crashes are the most prevalent challenges encountered while implementing algorithmic trading. Moreover, the financial markets are dynamic and turbulent structures that are too complex for straightforward algorithms, demanding the expertise of highly skilled ML scientists to make their applicability an investment choice. Indeed, complex quantitative strategies and technologically enhanced participants result in short–lived methods with harder ways of identifying patterns. However, recent developments and advances in AI are changing the industry and contributing to being more explored by the significant investment industry. These new techniques are receiving more attention to their potential, which promises to overcome other investment theories already settled in the industry for many decades.

Automated strategies must be flexible, not wholly reliant on pattern–based strategy, and capable of learning and adjusting on the go–just like humans, but faster. RL covers these concepts with a high precision level, which is why this thesis proposes a in–depth study on its most advanced methodologies. In our opinion, RL algorithms are also superior to any other ML system because they are quicker in recognizing an old and no longer predictable pattern (no longer lucrative to trade), as well as faster in identifying new recurring patterns that offer potential trading profits. Consequently, RL is preferable to other ML subcategories for making real–time decisions based on current market circumstances and the immediate consequences of their actions.

As a natural byproduct, this dissertation proposes an RF trading application in financial markets using as a source of information nearly solely the past prices and/or returns of assets. Note that the term "information" or "decision–making information" may refer to any measurable knowledge that an individual might retrieve from the market to substantiate their decision–making process. However, since we design applications that make use of an asset's historical prices and/or returns, we often present information as those quantifiable values.

Why does this dissertation intend to develop trading strategies based on past prices or returns as information?

We do not want to get enmeshed in the finer points of this question's topic and the theories that surround it, as a whole thesis may be devoted to this issue. Nonetheless, a simpler response requires basic consideration of the theoretical nature of financial markets. According to the efficient market hypothesis (EMH) [CN05, chap. 3], when the market's decision information consists only of past prices or returns (i.e., its weak form), systematic profits from trading are unattainable. This concept is because the market prices follow the fair value of its risky assets, also called the discounted present value (DPV), in an efficient manner. In a technical probability theory way, the market is a martingale stochastic process where the difference between the expected future return given an information measure (random variable), and the current return (random variable) is equal to zero:

$$\mathbb{E}\Big[R_{t+1}|S_t\Big] - R_t = 0.$$

From the perspective of microeconomics theory, a capital market is considered efficient when its prices or returns are determined as the outcome of supply and demand in a competitive manner and by rational players. These players rapidly process relevant pieces of information into the asset's prices accordingly (efficiency and rationality). In such a world, there should be no opportunities for profiting since new information is unforecastable, considering which price changes should be unforecastable too, i.e., the expected future return is zero given currently available data (fair game or martingale). Conversely, common sense dictates that neither the market's information is entirely unforecastable, nor is the economic agents' behavior often wholly rational when making decisions under uncertainty; moreover, the market process is also not a flawless martingale. The result lies in a frequently inefficient financial market that opens possibilities for profitable tradings (inefficiency and irrationality).

This debate between efficiency and rationality on one side and inefficiency and irrationality on the other is one of the most contentious in our field, and one that we can not ignore. Illustrious professor Andrew W. Lo [Lo17] brilliantly solved this issue using his theory of adaptative markets hypothesis (AMH), which shows the psychological incompleteness of the market's efficiency by the coexistence of rationality and irrationality notions. At some point, when the markets become unstable, economic agents often tend to make decisions instinctively, producing exploitable inefficiencies. According to common sense and the AMH theory, it seems reasonable that the financial market might be inefficient, thus opening opportunities for profiting in algorithm tradings.

How to benefit from market inefficiency circumstances?

The solution depends on the theoretical approach adopted to face those opportunities. As justified before, this master thesis considers that financial trading systems (FTSs) have to learn and adapt from a time–varying environment in a real–time manner. We believe that this kind of framework is able to detect and adjust profitable trading policies on the fly, without human intervention. Throughout this dissertation, we will look at many RL approaches that may be applicable in these situations.

Why do we not employ primary traditional and already established stochastic dynamic programming (DP) systems that guarantee optimal solutions instead of RL?

A last conundrum that someone reading this dissertation may consider is the existence of regular stochastic optimum controls, for instance, DP, that, in principle, may more efficiently find optimal financial trading strategies than RL or other tools. In general, DP requires a tedious process that includes complex mathematics to obtain the distribution of the transition probabilities and rewards. However, RL does not demand these conditions, and can produce nearly and exact optimal solutions despite this [Gos15, p. 198]. Additionally, due to the extraordinarily large dimensionality of an FTS issue, calculations are often infeasible. This is why RL is such a lovely method: it can handle a Markov decision process (MDP) without developing a theoretical model (transition probabilities and rewards), and without succumbing to the curse of costly computations owing to their dimensionality.

1.2 Research design and central issue

Professionals and quantitative researchers nowadays depend on a variety of conventional and modern computational methodologies to make fast and accurate decisions. Thus, in light of recent breakthroughs in big data analytics, the problem is to identify a reliable tool capable of understanding enormous volumes of unstructured data accessible on the market. Without a doubt, the markets' dynamism, complexity, evolutionary, and chaotic nature make this task unmanageable and sometimes ineffective. To that end, this thesis will examine mechanisms for generalizing knowledge gleaned from interactions with the financial market environment using computer learning processes. As previously said, there are various promising developers of AI techniques that actually prove their capacity to address complicated sequential decision–making dilemmas in a variety of domains, which should pose no difficulty when applied to the financial forecasting environment.

This master's thesis will examine an efficient and possibly profitable application of AI in financial markets, an area of computer science that is relatively underdeveloped. We will examine the theory and use of many of the most powerful, automated, and contemporary self–adaptive ML, known as RL. We focus on intelligent on–policy and off–policy control solutions based on function approximation processes from the family of RL methodologies. These titles may be troublesome to comprehend at the outset of this work, but we will gradually gain the essential knowledge throughout the theory chapters, and by the final analysis, the reader will be able to reconcile the theory with the final application case, as well as assess its profitability and reliability.

This quantitative research is divided into three significant parts: theory, application, and analysis. The first part comprehends the theory's evolution from the primary forms of the decision–process, and its evolutions during the decades, arriving at the most advanced form today applied in the field of computer science and finance. The second part applies the most advanced theory in order to solve a practical algorithm trading problem. Finally, the last part analyses the results of the application case, and its statistics and risks, and answers the most important research question of this study:

Can we have consistent profits using RL in the financial market?

The main goal of this dissertation is to demonstrate that RL can be a reliable and high–level investment tool like any other already established alternative in the financial domain, such as stochastic process, fundamental or technical analysis, econometrics, etc. Likewise, this research aims to propose a promising application of its optimized model to allow experts to enhance returns, reduce risk, and increase efficiency by systematically incorporating RL methods for algorithm trading decision–making.

1.3 Research methodology

Prior to starting the study, certain key guidelines of our research methodology should be considered. We conducted this study to assess the AI practice in FTSs. We set up the applied design as the research methodology of this thesis to meet the central research question. This option was made since our ambition was to build a consistent AI technique that would contribute to the financial trading algorithm industry, and the emerging model for this demand may serve as a credible demonstration of this. Simultaneously, we likewise took advantage of the quantitative research type owing to the nature of the investigation's analysis, which relies on measurements and statistics.

The programming language chosen for this work was Python 3.9, which includes the scientific programs NumPy and Pandas, the visualization packages Matplotlib and Seaborn, the ML library Pytorch, as well as the datetime and tqdm supplementary packages. We rigorously adhered to the constraints mandated by Python's best practices for code organization, referred to as Python Enhancement Proposals 8 (PEP8). PEP8 establishes style guidelines for Python users in order to facilitate code comprehension. Yet, everything was rewritten from the ground up, incorporating as many vectorized structures and operations, and high performance guidelines as possible. We cannot forget to mention the widespread use of abstraction, inheritance, encapsulation, and polymorphism in Object–Oriented Programming (OOP), especially when we engineered the RL environment as well as the three agents. Overall, everything was created from scratch and authored by the writer of this thesis, totaling approximately 200 lines of code for the RL environment, 650 lines for the RL agents, and thousands lines for the pipeline, analysis, and helper functions. The complete code is available in the author's GitHub repository at https://github.com/fabiorodp/uio_master_thesis under the GNU general public license, version 3, from 29th June 2007. It is important to mention that the reproducibility and results are achieved again with a high degree of reliability when the methodologies are replicated by the files in that GitHub repository.

Unfortunately, we did not have access to a professional data source such as Bloomberg or other high–level terminals currently available on the market. However, the data for this project was gathered straight from the B3 Sao Paulo Stock Exchange's website[1]. According to Brazilian legislation, they must make a raw data with all daily transactions publicly available for at least a short period of days (10 days). This data is registered every nano second, stored and freely published at the end of every daily session. The files also consist of the price and volume of every transaction that occurred during that day, for all Brazilian financial assets. Thus, throughout the one–year period of research, the author of this thesis continued to collect daily historical data from the B3 website, and treated it using his own Python code in order to make it ready for the RL environment used in this work.

Finally, a significant legal and ethical problem that we had to take into account is associated with the ownership, licensing and usage of the researched data. As mentioned earlier, our research utilizes data from B3, which are openly provided near the end of every day session. Despite that this data is open and free for the public, we were not authorized to republish its contents. As a consequence, it may compromise the reproducibility of our codes. To prevent such impacts, the reader may:

1.) use any alternative downloadable and open–source accessible online database, such as yahoo finance;

2.) follow the detailed processing construction of the data presented in the **Sections 1.4** and **5.1**,

without sacrificing reliability or performance of our environment or agents algorithms.

[1]https://www.b3.com.br/pt_br/market-data-e-indices/servicos-de-dados/market-data/cotacoes/cotacoes/

1.4 Reproducibility

This section introduces a guideline for understanding the usage of our built–from–scratch application programming interface (API), as well as reproducing the experiments performed in this thesis. Keeping in mind that we are processing a massive amount of data, the following steps must be carried out exactly as outlined for the algorithms to perform properly. Further, we recommend reading **Chapter 5** and **Chapter 6** first in order to comprehend the specifics of this tutorial. It is important to mention that all classes, modules, functions, and code–snippets are very well–commented and with their corresponding docstrings, so the user may easily replicate the experiments. However, if there still are questions, the author is happy and available to answer them by e–mail at `fabior@uio.no`.

API's STRUCTURE:

The API consists of the following directories and files:

```
|--- ./
|   |--- thesis/
|   |--- LICENSE
|   |--- README.md
|   |--- .gitignore
|   |--- environment.py
|   |--- algorithms.py
|   |--- helper.py
|   |--- pipeline.py
|   |--- technicalAnalysis.py
|   |--- parsingData.py
|   |--- runPipeline.py
|   |--- results60min.py
|   |--- results500kticks.py
|   |--- analysis60min.py
|   |--- analysis500kticks.py
|   |--- combinedAnalysis.py
|---
```

where

- `thesis/`: Directory where the electronic version of this thesis is stored.
- `LICENSE`: API's license.
- `README.md`: API's repository landing page.
- `.gitignore`: List with ignored files and folders.
- `environment.py`: Market's environment dynamics.
- `algorithms.py`: Reinforcement learning algorithms.
- `helper.py`: Auxiliary functions necessary for the API to operate correctly.
- `pipeline.py`: Hyper–parameters search module.
- `technicalAnalysis.py`: Technical Analysis modules.
- `parsingData.py`: Code–snippet showing how to extract and parse market data.

- `runPipeline.py`: Code–snippet showing how to run the pipeline and hyper–parameter search.
- `results60min.py`: Code–snippet showing how to collect the results for 60–min intervals.
- `results500kticks.py`: Code-snippet showing how to collect the results for 500k–tick intervals.
- `analysis60min.py`: Code–snippet showing how to perform the analysis for 60–min intervals.
- `analysis500kticks.py`: Code–snippet showing how to perform the analysis for 500k–tick intervals.
- `combinedAnalysis.py`: Code–snippet showing how to run the combined analysis.

PARSING MARKET'S DATA:

We use txt or csv daily files that include intraday raw data for all transactions of all assets traded in the B3's stock–exchange. The files' contents must comply with the following standard structure:

```
RptDt;TckrSymb;UpdActn;GrssTradAmt;TradQty;NtryTm;TradId;TradgSsnId;TradDt
2022-04-29;WDOM21;0;510840,000;1;090053610;10;1;2022-04-29
2022-04-29;WINM21;0;110840,000;1;090053610;20;1;2022-04-29
2022-04-29;PETR4;0;34.34,000;24;090053610;30;1;2022-04-29
2022-04-29;WINM21;0;110840,000;1;090053610;40;1;2022-04-29
2022-04-29;VALE3;0;88.90,000;3;090053610;50;1;2022-04-29
2022-04-29;WINM21;0;110840,000;1;090053610;60;1;2022-04-29
2022-04-29;WINM21;0;110840,000;1;090053610;70;1;2022-04-29
2022-04-29;VIIA3;0;2.90,000;1;090053610;80;1;2022-04-29
2022-04-29;WINM21;0;110840,000;5;090053610;90;1;2022-04-29
```

where

- `RptDt` is the date of the trade.
- `TckrSymb` is the ticker of the traded contract.
- `UpdActn` is always 0.
- `GrssTradAmt` is the executed value of the traded contract.
- `TradQty` is the number of contracts traded.
- `NtryTm` is the timestamp of the operation.
- `TradId` is the ID of the operation.
- `TradgSsnId` is always 1.
- `TradDt` is the date of the trade.

Note that each trading day has a different file that must have the exact filename format of `TradeIntraday_yyyymmdd_1.txt`, where `yyyymmdd` denotes the year, month, and day, respectively. These files require ZIP compression in order to be manageable, so the compressed filename format also must be `TradeIntraday_yyyymmdd_1.zip`.

For the extraction and parsing modules to function effectively, these daily files must be placed in a folder designated to each contract, exactly as follows:

```
|--- ./data/WINJ21/
|   |   |--- raw/
|   |   |   |--- ...
|   |   |   |--- TradeIntraday_20210302_1.zip
|   |   |   |--- TradeIntraday_20210303_1.zip
|   |   |   |--- TradeIntraday_20210304_1.zip
|   |   |   |--- ...
|--- ./data/WINM21/
|   |   |--- raw/
|   |   |   |--- ...
|   |   |   |--- TradeIntraday_20220429_1.zip
|   |   |   |--- TradeIntraday_20220428_1.zip
|   |   |   |--- TradeIntraday_20220427_1.zip
|   |   |   |--- ...
|--- ...
```

Now, making use of our `helper.getAnAsset()` module in helper.py[2], the chosen ticker/contract will be extracted from the raw files. We recommend setting both `in_folder` and `out_folder` from `helper.getAnAsset()`'s attributes to `None`, then the system will configure the market's data in accordance with the API's default. Thus, we end up with the following directory structure:

```
|--- ./data/WINJ21/
|   |   |--- raw/
|   |   |   |--- ...
|   |   |--- extracted/
|   |   |   |--- ...
|   |   |   |--- WINJ21_20210302.csv
|   |   |   |--- WINJ21_20210303.csv
|   |   |   |--- WINJ21_20210304.csv
|   |   |   |--- ...
|--- ./data/WINM21/
|   |   |--- raw/
|   |   |   |--- ...
|   |   |--- extracted/
|   |   |   |--- ...
|   |   |   |--- WINM21_20210302.csv
|   |   |   |--- WINM21_20210303.csv
|   |   |   |--- WINM21_20210304.csv
|   |   |   |--- ...
|--- ...
```

For parsing the data to a chosen framed interval, we utilize our modules `helper.parseIntoTimeBars()` or `helper.parseIntoTickBars()`. An example of a data directory after `helper.parseIntoTickBars()` has been properly used should be:

```
|--- ./data/WINJ21/
|   |   |--- raw/
|   |   |   |--- ...
|   |   |   |--- TradeIntraday_20210302_1.zip
```

[2]https://github.com/fabiorodp/uio_master_thesis/blob/main/helper.py

```
|   |   |   |--- TradeIntraday_20210303_1.zip
|   |   |   |--- TradeIntraday_20210304_1.zip
|   |   |   |--- ...
|   |   |--- extracted/
|   |   |   |--- ...
|   |   |   |--- WINJ21_20210302.csv
|   |   |   |--- WINJ21_20210303.csv
|   |   |   |--- WINJ21_20210304.csv
|   |   |   |--- ...
|   |   |--- WINJ21_500000ticks.csv

|--- ...
```

An example of these procedures is found in the file `parsingData.py`.

PIPELINE AND HYPER–PARAMETER SEARCH:

Our module `pipeline.pipeline()` in the file pipeline.py[3] performs the hyper–parameter search described in **Chapter 6.1**, and saves the results as a json file for further examination. Now that we have successfully parsed all market data for all contracts/tickers, we may invoke that pipeline module to perform a hyper–parameter search for each individually ticker, for example:

```
pipeline.pipeline(
    fileName="./data/WINJ21/WINJ21_500000tick.csv",
    initInvest=5600*5,
    params=None,
    outFolder=None,
    verbose=True
)
```

When the attribute `params` is set to `None`, the standard hyper–parameter search described in the application part of this thesis is carried out. If we intend to modify the parameter space of the search, we should pay attention to the function's docstring first; otherwise, if it is not correctly implemented, it will not work appropriately. Also, we recommend setting `outFolder` from `pipeline.pipeline()`'s attributes to `None`, then the system will create a "./results/" directory inside the API's root folder, as follows:

```
|--- ./results/
|   |   |--- WINJ21_500000ticks.json
|--- ...
```

To collect all the findings and proceed to the next phase of the research, this procedure must be done for each contract/ticker individually. We decided to do this step independently for each contract since it enables us to parallelize the process and get the results more quickly.

An example of these procedures is found in the file `runPipeline.py`.

RESULTS

The findings gathered in **Chapter 6.1** were the product of the hyper–parameter search, but the inference of these results were handled by the code–snippets in `results500kticks.py` and `results60min.py`. There, three distinct modules were employed, as follows:

[3]https://github.com/fabiorodp/uio__master__thesis/blob/main/pipeline.py

- `helper.loadResults()`: To load results from files;
- `helper.topWorstBest()`: To filter the top worst/best results;
- `helper.plotPies()`: To show pie plots of results.

ANALYSIS

Chapter 6.2 performed an analysis of the most optimal RL models, running the code–snippets in `analysis500kticks.py`, `analysis60min.py`, and `combinedAnalysis.py`. There, several modules were used, as follows:

- `helper.run500times()`: To run 500 times each RL algorithm with different seeds.
- `helper.optimal500()`: To get the most optimal model among 500 models.
- `helper.plotReturnTrajectories()`: To line–plot return trajectories.
- `helper.plotMeanReturnTrajectory()`: To line–plot mean return trajectory.
- `helper.plotDist()`: To histogram–plot final return trajectories' values.
- `helper.plotBox()`: To box– and swarm–plot final return trajectories' values.
- `helper.plotAllBoxes()`: To box–plot all final return trajectories' values of all algorithms.

1.5 Research context

RL is described as a learning system with an agent operating in an environment, getting quantifiable positive or negative feedback (reward) after each choice, and maximizing the quantity of this information over time in order to improve and maximize the overall reward. One of the first concepts of feedback–based learning dates all the way back to the 1950s, when models were based on trial–and–error learning, a concept known as the psychology of animal learning. Simultaneously, optimum control and its solution via the use of value functions and DP were well–known academic concepts. *Richard Bellman* [Bel52] proposed the Bellman equation and value iteration in the mid–1950s, which *Ronald Howard* [How64] further developed the theory in the early 1960s by including MDP and policy iteration techniques to enhance DP. Additionally, *Marvin Minsk* [Min95] explored various challenges in large trial–and–error learning situations in his work titled "*Steps toward artificial intelligence.*" From this point, the term RL was coined and became well recognized in the literature.

Throughout the 1970s and 1980s, scientists concentrated their efforts on supervised AI rather than RL. Despite this, *A. Harry Klopf* ([Klo72], [Klo75], [Klo82]) established a clear separation between supervised and RL, allowing subsequent researches to be more distinguished and focused on "[*controlling*] *the environment toward desired ends and away from undesired ends*" [SB19,

p. 19]. During these decades, *Richard S. Sutton* and *Andrew G. Barto* [SB81] carried out fundamental contributions to the difference between supervised and RL.

Combining trial–and–error and DP perspectives resulted in a third thread devoted to *temporal–difference* (TD) systems, in which decision–making is accomplished through the difference of the successive estimates of a particular parameter, for example, the expected probability of winning the blackjack game during its rounds given a particular goal in mind. There were various contributors to the ideas of TD, such as *Arthur Samuel* [Sam59], who was the first to suggest and develop a learning technique that incorporated TD concepts. *Minsky* [Min95] discussed *Samuel*'s work associating to secondary reinforcement theories. *Klopf* [Klo72] linked TD with trial–and–error learning and correlated it to animal learning psychology. *Ian Witten* ([Wit76], [Wit77]) was the earliest publication of a process called tabular TD(0) for use as part of an adaptive controller for solving MDPs. A similar method was published by *Sutton* [Sut84] that developed an actor–critic architecture that uses TD learning combined with trial–and–error learning.

Chris Watkins [Wat89] built on past work by developing a framework known as Q–learning in which TD and optimum control were completely integrated. He was the first to clearly describe the so–called approximate DP technique, which combines DP with online learning tools for non–stationary situations. These non–stationary cases provided a fresh perspective on the theories and techniques for solving problems involving complete and incomplete knowledge systems.

Various research and theoretical variants were published in the 1990s and 2000s, although all were constrained by the period's limited computing capability. However, the spread of RL became apparent in the 2010s as computing power increased. Indeed, the today's processing capability evolution is far more and more affordable than it was 20–30 years ago, allowing us to test such ideas in previously imagined scenarios. For instance, we may easily apply RL to dynamic and non–stationary systems with a high degree of dimensionality. Additionally, as a consequence of these technological advancements, researchers may combine deep, recurrent, or any complicated neural network architecture with advanced RL theories to generate even more promising outcomes than previously possible. As a result, *Google DeepMind* and *Montreal Institute for Learning Algorithms* [Mni+16] released a paper titled "*Asynchronous methods for deep reinforcement learning*" in which they presented a novel application that is regarded as a game–changer in RL efficiency by the academic community.

Regarding the implementation of RL techniques in the field of finance, particularly in algorithm trading, we would like to highlight three successful contributions: [BC12], [CS15] and [Cor+19]. These academic works propose automated FTSs based on various RL algorithms and their application to legit datasets of daily stock prices.

The first paper developed a TD and a Kernel–based RL methods, both of which were influenced by various 1990s and 2000s publications. The novelty of this initial research was that the authors included a third signal, dubbed "*stay–out–from–the–market signal*," into the decision–making process of the algorithms, while previous works solely examined two signals: sell and buy. Overall, their results were quite positive, with their top models generating more than 100% in final accumulated returns over a span of around 5,000 time periods.

The second publication engineered a Q–learning and a SARSA function approximated RL. The methodology used in the first paper was generally repeated in the second but with advances regarding the different RL types, and employment of a particular basis functions (or "*squashing function*" term used in the paper). Again, the final outcome of their best panels was significantly high, with final accumulated returns exceeding 90% over a course of around 7,000 time intervals.

The third work developed the second paper further. The same methodology was covered again, but now with three distinct function approximation RL algorithms: SARSA, Q–learning and Greedy–GQ. Moreover, this study evaluated more options of reward functions: Shape and Calmar ratios. Further, based on a literature review from 2013, a recently developed block representation of the features was employed, achieving consistent benefits. Finally, the final accumulated results overcame the operative annual returns of 50% in a period of approximately 18 years.

1.6 Structure of the thesis

This thesis was divided into three sections: theory, application, and analysis. The first part addressed the foundation theories that underpin the landing descent operated in the second part. We adopted a sequential exhibition to expose these subjects because the most recent theory is based upon the development of past notions, and without a comprehension of these earlier concepts, the reader would be unable to grasp the more advanced theory. Throughout the sections, we proceeded with the fundamental conceptions from the early twentieth century, such as stochastic and Markov processes. Later, we introduced a framework for sequential decision–making, set up notations and entities, as well as resolved a simple control case operating exhaustive computations. As soon as the foundation theory was solidified, we then went on to the reinforcement theory, beginning with the tabular RL notion. Although the tabular theory was not employed in the application section, the functional approximation RL is based on it, and so requires a thorough comprehension. Thereby, we devoted a whole chapter to function approximation, in which three methods (SARSA, Q–learning, and Greedy–GQ) were explained, and they were employed in the application part.

The second part of this research centered on the application case, which encouraged the development of a state–of–the–art approximated RL. We supplied a complete explanation of the circumstance we wanted to address, as well as a mathematical formalization of the topic. These two chapters outlined in detail how raw data were dealt with and adopted, including the dynamics and specificities of the environment. Additionally, we explored the Brazilian stock exchange, particularly the futures markets for mini–index contracts. At the end of this part, we merged the theory with the application issue, braking down the methods and engineering a code for their proper operation.

The third part of this thesis raised an analysis of the preceding two parts, their theory as well as application, comprising the associated findings, discussion, and conclusion. We started by describing the process through which we discovered the optimal models and their profitability. Simultaneously, we inspected the hyper–parameters search and their impact on the outcomes. The findings were then presented statistically, using 500 stochastically distinct simulations of the best models. That way, we removed any possible argument of lucky factor and further accessed statistical features that were utilized to evaluate the models' robustness. Finally, we dealt with the research questions, looked at potential future works, made allusion and viewpoints on some open academic subjects that are unresolved in the literature, and yet offered a fine summary and conclusion to all accomplished tasks.

PART I

Theory

CHAPTER 2

Foundation theory

This chapter presents some essential stochastic process theories that are utilized in the classical Markov decision process (MDP) theory and later subjects. Our explanations will not focus only on the minutiae of those topics, but rather on introducing certain necessary definitions, propositions, terminologies, and notations and finally on solving a simple control optimization problem, namely an MDP with exhaustive computations. This kind of scenario is particularly interesting to the researchers since it is often used in the study of real–world systems and serves as the foundation for the dissertation's major theme – reinforcement learning. In the end, an exhaustive computations example is added for the sake of comparison with the more advanced techniques exhibited later.

2.1 Stochastic process

Unlike the other sections of this thesis, the content here is basically a collection of definitions, propositions, remarks, and properties that will be used later. Important to mention that some definitions are reproduced from the literature, with proper references, and sometimes adapted to our words, notations and additions if we consider relevant. This collection is particularly critical for our study because, e.g., on many occasions, we simply refer to the definition's name, and the reader must be acquainted with the subject in order to comprehend the complete concept. We do not cover all of the details from each subject, but only those that are important for our work; thus, if the reader is interested in knowing more about topics that are not primarily addressed in this dissertation, we recommend the following works: [Wal11], [Lin17], [Ros19], [Kle20], and [CB21]. Therefore, we start with the definitions of state space, stochastic process, and Markov process:

Definition 2.1.1 ([CB21, Def. 2.1] - **Stochastic process**). Let $(\Omega, \mathcal{F}, P)$ be a complete probability space, T an index set, and $(E, \mathcal{B})$ a measurable space. An $(E, \mathcal{B})$–valued *stochastic process* on $(\Omega, \mathcal{F}, P)$ is a family $(X_t)_{t \in T}$ of random variables, that is, a family of measurable mappings $X_t : (\Omega, \mathcal{F}) \longrightarrow (E, \mathcal{B})$ for $t \in T$.

Definition 2.1.2 (**State space**)**.** The state space $\mathcal{S}$ is the set of all possible states of an environment. More formally, is the measurable space $(\mathcal{S}, 2^{\mathcal{S}})$ where the stochastic process of interest will take values.

- Note that $(E, \mathcal{B}) = (\mathcal{S}, 2^{\mathcal{S}})$, where $\mathcal{S}$ is a finite or countable set.
- We will repeatedly use s for both the current state and s' for the next (future) state, where each $s, s' \in \mathcal{S}$.

Definition 2.1.3 (**Markov process**)**.** A stochastic process $\{X_t,\ t = 0, 1, 2, \ldots\}$ taking values in $\mathcal{S}$ values, i.e., $X_t = s \in \mathcal{S}$, is said to be a Markov process if following property is satisfied:

$$P(X_{t+1} = s' | X_0 = s_0, \ldots, X_{t-1} = s_{t-1}, X_t = s) = P(X_{t+1} = s' | X_t = s), \tag{2.1}$$

for all $s_0, \ldots, s_{t-1}, s, s' \in \mathcal{S}$ and $t \in \mathbb{N} \cup \{0\}$.

The transition probability (TP) provides insight into the process's evolution and it is assumed to not change with time, that is, the Markov process is time homogeneous.

Definition 2.1.4 ([Wal11, Def. 7.5] - **One–step TP**)**.** A Markov process has one–step transition probabilities defined as

$$p_{X_t}(s' \mid s) \overset{\text{def}}{=} P(X_{t+1} = s' | X_t = s), \tag{2.2}$$

and the **transition probability matrix (TPM)** denoted by

$$\mathbf{P} \overset{\text{def}}{=} \Big[p_{X_t}(s' \mid s) \Big], \tag{2.3}$$

for all $s, s' \in \mathcal{S}$, and $t \in \mathbb{N} \cup \{0\}$.

Remark 2.1.5. In the literature, it is common to leave out the sub–index t from $p_{X_t(\cdot)}$, resulting in $p_X(\cdot)$. Moreover, it is possible to find the notation $p(\cdot)$ for the same one–step transition probability $p_{X_t(\cdot)}$.

Remark 2.1.6 ([Wal11, pp. 194–195] - **TP's properties**). For all possible states $s, k, s' \in \mathcal{S}$, if $p_{X_t}(s' \mid s)$ is a transition probability and $\mathbf{P}$ a transition probability matrix, then we have the following properties:

(a) **The n–step TP**: Let n be a positive integer, then

$$\mathbf{P}^{(n)} \overset{\text{def}}{=} \Big[p_{X_{t+n}}(s' \mid s) \Big], \tag{2.4}$$

where $p_{X_{t+n}}(s' \mid s) = P(X_{t+n} = s' | X_t = s)$.

(b) **Chapman–Kolmogorov equation**: Let a and b be positive integers, then

$$\begin{aligned}
p_{X_{t+(a+b)}}(s' \mid s) &\overset{\text{def}}{=} P\Big(X_{t+(a+b)} = s' | X_t = s\Big) \\
&= \sum_k P\Big(X_{(t+a)+b} = s', X_{t+a} = k | X_t = s\Big) \\
&= \sum_k \frac{P\Big(X_{(t+a)+b} = s', X_{t+a} = k, X_t = s\Big)}{P\Big(X_t = s\Big)} \\
&\overset{(*)}{=} \sum_k \frac{P\Big(X_{(t+a)+b} = s', X_{t+a} = k, X_t = s\Big)}{P\Big(X_{t+a} = k, X_t = s\Big)} \cdot \frac{P\Big(X_{t+a} = k, X_t = s\Big)}{P\Big(X_t = s\Big)} \\
&= \sum_k P\Big(X_{(t+a)+b} = s' | X_{t+a} = k, X_t = s\Big) \cdot P\Big(X_{t+a} = k | X_t = s\Big) \\
&\overset{(**)}{=} \sum_k P\Big(X_{(t+a)+b} = s' | X_{t+a} = k\Big) \cdot P\Big(X_{t+a} = k | X_t = s\Big) \\
&= \sum_k p_{X_{(t+a)+b}}(s' \mid k) \cdot p_{X_{t+a}}(k \mid s) \\
&= \mathbf{P}^{(b)} \ \circ \ \mathbf{P}^{(a)},
\end{aligned} \tag{2.5}$$

where $\circ$ denotes matrix multiplication, (*) represents algebraic manipulation, and (**) denoted Markov property.

Definition 2.1.7 (**State transition**)**.** A state s makes a transition to state s' ($s \longrightarrow s'$), in other words, state s' is ***accessible*** from s, such that

$$p_{X_{t+n}}(s' \mid s) > 0.$$

Conversely, if the states s and s' are ***not accessible*** from each other, then

$$p_{X_{t+n}}(s' \mid s) = 0.$$

Following that, we explain some valuable terminologies for Markov processes, which are assumptions frequently used in some subsequent definitions.

Remark 2.1.8 ([Ros19, pp. 205–215]).

(a) If no other state is accessible from state s', then this state s' is called as an ***absorbing*** state.

(b) If two states are accessible to each other, then they are said to ***communicate*** and are denoted by $s \longleftrightarrow s'$.

(c) Suppose that a collection of states communicate with each other, even though through an intermediary state in the group; thus, they are in the same ***class***.

(d) Any two classes of states can be either ***identical*** or ***disjoint***.

(e) Suppose all states communicate with each other, even though through an intermediary state. In that case, the process is ***irreducible***, i.e., there is only one class.

(f) State s is said to be:

- ***recurrent*** if $\sum_{n=1}^{\infty} p_{X_n}(s|s) = \infty$, i.e., starting in state s, the process will re–enter infinitely often times to that state.
- ***transient*** if $\sum_{n=1}^{\infty} p_{X_n}(s|s) < \infty$, i.e., starting in state s, the process will re–enter finitely often times to that state.

(g) Recurrent and transient properties are also applied to classes.

(h) A ***null recurrent*** state has its long–run proportion equal to zero.

(i) A ***positive recurrent*** state has its long–run proportion greater than zero.

(j) All states in an irreducible process can neither be transient nor null recurrent. Still, they can all be positive recurrent.

(k) A Markov process that can only return to a specific state after two or more steps is called ***periodic***. The converse process is named ***aperiodic***.

(l) An irreducible, positive recurrent, and aperiodic Markov process is also called ***ergodic***.

The calculations of the long–run proportion, limiting probabilities, and long–run average reward are done based on the following definitions:

Definition 2.1.9 ([Ros19, pp. 215–231] - **long–run proportion**)**.** If an irreducible Markov process $\{X_t,\ t = 0, 1, 2, \ldots\}$ has only positive recurrent states, then the long–run proportion, also called ***stationary probabilities***, can be obtained by solving the linear equations

$$\pi(s') = \sum_{s} \pi(s)\ p_{X_{t+n}}(s'|s), \tag{2.6}$$

and

$$\sum_{s'} \pi(s') = 1, \tag{2.7}$$

where $\pi(s)$ and $\pi(s')$ denote the long–run proportion of time that the process re–enters state s and s' respectively.

Definition 2.1.10 ([Ros19, pp. 232–233] - **Limiting probabilities**)**.** If $\{X_t,\ t = 1, 2, \ldots\}$ is an ergodic Markov process, then the limiting probabilities

$$\alpha(s') = \lim_{n \longrightarrow \infty} P(X_{t+n} = s' \mid X_t = s) = \lim_{n \longrightarrow \infty} P(X_{t+n} = s') \tag{2.8}$$

always exist and do not depend on an initial state s. Hence, $\alpha(s') = \pi(s')$, and as in 2.1.9, they can be obtained by solving the linear equations

$$\alpha(s') = \sum_{s} \alpha(s)\ p_{X_{t+n}}(s'|s), \tag{2.9}$$

and

$$\sum_{s'} \alpha(s') = 1. \tag{2.10}$$

Definition 2.1.11 ([Ros19, p. 230] - **long–run average reward**). If $\{X_t,\ t = 1, 2, \ldots\}$ is an irreducible Markov process with positive recurrent states and stationary probabilities $\pi(s')$, then the long–run average reward is given by

$$\sum_{s'} \overline{r}(s')\ \pi(s'), \tag{2.11}$$

where $\overline{r}(s')$ is the average reward received whenever the process enters state $s' \in \mathcal{S}$ (see **Definition 2.2.5**).

2.2 Sequential decision–making framework, notations, and entities

The sequential decision–making framework is the underlying concept behind of feedback learning from environment interactions. This system follows a stochastic process involving an agent who can choose different actions under different situations, continuously optimizing its decisions. This process is better described by Markov processes where actions influence immediate and subsequent rewards. The agent, also called learner or decision–maker, selects actions to be performed in an environment that responds, presenting new situations and rewarding a specific quantity for each choice. The main goal here is the maximization of rewards over time through new actions choices made by the agent. Simply put, this process can be reduced in three signals into a loop between the agent and environment (see Figure 1.2). The first signal corresponds to the environment's data (state) that will motivate the agent's choices. The second concerns the selection made by the agent (action). Finally, the last one represents the reward obtained from the environment under an agent's decision.

The states represent the environment's existing context. Any measurable information from the real world could affect the desired outcome. The collection of states is a space set (see 2.1.2) containing all the factors and knowledge that the decision–making agent could benefit from comprehending it and in any form. The agent will learn from these representations of the real–world scenarios and wisely deal with uncertainties by selecting different available actions.

Definition 2.2.1 (**Action space**). The action space is the finite set

$$\mathcal{A} \overset{\text{def}}{=} \{a_1, a_2, \ldots, a_N\},$$

of all possible actions.

The agent chooses, through a function denoted by $\mathcal{A}(s)$, an action a that is available for a given state s.

Proposition 2.2.2 (**Action function**). *Let set $\mathcal{A}$ be an action space in a decision–making process. The action function $\mathcal{A}(s)$ returns a subset of $\mathcal{A}$ containing all available actions for the given state s. $\mathcal{A}(s)$ may be a proper subset of $\mathcal{A}$ or even the empty set.*

For every unit time (step), the system either switches to a new state or stays the same. This phenomenon occurs in a Markov decision problem, meaning that an action–state transition probability

$$p(s' \mid s,\ a) = P\Big(X_{t+1} = s' \mid X_t = s,\ \mathcal{A}(s) = a\Big), \tag{2.12}$$

under the influence of an action a from $\mathcal{A}(s)$ is applied for each step. We will often use the notation $s \longrightarrow s'$, which represents the transition from s to s' state, where s can be equal or different from s', and both $s, s' \in \mathcal{S}$.

Besides the individual mechanisms previously examined, there exists a further entity in control problems, known as policy. This policy comprises a tool that determines the action to be conducted in an environment.

Definition 2.2.3 (***Policy space***). The set of all $|\mathcal{S}|$–tuples

$$\boldsymbol{\psi} \overset{\text{def}}{=} \mathcal{A}(s_1) \times \mathcal{A}(s_2) \times \ldots \times \mathcal{A}(s_{|\mathcal{S}|}) \tag{2.13}$$

where $\psi \in \boldsymbol{\psi}$ is such that $\psi_i \in \mathcal{A}(s_i)$, for $i = 1, 2, \ldots, |\mathcal{S}|$, receives the name policy space.

In its turn, the reward signal is an objective function to measure the benefits or harms from the agent's decisions, which can be averaged or discounted when maximized by the agent. The former is the expected (average) reward, and the latter is the expected discounted reward, where both are computed over time, i.e., a long trajectory of state–action transitions.

Definition 2.2.4 (***Immediate reward function***). The immediate reward earned in the transition from $s \longrightarrow s'$ states under the influence of an action a is denoted by

$$r(s, a, s'). \tag{2.14}$$

Correspondingly, the immediate reward earned in the transition from $s \longrightarrow s'$ states under the effect of a policy ψ is

$$r(s, \psi, s'). \tag{2.15}$$

Definition 2.2.5 (***Average immediate reward***)**.** The average immediate reward for all possible transitions under the effect of an action a or a policy ψ are

$$\overline{r}(s,a) = \sum_{s' \in \mathcal{S}} p(s' \mid s,\ a)\ r(s,a,s'), \tag{2.16}$$

and

$$\overline{r}(s,\psi) = \sum_{s' \in \mathcal{S}} p(s' \mid s,\ a)\ r(s,\psi,s'), \tag{2.17}$$

respectively.

In summary, a Markov decision process has four elements (state, action, policy, and reward) that describe its dynamics:

1. The decision maker (agent) has the intelligence to choose the system's optimal control mechanism (policy) in a given situation, i.e., the agent chooses a particular action when the process is in a specific state.
2. The system's control mechanism itself (policy) is a tuple of state and action elements. For example, a policy $\psi = (2, 1)$ means action 2 should be selected' when in state 1, and action 1 when in state 2.
   - A policy can be stationary and/or deterministic. A stationary policy means that it does not change with time. A deterministic policy implies that the agent can only choose one action out of the many allowed.
   - If the number of actions allowed and the number of states are both finite quantities, then we have a limited number of possible policies, which is bounded above by $|\mathcal{A}|^{|\mathcal{S}|}$. For instance, an system with 2 states and 2 actions have $2^2 = 4$ policies, such that:

$$\boldsymbol{\psi} = \Big\{(1,1),(1,2),(2,1),(2,2)\Big\}.$$

3. The transition probability matrix (TPM) contains all the combinations of the transition probabilities, under a specific action or policy (see 2.12).
4. The transition reward matrix (TRM) is the transition or immediate reward earned in every action–state transition. It is important to state that this reward can be positive or negative. Similar to TPM, we have TRM also associated with each action or policy.

The basis of a Markov decision process (MDP) is the Markov chains theory, which is widely applied in parallel with the MDP framework, so it is essential to distinguish one from the other. Markov chains, also called uncontrolled Markov chains, have no external agency that can control the path taken in the stochastic process. On the other hand, MDPs run with several different control mechanisms with their own action–state transition probability matrix, i.e., multiple actions in each state. Consequently, an MDP has as many TPMs and TRMs as its number of possible actions selected in each state. Moreover, this MDP architecture combines two terms: "Markov" and "decision process."

The former term denotes the "Markov property" from the "Markov chains" basis, which resembles the "memoryless" Markov property with a stochastic process, i.e., the conditional probability distribution of the probable next state depends only on the present state and is independent of the earlier ones (see 2.1.3). Thus, this stochastic model involves a sequence of finite or countable events (discrete or continuous) such that the probability of the upcoming event is based only on the current state. The latter term, "decision process," represents a stochastic control process, which, in reinforcement learning, implies a function that helps the agent choose the best action to take, transitioning to the next state. This function describes the action–state transition probability conditioned by the states ($s', s \in \mathcal{S}$) and a particular action ($a \in \mathcal{A}(s)$) taken by the agent. Similarly, the reward function $r(s, a, s')$ returns the immediate reward received after a taken action $a \in \mathcal{A}(s)$ that occurs the transition from s to s'.

Before we go deep into the optimization of the decision's mechanisms, it would be wise to analyze a straightforward (one–step transition MDP) example to consolidate the subjects dissected until now. For simplicity, the example given below is not focused on maximizing rewards but only aims to assemble the fragment of theories altogether. The maximization segment will be presented in the following section when we will solve an MDP case using exhaustive computations.

Example 2.2.6 ([Gos15, pp. 147–150])**.** Let $\mathcal{S} = \{1, 2\}$ be the state space (see 2.1.2), $\mathcal{A} = \{1, 2\}$ be the action space (see 2.2.1), and $\mathcal{A}(s = 1) = \{1, 2\}$ and $\mathcal{A}(s = 2) = \{1, 2\}$ be the action function (see 2.2.2). This two–state MDP has two actions allowed for each state. The TPMs associated with each action $\mathbf{P}_a$ are

$$\mathbf{P}_1 = \begin{bmatrix} 0.7 & 0.3 \\ 0.4 & 0.6 \end{bmatrix} \quad \text{and} \quad \mathbf{P}_2 = \begin{bmatrix} 0.9 & 0.1 \\ 0.2 & 0.8 \end{bmatrix} \tag{2.18}$$

The TRMs under each action $\mathbf{R}_a$ are

$$\mathbf{R}_1 = \begin{bmatrix} 6 & -5 \\ 7 & 12 \end{bmatrix} \quad \text{and} \quad \mathbf{R}_2 = \begin{bmatrix} 10 & 17 \\ -14 & -13 \end{bmatrix}. \tag{2.19}$$

The full graphical diagram for this example can be illustrated as follows:

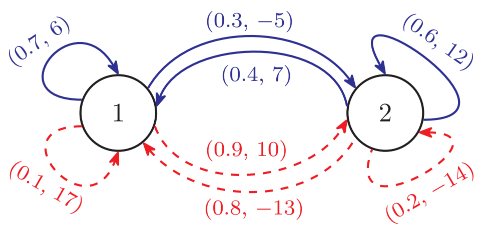

Figure 2.1: Graphical system with values of transition probability and transition reward for two different allowed actions (1 $\doteq$ solid blue, 2 $\doteq$ dashed red) for each state (s, s').

Consider a policy $\psi = (2, 1)$. The TPM associated with this policy contains the transition probabilities of action 2 in state 1 and action 1 in state 2, such that

$$\mathbf{P}_{(2,1)} = \begin{bmatrix} 0.9 & 0.1 \\ 0.4 & 0.6 \end{bmatrix}.$$

The one–step transition probability of the transition from state $s = 1$ to state $s' = 2$ when action $a = 1$ is selected is

$$p(s' \mid s,\ a) \overset{(2.12)}{=} p(2 \mid 1, 1) = 0.3.$$

The one–step transition probability of the transition from state $s = 2$ to state $s' = 2$ under policy $\psi = (2, 1)$ is

$$p\Big(s' \mid s,\ \psi\Big) \overset{(2.12)}{=} p\Big(2 \mid 2, (2, 1)\Big) = 0.6.$$

Like in the TPM case, we can obtain the TRM associated with the policy $\psi = (2, 1)$ by

$$\mathbf{R}_{(2,1)} = \begin{bmatrix} 10 & 17 \\ 7 & 12 \end{bmatrix}.$$

When policy ψ is observed, the immediate reward earned in the transition from state $s = 2$ to state $s' = 1$ is

$$r(s, \psi, s') \overset{(2.15)}{=} r\Big(2, (2, 1), 1\Big) = 7.$$

□

2.3 Solving a simple problem by exhaustive computations

A classic way of solving decision–making problems is through exhaustive computations utilizing all the theories reviewed previously. The solution is straightforward but only applicable for cases with small state space due to the necessity of high computing power. For simplicity, we will solve a two–state case, with two possible actions in each state, already presented in **Example 2.2.6**. First, we enumerate every policy that the agent can choose, followed by evaluating each state's stationary probabilities and its long–run average reward, and finally select the policy with the most satisfactory performance, the so–called optimal one.

In the **Example 2.2.6**, there are four possible policies that the agent can use to control the system:

$$\boldsymbol{\psi} = \Big\{(1, 1), (1, 2), (2, 1), (2, 2)\Big\}.$$

The TPMs and TRMs of these policies are generated from (2.18) and (2.19) respectively:

$$\mathbf{P}_{(1,1)} = \begin{bmatrix} 0.7 & 0.3 \\ 0.4 & 0.6 \end{bmatrix}, \qquad \mathbf{P}_{(1,2)} = \begin{bmatrix} 0.7 & 0.3 \\ 0.2 & 0.8 \end{bmatrix}$$

$$\mathbf{P}_{(2,1)} = \begin{bmatrix} 0.9 & 0.1 \\ 0.4 & 0.6 \end{bmatrix}, \qquad \mathbf{P}_{(2,2)} = \begin{bmatrix} 0.9 & 0.1 \\ 0.2 & 0.8 \end{bmatrix}$$

$$\mathbf{R}_{(1,1)} = \begin{bmatrix} 6 & -5 \\ 7 & 12 \end{bmatrix}, \qquad \mathbf{R}_{(1,2)} = \begin{bmatrix} 6 & -5 \\ -14 & 13 \end{bmatrix}$$

$$\mathbf{R}_{(2,1)} = \begin{bmatrix} 10 & 17 \\ 7 & 12 \end{bmatrix}, \qquad \mathbf{R}_{(2,2)} = \begin{bmatrix} 10 & 17 \\ -14 & 13 \end{bmatrix}.$$

From the matrices above and using (2.9) and (2.10), we get the limiting probabilities' linear equations for each state with respect to the individually corresponding policy:

For $\psi = (1,1)$:

$$\begin{aligned} 0.7\alpha_{(1,1)}(1) + 0.4\alpha_{(1,1)}(2) &= \alpha_{(1,1)}(1) \qquad &(\text{for } s' = 1) \\ 0.3\alpha_{(1,1)}(1) + 0.6\alpha_{(1,1)}(2) &= \alpha_{(1,1)}(2) \qquad &(\text{for } s' = 2) \\ 1.0\alpha_{(1,1)}(1) + 1.0\alpha_{(1,1)}(2) &= 1 \end{aligned}$$

For $\psi = (1,2)$:

$$\begin{aligned} 0.7\alpha_{(1,2)}(1) + 0.2\alpha_{(1,2)}(2) &= \alpha_{(1,2)}(1) \qquad &(\text{for } s' = 1) \\ 0.3\alpha_{(1,2)}(1) + 0.8\alpha_{(1,2)}(2) &= \alpha_{(1,2)}(2) \qquad &(\text{for } s' = 2) \\ 1.0\alpha_{(1,2)}(1) + 1.0\alpha_{(1,2)}(2) &= 1 \end{aligned}$$

For $\psi = (2,1)$:

$$\begin{aligned} 0.9\alpha_{(2,1)}(1) + 0.4\alpha_{(2,1)}(2) &= \alpha_{(2,1)}(1) \qquad &(\text{for } s' = 1) \\ 0.1\alpha_{(2,1)}(1) + 0.6\alpha_{(2,1)}(2) &= \alpha_{(2,1)}(2) \qquad &(\text{for } s' = 2) \\ 1.0\alpha_{(2,1)}(1) + 1.0\alpha_{(2,1)}(2) &= 1 \end{aligned}$$

For $\psi = (2,2)$:

$$\begin{aligned} 0.9\alpha_{(2,2)}(1) + 0.2\alpha_{(2,2)}(2) &= \alpha_{(2,2)}(1) \qquad &(\text{for } s' = 1) \\ 0.1\alpha_{(2,2)}(1) + 0.8\alpha_{(2,2)}(2) &= \alpha_{(2,2)}(2) \qquad &(\text{for } s' = 2) \\ 1.0\alpha_{(2,2)}(1) + 1.0\alpha_{(2,2)}(2) &= 1 \end{aligned}$$

Then, by solving the limiting probabilities' linear equations above, we obtain:

$$\begin{aligned} \alpha_{(1,1)}(1) &= 0.5714 \qquad \text{and} \qquad \alpha_{(1,1)}(2) = 0.4286 \\ \alpha_{(1,2)}(1) &= 0.4000 \qquad \text{and} \qquad \alpha_{(1,2)}(2) = 0.6000 \\ \alpha_{(2,1)}(1) &= 0.8000 \qquad \text{and} \qquad \alpha_{(2,1)}(2) = 0.2000 \\ \alpha_{(2,2)}(1) &= 0.6667 \qquad \text{and} \qquad \alpha_{(2,2)}(2) = 0.3333 \end{aligned}$$

Now, using (2.17), we find the average immediate reward for each possible transition:

For $\psi = (1,1)$:

$$\begin{aligned}
\bar{r}\Big(1,(1,1)\Big) &= p\Big(1 \mid 1,(1,1)\Big)r\Big(1,(1,1),1\Big) \\
&\quad + p\Big(2 \mid 1,(1,1)\Big)r\Big(1,(1,1),2\Big) \\
&= 0.7(6) + 0.3(-5) \\
&= 2.7
\end{aligned}$$

$$\begin{aligned}
\bar{r}\Big(2,(1,1)\Big) &= p\Big(1 \mid 2,(1,1)\Big)r\Big(2,(1,1),1\Big) \\
&\quad + p\Big(2 \mid 2,(1,1)\Big)r\Big(2,(1,1),2\Big) \\
&= 0.4(7) + 0.6(12) \\
&= 10
\end{aligned}$$

For $\psi = (1,2)$:

$$\begin{aligned}
\bar{r}\Big(1,(1,2)\Big) &= p\Big(1 \mid 1,(1,2)\Big)r\Big(1,(1,2),1\Big) \\
&\quad + p\Big(2 \mid 1,(1,2)\Big)r\Big(1,(1,2),2\Big) \\
&= 0.7(6) + 0.3(-5) \\
&= 2.7
\end{aligned}$$

$$\begin{aligned}
\bar{r}\Big(2,(1,2)\Big) &= p\Big(1 \mid 2,(1,2)\Big)r\Big(2,(1,2),1\Big) \\
&\quad + p\Big(2 \mid 2,(1,2)\Big)r\Big(2,(1,2),2\Big) \\
&= 0.2(-14) + 0.8(13) \\
&= 7.6
\end{aligned}$$

For $\psi = (2,1)$:

$$\begin{aligned}
\bar{r}\Big(1,(2,1)\Big) &= p\Big(1 \mid 1,(2,1)\Big)r\Big(1,(2,1),1\Big) \\
&\quad + p\Big(2 \mid 1,(2,1)\Big)r\Big(1,(2,1),2\Big) \\
&= 0.9(10) + 0.1(17) \\
&= 10.7
\end{aligned}$$

$$\begin{aligned}
\bar{r}\Big(2,(2,1)\Big) &= p\Big(1 \mid 2,(2,1)\Big)r\Big(2,(2,1),1\Big) \\
&\quad + p\Big(2 \mid 2,(2,1)\Big)r\Big(2,(2,1),2\Big) \\
&= 0.4(7) + 0.6(12) \\
&= 10
\end{aligned}$$

For $\psi = (2, 2)$:

$$\begin{aligned}\bar{r}\Big(1, (2,2)\Big) &= p\Big(1 \mid 1, (2,2)\Big) r\Big(1, (2,2), 1\Big) \\ &\quad + p\Big(2 \mid 1, (2,2)\Big) r\Big(1, (2,2), 2\Big) \\ &= 0.9(10) + 0.1(17) \\ &= 10.7\end{aligned}$$

$$\begin{aligned}\bar{r}\Big(2, (2,2)\Big) &= p\Big(1 \mid 2, (2,2)\Big) r\Big(2, (2,2), 1\Big) \\ &\quad + p\Big(2 \mid 2, (2,2)\Big) r\Big(2, (2,2), 2\Big) \\ &= 0.2(-14) + 0.8(13) \\ &= 7.6\end{aligned}$$

In the end, we use (2.11) to find the long–run average reward:

For $\psi = (1, 1)$:

$$\begin{aligned}\rho_{(1,1)} &= \alpha_{(1,1)}(1)\ \bar{r}\Big(1, (1,1)\Big) + \alpha_{(1,1)}(2)\ \bar{r}\Big(2, (1,1)\Big) \\ &= 0.5741(2.7) + 0.4286(10) \\ &= 5.83\end{aligned}$$

For $\psi = (1, 2)$:

$$\begin{aligned}\rho_{(1,2)} &= \alpha_{(1,2)}(1)\ \bar{r}\Big(1, (1,2)\Big) + \alpha_{(1,2)}(2)\ \bar{r}\Big(2, (1,2)\Big) \\ &= 0.4(2.7) + 0.6(7.6) \\ &= 5.64\end{aligned}$$

For $\psi = (2, 1)$:

$$\begin{aligned}\rho_{(2,1)} &= \alpha_{(2,1)}(1)\ \bar{r}\Big(1, (2,1)\Big) + \alpha_{(2,1)}(2)\ \bar{r}\Big(2, (2,1)\Big) \\ &= 0.8(10.7) + 0.2(10) \\ &= 10.56\end{aligned}$$

For $\psi = (2, 2)$:

$$\begin{aligned}\rho_{(2,2)} &= \alpha_{(2,2)}(1)\ \bar{r}\Big(1, (2,2)\Big) + \alpha_{(2,2)}(2)\ \bar{r}\Big(2, (2,2)\Big) \\ &= 0.6667(10.7) + 0.3333(7.6) \\ &= 9.66\end{aligned}$$

Hence, the policy $\psi = (2, 1)$ that achieved $\rho_{(2,1)} = 10.56$ (highest long–run average reward) represents the optimal policy for this control problem. □

As an evident conclusion, these extensive enumerations and computations are computationally costly, rendering the resolution of more complex problems unfeasible.

CHAPTER 3

Solutions based on tabular format

The previous chapters discussed some of the fundamental knowledge that underpins the thesis' topics. We became acquainted with the classical decision problem's dynamics, definitions, properties, notations, entities, etc., and, in the end, we solved a straightforward case using exhaustive computations. However, it turns out that such solutions are not feasible for problems with an ample state space. Moreover, we are not always aware of the required transition probabilities of the system to be controlled. Consequently, we need new tools that could produce the same high–level results but more efficiently and can be applied in additional types of adversities.

This chapter assesses what was learned previously while examining newer approaches to solving decision processes. Some new notations and the essential facts needed to understand the subsequent chapters are introduced here. Nevertheless, Bellman's equations, value equations, and policy control are the central subjects. Then, we quickly progress toward a more engineering–style exposition of the reinforcement learning evolution, covering key computational concepts such as Dynamic Programming, Monte Carlo, temporal difference, K–step bootstrapping, on–policy SARSA, and off–policy Q–learning methods.

3.1 Markov decision process

It is essential to keep in mind that this theory is restricted to countable Markov decision process instances as well as the discounted total expected reward condition. Yet, the findings also apply to Markov decision process with continuous state–action under certain technical conditions. This disclaimer extends to the results in subsequent sections and chapters.

As previously seen, the Markov decision process is obtained by embedding controls with feedback loops into the framework of Markov chains (see 1.2). This process gives rise to a trajectory of variables that starts as follows:

$$S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, \dots \ . \tag{3.1}$$

A countable Markov decision process is a combination of a stochastic process (see 2.1.1), a Markov chain (see 2.1.3), and a control mechanism (see 2.2.3), which is defined mathematically by the tuple

$$\Big\{\mathcal{S},\ \mathcal{A},\ p(s' \mid s,\ a),\ \mathcal{R},\ \gamma\Big\}, \tag{3.2}$$

where the following are its constituents:

- The first element $\mathcal{S}$ is the countable non–empty set of all possible states $s \in \mathcal{S}$. The set $\mathcal{S}$ can be discrete or continuous, and an observed state at time t, denoted by S_t, can represent a single value or a vector. $S_t = s$ and $S_{t+1} = s'$ are often denoted in this thesis as the current and future states respectively.
- The second $\mathcal{A}$ is the countable non–empty set of actions. Typically used in the literature, $\mathcal{A}(s)$ represents an action function (see 2.2.2) that returns a set of possible actions $A_t \in \mathcal{A}(s) \subseteq \mathcal{A}$ given a state $S_t = s$.
- The third is the one–step state transition probability (see 2.1.4), also called the probability measure. In our case, this measure is conditioned by the current state $S_t = s$ and the control variable $A_t = a$, i.e., $(s,\ a) \in \mathcal{S} \times \mathcal{A}$, such that

  $$p(s' \mid s,\ a) \overset{\text{def}}{=} P(S_{t+1} = s' \mid S_t = s,\ A_t = a). \tag{3.3}$$

  Literally, equation (3.3) reads as the probability of going into the future state s' if the agent is in state s and takes action a. Note that this transition probability does not depend on t, by the Markov property; therefore, the $p(\cdot)$'s sub–index t was omitted.
- The fourth is the set $\mathcal{R}$ of all possible random rewards R_{t+1}[1] $\in \mathcal{R} \subset \mathbb{R}$.
- The last one is the discount factor $0 \leq \gamma \leq 1$ that reflects the value in time of the rewards.

Since we deal with well–understood deterministic technical conditions, the agent's actions are the only variable that directly impacts the system. This influence exists because the agent only needs to know the current state and what action to choose; then, forecasting the upcoming state is possible. It can stochastically anticipated as to how the scenario develops with a high degree of assurance, making long–term planning predictable. However, the system's dynamics are still not entirely known, even though we can predict it with high confidence; thus, the states and rewards obtained by the agent are all random variables with well–defined probability distributions conditioned only on the preceding state and action. Hence, a joint probability distribution characterizes the state and reward pairs' transitions, i.e., the whole dynamics of the environment.

The joint transition probability is a mapping $p(\cdot) : \mathcal{S} \times \mathcal{R} \times \mathcal{S} \times \mathcal{A} \longrightarrow [0, 1]$ that combines $p(s' \mid s,\ a)$ and $p(r \mid s,\ a)$, such that

$$p(s',\ r \mid s,\ a) \overset{\text{def}}{=} P\{S_{t+1} = s',\ R_{t+1} = r \mid S_t = s,\ A_t = a\}, \tag{3.4}$$

for all $s, s' \in \mathcal{S}$, $r \in \mathcal{R}$, and $a \in \mathcal{A}(s)$, which specifies a probability distribution for each choice of s and a

$$\sum_{s' \in \mathcal{S}, r \in \mathcal{R}} p(s', r \mid s,\ a) = 1, \tag{3.5}$$

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

[1] We use R_{t+1} instead of R_t to represent the reward due to A_t because it highlights that the new state S_{t+1} and reward R_{t+1} are jointly settled.

Note that the joint probability can be called the probability of anchoring in the future state s prime and obtaining an associated reward r, given the agent is currently in state s and decides to take action a. Furthermore, it is necessary to observe that the joint transition probabilities (3.4) specify the state transition probabilities (3.3) in the following way:

$$p(s' \mid s,\ a) \overset{\text{def}}{=} \sum_{r \in \mathcal{R}} p(s',\ r \mid s,\ a). \tag{3.6}$$

Next, the immediate reward functions are commonly represented by:

1. The mapping $r(\cdot) : \mathcal{S} \times \mathcal{A} \longrightarrow \mathbb{R}$ that returns the expected value of a random reward R_{t+1} received for the state $S_t = s$ and the taken action $A_t = a$:

$$\begin{aligned} r(s,a) &\overset{\text{def}}{=} E\Big[R_{t+1} \mid S_t = s,\ A_t = a\Big] \\ &\overset{\text{[SB19, p. 49]}}{=} \sum_{r \in \mathcal{R}} r \sum_{s' \in \mathcal{S}} p(s',\ r \mid s,\ a). \end{aligned} \tag{3.7}$$

2. The three argument function $r(\cdot) : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \longrightarrow \mathbb{R}$ that outputs the expected immediate reward for state–action–next–state triples:

$$\begin{aligned} r(s,a,s') &\overset{\text{def}}{=} E\Big[R_{t+1} \mid S_t = s,\ A_t = a,\ S_{t+1} = s'\Big] \\ &\overset{\text{[SB19, p. 49]}}{=} \sum_{r \in \mathcal{R}} r\ \frac{p(s',\ r \mid s,\ a)}{p(s' \mid s,\ a)}. \end{aligned} \tag{3.8}$$

As previously remarked, a Markov decision process represents a technique for modeling sequential decision–making circumstances in which an agent interacts successively with an environment. For example, in a Markov decision process M, let $t \in \mathbb{N} \cup \{0\}$ symbolize the current stage of the system, and $S_t \in \mathcal{S}$ and $A_t \in \mathcal{A}(S_t) \subseteq \mathcal{A}$ characterize the system's random state and the decision maker's action, both at time t, respectively. When the agent takes an action, the following transition occurs:

$$(S_{t+1}, R_{t+1}) \overset{\text{def}}{\sim} p(\circ | S_t, A_t), \tag{3.9}$$

which means that a tuple containing the next state and reward is drawn from a transition probability measure.

The agent of this system chooses its actions depending on the observed state's history at every point in time. A policy or behavior is a rule that describes how actions are chosen. Suppose a stochastic process $\{S_t, A_t, R_{t+1};\ t \geq 0\}$ where (3.9) holds, and A_t is the action derived from the agent's behavior based on the system history. The purpose here is to devise a strategy for selecting actions that maximizes the expected total discounted return. A cumulative return $G_{t:T}$ is a specific function of the total cumulative reward. In a time–discounted case, that is the sum of the product between a received reward R_t and its discount factor γ that yields

$$G_{t:T} \overset{\text{def}}{=} \gamma^0 R_{t+1} + \gamma^1 R_{t+2} + \gamma^2 R_{t+3} + \ldots + \gamma^{T-t-1} R_T, \tag{3.10}$$

up to the final step T.

Note that definition 3.10 has a final step T, which ends in a state called ***terminal***. From that, we come up with two different situations that can happen in any experiment based on rewards. The ***episodic tasks*** or ***trial tasks*** occur when the agent–environment–reward breaks naturally into sub–sequences, and each of the sub–sequences has a mutually–exclusive ending and starting points from each other. On the other hand, the ***continuing tasks*** arise when the agent–environment–reward goes on continually without limit, turning $T = \infty$.

Another essential conception is ***discounting***, which is denoted by the parameter $0 \leq \gamma \leq 1$, often called the ***discount rate***. The discount rate determines the present value of future rewards, allowing the agent to maximize the sum of future rewards, not only immediately but also in the long term, resulting in the following:

$$\begin{aligned} G_{t:T} &\overset{\text{def}}{=} \gamma^0 R_{t+1} + \gamma^1 R_{t+2} + \gamma^2 R_{t+3} + \ldots + \gamma^{T-t-1} R_T \\ &= \sum_{k=0}^{T-t-1} \gamma^k \; R_{t+k+1} \end{aligned} \tag{3.11}$$

While the parameter γ fluctuates from 0 to 1, the variable G_t takes future rewards into account more strongly, i.e., the future reward is worth exponentially less than the nearest reward, possibly providing more meaningful earnings than an exclusively immediate approach without a farsighted picture. Keep in mind that when $\gamma = 1$, we have an ***undiscounted*** MDP with an agent considering all future rewards equivalent to the immediate reward. Conversely, if $\gamma = 0$, the agent only assesses the immediate reward and discards the long–term return.

Henceforth, a computationally recursive strategy can be applied in (3.11) to save expensive computations:

$$\begin{aligned} G_t &\overset{\text{def}}{=} \gamma^0 R_{t+1} + \gamma^1 R_{t+2} + \gamma^2 R_{t+3} + \ldots \\ &= R_{t+1} + \gamma \; (R_{t+2} + \gamma R_{t+3} + \ldots) \\ &= R_{t+1} + \gamma G_{t+1}, \end{aligned} \tag{3.12}$$

where $T = \infty$ in this case.

3.2 Value functions

With all of the components presented in the previous section, we can then pursue the MDP's goal of maximizing the expected total cumulative discounted reward, or simply, expected return. Consequently, a behavior mechanism that selects the system's judgments, so–called policy (see 2.2.3), is necessary. There exist two types of policies: deterministic and stochastic. The difference between them is that the former returns an action, whereas the latter provides a probability. We can explore both characterizations:

1. A mapping $\psi(\cdot) : \mathcal{S} \longrightarrow \mathcal{A}$ defines the **deterministic stationary policy** $\psi(S_t)$ that receives a state S_t as an argument and outputs an action A_t, such that
$$A_t \overset{\text{def}}{=} \psi(S_t). \tag{3.13}$$

2. A **stochastic stationary policy** $\psi(a \mid s)$ maps states to probability distributions over the action space, i.e., a probability distribution for an $A_t \in \mathcal{A}(S_t)$, given each $S_t \in \mathcal{S}$. Hence, since

$$A_t \overset{\text{def}}{\sim} \psi(\circ \mid S_t), \tag{3.14}$$

the process $\{S_t,\ t \geq 0\}$ is time–homogeneous Markov chain. It should be noted that the notation Ψ_{stat} is used throughout this dissertation to refer to the set of all stochastic stationary policies, and for convenience, we frequently use the term "policy" rather than "stochastic stationary policies". Nevertheless, it is important to remember that the sum of probabilities of all possible actions $a \in \mathcal{A}$ from a determined state $S_t = s$ is equal to 1:

$$\sum_{a \in \mathcal{A}} \psi(a \mid s) = 1. \tag{3.15}$$

In the preceding chapter, we introduced a straightforward method for identifying optimal behaviors in some MDPs (see 2.3). Such techniques provide us with a list of all conceivable behaviors to determine which ones yield the best possible value for each initial state. This plan, however, is impractical when the states and actions are numerous. A more effective approach is based on value functions, where we seek to determine their optimal values, the so–called optimal value function, and therefore the system's optimal behavior.

Accordingly , a clever way to solve an MDP without extensive computations (see 2.3) is to use a ***state–value function*** under a fixed policy $\psi \in \Psi_{\text{stat}}$ at time t. This state–value function numerically estimates "*how good it is for the agent to be in a given state*" [SB19, p. 58].

The value function that maps $V^\psi : \mathcal{S} \longrightarrow \mathbb{R}$ is called the ***state–value function***. This function specifies a conditional expected return when the system begins in the state $S_0 = s$ and follows a fixed policy $\psi \in \Psi_{\text{stat}}$ thereafter. That is expressed mathematically as

$$V_t^\psi(s) \overset{\text{def}}{=} E^\psi \Big[G_t \mid S_t = s \Big] \tag{3.16}$$

for all $s \in \mathcal{S}$, where G_t is defined in (3.12), and the superscript ψ on the expectation sign $E^\psi[\cdot]$ indicates that this expectation is also conditioned on a specific ψ being followed.

Similar to (3.16), the value function that maps $Q^\psi : \mathcal{S} \times \mathcal{A} \longrightarrow \mathbb{R}$ is called ***state–action–value function***. In this variation, a state $S_0 = s$ and a first action $A_0 = a$ are arbitrarily selected, and afterward, their subsequent decisions are chosen by applying a fixed policy $\psi \in \Psi_{\text{stat}}$. This policy selects all the subsequent actions and evaluates "*how good it is to perform a given action in a given state*" [SB19, p. 58]. That is mathematically defined as

$$Q_t^\psi(s,\ a) \overset{\text{def}}{=} E^\psi \Big[G_t \mid S_t = s,\ A_t = a \Big], \tag{3.17}$$

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

Note that the expected values in (3.16) and (3.17) denote an expected value of a random variable G_t; therefore, those expressions also result in a random variable.

3.3 Bellman equations

While equation (3.16) gives the expected reward value for each state, equation (3.17) gives the same besides taking an initial action at time step t. These formulas are used to solve an MDP in two different ways: When the transition probabilities are known, the problem is easily solved by dynamic programming; however, when the transition probabilities are unknown, both functions can be empirically estimated from experience, using ***Monte Carlo*** simulations. For any case, those value functions satisfy a recursive relationship similar to what we have seen in (3.12). So, the following is the state–value function:

$$\begin{aligned} V_t^{\psi}(s) &\overset{\text{def}}{=} E^{\psi}\Big[G_t \mid S_t = s\Big] \\ &\overset{(3.11)}{=} E^{\psi}\Big[\sum_{k=0}^{T-t-1} \gamma^k R_{t+k+1} \mid S_t = s\Big] \\ &\overset{(3.12)}{=} E^{\psi}\Big[R_{t+1} + \gamma G_{t+1} \mid S_t = s\Big] \\ &= E^{\psi}\Big[R_{t+1} + \gamma V_{t+1}^{\psi}(S_{t+1}) \mid S_t = s\Big] \\ &= \sum_{a \in \mathcal{A}} \psi(a|s) \sum_{r,\ s'} p(r,\ s' \mid s,\ a)\ \Big[r + \gamma\ V_{t+1}^{\psi}(s')\Big], \end{aligned} \tag{3.18}$$

for all $r \in \mathcal{R}$ and $s' \in \mathcal{S}$, where G_{t+1} is an unbiased estimate for $V_{t+1}^{\psi}(S_{t+1})$.

Proof. We prove the last equality of (3.18):

(i) From probability theory, we have the definition of expectations:

$$E[X] \overset{\text{def}}{=} \sum_{x \in X} x\ p(x).$$

(ii) Let X be $R_{t+1} + \gamma V_{t+1}^{\psi}(S_{t+1})$ under the condition $S_t = s$, and since R_{t+1} and $V_{t+1}^{\psi}(S_{t+1})$ are random variables that take values for all $r \in \mathcal{R}$ and $s' \in \mathcal{S}$, then

$$E\Big[R_{t+1} + \gamma V_{t+1}^{\psi}(S_{t+1}) | S_t = s\Big] \overset{\text{(i)}}{=} \sum_{r \in \mathcal{R},\ s' \in \mathcal{S}} p(r, s' \mid s)\Big[r + \gamma\ V_{t+1}^{\psi}(s')\Big].$$

(iii) However, note that the expectation in (3.18) has the superscripted policy ψ, which we have not considered in step (ii). That policy takes values for all $a \in \mathcal{A}$, following the technical condition in (3.15). Then, the probability distribution $p(r, s' \mid s)$ in (ii) turns to be a marginal probability considering all $a \in \mathcal{A}$, such that

$$\begin{aligned}
p(r, s' \mid s) &= \sum_{a \in \mathcal{A}} p(r, s', a \mid s) && \text{(marginal probability)} \\
&= \sum_{a \in \mathcal{A}} \frac{p(r, s', a, s)}{p(s)} && \text{(Bayes theorem)} \\
&= \sum_{a \in \mathcal{A}} \frac{p(r, s', a, s)}{p(s)} \, \frac{p(a, s)}{p(a, s)} && \text{(algebraic manipulation)} \\
&= \sum_{a \in \mathcal{A}} \frac{p(r, s', a, s)}{p(a, s)} \, \frac{p(a, s)}{p(s)} && \text{(algebraic manipulation)} \\
&= \sum_{a \in \mathcal{A}} p(r, s' | s, a) \; p(a|s) && \text{(Bayes theorem)} \\
&= \sum_{a \in \mathcal{A}} p(r, s' | s, a) \; \psi(a|s) && (3.14).
\end{aligned}$$

(iv) Finally, swapping $p(r, s' \mid s)$ from (ii) with the equivalent result in (iii), we end up with the complete proof:

$$\sum_{r \in \mathcal{R}, s' \in \mathcal{S}} p(r, s'|s) \Big[r + \gamma V^{\psi}_{t+1}(s') \Big] = \sum_{a \in \mathcal{A}} \psi(a|s) \sum_{r \in \mathcal{R}, s' \in \mathcal{S}} p(r, s'|s, a) \Big[r + \gamma V^{\psi}_{t+1}(s') \Big].$$

■

Likewise in (3.18), the state–action–value function is

$$\begin{aligned}
Q^{\psi}_t(s,\ a) &\overset{\text{def}}{=} E^{\psi} \Big[G_t \mid S_t = s,\ A_t = a \Big] \\
&\overset{(3.11)}{=} E^{\psi} \Big[\sum_{k=0}^{T-t-1} \gamma^k R_{t+k+1} \mid S_t = s,\ A_t = a \Big] \\
&\overset{(3.12)}{=} E^{\psi} \Big[R_{t+1} + \gamma G_{t+1} \mid S_t = s,\ A_t = a \Big] \qquad (3.19) \\
&= E \Big[R_{t+1} + \gamma \ Q^{\psi}_{t+1}(S_{t+1}, A_{t+1}) \mid S_t = s,\ A_t = a \Big] \\
&= \sum_{s', r} \ p(s',\ r \mid s,\ a) \ \Big[r + \gamma \ Q^{\psi}_{t+1}(s', a') \Big],
\end{aligned}$$

for all $r \in \mathcal{R}$, $s' \in \mathcal{S}$, and $a' \in \mathcal{A}$, where G_{t+1} is an unbiased estimate for $Q^{\psi}_{t+1}(S_{t+1}, A_{t+1})$. Consider how the expectation of the fourth equality has lost the policy ψ superscript, as the first action $A_0 = a$ is arbitrary given, and subsequent actions are determined by the state–action–value function enclosed in square brackets.

Proof. We prove the last equality of (3.19):

(i) From probability theory, we get the definition of expectations:

$$E[X] \overset{\text{def}}{=} \sum_{x \in X} p(x) \ x.$$

(ii) Let X be $R_{t+1} + \gamma Q^{\psi}_{t+1}(S_{t+1}, A_{t+1})$ under the condition of $S_t = s$ and $A_t = a$, and since R_{t+1} and $Q^{\psi}_{t+1}(S_{t+1}, A_{t+1})$ are random variables that take values for all $r \in \mathcal{R}$, $s' \in \mathcal{S}$, and $a' \in \mathcal{A}$,

$$E\Big[R_{t+1}+\gamma Q^{\psi}_{t+1}(S_{t+1}, A_{t+1})|S_t = s, A_t = a\Big] \overset{\text{(i)}}{=} \sum_{r\in\mathcal{R},\ s'\in\mathcal{S}} p(r, s' \mid s, a)\Big[r + \gamma\ Q^{\psi}_{t+1}(s', a')\Big].$$

■

The final equations (3.18) and (3.19) are the so–called ***Bellman equations***. Illustrious author Dr. Richard Bellman proposed the Bellman equations' theory in [Bel52] as part of his pioneering work on dynamic programming in early 1950's.

Due to the extreme importance of these concepts and to give the reader a comprehensive understanding of the topic, the diagrammatic representation of both equations is given in **Figure 3.1**.

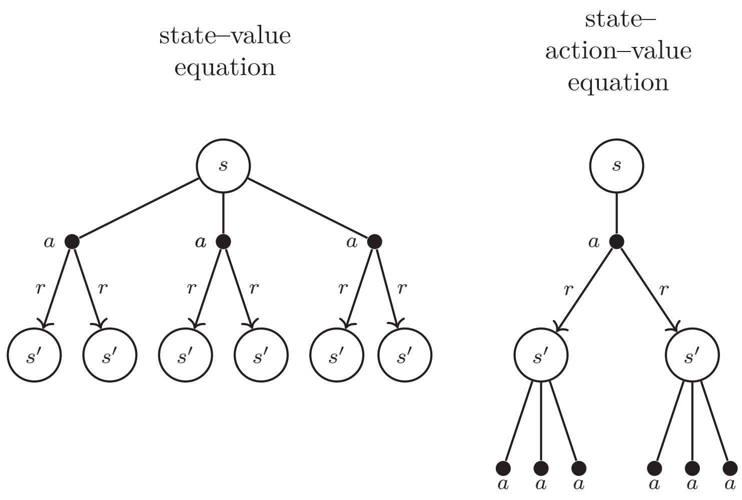

Figure 3.1: Diagram illustrating the state–value and state–action–value equations.

There are several relations between the state–value and state–action–value functions that the literature frequently makes use of, and we should be aware of them:

(1) If the first action input to the state–action–value equation be drawn from the policy, i.e., $a \sim \psi(a|s)$, then there exists the relation

$$\sum_{a\in\mathcal{A}} \psi(a \mid s)\ Q^{\psi}_{t}(s,\ a) = V^{\psi}_{t}(s), \tag{3.20}$$

for all $s \in \mathcal{S}$.

(2) Using (3.20), we obtain another variant for state–action–value function:

$$Q_t^{\psi}(s,\ a) = E^{\psi}\Big[R_{t+1} + \gamma\ V_{t+1}^{\psi}(S_{t+1}) \mid S_t = s, A_t = a\Big], \tag{3.21}$$

for all $s \in \mathcal{S}$. As we can see, the distinction between (3.21) and the fourth equality in (3.19) is quite subtle and only in the superscript ψ on the expectation sign, yet it significantly impacts the calculations.

Note that, since our thesis concerns stochastic policies, we omit the part of the theory dealing with Bellman equations for deterministic policies. Therefore, we refer interested readers to [SB19], and [Sze09], which brilliantly elucidate this theme.

3.4 Bellman optimality equations

For a finite MDP with several distinct states, the Bellman equation yields a set of linear equations defining the value function at time t for each state in terms of expected immediate rewards plus the expected value function at time $t+1$. As a result, knowing the MDP's transition probabilities allows us to solve this linear system faster than the exhaustive method presented in the previous chapter. To accomplish that, we still need to examine the ***Bellman optimality equation*** for state–value, and action–value functions.

The Bellman optimality equation for state–value function is denoted by $V_t^{\psi^*}(s)$ and is known to be the best result for a given state out of all possible policies.

Definition 3.4.1 (**Optimal state–value function**). If $V_t^{\psi^*}(s) \geq V_t^{\psi}(s)$, then $\psi^* \geq \psi$, for all $s \in \mathcal{S}$. Therefore, the optimal state–value function can be determined by

$$V_t^{\psi^*}(s) \overset{\text{def}}{=} \max_{\psi}\ V_t^{\psi}(s) \tag{3.22}$$

for all $s \in \mathcal{S}$, where ψ^* is the optimal policy for all states of the system.

Similarly, the Bellman optimality equation for action–value function returns the best expected reward for an initial action a in state s and then follows an optimal policy ψ^*.

Definition 3.4.2 (**Optimal action–value function**). If $Q_t^{\psi^*}(s,\ a) \geq Q_t^{\psi}(s,\ a)$, then $\psi^* \geq \psi$, for all $s \in \mathcal{S}$. Therefore, the optimal action–value function can be expressed by

$$Q_t^{\psi^*}(s,\ a) \overset{\text{def}}{=} \max_{\psi}\ Q_t^{\psi}(s,\ a) \tag{3.23}$$

for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$, where ψ^* is the optimal policy for all states of the system.

Intuitively, the Bellman optimality equation for state–value function needs to have the same expected return for the best action from that state; therefore, another equation for definition (3.4.1) is often used in the literature:

$$V_t^{\psi^*}(s) \overset{\text{def}}{=} \max_{a \in \mathcal{A}(s)} Q_t^{\psi^*}(s,\ a) \tag{3.24}$$

Moreover, there are several other versions for the optimal action–value function but in terms of the state–value function. For that, we must first determine a useful relation:

Proposition 3.4.3 ([DHB20, p. 294] - **state–action relation**)**.** *Consider random inputs $a \in \mathcal{A}$ to the action–value equation. If the first action input a is also drawn from the policy $\psi(a \mid s)$, then there exists the relation*

$$\begin{aligned} V_t^{\psi}(s) &\overset{def}{=} E^{\psi}\Big[G_t \mid S_t = s\Big] \\ &= \sum_{a \in \mathcal{A}} E^{\psi}\Big[G_t \mid S_t = s,\ A_t = a\Big]\ \psi(a \mid s) \\ &= \sum_{a \in \mathcal{A}} Q_t^{\psi}(s,\ a)\ \psi(a \mid s) \\ &= E^{\psi}\Big[Q_t^{\psi}(s,\ a)\Big]. \end{aligned} \tag{3.25}$$

Hence, using (3.24) and (3.25) in equations (3.22) and (3.23), we obtain the variations:

$$\begin{aligned} V_t^{\psi^*}(s) &\overset{(3.24)}{=} \max_{a \in \mathcal{A}(s)} Q_t^{\psi^*}(s,\ a) \\ &\overset{(3.17)}{=} \max_{a}\ E^{\psi^*}\Big[G_t \mid S_t = s,\ A_t = a\Big] \\ &\overset{(3.12)}{=} \max_{a}\ E^{\psi^*}\Big[R_{t+1} + \gamma\ G_{t+1} \mid S_t = s,\ A_t = a\Big] \\ &= \max_{a}\ E\Big[R_{t+1} + \gamma\ V_{t+1}^{\psi^*}(s') \mid S_t = s,\ A_t = a\Big] \\ &= \max_{a} \sum_{s',\ r} p(s',\ r \mid s,\ a)\ \Big[r + \gamma\ V_{t+1}^{\psi^*}(s')\Big], \end{aligned} \tag{3.26}$$

where these equalities (3.26) present extra varieties for the Bellman optimality equation for state–value function,

$$\begin{aligned} Q_t^{\psi^*}(s,\ a) &= E\Big[R_{t+1} + \gamma\ V_{t+1}^{\psi^*}(s') \mid S_t = s,\ A_t = a\Big] \\ &\overset{(3.24)}{=} E\Big[R_{t+1} + \gamma\ \max_{a'}\ Q_{t+1}^{\psi^*}(s', a') \mid S_t = s,\ A_t = a\Big] \\ &= \sum_{s',\ r} p(s',\ r \mid s,\ a)\ \Big[r + \gamma\ \max_{a'}\ Q_{t+1}^{\psi^*}(s',\ a')\Big], \end{aligned} \tag{3.27}$$

where these equalities in (3.27) give different versions of the Bellman optimality equation for action–value function.

Keep in mind that while we consider a finite discrete–state MDP formulation in this section, the same procedures can be applied to a continuous–state MDP model that replaces sums with integrals.

3.5 Value and policy iteration on dynamic programming

Richard Bellman [Bel52] also pioneered the theory underlying classic dynamic programming (DP). This method involves a collection of algorithms that provide a solution for optimal policies when the environment is fully observable and described by a Markov decision process. Dynamic programming is essential for reinforcement learning because the former substantiates almost all of the latter's approaches. Hence, reinforcement learning can be viewed as attempting to achieve much of the same effect as dynamic programming but with less computation and without assuming a perfect environment model. The key idea of a dynamic programming framework is to use the concept of a state–value or an action–value function, or both, which are representations of the quality of policies and/or states. In other words, they are performance metrics. As a result, the state value equation (3.18) can be sorted out by a DP's recursive method that might be more efficient than other techniques that use linear equation solutions. This efficiency is due to the fact that linear equations are much more costly than simple recursive computations. The idea adds an iterator index $\lambda = 1, 2, \ldots$ to enumerate the indices of each iteration. This approach result in, for instance,

$$V_t^{(\lambda)}(s\ ;\ \psi) \longleftarrow E^{\psi}\Big[R_{t+1} + \gamma\ V_{t+1}^{(\lambda)}(S_{t+1}\ ;\ \psi) \mid S_t = s\Big], \tag{3.28}$$

for all $s \in \mathcal{S}$.

See that equation (3.28) returns a performance metric value for each state $s \in \mathcal{S}$ associated with a policy ψ at the iteration step λ. So that, a value function vector $\overrightarrow{\mathbf{v}}_{\lambda}^{\psi}$ related to a policy ψ at the iteration step λ has $|\mathcal{S}|$ number of elements derived from equation (3.28) outputs. Starting with a single arbitrarily chosen policy ψ and some initial guess for $\overrightarrow{\mathbf{v}}_{\lambda}^{\psi} = \overrightarrow{0}$, the iteration continues until convergence at a given tolerance level is reached, or it can run until a certain number of steps are completed. This traditional algorithm receives the name of ***value iteration***, and employs a recursive procedure to directly find the optimal state–value function. The subsequent step–by–step, which was influenced by [Gos15, p. 165], demonstrates the approach more in details:

Step 1. Initialize the iterator counter $\lambda = 1$, and the value function vector with $|\mathcal{S}|$ number of elements containing initial values $\overrightarrow{\mathbf{v}}_{0}^{\psi} = \overrightarrow{0}$. Select an threshold $\epsilon > 0$.

Step 2. Compute the elements of the value function vector $\overrightarrow{\mathbf{v}}_{1}^{\psi}$, such that:

$$V_t^{(1)}(s\ ;\ \psi) \longleftarrow \max_{a \in \mathcal{A}(s)} E^{\psi}\Big[R_{t+1} + \gamma\ V_{t+1}^{(1)}(S_{t+1}\ ;\ \psi) \mid S_t = s\Big],$$

for all $s \in \mathcal{S}$.

Step 3. If

$$||\overrightarrow{\mathbf{v}}_{1}^{\psi} - \overrightarrow{\mathbf{v}}_{0}^{\psi}|| < \epsilon\frac{(1-\gamma)}{2\gamma},$$

go to **Step 4**. Otherwise increment λ by 1 and go back to **Step 2.**

Step 4. For each $s \in \mathcal{S}$, choose

$$\psi^{*}(s) \in \arg\max_{a \in \mathcal{A}(s)} V_t^{(\lambda)}(s\ ;\ \psi).$$

Note that the justification for using of the expression $\frac{(1-\gamma)}{2\gamma}$ is to ensure that the max norm of the difference between vectors converges to ϵ. For further considerations, we refer to [Gos15, Chapter 11].

Another classic DP approach is named as ***policy iteration***. We need two procedure parts to create a policy iteration algorithm: a way to evaluate a given policy (***policy evaluation***) and another to improve that given policy (***policy improvement***). Both can be thought of as sub–algorithms within a more extensive one. Our step–by–step example below, which was inspired by [Gos15, pp. 163–164], illustrates the method:

Step 1. Set an iterator counter $\lambda = 0$. Select an initial arbitrary policy $\psi^{(\lambda=0)}$.

Step 2. **Policy Evaluation**: Solve the linear system of equations

$$V_t^{\psi^{(0)}}(s) \longleftarrow E^{\psi^{(0)}}\Big[R_{t+1} + \gamma\, V_{t+1}^{\psi^{(0)}}(S_{t+1}) \mid S_t = s\Big],$$

for all $s \in \mathcal{S}$.

Step 3. **Policy Improvement**: Choose a new policy, such that

$$\psi^{(1)}(s) \in \arg\max_{a\in\mathcal{A}(s)} V_t^{\psi^{(0)}}(s),$$

for all $s \in \mathcal{S}$.

Step 4. If $\psi^{(1)}(s) = \psi^{(0)}(s)$ for each $s \in \mathcal{S}$, then stop the iteration and set $\psi^{*}(s) = \psi^{(1)}(s)$ for each $s \in \mathcal{S}$. Otherwise, increment λ by 1, and return to **Step 2**.

Furthermore, there is a commonly used term in the literature known as ***generalized policy iteration*** (GPI) [SB19, pp. 86–87], which refers to the general idea of allowing policy–evaluation and policy–improvement to interact regardless of the specifics of these two processes. This GPI concept is crucial because it generally describes many reinforcement learning approaches. In conclusion, reinforcement learning methods have a dynamic that performs a loop between the policy and value function. First, a value function is computed in relation to the given policy (evaluation part). The results of that value function are then used to improve the corresponding policy (improvement part). The **Figure 3.2** depicts the concept at work behind the scenes. As a result, when the evaluation and improvement processes stabilize, i.e., when changes are no longer produced, then the policy become optimal.

Finally, keep in mind that we made an inconspicuous assumption in this section. We assumed that the states of the environment are all discrete and distinct. However, most environments, at least those that are realistic, are not discrete. They are, in fact, continuous. Nonetheless, we do not need to be concerned about that in this early stage. More complex techniques have the potential to solve this problem by acting as function approximators for our policy and value functions. We will go over this in more detail when we get further along with the theory.

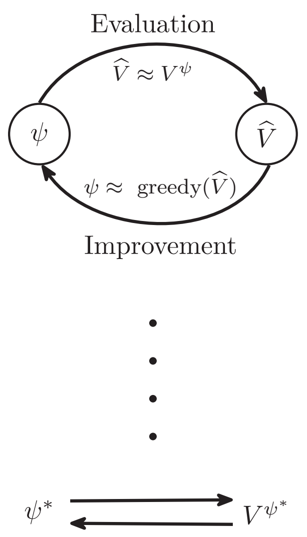
Evaluation
$\widehat{V} \approx V^{\psi}$
ψ
$\widehat{V}$
$\psi \approx$ greedy$(\widehat{V})$
Improvement
ψ^*
V^{ψ^*}

Figure 3.2: Diagram for a generalized policy iteration framework [SB19, p. 86] but with our notations.

3.6 Monte Carlo estimation

Monte Carlo technique is a simulation–based procedure for Markov decision process problem–solving. It makes no assumptions about the environment, i.e., it does not rely on the real transition probabilities (*model–free algorithm*) but can still achieve optimal behavior. Remember that the dynamic programming procedure demands a precise understanding of the transition probabilities to perform iteration steps in policy or value iteration algorithms. Conversely, only sample sequences of states, actions, and rewards from actual or simulated interaction with an environment are required for Monte Carlo approaches. Therefore, it can learn from actual experience, working directly with the definition of the value function. If we consistently draw episodes of T–step trajectory for the return random variable $\widehat{G}_{k:T}$, then we can estimate the action–value function at the state–action pair (s, a) using the empirical mean:

$$\begin{aligned} Q_t^{\psi}(s,\ a) &\approx \frac{1}{T} \sum_{k=1}^{T} E^{\psi}\Big[G_k \mid S_k = s,\ A_k = a\Big] \\ &= \frac{1}{T} \sum_{k=1}^{T} \widehat{Q}_t^{(k)}(s,\ a\ ;\ \psi), \end{aligned} \tag{3.29}$$

where k means each step needed for the system to reach the terminal state T. The ψ notation is interpreted as a followed policy when gathering the actions. This way of deciding the correct actions means that the Monte Carlo method is an ***on–policy*** algorithm. on–policy algorithms exclusively learn an policy from samples if these samples are generated using that same policy. Contrarily, ***off–policy*** algorithms can learn an policy from data yielded by other policies. However, it is beyond the scope of this thesis to examine all of Monte Carlo's variants; for further information, see [SB19].

Another major consideration is that the action–value function should be drawn separately for each state–action combination. So that, for each pairwise combination of $(s,\ a)$,

$$\widehat{Q}_t^{(k)}(s,\ a\ ;\ \psi) \longrightarrow Q_t^{\psi^*}(s,\ a) \text{ as } k \longrightarrow \infty$$

which follows from the strong law of large numbers theory.

Hence, if the Monte Carlo's agent has access to samples of the real–world environment, it can also find the optimal control. In summary, the agent should first generate a T–step trajectory using a policy. Then, the action–value function is estimated using equation (3.29). Following that, the policy improvement step corresponds to the greedy update of the policy iteration:

$$\psi' = \arg \max_{a \in \mathcal{A}(s)} \ \widehat{Q}_t^{(T)}(s,\ a\ ;\ \psi),$$

where ψ' represents a new policy after polity improvement step.

Finally, the new policy is employed to sample a new set of paths of a complete trajectory, and this process repeats until convergence or a predetermined number of repetitions are completed. As can be seen, each of these procedures came from GPI (see **Figure 3.2**) and are extended to the Monte Carlo case, where only sample experience is available.

A vital alternative method instead of waiting for the update of the action–value (or state–value) function after all T–paths are sampled (3.29) is to apply the iterated update rule invented by ***Robbins and Monro*** [RM51] but for the action–value equation case, as follows:

$$\begin{aligned} \widehat{Q}_t^{(k+1)}(s,\ a\ ;\ \psi) &\longleftarrow (1-\eta_k)\ \widehat{G}_k(s,\ a\ ;\ \psi) + \eta_k\ \widehat{Q}_t^{(k)}(s,\ a\ ;\ \psi) \\ &= \widehat{Q}_t^{(k)}(s,\ a\ ;\ \psi) + \eta_k \left[\widehat{G}_k(s,\ a\ ;\ \psi) - \widehat{Q}_t^{(k)}(s,\ a\ ;\ \psi)\right], \end{aligned} \tag{3.30}$$

where $0 < \eta_k < 1$ is the so–called ***learning rate*** or step–size parameter, and $\widehat{G}_k(s,\ a\ ;\ \psi)$ is a function under a policy ψ, at step k, denoting an unbiased estimation for $E^{\psi}\Big[G_k \mid S_k = s,\ A_k = a\Big]$.

If $\eta_k = \frac{1}{(k+1)}$ (decreases), it can be shown that such iterative updating converges to the actual empirical and theoretical averages. This procedure was invented in the early 1950s by Robbins and Monre [RM51], and it is still widely used in reinforcement learning. The most significant advantage of this technique is that it transforms the problem into an online learning environment, where updates are faster and occur after each observed estimation. This algorithm also benefits, even more satisfactorily, other reinforcement learning approaches, such as temporal difference learning, which is the next section's subject.

3.7 Temporal difference with one–step

As previously explained, Monte Carlo processes must wait until the completion of each episode before updating the value function. However, if the T–step trajectory is long, this effort may be tiresome and sluggish. As a solution, another model–free technique termed one–step temporal difference learning

TD(0) has the potential to speed that process up without losing its reliability. This approach updates the value function by waiting for an unique step/path of the entire trajectory, rather than proceeding to the terminal state as Monte Carlo does. Additionally, this method is sometimes referred to as an estimation over an estimation, whereas Monte Carlo estimates the experience of a complete trajectory.

Like Monte Carlo, TD(0) converts the value functions into update equations. Nonetheless, the algorithm merely draws a single estimation of the expected return and then computes an empirical mean with the old estimated value function, such that for the state–value case:

$$\begin{aligned} \widehat{V}_t^{(k+1)}(S_k \ ; \ \psi) \longleftarrow & \widehat{V}_t^{(k)}(S_k \ ; \ \psi) \\ & + \eta_k \left[R_{k+1} + \gamma \ \widehat{V}_t^{(k+1)}(S_{k+1} \ ; \ \psi) - \widehat{V}_t^{(k)}(S_k \ ; \ \psi) \right] \end{aligned} \tag{3.31}$$

where $k = 0$, η_k is the learning rate, and γ is the discount factor. The expression inside the square brackets in the right–hand side of the equation is commonly named TD(0) error, as follows:

$$\delta_k = R_{k+1} \ + \gamma \ \widehat{V}_t^{(k+1)}(S_{k+1} \ ; \ \psi) - \widehat{V}_t^{(k)}(S_k \ ; \ \psi). \tag{3.32}$$

This approach is deemed to be online and extremely quick, but it comes with a highly volatile trade–off. Because TD(0) comprises less information than the mean of a whole episode, significant variation is inescapable. Taking this issue into consideration and attempting to resolve it, TD(K), for $K > 0$, permits K–step updates and may therefore be utilized as a bootstrapped online algorithm. Several varieties of temporal difference learning apply one or multiple–step updates on state–value or action–value functions. We will examine this topic deeply in the next section.

3.8 Temporal difference with K–step

This section introduces K–step temporal difference TD(K) bootstrapping procedures that allow an intermediary technique between Monte Carlo and TD(0) methods. On one side, Monte Carlo approaches update the state–value equation relying on the whole sequence of observed rewards from the given initial state until the terminal state. On the other side, TD(0) updates the state–value equation depending exclusively on the next reward, i.e., updates after each state transition and reward, regardless of whether the subsequent state is terminal or not. Thus, one intermediate approach would be to update the state–value equation depending on an intermediate amount of state transitions and rewards, greater than one but less than all of them, until a terminal state. A backup diagram illustrating these explanations is shown in **Figure 3.3**.

Technically, a Markov's cumulative return representation is

$$\begin{aligned} G_{k:T} &\overset{\text{def}}{=} \gamma^0 \ R_{k+1} + \gamma^1 \ R_{k+2} + \gamma^2 \ R_{k+3} + \ldots + \gamma^{T-k-1} R_T \\ &= R_{k+1} + \gamma \ G_T \\ &= R_{k+1} + \gamma \ \widehat{V}_{t+1}^{(T)}(S_T \ ; \ \psi), \end{aligned} \tag{3.33}$$

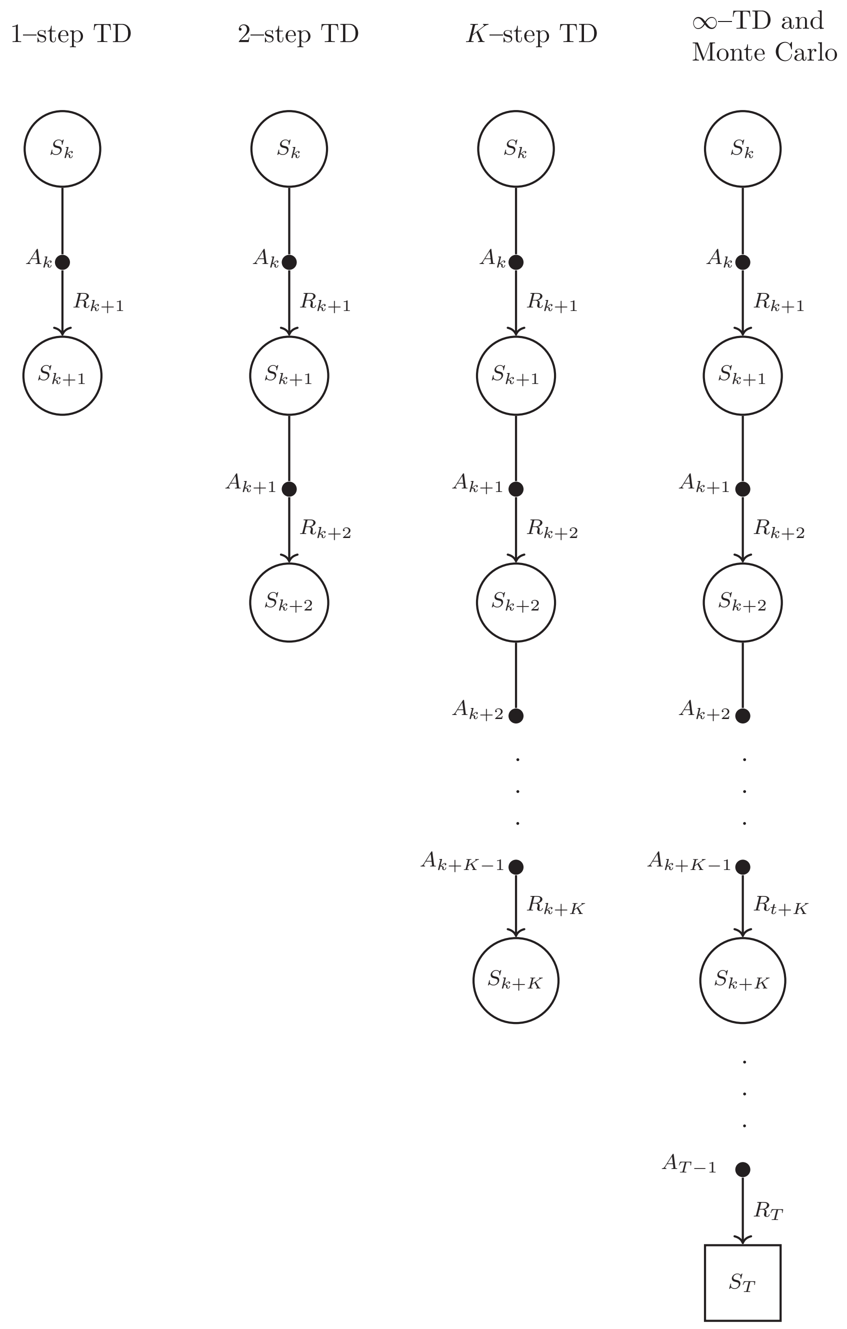

Figure 3.3: Diagram illustrating the difference between 1–step TD, 2–step TD, K–step TD, and Monte Carlo methods [SB19, p. 142].

where the terminal state T represents the last step of the entire trajectory/episode. Then, following the same idea in (3.33), a two–step return is, for example,

$$\begin{aligned} G_{k:k+2} &\overset{\text{def}}{=} \gamma^0\ R_{k+1} + \gamma^1\ R_{k+2} \\ &= R_{k+1} + \gamma\ G_{k+2} \\ &= R_{k+1} + \gamma\ \widehat{V}_{t+1}^{(k+2)}(S_{k+2}\ ;\ \psi), \end{aligned}$$

and for more generalized K–step

$$G_{k:k+K} \overset{\text{def}}{=} R_{k+1} + \gamma\ \widehat{V}_{t+1}^{(k+K)}(S_{k+K}\ ;\ \psi),$$

for all $K \geq 1$ and $0 \leq k < T - K$.

All these K–step cumulative returns $G_{k:k+K}$ are approximations to the entire return $G_{t:T}$, trimmed in K steps. Additionally, if $k + K$ is an index that is greater than the terminal state, then all rewards beyond the terminal state are assumed to be zero. As a result, when K–step returns are used, the state–value learning function becomes

$$V_t^{(k+1)}(S_k) \overset{\text{def}}{\longleftarrow} V_t^{(k)}(S_k) + \eta_k\ \Big[G_{k:k+K} - V_t^{(k)}(S_k)\Big], \tag{3.34}$$

and the state–action–value learning function

$$Q_t^{(k+1)}(S_k,\ A_k) \overset{\text{def}}{\longleftarrow} Q_t^{(k)}(S_k,\ A_k) + \eta_k\ \Big[G_{k:k+K} - Q_t^{(k)}(S_k,\ A_k)\Big], \tag{3.35}$$

where $0 \leq k < K$ for both equations.

3.9 On–policy and off–policy optimal controls

In the previous three sections, we mainly focused on the evaluation or prediction component of the value equation iteration. Now, let us focus our research on the control mechanism. As is typical, a control problem seeks to determine the optimal policy for the system under consideration. Before that, we again take into account the absence of knowledge about the actual transition probabilities of the environment, indicating that we are still in the model–free universe. Consequently, a tabular solution needs to be employed in the evaluation part. Following that, a control mechanism needs to be selected, and we are going to operate with two variety types: on–policy and off–policy.

On the one side, SARSA is an example of a method that has been extensively investigated for the first type, which is an on–policy control based on the assumption of a policy picks up an action. This control policy ψ is known to be an ϵ–greedy, as defined by the following:

$$\psi(S_{k+1}) = \begin{cases} \arg\max_{a' \in \mathcal{A}(S_{k+1})} \widehat{Q}_t^{(k+1)}(S_{k+1},\ a') & \text{with probability } 1-\epsilon, \\ U_k\Big(\mathcal{A}(S_{k+1})\Big) & \text{with probability } \epsilon, \end{cases} \tag{3.36}$$

where U_t randomly draws a uniformly distributed action from the set originated from $\mathcal{A}(S_{k+1})$. Therefore, the choice for a future action is always gathered from that policy. Hence, this notion becomes the following:

$$\widehat{Q}_t^{(k+1)}(S_k,\ A_k\ ;\ \psi) \longleftarrow \widehat{Q}_t^{(k)}(S_k,\ A_k\ ;\ \psi) + \eta_k\ \Big[R_{k+1} + \gamma\ \widehat{Q}_t^{(k+1)}\Big(S_{k+1},\ \psi(S_{k+1})\Big) - \widehat{Q}_t^{(k)}(S_k,\ A_k\ ;\ \psi)\Big], \tag{3.37}$$

for all $a' \in \mathcal{A}(S_{k+1})$ and $0 \leq k < K$.

This update practice operates for the quintuple $(S_k, A_k, R_{k+1}, S_{k+1}, A_{k+1})$, which symbolizes the algorithm's name (SARSA) and comprises an *episode* transition from one state–action pair to the next pair:

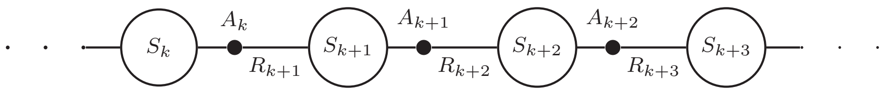

Figure 3.4: Diagram for SARSA's episodes [SB19, p. 129].

It is important to mention that this update rule is executed after each transition from a non–terminal state S_k, and if S_{k+1} is terminal, then the state–action equation is zero.

On the other side, Q–Learning is an example of control technique for the second type. Watkins invented this off–policy TD–learning method in his PhD dissertation in 1989 [Wat89]. This approach also assumes a model–free controlled Markov process with an agent as a controller employs the evaluation step with the action–value equation (3.19), and the control part by choosing an future action exclusively from a greedy mechanism, as follows:

$$\widehat{Q}_t^{(k+1)}(S_k,\ A_k\ ;\ \psi) \longleftarrow \widehat{Q}_t^{(k)}(S_k,\ A_k\ ;\ \psi) + \eta_k\ \Big[R_{k+1} + \gamma \max_{a' \in \mathcal{A}(S_{k+1})} \widehat{Q}_t^{(k+1)}(S_{k+1},\ a') - \widehat{Q}_t^{(k)}(S_k,\ A_k\ ;\ \psi)\Big], \tag{3.38}$$

for all $a' \in \mathcal{A}(S_{k+1})$ and $0 \leq k < K$.

In summary, the fundamental difference between on–policy SARSA and off–policy Q–learning is in terms of the online mechanism for choosing future actions throughout the learning process. For example, in SARSA, subsequent actions $a' \in \mathcal{A}(S_{k+1})$ adhere to the ϵ–greedy policy. Q–learning, in its turn, ensures that the future actions are always the ones that yield the highest value for its value function.

CHAPTER 4

Solutions based on function approximation

This chapter broadens our research to include methods for approximating functions. We extend the tabular approaches provided in the previous chapter to issues with arbitrarily large state space. Furthermore, we frequently address partially observable problems where some states reached during many of our tasks are unfamiliar to us or even to the process, meaning that there are no records for assessment. Therefore, to make rational choices in such states, one must generalize from earlier encounters with states that are comparable in some way to the current one. In such instances, even in a hypothetical scenario with unlimited time, data, and computer power, we cannot expect to find an optimal policy or value function. Instead, we want to obtain a decent approximation while operating restricted computing resources, and so, an inevitable question arises:

How could we derive a generalized approximation across a significantly extensive and even partially observable set if only a small portion of the set is available?

To answer this issue, we combine reinforcement learning with any existing generalization theory. Typically, the form of generalization used in this type of problem is function approximation, which takes examples from the desired function (in our case, the value function) and attempts to make generalizations from them to obtain an estimate of the whole function. Function approximation is often utilized in the field of supervised learning for linear and nonlinear problems and encompasses a wide variety of techniques and applications. While all of these strategies are theoretically applicable to reinforcement learning, several do not fit easily in practice. Thus, the following sections will go into further detail regarding this concept and any difficulties that may arise.

4.1 A weight parameter and an objective metric

Function approximated reinforcement learning estimates the value functions, that is, approximates $Q^{\psi}(s,\ a)$ or $V^{\psi}(s)$. Now, these equations are represented in a parameterized functional form with a weight vector $\overrightarrow{w} \in \mathbb{R}^d$. For example, the notation $\widehat{V}(s;\ \overrightarrow{w}) \approx V^{\psi}(s)$ represents the approximate value of state s, given the weight vector $\overrightarrow{w}$. Bear in mind that the dimensionality of $\overrightarrow{w}$ is typically less than the number of states, and thus altering one weight impacts the projected values of numerous other states. Consequently, when the value function is updated for a single state, this modification affects many other states, and according to [SB19, p. 197], "*such generalization makes the learning potentially more powerful but also potentially more difficult to manage and understand*".

Unlike the tabular approaches, the learned value function here is never equal to the actual one. Undoubtedly, a change to one state has a ripple effect on many others, and it is impossible to control all states' values accurately. Thus, quantifying how much we care about the inaccuracy in each state is vital, and the conventional way to achieve this, based on the literature, is to employ the ***mean squared error*** shown as follows:

$$MSE(\overrightarrow{w}) \overset{\text{def}}{=} \sum_{s \in \mathcal{S}} \mu(s) \sum_{a \in \mathcal{A}(s)} \psi(a|s) \left[Q^{\psi}(s,\ a) - \widehat{Q}(s,\ a;\ \overrightarrow{w})\right]^2, \tag{4.1}$$

where $\mu(s)$ is a state distribution with $\mu(s) \geq 0$ and $\sum_{s \in \mathcal{S}} \mu(s) = 1$.

Additionally, a metric assessing the degree to which approximation values deviate from genuine values, the ***root mean square deviation***, is formulated as:

$$RMSD(\overrightarrow{w}) \overset{\text{def}}{=} \sqrt{MSE(\overrightarrow{w})}. \tag{4.2}$$

The intuition behind the usage of this metric is to minimize the estimation error to identify the global optimum, "*for which* $MSE(\overrightarrow{w}^*) < MSE(\overrightarrow{w})$ *for all possible* $\overrightarrow{w}$" [SB19, p. 200]. While that is the primary purpose of our algorithms, it is not always attainable, resulting in obtaining the local optimum rather than the global optimum, "*for which* $MSE(\overrightarrow{w}^*) < MSE(\overrightarrow{w})$ *for all possible* $\overrightarrow{w}$ *in some neighborhood of* $\overrightarrow{w}^*$" [SB19, p. 200].

4.2 Stochastic gradient descent

Stochastic gradient descent (SGD) is a well–established method for approximating functions in supervised learning and is perfectly suitable for online reinforcement learning, although with some concerns. This section will delve into its theory and application to make it appropriate for reinforcement learning circumstances.

The first parameter to examine is the weight vector, which is composed of real–valued components in the following manner:

$$\overrightarrow{w}_t \overset{\text{def}}{=} \left[w_1, w_2, \ldots, w_d\right]^T, \tag{4.3}$$

for $j = 1, 2, \ldots, d$, where T after the square brackets indicates that the weight vector is transposed, denoting a column vector.

The objective is to develop a weighted linear function capable of approximating the value function in such a way that

$$\widehat{V}(S_t;\ \overrightarrow{w}_t) \longrightarrow V^{\psi}(S_t) \qquad \text{as} \qquad t \longrightarrow \infty, \tag{4.4}$$

for state–value function, and

$$\widehat{Q}(S_t,\ A_t;\ \overrightarrow{w}_t) \longrightarrow Q^{\psi}(S_t,\ A_t) \qquad \text{as} \qquad t \longrightarrow \infty, \tag{4.5}$$

for state–action–value function.

These estimate functions must be differentiable for all $s \in \mathcal{S}$, and $a \in \mathcal{A}$ (exclusively for state–action–value function), with respect to w_j. In this scenario, the SGD technique attempts to decrease inaccuracy in estimations, e.g., to minimize the MSE metric (4.1), by adjusting the weight vector after each iteration step t, such that

$$\begin{aligned} \overrightarrow{w}_{t+1} &\overset{\text{def}}{=} \overrightarrow{w}_t - \frac{1}{2}\,\eta_t\,\overrightarrow{\nabla}\Big[Q^{\psi}(S_t,\ A_t) - \widehat{Q}(S_t,\ A_t;\ \overrightarrow{w}_t)\Big]^2 \\ &= \overrightarrow{w}_t + \eta_t\,\Big[Q^{\psi}(S_t,\ A_t) - \widehat{Q}(S_t,\ A_t;\ \overrightarrow{w}_t)\Big]\,\overrightarrow{\nabla}\widehat{Q}(S_t,\ A_t;\ \overrightarrow{w}_t), \end{aligned} \tag{4.6}$$

where η_t is the learning rate at iteration step t, $\overrightarrow{\nabla}[\cdot]$ represents the gradient vector for the objective metric function seen in (4.1), and $\overrightarrow{\nabla}\widehat{Q}(S_t,\ A_t;\ \overrightarrow{w}_t)$ is the gradient vector for the function $\widehat{Q}(S_t,\ A_t;\ \overrightarrow{w}_t)$ that can be computed by:

$$\overrightarrow{\nabla}\widehat{Q}(S_t,\ A_t;\ \overrightarrow{w}_t) = \left[\frac{\partial\widehat{Q}(S_t,\ A_t;\ \overrightarrow{w}_t)}{\partial w_1}, \ldots, \frac{\partial\widehat{Q}(S_t,\ A_t;\ \overrightarrow{w}_t)}{\partial w_d}\right]^T. \tag{4.7}$$

It is critical to note that while we will continue to use the state–action–value function as an algebraic illustration for the remainder of the theory, keep in mind that the same holds for the state–value function.

Regarding convergence outcomes for SGD, it is assumed that the learning rate η_t gradually diminishes at each iteration step t. According to the theory in [RM51], if η_t declines satisfying criteria

$$\sum_{t=1}^{\infty} \eta_t = \infty \qquad \text{and} \qquad \sum_{t=1}^{\infty} \eta_t^2 < \infty, \tag{4.8}$$

then the precision error for the estimations of (4.6) converges to a local optimum (see [SB19]).

All of this theory is highly beneficial if we know the actual value function $V^{\psi}(S_t)$ or $Q^{\psi}(S_t,\ A_t)$. For example, this scenario is valid for supervised learning, as the algorithm is typically fed with input and output data samples and can infer from authentic value functions. Regrettably, this is not the case in here because the target value function is frequently unknown. With this issue in mind, we must look for the scenario where $Q^{\psi}(S_t,\ A_t)$ is unknown, implying that the update in (4.6) is not possible. Nevertheless, we may substitute the ***target value function*** with an unbiased estimate, represented here by θ_t. Note that if θ_t is an unbiased estimate for the state–action–value equation, then

$$Q^{\psi}(S_t,\ A_t) \approx E^{\psi}\Big[\theta_t | S_t = s,\ A_t = a\Big],$$

and the equation (4.6) turns to be

$$\overrightarrow{w}_{t+1} \overset{\text{def}}{=} \overrightarrow{w}_t + \eta_t\,\Big[\theta_t - \widehat{Q}(S_t,\ A_t;\ \overrightarrow{w}_t)\Big]\,\overrightarrow{\nabla}\widehat{Q}(S_t,\ A_t;\ \overrightarrow{w}_t), \tag{4.9}$$

which is guaranteed to converge to a local optimum if (4.8) holds.

For illustration and according to the Monte Carlo method's formulation (3.29, where $K = 1$), the actual value of a non-terminal state is approximated to the expected value of the subsequent return,

$$Q^{\psi}(S_t,\ A_t) \approx E^{\psi}\Big[G_t | S_t = s,\ A_t = a\Big],$$

with G_t being an unbiased estimate for the unknown $Q^{\psi}(S_t,\ A_t)$; hence, equation (4.6) can be expressed as

$$\overrightarrow{w}_{t+1} \overset{\text{def}}{=} \overrightarrow{w}_t + \eta_t \left[G_t - \widehat{Q}(S_t,\ A_t;\ \overrightarrow{w}_t)\right] \overrightarrow{\nabla}\widehat{Q}(S_t,\ A_t;\ \overrightarrow{w}_t). \tag{4.10}$$

On the other hand, bootstrapped and temporal difference scenarios, like bootstrapped Monte Carlo, TD(0), and TD(K), are not compatible with stochastic gradient descent [Bar93] because the convergence's technical criteria cannot be assured. They do, however, converge in a weaker form in ***semi–gradient*** instances, which were first described by Richard S. Sutton in [Sut84; Sut88], and recently in [SB19]. According to [SB19, p. 202], while semi–gradient situations may not converge strongly, they do converge consistently when the estimate value function is linearly combined with a weight vector $\overrightarrow{w}_t$ and a feature vector (this will be defined in the subsequent section). Moreover, such scenarios also offer substantial benefits, such as considerably quicker continuous and online learning. Consequently, we shall illustrate that concept with an example of one–step temporal difference's semi–gradient descent that makes use of

$$\begin{aligned}\theta_t &= G_t \\ &= R_{t+1} + \gamma G_{t+1} \\ &= R_{t+1} + \gamma \widehat{Q}(S_{t+1},\ A_{t+1};\ \overrightarrow{w}_t)\end{aligned} \tag{4.11}$$

as their estimate, and

$$\begin{aligned}\overrightarrow{w}_{t+1} \overset{\text{def}}{=} \overrightarrow{w}_t + \eta_t \Big[&R_{t+1} + \gamma\widehat{Q}(S_{t+1},\ A_{t+1};\ \overrightarrow{w}_t) \\ &- \widehat{Q}(S_t,\ A_t;\ \overrightarrow{w}_t)\Big]\ \overrightarrow{\nabla}\widehat{Q}(S_t,\ A_t;\ \overrightarrow{w}_t),\end{aligned} \tag{4.12}$$

becomes variation of their weight update (4.6), which converges to a point near the local optimum, the ***TD fixed point*** (see [SB19, pp. 205–2010] for proof and further details). Similarly, the equation for a semi–gradient K–step temporal difference is given by

$$\begin{aligned}\overrightarrow{w}_{t+K+1} \overset{\text{def}}{=} \overrightarrow{w}_{t+K} + \eta_{t+K}\ &\left[G_{t:t+K+1} - \widehat{Q}(S_t,\ A_t;\ \overrightarrow{w}_{t+K})\right] \\ &\overrightarrow{\nabla}\widehat{Q}(S_t,\ A_t;\ \overrightarrow{w}_{t+K}),\end{aligned} \tag{4.13}$$

for the K–step return

$$\begin{aligned}G_{t:t+K+1} \overset{\text{def}}{=} &\gamma^0\ R_{t+1} + \gamma^1\ R_{t+2} + \ldots + \gamma^K\ R_{t+K+1} \\ &+ \gamma^{K+1}\ \widehat{Q}(S_{t+K+1}, A_{t+K+1};\ \overrightarrow{w}_{t+K}),\end{aligned} \tag{4.14}$$

where $0 \leq t < T - K$, and T denotes the step–time for the terminal state.

4.3 Feature vector and basis function

As mentioned previously, a semi–gradient situation converges when the estimate function, estimate value function in our case, is a linear combination of weights and features. These features are included in a vector, ***feature vector***, that express a state–action pair (S_t, A_t). The constituents of a feature vector are characterized by a fixed expression, named as ***basis function***, that rely on the state and action variables' values. Generally speaking, the ideal technique is to specify a fixed basis function first and then describe an estimate value function as a function of that basis function. In this type of architecture, the coefficients of a basis function are the weights in the weight vector. Then, the linearly combined weights and features form the approximate value function.

Let us define a ***feature vector*** as $\overrightarrow{\phi}_t \in \mathbb{R}^d$, which has the same number of dimensions d as in $\overrightarrow{w}_t$, such that

$$\overrightarrow{\phi}_t \overset{\text{def}}{=} \Big[\phi_1(S_t,\ A_t), \phi_2(S_t,\ A_t), \ldots, \phi_d(S_t,\ A_t)\Big]^T, \tag{4.15}$$

for $i = 1, 2, \ldots, d$, where T after the square brackets indicates that the vector is transposed, being a column vector, and each component of the feature vector is a mapping $\phi_i : \mathcal{S} \times \mathcal{A} \longrightarrow \mathbb{R}$. Hence, the estimate value function is given by the inner vector product between the weight vector and the feature vector:

$$\begin{aligned}\widehat{Q}(S_t,\ A_t;\ \overrightarrow{w}_t) &\overset{\text{def}}{=} \overrightarrow{w}_t^T\ \overrightarrow{\phi}_t \\ &= \sum_{i,j=1}^{d} w_j\ \phi_i(S_t, A_t),\end{aligned} \tag{4.16}$$

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

Sutton and Barto [SB19, p. 205] formalize the mathematical terminology underlying feature vectors' components as "*basis functions because they form a linear basis for the set of approximate functions. Constructing d–dimensional feature vectors to represent states is the same as selecting a set of d basis functions.*"

With all of these aspects in mind, and particularly with the fact that we are now interacting over a value function that is a linear weighted estimate function, determining their gradient becomes trivial:

$$\overrightarrow{\nabla}\widehat{Q}(S_t,\ A_t;\ \overrightarrow{w}_t) \overset{\text{def}}{=} \overrightarrow{\phi}_t, \tag{4.17}$$

because

$$\begin{aligned}\overrightarrow{\nabla}\widehat{Q}(S_t, A_t;\ \overrightarrow{w}_t) &\overset{(4.7)}{=} \left[\frac{\partial\widehat{Q}(S_t, A_t;\ \overrightarrow{w}_t)}{\partial w_1}, \ldots, \frac{\partial\widehat{Q}(S_t, A_t;\ \overrightarrow{w}_t)}{\partial w_d}\right]^T \\ &\overset{(4.16)}{=} \left[\frac{\partial}{\partial w_1}\Big[w_1 \cdot \phi_1(S_t, A_t)\Big], \ldots, \frac{\partial}{\partial w_d}\Big[w_d \cdot \phi_d(S_t, A_t)\Big]\right]^T \\ &= \Big[\phi_1(S_t, A_t), \ldots, \phi_d(S_t, A_t)\Big]^T \\ &= \overrightarrow{\phi}_t.\end{aligned}$$

Notice now that the weight's update routine, given by equation (4.9), yields

$$\overrightarrow{w}_{t+1} \overset{\text{def}}{=} \overrightarrow{w}_t + \eta_t\ \Big[\theta_t - \widehat{Q}(S_t,\ A_t;\ \overrightarrow{w}_t)\Big]\ \overrightarrow{\phi}_t, \tag{4.18}$$

which is guaranteed to converge to a point near the local minimum under the learning rate's usual conditions (see 4.8).

It is worth noting again that while we utilized the state–action–value function $\widehat{Q}(S_t,\ A_t; \overrightarrow{w}_t) \approx Q^\psi(S_t,\ A_t)$, as a representative parameterized function, the same extends and holds for the state–value function $\widehat{V}(S_t; \overrightarrow{w}_t) \approx V^\psi(S_t)$.

4.4 Linearly weighted on–policy SARSA

In this section, we proceed with the parameterized version of the state–action value function

$$\widehat{Q}(S_t,\ A_t;\ \overrightarrow{w}_t) \approx Q^\psi(S_t,\ A_t), \tag{4.19}$$

where $\overrightarrow{w}_t \in \mathbb{R}^d$ is a finite d–dimensional weight vector at time t.

Now, let the update target value function θ_t in (4.9) be any approximation of (4.19) (right–hand side) with SARSA's one–step method represented as

$$\begin{aligned}\overrightarrow{w}_{t+1} \overset{\text{def}}{=} \overrightarrow{w}_t + \eta_t \Big[R_{t+1} + \gamma\ \widehat{Q}(S_{t+1},\ A_{t+1};\ \overrightarrow{w}_t)\\ - \widehat{Q}(S_t,\ A_t;\ \overrightarrow{w}_t)\Big] \overrightarrow{\nabla} \widehat{Q}(S_t,\ A_t;\ \overrightarrow{w}_t),\end{aligned} \tag{4.20}$$

which is called in the literature [SB19, p. 244] as an ***episodic semi–gradient one–step SARSA***, and the same reference guarantees this system's convergence property. The assumption behind the scenario here is that

$$\theta_t = R_{t+1} + \gamma\ \widehat{Q}(S_{t+1},\ A_{t+1};\ \overrightarrow{w}_t), \tag{4.21}$$

justified by the combination between (4.19) and one of the definitions' variation of state–action–value function in (3.19), as in the following manner:

$$Q^\psi(S_t,\ A_t) \approx E^\psi\Big[R_{t+1} + \gamma\ \widehat{Q}(S_{t+1},\ A_{t+1};\ \overrightarrow{w}_t) \mid S_t = s,\ A_t = a\Big].$$

Since we have already established in (4.16) the evaluation part, to build the control phase, we must combine such a state–action–value estimation (4.19) with an action selection, and a policy improvement approaches.

For the action selection, the estimate value function is used to evaluate each possible action $a \in \mathcal{A}(S_t)$ available in the current state S_t. Then, we attain the chosen action a by the ϵ–***greedy policy*** $\psi(S_t|A_t)$, such that

$$\psi(S_t|A_t) = \begin{cases} \arg\max_{a\in A(S_t)} \widehat{Q}(S_t,\ a;\ \overrightarrow{w}_t) & \text{with probability } 1-\epsilon,\\ U_t\Big(\mathcal{A}(S_t)\Big) & \text{with probability } \epsilon, \end{cases} \tag{4.22}$$

where U_t randomly draws a uniformly distributed action from $\mathcal{A}(S_t)$.

For policy improvement, we wait for the environment's output S_{t+1} after action decision ψ has been triggered in the environment; we then compute the R_{t+1} and so employ the (4.20) to update the estimate value function's coefficients. In this final step, also called weights' update, note that $\widehat{Q}(S_{t+1},\ A_{t+1};\ \overrightarrow{w}_t)$ is computed by repeating the ϵ–greedy policy (4.22). The accompanying pseudocode (see **Algorithm 1**) illustrates the whole process.

Algorithm 1 shows an episodic semi–gradient one–step SARSA because its weights are updated in every episode's step, as discussed before. Moreover, note that if S_t is a terminal state (like in line 10), it has a state–action–value equal to zero, i.e., $\widehat{Q}(S_{\text{Terminal}},\ A_t;\ \overrightarrow{w}) = 0$.

Algorithm 1: Episodic semi–gradient one–step SARSA [SB19, p. 244].

```
1  begin
2      By defining and/or initializing
           • a parameterized and differentiable $\widehat{Q} : \mathcal{S} \times \mathcal{A} \times \mathbb{R}^d \longrightarrow \mathbb{R}$.
           • a learning rate $0 < \eta \leq 1$.
           • a greedy parameter $0 < \epsilon << \leq 1$.
           • a discount rate $0 < \gamma \leq 1$.
           • the weight vector $\overrightarrow{w} \in \mathbb{R}^d$ (e.g., $\overrightarrow{w} = \overrightarrow{0}$).
3  while exists episodes do
4      for each episode's step $t = 0, 1, 2, \ldots$ do
5          $S_t \longleftarrow$ Input from the Environment ($S_0 \neq$ Terminal).
6          $A_t \longleftarrow$ e.g., $\epsilon$–greedy policy w.r.t. $\widehat{Q}(S_t,\ a;\ \overrightarrow{w})$.
7          $A_t \longrightarrow$ Output to the Environment.
8          $S_{t+1} \longleftarrow$ Input from the Environment.
9          $R_{t+1} \longleftarrow$ Compute and store.

10         if $S_{t+1}$ is terminal then
11             $\overrightarrow{w} \longleftarrow \overrightarrow{w} + \eta\Big[R_{t+1} - \widehat{Q}(S_t,\ A_t;\ \overrightarrow{w})\Big]\overrightarrow{\nabla}\widehat{Q}(S_t,\ A_t;\ \overrightarrow{w})$.
12             $\eta \longleftarrow$ small reduction of the learning rate.
13             BREAK (go to the next episode).
14         else
15             $A_{t+1} \longleftarrow$ e.g., $\epsilon$–greedy policy w.r.t. $\widehat{Q}(S_{t+1},\ a;\ \overrightarrow{w})$.
16             $\overrightarrow{w} \longleftarrow \overrightarrow{w} + \eta\Big[R_{t+1} + \gamma\widehat{Q}(S_{t+1},\ A_{t+1};\ \overrightarrow{w})$
17                         $-\widehat{Q}(S_t,\ A_t;\ \overrightarrow{w})\Big]\overrightarrow{\nabla}\widehat{Q}(S_t,\ A_t;\ \overrightarrow{w})$
18             $t \longleftarrow t + 1$ (go to the next episode's step).
```

In comparison, another SARSA variation called ***episodic semi–gradient*** ***K–step SARSA*** employs the same general gradient–descent update for state–action–value prediction (4.9), but with slight modifications regarding θ_t. In this instance, the K–step return generalizes from

$$\begin{aligned} G_{t:t+K+1} &\overset{\text{def}}{=} \gamma^0 R_{t+1} + \gamma^1 R_{t+2} + \ldots + \gamma^K R_{t+K+1} \\ &\quad + \gamma^{K+1}\widehat{Q}(S_{t+K+1},\ A_{t+K+1};\ \overrightarrow{w}_{t+K}), \end{aligned}$$

for $0 \leq t < T - K$, where T is the time–step for the terminal state, and

$$G_{t:t+K+1} \overset{\text{def}}{=} G_t$$

if $t \geq T - K$. Hence, the target θ_t turns to be $G_{t:t+K+1}$ because

$$\widehat{Q}(S_{t+1},\ A_{t+1};\ \overrightarrow{w}_t) = E^{\psi}\Big[G_{t:t+K+1} \mid S_{t+1} = s',\ A_{t+1} = a'\Big],$$

with accordance to the theory discussed in the preceding section, and so the updates of the K–step weight become

$$\begin{aligned}\overrightarrow{w}_{t+K+1} \overset{\text{def}}{=} \overrightarrow{w}_{t+K} + \eta_{t+K}\Big[G_{t:t+K+1} - \widehat{Q}(S_t,\ A_t;\ \overrightarrow{w}_{t+K})\Big] \\ \overrightarrow{\nabla}\widehat{Q}(S_t,\ A_t;\ \overrightarrow{w}_{t+K}),\end{aligned} \tag{4.23}$$

for $0 \leq t < T - K$. The pseudocode for this situation is included in **Algorithm 2**.

Finally, we could not leave out SARSA's variations for continuing tasks, but before that, we must mention a problematic technicality that arises with a discounted setting with function approximation and on–policy scenarios. When returns from each state can be computed and averaged individually, the continuing discounted technique has been highly efficacious in the tabular case. However, whether or not to employ this formulation in the function approximation method is debated. This argument arises because we lack distinctly defined states and instead have characteristics (features) representing states or groupings. Additionally, performance is evaluated almost solely utilizing reward sequences and actions, computed by averaging the returns across time. Recall that we do not have a beginning or end state, nor a particular time step, due to the continuing nature of the inputs. Therefore, "*the average of the discounted returns is proportional to the average rewards*" [SB19, p. 253], which gives us precisely the same outcomes, i.e., the discount factor K "*would have no effect*" (see [SB19, pp. 253–254] for more details and proof).

With those particulars in mind, the continuing problems affect the average instead of discounted reward setting. Moreover, these problems interact between agent and environment permanently without a well–defined ending or starting states. It is critical to note that the system we are describing contains an ergodicity (2.1.8) assumption, which is capable of ensuring the validity of the limits in the equation (4.24). Consequently, the average reward can be defined as

$$r(\psi) \overset{\text{[SB19, p. 249]}}{=} \lim_{t \longrightarrow \infty} E\Big[R_{t+1} \mid S_{0:t} = s,\ A_{0:t} \sim \psi\Big], \tag{4.24}$$

where $A_{0:t} \sim \psi$ means that the actions $\{A_0, A_1, ..., A_{t-1}, A_t\}$ are taken in accordance with the policy ψ.

As a result, the computation of returns is defined as the difference between individual rewards and the average reward.

$$G_t \overset{\text{[SB19, p. 250]}}{=} R_{t+1} - r(\psi) + R_{t+2} - r(\psi) + \dots \tag{4.25}$$

which is known in the literature as ***differential return*** [SB19, p. 250]. The corresponding ***differential value functions*** follow the same definitions and notations in (3.16) and (4.4) for state–value, and (4.5) for state–action value. The ***differential TD–error*** (3.32) form for state–value and state–action–value are respectively

$$\delta_t \overset{\text{def}}{=} R_{t+1} - \overline{R}_{t+1} + \widehat{V}(S_{t+1};\ \overrightarrow{w}_t) - \widehat{V}(S_t;\ \overrightarrow{w}_t), \tag{4.26}$$

$$\delta_t \overset{\text{def}}{=} R_{t+1} - \overline{R}_{t+1} + \widehat{Q}(S_{t+1}, A_{t+1};\ \overrightarrow{w}_t) - \widehat{Q}(S_t, A_t;\ \overrightarrow{w}_t), \tag{4.27}$$

Algorithm 2: Episodic semi–gradient K–step SARSA [SB19, p. 247].

```
1  begin
2      By defining and/or initializing
           • a parameterized and differentiable $\widehat{Q} : \mathcal{S} \times \mathcal{A} \times \mathbb{R}^d \longrightarrow \mathbb{R}$.
           • a number of episode's steps $K > 0$.
           • a learning rate $0 < \eta \leq 1$ for weight's update.
           • a greedy parameter $0 < \epsilon <\leq 1$.
           • a discount rate $0 < \gamma \leq 1$.
           • a weight vector $\overrightarrow{w} \in \mathbb{R}^d$ (e.g., $\overrightarrow{w} = \overrightarrow{0}$).
3  while exists episodes do
4      $T \longleftarrow \infty$.
5      for each episode's step $t = 0, 1, 2, \dots$ do
6          $S_t \longleftarrow$ Input from the Environment ($S_0 \neq$ Terminal).
7          $A_t \longleftarrow$ e.g., $\epsilon$–greedy w.r.t. $\widehat{Q}(S_t,\ a;\ \overrightarrow{w})$.
8          if $t < T$ then
9              $A_t \longrightarrow$ Output to the Environment.
10             $S_{t+1} \longleftarrow$ Input from the Environment.
11             $R_{t+1} \longleftarrow$ Compute and store.
12             if $S_{t+1}$ is terminal then
13                 $T \longleftarrow t + 1$.
14             else
15                 $A_{t+1} \longleftarrow$ e.g., $\epsilon$–greedy w.r.t. $\widehat{Q}(S_{t+1},\ a;\ \overrightarrow{w})$.
16         $\tau \longleftarrow t - K + 1$.
17         if $\tau \geq 0$ then
18             $G \longleftarrow \sum_{i=\tau+1}^{\min(\tau+K,T)} \gamma^{i-\tau-1} R_i$.
19             if $\tau + K < T$ then
20                 $G \longleftarrow G + \gamma^K \widehat{Q}(S_{\tau+K},\ A_{\tau+K};\ \overrightarrow{w})$.
21             $\overrightarrow{w} \longleftarrow \overrightarrow{w} + \eta\Big[G - \widehat{Q}(S_\tau,\ A_\tau;\ \overrightarrow{w})\Big]\overrightarrow{\nabla}\widehat{Q}(S_\tau,\ A_\tau;\ \overrightarrow{w})$.
22             $\eta \longleftarrow$ small reduction to ensure convergence.
23         if $\tau = T - 1$ then
24             BREAK (go to the next episode).
```

where $\overline{R}_{t+1}$ is a time dependent estimate (mean) of the average rewards $r(\psi)$. Meanwhile, SARSA's weights' updates resume in

$$\overrightarrow{w}_{t+1} \overset{\text{def}}{=} \overrightarrow{w}_t + \eta_t\ \delta_t\ \overrightarrow{\nabla}\widehat{V}(S_t;\ \overrightarrow{w}_t), \tag{4.28}$$

for state–value, where δ_t is given by (4.26), and

$$\overrightarrow{w}_{t+1} \overset{\text{def}}{=} \overrightarrow{w}_t + \eta_t\ \delta_t\ \overrightarrow{\nabla}\widehat{Q}(S_t, A_t;\ \overrightarrow{w}_t) \tag{4.29}$$

for state–action–value, where δ_t is given by (4.27). As usual, an example of this system's pseudocode follows in **Algorithm 3**.

Algorithm 3: Continuing semi–gradient one–step SARSA [SB19, p. 251].

```
1  begin
2      By defining and/or initializing
           • a parameterized and differentiable $\widehat{Q} : \mathcal{S} \times \mathcal{A} \times \mathbb{R}^d \longrightarrow \mathbb{R}$.
           • a learning rate $0 < \eta \leq 1$ for weight's update.
           • a learning rate $0 < \zeta \leq 1$ for estimate reward's update.
           • a greedy parameter $0 < \epsilon << \leq 1$.
           • a discount rate $0 < \gamma \leq 1$.
           • the weight vector $\overrightarrow{w} \in \mathbb{R}^d$ (e.g., $\overrightarrow{w} = \overrightarrow{0}$).
           • the average reward estimate $\overline{R} \in \mathbb{R}$ (e.g, $\overline{R} = 0$).
           • $S \longleftarrow$ Input from the Environment.
           • $A \longleftarrow$ e.g., $\epsilon$–greedy w.r.t. $\widehat{Q}(S,\ \circ;\ \overrightarrow{w})$.
3  while exists continuing steps do
4      $A \longrightarrow$ Output to the Environment.
5      $S' \longleftarrow$ Input from the Environment.
6      $R' \longleftarrow$ Compute and store.
7      $A' \longleftarrow$ e.g., $\epsilon$–greedy w.r.t. $\widehat{Q}(S',\ \circ;\ \overrightarrow{w})$.
8      $\delta \longleftarrow R_{t+1} - \overline{R} + \widehat{Q}(S', A';\ \overrightarrow{w}) - \widehat{Q}(S, A;\ \overrightarrow{w})$.
9      $\overline{R} \longleftarrow \overline{R} + \zeta\ \delta$.
10     $\zeta \longleftarrow$ small reduction to ensure convergence.
11     $\overrightarrow{w} \longleftarrow \overrightarrow{w} + \eta\ \delta\ \overrightarrow{\nabla}\widehat{Q}(S, A;\ \overrightarrow{w})$.
12     $\eta \longleftarrow$ small reduction to ensure convergence.
13     $S \longleftarrow S'$.
14     $A \longleftarrow A'$.
```

The case is similar with the bootstrapped TD version but with its own peculiarities regarding the return

$$G_{t:t+K+1} \overset{\text{def}}{=} R_{t+1} - \overline{R}_{t+K+1} + \ldots + R_{t+K+1} - \overline{R}_{t+K+1} + \widehat{Q}(S_{t+K+1}, A_{t+K+1};\ \overrightarrow{w}_{t+K}).$$

Then, the K–steps TD–error turns to be

$$\delta_t \overset{\text{def}}{=} G_{t:t+K+1} - \widehat{Q}(S_t, A_t;\ \overrightarrow{w}_t), \tag{4.30}$$

and so we apply that in an pseudocode example for differential semi–gradient K–steps SARSA (see **Algorithm 4**).

Algorithm 4: Continuing semi–gradient K–step SARSA [SB19, p. 255].

```
1  begin
2      By defining and/or initializing
         • a parameterized and differentiable $\widehat{Q} : \mathcal{S} \times \mathcal{A} \times \mathbb{R}^d \longrightarrow \mathbb{R}$.
         • a policy $\psi(\circ|S_t)$: e.g., $\epsilon$–greedy w.r.t. $\widehat{Q}(S_t,\ \circ;\ \overrightarrow{w})$.
         • a learning rate $0 < \eta \leq 1$ for weight's update.
         • a learning rate $0 < \zeta \leq 1$ for estimate reward's update.
         • a greedy parameter $0 < \epsilon << \leq 1$.
         • a discount rate $0 < \gamma \leq 1$.
         • the weight vector $\overrightarrow{w} \in \mathbb{R}^d$ (e.g., $\overrightarrow{w} = \overrightarrow{0}$).
         • the average reward estimate $\overline{R} \in \mathbb{R}$ (e.g, $\overline{R} = 0$).
         • $S_0 \longleftarrow$ Input from the Environment.
         • $A_0 \longleftarrow A_0 \sim \psi(\circ|S_0)$.
3  for each step $t = 0, 1, \ldots$ do
4      $A_t \longrightarrow$ Output to the Environment.
5      $S_{t+1} \longleftarrow$ Input from the Environment.
6      $R_{t+1} \longleftarrow$ Compute and store.
7      $A_{t+1} \longleftarrow A_{t+1} \sim \psi(\circ|S_0)$
8      $\tau \longleftarrow t - K + 1$ (time whose estimates start to update).

9      if $\tau \geq 0$ then
10         $\delta \longleftarrow \sum_{i=\tau+1}^{\tau+K}(R_i - \overline{R}) + \widehat{Q}(S_{\tau+K},\ A_{\tau+K};\ \overrightarrow{w}) - \widehat{Q}(S_\tau,\ A_\tau;\ \overrightarrow{w})$.

11         $\overline{R} \longleftarrow \overline{R} + \zeta\ \delta$.
12         $\zeta \longleftarrow$ small reduction to ensure convergence.
13         $\overrightarrow{w} \longleftarrow \overrightarrow{w} + \eta\ \delta\ \overrightarrow{\nabla}\widehat{Q}(S_\tau, A_\tau;\ \overrightarrow{w})$.
14         $\eta \longleftarrow$ small reduction to ensure convergence.
```

4.5 Linearly weighted off–policy Q–learning and variations

In the preceding section, we discussed several mechanisms of on–policy learning. Now, let us address off–policy, explaining Q–learning method and variants. The same rationale that underpins SARSA can be extended to Q–learning, where the parameterized form of the state–action–value function is equal (4.19). Correspondingly, the approximate state–action–value function is a linear combination between a weight and a feature vector (see 4.16). Thus, the weight's updates become

$$\overrightarrow{w}_{t+1} \overset{\text{def}}{=} \overrightarrow{w}_t + \eta_t \ \delta_{t+1} \ \overrightarrow{\nabla}\widehat{Q}(S_t,\ A_t;\ \overrightarrow{w}_t), \tag{4.31}$$

and the TD–error

$$\delta_{t+1} \overset{\text{def}}{=} R_{t+1} + \gamma \ \max_{a' \in \mathcal{A}} \widehat{Q}(S_{t+1},\ a'; \overrightarrow{w}_t) - \widehat{Q}(S_t,\ A_t; \overrightarrow{w}_t), \tag{4.32}$$

where $\max_{a' \in \mathcal{A}} \widehat{Q}(S_{t+1},\ a'; \overrightarrow{w}_t)$ is commonly identified in the literature as the ***greedy policy***.

Algorithm 5: Q–learning with function approximation [Sze09, p. 50].

```
1  begin
2      By defining and/or initializing
         • a parameterized and differentiable $\widehat{Q} : \mathcal{S} \times \mathcal{A} \times \mathbb{R}^d \longrightarrow \mathbb{R}$.
         • a learning rate $0 < \eta \leq 1$ for weight's update.
         • a discount rate $0 < \gamma \leq 1$.
         • the weight vector $\overrightarrow{w} \in \mathbb{R}^d$ (e.g., $\overrightarrow{w} = \overrightarrow{0}$).
         • $S \longleftarrow$ Input from the Environment.
         • $A \longleftarrow$ e.g., a greedy policy $\arg\max_{\circ \in \mathcal{A}} \widehat{Q}(S_0,\ \circ; \overrightarrow{w})$.
3  while exists state transitions do
4      $A \longrightarrow$ Output to the Environment.
5      $S' \longleftarrow$ Input from the Environment.
6      $R' \longleftarrow$ Compute and store.
7      $A' \longleftarrow$ e.g., a greedy policy $\arg\max_{\circ \in \mathcal{A}} \widehat{Q}(S',\ \circ; \overrightarrow{w})$.
8      $\delta \longleftarrow R' + \widehat{Q}(S', A';\ \overrightarrow{w}) - \widehat{Q}(S, A;\ \overrightarrow{w})$.
9      $\overrightarrow{w} \longleftarrow \overrightarrow{w} + \eta\ \delta\ \overrightarrow{\nabla}\widehat{Q}(S, A;\ \overrightarrow{w})$.
10     $\eta \longleftarrow$ small reduction to ensure convergence.
11     $S \longleftarrow S'$.
12     $A \longleftarrow A'$.
```

Note that off–policy methods are guaranteed to be stable and asymptotically unbiased in the tabular differential scenario, a subset of function approximation. Hence, combining them with a more general method, like function approximation via feature selection, may still be achievable to ensure stability. Despite that, the off–policy update rule outlined in (4.31) turns out to be far more distinct and problematic than the on–policy rule, owing to the former's inability to converge reliably [SB19; Sze09]. Additionally, traditional off–policy processes such as Q–learning may exhibit unstable behavior when coupled with linear value function approximation [Bai95]. As a result, researchers have been investigating several off–policy alternatives that exhibit more robust convergence properties, for instance, *Greedy–GQ*.

This off–policy method, dubbed Greedy–GQ, is the product of a scientific paper from 2010 co–authored by the most eminent researchers in the field of reinforcement learning (see [Mae+10]). In this work, they developed a gradient–based temporal difference learning algorithm for linear and nonlinear function approximation reinforcement learning that is robust under off–policy learning, i.e., the convergence properties are assured. Nonetheless, the authors assert that this thought–provoking solution has several other benefits, including "*1) Linear function approximation; 2) No restriction on the features used; 3) Online, incremental, with memory and per–time–step computation costs that are linear in the number of features; and 4) Convergent to a local optimum or equilibrium point*" [Mae+10, p. 1].

In summary, Greedy–GQ updates the weight parameter $\overrightarrow{w}_t \in \mathbb{R}^d$ in a manner similar to Q–learning with function approximation, except that a corrective term is added, that is, a supplementary sequence of weights $\overrightarrow{\kappa}_t \in \mathbb{R}^d$ and an extra step–size parameter ζ_t. This innovation culminates in the same target value function θ_{t+1} as in Q–learning, but with different weight–update guidelines:

$$
\begin{aligned}
&\theta_{t+1} = R_{t+1} + \gamma \max_{a' \in \mathcal{A}} \widehat{Q}(S_{t+1},\ a'; \overrightarrow{w}_t) \\
&\vartheta_{t+1} \longleftarrow \Big[\theta_{t+1} - \widehat{Q}(S_t,\ A_t; \overrightarrow{w}_t)\Big] \\
&\delta_{t+1} \longleftarrow \vartheta_{t+1} \overrightarrow{\nabla} \widehat{Q}(S_t,\ A_t;\ \overrightarrow{w}_t)\ -\ \gamma\ \widehat{Q}(S_t,\ A_t; \overrightarrow{\kappa}_t) \\
&\overrightarrow{w}_{t+1} \longleftarrow \overrightarrow{w}_t\ +\ \eta_t\ \delta_{t+1}\ \overrightarrow{\nabla} \widehat{Q}(S_{t+1},\ a';\ \overrightarrow{w}_t) \\
&\overrightarrow{\kappa}_{t+1} \longleftarrow \overrightarrow{\kappa}_t + \zeta_t \Big[\vartheta_{t+1} - \widehat{Q}(S_t,\ A_t; \overrightarrow{\kappa}_t)\Big] \overrightarrow{\nabla} \widehat{Q}(S_t,\ A_t;\ \overrightarrow{w}_t),
\end{aligned}
$$

for $a' \sim \arg\max_{a' \in \mathcal{A}} \widehat{Q}(S_{t+1},\ a'; \overrightarrow{w}_t)$.

PART II

Application

CHAPTER 5

State–of–the–art RL algorithms applied to financial markets

This chapter develops three examples of state–of–the–art reinforcement learning algorithms applied to financial markets. Our ultimate purpose has been to design a trading agent that, at the very least, demonstrates rational behavior by ensuring all the technical conditions intrinsic to reinforcement learning. Moreover and more idealistically, we are interested in producing gains over a trade period, which should exceed those achieved through a simple buy–and–hold strategy.

We design an estimate SARSA, estimate Q–learning, and estimate Greedy–GQ agents. These trading agents require no estimation of the probability transition of the environment and are therefore model–free. We decided not to work on model–based reinforcement learning because of the complexity of financial markets, which is deemed unsuitable for forecasting market behavior. Indeed, model–free reinforcement learning is satisfactorily suited to our goals, where an agent makes no effort to predict reward functions or transition probabilities but to learn the optimal policy through experience.

Each of our applications is obtained from Markov decision processes combined with Bellman's theory, both of which are extensively covered in the theory part of this thesis. Simply put, each linear interpolation with respect to the samples of state–action pair's value is updated to execute a specific algorithm's convergence routine. As observations of the environment are evaluated and under certain conditions,

(i) given enough informative features, the true value function $Q_t^{\psi}(S_t, A_t)$ is possibly reached by an approximation SARSA, whereas,

(ii) the same should apply for approximate Q–learning and Greedy–GQ, because

(iii) the estimated $\widehat{Q}_t(S_t, A_t \ ; \ \overrightarrow{w}_t)$ values may converge to the true value of that value function.

Consequently, our algorithms should enable the trading agent to select the most objectively beneficial actions in any given state to perform wisely in the financial markets.

5.1 Specification of the problem

Our goal is to develop an algorithm that trades a single asset optimally. This asset is the mini Bovespa index, or commonly, mini index[1], a futures derivative contract used to trade the Bovespa index's future values (IBOV). Technically defined, IBOV characterizes a weighted average of the performance of the most actively traded firms on the Sao Paulo Stock Exchange[2]. However, unlike the IBOV, the mini index is traded on a mini–cap basis, proportional to 20% of the IBOV, allowing small investors to trade the most prominent companies with a smaller financial scale. Furthermore, the mini index minimum's movement is one tick, or typically spoken, five points. For example, one upward tick from 100 to 105 equals 5 points variation. As a result, a minimum fluctuation of one tick or five points is equivalent to BRL 1.00 for every mini index contract. Additionally, because investors are not permitted to negotiate fractions of a mini index contract, the most diminutive contract size that can be negotiated is one, and the largest is unlimited.

As is the case of any other asset traded on the stock exchanges, it is crucial to know the mini index code to trade it through the brokerage house's Home Broker or any trading platform. The code always begins with a three–character radical (WIN), followed by a letter indicating the month of expiration and two final digits indicating the year of expiration. For illustration, WING22 denotes a mini index contract with the month and year of expiration of February 2022. The months and expiration code are listed in **Table 5.1**.

Month	Code
February	G
April	J
June	M
August	Q
October	V
December	Z

Table 5.1: Mini index's expiration months and codes.

Each mini index contract expires on the Wednesday closest to the 15th of each even month. For our example of WING22, market agents exchanged WING22 until 16.02.2022. It is worth noting that one might have traded WINJ22 prior to 16.02.2022 (when the previous contract WING22 was still valid). However, the most liquid period for a futures contract, especially that following stock market indexes, is immediately following the previous contract's expiration date until one day before the current contract's expiration. Further, it is uncommon to trade on the current contract's expiration day, despite it being possible. Again, market practices are to not trade on the current contract's expiration date because of liquidity reasons. As a corollary, we implemented this concept in our simulations. For WING22, for instance, the period evaluated was exclusively from 15.12.2021 (the prior contract's expiration) until 15.02.2022.

[1]https://www.b3.com.br/pt_br/produtos-e-servicos/negociacao/renda-variavel/futuro-mini-de-ibovespa.htm

[2]https://www.b3.com.br/pt_br/market-data-e-indices/indices/indices-amplos/indice-ibovespa-ibovespa-composicao-da-carteira.htm

(one day before the current contract's expiration). **Table 5.2** summarizes our available data, together with their associated code and period, for use in our applications.

Code	Period
WINJ21	17.02.2021 - 13.04.2021
WINM21	14.04.2021 - 15.06.2021
WINQ21	16.06.2021 - 17.08.2021
WINV21	18.08.2021 - 12.10.2021
WINZ21	13.10.2021 - 14.12.2021
WING22	15.12.2021 - 15.02.2022

Table 5.2: Available data for use in our applications.

As we can see, we are covering a one–year time range. Moreover, the stock market was highly volatile during this period owing to a pandemic (COVID–19) and the beginning of a conflict in Europe (Russian–Ukraine War). This way, our algorithms faced the challenge to adapt to and overcome these global markets adversities, making the results even more exciting.

To trade a regular mini index contract in the São Paulo stock exchange futures markets, an investor must deposit a guarantee equal to a percentage of the contract's total value, the so–called margin, to the broker house. A margin is a sum of money deposited by the counterparties to secure a transaction when a contract is fulfilled. This guarantee may be in the form of cash or any other securities owned by the investor, including bonds and fixed income positions. This margin amount must be held in the broker's account while buyers and sellers maintain open positions, i.e., remain linked to futures contracts. Margin is refunded when trades are closed.

There are two types of margins, one for day trades and the other for positions of more than one days. The brokerage company defines the guarantee margin for day trades, whereas B3 defines the guarantee margin for activities lasting more than one day. Typically, the margin is around 25,00 BRL for a day trade, while 2.800,00 BRL for long–term positions [3]. Since our application case trades a single mini index contract in a day trade or long–term position manner, the margin guarantee needed is 2.800,00 BRL. This means that we must make 2.800,00 BRL available to the brokerage company if we open a position in the mini index market.

Regarding brokerage's commissions, there is a trend toward brokers charging no commission[4] [5] [6] for trading mini index futures contracts. Thus, there is no commission on each deal, which gives us a significant advantage when trading futures contracts on a high–frequency basis with nearly zero costs. It is essential to mention that each transaction involves some stock exchange fees, but they are significantly low and do not affect the outcomes; we will not address that point.

The financial data we have available covers every trading activity on the B3 Stock Exchange in São Paulo. This data came in the form of a raw feed, which

[3]https://www.clear.com.br/site/Content/pdf/ebook_clear_mini_crontratos.pdf?idOrigem=ebook
[4]https://www.modalmais.com.br/planos-modalmais/corretagem-zero
[5]https://www.clear.com.br/site/corretagem-zero
[6]https://www.xpi.com.br/custos-operacionais/

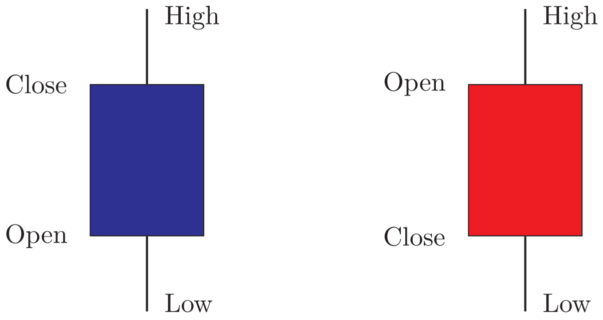

Figure 5.1: Bars illustration.

contains a variety of unstructured information that enables us to reconstruct the trading season in its entirety. To apply our algorithms to unstructured data, we must first parse it, extract what is needed, and store it in a systematized format. The representation of the extracted data we employ is the standard time bars. For the interested reader, we refer to [Pra18] for other possible market representations, such as tick, volume, dollar, tick imbalance bars, and others. This method is highly prevalent in the financial industry, where the APIs of the majority of data vendors incorporate it. A series of observations with an uneven frequency allocation is transformed into a homogeneous time series. In other words, these time bars are created by collecting data at regular intervals and parsing it into bars once every minute. The illustration of these bars are showed in **Figure 5.1**, and the following pieces of information are encapsulated in those bars:

- Timestamp
- Open (the first price of the time bar)
- High (the highest price reached during the time period of the bar)
- Low (the lowest price achieved during the time period of the bar)
- Close (the last price of the time bar)

Time bars are probably the type of representation used most by practitioners and academics, but they may not be the best. Indeed, neither do markets evolve in constant intervals nor are activity periods regular. Moreover, serial correlation, heteroscedasticity, and non–normality of returns are often observed in time–sampled series, which have poor statistical properties [Pra18, p. 26]. Thus, similar to time bars, this dissertation considers tick bars. The purpose of tick bars is to retrieve open, high, low, and close (OHLC) prices whenever a predefined number of transactions, also called ticks, occur. Consequently, we have a representation technique that takes into account both distinct active periods and non–constant expansion. Additionally, this approach has a higher statistical relevance. Numerous studies, beginning with [MT67] and extending to [AG00], prove that sampling as a function of trade activity produces an independent identically distributed (IID) Gaussian process. Indeed, a large number of statistical models rely on this IID premise, which is impossible to accomplish with time bars.

After establishing all of this knowledge, we may precisely identify our problem. These trading times on the futures market (**Table 5.2**) were highly turbulent. This volatility was caused by the world's difficulties, as mentioned previously, and as a result, the mini index alternated several times between bearish and bullish trends, with some gaining and others losing significantly. Taking that into account, we would rather have an intelligent algorithm that learns via its actions on the market and thus profits from its operations. However, the primary challenge is determining whether our algorithm is intelligent enough and profitable. To do so, we constructed a conventional buy–and–hold benchmark and determined whether our algorithm could outperform it. Additionally, we adhered to all technical requirements and always verified the rationale behind them. This aspect is critical because sometimes an algorithm exhibits satisfactory performance, but its technical elements do not make sense.

Apart from that, we want a single unity contract trade at a time, with no chance of enlarging the position, thus incurring only one margin call for that process. This assumption is necessary because it allows us to quantify the agent's initial investment in our system, which is two times the required margin call value (5.600,00 BRL). Notwithstanding that, this margin is necessary because when open positions fluctuate, primarily in the negative direction, we are compelled to restore the guarantee margin, always maintaining it at the required amount to continue the operation. Therefore, we assume that twice the required amount suffices to resolve any pertinent issues.

As previously stated, we are trading a mini index. This futures contract was chosen because it does not require a large sum of money to initiate short– and long–term trades, and, more importantly, this market has a high level of liquidity. This point cannot be ignored since a lack of liquidity in the market can wreck any trade. Indeed, spread and slippage are undesirable characteristics in our scenario, which are consequences of low liquidity. The former is the difference between the best bid and best ask booking prices for the trading asset. The second is the discrepancy between the expected price of a transaction and its actual executed price. However, those issues are minimized in our application scenario. Due to the super high liquidity of the mini index, there usually is not a spread between the bid and ask booking prices. Additionally, we fight the slippage by adopting limit orders at closing prices as our hypothetical procedure. This latter approach may lead to a few orders not being executed, which we ignore for the sake of simplicity. it is necessary to state that we are more interested in determining whether the algorithm can act prudently in the market than in covering operational issues, which would require another entire dissertation to solve. Regardless, this topic is raised here and may serve as a focal point for future research.

Finally, we select to portray data in systematized formats of one and three minutes, as well as 15.000 ticks. These representations are accomplished by parsing raw data into tables that contain the OHLC prices for each timestamp. This technique is carried out by a Python function in the *helper.py*[7] file in our GitHub repository.

[7]https://github.com/fabiorodp/uio_master_thesis/blob/main/helper.py

Time	Open	High	Low	Close
05.29.2021 09:00:00	10.00	13.00	09.00	12.00
05.29.2021 10:00:00	12.00	14.00	11.00	14.00
05.29.2021 11:00:00	14.00	16.00	10.00	10.00

Table 5.3: Example of a state matrix $\mathbf{S} \in \mathbb{R}^{(2+1)\times 4}$, where $n = 2$ and $t =$"05.29.2021 11:00".

5.2 Mathematical formalization of the problem

Our reinforcement learning problem is formalized as a discrete–time stochastic control process $\{\mathcal{S}, \mathcal{A}, \mathcal{R};\ t = 0, 1, 2, \ldots\}$ in which an algorithm trader interacts with a market environment. This algorithm begins by gathering an initial observation ($S_t = \mathbf{S}$, where $S_t \in \mathcal{S}$) of the market. Every market observation $S_t \in \mathbb{R}^{(n+1)\times 4}$ comprises a matrix of $(n+1)$ continuing time with OHLC[8] price values. For instance, consider the following matrix, which derives its constituents from **Table 5.3**:

$$\mathbf{S} = \begin{bmatrix} 10.00 & 13.00 & 09.00 & 12.00 \\ 12.00 & 14.00 & 11.00 & 14.00 \\ 14.00 & 16.00 & 10.00 & 10.00 \end{bmatrix} \tag{5.1}$$

Then, at each time step, the agent must perform an action ($A_t = a$, where $A_t \in \mathcal{A}(\mathit{tradingStatus}) \subset \mathcal{A}$), such that:

$$a = \begin{cases} 0 & \text{do nothing,} \\ +1 & \text{go long,} \end{cases} \tag{5.2}$$

if *tradingStatus*$= -1$ (short on the market), or

$$a = \begin{cases} -1 & \text{go short,} \\ 0 & \text{do nothing,} \\ +1 & \text{go long,} \end{cases} \tag{5.3}$$

if *tradingStatus*$= 0$ (not on the market), or

$$a = \begin{cases} -1 & \text{go short,} \\ 0 & \text{do nothing,} \end{cases} \tag{5.4}$$

if *tradingStatus*$= 1$ (long on the market). These actions are determined in accordance with a policy ψ, where only be ϵ–greedy (SARSA) or greedy (Q–learning) in our cases, which describes the agent's behavior, suggesting each action should be selected for each potential state. This picked action is forwarded to an environment that, as a result of each selection, outputs the next state's observation ($S_{t+1} = \mathbf{S}'$, where $S_{t+1} \in \mathcal{S}$). Consistently with our example, **Table 5.4** and **Figure 5.2**, the matrix $\mathbf{S}' \in \mathbb{R}^{(n+1)\times 4}$ turns to be:

[8]Open, high, low, close prices.

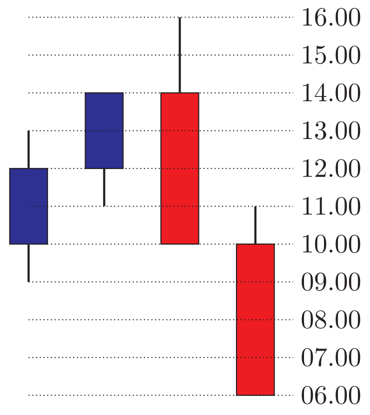

Figure 5.2: Illustration bars of **Table 5.3** and **Table 5.4** combined.

Time	Open	High	Low	Close
05.29.2021 10:00:00	12.00	14.00	11.00	14.00
05.29.2021 11:00:00	14.00	16.00	10.00	10.00
05.29.2021 12:00:00	10.00	11.00	06.00	06.00

Table 5.4: Example of a state matrix prime $\mathbf{S}' \in \mathbb{R}^{(2+1)\times 4}$, where $n = 2$ and $t+1 =$"05.29.2021 12:00".

$$\mathbf{S}' = \begin{bmatrix} 12.00 & 14.00 & 11.00 & 14.00 \\ 14.00 & 16.00 & 10.00 & 10.00 \\ 10.00 & 11.00 & 06.00 & 06.00 \end{bmatrix} \tag{5.5}$$

Along with the output of the next state, the environment also displays the evolution of the reward associated with the initial investment capital and performed actions. Since we are developing an algorithm that attempts to maximize profits from trades, this reward's evolution is the profitability's balance attained from interactions in the financial market. Our algorithm trader starts their operations with a quantified initial capital of 28.000 points (2 contracts margin call). Then, as long as the time steps of the market go on, the environment outputs the surplus of this investment based on each taken action on the market. For instance, suppose the current balance is 28.000 points and the agent signals the environment to make a purchase. If this action results in a profit of 1.000 points, at the current time step, the reward balance output is 29.000 points (28.000 + 1.000).

This evolution of the reward process continuously goes on for each trading time step $t = 1, 2, ...$ and appends the results in the environment method's variable identified as a vector $\overrightarrow{histRprime} \in \mathbb{R}^t$. Provided that, we may define ($R_{t+1} = r(\overrightarrow{histRprime})$, where $R_{t+1} \in \mathcal{R} \subset \mathbb{R}$), and study three choices for this function:

a.) The first one we identify by the name of *minusMean*,

$$R_{t+1} \overset{\text{def}}{=} \text{histRprime}_t - \text{mean}(\overrightarrow{histRprime}),$$

where the computation of returns is done by calculating the difference between immediate reward and the average reward up to the current time step (see 4.25). The hunch here is to build up a reinforcement algorithm compatible with the theory of continuing semi–gradient one–step like **Algorithm 3**.

b.) The second one is referred to as *immediate*,

$$R_{t+1} \overset{\text{def}}{=} \text{histRprime}_t,$$

which simply takes the immediate reward of the current time step.

c.) We specify the third one as *mean*

$$R_{t+1} \overset{\text{def}}{=} \text{mean}(\overrightarrow{histRprime}),$$

which computes the mean of the appended evolution of the rewards.

The Python's class called *Environment* present in the file *environment.py*[9] at our GitHub repository does the job of reproducing the market dynamics and outputting the necessary data.

5.3 Algorithm engineering

This master's thesis section discusses several engineering alternatives, such as approximate SARSA, approximate Q–learning, and approximate Greedy–GQ, as well as their formalization. We developed these agents running on an environment mechanics specified in the early section, which facilitates data sourcing and allows a straightforward testing in our trials. We also implemented these trading agents as Python's class, in the *algorithms.py*[10] file at our GitHub repository, that performs these approximation reinforcement learning theories. Each of the implementations, whether SARSA, Q–learning, or Greedy–GQ, has its own set of attributes that the agent class must receive. This class maintains, updates, and stores feature weights and Q–values, in addition to keeping a link with the environment class.

To begin, we adhered to the theory framework of linear reinforcement learning (see **Sections 4.4** and **4.5**), which has been the subject of many recent and successful academic articles (see **Section 1.5**). The encoding of environmental states as features is the most critical part of linear reinforcement learning algorithms and can have a massive effect on performances. This requires the determination of a basis function that maps states onto features. Since we are working with model–free market data that lacks well–defined transition probabilities and probability distributions, this basis function's theory is required, in accordance with the theory in **Section 4.3**. Thus, we introduce three types of basis functions, namely *sigmoid*, *hyperbolic tangent*, and *sigmoid123*.

A *sigmoid* function is a mathematical function with a distinctive S–shaped curve that maps the entire real line to a limited range, such as between 0 and

[9]https://github.com/fabiorodp/uio_master_thesis/blob/main/environment.py
[10]https://github.com/fabiorodp/uio_master_thesis/blob/main/algorithms.py

1. One frequent application of a sigmoid function is to transform a real value to a probability score, which is easier to interpret. Additionally, all sigmoid functions are monotonic and have a first derivative; hence, satisfies the technical condition for a basic function, and we can express its mathematical formula as follows:

$$b_{\text{sigmoid}}(x) = \frac{1}{1+e^{-x}}. \tag{5.6}$$

The *hyperbolic tangent* function is another frequent variant of a sigmoid function. This function maps any real–valued input to the range of -1 to 1. Due to the fact that it inherits all of the benefits of sigmoid, it also meets the technical prerequisites for being a basis function, and its formula is presented as follows:

$$b_{\text{hypTanh}}(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}. \tag{5.7}$$

The last basis function is *sigmoid123*. We derived this formulation from academic papers [CS15] and [Cor+19], in which the authors achieved positive outcomes. Given the fact that this approach was beneficial for other reinforcement learning algorithms, we repeat it here to compare final results. We adopted the term *sigmoid123* since its formula constructs a sigmoid with some tweaks; nevertheless, unlike the other two functions, this one returns discrete responses, as shown:

$$b_{\text{sigmoid123}}(x) = \frac{a}{1+b\cdot e^{-c\cdot x}} - d, \tag{5.8}$$

where $a = 2$, $b = 1$, $c = 10^{15}$, $d = -1$, and

$$b_{\text{sigmoid123}}(x) = \begin{cases} 1 & \text{if } x < 0, \\ 2 & \text{if } x = 0, \\ 3 & \text{if } x > 0. \end{cases} \tag{5.9}$$

After setting the basis function, we must bring about the basis vector, which will belong to an element of the feature vector subsequently. This basis vector $\overrightarrow{b} \in \mathbb{R}^{n+1}$ has the following elements:

$$b_i = \begin{cases} b\Big(l(\mathbf{S})\Big) & \text{if } i = 1, 2, \ldots, n. \\ b\Big(l(\overline{R})\Big) & \text{if } i = n+1, \end{cases} \tag{5.10}$$

where the function $l(\cdot)$ is defined as the log return, and $\overline{R}$ is the current trade profit or loss (PL)'s ratio, such that

$$l(x) = \begin{cases} \ln\Big(\frac{x[j+1,\ 4]}{x[j,\ 4]}\Big) & \text{if } x = \mathbf{S}, \\ \ln\Big(\frac{\text{current close price}}{|\text{entry price}|}\Big) & \text{if } x = \overline{R}, \end{cases} \tag{5.11}$$

for $j = 1, 2, \ldots, n$, and the column index number 4 represents the close time in the state matrix. In keeping with other academic practices (see **Section 1.5**), it is meaningful to mention that we likewise operate log returns rather than simple returns.

After completing all the above schemes, we can figure out our feature vector $\overrightarrow{\phi} \in \mathbb{R}^{d}$, where $d = |\mathcal{A}| \cdot (n+1)$. Similar to [Cor+19] and [GDH13], we employ a block representation of the feature vector, also known as feature expansion technique, which was first proposed by [Ger+11] and further enhanced by [Ger+13]. This system, which was developed specifically for estimations of linear reinforcement learning, copies the basis vector to one of the three slots in the feature vector, applying the constraint:

$$\vec{\phi} = \begin{cases} \begin{bmatrix} \vec{b} & \vec{0} & \vec{0} \end{bmatrix}^T & \text{if } A_t = -1, \\ \begin{bmatrix} \vec{0} & \vec{b} & \vec{0} \end{bmatrix}^T & \text{if } A_t = 0, \\ \begin{bmatrix} \vec{0} & \vec{0} & \vec{b} \end{bmatrix}^T & \text{if } A_t = 1, \end{cases} \tag{5.12}$$

where $\vec{0} \in \mathbb{R}^{n+1}$.

As discussed in **Section 4.3**, the linear reinforcement learning framework specifies the estimate value function as the inner vector product of weight and feature vectors, as follows:

$$\begin{aligned} \widehat{Q}(S_t, A_t;\ \vec{w}_t) &= \vec{w}_t^T \cdot \vec{\phi}_t \\ &= \sum_{i,j=1}^{d} w_j \cdot \phi_i(S_t, A_t). \end{aligned} \tag{5.13}$$

As a result, we call for a way of initialization of the weights. The chosen method is by the uniform distribution of values between 0 and 1, so that $\vec{w} \sim \text{Uniform}(0, 1) \in \mathbb{R}^d$ portrays this vector.

The purpose of reinforcement learning approximation solution methods is to discover a suitable local minimum for $\text{MSE}(\vec{w})$ by using stochastic gradient descent (SGD) techniques (see 4.1 and **Sections 4.1** and **4.2**). This entails the use of a weight–update algorithm at each time step, according to what we explained in **Section 4.3**. For that, we defined a generalized weight update routine in (4.18), but it is repeated here again for convenience:

$$\vec{w}_{t+1} \overset{\text{def}}{=} \vec{w}_t + \eta_t \left[\theta_t - \widehat{Q}(S_t,\ A_t;\ \vec{w}_t)\right] \vec{\phi}_t, \tag{5.14}$$

where η_t is the learning rate examined at the end of **Section 3.6**, and $\vec{\phi}_t = \nabla\widehat{Q}(S_t,\ A_t;\ \vec{w}_t)$ corresponding to the proof of (4.17). Note that the target value function θ_t is an estimate for the expected future returns with respect to an estimate of its related value function (see 4.11), which we already know how to compute by (5.13), but now for the state–action pair of (S_{t+1}, A_{t+1}), i.e.,

$$\begin{aligned} \theta_t &= R_{t+1} + \gamma G_{t+1} \\ &= R_{t+1} + \gamma Q^{\psi}(S_{t+1},\ A_{t+1}) \\ &\approx R_{t+1} + \gamma \widehat{Q}(S_{t+1},\ A_{t+1};\ \vec{w}_t) \\ &= R_{t+1} + \gamma \sum_{i,j=1}^{d} w_j \cdot \phi_i(S_{t+1}, A_{t+1}). \end{aligned} \tag{5.15}$$

However, the critical concern at this point is how to pick up the future action A_{t+1} and properly establish the target value function once we have S_{t+1}. In our dissertation's case studies, we employ three distinct techniques to solve this issue. It is necessary to highlight that they all adopt the accompanying generic target value function (5.15) and generic weight update (5.14), but with various alterations that produce their unique advantages and ensure their convergence characteristics.

The first is a linearly weighted on–policy SARSA, which is extensively discussed in **Section 4.4**. In the same way as in (5.15), we define its target value function as

$$\theta_t = R_{t+1} + \gamma\ \widehat{Q}(S_{t+1},\ a';\ \overrightarrow{w}_t),$$

where $a' \sim \epsilon$–greedy policy agreeing to

$$a' = \begin{cases} \arg\max_{a' \in A(tradingStatus)} \widehat{Q}(S_{t+1},\ a';\ \overrightarrow{w}_t) & \text{with probability } 1-\epsilon, \\ U_t\Big(\mathcal{A}(tradingStatus)\Big) & \text{with probability } \epsilon, \end{cases}$$

where U_t randomly draws a uniformly distributed action from $\mathcal{A}(tradingStatus)$. From there, the final equation for weight update becomes

$$\overrightarrow{w}_{t+1} \longleftarrow \overrightarrow{w}_t + \eta_t \Big[R_{t+1} + \gamma \sum_{i,j=1}^{d} w_j \cdot \phi_i(S_{t+1}, a') - \sum_{i,j=1}^{d} w_j \cdot \phi_i(S_t, A_t)\Big] \overrightarrow{\phi}_t,$$

for $a' \sim \epsilon$–greedy policy. It is crucial to mention that we initialize the weight vector here in two distinct ways: $\overrightarrow{w}_0 = \overrightarrow{0}$ and $\overrightarrow{w}_0 \sim \text{Uniform}(x) \in (0, 1)$.

The second is a linearly weighted off–policy Q–learning covered in **Section 4.5**, where we determine its target value function as:

$$\theta_t = R_{t+1} + \gamma \max_{a' \in \mathcal{A}(\text{tradingStatus})} \widehat{Q}(S_{t+1},\ a';\ \overrightarrow{w}_t), \tag{5.16}$$

where $a' \sim$ greedy policy. Hence, the final equation for weight update becomes:

$$\begin{aligned} w_{t+1} \longleftarrow w_t + \eta_t \Big[R_{t+1} + \gamma \max_{a' \in \mathcal{A}(\text{tradingStatus})} \sum_{i,j=1}^{d} w_j \cdot \phi_i(S_{t+1}, a') \\ - \sum_{i,j=1}^{d} w_j \cdot \phi_i(S_t, A_t)\Big] \overrightarrow{\phi}_t, \end{aligned}$$

for $a' \sim$ greedy policy. Keep in mind that we initialize the weight vector as $\overrightarrow{w}_0 \sim \text{Uniform}(x) \in (0, 1)$ for this second system.

The last one is Greedy–GQ (see [Mae+10] and **Section 4.5**). This method stores quantities for a supplementary sequence of weights $\overrightarrow{\kappa}_t \in \mathbb{R}^d$ and an extra step–size parameter ζ_t, culminating in the same target value function θ_t as in

Q–learning (5.16), but with different weight update guidelines:

$$\begin{aligned}
&\theta_t = R_{t+1} + \gamma \max_{a' \in \mathcal{A}(\text{tradingStatus})} \sum_{i,j=1}^{d} w_j \cdot \phi_i(S_{t+1}, a') \\
&\vartheta_{t+1} \longleftarrow \Big[\theta_{t+1} - \sum_{i,j=1}^{d} w_j \cdot \phi_i(S_t, A_t)\Big] \\
&\overrightarrow{w}_{t+1} \longleftarrow \overrightarrow{w}_t + \eta_t \Big[\vartheta_{t+1} \overrightarrow{\phi}_t - \gamma \sum_{i,j=1}^{d} \kappa_j \cdot \phi_i(S_t, A_t)\Big] \overrightarrow{\phi}_{t+1} \\
&\overrightarrow{\kappa}_{t+1} \longleftarrow \overrightarrow{\kappa}_t + \zeta_t \Big[\vartheta_{t+1} - \sum_{i,j=1}^{d} \kappa_j \cdot \phi_i(S_t, A_t)\Big] \overrightarrow{\phi}_t,
\end{aligned}$$

for $a' \sim$ greedy policy, where we initialize the weight vectors as $\overrightarrow{w}_0, \overrightarrow{\kappa}_0 \sim \text{Uniform}(x) \in (0, 1)$ for this last scheme.

Now, all that remains to be done is to determine the learning rates $\eta, \zeta \in (0, 1)$ that match to the theoretically required step rates for the learning process to be convergent (see [RM51] and discussion at the end of **Section 3.6**). These parameters may decrease in value over time as the model approaches its optimal point, which is why we created a learning rate scheduler function, namely *lrSchedulerFct*. After a fixed and pre–defined amount of time steps, this function cuts the learning rate by half. Furthermore, we have the option of keeping the learning rate constant with no cutback.

Ultimately, the pseudocodes for the estimate SARSA, estimate Q–learning, and estimate Greedy–GQ algorithms are in (6), (7), and (8) respectively. They were built as Python's classes and stored in *algorithms.py*[11] file in our GitHub repository.

[11] https://github.com/fabiorodp/uio_master_thesis/blob/main/algorithms.py

Algorithm 6: Pseudocode for estimate SARSA algorithm.

```
begin
    By initializing the constructor with
        • $S \longleftarrow$ Environment.
        • $A \sim \epsilon$–greedy policy.
for each time step do
    $A \longrightarrow$ Environment.
    $S', R' \longleftarrow$ Environment.
    $A' \sim \epsilon$–greedy policy.
    $\delta \longleftarrow r(R') + \gamma \widehat{Q}(S',\ A';\ \overrightarrow{w}) - \widehat{Q}(S,\ A;\ \overrightarrow{w})$.
    $\overrightarrow{w} \longleftarrow \overrightarrow{w} + \eta\ \delta\ \overrightarrow{\nabla} \widehat{Q}(S, A;\ \overrightarrow{w})$.
    $\eta \longleftarrow lrSchedulerFct$.
    $S \longleftarrow S'$.
    $A \longleftarrow A'$.
    $t \longleftarrow t + 1$.
```

Algorithm 7: Pseudocode for estimate Q–learning algorithm.

```
begin
    By initializing the constructor with
        • $S \longleftarrow$ Environment.
        • $A \sim$ greedy policy w.r.t. $\arg\max_{a \in \mathcal{A}(tradingStatus)} \widehat{Q}(S',\ a;\ \overrightarrow{w})$.
for each time step do
    $A \longrightarrow$ Environment.
    $S', R' \longleftarrow$ Environment.
    $A' \sim$ greedy policy w.r.t. $\arg\max_{a \in \mathcal{A}(tradingStatus)} \widehat{Q}(S',\ a;\ \overrightarrow{w})$.
    $\delta \longleftarrow r(R') + \gamma \widehat{Q}(S',\ A';\ \overrightarrow{w}) - \widehat{Q}(S,\ A;\ \overrightarrow{w})$.
    $\overrightarrow{w} \longleftarrow \overrightarrow{w} + \eta\ \delta\ \overrightarrow{\nabla} \widehat{Q}(S, A;\ \overrightarrow{w})$.
    $\eta \longleftarrow lrSchedulerFct$.
    $S \longleftarrow S'$.
    $A \longleftarrow A'$.
    $t \longleftarrow t + 1$.
```

Algorithm 8: Pseudocode for estimate Greedy–GQ algorithm.

```
1  begin
2      By initializing the constructor with
           • $S \longleftarrow$ Environment.
           • $A \sim$ greedy policy w.r.t. $\arg\max_{a \in \mathcal{A}(tradingStatus)} \widehat{Q}(S',\ a;\ \overrightarrow{w})$.
3  for each time step do
4      $A \longrightarrow$ Environment.
5      $S', R' \longleftarrow$ Environment.
6      $A' \sim$ greedy policy w.r.t. $\arg\max_{a \in \mathcal{A}(tradingStatus)} \widehat{Q}(S',\ a;\ \overrightarrow{w})$.
7      $\vartheta \longleftarrow r(R') + \gamma \widehat{Q}(S',\ A';\ \overrightarrow{w}) - \widehat{Q}(S,\ A;\ \overrightarrow{w})$.
8      $\overrightarrow{w} \longleftarrow \overrightarrow{w} + \eta \left[\vartheta\, \overrightarrow{\nabla} \widehat{Q}(S, A;\ \overrightarrow{w}) - \gamma \overrightarrow{\kappa}^T \overrightarrow{\nabla} \widehat{Q}(S, A;\ \overrightarrow{w})\right] \overrightarrow{\nabla} \widehat{Q}(S', A';\ \overrightarrow{w})$.
9      $\overrightarrow{\kappa} \longleftarrow \overrightarrow{\kappa} + \zeta \left[\vartheta - \overrightarrow{\kappa}^T \overrightarrow{\nabla} \widehat{Q}(S, A;\ \overrightarrow{w})\right] \overrightarrow{\nabla} \widehat{Q}(S, A;\ \overrightarrow{w})$.
10     $\eta, \zeta \longleftarrow lrSchedulerFct$.
11     $S \longleftarrow S'$.
12     $A \longleftarrow A'$.
13     $t \longleftarrow t + 1$.
```

PART III

Analysis

CHAPTER 6

Assessment of applicability, profitability, and reliability

We have intensively dealt with all the theoretical and applicative engineering characterizations so far. Now, in this third part of the thesis, it is time to focus on our study's analysis, incorporating its findings, a discussion combining results, statistics and theory, and subsequently a conclusion to the entire investigation. In the discussion, we also tackle the thesis's fundamental research question, look into future works, and recommend some unresolved academic issues in the literature. In the conclusion, we review all the themes and the specific aims achieved.

6.1 Results

First, we established a baseline before undertaking any experiment. There were many alternatives for that, but we opted for a straightforward buy–and–hold strategy. Other investment tactics are more sophisticated and would deviate from the thesis's aim. On the contrary, buy–and–hold is a popular and conservative approach whereby an individual purchases an investment asset and holds them for an extended length of time regardless of the market movements. Typically, this investor is unconcerned with short–term price swings and technical indicators, seeking mostly long–term ventures.

Table 6.1 demonstrates how buy–and–hold fared from 17.02.21 to 15.02.22,

Code	Period	First trade	Last trade	B&H PL
WINJ21	17.02.21 - 13.04.21	119,695	119,010	-685
WINM21	14.04.21 - 15.06.21	119,490	130,075	10,585
WINQ21	16.06.21 - 17.08.21	130,705	117,150	-13,555
WINV21	18.08.21 - 12.10.21	118,890	112,100	-6,790
WINZ21	13.10.21 - 14.12.21	113,370	106,520	-6,850
WING22	15.12.21 - 15.02.22	108,400	115,135	6,735
Total	17.02.21 - 15.02.22	-	-	-10,560

Table 6.1: Buy–and–hold benchmark strategy, comprising the first and last trade prices, as well as the profit or loss of the periods, all of which are measured by points.

breaking through each mini index expiration date. As shown, the market's volatility was exceptionally high, culminating in episodes of significant losses and occasional spectacular gains. This might be explained by the international scenario in which the world found itself, namely the COVID–19 pandemic restrictions and the commencement of the Russia–Ukraine War. This global situation is relevant for our research because it allows us to visualize even more clearly if reinforcement learning algorithms are capable of quickly learning from adversity and make worthwhile decisions while still providing robust profits.

Following that, as previously reported but worth repeating, we will delve into three reinforcement learning designs across two distinct time–framed periods (60 minutes and 500000 ticks). To that end, we conducted a preliminary examination of the algorithms' achievement by running several simulations and comparing their behavior when different parameters were varied. More precisely, we investigated the performance difference between multiple alternative values for *rlType* (types of reinforcement learning framework), n (number of close prices to look back), *basisFctType* (types of basis function), *rewardType* (types of reward function), η (learning rate), ζ (second learning rate only for Greedy–GQ), γ (discount rate), ϵ (probabilities for exploitation vs exploration), *lrScheduler* (gradually reducing the learning rate or keeping it constant), and *initType* (types of initialization of the weight vector) while running 50 *seeds* (stochastic differently simulations) for each parameter combination. This hyper–parameter tuning is developed in the file called *pipeline.py*[1] containing a massive pipeline with the settings listed in **Table 6.2**.

hyper–parameters	Values
rlType	SARSA, QLearn, Greedy–GQ
n	$5, 25, 50$
basisFctType	sigmoid, sigmoid123, hypTanh
rewardType	minusMean, immediate, mean
η	0.1, 0.01
ζ	0.1, 0.01
γ	1, 0.95
ϵ	0.15, 0.1
lrScheduler	0, 200
initType	uniform01, zeros
seeds	1, ..., 51

Table 6.2: hyper–parameters' settings for the pipeline.

The initial step for determining the optimal system was to design this gigantic pipeline and test all potential parametric's combinations. This investigation ran for several weeks, nearly two months, and yielded extremely interesting results.

It is critical to note that, while we attempt to maximize returns throughout the trading process, we consider the final profit or loss (PL) at the terminal point to reflect our algorithms' performance. To make it easier to comprehend the algorithms' accomplishments, we deduct the initial investment capital from the final return (G_T) to get solely the profit or loss of the systems, which we

[1] https://github.com/fabiorodp/uio_master_thesis/blob/main/pipeline.py

	hyper–parameters	Final PL
1	**QLearn, 50, sigmoid123, minusMean, 0.01, 0.95, 200**	**-20,487**
2	**Greedy–GQ, 25, sigmoid, minusMean, 0.01, 1, 0.01, 0**	**-13,995**
3	Greedy–GQ, 25, sigmoid, minusMean, 0.01, 1, 0.01, 200	-13,995
4	QLearn, 50, sigmoid123, minusMean, 0.01, 0.95, 0	-13,304
5	Greedy–GQ, 5, sigmoid123, minusMean, 0.01, 1, 0.01, 0	-10,477
6	Greedy–GQ, 50, sigmoid123, minusMean, 0.01, 0.95, 0.1, 200	-8,880
7	Greedy–GQ, 50, sigmoid123, minusMean, 0.01, 0.95, 0.1, 0	-8,838
8	Greedy–GQ, 5, sigmoid123, immediate, 0.1, 1, 0.1, 200	-8,512
9	Greedy–GQ, 50, sigmoid123, minusMean, 0.01, 1, 0.1, 200	-8,292
10	Greedy–GQ, 50, sigmoid123, immediate, 0.1, 1, 0.1, 200	-7,828
11	Greedy–GQ, 50, sigmoid123, minusMean, 0.01, 1, 0.1, 0	-7,706
12	QLearn, 50, sigmoid, minusMean, 0.01, 0.95, 0	-7,650
13	**SARSA, 50, sigmoid123, minusMean, 0.01, 0.95, 0.15, uniform01, 200**	**-7,531**
14	Greedy–GQ, 5, sigmoid, minusMean, 0.01, 1, 0.01, 0	-7,314
15	SARSA, 50, sigmoid123, minusMean, 0.01, 1, 0.1, uniform01, 200	-7,230
16	QLearn, 5, hypTanh, immediate, 0.1, 0.95, 0	-7,050
17	QLearn, 5, hypTanh, immediate, 0.1, 1, 0	-7,050
18	Greedy–GQ, 25, sigmoid, mean, 0.1, 0.95, 0.01, 0	-7,008
19	Greedy–GQ, 25, sigmoid, minusMean, 0.01, 1, 0.1, 0	-6,995
20	SARSA, 50, sigmoid123, minusMean, 0.01, 0.95, 0.15, zeros, 200	-6,762

Table 6.3: Top 20 worst combination of hyper–parameters for 60 minutes' time–framed periods. Final profit and loss values (Final PL) presented in points.

	hyper–parameters	Final PL
1	**Greedy–GQ, 5, sigmoid, minusMean, 0.01, 0.95, 0.1, 200**	**16,896**
2	**QLearn, 5, sigmoid, minusMean, 0.01, 0.95, 0**	**16,073**
3	QLearn, 5, sigmoid123, minusMean, 0.01, 0.95, 0	15,615
4	QLearn, 5, sigmoid123, minusMean, 0.01, 0.95, 200	15,450
5	QLearn, 25, sigmoid, minusMean, 0.1, 0.95, 0	14,028
6	QLearn, 5, sigmoid, minusMean, 0.01, 0.95, 200	13,076
7	QLearn, 25, sigmoid, minusMean, 0.1, 0.95, 200	12,721
8	QLearn, 25, sigmoid123, minusMean, 0.1, 1, 200	12,550
9	Greedy–GQ, 50, sigmoid, minusMean, 0.01, 0.95, 0.01, 200	12,374
10	QLearn, 5, sigmoid, minusMean, 0.1, 0.95, 0	12,033
11	QLearn, 25, hypTanh, minusMean, 0.1, 1, 0	12,003
12	QLearn, 25, hypTanh, minusMean, 0.1, 0.95, 0	12,001
13	QLearn, 25, hypTanh, minusMean, 0.1, 1, 200	11,348
14	QLearn, 25, hypTanh, minusMean, 0.1, 0.95, 200	11,348
15	QLearn, 50, hypTanh, minusMean, 0.1, 1, 200	10,926
16	QLearn, 50, hypTanh, minusMean, 0.1, 0.95, 200	10,926
17	**SARSA, 5, hypTanh, minusMean, 0.01, 1, 0.1, zeros, 200**	**10,776**
18	SARSA, 5, hypTanh, minusMean, 0.01, 0.95, 0.1, zeros, 200	10,689
19	SARSA, 5, hypTanh, minusMean, 0.1, 0.95, 0.1, zeros, 200	10,610
20	SARSA, 5, hypTanh, minusMean, 0.1, 1, 0.1, zeros, 200	10,610

Table 6.4: Top 20 best combination of hyper–parameters for 60 minutes' time–framed periods. Final profit and loss values (Final PL) presented in points.

refer to as *Final PL* in our tables, and define as follows:

$$PL \overset{\text{def}}{=} G_T - \text{Initial Investment Capital},$$

where T is the terminal state, and Initial Investment Capital equals to 28,000 points in our cases.

From **Table 6.3** and **6.4**, we may now conduct a preliminary assessment of these findings, adopting the PL performance metric for a 60–minute time–framed period.

- The first convincing evidence that highlights the success of these reinforcement learning methods is that merely 4 out of over 1500 parametric combinations yielded an inferior performance based on our benchmark. This first piece of evidence indicates unequivocally that reinforcement learning methods can easily outperform a simple buy–and–hold strategy, even when the algorithms are neither fine–tuned nor optimized properly.

- Another noteworthy observation from these data is the drop ratio of the twenty worst and twenty best performances. On the one hand, the worst–case scenario led to a loss of 20,487, while the 20th worst case resulted in a loss of 6,762 points. This negative view brought in a drop ratio of around 66%. On the other hand, the best scenario generated a profit of 16,896, while the twentieth best cases generated a profit of 10,610 points. This positive view resulted in a drop ratio of around 37%. As we can see, the loser side turns to be a winner more quickly than the winner to a loser. This behavior is desirable because it emphasizes the goal of maximizing returns, where losses must frequently be offset by profits.

Another sort of evaluation that we may do is one that concerns the hyper–parameters. It is necessary to emphasize that the purpose of this master's dissertation is not to uncover the most generalized, fine–tuned and/or optimal algorithm that best maximizes profit. For that, it would have been necessary to run many more combinations of hyper–parameters, explore more financial contracts, and consider the uniqueness of each different asset, which should definitely be further researched in futures works. However, as we are dealing with limited computational power (a personal laptop), limited time, and limited data source (free data from B3's exchange), we narrowed our aim to presenting a more relevant discussion, such as the applicability, profitability, and reliability of the reinforcement learning methodologies. Besides, despite these restrictions, **Table 6.3** and **6.4** can provide useful information about the hyper–parameters and perhaps about the algorithms:

- Greedy–GQ provided the highest return. Nevertheless, further research is essential before concluding that this strategy is the best.

- The hyper–parameters n equals 50 and *basisFctType* equals "sigmoid123" appeared more consistently in the top 20 worst results. Due to the fact that these hyper–parameters are not frequently detected in the top twenty best returns, this may imply that they are not suitable for the success of our models.

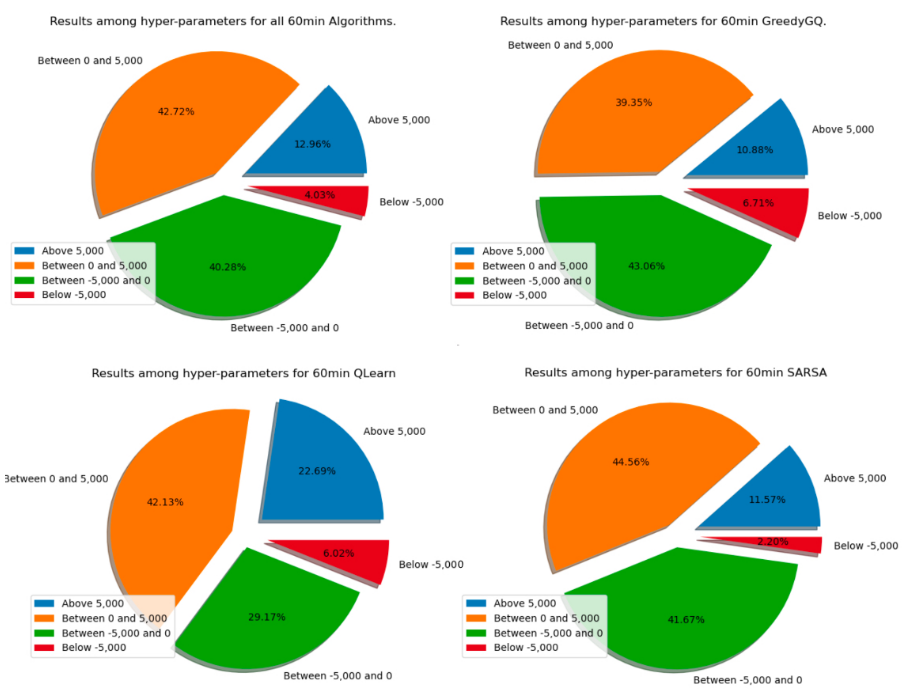

Figure 6.1: Four pie plots containing the proportions for "Above 5,000", "Between 0 and 5,000", "Between -5,000 and 0", and "Below -5,000" profit/loss results in points. These results are the entire possible combinations of hyper–parameters. Each combination was run from 17.02.21 to 15.02.22 in a 60–min frame. Top left, top right, bottom left and bottom right represent 60–min all, Greedy–GQ, QLearn, and SARSA algorithms respectively.

- The hyper–parameter n equals 5 appeared regularly in the top twenty greatest returns but infrequently in the top twenty worst. This evidence reveals that the basis vector does not need to look far back in time to learn about best market practices. Similarly, although from a different point of view, a large value for n implies that the basis vector must look far behind in time, which may confuse judgments and culminate in losses.
- Despite not being the optimal basis function, “hypTanh” routinely featured in the top twenty best PLs and only rarely in the top twenty worst. This suggests that “hypTanh” may be a more stable function than “sigmoid” or “sigmoid123”, that is, it is likely to produce sustainable profits while having rare losses.
- Looking upon the other parameters, we found non–standard patterns of results, making it difficult to have conduct accurate convictions about them.

Figure 6.1 presents the proportionality of the final profit or loss in four different groups: “Above 5,000”, “Between 0 and 5,000”, “Between -5,000 and 0”, and “Below -5,000”. The intention although is to detect which reinforcement learning type is more stable, that is, which model consistently and more frequently provides the most positive results, among the various combinations of hyper–parameters, even though they are not wholly adjusted. Moreover, we picked 5,000 points as a border reference since it represents around 18% of our return on investment, which is a significant amount in financial markets in terms of annualized profit. Hence, from the findings, we may draw the following:

- The top left pie chart summarizes the findings for all algorithms and parameters searched. As we can see, most of the results are favorable, with an impressive 12.96% exceeding 5,000 points. On the other extreme, a tiny portion is less than -5,000 points (4.03%), which is quite desirable and appears to be related to the algorithms' success.
- When examining the other three pie charts, Greedy–GQ (top right pie) appears to be the most unstable algorithm. Although a considerable number of outcomes remain favorable for Greedy–GQ, we detect a greater number on the negative side, both for "Between -5,000 and 0" and "Below -5,000" levels, when compared with the overall results (top left pie).
- The pie chart plotted on the bottom right concerns for the QLearn algorithm. According to the plots, this approach is the superior option compared with the other two. This dominance is attributable to the fact that 22.69% of outcomes are greater than 5,000 points, while 12.96%, 10.88% and 11.57% were achieved for all, Greedy–GQ, and SARSA. This resilient percentile implies that, even when hyper–parameters are altered, the method preserves a high level of performance, making it highly solid. On the negative side, the number of outcomes below -5.000 points (6.02%) rose when compared with the overall measure of 4.03%, or even 2.20% for SARSA. However, this negative finding is not problematic when we consider that it is still lower than the 6.71% produced by the Greedy–GQ. Additionally, the 29.17% achieved for the "Between -5,000 and 0" group is substantially less than 40.28%, 43.06%, and 41.67% for all, Greedy–GQ, and SARSA respectively, demonstrating yet again how robust QLearn appears to be.
- The bottom–left pie represents SARSA. This system achieved a stunning 2.20% of results in the category "Below -5,000," as opposed to the other approaches. The remaining categories keep nearly identical percentages to the aggregate ones. These facts may indicate that SARSA is best suited for cautious agents who minimize downside risks.

As we did for the time–framed periods of 60 min, we may also design assessments concerning the models, hyper–parameters, and other specifications adopting the PL performance metric for 500k ticks' time–framed intervals, but this time taking as reference **Table 6.5** and **6.6**:

- It is essential to keep in mind first that both the greatest profits and losses climbed significantly. The increase of profits points out that the algorithms are becoming more adept at determining how to act in the market, resulting in more earnings. As previously stated and consistent with literature [Pra18, p. 26], 500k ticks' time–framed intervals may provide greater statistical significance to the data, allowing for better understanding and easier discovery of its optimal configuration.

Conversely, a rise in losses seems to be an unavoidable trade-off for increased earnings. Contrasted with 60–minute time–framed data, proper parameter tuning is now required owing to the fact that flawed settings might cause considerable losses.

- As observed for 60–minute, the drop ratio between the top worst outcomes is greater than that of the top best. Indeed, the losing scenario has a drop ratio of 78.53%, while the winning scenario has 57.18%. Take note that we are now working with the top 50, rather than the top 20 as was the case in the 60–minute.
- This time, SARSA proved to be a very effective strategy, generating really high revenue. Additionally, the top 26 occurrences were SARSA algorithms, with the best QLearn algorithm coming in at 27th and the greatest Greedy–GQ algorithm coming in at 49th. SARSA's tremendous profit (48,246 points) was much more than that of its competitors QLearn and Greedy–GQ, which attained 36,675 and 21,159 points respectively.
- The winner **Table 6.6** manifests a preponderance of lower values for the hyper–parameter n, while the loser **Table 6.5** displays a preeminence of higher qualities. Similar to what we saw in the 60 minutes' time–framed case, a higher value for n leads to a longer basis vector with a wider view back in market's history prices, and consequently, this lengthy vector may be more detrimental than helpful to the models.
- It is a hurtful highlight for the "sigmoid" basis function, which yields excessively negative results with the biggest losses in all top 13 worst PLs. Indeed, there were 34 instances of the "sigmoid" basis function in the top 50 worst outputs, followed by 10 occurrences of the "sigmoid123" and just 6 cases of "hypTanh". Besides, the "sigmoid" appears only twice on the winning outcomes' list. Thus, these pieces of evidence unequivocally suggest that "sigmoid" basis function can be very erratic, i.e., it repeatedly gives extremely large losses while just occasionally producing profits.
- As we clearly see in **Table 6.6**, the "hypTanh" basis function was consistently leading the best outcomes with very generous revenues when compared with the other functions. This circumstance is well proven with a frequency of 47 out of 50 occurrences in the top winners' table, as opposed to only 6 out of 50 in the top losers' table. Additionally, the "hypTanh" seems to go on vigorously with any kind of reinforcement learning approach, but particularly well with SARSA, which achieved the best results.
- In the top winners' **Table 6.6**, 40 out of 50 instances, including the greatest one, selected 5, "hypTanh" and "minusMean" as the optimal parameters n, *basisFctType* and *rewardType*, respectively. Together, these three parameters did not make it into the top worst **Table 6.5**. As a result, these results strongly indicate that their arrangement is optimal for our proposed case.
- Considering the remaining parameters, we identified non–standard patterns, making exact convictions about them being precarious.

	hyper–parameters	Final PL
1	QLearn, 25, sigmoid, minusMean, 0.01, 0.95, 200	-70,280
2	QLearn, 25, sigmoid, minusMean, 0.01, 0.95, 0	-61,570
3	Greedy–GQ, 5, sigmoid, minusMean, 0.01, 1, 0.1, 0	-52,360
4	QLearn, 25, sigmoid, mean, 0.01, 0.95, 200	-48,430
5	Greedy–GQ, 5, sigmoid, minusMean, 0.01, 1, 0.1, 200	-47,554
6	Greedy–GQ, 5, sigmoid, minusMean, 0.01, 0.95, 0.1, 0	-47,363
7	QLearn, 25, sigmoid, immediate, 0.01, 0.95, 0	-43,732
8	Greedy–GQ, 5, sigmoid, minusMean, 0.01, 0.95, 0.1, 200	-41,756
9	QLearn, 50, sigmoid, minusMean, 0.01, 0.95, 200	-41,670
10	QLearn, 25, sigmoid, mean, 0.01, 0.95, 0	-40,501
11	QLearn, 50, sigmoid, immediate, 0.01, 0.95, 200	-37,944
12	SARSA, 25, sigmoid, minusMean, 0.01, 0.95, 0.1, uniform01, 0	-36,189
13	QLearn, 25, sigmoid, minusMean, 0.01, 1, 200	-34,410
14	QLearn, 5, sigmoid123, minusMean, 0.01, 1, 200	-33,302
15	QLearn, 5, sigmoid, mean, 0.1, 0.95, 200	-31,826
16	QLearn, 25, sigmoid, minusMean, 0.01, 1, 0	-30,961
17	QLearn, 25, sigmoid123, minusMean, 0.01, 0.95, 200	-28,284
18	SARSA, 25, sigmoid, immediate, 0.01, 0.95, 0.15, uniform01, 0	-23,956
19	Greedy–GQ, 5, sigmoid123, minusMean, 0.01, 1, 0.01, 200	-23,856
20	SARSA, 25, sigmoid, immediate, 0.01, 0.95, 0.1, uniform01, 0	-23,089
21	SARSA, 25, sigmoid, minusMean, 0.01, 0.95, 0.1, uniform01, 200	-22,099
22	SARSA, 25, sigmoid, minusMean, 0.01, 0.95, 0.1, zeros, 0	-21,759
23	Greedy–GQ, 25, sigmoid, minusMean, 0.01, 0.95, 0.1, 200	-21,698
24	SARSA, 25, sigmoid, immediate, 0.01, 0.95, 0.1, zeros, 0	-21,313
25	Greedy–GQ, 25, sigmoid, minusMean, 0.01, 1, 0.1, 200	-20,817
26	QLearn, 5, sigmoid, minusMean, 0.01, 0.95, 0	-20,262
27	QLearn, 5, sigmoid123, minusMean, 0.01, 0.95, 200	-20,210
28	SARSA, 25, sigmoid123, minusMean, 0.01, 0.95, 0.1, zeros, 0	-19,976
29	SARSA, 25, sigmoid, mean, 0.01, 0.95, 0.1, uniform01, 0	-19,959
30	Greedy–GQ, 25, sigmoid, minusMean, 0.01, 0.95, 0.1, 0	-19,846
31	SARSA, 25, sigmoid, minusMean, 0.01, 0.95, 0.15, uniform01, 0	-19,450
32	Greedy–GQ, 5, sigmoid123, minusMean, 0.01, 1, 0.01, 0	-19,302
33	Greedy–GQ, 25, sigmoid, minusMean, 0.01, 1, 0.1, 0	-19,237
34	QLearn, 5, hypTanh, immediate, 0.1, 1, 0	-18,865
35	Greedy–GQ, 5, sigmoid123, minusMean, 0.1, 0.95, 0.01, 200	-18,638
36	QLearn, 5, sigmoid, minusMean, 0.01, 0.95, 200	-18,635
37	Greedy–GQ, 5, sigmoid, immediate, 0.1, 0.95, 0.1, 200	-18,472
38	SARSA, 25, sigmoid, minusMean, 0.01, 0.95, 0.15, zeros, 0	-18,393
39	QLearn, 25, sigmoid123, minusMean, 0.01, 0.95, 0	-18,316
40	SARSA, 25, sigmoid, immediate, 0.01, 0.95, 0.15, zeros, 0	-18,232
⋮	⋮	⋮
49	QLearn, 5, hypTanh, immediate, 0.1, 0.95, 200	-15,083
50	QLearn, 5, hypTanh, immediate, 0.1, 1, 200	-15,083

Table 6.5: Top 50 worst combination of hyper–parameters for 500,000 ticks' time–framed periods. Final profit and loss values (Final PL) presented in points.

	hyper–parameters	Final PL
1	**SARSA, 5, hypTanh, minusMean, 0.1, 1, 0.1, uniform01, 0**	**48,246**
2	SARSA, 5, hypTanh, minusMean, 0.01, 0.95, 0.1, uniform01, 200	47,756
3	SARSA, 5, hypTanh, minusMean, 0.01, 1, 0.1, uniform01, 200	47,756
4	SARSA, 5, hypTanh, minusMean, 0.1, 0.95, 0.1, uniform01, 0	47,655
5	SARSA, 5, hypTanh, minusMean, 0.1, 1, 0.1, uniform01, 200	45,236
6	SARSA, 5, hypTanh, minusMean, 0.1, 0.95, 0.1, uniform01, 200	45,209
7	SARSA, 5, hypTanh, minusMean, 0.01, 0.95, 0.1, zeros, 200	43,735
8	SARSA, 5, hypTanh, minusMean, 0.01, 1, 0.1, zeros, 200	43,735
9	SARSA, 5, hypTanh, minusMean, 0.1, 1, 0.1, zeros, 200	43,723
10	SARSA, 5, hypTanh, minusMean, 0.1, 0.95, 0.1, zeros, 200	43,723
11	SARSA, 5, hypTanh, minusMean, 0.01, 1, 0.1, uniform01, 0	43,168
12	SARSA, 5, hypTanh, minusMean, 0.01, 0.95, 0.1, uniform01, 0	43,167
13	SARSA, 5, hypTanh, minusMean, 0.01, 1, 0.15, uniform01, 200	42,157
14	SARSA, 5, hypTanh, minusMean, 0.01, 0.95, 0.15, uniform01, 200	42,157
15	SARSA, 5, hypTanh, minusMean, 0.01, 0.95, 0.15, zeros, 200	41,477
16	SARSA, 5, hypTanh, minusMean, 0.01, 1, 0.15, zeros, 200	41,477
17	SARSA, 5, hypTanh, minusMean, 0.1, 0.95, 0.15, zeros, 200	41,449
18	SARSA, 5, hypTanh, minusMean, 0.1, 1, 0.15, zeros, 200	41,449
19	SARSA, 5, hypTanh, minusMean, 0.01, 0.95, 0.15, zeros, 0	41,343
20	SARSA, 5, hypTanh, minusMean, 0.01, 1, 0.15, zeros, 0	41,343
21	SARSA, 5, hypTanh, minusMean, 0.1, 1, 0.15, zeros, 0	40,997
22	SARSA, 5, hypTanh, minusMean, 0.1, 1, 0.1, zeros, 0	40,997
23	SARSA, 5, hypTanh, minusMean, 0.1, 0.95, 0.15, zeros, 0	40,873
24	SARSA, 5, hypTanh, minusMean, 0.1, 0.95, 0.1, zeros, 0	40,798
25	SARSA, 5, hypTanh, minusMean, 0.01, 0.95, 0.1, zeros, 0	40,214
26	SARSA, 5, hypTanh, minusMean, 0.01, 1, 0.1, zeros, 0	40,214
27	**QLearn, 5, hypTanh, minusMean, 0.1, 0.95, 200**	**36,675**
28	QLearn, 5, hypTanh, minusMean, 0.1, 1, 200	36,604
29	SARSA, 5, hypTanh, minusMean, 0.01, 1, 0.15, uniform01, 0	36,039
30	SARSA, 5, hypTanh, minusMean, 0.01, 0.95, 0.15, uniform01, 0	36,039
31	QLearn, 5, hypTanh, minusMean, 0.1, 0.95, 0	31,825
32	QLearn, 5, hypTanh, minusMean, 0.1, 1, 0	31,701
33	QLearn, 5, hypTanh, minusMean, 0.01, 0.95, 200	29,864
34	QLearn, 5, hypTanh, minusMean, 0.01, 1, 200	29,834
35	SARSA, 5, hypTanh, minusMean, 0.1, 0.95, 0.15, uniform01, 200	28,968
36	SARSA, 5, hypTanh, minusMean, 0.1, 1, 0.15, uniform01, 200	28,968
37	SARSA, 5, hypTanh, minusMean, 0.1, 0.95, 0.15, uniform01, 0	26,993
38	SARSA, 5, hypTanh, minusMean, 0.1, 1, 0.15, uniform01, 0	26,768
39	QLearn, 50, sigmoid, immediate, 0.1, 0.95, 0	26,287
40	QLearn, 50, sigmoid, mean, 0.1, 0.95, 0	25,858
⋮	⋮	⋮
49	**Greedy–GQ, 5, sigmoid123, minusMean, 0.1, 0.95, 0.1, 200**	**21,159**
50	SARSA, 25, hypTanh, minusMean, 0.01, 1, 0.15, uniform01, 200	20,655

Table 6.6: Top 50 best combination of hyper–parameters for 500,000 ticks' time–framed periods. Final profit and loss values (Final PL) presented in points.

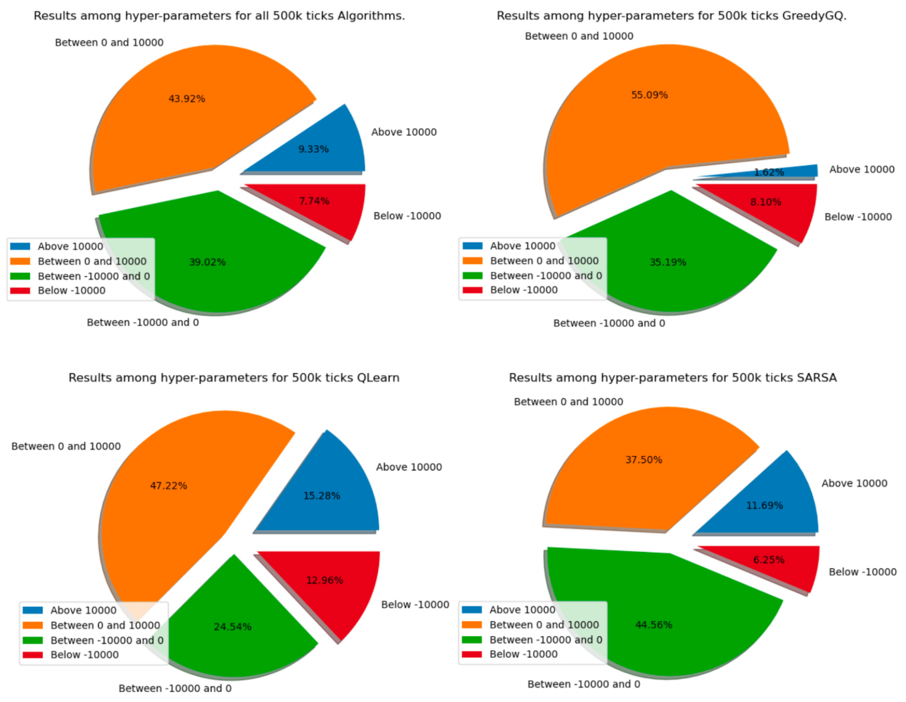

Figure 6.2: Four pie charts representing the proportions for "Above 10,000", "Between 0 and 10,000", "Between -10,000 and 0", and "Below -10,000" profit/loss results in points. These results are the entire possible combinations of hyper–parameters. Each combination was run from 17.02.21 to 15.02.22 in a 500k ticks' frame. Top left, top right, bottom left and bottom right plots represent 500k ticks all, Greedy–GQ, QLearn, and SARSA algorithms respectively.

We also exhibit pie charts, **Figure 6.2**, that illustrate the proportionality of the ultimate profit or loss in four distinct categories: "Above 10,000," "Between 0 and 10,000," "Between -10,000 and 0," and "Below -10,000." The outcome is identical to that given in the 60 minutes' time–framed intervals, such as establishing which reinforcement learning type is more stable among the permutations of hyper–parameters, even when they are not completely adjusted. Nota bene: we chose 10,000 points as our new border reference. Since the earnings and losses were greater in this scenario, 10,000 points were selected to disclose more plausible and intriguing information, and so, we can make the following inferences:

- Similar to what we reported for the analysis for 60 minutes' time frame, the 500k ticks for the top–left pie chart might have a similar rationale, but with wider boundary references. As can be seen, most outcomes are positive, with 53.25% scoring over 0 point and an amazing 9.33% above 10,000 points. On the other side, a small percentage is less than -10,000 points (7.74%). This number reveals that there are more parameter options for the positive side than for negative.
- When checking the remaining three pie charts, Greedy–GQ (top–right

pie) looks to be the least effective algorithm. This evaluation is possible because of the significant number of responses in the category “Between 0 and 10,000” (55.09%), whereas the category “Above 10,000” has extremely few responses (1.62%).

- Similar to the 60–minute case, the QLearn method is still the most stable one. This observation is based on the fact that it has the greatest number of positive parameter combinations (62.5%). Additionally, it has the highest number of hyper–parameter permutations, in both of the two positive boundary groups, than the other algorithms.
- Although the majority of SARSA’s results are negative (50.81%), when its hyper–parameters are appropriately set, it seems to be a robust algorithm with a large percentage of outcomes in the category “Above 10,000” (11.69%).

Furthermore, we cannot neglect the return on investment (ROI). Return on investment is another common financial performance metric that is used to determine the effectiveness or profitability of an investment. To calculate the ROI, we divide the PL of an investment by its initial investment capital, and the result is denoted either as a percentage or as a ratio:

$$ROI = \frac{PL}{\text{Initial Investment Capital}},$$

where Initial Investment Capital equals to 28,000 points in our cases.

Algorithm	**Period**	**ROI for 60 min**	**ROI for 500k ticks**
Buy & Hold	17.02.21 - 15.02.22	-37.71%	-37.71%
Greedy–GQ	17.02.21 - 15.02.22	60.34%	75.56%
QLearn	17.02.21 - 15.02.22	57.40%	130.98%
SARSA	17.02.21 - 15.02.22	38.48%	172.30%

Table 6.7: Return on investment (ROI) for the different and best set ups. The initial investment capital considered for our application cases was 28,000 points.

Finally, **Table 6.7** depicts that the reinforcement learning algorithms investigated easily outperform the buy–and–hold benchmark in both the 60 minutes’ and 500k ticks’ time–framed intervals. This superiority is observable even when these methods are not entirely fine–tuned or optimized, confirming the extraordinary capability of these approaches.

6.2 Discussion

In this early phase of our discussion, we will dive deep into the best models and their optimal hyper–parameters using a more statistical approach. To do that, we chose one best combination of hyper–parameters for each reinforcement learning technique for each time frame, i.e., the parameter mix that produced the greatest ROI for each time–framed interval, such the ones in **Table 6.8**.

Algorithm	Parameters	time–framed interval
Greedy–GQ	5, sigmoid, minusMean, 0.01, 0.95, 0.1, 200	60 min
Greedy–GQ	5, sigmoid123, minusMean, 0.1, 0.95, 0.1, 200	500k ticks
QLearn	5, sigmoid, minusMean, 0.01, 0.95, 0	60 min
QLearn	5, hypTanh, minusMean, 0.1, 0.95, 200	500k ticks
SARSA	5, hypTanh, minusMean, 0.01, 1, 0.1, zeros, 200	60 min
SARSA	5, hypTanh, minusMean, 0.1, 1, 0.1, uniform01, 0	500k ticks

Table 6.8: Optimal parameters for each reinforcement learning model from **Section 6.1**.

Next, we ran 500 simulations with diverse seeds employing the parameters we selected to assess their returns pathways to the end of their profit or loss. These simulations with different seeds are vital because they enable us to compel our algorithms to operate 500 stochastically distinct drawings, thereby eliminating the luck factor. Further, the key cause of worry in this stage of our dissertation, and the reason we want to exclude it from our conjecture, is the possibility of any relevant statistical attribute disqualifying any assumption of success of our research.

		Mean final return	
Algorithm	time–framed interval	50 seeds	500 seeds
Greedy–GQ	60 min	44,896	39,907
Greedy–GQ	500k ticks	49,159	44,622
QLearn	60 min	44,073	42,372
QLearn	500k ticks	64,675	65,942
SARSA	60 min	38,776	36,604
SARSA	500k ticks	76,246	74,744

Table 6.9: Mean final returns in points for different number of seeds (simulations). The values for 50 seeds were extracted from **Section 6.1**.

Table 6.9 exposes the mean final returns for our experiments. The 50 seeds' values were already known from the previous section, and we computed the 500 seeds' results applying the methodology pointed out at the beginning of this section. As can be confirmed, raising the number of simulations had no discernible effect on the algorithms' mean final return. Despite the fact that virtually all setups lost a few points, except for QLearn 500k ticks that increased, these optimal systems are sustainable and capable of retaining their statistical properties even when more statistically different samples are collected.

Next, the figures 6.3, 6.4, 6.5, 6.6, 6.7, and 6.8 illustrate how the 500 distinct seeds' simulations performed in 60–minute and 500k–tick frames for GreedyGQ, QLearn, and SARSA. Each seed has a return trajectory illustrated by the top–left line plots. The bottom–left line plot shows the average of all lines from the top–left plot. The distribution of each seed's final returns can be seen in the top–right histogram, while the bottom–right is a box and swarm plot, exhibiting the final returns for each seed.

Those studies play an important role in our assessments because the reinforcement learning framework attempts to maximize returns throughout each time step via actions and feedback. As a result, we have a clean and big

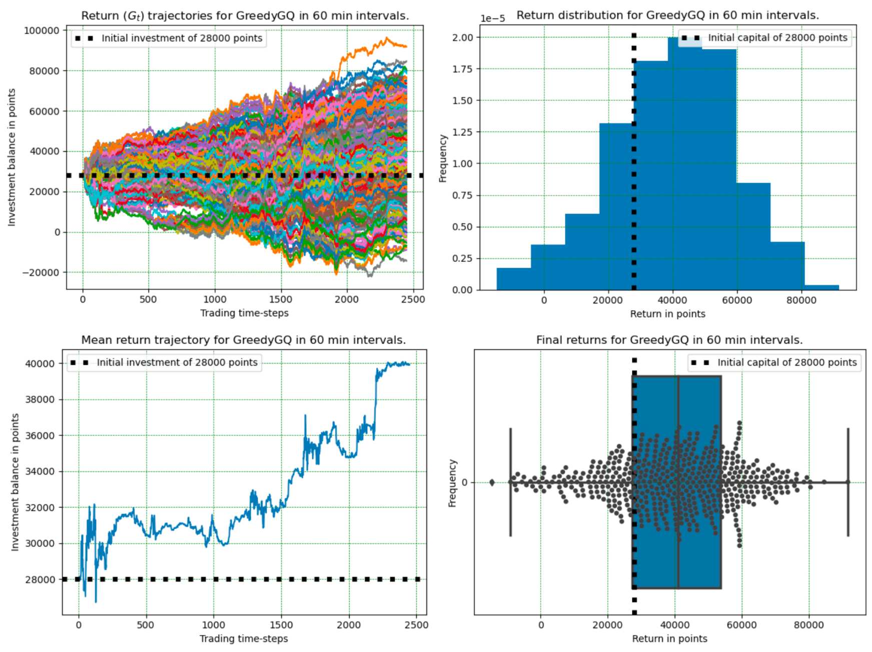

Figure 6.3: Lines, histograms, box, and swarm plots displaying the return results for 500 simulations of various seeds for the method Greedy–GQ across 60–minute time intervals.

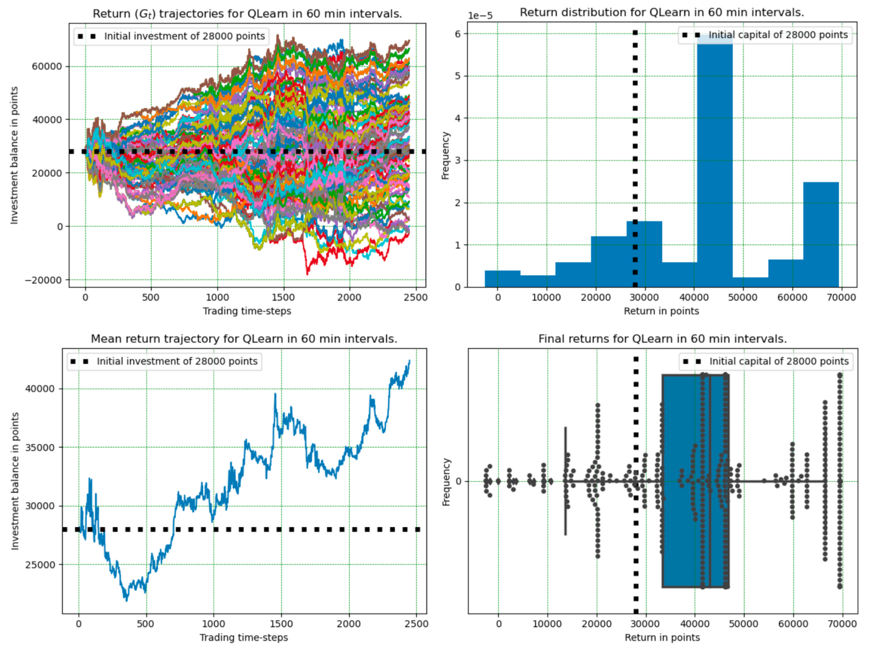

Figure 6.4: Lines, histograms, box, and swarm plots displaying the return results for 500 simulations of various seeds for the method QLearn across 60–minute time intervals.

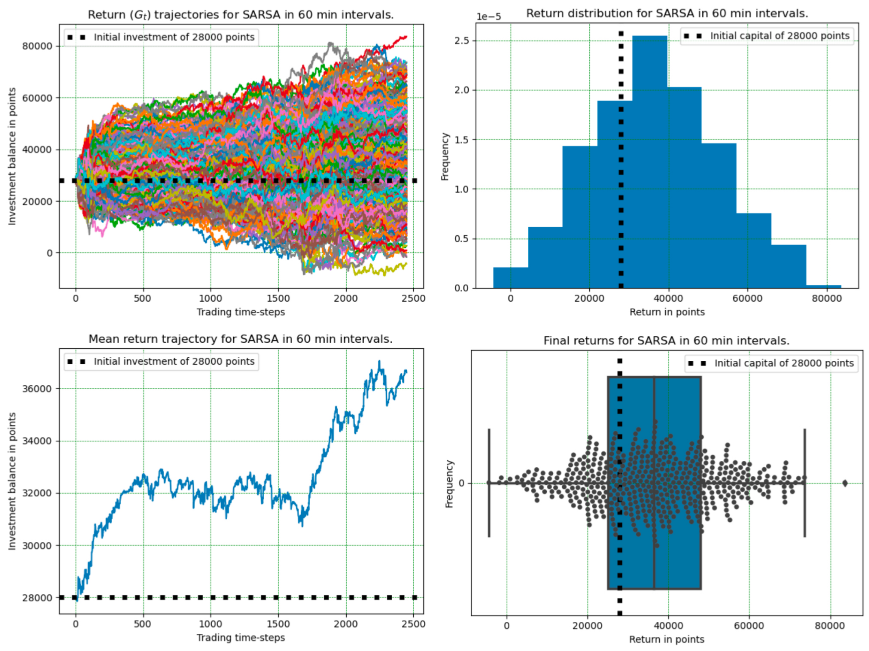

Figure 6.5: Lines, histograms, box, and swarm plots displaying the return results for 500 simulations of various seeds for the method SARSA across 60–minute time intervals.

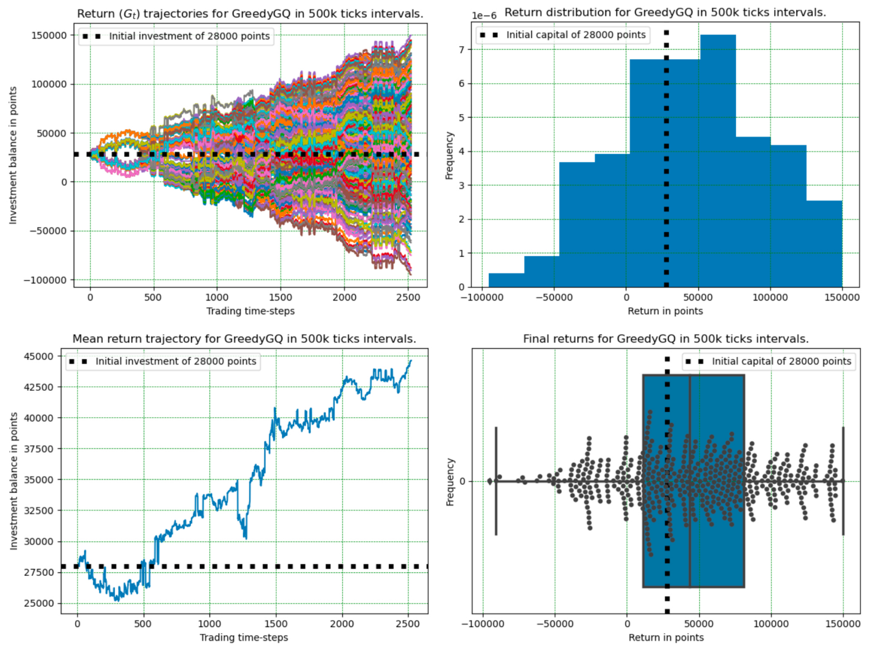

Figure 6.6: Lines, histograms, box, and swarm plots displaying the return results for 500 simulations of various seeds for the method Greedy–GQ across 500k–tick intervals.

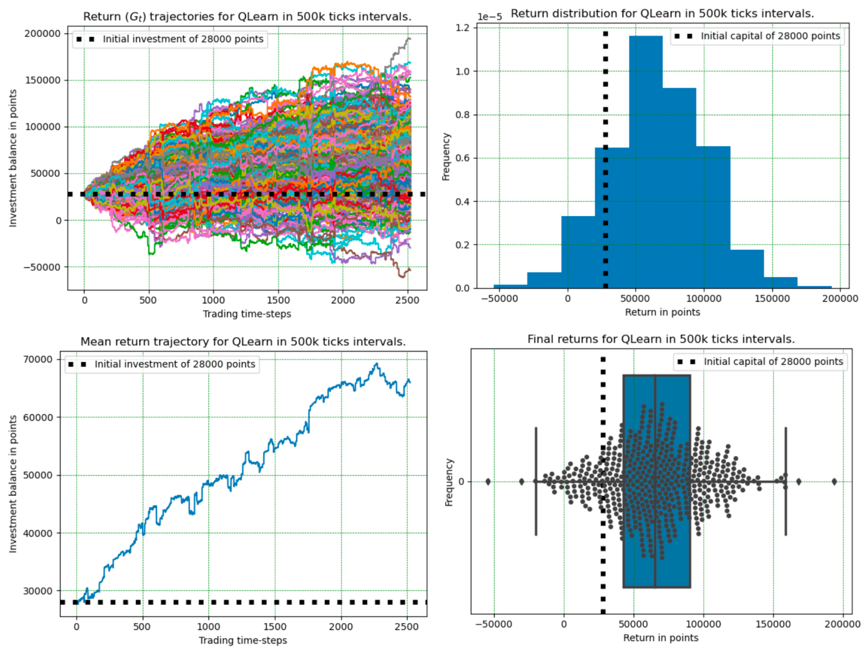

Figure 6.7: Lines, histograms, box, and swarm plots displaying the return results for 500 simulations of various seeds for the method QLearn across 500k–tick intervals.

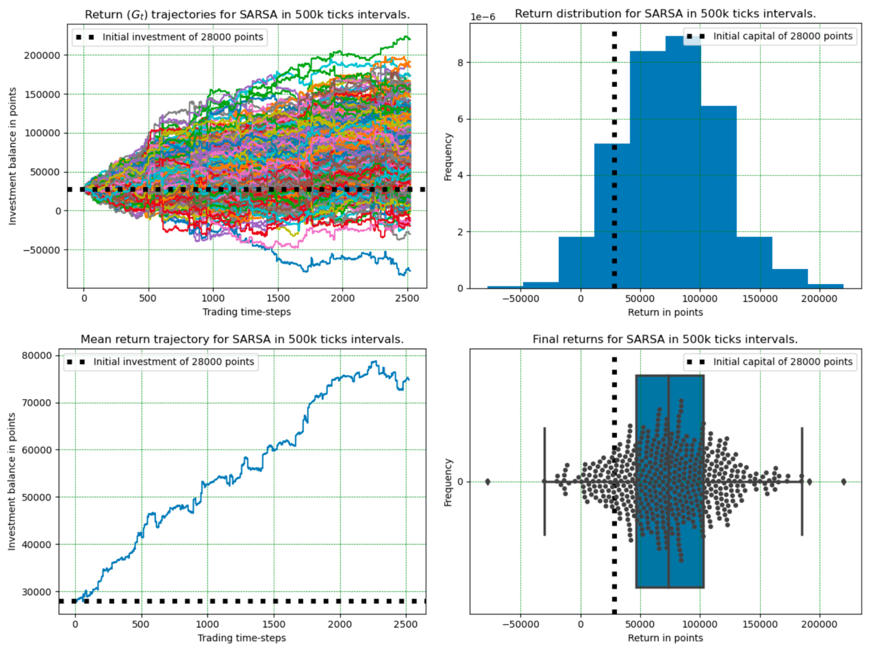

Figure 6.8: Lines, histograms, box, and swarm plots displaying the return results for 500 simulations of various seeds for the method SARSA across 500k–tick intervals.

picture for what occurred throughout the entire dynamics of our approaches to the various seeds, and based on a careful inspection of the figures, we may make the following observations:

- The top–left lines for all of these scenarios have significantly more return trajectories that result in profit, i.e., over 28,000 points, than those that end in loss. Additionally, the return pathways among different seeds have a tendency to go along the positive side throughout the entire window of our experimentation. This positive dominance emphasizes the model's ability to learn an optimal behaviour in such a manner that it avoids actions that might cut down its returns. Conversely, we need to recognize fewer trajectories running along the negative side, with some outliers with high variations. These circumstances may indicate that the model sometimes, albeit less often than in the positive case, struggles to learn to acquire optimal actions. Besides, we can pay attention to certain lines that fall to zero, that is, that lose 100% of the starting capital, before rebounding or growing losses, suggesting that the initial capital level (28,000 points) may be insufficient. Since we are analyzing the model's applicability and learn–ability, not the most adjusted initial investment capital, this supposed inaccuracy requires supplementary investigation, which is outside of the scope of this thesis.

- We can find out that all systems arise in consistent earnings by analyzing the average return trajectory. Indeed, the bottom–left line plot for all approaches includes periods of losses, sidewalks, but gains take place more often. These patterns are an excellent demonstration of how the methodologies successfully adapt and learn in response to changing market conditions over time.

- The histogram plots (top–left plots) are also an excellent source of knowledge showing how the distribution of the final returns ended up. These information explains us if the model is more likely to generate profits or losses due to the fact that they allow us to see their mean and the spread of the results. Once more, all ways are likely to produce more profits than losses, since all mean values are clearly found after the starting capital value of 28,000 points, and so are the majority of outcomes.

- The box and swarm plots are powerful tools for graphically presenting the dispersion and skewness of the final returns statistics, such as frequency location, median, minimum and maximum values, quartiles, outliers, and more. We can assess that some of these plots, especially in **Figures 6.4, 6.7, 6.8**, provide additional evidence for the algorithms' robustness, suggesting that 75% of all final return occurrences exceed the initial capital of 28,000 points. In fact, the initial capital line of 28,000 points for all figures are either located before the 25% quartile, or in its proximity between 25% and 50%, evidencing that there are more occurrences in the positive side than in the negative. As a result, we have a greater probability of profit than loss. It is worth emphasizing that certain systems have minimum and maximum quantities that are distinctly lower and/or higher than its mean value. This indicates that such algorithms exhibit

	60 min			500k ticks		
	Greedy–GQ	**QLearn**	**SARSA**	**Greedy–GQ**	**QLearn**	**SARSA**
count	500	500	500	500	500	500
mean	39,907	42,372	36,604	44,622	65,942	74,744
std	**19,038**	**17,070**	**16,050**	**51,556**	**35,266**	**41,195**
min	-14,505	-2,515	-4,300	-95,165	-53,975	-77,620
25%	27,350	33,350	25,088	11,392	42,651	46,577
50%	41,090	42,992	36432	43,627	65,327	73,465
75%	53,742	46,783	47,915	81,028	90,083	102,510
max	91,665	69,455	83,610	149,710	193,735	220,090

Table 6.10: Descriptive statistics for each optimal method.

a greater degree of variation than others. Because of the considerable relevance of this issue, we shall analyze it in depth with the help of **Table 6.10**.

Table 6.10 shows the descriptive statistics for each optimal reinforcement learning approach. It is basically a summary statistic that quantifies the characteristics of the final returns. Standard deviation measurements are more valuable to us at this point of our dissertation. The remaining numbers are visually appealing, but have previously been studied in the context of the prior charts. The standard deviation is an analytical term that points out the degree of variance in a group of events. As detected previously, some algorithms, such as Greedy–GQ 500k ticks (6.6) and SARSA 500k ticks (6.8), have a larger variance than others, for example, QLearn 60min (6.4) and SARSA 60min (6.5). These outputs that deviate significantly from its mean consequently present a high standard deviation as well. In fact, Greedy–GQ 500k ticks and SARSA 500k ticks exhibit standard deviations of 51,556 and 41,195, respectively, compared with 17,070 and 16,050 for QLearn 60min and SARSA 60min. Despite the high standard deviation for SARSA 500k ticks, just one outlier example was on the extreme negative side; the rest were on the positive side, and even over 75% of their final outcomes were on the profit side of their box plot (6.8). Thus, we conclude that this high standard deviation metric does not invalidate the success of our methods; rather, it highlights a key trade–off: if we want to set up a reinforcement learning algorithm to yield higher revenues, we must be willing to tolerate a higher risk of loss sometimes.

Although the purpose of this research is not to investigate ways for avoiding or minimizing deviations, it is interesting to mention one very elegant strategy from [Cor+19, p. 12]. The author creates a functional financial trading system in that study, which runs the same algorithm several times in parallel to obtain the average action for each time step. This concept is used to eliminate outlier actions and hence might also have the potential to put an end to outlier returns with large losses.

Inevitably, at this moment of our investigation, one challenging question may surrounds our consciousness:

Which optimal model appears to be the most appropriate?

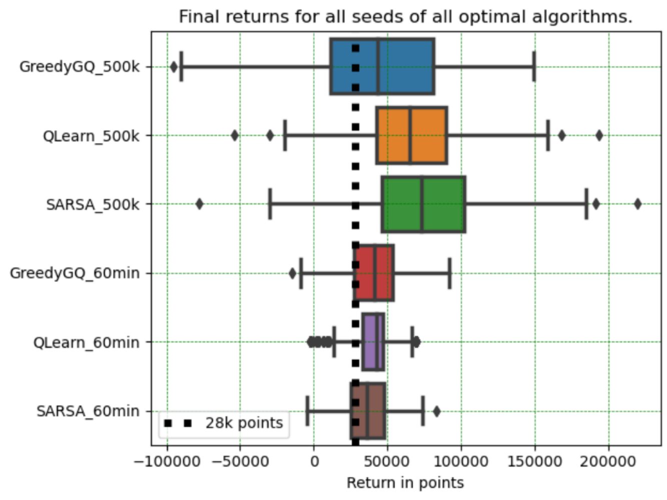

Figure 6.9: All optimal methods together.

The answer to this query appears to be tied to the style of the human investor behind the reinforcement learning algorithms. Put differently, the settings of a reinforcement learning algorithm must be determined by a human investor that can have an aggressive profile: willing to take on higher risks in pursuit of greater profits; or a conservative style: willing to accept lower but sustained returns of diminished risks. Therefore, **Figure 6.9** may assist us in overcoming this inescapable doubt by identifying the suitable options for these two investor kinds outlined.

The box plots in **Figure 6.9** depict all of our considered optimal scenarios. Typically, the vertical black dotted line determines the initial capital of 28,000 points. As we can see, models with time intervals of 500k ticks are more likely to have a broader box plot than models with time periods of 60 minutes. Remarkably, the 500k periods still exhibit higher mean values for final returns than 60 minutes' period, proving that even with a bigger volatility, they still generate higher earnings. This interpretation corroborates what we previously stated about the risk–reward trade–off. Thus, a human investor seeking increased earnings at the expense of possible high risks is more inclined to choose 500k time–framed intervals as the setting for their reinforcement learning environment. Considering the type of reinforcement learning technique, this aggressive investor may prefer SARSA, as the starting capital line is at the furthest point of the left side from the 25% quartile than in Greedy–GQ and QLearn. As a result, SARSA offers a greater likelihood of profitability than the others. Furthermore, the mean value for SARSA 500k ticks is further toward the right side than the others, signifying bigger earnings.

For a more conservative investor, a time–framed interval of 60 minutes might be favorable for the reinforcement learning environment. Indeed, the box plots are substantially narrower than the 500k–tick ones, implying that there is less volatility, i.e., less risks as explained before. Nonetheless, these time intervals yield considerable rewards with a high probability of success, as the initial capital line remains close to the 25% quartile. Since the beginning capital line is farther away from the left side of the 25% quartile in QLearn than it is in Greedy–GQ or SARSA, this conservative investor may choose QLearn as their preferable reinforcement learning technique. As a result, QLearn is more likely to succeed financially than the others. Furthermore, the mean value for QLearn 60 minutes is more to the right side than the others, offering a higher potential of profits.

Can we have consistent profits using reinforcement learning in the financial market?

Up to this point, we have conducted a thorough examination of the theory, application, and analysis of the findings. Returning to the central research question of this master's thesis, which is repeated above, it seems that the profitability of the reinforcement learning techniques applied to financial trading systems is not only plausible, but also reliable. It is important to emphasize how effectively the functional approximation reinforcement learning approaches performed in our problematic scenario. All algorithm types, including on–policy SARSA and off–policy Q–Learning and Greedy–GQ, outperformed the buy–and–hold benchmark, bringing about significant profits. Impressively, even Q–Learning, a functional approximation reinforcement learning methodology that lacks a technical proof of convergence (see [SB19, Chapter 11] for a more theoretical debate) performed as well as the top approaches, perhaps the best. This is not entirely surprising given the widespread usage and success of function–approximated Q–Learning in other problems in the scientific world. Although the convergence issue is a current open academic subject that needs deeper examination in future works, we may postulate that since Q–Learning does not operate randomly in the market, as SARSA does, it may reduce losses and safeguard the earnings' robustness. In other words, whereas Q–Learning and Greedy–GQ behave greedily in the market, always choosing the action with the best chance of receiving the highest reward, SARSA operates greedily in a ϵ–greedy manner, and hence somethings randomly act in the market. This might explain why Q–Learning and Greedy–GQ were the algorithms with the greatest ultimate return for the 60–minute time periods. SARSA, on the other hand, had the greatest final return in 500k–tick intervals because, conceivably, it may have accomplished a better job of exploring and exploiting the actions, thereby uncovering a greater return pathway than the off–policy solutions.

Another unresolved academic topic in the doctrines (see [SB19, pp. 249–254]) that we may touch on, but in an applied approach, is the discounted control setting in continuing on–policy function approximation reinforcement learning. The discount factor gamma, according to relevant literature, would have no influence on the findings. We somewhat confirmed this in our studies since two of our two best on–policy models had $\gamma = 1$ as their optimum parameter value. A parameter search among various gamma values revealed that when gamma is set to one, i.e., when it has no influence, the algorithm produces the strongest results, confirming what [SB19, p. 253] claims:

> "... *if we optimized discounted value over the on–policy distribution, then the effect would be identical to optimizing undiscounted average reward; the actual value of γ would have no effect. This strongly suggests that discounting has no role to play in the definition of the*

> *control problem with function approximation. One can nevertheless go ahead and use discounting in solution methods. The discounting parameter γ changes from a problem parameter to a solution method parameter! However, in this case we unfortunately would not be guaranteed to optimize average reward (or the equivalent discounted value over the on–policy distribution)."*

The in–depth theoretical justifications for this occurrence are beyond the scope of this thesis, and interested readers are referred to the conclusion of the debate in [SB19, p. 254].

CHAPTER 7

Conclusion

Reinforcement learning techniques constitute the subject of our thesis. In our point of view, because of the ability to learn by feed–backs, we proposed that reinforcement learning is the most appropriate way for capturing financial market nuances more quickly and efficiently than other artificial intelligence methods. Moreover, as discussed in the introductory chapter, it seems reasonable that the financial market is inefficient, based on common sense and *adaptive markets hypothesis* [Lo17], creating a potential for earning from algorithmic trading. Furthermore, we evidenced that reinforcement learning may carry out the same, if not better, degree of achievement than traditional stochastic optimal controls, especially when dealing with Markov decision problems without developing a theoretical model (transition probabilities and rewards) and without falling victim to the curse of expensive computations imposed by their high dimensionality.

We opened the first part of our thesis engaging in the theory of reinforcement learning. Chapter two presented the foundational concepts and reviewed key definitions, such as of stochastic process, state space, Markov process, transition probabilities and matrices, actions, policies, and rewards. We also solved an example of Markov decision process using classical exhaustive calculations and realized that this general approach is infeasible for the purposes of our investigation. Chapter three's primary topics included Markov decision process, value equations, Bellman's equations, and value and policy iteration and control. Besides that, we quickly transitioned to a more engineering–oriented exposition of the tabular reinforcement learning evolution, covering key computational concepts such as Dynamic Programming, Monte Carlo, Temporal Difference, k–step bootstrapping, and on–policy and off–policy controls. These approaches showed they can be taken advantage of when we are unaware of the transition probabilities for the system under control and thus produce the same high–level outcomes as the preceding older processes, but more efficiently, and with the added benefit of being applicable to other types of adversity. –While these developments are indispensable to the evolution of the theory, they were still inapplicable to our study case, which is why we needed to move on to the next chapter–. Given that we were dealing with a super–high–dimensional system with an incomplete state space, as well as unknown non–static transition probabilities and rewards, a function approximation reinforcement learning proposal appeared to be the most appropriate for our application problem, so its specificity was extensively addressed in chapter four.

Part two of this thesis focused on developing cutting–edge reinforcement learning algorithms for use in the B3's future financial market, particularly for trading the B3's mini index contract. Three function approximated reinforcement learning algorithms, SARSA, Q–Learning, and Greedy–GQ, were developed from scratch with three available controls: buy, do nothing, and sell. It is meaningful to point out that the algorithms were designed to execute just one transaction at a time, i.e., they were incapable of doubling positions. For the trading period from 17.02.2021 to 15.02.2022, we adopted pre–processing and hyper–parameter tuning in two distinct time–framed intervals of 500k–tick and 60–minute. Several basis functions (hyperTanh, sigmoid, and sigmoid123) and reward functions (minusMean, immediate, and mean) were also evaluated. Finally, the feature vector was represented in a block format; the other settings, as well as the mathematical formulation and theoretical framing, were explored thoroughly in chapter five. When all of those factors were combined, we could see that we had engineered an outstanding tool for overcoming any trading obstacles and perhaps making profits.

The last part of our research included the analysis of the findings and discussion. Everything came together to provide a satisfactory result: all of our panels made a large yearly return on investment (from 38% up to approx. 172%), outperforming the pre–established buy and hold benchmark of -37,71%. By evaluating the various algorithm configurations and the data behind numerous seed simulations, we confirmed that all three reinforcement learning approaches are not only lucrative but also trustworthy in determining the optimal action to take in the market. Furthermore, since all algorithms had generated robust gains, we concluded that the optimal model is a matter of personal preference for the human investor:

- aggressive: prepared to take on larger risks for higher profits;
- conservative: willing to accept lower but sustained revenues in exchange for less risk

As a result, the most appropriate algorithm for an aggressive investor profile would be SARSA with 500k–tick intervals. Conversely, the preferred design for a conservative investor style would be Q–Learning with 60–minute periods. Lastly, our central research question of whether consistent gains can be made in the financial market using reinforcement learning was answered affirmatively as a direct consequence of the trustworthy findings and substantial earnings.

Finally, as we also discussed in the analytical section, there are unresolved academic topics in reinforcement learning theory that might be beneficial to address in future works, which might improve the algorithms for trading purposes too. It is well known that the Q–Learning off–policy technique for function approximating reinforcement learning does not have a strong convergence guarantee, although its application delivers robust results. To address this problem, the Greedy–GQ method was developed; however, as shown, this novel off–policy with better convergence security did not perform as well as desired when compared with Q–Learning findings. Thus, an off–policy approach with proven convergence properties and a capability to achieve higher results is still missing.

Appendices

APPENDIX A

Reinforcement Learning Algorithms

A.1 Environment

```python
# Author: Fabio Rodrigues Pereira
# E-mail: fabior@uio.no

import numpy as np
import pandas as pd
from datetime import datetime
from technicalAnalysis import bollingerBands, ema, lwma

# suppress warnings
import warnings

warnings.filterwarnings('ignore')


class Environment:
    """
    Class method that reproduces the B3 stock exchange dynamics.
    """

    @staticmethod
    def applyTA(data, freq, std_value=2.0, column_base='close'):
        """Module to compute some technical analysis metrics."""

        data = ema(
            data=data,
            freq=freq,
            column_base=column_base
        )

        data = bollingerBands(
            data=data,
            freq=freq,
            std_value=std_value,
            column_base=column_base
        )

        data = lwma(
            data=data,
            freq=freq,
            column_base=column_base
        )
        return data

    @staticmethod
    def ln(currentPrice, previousPrice):
        """Computing a feature."""

        return np.log(currentPrice / previousPrice)

    @staticmethod
    def cleanCurrentTrade():
        """Clean memory."""

        return {
            "time": [],
            "entryPrice": 0,
            "currentTradePLs": [0],
            "histTradePLs": [],
            "lnTradePLs": [],
        }

    def __init__(self, n, fileName="data/WING22/WING22_1min_OLHCV.csv",
                 initInvest=5600*5, seed=0):

        # seeding the experiment
        self.seed = seed
        if seed != 0:
            self.seed = seed
            np.random.seed(self.seed)

        self.n = n
        self.fileName = fileName
        self.initInvest = initInvest
        self.t = 1
        self.data = pd.read_csv(f"{self.fileName}", sep=";")

        self.S = self.data.iloc[: self.n + self.t, :]

        self.S = self.applyTA(
            data=self.S,
            freq=self.n + 1,
            std_value=2.0,
            column_base='close'
        )

        self.terminal = True if len(self.S) == len(self.data) else False

        self.entryPrice = 0
        self.tradeRandEpsilon = False
        self.tradeStatus = 0
        self.Rprime = self.initInvest
        self.tradePL = 0
        self.lnTradePL = 0

        self.tradeMemory = {
            "time": [],
            "entryPrice": 0,
            "currentTradePLs": [0],
            "histTradePLs": [],
            "lnTradePLs": [],
        }

```

```python
        self.histTradeMemory = []
        self.histRprime = []
        self.histTradePLs = []

    def runNext(self, A):
        """Run next time-step."""

        Sprime = self.data.iloc[: self.n + 1 + self.t, :]  # n=2 + 1 + t=1
        Sprime = self.applyTA(
            data=Sprime,
            freq=self.n + 1,
            std_value=2.0,
            column_base='close'
        )

        C = Sprime.iloc[-2, 3]
        Cprime = Sprime.iloc[-1, 3]

        try:
            timePrime = datetime.strptime(Sprime.index[-1],
                                          '%Y-%m-%d %H:%M:%S')
        except:
            timePrime = datetime.strptime(Sprime.index[-1],
                                          '%Y-%m-%d %H:%M:%S.%f')

        if A == -1:  # limit short
            if self.tradeStatus == 0:  # new trade opened
                self.tradeStatus = -1
                self.entryPrice = C
                tradePL = self.entryPrice - Cprime
                deltaTradePLs = tradePL - self.tradePL
                self.Rprime += deltaTradePLs
                self.tradePL = tradePL
                self.lnTradePL = self.ln(Cprime, abs(self.entryPrice))

                self.saveNewTrade(
                    timePrime=timePrime,
                    entryPrice=self.entryPrice,
                    tradePL=self.tradePL,
                    lnTradePL=self.lnTradePL
                )

            elif self.tradeStatus == 1:  # current trade closed
                self.histTradePLs.append(self.tradePL)
                self.tradeStatus = 0
                self.entryPrice = 0
                self.tradePL = 0
                self.Rprime += self.tradePL
                self.lnTradePL = 0
                self.histTradeMemory.append(self.tradeMemory)
                self.tradeMemory = self.cleanCurrentTrade()

        elif A == 0:  # do nothing
            if self.tradeStatus == 1:  # on a short trade
                tradePL = self.entryPrice + Cprime
                deltaTradePLs = tradePL - self.tradePL
                self.Rprime += deltaTradePLs
                self.tradePL = tradePL
                self.lnTradePL = self.ln(Cprime, abs(self.entryPrice))

                self.saveUpdateTrade(
                    timePrime=timePrime,
                    tradePL=self.tradePL,
                    lnTradePL=self.lnTradePL
                )

            elif self.tradeStatus == -1:  # on a long trade
                tradePL = self.entryPrice - Cprime
                deltaTradePLS = tradePL - self.tradePL
                self.Rprime += deltaTradePLS
                self.tradePL = tradePL
                self.lnTradePL = self.ln(Cprime, abs(self.entryPrice))

                self.saveUpdateTrade(
                    timePrime=timePrime,
                    tradePL=self.tradePL,
                    lnTradePL=self.lnTradePL
                )

        elif A == 1:  # limit long
            if self.tradeStatus == 0:  # new trade opened
                self.tradeStatus = 1
                self.entryPrice = -C
                tradePL = self.entryPrice + Cprime
                deltaTradePLs = tradePL - self.tradePL
                self.Rprime += deltaTradePLs
                self.tradePL = tradePL
                self.lnTradePL = self.ln(Cprime, abs(self.entryPrice))

                self.saveNewTrade(
                    timePrime=timePrime,
                    entryPrice=self.entryPrice,
                    tradePL=self.tradePL,
                    lnTradePL=self.lnTradePL
                )

            elif self.tradeStatus == -1:  # current trade closed
                self.histTradePLs.append(self.tradePL)
                self.tradeStatus = 0
                self.entryPrice = 0
                self.tradePL = 0
                self.Rprime += self.tradePL
                self.lnTradePL = 0
                self.histTradeMemory.append(self.tradeMemory)
                self.tradeMemory = self.cleanCurrentTrade()

        if len(Sprime) == len(self.data):
            self.terminal = True
            if self.tradeStatus != 0:
                self.histTradePLs.append(self.tradePL)

        self.histRprime.append(self.Rprime)
        self.S = Sprime
        self.t += 1

    def saveNewTrade(self, timePrime, entryPrice, tradePL, lnTradePL):
        """Save data in memory."""

        self.tradeMemory["time"].append(timePrime)
        self.tradeMemory["entryPrice"] = entryPrice
        self.tradeMemory["currentTradePLs"].append(tradePL)
        self.tradeMemory["lnTradePLs"].append(lnTradePL)

    def saveUpdateTrade(self, timePrime, tradePL, lnTradePL):
        """Save data in memory."""

        self.tradeMemory["time"].append(timePrime)
        self.tradeMemory["currentTradePLs"].append(tradePL)
        self.tradeMemory["lnTradePLs"].append(lnTradePL)
```

A.2 Agents

```
# Author: Fabio Rodrigues Pereira
# E-mail: fabior@uio.no

import numpy as np
import torch as tr
from datetime import datetime


class QLearn:
    """
    Reinforcement Learning algorithm with function approximation
    and off-policy Q-learning control.
    """

    @staticmethod
    def ln(currentPrice: tr.Tensor, previousPrice: tr.Tensor) -> float:
        """Computing a feature by log-return."""

        return tr.log(currentPrice / previousPrice).item()

    @staticmethod
    def basisFunction(x: float, basisFctType: str = "sigmoid") -> float:
        """Basis function."""

        if basisFctType == "sigmoid123":
            a, b, c, d = 2, 1, 10**15, -1
            return (a / (1 + b * np.exp(-c * x))) - d

        elif basisFctType == "hypTanh":
            return np.tanh(x)

        elif basisFctType == "relu":
            return np.max([0, x])

        elif basisFctType == "sigmoid":
            return 1 / 1 + np.exp(-x)

        else:
            raise ValueError(f"basisFctType = {basisFctType} not recognized!")

    @staticmethod
    def rewardFunction(Gtplus1, rewardType):
        """Reward function."""

        if rewardType == "shapeRatio":
            r = np.mean(Gtplus1) / np.sqrt(np.var(Gtplus1))
            r = 0.0 if np.isnan(r) else r
            return r

        elif rewardType == "minusMean":
            return np.mean(Gtplus1)

        elif rewardType == "immediate":
            return 0.0

        elif rewardType == "mean":
            return 0.0

        else:
            raise ValueError(f"ERROR: rewardType {rewardType} "
                             f"not recognized...")

    @staticmethod
    def verbose(verbose, dfPrimeS, tradeStatus, primeTradeStatus, A, primeA,
                tau, primeTau, entryPrice, primeTradePL, lnPrimeTradePL,
                primeR):
        """Print steps of the algorithm."""

        if verbose is True:
            print(f"\nThe prime time is {dfPrimeS.index.to_list()[-1]}.")
            print(f"The prime closed price is {dfPrimeS.iloc[-1, 3]}.")
            print(f"\ntradeStatus = {tradeStatus}.")
            print(f"primeTradeStatus = {primeTradeStatus}.")
            print(f"A = {A}.")
            print(f"primeA = {primeA}.")
            print(f"tau = {tau}.")
            print(f"primeTau = {primeTau}.")
            print(f"entryPrice = {entryPrice}.")
            print(f"primeTradePL = {primeTradePL}.")
            print(f"lnPrimeTradePL = {lnPrimeTradePL}.")
            print(f"primeR = {primeR}.")

    @staticmethod
    def getCurrentDayTime(dataFrame):
        """Get current date and time."""

        currentDayTime = dataFrame.index[-1]
        return datetime.strptime(currentDayTime, '%Y-%m-%d %H:%M:%S')

    @staticmethod
    def initWeights(initType, dimensions):
        """Initializing weight vector."""

        if initType == "zeros":
            raise ValueError(f"ERROR: initType {initType} not allowed for "
                             f"off-policy methods!")

        elif initType == "uniform01":
            wVector = tr.zeros((dimensions, 1), dtype=tr.double)
            wVector = wVector.uniform_()
            return wVector

        else:
            raise ValueError(f"ERROR: initType {initType} not recognized!")

    def __init__(self, env, n, initInvest=5600*5, eta=0.01, gamma=0.95,
                 initType="uniform01", rewardType="minusMean",
                 basisFctType="sigmoid", typeFeatureVector="block",
                 lrScheduler=0, verbose=False, seed=0):

        # agent's variables
        self.env = env
        self.n = n                             # conjugated states
        self.initInvest = initInvest           # initial investment
        self.eta = eta                      # learning rate
        self.gamma = gamma                  # discount factor
        self.initType = initType            # zeros, uniform01
        self.rewardType = rewardType        # shapeRatio, mean, sum
        self.basisFctType = basisFctType    # hypTanh123, tanh, relu, sigmoid
        self.typeFeatureVector = typeFeatureVector  # block, nonblock
        self.lrScheduler = lrScheduler
        self.verbose = verbose

        # seeding the experiment
        if seed != 0:
            self.seed = seed
            np.random.seed(self.seed)
            tr.manual_seed(self.seed)

        self.spaceA = ["sell", "buy", "hold"]
        self.zeroVector = tr.zeros((1, n + 1), dtype=tr.double)

        if self.typeFeatureVector == "block":
            self.d = len(self.spaceA) * (self.n + 1)
        else:
            self.d = self.n + 1

        # init weight vector
        self.w = self.initWeights(self.initType, self.d)

        # memory
        self.memory = pd.DataFrame(
            columns=['open', 'high', 'low', 'close', 'volume',
                     'tradeStatus', 'A', 'primeA', 'tau', 'tradePL',
                     'primeR']
        )
        self.memoryW = None
        self.memoryNablaQ = None
        self.TDErrors = []
        self.t = 1

        # initial variables
        self.dfS = self.env.S
        self.S = tr.from_numpy(self.dfS.values)
        self.A = 0
        self.tradeStatus, self.tradePL = 0, 0

        f = self.getFeatureVector(
            S=self.S,                           # current state
            A=self.A,                           # action t
            tradeStatus=self.tradeStatus,       # tradeStatus
            tradePL=self.tradePL                # current trade profit
        )

        self.Q = (self.w.T @ f).item()
        self.nablaQ = f

    def getBasisVector(self, S, tradeStatus, tradePL):
        """Generate the basis vector."""

        b = tr.zeros((1, self.n + 1), dtype=tr.double)

        for i in reversed(range(2, self.n + 2)):
            currentPrice = S[-i+1, 3]
            previousPrice = S[-i, 3]

```

```
                b[0, -i] = self.basisFunction(
                    x=self.ln(currentPrice, previousPrice),
                    basisFctType=self.basisFctType
                )

        if tradeStatus == 0:
            b[0, -1] = self.basisFunction(
                x=0,
                basisFctType=self.basisFctType
            )

        else:
            b[0, -1] = self.basisFunction(
                x=tradePL,
                basisFctType=self.basisFctType
            )

        return b

    def getFeatureVector(self, S, A, tradeStatus, tradePL):
        """Generate the future vector."""

        b = self.getBasisVector(
            S=S,
            tradeStatus=tradeStatus,
            tradePL=tradePL
        )

        if self.typeFeatureVector == "block":
            f = tr.zeros((self.d, 1), dtype=tr.double)
            if A == -1:
                f = tr.hstack(
                    (b, self.zeroVector, self.zeroVector)
                ).T
            elif A == 0:
                f = tr.hstack(
                    (self.zeroVector, b, self.zeroVector)
                ).T
            elif A == 1:
                f = tr.hstack(
                    (self.zeroVector, self.zeroVector, b)
                ).T

            return f

        else:
            if tradeStatus == 0:
                b[0, -1] = self.basisFunction(
                    x=0,
                    basisFctType=self.basisFctType
                )
                return b.T

            else:
                b[0, -1] = self.basisFunction(
                    x=tradePL,
                    basisFctType=self.basisFctType
                )
                return b.T

    def greedyPolicy(self, S, As, tradeStatus, tradePL):
        """ Perform greedy policy."""

        Q, nablaQ = {}, {}
        for a in As:
            f = self.getFeatureVector(
                S=S,                            # current state
                A=a,                            # action t
                tradeStatus=tradeStatus,        # tradeStatus
                tradePL=tradePL                 # current trade PL
            )
            Q[a] = (self.w.T @ f).item()
            nablaQ[a] = f

        # checking equal Q values for different actions.
        equalQs = {k: v for k, v in Q.items()
                   if list(Q.values()).count(v) > 1}

        # if equalQ is detected, do not trade, i.e., select action 0.
        argmax = max(Q, key=Q.get) if len(equalQs) <= 1 else 0

        return argmax, Q[argmax], nablaQ[argmax]

    def spaceAs(self, tradeStatus):
        """Filter action space."""

        if tradeStatus == -1:
            return [0, 1]

        elif tradeStatus == 0:
            return [-1, 0, 1]

        elif tradeStatus == 1:
            return [0, -1]

    def saveMemory(self, dfS, tradeStatus, A, primeA, tau, tradePL,
                   primeR, nablaQ):
        """Save results in memory."""

        col = dfS.columns.to_list()
        extCols = ["tradeStatus", "A", "primeA", "tau", "tradePL", "primeR"]
        keys = col+extCols

        val1 = [dfS[k][-1] for k in dfS.keys().to_list()]
        val2 = [tradeStatus, A, primeA, tau, tradePL, primeR]
        vals = val1+val2

        timeIdx= dfS.index.to_list()[-1]

        memory = pd.DataFrame(vals).T
        memory.columns = keys
        memory.index = [timeIdx]
        self.memory = pd.concat([self.memory, memory], axis=0)

        if self.memoryW is None:
            mem = self.w.T.tolist()
            df = pd.DataFrame(mem)
            df.index = [self.memory.index.to_list()[-1]]
            self.memoryW = df

        else:
            mem = self.w.T.tolist()
            df = pd.DataFrame(mem)
            df.index = [self.memory.index.to_list()[-1]]
            self.memoryW = pd.concat([self.memoryW, df], axis=0)

        if self.memoryNablaQ is None:
            mem = nablaQ.T.tolist()
            df = pd.DataFrame(mem)
            df.index = [self.memory.index.to_list()[-1]]
            self.memoryNablaQ = df

        else:
            mem = nablaQ.T.tolist()
            df = pd.DataFrame(mem)
            df.index = [self.memory.index.to_list()[-1]]
            self.memoryNablaQ = pd.concat([self.memoryNablaQ, df], axis=0)

    def lrSchedulerFct(self):
        """Learning rate scheduler."""

        if self.lrScheduler != 0:
            if self.t % self.lrScheduler == 0:
                self.eta /= 2

    def run(self) -> None:
        """Run computations for the current time-step."""

        # get environment’s information
        self.env.runNext(A=self.A)
        dfPrime = self.env.S
        Sprime = tr.from_numpy(dfPrime.values)

        if self.rewardType == "mean":
            Rprime = np.mean(self.env.histRprime)
        else:
            Rprime = self.env.histRprime[-1]

        # compute the type of reward needed
        Rline = self.rewardFunction(
            Gtplus1=self.env.histRprime,
            rewardType=self.rewardType
        )

        # compute A’, Q’ and nabla’
        Aprime, Qprime, nablaQprime = self.greedyPolicy(
            S=Sprime,
            As=self.spaceAs(self.env.tradeStatus),
            tradeStatus=self.env.tradeStatus,
            tradePL=self.env.lnTradePL
        )

        # compute TD-error
        TDError = Rprime - Rline + self.gamma * Qprime - self.Q

        # reduce learning rate
        self.lrSchedulerFct()

        # update weights
        self.w += self.eta * TDError * self.nablaQ

        # save data
        self.saveMemory(
            dfS=dfPrime,
            tradeStatus=self.env.tradeStatus,
            A=self.A,
            primeA=Aprime,
            tau=len(self.env.tradeMemory["currentTradePLs"])-1,
            tradePL=self.env.tradePL,
            primeR=Rprime,
            nablaQ=self.nablaQ
        )

        # update variables for the next time-step
        self.dfS = dfPrime
        self.S = Sprime
        self.A = Aprime
        self.Q, self.nablaQ = Qprime, nablaQprime
        self.TDErrors.append(TDError)
        self.t += 1


class SARSA(QLearn):
    """
    Reinforcement Learning algorithm with function approximation
    and on-policy SARSA control.
    """

    @staticmethod
    def initWeights(initType, dimensions):
        """Initializing weight vector..."""

        if initType == "zeros":
            wVector = tr.zeros((dimensions, 1), dtype=tr.double)
            return wVector

        elif initType == "uniform01":
            wVector = tr.zeros((dimensions, 1), dtype=tr.double)
            wVector = wVector.uniform_()
            return wVector

        else:
            raise ValueError(f"ERROR: initType {initType} not recognized!")

    def __init__(self, env, n, initInvest=5600*5, eta=0.01, gamma=1.0,
                 epsilon=0.1, initType="uniform01", rewardType="minusMean",
                 basisFctType="sigmoid", typeFeatureVector="block",
                 lrScheduler=0, verbose=False, seed=0):
        super().__init__(env, n, initInvest, eta, gamma, initType,
                         rewardType, basisFctType, typeFeatureVector,
                         lrScheduler, verbose, seed)

        # epsilon for epsilon-greedy policy
        self.epsilon = epsilon

        # counters
        self.randEpsilon = 0
        self.countRandETrue = 0

    def epsilonGreedyPolicy(self, S, As, tradeStatus, tradePL):
        """Performing the epsilon-greedy policy."""

        self.randEpsilon = np.random.uniform(low=0, high=1, size=None)
        if self.epsilon >= self.randEpsilon:
            self.countRandETrue += 1
            a = np.random.choice(
                As,
                size=None,
                replace=False,
                p=None
            )
            f = self.getFeatureVector(
                S=S,                                # current state
                A=a,                                # action t
                tradeStatus=tradeStatus,            # tradeStatus
                tradePL=tradePL                     # current trade PL
            )
            q = (self.w.T @ f).item()
            return a, q, f

        else:
            Q, nablaQ = {}, {}
            for a in As:
                f = self.getFeatureVector(
                    S=S,                            # current state
                    A=a,                            # action t
                    tradeStatus=tradeStatus,        # tradeStatus
                    tradePL=tradePL                 # current trade PL
                )
                Q[a] = (self.w.T @ f).item()
                nablaQ[a] = f

            # checking equal Q values for different actions.
            equalQs = {k: v for k, v in Q.items()
                       if list(Q.values()).count(v) > 1}

            # if equalQ is detected, do not trade, i.e., select action 0.
            argmax = max(Q, key=Q.get) if len(equalQs) <= 1 else 0

            return argmax, Q[argmax], nablaQ[argmax]

    def run(self):
        """Run computations for the current time-step."""

        # get environment's information
        self.env.runNext(A=self.A)
        dfPrime = self.env.S
        Sprime = tr.from_numpy(dfPrime.values)

        if self.rewardType == "mean":
            Rprime = np.mean(self.env.histRprime)
        else:
            Rprime = self.env.histRprime[-1]

        # compute the type of reward needed
        Rline = self.rewardFunction(
            Gtplus1=self.env.histRprime,
            rewardType=self.rewardType
        )

        # compute A', Q' and nabla'
        Aprime, Qprime, nablaQprime = self.epsilonGreedyPolicy(
            S=Sprime,
            As=self.spaceAs(self.env.tradeStatus),
            tradeStatus=self.env.tradeStatus,
            tradePL=self.env.lnTradePL
        )

        # compute TD-error
        TDError = Rprime - Rline + self.gamma * Qprime - self.Q
```

```python

        # reducing learning rate
        self.lrSchedulerFct()

        # update weights
        self.w += self.eta * TDError * self.nablaQ

        # save data
        self.saveMemory(
            dfS=dfPrime,
            tradeStatus=self.env.tradeStatus,
            A=self.A,
            primeA=Aprime,
            tau=len(self.env.tradeMemory["currentTradePLs"])-1,
            tradePL=self.env.tradePL,
            primeR=Rprime,
            nablaQ=self.nablaQ
        )

        # update variables for the next time-step
        self.dfS = dfPrime
        self.S = Sprime
        self.A = Aprime
        self.Q, self.nablaQ = Qprime, nablaQprime
        self.TDErrors.append(TDError)
        self.t += 1


class GreedyGQ(QLearn):
    """
    Reinforcement Learning algorithm with function approximation
    and off-policy Greedy-GQ control.
    """

    @staticmethod
    def initKappa(initType, dimensions):
        """Initializing kappa weight vector."""

        if initType == "zeros":
            raise ValueError(f"ERROR: initType {initType} not allowed for "
                             f"off-policy methods!")

        elif initType == "uniform01":
            kappaVector = tr.zeros((dimensions, 1), dtype=tr.double)
            kappaVector = kappaVector.uniform_()

            return kappaVector

        else:
            raise ValueError(f"ERROR: initType {initType} not recognized!")

    def __init__(self, env, n, initInvest=5600*5, eta=0.01, gamma=0.95,
                 initType="uniform01", rewardType="minusMean", zeta=0.01,
                 basisFctType="sigmoid", typeFeatureVector="block",
                 lrScheduler=0, verbose=False, seed=0):

        super().__init__(env, n, initInvest, eta, gamma, initType,
                         rewardType, basisFctType, typeFeatureVector,
                         lrScheduler, verbose, seed)

        self.zeta = zeta
        self.kappa = self.initKappa(self.initType, self.d)

    def run(self):
        """Run computations for the current time-step."""

        # get environment’s information
        self.env.runNext(A=self.A)
        dfPrime = self.env.S
        Sprime = tr.from_numpy(dfPrime.values)

        if self.rewardType == "mean":
            Rprime = np.mean(self.env.histRprime)
        else:
            Rprime = self.env.histRprime[-1]

        # compute the type of reward needed
        Rline = self.rewardFunction(
            Gtplus1=self.env.histRprime,
            rewardType=self.rewardType
        )

        # compute A’, Q’ and nabla’
        Aprime, Qprime, nablaQprime = self.greedyPolicy(
            S=Sprime,
            As=self.spaceAs(self.env.tradeStatus),
            tradeStatus=self.env.tradeStatus,
            tradePL=self.env.lnTradePL
        )

        # compute TD-error
        vartheta = Rprime - Rline + self.gamma * Qprime - self.Q

        # reducing learning rate
        self.lrSchedulerFct()

        # update weights
        self.w += self.eta * (vartheta * self.nablaQ - self.gamma *
                              (self.kappa.T @ self.nablaQ)) * nablaQprime

        self.kappa += self.zeta * (vartheta -
                                   (self.kappa.T @ self.nablaQ)) * self.nablaQ

        # save data
        self.saveMemory(
            dfS=dfPrime,
            tradeStatus=self.env.tradeStatus,
            A=self.A,
            primeA=Aprime,
            tau=len(self.env.tradeMemory["currentTradePLs"]) - 1,
            tradePL=self.env.tradePL,
            primeR=Rprime,
            nablaQ=self.nablaQ
        )

        # update variables for the next time-step
        self.dfS = dfPrime
        self.S = Sprime
        self.A = Aprime
        self.Q, self.nablaQ = Qprime, nablaQprime
        self.TDErrors.append(vartheta)
        self.t += 1
```

Bibliography

[AG00] Ané, T. and Geman, H. 'Order Flow, Transaction Clock, and Normality of Asset Returns'. In: *The Journal of Finance* vol. 55, no. 5 (2000), pp. 2259–2284.

[Bai95] Baird, L. 'Residual Algorithms: Reinforcement Learning with Function Approximation'. In: *Machine Learning Proceedings 1995*. Ed. by Prieditis, A. and Russell, S. San Francisco (CA): Morgan Kaufmann, 1995, pp. 30–37.

[Bar93] Barnard, E. 'Temporal-Difference Methods and Markov Models'. In: *Systems, Man and Cybernetics, IEEE Transactions on* vol. 23 (Apr. 1993), pp. 357–365.

[BC12] Bertoluzzo, F. and Corazza, M. 'Testing Different Reinforcement Learning Configurations for Financial Trading: Introduction and Applications'. In: *Procedia Economics and Finance* vol. 3 (2012). International Conference Emerging Markets Queries in Finance and Business, Petru Maior University of Tîrgu-Mures, ROMANIA, October 24th - 27th, 2012, pp. 68–77.

[Bel52] Bellman, R. 'On the Theory of Dynamic Programming'. In: *Proceedings of the National Academy of Sciences* vol. 38, no. 8 (1952), pp. 716–719. eprint: https://www.pnas.org/content/38/8/716.full.pdf.

[CB21] Capasso, V. and Bakstein, D. *An Introduction to Continuous-Time Stochastic Processes*. Fourth edition. Theory, Models, and Applications to Finance, Biology, and Medicine. Cham, Switzerland: Birkhauser, 2021.

[CN05] Cuthbertson, K. and Nitzsche, D. *Quantitative Financial Economics*. Second Edition. Stocks, bonds and foreign exchange. West Sussex, England: John Wiley Sons, Ltd, 2005.

[Cor+19] Corazza, M. et al. *A comparison among Reinforcement Learning algorithms in financial trading systems*. Working Papers 2019:33. Department of Economics, University of Venice "Ca' Foscari", 2019.

[CS15] Corazza, M. and Sangalli, A. *Q-Learning and SARSA: a comparison between two intelligent stochastic control approaches for financial trading*. Working Papers 2015:15. Department of Economics, University of Venice "Ca' Foscari", 2015.

[DHB20] Dixon, M. F., Halperin, I. and Bilokon, P. *Machine Learning in Finance*. From Theory to Practice. Cham, Switzerland: Springer, 2020.

[GDH13] Geramifard, Dann, C. and How, J. P. 'Off-Policy Learning Combined with Automatic Feature Expansion for Solving Large MDPs Alborz'. In: 2013.

[Ger+11] Geramifard, A. et al. 'Online Discovery of Feature Dependencies.' In: Jan. 2011, pp. 881–888.

[Ger+13] Geramifard, A. et al. 'Batch-iFDD for representation expansion in large MDPs'. In: *Uncertainty in Artificial Intelligence - Proceedings of the 29th Conference, UAI 2013* (Sept. 2013).

[Gos15] Gosavi, A. *Simulation-Based Optimization*. Second Edition. Parametric Optimization Techniques and Reinforcement Learning. Missouri University of Science and Technology, Rolla, MO, USA: Springer, 2015.

[How64] Howard, R. A. *Dynamic programming and Markov processes*. English. M.I.T. Press Cambridge, Mass, 1964, p. 1 v.

[Kle20] Klenke, A. *Probability Theory*. Third edition. A Comprehensive Course. Mainz, Germany: Springer, 2020.

[Klo72] Klopf, A. H. 'Brain Function and Adaptive Systems: A Heterostatic Theory'. In: Bedford, MA, USA: Air Force Cambridge Research Laboratories, 1972.

[Klo75] Klopf, A. H. 'A Comparison of Natural and Artificial Intelligence'. In: *SIGART Bull.*, no. 52 (June 1975), pp. 11–13.

[Klo82] Klopf, A. H. *The Hedonistic Neuron: A Theory of Memory, Learning, and Intelligence*. Hemisphere Publishing Corporation, 1982.

[Lin17] Lindstrom, T. *Spaces*. An introduction to real analysis. Providence, Rhode Island: American Mathematical Society, 2017.

[Lo17] Lo, A. W. *Adaptive Markets*. Second Edition. Financial evolution at the speed of thought. United Kingdom: Princeton University Press, 2017.

[Mae+10] Maei, H. et al. 'Toward Off-Policy Learning Control with Function Approximation'. In: Aug. 2010, pp. 719–726.

[Min95] Minsky, M. 'Steps toward Artificial Intelligence'. In: *Computers Thought*. Cambridge, MA, USA: MIT Press, 1995, pp. 406–450.

[Mni+16] Mnih, V. et al. 'Asynchronous Methods for Deep Reinforcement Learning'. In: *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*. ICML'16. New York, NY, USA: JMLR.org, 2016, pp. 1928–1937.

[MT67] Mandelbrot, B. and Taylor, H. M. 'On the Distribution of Stock Price Differences'. In: *Operations Research* vol. 15, no. 6 (1967), pp. 1057–1062.

[Pra18] Prado, M. L. de. *Advances in Financial Machine Learning*. Hoboken, New Jersey: Wiley, 2018.

[RM51] Robbins, H. and Monro, S. ‘A Stochastic Approximation Method’. In: *The Annals of Mathematical Statistics* vol. 22, no. 3 (1951), pp. 400–407.

[Ros19] Ross, S. M. *Introduction to Probability Models.* Twelfth edition. Oxford, UK: Elsevier, 2019.

[Sam59] Samuel, A. L. ‘Some Studies in Machine Learning Using the Game of Checkers’. In: *IBM Journal of Research and Development* vol. 3, no. 3 (1959), pp. 210–229.

[SB19] Sutton, R. and Barto, A. *Reinforcement Learning. An introduction.* Second edition. Adaptive computation and machine learning series. Cambridge, MA: The MIT Press, 2019.

[SB81] Sutton, R. and Barto, A. ‘Toward a modern theory of adaptive networks: Expectation and prediction.’ In: vol. 88 (1981).

[Sut84] Sutton, R. S. ‘Temporal credit assignment in reinforcement learning’. PhD thesis. Cambridge, MA: University of Massachusetts Amherst, 1984.

[Sut88] Sutton, R. S. ‘Learning to Predict by the Methods of Temporal Differences’. In: *Machine Learning* vol. 3 (1988), pp. 9–44.

[Sze09] Szepesvári, C. *Algorithms for Reinforcement Learning.* 2009.

[Wal11] Walsh, J. B. *Knowing the Odds: An Introduction to Probability.* Graduate Studies in Mathematics: v. 139. USA: American Mathematical Society, 2011.

[Wat89] Watkins, C. J. C. H. ‘Learning from Delayed Rewards’. PhD thesis. Cambridge, UK: King’s College, May 1989.

[Wit76] Witten, I. H. ‘Learning to Control.’ PhD thesis. University of Essex, 1976.

[Wit77] Witten, I. H. ‘An adaptive optimal controller for discrete-time Markov environments’. In: *Information and Control* vol. 34, no. 4 (1977), pp. 286–295.