BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

REINFORCEMENT LEARNING FOR AUTOMATED STOCK PORTFOLIO ALLOCATION

VYUŽITÍ ZPĚTNOVAZEBNÉHO UČENÍ PRO AUTOMATICKOU ALOKACI AKCIOVÉHO PORTFOLIA

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **ZDENĚK LAPEŠ**
AUTOR PRÁCE

SUPERVISOR **doc. RNDr. MILAN ČEŠKA, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2023

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Bachelor's Thesis Assignment

148202

Institut:	Department of Intelligent Systems (UITS)
Student:	**Lapeš Zdeněk**
Programme:	Information Technology
Specialization:	Information Technology
Title:	**Reinforcement Learning for Automated Stock Portfolio Allocation**
Category:	Artificial Intelligence
Academic year:	2022/23

Assignment:

1. Study the state-of-the-art methods for automated stock portfolio allocation. Focus on the methods based on reinforcement learning and planning in Markov Decision Processes.
2. Experimentally evaluate selected open access tools for automated portfolio allocation including e.g. FinRL-Meta and identify their weak points.
3. Propose and implement improvements of a selected method/tool allowing to mitigate theseweak points.
4. Using suitable benchmarks and datasets, perform a detailed experimental evaluation of the implemented improvements with the focus on the portfolio allocation returns.

Literature:
Rao A., Jelvis T., Foundations of Reinforcement Learning with Applications in Finance. 1st Edition, Taylor & Francis 2022
* Li, Xinyi and Li, Yinchuan and Zhan, Yuancheng and Liu, Xiao-Yang, Optimistic Bull or Pessimistic Bear: Adaptive Deep Reinforcement Learning for Stock Portfolio Allocation, In ICML 2019.
* Liu X.-Y. Rui J. Gao J. aj.: FinRL-Meta: A Universe of Near-Real Market Environments for Data-Driven Deep Reinforcement Learning in Quantitative Finance. Workshop on Data Centric AI 35th Conference on Neural Information Processing Systems at NeurIPS 2021.
* Mao Guan and Xiao-Yang Liu. 2021. Explainable Deep Reinforcement Learning for Portfolio Management: An Empirical Approach. In ICAIF 2021.

Requirements for the semestral defence:
Items 1, 2, and partially 3.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Češka Milan, doc. RNDr., Ph.D.**
Head of Department:	Hanáček Petr, doc. Dr. Ing.
Beginning of work:	1.11.2022
Submission deadline:	10.5.2023
Approval date:	3.11.2022

Abstract

This thesis is focused on the topic of reinforcement learning applied to a task of portfolio allocation. To accomplish this objective, the thesis first presents an overview of the fundamental theory, which includes the latest value-based and policy-based methods. Following that, the thesis describes the Stock portfolio environment, and finally, the experimental and implementation details are presented. The creation of datasets is discussed in detail, along with the rationale and methodology behind it. The RL agent is then trained and tested on three datasets, and the results obtained are promising and outperform common benchmarks. However, it was discovered that the annual return of the agent is still not better than the returns generated by the world's top investors. The pipeline was implemented in Python 3.10, and technology from Weights & Biases was used to monitor all datasets, models, and hyperparameters. In conclusion, this work represents a significant step forward in the development of more effective RL agents for financial investments, with the potential to exceed even the performance of the world's greatest investors.

Abstrakt

Tato práce je zaměřena na téma posilovacího učení aplikovaného na úlohu alokace portfolia. K dosažení tohoto cíle práce nejprve uvádí přehled základní teorie, která zahrnuje nejnovější metody založené na hodnotách a politikách. Následně je v práci popsáno prostředí portfolia Stock a nakonec jsou uvedeny podrobnosti o experimentu a implementaci. Podrobně je rozebrána tvorba datových souborů a její zdůvodnění a metodika. RL agent je poté vycvičen a otestován na třech datových sadách a získané výsledky jsou slibné a překonávají běžné benchmarky. Bylo však zjištěno, že roční výnos agenta stále není lepší než výnosy generované nejlepšími světovými investory. Pipeline byla implementována v jazyce Python 3.10 a ke sledování všech datových sad, modelů a hyperparametrů byla použita technologie Weights & Biases. Závěrem lze říci, že tato práce představuje významný krok vpřed ve vývoji efektivnějších RL agentů pro finanční investice, kteří mají potenciál překonat i výkonnost nejlepších světových investorů.

Keywords

artificial intelligence, reinforcement learning, actor-critic, neural networks, stock portfolio allocation, portfolio allocation theory, stock market

Klíčová slova

umělá inteligence, posilované učení, aktor-kritik, neuronové sítě, alokace akciového portfolia, teorie alokace portfolia, akciový trh

Reference

LAPEŠ, Zdeněk. *Reinforcement Learning for Automated Stock Portfolio Allocation.* Brno, 2023. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor doc. RNDr. Milan Češka, Ph.D.

Rozšířený abstrakt

Tato práce se zaměřuje na využití posilovaného učení pro alokaci akciového portfolia. V úvodu jsou vysvětleny základní pojmy a algoritmy posilovaného učení, včetně *Markovových procesů*, které jsou jeho základem. Poté si řekneme o *Value based* metodách, mezi které patří Dynamic Programming, Monte Carlo a Temporal Difference. Value-Based metody se snaží naučit, jaká je kvalitu stavů, ve kterých se agent nacházel, a nebo kvalitu akcí provedených agentem v daných stavech. Dále jsou vysvětleny *Policy based* metody, mezi které patří Stochastic Policy Gradient a REINFORCE (Monte Carlo Policy Gradients), které se snaží najít nejlepší rozhodovací strategii agenta, při které bude schopen dosáhnout svého cíle. Na závěr zmíníme teorii o hlubokém posilovaném učení, která využívá neuronové sítě pro naučení vhodné policy a také vysvětlíme koncept učení *Aktor-kritik*, který kombinuje Value based a Policy based metody pro trenování agenta. V naší práci jsme použili následující algoritmy A2C, SAC, DDPG, PPO a TD3.

V dalších kapitolách se věnujeme problematice *alokace portfolia* a přiblížíme přístup k trenování agenta s posilovaným učením. Popíšeme také, jak jsme přistupovali k vytváření datasetu, včetně výběru jednotlivých indikátorů z následujicích analýz *fundamentalní* a *technickou* k popisu finančního prostředí. Poté vysvětlíme implementaci *prostředí pro alokaci portfolia*, stanovíme akční prostor, který vymezuje možnosti agenta, v úkonu vybírání, jak moc bude která spolenost nakoupena do portfolia. V neposlední řadě vysvětlíme funkci odměn, která slouží k ohodnocování kvality provedených rozhodnutí agentem v prostředí.

Dále představujeme způsob, kterým jsme vybírali hyperparametry, které pomáhají agentovi učit se chovat v daném prostředí. Hyperparametry volíme pomocí tzv metody *hyperparameters tuning*. A rozebereme zde seznamem parametrů, nad kterými se tuning prováděl, a vysvětlíme, jak a proč byli hodnoty těchto parametrů ohraničeny. Jako další experiment představujeme zkoumání vlivu datasetu na výkon agenta. V této části jsou představeny výsledky agentů, kteří byli natrénováni na třech různých datasetech a porovnáme, jak se agentovi dařilo v prostředí chovat. Dále je zde prezentován vliv každého datasetu na výkon agenta. Agenti jsou natrénováni na těchto datasetech *fundamentální dataset*, *technický dataset* a *kombinace* a jsou porovnáni mezi sebou. Následuje popis experimentu zaměřeného na *robustnost modelu*. Nejprve je proveden výběr nejlepší konfigurace z předchozího experimentu, kde byl prováděn hyperparametrický tuning. Následně natrénujeme několik modelů na stejné konfiguraci hyperparametrů a ty potom provnáme podle metriky *výkon portfolia*, která vyjadřuje, jak velké zhodnocení dosáhla daná skladba akcií v portfoliu.

V posledním experimentu jsou naše natrénované modely porovnány s *veřejně dostupným* modelem využívajícím posilované učení pro alokaci portfolia, dále s standardními *strategiemi* pro alokaci portfolia, jako je *Maximalizace Sharpeho poměru* nebo *Minimalizace rizika* portfolia, a nakonec s *indexy*, jako jsou *Dow Jones Index*, *Nasdaq Composite*, *S&P500* a *Russell 2000*. A experimentálně bylo zjištěno, že agent je schopen porazit všechny porovnávané strategie a většinu porovnávaných indexů ve zhodnocení portfolia. Také je dikutováno, jak si agent vedl, např. v době propadů na burze, a jakých výsledků bylo možné dosáhnout průměrným ročním zhodnocením.

Experimenty byly prováděny z účelem vytvořit a zjistit, jakých výsledků může agent dosáhnout, zda bude možné používat daného agenta na reálné burze. Z výsledků výše popsaných experimentů jsme experimentálně zjistili, že agent je schopen provádět alokaci portfolia dostatečně přesně, aby porazil standardní strategie a většinu porovnávaných indexů. Během experimentů jsme zjistili, jakých výsledků je možné agentem dosáhnout. Všechny výsledky jsou prezentovány veřejně ze stránky *Weights & Biases*, kde jsou uloženy všechny datasety a modely, a také současně všechny logované údalosti při tréninku a testování mod-

elů. Tím jsme chtěli zajistit reprodukovatelnost výsledků a snadnou dostupnost pro další výzkumníci v této oblasti.

Reinforcement Learning for Automated Stock Portfolio Allocation

Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of Mr. Doc. RNDr. Milan Češka Ph.D. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Zdeněk Lapeš
June 12, 2023

Acknowledgements

I would like to thank my supervisor, Mr. Milan Češka, for his guidance and support during the preparation of this thesis. I would also like to thank my family and friends for their support.

Contents

1 Introduction **4**
- 1.1 Background . . . 4
- 1.2 Limitations . . . 6
- 1.3 Aim of the Thesis . . . 6

2 Reinforcement Learning **7**
- 2.1 Introduction . . . 8
- 2.2 Markov Theory . . . 9
  - 2.2.1 Markov Process . . . 10
  - 2.2.2 Markov Reward Process . . . 12
  - 2.2.3 Markov Decision Process . . . 14
- 2.3 Value-based learning . . . 18
  - 2.3.1 Dynamic Programming . . . 18
  - 2.3.2 Monte Carlo Methods . . . 20
  - 2.3.3 Temporal-Difference Methods . . . 21
  - 2.3.4 Function Approximation . . . 23
- 2.4 Policy-based learning . . . 25
  - 2.4.1 Stochastic Policy Gradient Methods . . . 25
  - 2.4.2 REINFORCE: Monte Carlo Policy Gradient . . . 27
  - 2.4.3 Actor-Critic Policy Gradient Methods . . . 28

3 Stock Portfolio Allocation **30**
- 3.1 Data Engineering . . . 30
  - 3.1.1 Data Collection . . . 31
  - 3.1.2 Data Cleaning . . . 31
  - 3.1.3 Feature Engineering . . . 31
  - 3.1.4 Dataset Splitting . . . 36
- 3.2 Environment Modeling . . . 37
  - 3.2.1 Portfolio Management Task . . . 37
  - 3.2.2 How we determine agent performance . . . 39

4 Experiments and Results **40**
- 4.1 Hardware & Tools . . . 40
  - 4.1.1 Hardware . . . 40
  - 4.1.2 Weights & Biases . . . 41
- 4.2 Focus of the experiments . . . 41
- 4.3 First experiment: Hyperparameter Tuning . . . 42
  - 4.3.1 Time Steps . . . 42

4.3.2 Selection of hyperparameters . . . 42
4.3.3 Hyperparameter Sweep Results . . . 43
4.4 Datasets Impact . . . 44
4.5 Testing of Robustness . . . 45
4.6 Baselines & Benchmarks . . . 46
4.6.1 Comparing Portfolio Performance: Analysis from 2017 to 2022 . . . 46
4.6.2 Drawdowns Analysis from 2017 to 2022 . . . 47
4.6.3 Performance of Different Portfolios: Monthly and Annualy . . . 48
4.7 Portfolio Metrics . . . 50
4.7.1 Comparison with State-of-the-Art AI4Finance . . . 50
4.7.2 Comparison with Indexes and Strategies . . . 51
4.8 Summary . . . 52

5 Conclusions **54**

Bibliography **55**

A Reinforcement Learning Algorithms **57**
A.1 Value Iteration Algorithm in DP . . . 57
A.2 Q-Learning Off-policy TD Control . . . 57
A.3 REINFORCE: Monte-Carlo Policy-Gradient Control . . . 58

B Setting up and running the program **59**
B.1 Prepare Environment . . . 59
B.2 Datasets . . . 59
B.3 Baseline . . . 59
B.4 Examples of running Train/Test program . . . 60
B.4.1 `.env` file . . . 60
B.4.2 Run the program to print the help message . . . 60
B.4.3 Single Run (train/test) . . . 60
B.4.4 Sweep Run: 3 runs with random hyperparameters over 2 datasets and 5 algorithms (train/test) . . . 61

C Graphs **62**
C.1 Hyperparameters Tuning Results . . . 62

List of Figures

2.1 The agent interacts with the environment and learns to maximize the cumulative reward over time T. 8
2.2 The relationship between model, value function, and policy. Source [18]: . . 9
2.3 Markov Process with Start state S_0 and Terminal state S_3, because there is no edge from S_3. 12
2.4 Generalized Policy Iteration convergence. 20

3.1 The pipeline for solving the RL problem in the stock market [9] 30
3.2 Correlation matrix of technical indicators after removing correlated indicators. 36
3.3 How the agent produces the weights for the portfolio [4]. 37

4.1 Cumulative returns of the best(zfjr0ks0) and worst(p3irnh80) performing models, indexes (DJI, GSPC, IXIC, RUT), and strategies (minimum variance and maximum Sharpe ratio), during the testing period (2017-2022). 47
4.2 Drawdown analysis of the best and worst performing models, and indexes (DJI and IXIC). 48
4.3 The monthly and annual returns of the model with the highest *test/total_reward* 49
4.4 The monthly and annual returns of DJI Index 49
4.5 The monthly and annual returns of IXIC Index 50

C.1 W&B Chart of parameters and their performance 62
C.2 W&B Chart of parameters and their performance 62
C.3 W&B Chart of parameters and their performance 63

Chapter 1

Introduction

1.1 Background

The *Portfolio allocation problem* is to spread appropriate finite cash budget into financial instruments [10]. Under the financial instruments, we can imagine Stocks, Bonds, Mutual Funds, Commodities, Derivatives, Real Estate Investments Trusts (REITs), Exchange-Traded Funds (ETFs), and many more. The outcome should be to increase the initial capital over the course of a selected investing horizon, which can vary from a few days to decades. Portfolio management is essential for investors, particularly those who manage large sums of money such as institutional investors, pension funds, and wealthy individuals. While allocating assets instead of cash one must think about minimizing risk and maximizing the expected return on the investment. For that, the key considered strategy is *diversification*, which involves spreading investments across different instrument classes and markets in order to reduce the overall risk of the portfolio. A portfolio full of different assets can change over time due to market conditions, where the value of other assets may increase or decrease, which may cause the portfolio to become imbalanced. *Re-balancing* ensures that the portfolio remains aligned with the investor's goals and risk tolerance.

Among other portfolio allocation strategies could be mentioned *Modern portfolio theory (MPT)* to optimally allocate assets in a portfolio [19]. MPT uses statistical tools to determine the efficient frontier, which is the set of optimal portfolios that offer the highest expected return for a given level of risk, or the lowest risk for a given level of expected return [22]. Another approach is *Mean-variance optimization*, which uses mathematical models to determine the optimal portfolio based on an investor's risk tolerance and expected returns [15].

These approaches are not too appropriate for portfolio management, because the stock market is stochastic, volatile, quickly changing, and uncertain environment. These strategies are not flexible enough to adapt to the changing environment like the stock market, because they assume the future will be similar to the past, which may not always be accurate.

So, the most recent state-of-the-art portfolio management strategies are based on machine learning techniques. *Reinforcement learning (RL)* is a type of machine learning that is well-suited for solving problems involving decision-making and control [17]. In the context of portfolio allocation, RL can be used to optimize the allocation of assets in a portfolio in order to maximize returns or minimize risk. RL algorithms can learn from historical data and adapt to changing market conditions, which can lead to more efficient and profitable portfolio management. The benefits of RL have been used in many different fields, such as robotics, games, and finances.

In the last decade, RL has become popular, because of its ability to learn difficult tasks in a variety of domains without knowing the environment model [17]. RL has advantages, such as flexibility, adaptability, and utilization of various information like e.q. experience gained from the environment under certain conditions. The agent is trained under a certain policy in a particular environment, which is modeled using *Markov Decision Process (MDP)*. MDP is a mathematical framework for modeling sequential decision-making problems [13]. MDP can be used to model the fully observable environment, where the agent can observe the state of the environment. If the environment is not fully observable, then the agent can observe only a part of the state of the environment, which is called *partially observable Markov decision process (POMDP)* [7]. In finances, the environment is usually fully observable, because the agent can observe the state of the environment. MDP is composed of the following elements:

1. **State:** The state is the current situation of the environment.
2. **Action:** The action is the decision that the agent can take.
3. **Reward:** The reward is the feedback that the agent receives after taking an action.
4. **Transition:** The transition is the change of the state after taking an action.

In agent training we handle the following problems:

- **State space**
  The state space is a finite set of all possible configurations of the environment. In the context of portfolio allocation, the state space can be defined as the finite set of all possible instrument features (fundamental and technical analysis) and their weights in the portfolio.
- **Action space**
  Action space should be designed so that the agent weights the assets in the portfolio. Here the question is: Should be this asset in the portfolio and if yes, what is the weight of this asset in the portfolio? These decisions are crucial for the performance of the agent. It is really difficult to find the optimal policy for the portfolio allocation because the agent has to choose between multiple assets with various differences in information about the assets. Also, actions should be considered profitable and safe in the long term, which means that the agent usually has to make decisions based on long-term rewards or on the defined investment horizon.
- **Reward function**
  The reward should reflect the agent's performance in the environment. Is the current portfolio value increasing or decreasing after the agent takes actions proposed by the policy?

When the state space is too large, then is merely impossible to be explored with the limited computational resources *Deep Reinforcement Learning (DRL)* can be used. DRL is a subfield of Reinforcement Learning (RL) that combines the use of deep neural networks with RL algorithms. In traditional RL, the agent's policy and value functions are typically represented by simple, hand-designed features or a small number of parameters. In contrast, DRL uses deep neural networks to represent these functions, allowing the agent to learn from high-dimensional and complex inputs. DRL algorithms are used to train agents to perform

a wide range of tasks, such as playing video games, controlling robotic arms, and driving cars. There are several popular algorithms in DRL, such as: *Deep Q-Network (DQN)*, *Deep Deterministic Policy Gradient (DDPG)*, *Proximal Policy Optimization (PPO)*, *Soft Actor-Critic (SAC)*, and *Twin Delayed Deep Deterministic Policy Gradient (TD3)*.

1.2 Limitations

1. **Data availability:** DRL models require large amounts of historical data to train effectively, which may be difficult to obtain for certain assets or markets.
2. **Model Over-fitting:** DRL models can easily over-fit to the training data, leading to poor performance on unseen data.
3. **High computational cost:** DRL models can require significant computational resources to train agents.
4. **Risk management:** DRL models may not be able to effectively handle risk management, such as different market situations (Market sentiment, Bull and Bear markets).

1.3 Aim of the Thesis

We will evaluate the performance of portfolio allocation methods based on DRL and compare them to traditional portfolio optimization techniques (MPT, Mean-Variance). Our goal is to determine the potential of DRL for portfolio allocation and identify the limitations of DRL-based portfolio allocation methods for future research.

The thesis objectives are:

- **Experimental evaluation & Benchmarks**
  Compare existing portfolio allocation agents. Evaluate the performance of the RL agents by comparing them with the baseline portfolio management strategies, such as MPT, Mean-Variance Optimization, and indexes (DJI, Nasdaq-100, S&P500, RUSSEL2000).
- **Dataset**
  Create a suitable datasets for the portfolio allocation problem. Datasets will be focused on the company's financial data, such as fundamental and technical analysis data.
- **Reimplementation**
  Try to improve current agents (Portfolio Allocation agent from **FinRL** [4]) with new datasets, focusing on Data Engineering and different DRL algorithms.

The thesis is be implemented using the programming language Python3 and open-source libraries such as NumPy, Pandas, Stable Baselines3, and OpenAI Gym.

Chapter 2

Reinforcement Learning

The motivation for this chapter comes from the influential book on reinforcement learning by Richard Sutton and Andrew Barto and Foundations of Reinforcement Learning with Application in Finance by Ashwin Rao and Tikhon Jelvis [13, 17].

This chapter discusses the application of Reinforcement Learning techniques in the context of portfolio allocation. We provide an introduction to *Markov Theory* in section 2.2, first examining *Markov processes*, *Markov reward processes* and *Markov decision processes*, which are the basic building blocks of Reinforcement Learning. We then describe *Model Free Methods* in section 2.3 and section 2.4. We discuss *Model Free* Reinforcement Learning Methods in detail with the advantages and disadvantages of each approach. Let us start with an introduction to artificial intelligence in general and its different types of learning:

Supervised Learning In supervised learning (SL), a model is trained on a labeled dataset, where the input data is paired with corresponding output labels. The model learns to make predictions based on the labeled examples, and the goal is to minimize the error between predicted outputs and actual labels. Common applications of supervised learning include image classification, speech recognition, and sentiment analysis.

Semi-supervised Learning Semi-supervised learning (SSL) is a combination of supervised and unsupervised learning. It uses a small labeled dataset along with a large unlabeled dataset for training. The model leverages the limited labeled examples to learn patterns from the unlabeled data and then makes predictions on unseen data. SSL is useful when obtaining labeled data is expensive or time-consuming. It is often used in scenarios where obtaining a large labeled dataset is challenging, such as in medical diagnosis or fraud detection.

Unsupervised Learning In unsupervised learning (UL), the model learns from unlabeled data without any predefined output labels. The goal is to find underlying patterns, structures, or relationships within the data. Everyday unsupervised learning tasks include clustering, dimensional reduction, and anomaly detection. UL is used in scenarios where labeled data is scarce or not available, and the model needs to discover patterns autonomously from the data.

The last type is Reinforcement Learning and this entire chapter will be devoted to it, let's dive into it in more detail.

2.1 Introduction

Reinforcement learning (RL) is an exciting field at the intersection of artificial intelligence (AI) and machine learning (ML) that deals with training agents to make optimal decisions in dynamic environments. RL is inspired by the way humans learn from experience, like `trial-and-error`, and an agent interacts with an environment the same way and receives feedback in the form of *rewards* (typically positive number, e.g.: 1) or *penalties* (typically negative number, e.g.: -1), and uses this feedback to learn and improve its decision-making abilities, section 2.1.

At the heart of RL lies the concept of an agent, which takes actions in an environment to achieve specific goals. The environment is typically modeled as a Markov decision process (MDP), later defined in section 2.2.3, which is a mathematical framework that describes how an agent interacts with an environment in discrete time steps.

The goal of an RL agent is to learn a policy, denoted by π, which is a mapping from states to actions that maximize the cumulative reward G_t over time T. The agent uses this policy to select actions at each time step, and the environment responds with a new state and a reward. The agent then updates its policy based on the observed rewards and states, aiming to improve its decision-making abilities and achieve higher rewards in the long run. This is the RL advantage because, unlike supervised learning, RL does not require labeled data [14].

The sequence of states, actions, and rewards that the agent experiences is called a trajectory, and it looks like this:

$$(S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, \ldots, S_{T-1}, A_{T-1}, R_T, S_T) \tag{2.1}$$

This sequence of *state-action-reward* can be finite or infinite, depending on the environment and the agent's goal. A pretty good example of this is the game of chess, where the game ends when one of the players wins or the game is a draw. In this case, the trajectory is finite, and the agent's goal is to maximize the cumulative reward over time T. On the other hand, the self-driving car example is an infinite-horizon problem, where the agent's goal is to maximize the cumulative reward over an infinite time horizon or until the car reaches its destination [18].

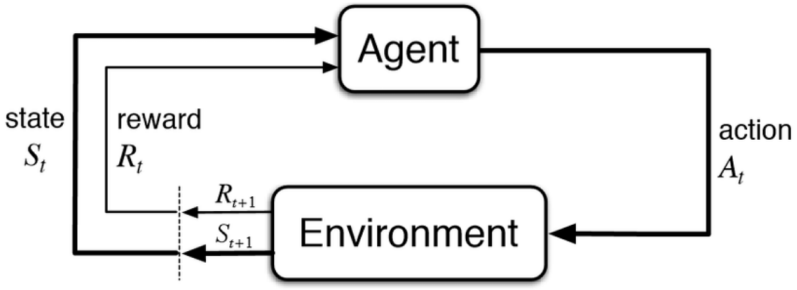


Figure 2.1: The agent interacts with the environment and learns to maximize the cumulative reward over time T.

Because RL algorithms are categorized by the way it learns, there are many different types of RL algorithms. Based on the different approaches we can categorize RL algorithms into these classes:

- **Value-based**
  - No Policy (implicit)
  - Value function
- **Policy-based**
  - Policy
  - No Value function
- **Actor-critic**
  - Policy
  - Value function
- **Model-based**
  - Policy and/or Value function
  - Model
- **Model-free**
  - Policy and/or Value function
  - No model

As we can see in fig. 2.2, the main difference is that the agent/algorithm learns the decision process based on *Value-based methods*, *Policy-based methods*, *Model-based models* or some other combination of these classes. In this section, we explain the concepts of *Value-based methods* in section 2.3 and *Policy-based methods* in section 2.4, and leave untouched *Model-based methods* as this work does not use them in the implementation.

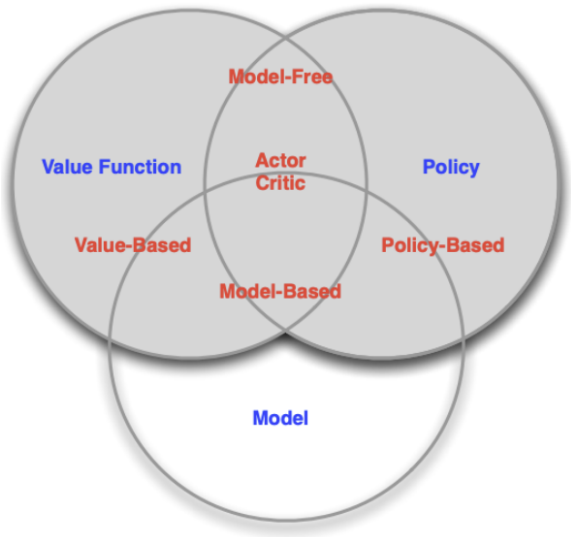


Figure 2.2: The relationship between model, value function, and policy. Source [18]:

2.2 Markov Theory

This section provides an introduction to Markov theory, which is the fundamental building block of Reinforcement Learning. We begin by *Markov Process*, described in section 2.2.1, which is a stochastic process that satisfies the Markov property. We then move on to *Markov Reward Process*, described in section 2.2.2, which is a Markov process with a reward function. Finally, we describe the most important for RL *Markov Decision Process*, described in section 2.2.3, which is a Markov Reward process with decision-making ability.

2.2.1 Markov Process

The Markov process (MP), also known as Markov Chains, describes the states of an environment and models the dynamics of state transitions. In an MP, an agent can only observe the changing states of the environment and has no influence over them. Markov process has two key properties. Firstly, state transitions are non-deterministic. States are modeled as realizations of random variables, defined in section 2.2.1. Secondly, the future state is only dependent on the current state, and not on previous states, simplifying causality with the Markov property [13, 3].

Probability Functions

The first property of an MP states that each concrete state of an environment is the realization of a discrete random variable X from set V with a certain probability $\Pr[X = x]$, where $x \in V$ and set V contains all states of an environment. A state is a realization of a random experiment that the environment assumes with a certain probability, and this can be represented as a probability function [18]:

$$P(X = X(\omega)) = P(X = x) \tag{2.2}$$

The repeated successive execution of a random experiment can be represented as a stochastic process, which is a sequence of random variables, e.g., $X_t(\omega), X_{t+1}(\omega), \ldots X_n(\omega)$, where a single term can be shortened to X_t and it represents the state of the environment at time t, $t \in \mathbb{N}$, ω is an elementary outcome of all possible outcomes Ω [3, 13].

Stochastic Process (Random Process)

A stochastic process is defined as a collection of random variables defined on a common probability space $(\Omega, \mathcal{F}, P)$, where Ω is a sample space, $\mathcal{F}$ is a σ-algebra, and P is a probability measure, and the random variables, indexed by some set T, all take values in the same mathematical space S, which must be measurable with respect to some σ-algebra Σ. In other words, a stochastic process is a collection of random variables X_t, indexed by time t, so the definition is [21]:

$$X_0, X_1, X_2, \cdots \text{ for discrete-time} \tag{2.3}$$

or

$$\{X_t\}_{t \geq 0} \text{ for continuous-time.} \tag{2.4}$$

In stochastic processes the probability that the environment assumes a certain state depends on the realized states of previous random variables. For example, if the weather forecast is assumed to be a stochastic process, then yesterday's weather may still have an influence on tomorrow's weather. To represent this causality complicates the modeling of stochastic processes so that with the definition of Markov property, in section 2.2.1, the dependence of future states is assumed only on the current state. This is the second important property of the Markov process [18].

Markov Property

The Markov property, which is defined using conditional probability, states that "The future is independent of the past, given the present." The stochastic process has the Markov

property if and only if, for all time steps $t \in I$, where I is some (totally ordered) set, the conditional probability of the next state given the current state is equal to the conditional probability of the next state given all the previous states: $\Pr[X_{t+1}|X_t] = \Pr[X_{t+1}|X_1, \ldots, X_t]$

This property has several advantages in practical reinforcement learning, including the uniqueness and distinctiveness of states, as well as the ability to precisely formulate the probability of state transitions, defined as [17, 18]:

$$\mathcal{P}(x'|x) = \Pr\left[X_{t+1} = x'|X_t = x\right] \tag{2.5}$$

Given n possible states, $s \in \mathcal{S}$, then the probability of transitioning from state s to state s' can be represented as a matrix $\mathcal{P}$, and because probability summation rule, the sum of transition probabilities from state s to any other state s' must equal to 1.

Definition 1. *The Markov process is a stochastic process that satisfies the Markov property and is described as tuple* $(\mathcal{S}, \mathcal{P})$ *for which holds:[1]*

- $\mathcal{S} = s_1, s_2, \cdots, s_n$ *is a finite set of states*
- $\mathcal{P}$ *is an* $n \times n$ *transition probability matrix which sums to 1 for each row, so each value* p_{ij} *is the probability of transitioning from state* s_i *to state* s_j *in interval* $\langle 0; 1 \rangle$

Starting States

The probability distribution of start states is denoted as $\mu : N \to [0, 1]$ in order to perform simulations and compute the probability distribution of states at specific future time steps. A Markov Process is fully specified by the transition probability function $\mathcal{P}$, which governs the complete dynamics of the process.

- Specification of the transition probability function $\mathcal{P}$.
- Specification of the probability distribution of start states (denote this as $\mu : N \in [0, 1]$).

Given μ and $\mathcal{P}$, we can generate sampling traces of the Markov Process and answer questions such as the probability distribution of states at specific future time steps or the expected time of the first occurrence of a specific state, given a certain starting probability distribution μ. The separation of concerns between $\mathcal{P}$ and μ is key to the conceptualization of Markov Processes [13].

Terminal States

Markov Processes can terminate at specific state (e.g., based on rules for winning or losing in games). Termination can occur after a variable number of time steps (episodic) or after a fixed number of time steps (as in many financial applications). If all sampling traces of the Markov Process reach a terminal state, they are called episodic sequences. The notion of episodic sequences is important in Reinforcement Learning. In some financial applications, the Markov Process terminates after a fixed number of time steps T, and states with time index $t = T$ are labeled as terminal states. States with time index $t < T$ transition to states S_{t+1} with time index $t + 1$ [13].

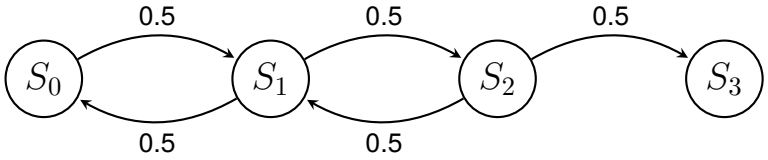


Figure 2.3: Markov Process with Start state S_0 and Terminal state S_3, because there is no edge from S_3.

The examples shown here only include states and transition probabilities. To fully define an environment within the framework of RL, actions, and rewards also need to be defined, so in the next section, we will introduce Markov Reward Process (MRP) and Markov Decision Process (MDP) to be able to define RL environments [17].

2.2.2 Markov Reward Process

Markov Reward Process (MRP) is a Markov Process with rewards. These rewards are random, and all we need to do is to specify the probability distributions of these rewards as we make state transitions. The main purpose of Markov Reward Processes is to calculate how much reward we would accumulate (in expectation, from each of the non-terminal states) if we let the process run indefinitely, bearing in mind that future rewards need to be discounted appropriately γ (otherwise, the sum of rewards could blow up to ∞). In order to solve the problem of calculating expected accumulative rewards, defined in section 2.2.2, from each non-terminal state, we will first set up some formalism for Markov Reward Processes and develop some theory on calculating rewards accumulation [13].

The main objective of an RL agent is to maximize the sum of rewards from each time step. The agent can observe different episodes in the Markov process but lacks the means to determine the actual quality of an episode. By calculating the reward, we can precisely measure the goodness of an episode or even a single state using the *state-value function*, defined in section 2.2.2. This allows the agent to actively transition to favorable states and maximize the reward [13].

Definition 2. ***Markov Reward Process*** *is a Markov Process, along with a time-indexed sequence of Reward random variables $R_t \in D$ (a countable subset of $\mathbb{R}$) for time steps $t = 1, 2, \ldots$, satisfying the Markov Property (including Rewards): $\mathcal{P}(R_{t+1}, S_{t+1})|S_t, S_{t-1}, \ldots, S_0] = P[(R_{t+1}, S_{t+1})|S_t]$ for all $t \geq 0$. MRP is a tuple $(\mathcal{S}, \mathcal{P}, \mathcal{R}, \gamma)$ for which holds:[13]*

- $\mathcal{S} = s_1, s_2, \cdots, s_n$ *is a finite set of states*
- $\mathcal{P}$ *is an $n \times n$ transition probability matrix which sums to 1 for each row, so each value p_{ij} is the probability of transitioning from state s_i to state s_j in interval $\langle 0; 1 \rangle$*
- $\mathcal{R}$ *is a sequence of random variables $R_1, R_2, \cdots, R_n$ where R_t is a random variable that represents the reward for transitioning from state s_t to state s_{t+1}*
- γ *is a discount factor in interval $\langle 0; 1 \rangle$*

$$\mathcal{P}(s, r, s') = \Pr\left[R_{t+1} = r, S_{t+1} = s' | S_t = s\right] \text{ for time steps } t = 0, 1, 2, \ldots \tag{2.6}$$

$$\text{for all } s \in N, r \in D, s' \in S, \text{ such that } \sum_{s' \in S} \sum_{r \in D} \mathcal{P}(s, r, s') = 1 \text{ for all } s \in N$$

Reward function

The reward function $\mathcal{R}(s)$ is a function that maps a state s to a reward r and specifies how much reward and agent expects from the environment given current state s. If an agent is in a state s at time t, the agent receives reward R_{t+1} at time $t+1$, when it transitions to a subsequent state s'. Rewards of an episode can be represented as a sequence $(R_1, R_2, \ldots, R_t)$ [18].

When to receive the reward? An Reinforcement Learning Agent, which we will introduce later, should receive a reward for a good action and a penalty for a bad action. Good actions are those that lead to the agent's main goal, while bad actions are those that lead to a state that is not desirable for the agent. That is, an agent should not receive (one small) reward when it can then receive a large penalty, for example in chess by taking one piece, when it can then lose the game by getting checkmate [17].

Expected reward

The expected reward, denoted as G_t at time t is the discounted sum of rewards in a single episode, is defined as:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{2.7}$$

We can also calculate the expected rewards for state-action pairs as a two-argument function $r : S \times A \to R$, is defined as:

$$r(s, a) = \sum_{t=0}^{\infty} \gamma^t R_{t+1} P(S_{t+1} = s' | S_t = s, A_t = a) \tag{2.8}$$

Discount factor γ The calculation also involves the discount factor γ which is a value in the interval $\langle 0, 1 \rangle$. If γ is equal to one, then the expected reward can goes to infinity, that is why the agent can only calculate the reward in the case of always terminating episodes. If γ is less than one, the reward has a finite value, allowing the agent to determine the quality of an episode. The discount factor is not only useful mathematically but also for tuning the agent's rewards. If early rewards in an episode are more significant than later ones, γ should be close to zero. If the rewards represent monetary gains, then it is the case, as early rewards earn additional interest. On the other hand, the closer γ is to one, the more important later rewards are [13, 17, 12].

State-value Function

The state-value function provides information about the long-term expected reward for each state in an environment. With this information, an agent can determine which state to transition to in order to maximize the reward of an episode. Specifically, the agent should choose the state that has the highest long-term expected reward. If the agent observes a sequence of states and rewards, it can remember the subsequent rewards for each state, calculate the reward of an episode, and iteratively update the probabilities for a higher occurrence of rewards over multiple episodes. As the number of episodes approaches infinity, the estimated probabilities converge to the true probabilities, and the long-term expected

reward of a state can be accurately determined. Intuitively, the more episodes and consequent rewards an agent observes, the better it can estimate the value of each state. We will introduce the methods for determining the state-value function and the action-value function in the next subsection, along with a recursive iterative approach for calculating the state-value function based on Bellman's equation [18].

Since episodes may start with different states due to state transition probabilities, the expected reward of a particular state is the expected value of the conditional density function over the probabilities of rewards for that state. Thus, G_t can be treated mathematically as a continuous random variable. To derive *Bellman's equation*, defined later in this subsection, we need to make use of the definition of the *(Recursive) State-value function* [18, 13].

$$\begin{aligned} v(s) &= E[G_t|S_t = s] \\ &= E[R_{t+1} + \gamma G_{t+1}|S_t = s] \\ &= E[R_{t+1} + \gamma v(S_{t+1})|S_t = s] \\ &= \mathcal{R}(s) + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}(s'|s) v(s') \end{aligned} \tag{2.9}$$

where γ is the discount factor, $\mathcal{R}(s)$ is the immediate reward for state s, $\mathcal{P}(s'|s)$ is the transition probability from state s to state s', and $\mathcal{S}$ is the set of all possible states in the environment. The last equation expresses that the long-term expected reward of a state depends only on the immediate reward and the long-term expected reward of the subsequent states.

In the case of Finite Markov Reward Processes, let's assume that the state space is denoted as $\mathcal{S} = \{s_1, s_2, \ldots, s_n\}$, and the subset of states of interest is denoted as $\mathcal{N}$ with $m \leq n$ states. We can use bold-face notation to represent functions as column vectors and matrices since we are dealing with finite states/transitions. So, V is a column vector of length m, P is an $m \times m$ matrix, and R is a column vector of length m (with rows/columns corresponding to states in $\mathcal{N}$). We can express the equation in vector and matrix notation as follows:

$$\begin{aligned} \boldsymbol{V} &= \boldsymbol{\mathcal{R}} + \gamma \boldsymbol{\mathcal{P}} \cdot \boldsymbol{V} \\ &= (\boldsymbol{I_m} - \gamma \boldsymbol{\mathcal{P}})^{-1} \cdot \boldsymbol{\mathcal{R}} \end{aligned} \tag{2.10}$$

where $\boldsymbol{I_m}$ is the identity matrix of size $m \times m$, and γ is the discount factor [13].

By extending the section 2.2.2 to include actions, we arrive at the most important equation in all of the reinforcement learning: *The Bellman equation*, defined in section 2.2.3. It states that to calculate the long-term expected reward from a state, an agent only needs to add together the reward of the current state and the long-term expected reward of the next state [18].

2.2.3 Markov Decision Process

The Markov Decision Process (MDP) allows an agent to actively influence changes in the state of the environment through its actions. Within the MDP, the agent has the ability to jointly determine the subsequent state to which the environment should transition. The agent's primary goal is to strategically choose actions that maximize the expected payoff. In the previous subsections, we addressed the aspect of *sequential uncertainty (e.g., MRP)* using the Markov process framework and extended it to include the uncertain reward $R_t \in \mathcal{R}$ at each state transition $p(s', r, s)$, referred to as Markov reward processes. However, this

framework lacks the notion of *sequential decision making*, and in this section, this notion is introduced in terms of MDP, a generalization of MRP that includes the notion of *sequential decision making* [13].

Definition 3. Markov decision process *is the Markov reward process with actions. It is an tuple* $(S, A, \mathcal{P}, \mathcal{R}, \gamma)$*, where:*

- $\mathcal{S}$ *is a finite set of states, known as the State Space*
- $\mathcal{A}$ *is a finite set of actions, known as the Action Space*
- $\mathcal{P}$ *is a transition probability function* $p(s', r, s, a)$*, which is a function that maps a state* s*, an action* a*, a next state* s' *and a reward* r *to a probability* $p(s', r, s, a)$
- $\mathcal{R}$ *is a reward function* $r(s, a)$*, which is a function that maps a state* s *and an action* a *to a reward* $r(s, a)$
- $\gamma \in [0, 1]$ *is the discount factor*

Stochastic Policy function

A Stochastic Policy function, denoted by π, in the context of MDPs, is a function that maps states to actions. It represents the agent's decision-making strategy for selecting actions based on the current state.

Definition 4. *A policy is defined as* $\pi : \mathcal{S} \to \mathcal{A}$*, where* $\mathcal{S}$ *is the set of states and* $\mathcal{A}$ *is the set of actions. Notation for a policy is as follows:*

$$\pi(a|s) = \mathbb{P}(A_t = a|S_t = s) \tag{2.11}$$

Policy refers to the specification of an Agent's actions based on the current state in a Markov Decision Process. The policy can be deterministic, meaning it selects a single action for each state. It is represented as a function $\pi_D : N \to \mathcal{A}$, where $\pi_D(s)$ represents the action to be taken in state s. Or it can be stochastic, meaning it selects actions probabilistically based on some probability distribution over actions for each state. A policy is a function that maps states to actions and represents the agent's decision-making strategy. Mathematically, a Policy is represented as a function $\pi : N \times A \to [0, 1]$, where N represents the state space and A represents the action space. The function $\pi(s, a)$ represents the probability of taking action a in state s at time step $t = 0, 1, 2, \ldots$, for all $s \in N$ and $a \in A$. It is assumed that the sum of probabilities for all actions in a given state is equal to 1. A Policy is usually assumed to be Markovian, meaning that the action probabilities depend only on the current state and not the history. It is also assumed to be stationary, meaning that the action probabilities do not change over time. However, if the policy needs to depend on the time step t, we can include t as part of the state, which would make the policy stationary but may increase computational cost due to the enlarged state space. In the more general case, where states or rewards are uncountable, the same concepts apply except that the mathematical formalism needs to be more detailed and more careful. Specifically, we'd end up with integrals instead of summations, and probability density functions (for continuous probability distributions) instead of probability mass functions (for discrete probability distributions). For ease of notation and more importantly, for ease of understanding of the core concepts (without being distracted by heavy mathematical formalism), we've chosen to stay with discrete-time, countable $\mathcal{S}$, countable $\mathcal{A}$ [13].

Policies and action-value functions are closely related and are used interchangeably in many reinforcement learning algorithms, but they are conceptually distinct.

State-value Function for Stochastic Policy π

The value function $V_\pi(s)$ for a stochastic policy π is the expected cumulative discounted reward the agent can obtain from state s by following policy π and then continuing to follow π thereafter. It can be computed recursively using the Bellman equation, which relates the Value Function of a state.to the rewards and transitions of the MDP. It is defined as:

$$\begin{aligned} v_\pi(s) &\doteq {}_\pi\left[g_t | S_t = s\right] \\ &= \left[r_{t+1} = \gamma g_{t+1} | S_t = s\right] \\ &= {}_\pi\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | S_t = s\right] \\ &= \sum_a \pi(a|s) \sum_{s',r} p(r, s'|s, a) \left[r + \gamma v_\pi(s')\right], \text{ for all } s \in \mathcal{S} \end{aligned} \tag{2.12}$$

where $\mathbb{E}_\pi$ denotes the expectation with respect to the states and rewards generated by following policy π [17].

Action-value Function

Action-Value Function $q_\pi(s, a)$, which maps a (state, action) pair to the expected reward originating from that pair when following the policy π. The Action-Value Function is denoted by q_π and is crucial in developing various Dynamic Programming and Reinforcement Learning algorithms for the MDP Prediction problem [17].

Definition 5. *The action-value function $Q_\pi(s, a)$ for a policy π is the expected cumulative discounted reward the agent can obtain from state s, taking action a, and then following policy π thereafter. It is defined as:*

$$q_\pi(s, a) = \mathbb{E}_\pi\left[\sum_{t=0}^{\infty} \gamma^t R_{t+1} | S_0 = s, A_0 = a\right] \tag{2.13}$$

The table of action-value functions is also called Q-Table. A policy and the action-value functions of an agent can both be represented by $|\mathcal{S}| \times |\mathcal{A}|$ matrix. Implementations of RL algorithms then mostly use only the matrix of action-value functions and infer the policy from them. The goal of an agent is to maximize the return. In the above context, this means to find a policy that yields as much return as possible, therefore in section 2.2.3 we define the *Optimal Value Funtions* [17].

To avoid confusion, v_π is referred to as the State-Value Function, while q_π is referred to as the Action-Value Function. The Action-Value Function provides information about the expected returns from specific state-action pairs and is useful for making decisions about which actions to take in an MDP. The relationship between the state-value function and the action-value function can be defined as:

$$v_\pi(s) = \max_{a \in \mathcal{A}} q_\pi(s, a) \tag{2.14}$$

Optimal State-value and Action-value Functions

The goal of an agent is to maximize the reward, which means finding a policy that yields the highest possible return. To compare two policies, let π and π' be two different stochastic

policies. Then π' is considered better or equal to π if $v_{\pi'}(s) \geq v_\pi(s)$ holds for all states s. It can be shown that there is always at least one policy that is better or equal to all other policies, and this policy is called the optimal policy π_*. If an agent uses the optimal policy, then for all states and actions the agent will use the optimal state-value function:

Definition 6. *The Optimal State-value function $v_*(s)$ is the maximum value function over all possible policies and is defined as:*

$$v_*(s) = \max_\pi v_\pi(s) \tag{2.15}$$

and the optimal action-value function:

Definition 7. *The Optimal Action-value function $q_*(s, a)$ is the maximum action-value function over all possible policies. It is defined as:*

$$q_*(s, a) = \max_\pi q_\pi(s, a) \tag{2.16}$$

If the optimal action-value q_* function is found, the optimal policy π_* can be derived. The optimal policy function $\pi_*(a|s)$ always selects an action a with a selection probability of 1 for which $q_*(s, a)$ is maximal for a given state s. It also follows that the optimal policy is deterministic, meaning for the same state, the optimal policy function always selects exactly the same action. The transition from stochastic to deterministic policy is explained by the *greedy selection*[1] of actions in the last equation [2, 13]:

$$\pi_*(a|s) = \begin{cases} 1, & \text{if } a = \arg\max_q^*(s, a) \\ 0, & \text{else} \end{cases} \tag{2.17}$$

Bellman's Optimality Equation

The optimal state-value function $v_*(s)$ must satisfy Bellman's equation, which provides a recursive rule for determining the long-term expected return of each state. In order to incorporate actions, the State-value function for a Markov Reward Process (MRP) is extended to include actions as the second parameter, leading to Bellman's optimality equation for $v_*(s)$ [17]:

$$\begin{aligned} v_*(s) &\doteq \max_{a \in \mathcal{A}} q_{\pi_*}(s, a) \\ &= \max_{a \in \mathcal{A}} \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) | S_t = s, A_t = a] \\ &= \max_{a \in \mathcal{A}} \sum_{s', r} p(r, s'|s, a) \left[r + \gamma v_*(s') \right] \end{aligned} \tag{2.18}$$

This equation derives the recursive relation to the subsequent state, allowing the reference to the optimal policy to be omitted. Similarly, the Action-value function $q_*(s, a)$ can be derived as [17]:

$$\begin{aligned} q_*(s, a) &= \mathbb{E}[R_{t+1} + \gamma \max_{a' \in \mathcal{A}} q_*(S_{t+1}, a') | S_t = s, A_t = a] \\ &= \sum_{s', r} p(r, s'|s, a) \left[r + \gamma \max_{a' \in \mathcal{A}} q_*(s', a') \right] \end{aligned} \tag{2.19}$$

[1] Always choose action with the highest expected reward.

The Bellman Optimality equation can be used to recursively calculate value functions by updating the value of each state based on the immediate reward and the long-term expected reward of the next state. With this solid understanding of Markov Decision Processes (MDP), which form the foundation for Reinforcement Learning (RL), we can delve into the different learning methods used in RL [17].

2.3 Value-based learning

This section presents value-based algorithms that iteratively compute state-value or action-value functions of states and actions. These functions are then used to derive continuously improving policies. Therefore, these algorithms are referred to as value-based algorithms. In section 2.3.1 we address Planning algorithm, which requires prior knowledge of transition probabilities and rewards. In section 2.3.2 we address Monte Carlo methods, that do not require prior knowledge of transition probabilities and rewards. In section 2.3.3 we address Temporal Difference methods, finally in section 2.3.4 we address Function Approximation methods, that deals with the large state/action spaces.

2.3.1 Dynamic Programming

The term dynamic programming (DP) refers to a collection of algorithms that can be used to compute optimal policies in Markov decision processes (MDPs) when a perfect model of the environment is available. Classical DP algorithms are limited in reinforcement learning due to their assumptions of a perfect model and computational expense. DP uses value functions to organize the search for good policies, and the Bellman optimality equations define optimal value functions (*State-value function* and *Action-value function*). DP algorithms are effective for problems with finite states, action, and reward sets. However, for continuous problems, exact solutions are limited. One approach is to approximate solutions by discretizing the state and action spaces and applying finite-state DP methods. The second approach is to use function approximation to approximate the value functions. DP planning algorithms consist of the evaluation of a given policy, called policy prediction, defined in section 2.3.1 and the subsequent improvement of the policy, called policy improvement, defined in section 2.3.1. The goal of these two steps is to infer $v_*(s)$ or $q_*(s, a)$ for all the states and actions in a procedure called Generalized Policy Iteration, defined in section 2.3.1 [17, 18].

Policy Iteration

Policy iteration consists of 2 steps: *policy evaluation* and *policy improvement*. The policy prediction step, defined in section 2.3.1, is used to estimate the value function v_π for a given policy π. The policy improvement step, defined in section 2.3.1 is used to improve the policy π by selecting the action that maximizes the value function v_π [17].

Policy Prediction Policy evaluation, or prediction, is the process of computing the state-value function for an arbitrary policy. The state-value function denoted as $v_\pi(s)$, represents the expected sum of discounted rewards from a given state s when following policy π. If the dynamics of the environment are completely known, the policy evaluation can be formulated as a system of simultaneous linear equations, which can be solved iteratively using update rules based on the Bellman equation for v_π. This process involves finding

successive approximations of the value function, starting from an initial approximation, and updating it according to the Bellman equation until convergence is achieved to $v_k(s) \approx v_\pi(s)$ with:

$$v_{k+1}(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \left(R(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a) v_k(s') \right) \tag{2.20}$$

for all states, this equation is analogous to Bellman's optimality equation for $v_*(s)$ in iterative form. Once $v_\pi(s)$ is computed for a given policy, the action-value function can be obtained [17, 18]:

$$q_\pi(s,a) = \mathbb{E}_\pi[G_t | S_t = s, A_t = a] = R(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a) v_\pi(s') \tag{2.21}$$

Policy Improvement This equation represents Bellman's equation for $q_\pi(s,a)$ and can be utilized in the Policy Improvement step to generate a new policy. It is important to verify that the new deterministic policy is better than or equal to the old policy, given that this condition holds for all states:

$$q_\pi(s, \pi'(s)) \geq v_\pi(s) \tag{2.22}$$

This means that choosing action $a' = \pi'(s)$ in state s under policy π produces a better result than action $a = \pi(s)$.Afterward the action-value function $q_{\pi'}(s,a)$ is computed for the new policy π' in the Policy Evaluation step and improved with $\pi'(s) = \arg\max_a q_{\pi'}(s,a)$ [18, 17].

Value Iteration

Value iteration effectively combines, in each of its sweeps, one sweep of policy evaluation and one sweep of policy improvement. Faster convergence is often achieved by interposing multiple policy evaluation sweeps between each policy improvement sweep. In general, the entire class of truncated policy iteration algorithms can be thought of as sequences of sweeps, some of which use policy evaluation updates and some of which use value iteration updates. Because the max operation in the update equation definition 8 is the only difference between these updates, this just means that the max operation is added to some sweeps of policy evaluation. All of these algorithms converge to an optimal policy for discounted finite MDPs. The algorithm pseudocode is presented in appendix A.1 [17].

Definition 8. ***Value Iteration Equation*** *is that iteratively computes the state-value function $V_k(s)$ for a given policy π using the Bellman equation:*

$$\begin{aligned} v_{k+1}(s) &\doteq \max_a \mathbb{E}\left[R_{t+1} + \gamma v_k(S_{t+1}) | S_t = s, A_t = a\right] \\ &= \max_a \sum_{s',r} p(s', r|s, a)[r + v_k(s')] \end{aligned} \tag{2.23}$$

Generalized Policy Iteration

If Policy Evaluation and Policy Improvement steps are executed enough times and no more improvement of the old policy over the new policy is found, then optimal q_* and π_* are found. Well-known algorithms in DP which implement this Generalized Policy Iteration are the Policy Iteration algorithm and its special case the Value Iteration algorithm, defined above in section 2.3.1. They use the equations Bellman's equations for $v_*(s)$ and $q_*(s,a)$

and a variant of the Generalized Policy Iteration method, where $v_\pi(s)$ does not have to be determined. The iteration in the Policy Evaluation step is immediately followed by the Policy Improvement step [17, 8].

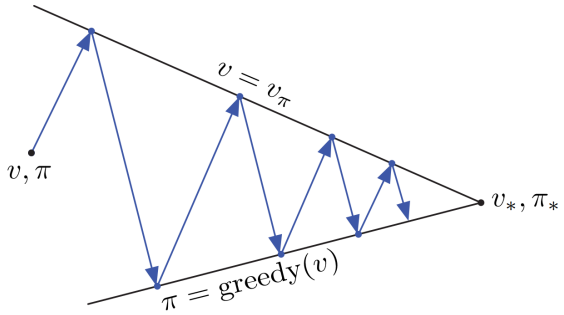
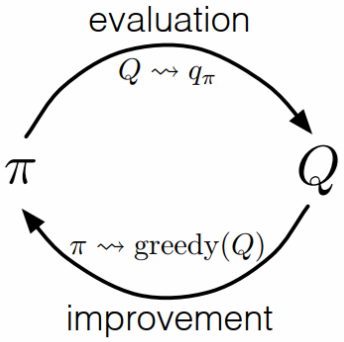


Figure 2.4: Generalized Policy Iteration convergence.

2.3.2 Monte Carlo Methods

In this chapter, we explore Monte Carlo methods for estimating value functions in the context of reinforcement learning (RL) without assuming complete knowledge of the environment (*transition probabilities* and *reward function*). Monte Carlo methods use sample sequences of states, actions, and rewards from actual or simulated interactions with the environment for learning. These methods can be used to solve the RL problem for episodic tasks, where experience is divided into episodes and value estimates and policies are updated only on the completion of an episode.



MC methods allow an agent to infer the optimal policy π_* from the optimal action-value function $q_*(s, a)$ estimated through experience. MC methods use the iterative procedure Generalized Policy Iteration to incrementally infer π_*. First, a finite trajectory $(S_0, A_0, R_1, S_1, A_1, \cdots, R_n, S_n)$ is generated, where actions are selected according to a stochastic policy $\pi(a|s)$ and states and rewards come from the unknown environmental dynamics. From this episode, the return G_t is computed for all state-action pairs reached. Then, the policy evaluation step is performed using the update equation [17, 18, 13]:

$$q_{k+1}(s, a) = q_k(s, a) + \alpha(G_t - q_k(s, a)) \tag{2.24}$$

where $\alpha \in (0, 1)$ is a learning rate and G_t is the sum of discounted rewards starting from action a in state s. The term $G_t - q_k(s, a)$ corrects the value of $q_{k+1}(s, a)$ in the direction of the target G_t. The index k represents the current episode, and in the limiting case, $q_k(s, a) = q_\pi(s, a)$, where π is the optimal policy [17].

$$\pi_0 \xrightarrow{E} q_{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} q_{\pi_1} \xrightarrow{I} \pi_2 \xrightarrow{E} q_{\pi_2} \xrightarrow{I} \cdot \pi_* \xrightarrow{E} q_{\pi_*} \tag{2.25}$$

Since MC methods do not require knowledge of the environment dynamics, a non-deterministic policy should be used for episode generation to ensure the exploration of all

states and actions. A stochastic policy, such as the ϵ-greedy approach, is commonly used in MC methods for exploration [18].

Epsilon-Greedy Policy Exploration The **ϵ-greedy** approach for exploration selects actions randomly according to:

$$\pi'(a|S) = \begin{cases} 1 - \epsilon + \frac{\epsilon}{|\mathcal{A}|} & \text{if } a = A^* \\ \frac{\epsilon}{|\mathcal{A}|} & \text{if } a \neq A^* \end{cases} \tag{2.26}$$

where $A^* = \arg\max_a(S_t, a)$. A Greedy action is an action with the highest estimated action value according to the current policy. The process of modifying action values in the policy evaluation step and setting a new policy in the policy improvement step is referred to as training the RL agent. It follows from eq. (2.24) that MC methods use the G_t return of an episode, which can only be computed when the episode is completed. Therefore, MC methods can only be used in environments with finite, i.e. always ending, episodes. This strategy is called *bootstrapping*, defined in section 2.3.3, when the methods do not need to wait for the end of the episode, but only need *n-steps* to improve the policy, the whole strategy is called *n-steps bootstrapping* and next method *Temporal-Difference* uses this strategy to improve convergence to optimal value-function citeFITMT25127, sutton2018reinforcement, rl-course-david-silver.

2.3.3 Temporal-Difference Methods

Temporal-difference (TD) learning is a fundamental concept in reinforcement learning that combines ideas from Monte Carlo and dynamic programming (DP). TD methods allow for learning from raw experience without a model of the environment's dynamics, similar to Monte Carlo methods, but also update estimates based on other learned estimates, without waiting for a final outcome (like DP methods) using *n-step bootstrapping*, defined in section 2.3.3. The relationship between TD, DP, and Monte Carlo is a recurring theme in reinforcement learning theory, and these ideas can be blended and combined in various ways. Both TD and Monte Carlo methods are used for policy evaluation, estimating the value function for a given policy. However, TD methods differ from Monte Carlo methods in that they update their value functions $v_{t+1}(s)$ or $q_{t+1}(a, s)$ estimates based on the non-terminal states observed in the experience, and they do not have to wait until the end of the episode to determine the increment to the value function [14, 17].

n-step Bootstrapping

The n-step bootstrapping is a strategy to improve the convergence of the value function. The idea is to look only *n-steps* ahead to update value function, instead of using the return of whole the episode. We show the difference in TD and MC methods.

The *TD target* is the sum of the reward and the discounted value of the next state, and it serves as the target for the TD update. The 1-step TD target G_t is:

$$G_t = R_{t+1} + \gamma V(S_{t+1}) \tag{2.27}$$

or for 2-step TD target G_t:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 V(S_{t+2}) \tag{2.28}$$

and for n-step TD target G_t:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n V(S_{t+n}) \tag{2.29}$$

while Monte Carlo Target G_t:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots + \gamma^{T-t-1} R_T \tag{2.30}$$

The learning rule for (simplest) TD(0), also known as 1-step TD is the following update equation:

$$V(S_t) = V(S_t) + \alpha \cdot (R_{t+1} + \gamma \cdot V(S_{t+1}) - V(S_t)) \tag{2.31}$$

where $V(S_t)$ is the estimated value of state S_t, α is the learning rate, R_{t+1} is the reward received at time step $t+1$, γ is the discount factor, and $V(S_{t+1})$ is the current estimate of the value of the next state. On the other for the *n-step bootstrapping* the learning rule for TD(n), also known as n-step TD is the following update equation:

$$V_{t:t+n} = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n V(S_{t+n}) \text{ , for } 0 \leq t \leq T \tag{2.32}$$

The *TD Error* is the difference between the TD target and the current value function estimate:

$$R_{t+1} + \gamma \cdot V(S_{t+1}) - V(S_t) \tag{2.33}$$

and represents the error in the current estimate of the value function, and the TD target represents the updated estimate of the value function for the current state. By using the TD error and TD target in the update rule, TD methods can learn from raw experience and update the value function at each time step, making them well-suited for online learning tasks [17].

In addition to TD(0), there are also other variations of TD methods such as TD(λ), which is a family of methods that use eligibility traces to combine TD(0) updates with Monte Carlo-like updates. TD(λ) methods have a parameter λ that controls the trade-off between the bias and variance of the updates. When λ is set to 0, TD(λ) reduces to TD(0), and when λ is set to 1, TD(λ) reduces to Monte Carlo updates. TD(λ) methods update the value function for not only the current state but also for all the states visited in the episode, weighted by their eligibility traces. The eligibility traces keeps track of the recent history of state visits and decay over time with a decay factor λ, which determines the credit assignment to different states [17].

TD methods do not have to wait until the end of the episode to update the value function. Instead, they can make updates at each time step, using the observed reward and the estimate of the value function for the next state. This makes TD methods more computationally efficient and allows for online learning in environments where episodes are long or never-ending. Like DP and Monte Carlo methods, TD methods also use the iterative Generalized Policy Iteration (GPI) procedure to determine the optimal policy. The policy evaluation step in TD methods differs from Monte Carlo methods in that it uses the TD error, which is the difference between the TD target and the current estimate of the value function, as the update rule [17].

On-Policy Learning (SARSA)

The TD methods use the action-value function for the evaluation step and iteratively determine $q_\pi(s, a)$. A TD method that implements this practice is called SARSA and adjusts

the action-value function in the evaluation step for a state-action pair at each time-step of the episode as [17, 14]:

$$q_{t+1}(s,a) = q_t(s,a) + \alpha \left(R_{t+1} + \gamma q_t(s',a') - q_t(s,a)\right) \tag{2.34}$$

The TD methods, such as SARSA, utilize the action-value function for evaluating and updating the action-value estimates. SARSA adjusts the action-value function at each time step of the episode based on the sequence of transitions (s, a, r, s', a') generated by the environment dynamics and the current epsilon-greedy policy. The updated action-value function is used to determine the new policy at the next time step. This process is repeated iteratively until optimal action value and policy functions are found. SARSA is an on-policy algorithm, as it evaluates and improves the same policy used for selecting actions [14]:

$$q_{t+1}(s,a) = q_t(s,a) + \alpha \left(R_{t+1} + \gamma \max_{a^*} q_t(s',a^*) - q_t(s,a)\right) \tag{2.35}$$

Off-Policy Learning (Q Learning)

On the other hand, off-policy algorithms, like Q-learning, evaluate and improve one policy while using a different policy for selecting actions. This can be useful, for example, when introducing a new version of an agent with a different policy to learn from an old agent with a well-performing policy. In Q-learning, the action maximizing action (a^*) of all action-value functions at a fixed state s' is always chosen when updating the action-value function for a given state-action pair (s, a), regardless of which action a^* of the policy was chosen. After updating the action-value function, the policy improvement step is performed using the epsilon-greedy approach with respect to the current policy (π) and optionally the old policy (μ) as well. An off-policy TD control algorithm known as Q-learning, defined by [18, 17]:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_{a} Q(S_{t+1}, a) - Q(S_t, A_t)\right] \tag{2.36}$$

In this case, the learned action-value function, Q, directly approximates q_*, the optimal action-value function, independent of the policy being followed. This dramatically simplifies the analysis of the algorithm and enables early convergence proofs. The policy still has an effect in that it determines which state-action pairs are visited and updated. However, all that is required for correct convergence is that all pairs (transition matrix) continue to be updated. This is a minimal requirement in the sense that any method guaranteed to find optimal behavior in the general case must require it. The Q-learning algorithm is show in algorithm 1 [14, 17].

2.3.4 Function Approximation

The Monte Carlo and Temporal Difference algorithms discussed earlier are known as tabular methods because they rely on a matrix of action returns and values of dimension $|\mathcal{S}|\mathcal{A}|$. However, when the number of states or actions is extremely large, as in the case of *Portfolio Allocation*, where the state space is the entire stock market, which may include data from thousands of stocks, the economic situation of the entire world, and the current political situation, all of these things need to be taken into account, so the state space is huge. The agent managing which assets it allocates funds to uses these observations, which very often change every single second, so this matrix becomes excessively huge. This poses a problem in terms of memory requirements, and it becomes almost impossible for an agent using MC or TD methods to comprehensively evaluate all the states and derive a policy [17, 18].

Approximate Methods, which utilize parametrized function approximators such as decision trees, regression methods, or neural networks, differ from Tabular Methods in that they do not rely on matrices to estimate action-value returns. These methods have emerged as a solution approach to the limitations of Tabular Methods, as they are more memory efficient and capable of generalization. When neural networks are used as function approximators, then these algorithms are classified as Deep Reinforcement Learning (DRL) [18].

The mean square error is defined as:

$$\overline{VE}(\boldsymbol{w}) \doteq \sum_{s \in \mathcal{S}} \mu(s) \left[v_\pi(s) - \hat{v}(s, \boldsymbol{w}) \right] \tag{2.37}$$

where $\mu(s)$ is the probability of visiting state s, and $\hat{v}(s, \boldsymbol{w})$ is the output of the function approximator for state s and weight vector $\boldsymbol{w}$. The objective is to minimize the error between the target output and the output of the function approximator, this is done by minimizing the $\overline{VE}(s)$.

Stochastic Gradient Descent

Stochastic gradient descent (SGD) is a widely used method for function approximation in value prediction. It is well-suited for online reinforcement learning. In SGD, the weight vector is updated at each time step based on a small step in the direction of the negative gradient of the squared error for a single example. The step size is controlled by a positive parameter called the learning rate. The target output for each example may be a noisy approximation or a bootstrapping target [17].

The Approximate Solution Methods use a *function approximator* to approximate the action-value function $q_\pi(s, a)$ or the state-value function $v_\pi(s)$ as: $\hat{v}(s, \boldsymbol{w}) \approx v_\pi(s)$ or $\hat{q}(s, a, \boldsymbol{w}) \approx q_\pi(s, a)$, where $\boldsymbol{w} \in \mathbb{R}^d, d \in \mathbb{N}$

The Gradient-descent update for state-value prediction is:

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - \alpha \left[v_\pi(S_t) - \hat{v}(S_t, \boldsymbol{w}) \right] \nabla \hat{v}(S_t, \boldsymbol{w}) \tag{2.38}$$

where α is the learning rate, and $\nabla \hat{v}(S_t, \boldsymbol{w})$ is the gradient of the function approximator with respect to the weight vector $\boldsymbol{w}$.

The eq. (2.38) can be extended to n-step returns:

$$\boldsymbol{w}_{t+n} \doteq \boldsymbol{w}_{t+n-1} + \alpha \left[G_{t:t+n} - \hat{v}(S_t, \boldsymbol{w}_{t+n-1}) \right] \nabla \hat{v}(S_t, \boldsymbol{w}_{t+n-1}) \tag{2.39}$$

where the n-step return is:

$$G_{t:t+n} \doteq R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n \hat{v}(S_{t+n}, \boldsymbol{w}_{t+n-1}) \tag{2.40}$$

On the other hand, the Gradient-descent update for action-value prediction is:

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t + \alpha \left[U_t - \hat{q}(S_t, A_t, \boldsymbol{w}_t) \right] \nabla \hat{q}(S_t, A_t, \boldsymbol{w}_t) \tag{2.41}$$

where the U_t is the TD target, as mentioned earlier, and the $\nabla \hat{q}(S_t, A_t, \boldsymbol{w}_t)$ is the gradient of the function approximator with respect to the weight vector $\boldsymbol{w}$ [17].

The weights for one-step SARSA are updated using the following rule:

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t + \alpha \left[R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1}, \boldsymbol{w}) - q_\pi(S_t, A_t, \boldsymbol{w}) \right] \nabla q_\pi(S_t, A_t, \boldsymbol{w}) \tag{2.42}$$

and can be extended to *n-step SARSA*:

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t + \alpha \left[G_{t:t+n} - \hat{q}_\pi(S_t, A_t, \boldsymbol{w}) \right] \nabla \hat{q}_\pi(S_t, A_t, \boldsymbol{w}) \tag{2.43}$$

The update is performed in a way that each vector's component responsible for a part of the error is updated accordingly, with a higher share of the error being adjusted if the slope of the error with respect to that component is large. The weights are adjusted until the difference between two subsequent updates is less than a defined ε value (typically a small number), at which point q_π is considered found and the optimal policy can be derived. In practice, early stopping is often used in deep learning instead of ε. Now that the vectors of state-action values are no longer represented as a look-up table, they can be written as an approximation by the neural network. Furthermore, the policy improvement step can also incorporate the ϵ-greedy approach for action selection [18, 13, 13].

$$\hat{\boldsymbol{y}} = \begin{pmatrix} \hat{q}(s, a_1, \boldsymbol{w}) \\ \vdots \\ \hat{q}(s, a_n, \boldsymbol{w}) \end{pmatrix} \tag{2.44}$$

2.4 Policy-based learning

In this section, we explore a new approach that involves learning a parameterized policy without relying on action-value estimates. The policy's parameter vector is denoted as $\boldsymbol{\theta}$, and the probability of selecting actions is represented as $\pi(a|s, \boldsymbol{\theta})$. We focus on methods that update the policy parameter based on the gradient of a scalar performance measure $J(\boldsymbol{\theta})$, aiming to maximize performance without necessarily using a value function for action selection. But later, in section 2.4.3 we will describe *Actor-Critic* methods that combine the policy gradient methods with the value-based methods [17].

2.4.1 Stochastic Policy Gradient Methods

All algorithms that use a parameterized policy are referred to as policy gradient methods. We can represent the policy with a parameter vector $\boldsymbol{\theta} \in \mathbb{R}^d$, where $d \in \mathbb{N}$. Then, extending eq. (2.11), we can formulate the probability of executing action a in the state s with parameters $\boldsymbol{\theta}$ at time-step t of policy $\pi_{\boldsymbol{\theta}}$ as:

$$\pi(a|s, \boldsymbol{\theta}) = \Pr\left[A_t = a | S_t = s, \boldsymbol{\theta}_t = \boldsymbol{\theta} \right] \tag{2.45}$$

If a performance measure $J(\boldsymbol{\theta})$ is introduced on the parameter vector, then the policy gradient methods can iteratively update the parameter vector using the gradient ascent procedure as follows:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha \widehat{\nabla J(\boldsymbol{\theta}_t)} \tag{2.46}$$

where $\widehat{\nabla J(\boldsymbol{\theta}_t)}$ is a stochastic estimate whose expectation is the true gradient of $J(\boldsymbol{\theta})$ concerning $\boldsymbol{\theta}$.

Deriving a loss function for the performance measure in policy gradient methods is not as straightforward as in the case of value-based methods. Similar to the distinction between MC methods and TD methods, we need to differentiate between finite and infinite episodes. For both cases, the theorem on policy gradients provides a way to compute the gradient $\nabla J(\boldsymbol{\theta})$ [17, 18].

The Policy Gradient Theorem

For finite episodes, we can measure the quality of the parameter vector based on the return of the first state of the episode. The policy gradient theorem for the episodic case establishes the gradient of the return of the first state of the episode with respect to the parameter vector.

First note that the gradient of the state-value function can be written in terms of the action-value function as [17]:

$$\begin{aligned}
\nabla v_{\pi_{\boldsymbol{\theta}}}(s) &= \nabla \left(\sum_a \pi(a|s,\boldsymbol{\theta}) q_{\pi_{\boldsymbol{\theta}}}(s,a) \right) \\
&= \sum_a (\nabla \pi(a|s,\boldsymbol{\theta}) q_{\pi_{\boldsymbol{\theta}}}(s,a) + \pi(a|s,\boldsymbol{\theta}) \nabla q_{\pi_{\boldsymbol{\theta}}}(s,a)) \\
&\vdots \\
&= \sum_s \sum_{k=0}^{\infty} Pr(s \to x, k, \pi_{\boldsymbol{\theta}}) \sum_a \nabla \pi(a|s,\boldsymbol{\theta}) q_{\pi_\theta}(s,a)
\end{aligned} \tag{2.47}$$

where $\Pr[s \to x, k, \pi_{\boldsymbol{\theta}}]$ is the probability of a state transition from s to x in k steps under policy $\pi(a|s,\boldsymbol{\theta})$. This probability thus serves as a weighting of the gradient of the return. Moreover, this can be further written:

$$\begin{aligned}
\nabla J(\boldsymbol{\theta}) &= \nabla v_{\pi_{\boldsymbol{\theta}}}(s_0) \\
&= \sum_s \left(\sum_{k=0}^{\infty} Pr(s_0 \to s, k, \pi_{\boldsymbol{\theta}}) \right) \sum_a \nabla \pi(a|s,\boldsymbol{\theta}) q_{\pi_{\boldsymbol{\theta}}}(s,a) \\
&= \sum_s \eta(s) \sum_a \nabla \pi(a|s,\boldsymbol{\theta}) q_{\pi_{\boldsymbol{\theta}}}(s,a) \\
&= \sum s' \eta(s') \sum_s \frac{\eta(s)}{\sum_{'s} \eta(s')} \sum a \nabla \pi(a|s,\boldsymbol{\theta}) q_{\pi_{\boldsymbol{\theta}}}(s,a) \\
&= \sum_{s'} \eta(s') \sum_s \mu(s) \sum_a \nabla \pi(a|s,\boldsymbol{\theta}) q_{\pi_{\boldsymbol{\theta}}}(s,a) \\
&\propto \sum_s \mu(s) \sum_a \nabla \pi(a|s,\boldsymbol{\theta}) q_{\pi_{\boldsymbol{\theta}}}(s,a)
\end{aligned} \tag{2.48}$$

where s_0 is the first state of the episode, s' is the next state, $\eta(s)$ is the probability of being in state s at particular time-step and G_t is the return of the episode [17, 18].

The eq. (2.48) shows, that $J(\boldsymbol{\theta})$ is proportional to the probability sum of $\sum_s \mu(s)$, which denotes the likelihood that the agent is in the state s, multiplied by the sum of weighted gradients of the probabilities for the selection of actions. The proportionality constant is represented as $\sum_s \eta(s)$. Practically, this equation provides insight that when the agent is in a state, it should move in the direction of weights that represent the greatest increase in the probability of selecting an action in that state, weighted by the expected return for that state and action [18].

In the case of infinite episodes, the performance measure $J(\boldsymbol{\theta})$ is the average reward per time-step. The gradient $\nabla J(\boldsymbol{\theta})$ can then be determined by the theorem of policy gradients

for infinite episodes, see eq. (2.49), thus the calculations of the gradient for the infinite case correspond to those of the episodic case.

$$\nabla J(\boldsymbol{\theta}) = \sum_s \mu(s) \sum_a \nabla \pi(a|s, \boldsymbol{\theta}) q_{\pi_\theta})(s, a) \tag{2.49}$$

Discrete Action Space

Policy gradient methods allow for flexible parameterization of the policy as long as it is differentiable with respect to its parameters. A common approach for discrete action spaces is to use softmax in action preferences, where actions with higher preferences have higher probabilities of being selected:

$$\pi(a|s, \boldsymbol{\theta}) = \frac{e^{h(s,a,\boldsymbol{\theta})}}{\sum_b e^{h(s,b,\boldsymbol{\theta})}}, \tag{2.50}$$

where $\pi(a|s, \theta)$ is the probability of selecting action a in state s with policy parameterized by θ, and $h(s, a, \theta)$ is the parameterized numerical preference for state-action pair (s, a). The preferences $h(s, a, \theta)$ can be parameterized using deep neural networks or linear features:

$$h(s, a, \boldsymbol{\theta}) = \boldsymbol{\theta}^T x(s, a), \tag{2.51}$$

where θ is the vector of parameters (e.g., connection weights) and $x(s, a)$ is a feature vector for state-action pair (s, a). Choosing an appropriate reduction schedule for the temperature parameter in softmax can be challenging without prior knowledge of the true action values [17].

2.4.2 REINFORCE: Monte Carlo Policy Gradient

The REINFORCE algorithm is a purely policy-based approach that utilizes the return of complete finite episodes for parameter updates, that is the reason of *Monte Carlo* in the name of this method. Similar to classical TD methods, it adjusts the return gradually with each time step, unlike MC methods where the return is updated at the end of each episode [14, 17].

First, we extend the introduction of A_t in REINFORCE similar to how we introduced S_t in eq. (2.48), by replacing a sum with an expectation under the policy π, and then sampling from it. We add the necessary weighting by multiplying and dividing the summed terms by $\pi(a|S_t, \boldsymbol{\theta})$ without changing the equality. Continuing from eq. (2.48), we have [17, 14]:

$$\begin{aligned}
\nabla J(\boldsymbol{\theta}) &\propto E_\pi \left[\sum_a q_{\pi_{\boldsymbol{\theta}}}(s, a) \nabla \pi(a|s, \boldsymbol{\theta}) \right] \\
&= E_\pi \left[\sum_a \pi(a|s, \boldsymbol{\theta}) q_{\pi_{\boldsymbol{\theta}}}(s, a) \frac{\nabla \pi(a|s, \boldsymbol{\theta})}{\pi(a|s, \boldsymbol{\theta})} \right] \\
&= E_\pi \left[\sum_a \pi(a|s, \boldsymbol{\theta}) q_{\pi_{\boldsymbol{\theta}}}(s, a) \nabla \log \pi(a|s, \boldsymbol{\theta}) \right] \\
&= E_\pi \left[\sum_a G_t \nabla \log \pi(a|s, \boldsymbol{\theta}) \right]
\end{aligned} \tag{2.52}$$

In this algorithm, the weight vector $\boldsymbol{\theta}$ is updated using the product of the return G_t and the gradient vector $\nabla \ln \pi(a|s, \boldsymbol{\theta})$, which represents the steepest increase in the probability of selecting action a in state s. This policy parameterization vector adjustment ensures that $\boldsymbol{\theta}$ is updated with the scaling factor G_t. Update equation for the policy parameterization vector $\boldsymbol{\theta}$ is [18, 17, 14]:

$$\begin{aligned} \boldsymbol{\theta}_{t+1} &\doteq \boldsymbol{\theta}_t + \alpha G_t \frac{\nabla \pi(a|s, \boldsymbol{\theta}_t)}{\pi(a, s, \boldsymbol{\theta}_t)} \\ \boldsymbol{\theta}_{t+1} &\doteq \boldsymbol{\theta}_t + \alpha G_t \nabla \ln \pi(a|s, \boldsymbol{\theta}_t) \end{aligned} \tag{2.53}$$

and the final equation used in the algorithm looks like this (including the discount factor):

$$\boldsymbol{\theta}_{t+1} \doteq \boldsymbol{\theta}_t + \alpha \gamma^t G_t \nabla \ln \pi(a|s, \boldsymbol{\theta}_t) \tag{2.54}$$

The algorithm employs stochastic gradient ascent for optimization, as the weights are updated at each time step. Depending on whether the action space of the environment is discrete or continuous, the policy parametrizations described in the previous subsection can be used for the gradient $\nabla \ln \pi(a|s, \boldsymbol{\theta}_t)$. One drawback of this algorithm is the high variability in returns between time steps and the associated "slowness" of the learning process. This variance arises from the non-adaptive and loose formulation of the scaling factor. See algorithm 2 for the pseudocode of the REINFORCE algorithm.

The improvement of the REINFORCE algorithm is the REINFORCE with Baseline $b(s)$.

$$\nabla J(\boldsymbol{\theta}) \propto E_\pi \left[\sum_a (q_\pi(s, a) - b(s)) \nabla \log \pi(a|s, \boldsymbol{\theta}) \right] \tag{2.55}$$

$$\boldsymbol{\theta}_{t+1} \doteq \boldsymbol{\theta}_t + \alpha \gamma^t (G_t - b(s)) \nabla \ln \pi(a|s, \boldsymbol{\theta}_t) \tag{2.56}$$

One natural choice for the baseline is an estimate of the state value, $\hat{v}(S_t, w)$, where $w \in \mathbb{R}^d$ is a weight vector learned by one of the methods presented in previous sections [18, 17, 14]. The reason why we introduce REINFORCE with Baseline is that the next section 2.4.3 will introduce Actor-Critic methods, which are a combination of REINFORCE with Baseline and TD methods.

2.4.3 Actor-Critic Policy Gradient Methods

In actor-critic methods, the state-value function is used to assess actions, including the second state of a transition, and estimate the one-step return. This is different from REINFORCE with baseline, which only estimates the value of the first state. One-step actor-critic methods are fully online and incremental, making them easier to understand and implement compared to eligibility trace methods. A vector of constants can be used as the baseline. The concept of Actor-Critic methods can be formulated, when a function approximator with bootstrapping, such as TD(0) or SARSA(0), is used as the baseline. In this case, the Critic computes the scaling factor and the Actor adjusts the policy parameter $\boldsymbol{\theta}$. The weight update formula for the Actor-network is then as follows [17, 18]:

$$\begin{aligned} \theta_{t+1} &\doteq \theta_t \alpha \left(G_{t:t+1} - \hat{v}(S_t, \boldsymbol{w}) \right) \nabla \ln \pi(A_t|S_t, \theta_t) \\ &= \theta_t + \alpha \left(R_{t+1} + \gamma \hat{v}(S_{t+1}, \boldsymbol{w} - \hat{v}(S_t, \boldsymbol{w})) \right) \nabla \ln \pi(A_t|S_t, \theta_t) \\ &= \theta_t + \alpha \delta_t \nabla \ln \pi(A_t|S_t, \theta_t) \end{aligned} \tag{2.57}$$

The policy gradient update formula can be written as:

$$\nabla J(\boldsymbol{\theta}) \propto \sum \mu(s) \sum (q_\pi(\boldsymbol{\theta})(s,a) - b(s)) \nabla \pi(a|s,\boldsymbol{\theta}) \tag{2.58}$$

Continuous Action Space

Continuous Action-space Policy Gradient methods can be used in environments with discrete as well as with continuous action space. Value-based methods have the disadvantage that they can only be used in environments with discrete action space. Whether a given policy gradient method can be used on discrete or continuous action space is determined by the policy parametrization. In the case of a discrete action space, a numerical value function, denoted as $h(s,a,\theta) \in \mathbb{R}$, can be computed for each state-action pair. The computation of these values can be done, for example, via a neural network. An exponential Softmax distribution can then specify the probability with which an action should be selected, as shown in eq. (2.50) [18, 17]:

For large or continuous action spaces, instead of evaluating individual actions, parameters of distribution functions such as the Normal distribution are computed. The parameters of the Normal distribution are the mean $\mu \in \mathbb{R}$ and the variance $\sigma^2 \leq 0$. The Normal distribution is defined on $\mathbb{R}$, which makes it suitable for continuous action spaces. The mean and variance can be parametrized and approximated as [17]:

$$\pi(a|s,\theta) = \frac{1}{\sqrt{2\pi}\sigma(s,\theta)} \exp\left(-\frac{(a-\mu(s,\theta))^2}{2\sigma(s,\theta)^2}\right) \tag{2.37}$$

From eq. (2.37), it can be seen that the policy parameter vector can be described as $\theta = (\theta_\mu, \theta_\sigma)$, and its parameters can be computed using a single neural network. This enables the implementation of continuous actions for stochastic policies [18, 17, 14].

Chapter 3

Stock Portfolio Allocation

In this chapter, our primary objective is to present the methodology we used to tackle the Reinforcement Learning problem, with a main focus on the Portfolio Allocation task. We will provide a detailed explanation of the entire pipeline fig. 3.1, starting from the *Data Engineering Step*. In this step, the data is prepared for use in reinforcement learning. This involves collecting raw data from various sources, transforming the data into features suitable for reinforcement learning, and splitting the dataset into training and testing sets section 3.1. Then go on the *Environment Modeling Step*. This step involves configuring the environment in which the RL model will operate, including setting up the state and action spaces and defining the reward function, this is described in section 3.2. In the next chapter we also explain the *Agent Layer*, where the model itself is developed here. It includes training the model on the training set, evaluating the model's performance on the testing set, fine-tuning the model's hyperparameters to optimize its performance, and conducting robustness testing on unseen data multiple times to ensure that the model can generalize well, this is discussed in chapter 4.

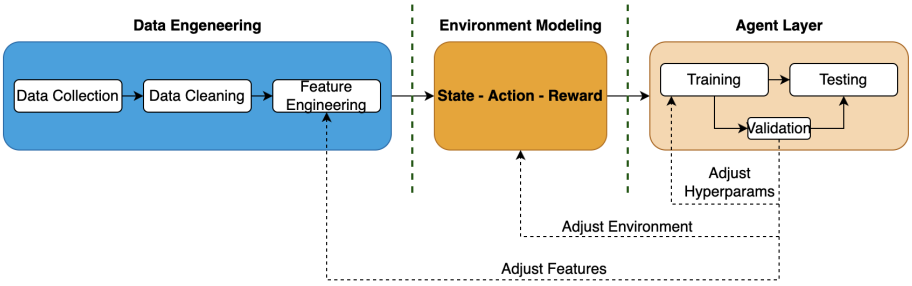


Figure 3.1: The pipeline for solving the RL problem in the stock market [9]

3.1 Data Engineering

Data engineering is critical in AI to ensure the quality, preparation, integration, scalability, governance, and performance optimization of models. It involves cleaning and transforming data for training an accurate model. In this section, we describe our approach to data engineering used for Portfolio Management by first defining the data collection section 3.1.1, data cleaning section 3.1.2, feature engineering section 3.1.3 and dataset splitting section 3.1.4.

3.1.1 Data Collection

Raw data was gathered from these sources:

- Yahoo Finance: `https://finance.yahoo.com/`
- Financial Modeling Prep: `https://financialmodelingprep.com/`
- TradingView: `https://www.tradingview.com/`

these sources provide financial data for 65000+ tickers, with a history of over 30 years.

We focus on tickers (companies) from *Dow Jones Industrial Average (DJIA)*, which includes **30 companies**. More about which companies are included in DJIA, you can read in [20].

3.1.2 Data Cleaning

We use the following company data in our datasets: prices, financial statements, and technical indicators. Prices are used to calculate earnings, and financial statements are used to calculate fundamentals. Fundamental indicators and technical analysis indicators are used for feature engineering. Data cleaning is done by removing rows with missing values. Missing values are replaced by the average of the column. If data is completely missing on a date, all rows up to that next date are removed for the other companies. This can happen mostly for companies that are not listed in any time period. Data cleaning is performed for all data sets.

3.1.3 Feature Engineering

In this subsection, we describe our approach to feature engineering which we divide into three parts: *Fundamental Analysis*, described in section 3.1.3, *Technical Analysis*, described in section 3.1.3, and *Combined Fundamental and Technical Analysis* is described in section 3.1.3. The datasets comprise additional attributes such as candlestick patterns (price) and volumes, which are not discussed in this subsection.

Fundamental Analysis

The first features that we used for our datasets are fundamental information about companies. This type of data is essentially the key and crucial information that is utilized to analyze the financial well-being, performance, and worth of a company or asset. It encompasses information regarding the financial statements, business operations, management team, industry, and economic backdrop of a company. Investors, analysts, and financial experts commonly use fundamental data to make informed investment decisions and evaluate the inherent value of an asset. The subsequent text outlines the features that were incorporated into our dataset. There also exists a plethora of other fundamental features that can be used, but we have chosen to focus on the following features, with the theoretical concepts sourced from [6]:

Operating Margin The Operating Margin represents how efficiently a company is able to generate profit through its core operations. It is expressed on a per-sale basis after accounting for variable costs but before paying any interest or taxes (EBIT). Higher margins are considered better than lower margins and can be compared between similar competitors

but not across different industries. To calculate the operating margin, divide operating income (earnings) by sales (revenues):

$$Operating\ Margin = \frac{Operating\ Income}{Net\ Sales} \tag{3.1}$$

where Operating Income refers to the adjusted revenue of a company after all expenses of operation and depreciation are subtracted. Expenses of operation or operating expenses are simply the costs incurred in order to keep the business running. Net Sales may be defined as money paid by customers. Sales are a company's core revenue for a given period. Operating margin, expressed as a percentage, indicates the earnings generated from each dollar of sales after accounting for direct costs. Higher margins mean more profit from each sale.

Net Profit Margin The Net Profit Margin is a profitability ratio that measures a company's ability to generate income after all expenses and taxes have been paid Net Income. It is calculated by dividing a company's Net Income, by its Total Revenue. The Net Profit Margin is a measure of how much of each dollar of revenue is left over after all expenses and taxes have been paid. It is a useful metric for comparing the profitability of different companies in the same industry. The higher the net profit margin, the more profitable a company is. The net profit margin is calculated as follows:

$$Net\ Profit\ Margin = \frac{Net\ Income}{Total\ Revenue} \tag{3.2}$$

Return On Assets Return on assets (ROA) is a measure of profitability that calculates how much profit a company makes with the money it has invested. It is calculated by dividing a company's Net Income by its Total Assets. The higher the ROA, the more profitable a company is. The ROA is calculated as follows:

$$Return\ On\ Assets = \frac{Net\ Income}{Total\ Assets} \tag{3.3}$$

Return On Equity Return on equity (ROE) is a measure of profitability that calculates how much profit a company makes with the money shareholders have invested. It is calculated by dividing a company's Net Income by its Average Shareholders' Equity. The higher the ROE, the more profitable a company is. The ROE is calculated as follows:

$$Return\ On\ Equity = \frac{Net\ Income}{Average\ Shareholders'\ Equity} \tag{3.4}$$

Current Ratio The current ratio is a liquidity ratio that measures a company's ability to pay short-term and long-term obligations. It is calculated by dividing a company's Current Assets by its Current Liabilities. The higher the current ratio, the more capable a company is of paying its short-term and long-term obligations. The current ratio is calculated as follows:

$$Current\ Ratio = \frac{Current\ Assets}{Current\ Liabilities} \tag{3.5}$$

where Current Assets is Cash&Equivalent + Short Term Investments + Account Receivable + Inventory. If the Current Ratio is lower than 1 that is a sign of financial distress, and the company could be unable to pay its short-term and long-term obligations. On the other hand, if the Current Ratio is too high, it could be a sign that the company is not using its assets efficiently.

Quick Ratio The quick ratio is a liquidity ratio that measures a company's ability to pay short-term obligations. It is calculated by dividing a company's Quick Assets by its Current Liabilities. The higher the quick ratio, the more capable a company is of paying its short-term obligations. The quick ratio is calculated as follows:

$$Quick\ Ratio = \frac{Quick\ Assets}{Current\ Liabilities} \tag{3.6}$$

where Quick Ratio is Cash & Equivalents + Marketable Securities + Net Accounts Receivable. If the company would have a Quick Ratio is lower than 1, it would be a sign of financial distress, and the company would be unable to pay its short-term obligations.

Cash Ratio The cash ratio is a liquidity ratio that measures a company's ability to pay short-term obligations. It is calculated by dividing a company's Cash & Equivalents by its Current Liabilities. The higher the cash ratio, the more capable a company is of paying its short-term obligations. The cash ratio is calculated as follows:

$$Cash\ Ratio = \frac{Cash\&Equivalent}{Current\ Liabilities} \tag{3.7}$$

Inventory Turnover The inventory turnover ratio is a measure of how efficiently a company is managing its inventory. It is calculated by dividing a company's COGS by its Average Value of Inventory. The higher the inventory turnover ratio, the more efficiently a company is managing its inventory. The inventory turnover ratio is calculated as follows:

$$Inventory\ Turnover = \frac{COGS}{Average\ Value\ of\ Inventory} \tag{3.8}$$

where COGS is an acronym for Cost of Goods Sold, which is also known as the cost of sales. To calculate inventory turnover, analysts use COGS instead of sales because inventory is valued at cost. Some companies may use sales instead of COGS, which can inflate the ratio.

Receivables Turnover The receivables turnover ratio is a measure of how efficiently a company is managing its accounts receivable. It is calculated by dividing a company's Net Credit Sales by its Average Accounts Receivable. The higher the receivables turnover ratio, the more efficiently a company is managing its accounts receivable. The receivables turnover ratio is calculated as follows:

$$Receivables\ Turnover = \frac{Net\ Credit\ Sales}{Average\ Accounts\ Receivable} \tag{3.9}$$

where *Net Credit Sales* is the total sales minus the cash sales. The receivables turnover ratio measures a company's ability to collect accounts receivable. A low ratio suggests difficulty collecting payments, while a high ratio may indicate efficient collection, but may also suggest lost sales due to not extending credit long enough.

Payables Turnover The accounts payable turnover ratio is a short-term liquidity measure used to quantify the rate at which a company pays off its suppliers. Accounts payable turnover shows how many times a company pays off its accounts payable during a period. It is calculated by dividing a company's total supply purchases by its average accounts

payable. The higher the payables turnover ratio, the more efficiently a company is managing its accounts payable. The payables turnover ratio is calculated as follows:

$$Payables\ Turnover = \frac{TSP}{(BAP + EAP)/2} \tag{3.10}$$

where TSP is the total supply purchase, BAP is the beginning accounts payable, and EAP is the ending accounts payable. If the payables turnover ratio is too low, it could be a sign that the company has trouble paying its suppliers. If the payables turnover ratio is too high, it could be a sign that the company is not using its assets efficiently.

Debt Ratio The debt ratio is a measure of a company's financial leverage. It is calculated by dividing a company's Total Liabilities by its Total Assets. The higher the debt ratio, the more debt a company is using to finance its assets. The debt ratio is calculated as follows:

$$Debt\ Ratio = \frac{Traditional\ Debt + (Accounts\ Payable + Taxes\ Payable)}{TotalAssets} \tag{3.11}$$

A lower debt ratio is preferable. A Debt Ratio greater than 1.0 implies that a company has more debt than assets, whereas a Debt Ratio less than 1 indicates that a company has more assets than debt.

Debt Equity Ratio The Debt to Equity Ratio (D/E) is a measure of a company's financial leverage. It is calculated by dividing a company's Total Liabilities by its Total Equity. The higher the debt-equity ratio, the more debt a company is using to finance its assets. The debt-equity ratio is calculated as follows:

$$Debt\ Equity\ Ratio = \frac{Total\ Liabilities}{Total\ Equity} \tag{3.12}$$

where Total Liabilities are Traditional Debt + (Accounts Payable + Taxes Payable). A lower Debt Equity Ratio is preferable.

Price Earnings Ratio The Price Earnings Ratio (P/E) is a measure of a company's value relative to its earnings. It is calculated by dividing a company's Stock Price by its Earnings per Share. The higher the price-earnings ratio, the more expensive a company's stock is relative to its earnings. The price-earnings ratio is calculated as follows:

$$Price\ Earnings\ Ratio = \frac{Stock\ Price}{Earnings\ Per\ Share} \tag{3.13}$$

High P/E implies investors anticipate greater future earnings growth compared to low P/E companies. A low P/E can signal undervaluation or exceptional performance relative to past trends.

Price Book Value Ratio The Price Book Value Ratio (P/B) is a measure of a company's value relative to its book value. It is calculated by dividing a company's Stock Price by its Book Value per Share. The higher the price-book value ratio, the more expensive a company's stock is relative to its book value. The price-book value ratio is calculated as follows:

$$Price\ Book\ Value\ Ratio = \frac{Stock\ Price\ per\ Share}{Book\ Value\ per\ Share} \tag{3.14}$$

A low P/B ratio, especially below one, could signal undervaluation where the stock price is lower than the value of the company's assets. A P/B ratio above one indicates the stock is trading at a premium to book value, potentially overvalued. For instance, a P/B ratio of three means the stock trades at three times its book value.

Dividend Yield The dividend yield, expressed as a percentage, is a financial ratio (dividend/price) that shows how much a company pays out in dividends each year relative to its stock price. It is calculated by dividing a company's Annual Dividend by its Stock Price. It is a measure of a company's profitability. The higher the dividend yield, the more profitable a company is. The dividend yield is calculated as follows:

$$Dividend\ Yield = \frac{Annual\ Dividend}{Stock\ Price} \tag{3.15}$$

Technical Analysis

Technical analysis is a method used in financial markets such as stocks, currencies, and commodities to analyze historical price data and identify patterns, trends, and signals that can be used to make trading decisions. Technical analysis is based on the belief that historical price and volume data can provide insight into future price movements and focuses primarily on the analysis of price charts and other technical indicators. In this section we present the technical indicators that we have selected based on the correlation between all indicators.

Technical indicators are calculated based on historical share price data. To calculate these indicators, the Finta framework was used, which provides more than 100 different functions for calculating technical indicators. The framework is written in Python and is available on GitHub [11].

First, we calculate all technical indicators for each stock present in the dataset, followed by determining the correlation matrix between all indicators. This matrix comprises the correlation coefficient values between each pair of indicators, which serve as a gauge for measuring the linear correlation between two variables. We apply a drop threshold of 0.5 for the correlation coefficient, implying that if $|\rho| > 0.5$, we discard either the first or second indicator, where $|\rho|$ represents the absolute value of the correlation between the two variables (indicators). As the image (correlation matrix) for all indicators is extensive, we only present the correlation matrix of the final indicators (correlation matrix of the uncorrelated indicators), in fig. 3.2, that we utilized in the *Technical Analysis Dataset*. Since the correlation matrix has symmetry, only the segment above the diagonal is presented.

Combined Fundamental and Technical Analysis

Combining datasets is a process in which two or more datasets are merged or joined together to create a new dataset that contains information from both. In this case, the combined fundamental and technical analysis involves combining features from both fundamental analysis and technical analysis, both described in section 3.1.3.

By combining fundamental and technical analysis, we can create a more comprehensive and accurate picture of the market. For example, fundamental analysis can help identify undervalued or overvalued assets, while technical analysis can help identify trends and potential buying or selling opportunities. The specific features used in the combined analysis can depend on the specific dataset and the reinforcement learning task being performed. However, we leave all features from both analyses.

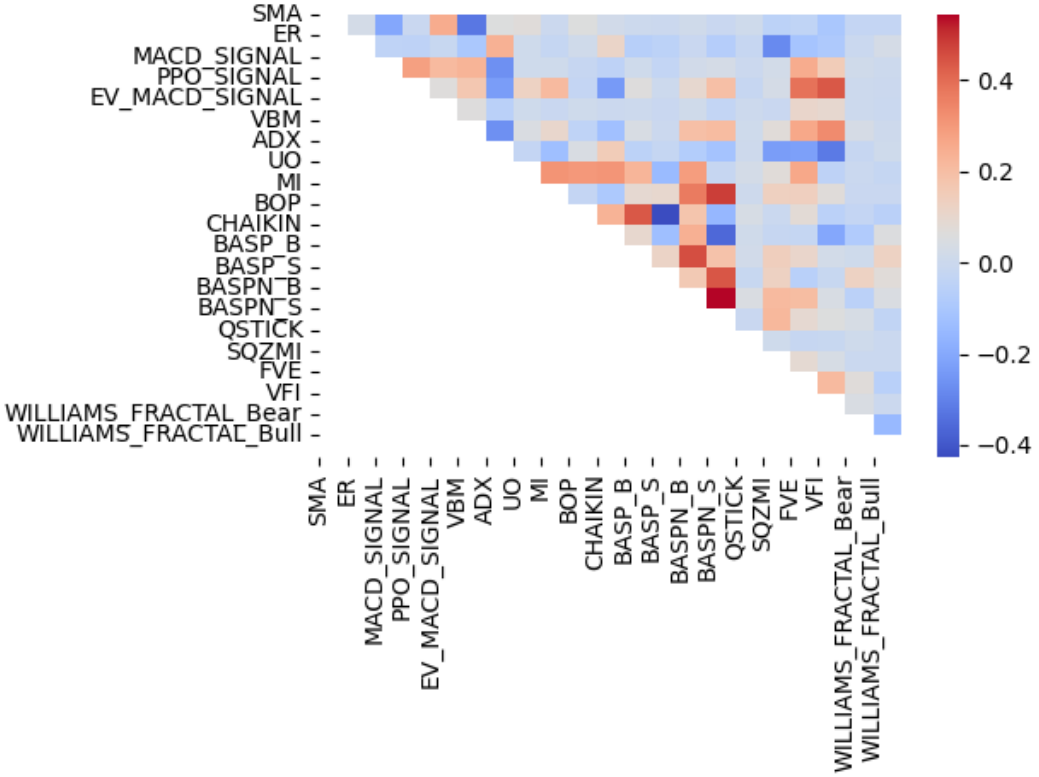


Figure 3.2: Correlation matrix of technical indicators after removing correlated indicators.

3.1.4 Dataset Splitting

Dataset splitting refers to the process of dividing a dataset into two or more subsets to be used for training and testing reinforcement learning models. In this particular case, the dataset is a time series data, which means that the order of the data points is important, and the dataset must be split in a way that preserves the temporal ordering.

The division coefficient of 0.6 means that 60% of the dataset will be used for training, and 40% will be used for testing. This is a common split ratio, but the exact ratio can vary depending on the size and complexity of the dataset, as well as the specific reinforcement learning task being performed.

In this case, since the dataset consists of **daily data** and spans a period from **2008-03-20 to 2022-12-16**, we can split it based on time. The first 60% of the data, starting from the beginning of the dataset on 2008-03-20, will be used for training, and the remaining 40% of the data will be used for testing. This ensures that the model is trained on past data and tested on future data, which is a more realistic scenario.

The training set will be used to train the reinforcement learning model, while the testing set will be used to evaluate its performance. By splitting the dataset, we can avoid overfitting, which is when the model memorizes the training data and performs poorly on new, unseen data. The testing set provides a way to measure the model's generalization performance on unseen data, which is an important metric for evaluating the model's effectiveness.

Dataset splitting is an important step in reinforcement learning that helps ensure that the model is trained and tested on different subsets of data. In this case, the dataset is split into training and testing sets based on time, with 60% of the data used for training and 40% used for testing. This split ensures that the model is evaluated on unseen data and can generalize well to new data.

3.2 Environment Modeling

In the preceding section, we explained the process of creating datasets and selecting features to represent the state of the environment. This section delves deeper into the modeling of the entire environment, encompassing the *Action Space*, *State Space*, and *Reward function*. The action space is defined as a vector of weights, where each weight denotes the percentage amount that a particular stock is represented in the portfolio. This is discussed in detail in section 3.2.1. The state space is a matrix of K features of each stock (N stocks) that reflects the state of the environment at each time step t, as explained in section 3.2.1. The reward function is employed to calculate the reward for each action taken in the current state at time step t, as elaborated in section 3.2.1. The objective of the agent is to maximize the reward by selecting the best action/weights for each state, which involves reallocating the stock weights in the portfolio. In other words, the agent aims to determine the optimal weights for each stock in the portfolio at a particular time step t to obtain maximum portfolio appreciation. The fig. 3.3 illustrates how the agent generates the weights for the portfolio.

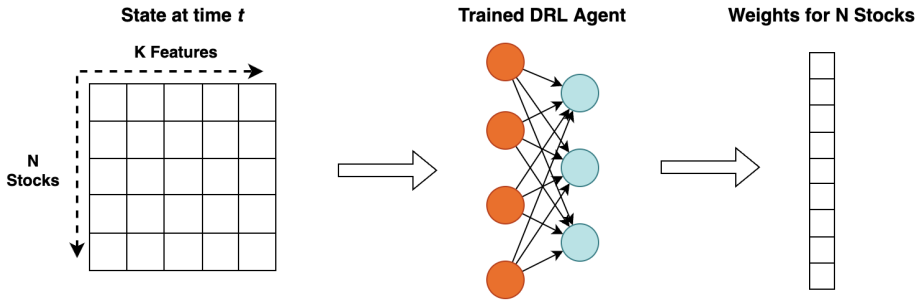


Figure 3.3: How the agent produces the weights for the portfolio [4].

3.2.1 Portfolio Management Task

Portfolio allocation or management refers to the process of selecting and distributing investments in a way that meets the investor's goals and risk tolerance. It involves diversifying the portfolio across different asset classes, such as stocks, bonds, and real estate, to reduce risk and increase returns. In our thesis we focus on diversifying the capital between N **stocks**, where N is the number of stocks in the portfolio. The ultimate goal of portfolio management is to maximize the returns while minimizing the risk for the investor. The risk can be evaluated as the variance of the portfolio returns [4].

State and Observation Space

In the context of portfolio management, the *State Space* and *Observation space* refer to the same set of variables. The state space is a matrix with dimensions $N \times K$, where N

represents the number of risky assets, or stocks, and K represents the number of features used to describe the environment. These features were discussed in detail in the previous section section 3.1, and consist of fundamental or technical indicators that describe the current condition of each stock at a specific time step t. In our case, the time step is defined as one day, so we use **daily data** representing one day of stock market activity.

Action Space

The action space for a portfolio allocation agent is a vector of $N+1$ values, which represents the weights of N stocks and the weight of cash (How much is each stock represented in the portfolio). The reason for including the weight of cash in the action space is that sometimes investors do not want to invest all of their capital and prefer to keep some cash as a reserve for future trades. The action provided by the agent is a vector of values **bounded between 0 and 1**.

The action vector is then normalized using the *softmax function*. The softmax function converts the original vector of real values into a new vector of real values that fall within the range of 0 to 1, and all values summed to 1. This allows for a more intuitive representation of the weights and ensures that the weights always add up to 1 [4].

Action vector is provided by an agent at time step t and is defined as follows:

$$\boldsymbol{a}(t) = [a_1(t), a_2(t), \ldots, a_N(t), a_{cash}(t)] \tag{3.16}$$

where $a_i(t)$ is the value before normalization for stock i at time step t and $a_{cash}(t)$ is for cash at time step t. The weights vector $\boldsymbol{w}(t)$ is defined as follows:

$$\begin{aligned} \boldsymbol{w}(t) = \text{softmax}(\boldsymbol{a}(t)) &= \frac{\exp(\boldsymbol{a}(t))}{\sum_{i=1}^{N} \exp(a_i(t))} \\ &= [w_1(t), w_2(t), \ldots, w_N(t), w_{cash}(t)] \end{aligned} \tag{3.17}$$

where $w_i(t)$ is the normalized weight for stock i at time step t and $w_{cash}(t)$ is for cash at time step t. All weights vectors must sum to 1, that is the first constraint, and it directly implies the second constrained that each weight must be bounded between 0 and 1:

$$\sum_{i=1}^{N} w_i(t) = 1; \quad w_i(t) \in [0, 1]; \quad \text{for } t = 1, \ldots, T \tag{3.18}$$

Reward Function

The reward function represents the change in the value of the portfolio from one time step to the next. First we need to define *Price Relative Vector* $\boldsymbol{y}(t) \in \mathbb{R}^N$:

$$\boldsymbol{y}(t) = \left[\frac{p_1(t)}{p_1(t-1)}, \frac{p_2(t)}{p_2(t-1)}, \cdots, \frac{p_N(t)}{p_N(t-1)}, 1 \right] \text{ , for } t = 1, \ldots, T \tag{3.19}$$

where the $p_i(t)$ is the price of a risky asset (stock) at time step t, and $\frac{p_i(t)}{p_i(t-1)}$ represents the relative change in price from one-time slot to the next of stock i, we incorporate into the vector also the change of cash, which is constant value 1 (we assume that cash value does not change its value).

Now we can define our **Reward function** as the rate of portfolio return:

$$\rho(t) = \boldsymbol{w}(t-1)^\top \boldsymbol{y}(t) - 1 \text{ , for } t = 1, \ldots, T \tag{3.20}$$

The reward function measures the change in the portfolio's value between consecutive time steps. We assume that there are no transaction costs associated with each operation, as these can vary depending on the broker. Additionally, we don't factor in any dividends. Instead, the reward is solely based on the change in the value of each stock.

Lastly we defined the *portfolio value* $v(t) \in \mathbb{R}$ at time step t. First let the $v(0)$ be an initial capital, then for $t = 1, \ldots, T$ is portfolio value defined as follows:

$$\begin{aligned} v(t) &= v(0) \prod_{\tau=1}^{t} [\rho(\tau) + 1] \\ &= v(t-1)(\rho(t) + 1) \text{ , for } t = 1, \ldots, T \end{aligned} \tag{3.21}$$

where $\rho(t)$ is change in portfolio value (*Reward*) at time step t.

3.2.2 How we determine agent performance

The agent's performance is assessed using the *test/total_reward* attribute. To compute the value of *test/total_reward*, we initialize the portfolio value v at time step $t = 0$ to value 1, so $v(0) = 1$ and then further calculate it as the cumulative product of the rewards obtained by the agent during the testing phase, as shown in eq. (3.21).

For example, if the agent's received a reward at time step $t = 1$ is $\rho(1) = -0.1$, which implies that the portfolio is currently worth 0.9 or 90% from its previous value, in other words, portfolio value decreased by 10%, then the value of *test/total_reward* would be at the start of the times step $t = 2$:

$$\begin{aligned} v(1) &= (CurrentReward + 1) \times CurrentPortfolioValue \\ &= (\rho(1) + 1) \times v(0) \\ &= (-0.1 + 1) \times 1 = 0.9 \end{aligned} \tag{3.22}$$

Let now consider the second time step $t = 2$. If in the second time step, the agent receives a reward of 0.1, the value of *test/total_reward* would be at the start of the time step $t = 3$:

$$\begin{aligned} v(2) &= (CurrentReward + 1) \times CurrentPortfolioValue \\ &= (\rho(2) + 1) \times v(1) \\ &= (0.1 + 1) \times 0.9 = 0.99 \end{aligned} \tag{3.23}$$

Therefore, if the final value of *test/total_reward* is 1.9, it indicates that the agent obtained a *total cumulative reward* of 0.9 throughout all the steps, which is equivalent to a portfolio return of 90% over the certain period.

Chapter 4

Experiments and Results

In this chapter, we will present the outcomes of our experiments, including any issues encountered during the experimentation process. We demonstrate the performance of our agent in a given environment through a series of experiments and provide additional details on hyperparameter selection and comparison of different hyperparameter settings in section 4.3. We also evaluate and compare the advantages and disadvantages of the datasets used in training our model in 4.4, with a focus on identifying the most appropriate dataset. The robustness of our model is assessed in 4.5. Furthermore, we compare the performance of our model with standard indices such as the *S&P 500 Index* and *DJI Index*, and discuss the results of our experiments with baseline models and benchmarks. To maintain consistency in our results, we primarily compare our model's performance with the DJI Index as the datasets are derived from companies in the DJI Index. However, we also compare our results with other indices and *State-of-the-Art AI4Finance* in 4.6. All experiments were documented during our testing period from **2017-01-25** to **2022-12-16**. Last assessment will involve examining important portfolio metrics, as described in section 4.7. Finally, we summarize our findings and discuss the results of our experiments in 4.8.

In addition, we provide details of our approach to performance testing and the MLOps concepts used, including tracking training and testing of our models, in Section 4.1.2.

To ensure *Reproducibility* of our work we also make publicly available on the W&B website all datasets, models, and training/testing logs with a history of all hyperparameters for each algorithm. **By visiting the following URL: `https://wandb.ai/investai/portfolio-allocation`**, anyone can access the results of each run, and reproduce our findings while comparing them with their own experiments.

4.1 Hardware & Tools

4.1.1 Hardware

The computer used for experiments has the following specifications:

- Operating System: Ubuntu 20.04.6 LTS (GNU/Linux 5.4.0-146-generic x86_64)
- CPU: 2 x Intel Xeon CPU E5-2620 v3 @ 2.40GHz, each with six cores, for a total of 12 cores.
- RAM: 2 x 32 GB RAM running at 2133MHz, using quad-channel architecture for faster memory access

- GPU: 4 x NVIDIA GTX 1080 (Pascal) with 8GB RAM each, providing a total of 32GB GPU memory.

4.1.2 Weights & Biases

Weights & Biases (W&B) is a powerful ML experiment tracking and visualization tool that helps data scientists and machine learning practitioners manage their experiments. With W&B, users can log experiment metrics in real-time, track hyperparameters, and compare and reproduce experiments easily. W&B offers various visualization tools like interactive plots, histograms, and confusion matrices, which help users analyze and understand experimental results.

- Experiment tracking: W&B allows users to log experiment metrics such as loss, accuracy, and other custom metrics in real-time during training. These metrics are logged to a central dashboard, making it easy to monitor and compare multiple experiments.
- Hyperparameter tuning: W&B supports hyperparameter sweeps, allowing users to explore different hyperparameter configurations in parallel and find optimal hyperparameter settings for their models.
- Visualization: W&B provides a variety of visualization tools, including interactive graphs, histograms, confusion matrices, and more, to help users analyze and understand experimental results. We use some of the W&B graphs to compare hyperparameters and their values to understand their effect on the overall range of rewards appendix C.1.
- Artifact management: W&B allows users to log and version datasets, models, and other artifacts, making it easy to track and reproduce experiments with specific data and model versions.
- Collaboration: W&B enables team collaboration by allowing users to share experiment results, visualizations, and artifacts with team members, facilitating communication and collaboration among team members.

Due to the variety of features available through the W&B API, we have chosen not to provide a comprehensive description of each one. However, we recommend looking at the W&B documentation available at `https://docs.wandb.ai/` for an explanation of all APIs and features.

4.2 Focus of the experiments

Our aim is to determine the best *hyperparameters* and *datasets* for training and testing, that provide the agent to get the highest portfolio value over time. To achieve this, we use a *hyperparameter sweeping* approach to testing the hyperparameters for each algorithm and dataset. The performance of each hyperparameter configuration is determined based on the model's output, using a metric named as *test/total_reward*. This metric represents the cumulative product of the portfolio value, as described by the equation in eq. (3.21) and then further described in section 3.2.2. The higher the value of *test/total_reward*, the better the performance of the model.

We compare the best and worst performing models with State-of-the-Art AI4Finance and common indexes and strategies as baselines. Our results indicate that appropriate hyperparameters and good featured datasets can train the agent sufficiently well to outperform common indexes and other publicly available models for Portfolio Allocation (using Reinforcement Learning). This makes them valuable tools for portfolio allocation in financial markets.

Thus, the main goal of this chapter is to evaluate the trained models' performance through rigorous experimentation and identify the best *hyperparameters* and *datasets* for optimal performance.

We also test the model's *robustness* by assessing its stability using the best hyperparameters for the dataset and the specific algorithm used to train the model. This helps us ensure that the model is reliable and can perform consistently well under different conditions. The robustness test helps us identify potential weaknesses or limitations that may need to be addressed.

4.3 First experiment: Hyperparameter Tuning

Since the configuration of hyperparameters is crucial for the performance of any machine-learning model. We performed a hyperparameter sweep to find the best hyperparameters for our models through a bunch of hyperparameters shown in table 4.1. In this case, the hyperparameter sweep was performed on 27 hyperparameters using the Weights & Biases Sweep, this helps us to track and visualize the training process and store a bunch of hyperparameter configurations. The range or value of each hyperparameter was chosen based on prior knowledge.

4.3.1 Time Steps

The only hyperparameter, which stays the same throughout *hyperparameter sweeping* was *Time Steps* parameter. We conducted experiments by varying its value from 20000 to 100000 and found that values up to $50,000$ still improve the model's performance. However, values above 50000 did not provide any significant improvement in the model's performance, and the training time increased considerably. As a result, we decided to use **50000 time steps** for all the models that we experimented with in this chapter.

4.3.2 Selection of hyperparameters

In this subsection, we will discuss other hyperparameters and explain how we selected the range of values for them. Once the range or values of each hyperparameter was set, as shown in table 4.1, the process of finding the best configuration of hyperparameters was a kind of brute force, except that we at least restrict the exploration space in which the W&B Sweep is trying to find optimal hyperparameter through randomly selecting each hyperparameter for the given run.

The Sweep involved **5 distinct** reinforcement learning algorithms - namely *A2C*, *PPO*, *SAC*, *DDPG*, and *TD3* - and each of these algorithms was trained on **3 unique datasets**, which were introduced in the preceding chapter and referred to as Fundamental, Technical, and Combined. In total, this resulted in **15 different combinations** of algorithms and datasets. Furthermore, each combination was subjected to **10 rounds** of training, cul-

minating in a total of **150 train/test runs**. These runs are accessible on W&B at the following URL: `https://wandb.ai/investai/portfolio-allocation`.

It is worth emphasizing that when performing a hyperparameter sweep, certain hyperparameters may be selected even though the algorithm does not require them, that means in such cases, we simply *discard* those unnecessary chosen hyperparameters and only pass the necessary ones to the algorithm.

In table 4.1, the hyperparameters that underwent tuning during the hyperparameter sweep are displayed. The values enclosed in <> indicate the range from which W&B Sweep can randomly select a value for a particular run, while those enclosed in [ ] represent the list of values from which W&B Sweep can randomly choose a value for a particular run. The performance of the agent with respect to the hyperparameters is presented in the graphs, which compare and analyze the performance of several of the best and worst models. These graphs can be found in appendix C.1.

Parameter	**Values/range**
learning_rate	<0.0001, 0.01>
n_steps	[32, 64, 128, 256, 512, 1024, 2048]
gamma	<0.9, 0.999>
gae_lambda	<0.8, 0.999>
ent_coef	<0.0001, 0.01>
vf_coef	<0.0001, 0.01>
max_grad_norm	<0.5, 0.99>
rms_prop_eps	<0.0001, 0.01>
sde_sample_freq	<4, 32>
batch_size	[32, 64, 128, 256, 512, 1024, 2048, 4096, 8192]
n_epochs	<1, 10>
clip_range	<0.1, 0.3>
clip_range_vf	[None, 0.05, 0.1, 0.15, 0.2]
target_kl	<0.01, 0.05>
buffer_size	[1000, 2000, 3000, 4000, 5000]
learning_starts	<100, 1000>
tau	<0.001, 0.01>
train_freq	<1, 4>
gradient_steps	<1, 4>
target_update_interval	<1, 4>
target_entropy	<0.1, 0.2>
policy_delay	<1, 4>
target_policy_noise	<0.1, 0.2>
target_noise_clip	<0.1, 0.2>
exploration_fraction	<0.1, 0.2>
exploration_initial_eps	<0.1, 0.2>
exploration_final_eps	<0.1, 0.2>

Table 4.1: Hyperparameters range/values to select from, for the hyperparameter sweep.

4.3.3 Hyperparameter Sweep Results

We evaluate the agent's performance based on *test/total_reward* attribute. In appendix C.1 is shown how the hyperparameter tuning was performed on given hyperparameters and how much each influenced the *test/total_reward*. It distinguishes individual training runs, and the color of the curve corresponds to its value, a higher value of *test/total_reward* is represented by an orange curve, indicating better performance, conversely, a lower value is represented by a purple curve, indicating poor performance. These graphs have two columns: the

first column represents the *algorithm*, and the last column represents the *test/total_reward*. The hyperparameters, which were tuned during the hyperparameter sweep, are located between these two columns. Including three graphs instead of one makes the results more readable and easier to analyze.

The table 4.2 presents the configurations of the top 2 and bottom 2 models, including their hyperparameters and the corresponding *test/total_reward* values.

Model ID	p3irnh80	8tml2ozg	zfjr0ks0	pky1wslb
algo	DDPG	A2C	A2C	PPO
learning_rate	0.00685	0.00671	0.00698	0.00471
n_steps	256	128	128	32
gamma	0.9294	0.94108	0.93858	0.9289
gae_lambda	0.99624	0.90203	0.9383	0.85115
ent_coef	0.00203	0.0013	0.00525	0.00173
vf_coef	0.0082	0.00683	0.00125	0.00527
max_grad_norm	0.71475	0.56558	0.94281	0.77201
rms_prop_eps	0.00024	0.00435	0.0075	0.0067
sde_sample_freq	30	26	23	15
batch_size	128	64	8192	64
n_epochs	5	8	7	8
clip_range	0.18938	0.19559	0.2274	0.22578
clip_range_vf	0.05		0.15	0.05
target_kl	0.02394	0.04957	0.03176	0.03337
buffer_size	2000	5000	2000	2000
learning_starts	387	163	569	165
tau	0.00815	0.00461	0.00269	0.00469
train_freq	2	2	3	2
gradient_steps	1	3	1	3
target_update_interval	3	2	4	2
target_entropy	0.19013	0.15826	0.18233	0.13979
policy_delay	2	4	2	3
target_policy_noise	0.11459	0.15052	0.10674	0.12649
target_noise_clip	0.18538	0.10527	0.18672	0.14197
exploration_fraction	0.12082	0.10871	0.16125	0.16269
exploration_final_eps	0.18574	0.10366	0.10618	0.19131
exploration_initial_eps	0.16609	0.10003	0.16434	0.1039
test/total_reward	**1.97214**	**1.97061**	**1.62093**	**1.63348**

Table 4.2: Best and worst model and their training hyperparameters. All hyperparameters are also available online URL: `https://wandb.ai/investai/portfolio-allocation` with much more details and experiments.

4.4 Datasets Impact

In this section, we will assess the models based on the *datasets* they were trained on. Specifically, we will show and evaluate the **6 best** and **6 worst** agents/models according to the specific datasets (*Fundamental*, *Technical*, and *Combined*). The outcomes of this comparison are presented in table 4.3. Although the results in table 4.3 may vary depending on the hyperparameters, we can still observe that the *Technical Dataset* is the worst, while the *Combined Dataset* is the best, and the *Fundamental Dataset* is good enough for training the agent.

Our assumption that the **Fundamental Dataset** would be sufficient for training a profitable agent has been confirmed, although the **Combined Dataset** can yield better

results despite being more complex and difficult to train. As expected, the **Technical Dataset** performed the worst, as technical indicators are not as reliable as fundamental indicators for assessing the health and future growth of a company. Since the *Technical Analysis* is based on the assumption that the past price movements can predict future price movements, it is not surprising that the *Technical Dataset* performed the worst and it shows that the market history does not guarantee the same results in the future.

Model ID	Dataset Type	test/total_reward
p3irnh80	Combined	**1.972**
8tml2ozg	Combined	**1.970**
geaioz9h	Fundamental	**1.965**
8iq9e37s	Combined	**1.956**
y3zz2sv3	Combined	**1.955**
4qr3nk43	Fundamental	**1.945**
zfjr0ks0	Fundamental	**1.620**
pky1wslb	Combined	**1.633**
2161deh4	Technical	**1.642**
ipl1v8io	Technical	**1.643**
2161deh4	Technical	**1.642**
2161deh4	Technical	**1.642**

Table 4.3: Examples of 6 best models in the upper part and 6 worst models in the lower part of the table.

4.5 Testing of Robustness

In this section, we will examine whether the hyperparameters setup, which seems the best (*p3irnh80*) is robust or not by conducting multiple tests on the models and training them with the same hyperparameters. We trained **11 models** using the same hyperparameters but different *seeds*, and tested them all on the *Combined Dataset*. The results are summarized in table 4.4, which shows the *test/total_reward* for each of the 11 models. The mean *test/total_reward* across the models is **1.824**, indicating that the models perform well on the test dataset. Although there is some variation in performance between the models, with a spread of **0.249** between the highest and lowest cumulative reward values (*test/total_reward*), the distinction is significant since it has the potential to greatly impact the long-term returns. Overall, these results suggest that the models are robust enough to be used in real-life scenarios. It is worth noting that the variation in performance between the models may be due to differences in the random seeds used during training, which can affect the initial conditions of the learning process.

Table 4.4: Robust Test: Trained models using hyperparameters from the best model trained using sweep. The mean is 1.824 and the spread between the highest and the lowest reward is 0.249.

Model ID	test/total_reward
kyr89ols	1.723
l75jybwn	1.769
axq1epdr	1.776
3hiqutvc	1.785
sey6enti	1.799
w1xo3vul	1.806
4y7nkqyj	1.807
mliee6kz	1.864
e5pzquaz	1.882
r92of0f3	1.887
p3irnh80	1.972

4.6 Baselines & Benchmarks

In this section, we compare the performance of the agent to several baselines, including well-known market indexes as: **The S&P 500 Index (GSPC)**,**The Dow Jones Industrial Average (DJI)**, **The Russell 2000 (RUT)**, **The NASDAQ Composite (IXIC)**. And in our comparison, we also incorporate investment strategies as **Minimum Variance**, **Maximum Sharpe Ratio** [6].

4.6.1 Comparing Portfolio Performance: Analysis from 2017 to 2022

In fig. 4.1, we demonstrate that an agent can achieve successful outcomes when trained with appropriate hyperparameters and dataset (model id: *zfjr0ks0*), while unsuitable hyperparameters and dataset can result in unsatisfactory outcomes (model id: *p3irnh80*). The *zfjr0ks0* model surpassed standard indexes such as *DJI*, *GSPC*, *IXIC*, and *RUT*, although not consistently throughout the testing period. It is evident that the *zfjr0ks0* model outperformed almost all baseline indexes and strategies, with the exception of *IXIC (NASDAQ Composite)* during a specific period from 2020 until the end of the testing period in 2022, when our model began to outperform even *IXIC*. There could be various reasons why our agent was outperformed by the Nasdaq 100 during the testing period. However, we believe that the primary reason is that our *agent* is designed to operate with only **30 stocks**, while the *NASDAQ Comosite* comprises **100 stocks**. This limited scope of stocks may have resulted in fewer investment opportunities for our agent, which could have led to lower returns compared to the Nasdaq 100. Nonetheless, it is worth noting that our model did eventually outperform the Nasdaq 100 Index by the end of 2023.

Currently, we lack adequate data to conduct a thorough comparison between our developed model and the State-of-the-Art AI4Finance. However, in the following subsection where we will have the necessary data, we intend to perform a comprehensive comparison.

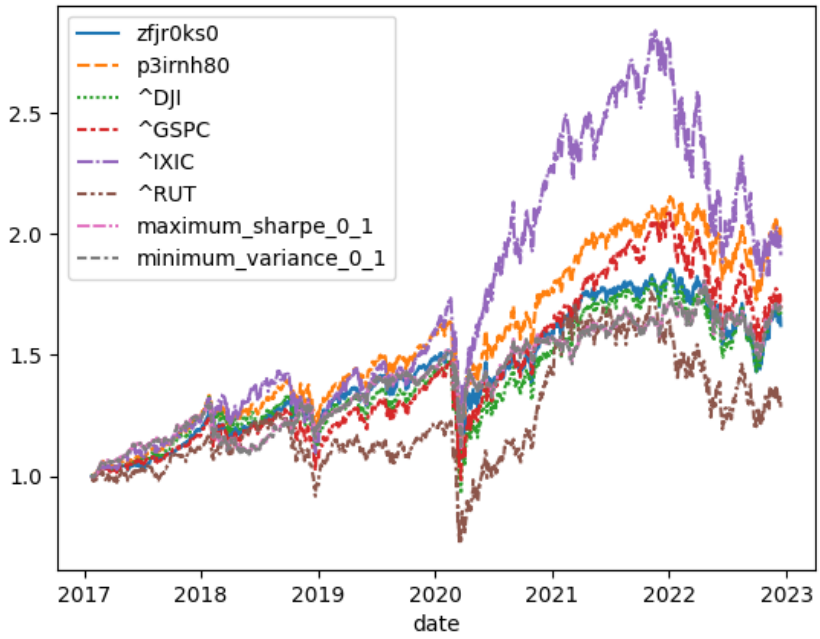
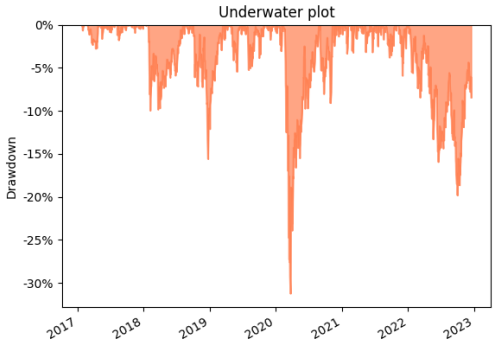


Figure 4.1: Cumulative returns of the best(zfjr0ks0) and worst(p3irnh80) performing models, indexes (DJI, GSPC, IXIC, RUT), and strategies (minimum variance and maximum Sharpe ratio), during the testing period (2017-2022).

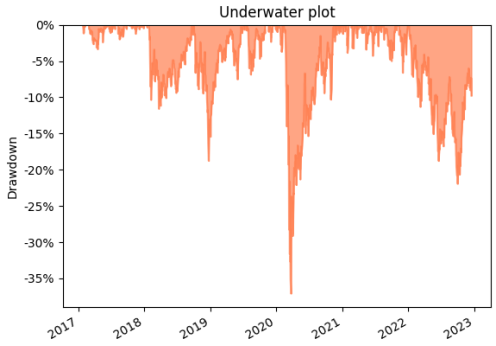
4.6.2 Drawdowns Analysis from 2017 to 2022

Now we look at how the portfolios performing in drawdowns[1] Based on the drawdown graphs in fig. 4.2, we observe that the best model, trained on appropriate datasets and hyperparameters, outperforms the DJI index for larger drawdowns (around 6%). Since our dataset comprises companies included in the DJI index we decide for our analysis to compare the drawdowns of the model with the *highest and lowest test/total_reward*, as well as *two indexes* (*IXIC* and *DJI*). The *IXIC* (Nasdaq 100 Index) has been chosen as the baseline for performance in drawdowns since it outperformed our model during the period between 2020 and 2022.

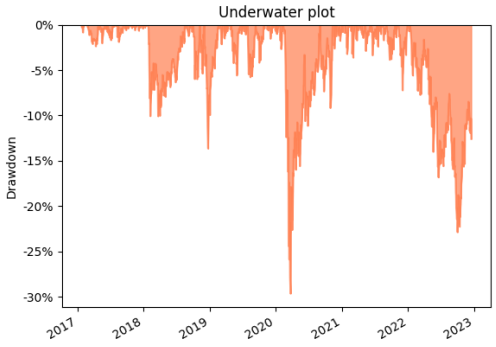
[1]In the stock market, drawdown refers to the percentage decline in an investment's value from its peak to its subsequent low point. Essentially, it represents the loss experienced by an investor in a particular investment over a certain period of time.



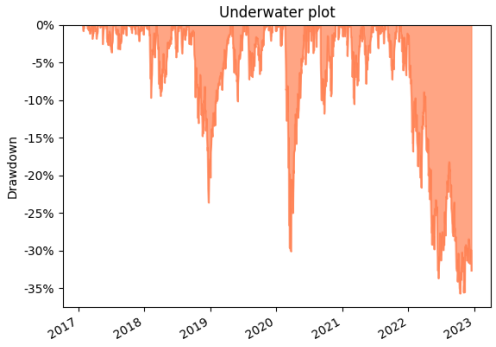
(a) The model with the highest *test/total_reward*



(b) The Dow Jones Industrial Average (DJI)



(c) The model with the lowest *test/total_reward*



(d) The Nasdaq 100 Index (IXIC)

Figure 4.2: Drawdown analysis of the best and worst performing models, and indexes (DJI and IXIC).

4.6.3 Performance of Different Portfolios: Monthly and Annualy

In figs. 4.3 to 4.5, we compared our models against baseline indexes and strategies based on their *monthly* and *annual* returns. We presented the results in a graph, where the model with the highest return was compared to the *DJI* and *IXIC* indexes. Our model consistently outperformed the DJI index in most months and years, but was outperformed by the IXIC index in 2019, 2020, and 2021. However, in 2022, when the IXIC index lost more than **-30%** of its value, our model was able to keep the majority of its value and only lost less than **-8%**.

It's worth noting that our models achieved significant outperformance during the big drawdown period caused by the COVID-19 pandemic. The our model performed better than the DJI index during February, March, and April of 2020, indicating that our model was able to learn which companies with specific features were better to hold during this period.

Although there were some months where the DJI index performed better than our models, the performance difference was not significant. Overall, we concluded that our models performed well in the majority of months, as demonstrated by the *annual graphs*.

It's important to note that our model was trained on companies from the DJI index, which may have influenced the comparison with the IXIC index. Therefore, the comparison may not be entirely fair. Nonetheless, the results suggest that our model is able to learn what companies to hold during certain time periods and can outperform baseline indexes.

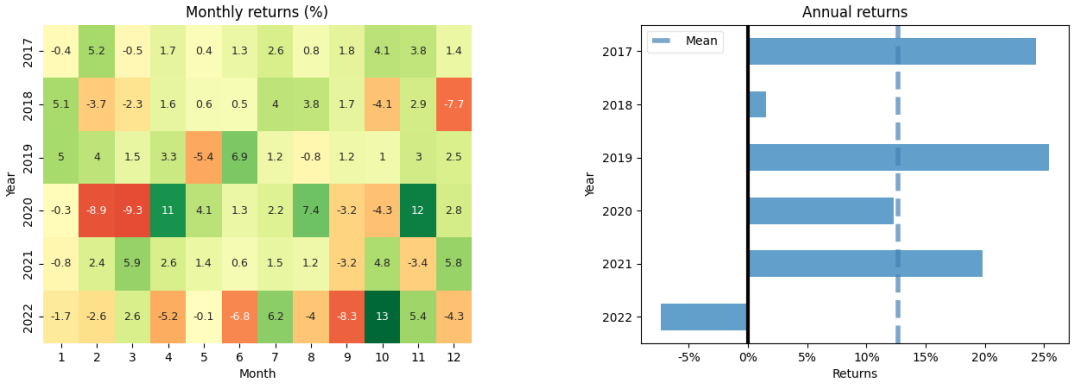


Figure 4.3: The monthly and annual returns of the model with the highest *test/total_reward*

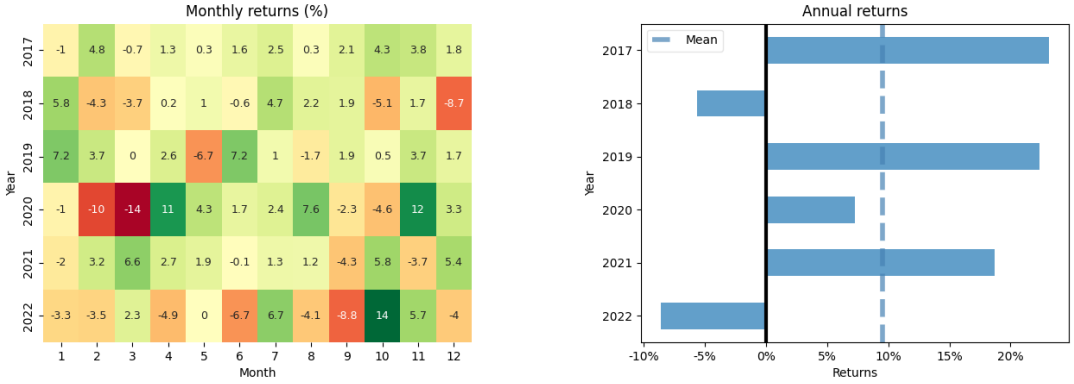


Figure 4.4: The monthly and annual returns of DJI Index

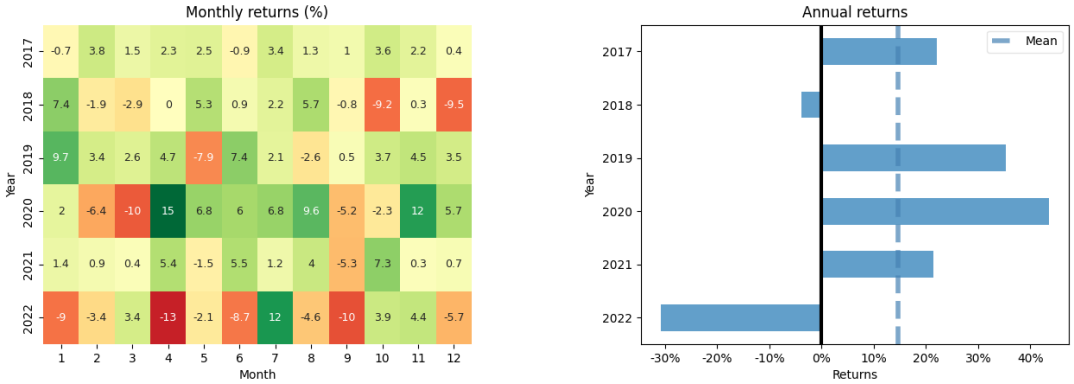


Figure 4.5: The monthly and annual returns of IXIC Index

4.7 Portfolio Metrics

This section presents the *Portfolio Metrics Comparison* of the models developed in this thesis, compared to various indexes, strategies, and a model developed by AI4Finance [4].

We do not provide an explanation of the metrics used to compare models because they are more related to Economic/Finance topics. However, we have referenced the Pyfolio website, where these metrics are fully described, for readers who seek additional information on the topic. The performance metrics were calculated using the Pyfolio framework and readers can find more information on the metrics by referring to [5, 16].

4.7.1 Comparison with State-of-the-Art AI4Finance

The table 4.5 compares our models with the model from AI4Finance [4]. Due to the lack of data, we used the results from the published paper and comparison was made only for the period from **2020-07-01** to **2021-10-29**.

As shown in the table 4.5, our models are not the best compared to the AI4Finance model. The AI4Finance model outperforms our models in almost all performance metrics, but our models are still better than the indexes and common strategies. Our models were outperformed by approximately 3.1% annually, which is quite significant. This could be due to various reasons, such as using different hyperparameters or training the models on different datasets. Our models were trained on a combined dataset of technical and fundamental analysis, while the AI4Finance model was trained only on technical analysis but included the covariance matrix between stocks, which we did not consider in our datasets. Of course it would be interesting to compare our models with the AI4Finance model over a longer period, but unfortunately, we do not have the data for that.

Metric / Model ID	zfjr0ks0	p3irnh80	AI4Finance
Annual return	0.213	0.265	**0.296**
Cum. returns	0.295	0.369	**0.415**
Annual volatility	0.128	0.130	**0.140**
Sharpe ratio	1.573	1.866	**1.930**
Calmar ratio	2.323	2.911	**3.350**
Stability	0.893	**0.931**	0.920
Max drawdown	-0.092	-0.091	**-0.088**
Omega ratio	1.299	1.367	**1.380**
Sortino ratio	2.349	2.792	**2.940**
Skew	-0.127	-0.228	**-0.110**
Kurtosis	1.574	**1.580**	1.450
Tail ratio	**1.110**	1.028	1.100
Daily value at risk	-0.015	**-0.014**	-0.017
Beta	0.888	0.920	**0.980**
Alpha	-0.025	0.009	**0.020**

Table 4.5: Performance metrics of the models vs. AI4Finance model, during the testing period of 2020-07-01 to 2021-10-29.

4.7.2 Comparison with Indexes and Strategies

The table 4.6 shows the comparison of our models with the S&P 500 (GSPC), NASDAQ Composite (IXIC), Dow Jones Industrial Average (DJI), Russell 2000 (RUT), Maximum Sharpe Ratio strategy, and Minimum Variance strategy. The comparison is based on the testing period from **2017-01-25** to **2022-12-16**.

As seen in table 4.6, our models outperform most of the other models/strategies in almost all performance metrics. The **green bold values** indicate the best results for each metric. However, the best values for a portfolio can vary depending on the desired outcome and other factors.

Metric / Portfolio maintainer	zfjr0ks0 (model)	p3irnh80 (model)	DJI Index	GSPC Index	IXIC Index	RUT Index	Max Sharpe Ratio Portfolio	Min Variance Portfolio
Annual return	0.085	**0.122**	0.089	0.094	0.116	0.043	0.093	0.092
Cum. returns	0.621	**0.972**	0.654	0.695	0.911	0.284	0.686	0.683
Annual volatility	0.181	0.188	0.202	0.203	0.239	**0.253**	0.158	0.159
Sharpe ratio	0.543	**0.706**	0.525	0.544	0.581	0.295	0.641	0.640
Calmar ratio	0.288	**0.391**	0.241	0.276	0.325	0.101	0.350	0.349
Stability	0.855	**0.913**	0.798	0.845	0.820	0.461	0.905	0.906
Max drawdown	-0.297	-0.312	-0.371	-0.339	-0.357	-0.431	-0.265	**-0.264**
Omega ratio	1.118	**1.158**	1.114	1.115	1.116	1.057	1.136	1.135
Sortino ratio	0.763	**0.994**	0.723	0.751	0.802	0.404	0.907	0.904
Skew	-0.249	-0.312	-0.593	-0.549	-0.465	-0.782	-0.246	**-0.240**
Kurtosis	16.819	19.569	**19.640**	14.205	7.418	10.450	14.315	14.286
Tail ratio	0.883	0.884	0.891	0.859	0.862	**0.984**	0.937	0.930
Daily value at risk	-0.022	-0.023	-0.025	-0.024	-0.030	-0.032	-0.019	**-0.018**
Beta	0.879	0.921	1.000	0.966	1.007	**1.083**	0.636	0.637
Alpha	0.005	**0.036**	0.000	0.008	0.032	-0.039	0.034	0.035

Table 4.6: Performance metrics of the models vs. indexes and strategies, during the testing period of 2017-01-25 to 2022-12-15.

4.8 Summary

In this chapter, we conducted empirical experiments to evaluate how well a Deep Reinforcement Learning (DRL) agent can perform in the portfolio management task. Our results show that DRL has a huge potential in portfolio allocation as it can outperform common strategies and indexes. We used DRL algorithms to explore the relationship between the reward (i.e., portfolio return) and the input (i.e., features) and measured the prediction power using portfolio metrics. We found that the Fundamental Features or a combination

of Fundamental and Technical Analysis features can effectively describe the environment and be used to train DRL agents.

Portfolio management is a challenging task that requires a lot of data, which is often expensive and difficult to obtain. Our Stock Portfolio Allocation Environment, modeled using MDP, aims to identify how stocks are valued and which State Space features affect price movements. We assume that the Financial Market operates like any other market where the price of a stock or asset increases when people buy it and decreases when people sell it. Therefore, our model attempts to predict how much each feature affects future price movements (people's interest to buy/sell). Since the Stock Market is predominantly controlled by "Big Money" players such as banks and hedge funds, we believe that our research is crucial for portfolio managers to improve their decision-making capabilities.

In addition to the future work mentioned earlier, we see a great opportunity to extend our model by incorporating a system that automatically places orders to the broker based on the weight predicted by our model. This would streamline the entire process of portfolio management and eliminate the need for manual intervention.

Furthermore, we believe that our model can benefit from the inclusion of additional features such as sentiment analysis, macroeconomic factors, and industry-specific factors. The inclusion of these features can help our model to better capture the market's behavior and improve its performance.

Another area of improvement that we believe warrants further exploration is the reward function. While our experiments have shown that the reward function we used was effective to some extent, we acknowledge that it may not capture all the nuances of the market. Therefore, we recommend exploring alternative reward functions that can better account for the goodness or badness of actions taken by the agent.

Overall, we believe that our study has demonstrated the potential of DRL in portfolio allocation, but there are still many avenues for improvement and further research. By incorporating these future works, we can enhance the performance of our model and make it more suitable for real-world applications.

Chapter 5

Conclusions

The aim of this thesis was to develop a profitable portfolio allocation strategy using deep reinforcement learning, and we successfully achieved this goal as outlined in Chapter 1.

One of the main contributions of this work is the creation of three datasets that describe the environment for the agent, as well as the implementation of a training and testing pipeline that is connected to the Weights and Biases platform. We were able to identify the most appropriate dataset for training the agent, which consists of a combination of fundamental and technical analysis features. We found that technical analysis alone is not sufficient to predict future stock prices, but it can serve as a useful indicator for the agent to learn the best strategy. In contrast, the fundamental analysis provides a good enough description of the agent's world to predict stock prices without any additional factors.

Furthermore, we independently configured hyperparameters for each common RL algorithm and dataset and made them publicly available through Weights and Biases. This transparency and reproducibility of our work make it easier for other researchers to replicate and improve upon our results.

Although our agent's performance could not match the annual returns of investors like Warren Buffett or Peter Lynch, who consistently achieve over 20%, we were able to generate an annual return of over 12%. Based on our model's returns, there is considerable room for improvement, which could enable us to surpass even these legendary investors.

We found that our agent could learn the best strategy even in the case of the COVID-19 pandemic and other major drops in the stock market. Specifically, the agent with the lowest cumulative return was more efficient during such events, while the agent with the highest cumulative return was more effective during draw-downs and skyrocketing markets.

Overall, our RL-based approach for portfolio allocation holds great potential, and we suggest future work to further enhance our results by incorporating additional factors such as sentiment analysis, macroeconomic factors, and industry-specific factors. These enhancements could enable us to train more accurate RL agents that surpass even the top investors in the field. It will be intriguing to observe how these factors can be implemented in real-time market scenarios to predict market trends and facilitate investment decisions.

Finally, we believe that our work can inspire further research in this area. In conclusion, our findings, along with the availability of data and computational resources, suggest that RL-based portfolio allocation is a promising direction for future investment research.

Bibliography

[1] Abate, A., Andriushchenko, R., Češka, M. and Kwiatkowska, M. Adaptive formal approximations of Markov chains. *Performance Evaluation*. 2021, vol. 148, p. 102207. DOI: https://doi.org/10.1016/j.peva.2021.102207. ISSN 0166-5316. Available at: https://www.sciencedirect.com/science/article/pii/S0166531621000249.

[2] Adamczyk, J., Makarenko, V., Arriojas, A., Tiomkin, S. and Kulkarni, R. V. *Bounding the Optimal Value Function in Compositional Reinforcement Learning*. 2023.

[3] Chan, K., Lenard, C. and Mills, T. An Introduction to Markov Chains. In:. December 2012. DOI: 10.13140/2.1.1833.8248.

[4] Guan, M. and Liu, X.-Y. *Explainable Deep Reinforcement Learning for Portfolio Management: An Empirical Approach*. arXiv, 2021. DOI: 10.48550/ARXIV.2111.03995. Available at: https://arxiv.org/abs/2111.03995.

[5] Inc., Q. *Pyfolio* [https://github.com/quantopian/pyfolio]. GitHub, 2020.

[6] Investopedia. *Investopedia* [online]. [cit. 2023-04-16]. Available at: https://www.investopedia.com.

[7] Kurniawati, H. *Partially Observable Markov Decision Processes (POMDPs) and Robotics*. 2021.

[8] Liu, D., Wei, Q. and Yan, P. Generalized Policy Iteration Adaptive Dynamic Programming for Discrete-Time Nonlinear Systems. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*. 2015, vol. 45, no. 12, p. 1577–1591. DOI: 10.1109/TSMC.2015.2417510.

[9] Liu, X.-Y., Xia, Z., Rui, J., Gao, J., Yang, H. et al. *FinRL-Meta: Market Environments and Benchmarks for Data-Driven Financial Reinforcement Learning*. 2022.

[10] Oshingbesan, A., Ajiboye, E., Kamashazi, P. and Mbaka, T. *Model-Free Reinforcement Learning for Asset Allocation*. arXiv, 2022. DOI: 10.48550/ARXIV.2209.10458. Available at: https://arxiv.org/abs/2209.10458.

[11] peerchemist. *Finta* [https://github.com/peerchemist/finta]. GitHub [cit. 2023-04-16].

[12] Perotto, F. S. and Vercouter, L. Tuning the Discount Factor in Order to Reach Average Optimality on Deterministic MDPs. In: *International Conference on Innovative Techniques and Applications of Artificial Intelligence*. Cambridge, United Kingdom: [b.n.], 2018. DOI: 10.1007/978-3-030-04191-5_7. Available at: https://hal.science/hal-02363599.

[13] Rao, A. and Jelvis, T. *Foundations of Reinforcement Learning with Applications in Finance*. 1stth ed. CRC Press, 2022. ISBN 9781000801101. Available at: https://books.google.cz/books?id=n_-VEAAAQBAJ.

[14] Silver, D. *Introduction to Reinforcement Learning with David Silver* [online]. 2015 [cit. 2023-04-08]. Available at: https://www.deepmind.com/learning-resources/introduction-to-reinforcement-learning-with-david-silver.

[15] Soeryana, E., Fadhlina, N., Firman, S., Rusyaman, E. and Supian, S. Mean-variance portfolio optimization by using time series approaches based on logarithmic utility function. *IOP Conference Series: Materials Science and Engineering*. january 2017, vol. 166, no. 1, p. 012003. DOI: 10.1088/1757-899X/166/1/012003.

[16] Sravz. *Pyfolio - Return Analysis* [online]. [cit. 2023-04-08]. Available at: https://docs.sravz.com/docs/analytics/pyfolio-returns-analysis/.

[17] Sutton, R. and Barto, A. *Reinforcement Learning, second edition: An Introduction*. 2ndth ed. MIT Press, 2018. Adaptive Computation and Machine Learning series. ISBN 9780262039246. Available at: https://books.google.cz/books?id=5s-MEAAAQBAJ.

[18] Vosol, D. *Aplikace posilovaného učení v řízení autonomního vozidla*. Brno, CZ, 2022. Master's thesis. Brno University of Technology, Faculty of Information Technology. Available at: https://www.fit.vut.cz/study/thesis/25127/.

[19] Wikipedia contributors. *Modern portfolio theory — Wikipedia, The Free Encyclopedia*. 2021. [Online; accessed 16-April-2023]. Available at: https://en.wikipedia.org/w/index.php?title=Modern_portfolio_theory&oldid=1043516653.

[20] Wikipedia contributors. *Dow Jones Industrial Average — Wikipedia, The Free Encyclopedia*. 2023. [Online; accessed 16-April-2023]. Available at: https://en.wikipedia.org/w/index.php?title=Dow_Jones_Industrial_Average&oldid=1141766585.

[21] Wikipedia contributors. *Stochastic process — Wikipedia, The Free Encyclopedia*. 2023. [Online; accessed 10-April-2023]. Available at: https://en.wikipedia.org/w/index.php?title=Stochastic_process&oldid=1148510872.

[22] Širůček, M. and Křen, L. Application of Markowitz Portfolio Theory by Building Optimal Portfolio on the US Stock Market. *Acta Universitatis Agriculturae et Silviculturae Mendelianae Brunensis*. Mendel University Press. september 2015, vol. 63, no. 4, p. 1375–1386. DOI: 10.11118/actaun201563041375. Available at: http://dx.doi.org/10.11118/actaun201563041375.

Appendix A

Reinforcement Learning Algorithms

In this appendix, we present pseudocode for various RL methods and algorithms, which are used in the field of RL research and applications.

A.1 Value Iteration Algorithm in DP

Value Iteration Algorithm
Algorithm parameter: a small threshold $\epsilon > 0$ determining accuracy of estimation
Initialize $V(s)$, for all $s \in S$, arbitrarily except that $V(\text{terminal}) = 0$
Loop:
  for each $s \in S$:
    $v \leftarrow V(s)$
    $V(s) \leftarrow \max_a \sum_{s',r} p(s', r|s, a)(r + \gamma V(s'))$
    $\delta \leftarrow \max(\delta, |v - V(s)|)$
until $\delta < \epsilon$
Output a deterministic policy π, such that
$\pi(s) \leftarrow \arg\max_a \sum_{s',r} p(s', r|s, a)(r + \gamma V(s'))$

A.2 Q-Learning Off-policy TD Control

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\epsilon > 0$
Initialize $Q(s, a)$ for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$ arbitrarily, except that $Q(\text{terminal}, \cdot) = 0$
Loop for each episode: for *each episode* **do**
  Initialize S **Loop for each step of episode: while** S *is not terminal* **do**
    Choose A from S using policy derived from Q (e.g., ϵ-greedy) Take action A, observe R, S' $Q(S, A) \leftarrow Q(S, A) + \alpha \cdot [R + \max_a Q(S', a) - Q(S, A)]$
    $S \leftarrow S'$
  end
end

Algorithm 1: Q-learning (Off-policy TD Control)

A.3 REINFORCE: Monte-Carlo Policy-Gradient Control

`/* Input:` `*/`
a differentiable policy parameterization $\pi(a|s, \boldsymbol{\theta})$;
a differentiable state-value parameterization $\hat{v}(a|\boldsymbol{w})$;
Algorithm parameters: step size $\alpha_{\boldsymbol{\theta}} > 0$, $\alpha_{\boldsymbol{w}} > 0$;
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ and state-value weights $\boldsymbol{w} \in \mathbb{R}^{d}$ arbitrarily (e.g., $\boldsymbol{\theta} = \mathbf{0}, \boldsymbol{w} = \mathbf{0}$);
while *Loop forever (for each episode):* **do**
  Generate an episode $S_0, A_0, R_1, \ldots, S_T - 1, A_T - 1, R_T$, following $\pi(\cdot|\cdot, \theta)$;
  for *Loop for each step of the episode* $t = 0, 1, \ldots, T - 1$*:* **do**
    $G_t \leftarrow \sum_{k=t+1}^{T} R_k$;
    $\delta \leftarrow G - \hat{v}(S_t, \boldsymbol{w})$;
    $\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha^{\boldsymbol{w}} \delta \nabla \hat{v}(S_t, \boldsymbol{w})$;
    $\theta \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} \gamma^t \delta \nabla \ln \pi(A_t|S_t, \boldsymbol{\theta})$;
  end
end

Algorithm 2: REINFORCE Algorithm

Appendix B

Setting up and running the program

The root directory of the implementation code contains a `README.md` file that provides instructions for installing the program. Following the steps outlined in this file should be sufficient for completing the installation process.

The program has been tested on both MacOS 13.2.1 and Ubuntu 20.04.6 LTS (GNU/Linux 5.4.0-146-generic x86_64), using Python version 3.10.11.

B.1 Prepare Environment

Run these commands from the root directory of the implementation code. It will create a virtual environment and install the required packages:

```
mkdir -p out/baseline out/dataset out/model # create directories for results
python3 -m venv venv # create virtual environment
source venv/bin/activate # activate virtual environment
pip3 install -r requirements.txt # install required packages
```

B.2 Datasets

Datasets are available at the following links:

1. `https://wandb.ai/investai/portfolio-allocation/artifacts/dataset/stockcombineddailydataset/v0/files`
2. `https://wandb.ai/investai/portfolio-allocation/artifacts/dataset/stockfadailydataset/v2/files`
3. `https://wandb.ai/investai/portfolio-allocation/artifacts/dataset/stockfadailydataset/v0/files`.

Please download them and place them in the `out/dataset` folder.

B.3 Baseline

The baseline is available at the following link:

1. `https://wandb.ai/investai/portfolio-allocation/artifacts/baseline/baseline/v1/files`

Please download it and place it in the `out/baseline` folder.

B.4 Examples of running Train/Test program

All examples assume that the current working directory is the root directory of the implementation code.

B.4.1 `.env` file

The scripts expect the *.env* file in the root directory of the implementation code. This file contains the following variables, please fill the `WANDB_API_KEY` with your API key:

```
# W&B
WANDB_API_KEY=''
WANDB_ENTITY='investai'
WANDB_PROJECT='portfolio-allocation'
WANDB_TAGS='["None"]'
WANDB_JOB_TYPE='train'
WANDB_RUN_GROUP='exp-1'
WANDB_MODE='online'
WANDB_DIR='${PWD}/out/model'
```

If you don't want to use Weights & Biases, you can remove the arguments with prefix `-wandb` from the examples (sweep run will not work), if you want to use it, it will be necessary to run the following command before running any of the following commands:

```
wandb login # to login to your W&B account
```

B.4.2 Run the program to print the help message

```
PYTHONPATH=$PWD/investai python3 \
    investai/run/portfolio_allocation/thesis/train.py \
       --help
```

B.4.3 Single Run (train/test)

```
PYTHONPATH=$PWD/investai python3 \
    investai/run/portfolio_allocation/thesis/train.py \
    --dataset-paths out/dataset/stockfadailydataset.csv \
    --algorithms ppo \
    --project-verbose='i' \
    --train-verbose=1 \
    --total-timesteps=1000 \
    --train=1 \
    --test=1 \
    --env-id=1 \
```

```
    --wandb=1 \
    --wandb-run-group="exp-run-1" \
    --wandb-verbose=1 \
    --baseline-path=out/baseline/baseline.csv
```

B.4.4 Sweep Run: 3 runs with random hyperparameters over 2 datasets and 5 algorithms (train/test)

```
PYTHONPATH=$PWD/investai python3 \
    investai/run/portfolio_allocation/thesis/train.py \
    --dataset-paths \
        out/dataset/stockfadailydataset.csv \
        out/dataset/stockcombineddailydataset.csv \
    --algorithms \
        ppo \
        a2c \
        td3 \
        ddpg \
        sac \
    --project-verbose='i' \
    --train-verbose=1 \
    --total-timesteps=1000 \
    --train=1 \
    --test=1 \
    --env-id=1 \
    --wandb=1 \
    --wandb-sweep=1 \
    --wandb-sweep-count=3 \
    --wandb-verbose=1 \
    --wandb-run-group="exp-sweep-1" \
    --baseline-path=out/baseline/baseline.csv
```

Appendix C

Graphs

C.1 Hyperparameters Tuning Results

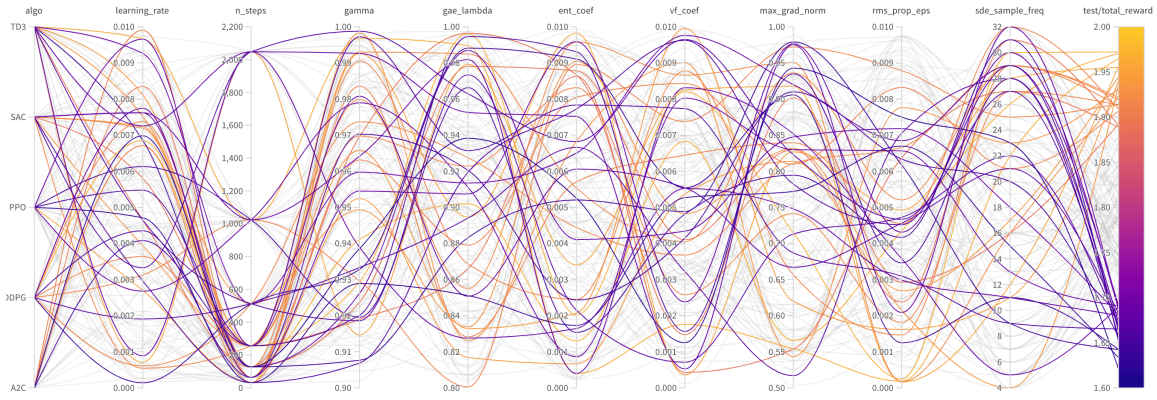


Figure C.1: W&B Chart of parameters and their performance

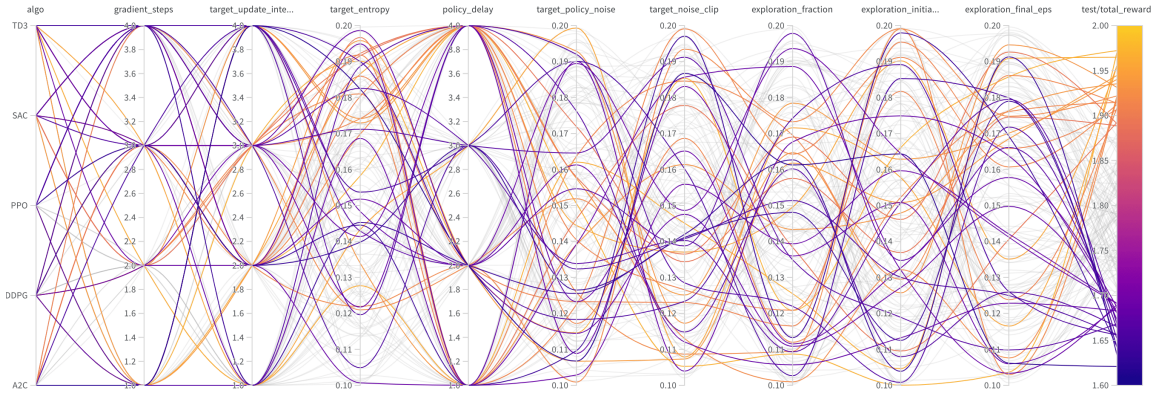


Figure C.2: W&B Chart of parameters and their performance

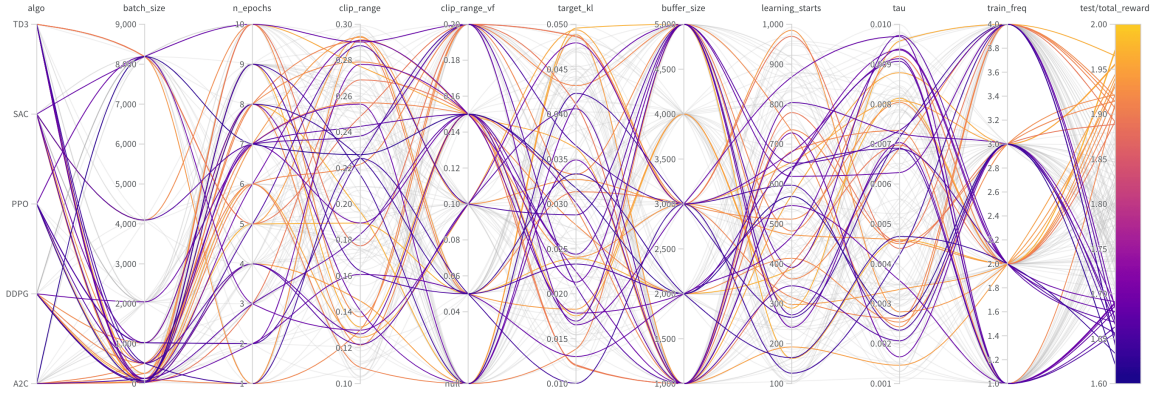


Figure C.3: W&B Chart of parameters and their performance