DEEP REINFORCEMENT LEARNING APPROACH FOR TRADING AUTOMATION IN THE STOCK MARKET

A Thesis

by

Taylan KABBANI

Submitted to the
Graduate School of Sciences and Engineering
In Partial Fulfillment of the Requirements for
the Degree of

Master of Science

in the
Data Science

Özyeğin University
December, 2021

DEEP REINFORCEMENT LEARNING APPROACH FOR TRADING AUTOMATION IN THE STOCK MARKET

Approved by:

Prof. Dr. Ekrem Duman, Advisor
Department of Industrial Engineering
Özyeğin University

Prof. Dr. Ali Fuat Alkaya
Department of Computer Engineering
Marmara University

Asst. Prof. Dr. Erinç ALBEY
Department of Industrial Engineering
Özyeğin University

Date Approved: 28 December 2021

To My Parents

ABSTRACT

Deep Reinforcement Learning (DRL) algorithms can scale to previously intractable problems. The automation of profit generation in the stock market is possible using DRL, by combining the financial assets price "prediction" step and the "allocation" step of the portfolio in one unified process to produce fully autonomous system capable of interacting with its environment to make optimal decisions through trial and error. In this study, a continuous action space approach is adopted to give the trading agent the ability to gradually adjust the portfolio's positions with each time step (dynamically re-allocate investments), resulting in better agent-environment interaction and faster convergence of the learning process. In addition, the approach supports the managing of a portfolio with several assets instead of a single one. This work represents a novel DRL model to generate profitable trades in the stock market, effectively overcoming the limitations of supervised learning approaches. We formulate the trading problem as Partially Observed Markov Decision Process (POMDP) model, considering the constraints imposed by the stock market, such as liquidity and transaction costs. More specifically, we design an environment that simulates the real-world trading process by augmenting the state representation with ten different technical indicators and sentiment analysis of news articles for each stock. We then solve the formulated POMDP problem using the Twin Delayed Deep Deterministic Policy Gradient (TD3) algorithm and achieved a 2.68 Sharpe ratio on the test dataset. From the point of view of stock market forecasting and the intelligent decision-making mechanism, this study demonstrates the superiority of deep reinforcement learning in financial markets over other types of machine learning such as supervised learning and proves the credibility and advantages of strategic decision-making using DRL.

ÖZETÇE

Deep Reinforcement Learning (DRL) algoritmaları, daha önceki zorlu sorunlara ölçeklenebilir. Hisse senedi piyasasında kâr üretiminin otomasyonu, finansal varlıkların fiyat "tahmin" adımı ve portföyün "tahsis" adımını tek bir birleşik süreçte birleştirerek, deneme yanılma yoluyla optimal kararlar almak için çevresiyle etkileşime girebilen tamamen özerk bir sistem üretmek için DRL kullanılarak mümkündür. Bu çalışmada, alım-satım aracısına portföyün pozisyonlarını her zaman adımı için kademeli olarak ayarlama (yatırımları dinamik olarak yeniden tahsis etme) özelliği vermek için sürekli bir eylem alanı yaklaşımı benimsenmiştir, bu da daha iyi aracı-ortam etkileşimi ve öğrenme sürecinin daha hızlı yakınsaması ile sonuçlanmıştır. Ayrıca yaklaşım, tek bir varlık yerine birkaç varlık içeren bir portföyün yönetilmesini destekler. Bu çalışma, denetimli öğrenme yaklaşımlarının sınırlamalarını etkin bir şekilde aşarak borsada karlı işlemler oluşturmak için yeni bir DRL modelini temsil etmektedir. Ticaret problemini, likidite ve işlem maliyetleri gibi hisse senedi piyasasının getirdiği kısıtlamaları dikkate alarak Partially Observed Markov Decision Process (POMDP) modeli olarak formüle ediyoruz. Daha spesifik olarak, her hisse senedi için on farklı teknik gösterge ve haber makalelerinin duygu analizi ile durum temsilini artırarak gerçek dünyadaki ticaret sürecini simüle eden bir ortam tasarlıyoruz. Daha sonra formüle edilmiş POMDP problemini Twin Delayed Deep Deterministic Policy Gradient (TD3) algoritmasını kullanarak çözdük ve test veri setinde 2.68 Sharpe oranı elde ettik. Hisse senedi piyasası tahmini ve akıllı karar verme mekanizması açısından bu makale, finansal piyasalarda derin pekiştirmeli öğrenmenin denetimli öğrenme gibi diğer makine öğrenimi türlerine göre üstünlüğünü göstermekte, aynı zamanda DRL yaklaşımının güvenilirliğini ve stratejik karar vermenin avantajlarını kanıtlamaktadır.

ACKNOWLEDGEMENTS

I am overwhelmed in all humbleness and gratefulness to acknowledge my depth to all those who have helped me put these ideas well above the level of simplicity and into something concrete.

I want to express my special thanks to my research supervisor Prof. Ekrem Duman for his continuous support during my research and my deep and sincere gratitude to Özyeğin University for offering me this opportunity to continue my research fully scholarship after the sudden closure of my previous institution.

Any attempt at any level can not be realized without the support of friends. So I want to thank my friends, colleagues, and my flatmate for their support and understanding during this period of my life.

Lastly, my family deserves endless gratitude: my parents, who motivated me and provided me with all kinds of support not only during this phase but during my whole life, my brother for putting a smile on my face in my darkest moments, my sister for her helpful guidance and support. To my family, I give everything, including this.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER I

INTRODUCTION

1.1 Justification of the Study

Financial markets are at the heart of the modern economy. They provide an avenue for the sale and purchase of assets such as bonds, stocks, foreign exchange, and derivatives; however, their complicated environment should be well studied to profit from such markets. The prime objective of any investor when investing in any of these markets is to minimize the risk involved in the trading process and maximize the profits generated. This objective can be met by successfully predicting the prices or trends of the assets in the market and optimally allocating the capital among the selected assets. This process is very challenging for a human to consider all relevant factors in a complex and dynamic environment; therefore, the design of adaptive automated trading systems capable of meeting the investor's objective and bringing more stagnant wealth into the global market has been an intensive research topic. Many efforts have been made in the past decade to design such trading systems, the majority of these efforts were focusing on using *Supervised learning* (SL) techniques, which in essence train a predictive model (e.g., Neural Network, Random Forest,...) on historical data to forecast the trend direction of the market, regardless of their popularity these techniques suffered from various limitations which led to sub-optimal results [2].

One of the significant drawbacks of SL methods is that they only consider one aspect of the trading problem: predicting future prices and neglecting many other aspects and factors that play a crucial role in successfully generating profitable trades. One of these aspects is the control problem of optimally diversifying the capital to be

invested among the selected assets in the portfolio in a way that maximizes profits and minimizes risks. This is a well-known problem known as the *Portfolio Optimization Problem* (POP) [3] or Portfolio Selection Problem. Another major drawback of the methods that rely on Supervised learning is that they seek minimization of prediction error regardless of the risk involved, which is not in the investor's interest. Also, exogenous constraints by the market environment (e.g., lack of liquidity, transition costs) are not being considered at all in most cases [4]. In addition, financial markets data is extremely noisy, hence employing an algorithm with enormous learning capacity such as a deep learning algorithm in such an environment will often lead to overfitting [5].

Reinforcement Learning (RL) offers to solve the aforementioned drawbacks of Supervised Learning approaches in trading financial markets by combining the financial assets price "prediction" step and the "allocation" step of the portfolio in one unified process to optimize the objective of the investor, where the trading agent (the algorithm) interacts with the environment (the model) to take the optimal decision [4]. In addition, financial data is highly time-dependent (function of time), making it a perfect fit for Markov Decision Processes (MDP), which is the core process of solving RL problems. MDP captures the entire past data and defines the entire history of the problem in just the agent's current state, and that is highly crucial when it comes to modeling financial market data. For instance, a company whose annual report has just been released and has suffered heavy losses, however, the history of the company was good, a supervised learning algorithm would predict a good performance based on the excellent history ignoring the current released report. RL, on the contrary, the decision made by the agent will be heavily impacted, no matter the history or goodwill of the company this news will negatively impact the transition from one state to another [6].

1.2 Objectives

The main objective of this thesis is to design a Reinforcement Learning approach for successful trading automation in the stock market. We specifically deploy *Deep Reinforcement Learning* (DRL), which is a subfield of RL that incorporates deep learning into the RL framework, allowing us to solve problems with enormous state and action spaces such as the stock trading problem. Most works that studied the RL's applications in financial markets and particularly in trading stocks considered discrete action spaces, i.e., buy, hold, and sell a fixed number of shares for a single asset. In contrast, in this work, a continuous action space approach is adopted to give the trading agent the ability to gradually adjust the portfolio's positions with each time step (dynamically re-allocate investments), resulting in better agent-environment interaction and faster convergence of the learning process. In addition, the approach supports the managing of a portfolio with several assets instead of a single one.

We first propose a novel formulation of the problem or what is referred to as the *environment* as a Partially Observed Markov Decision Process (POMDP) model considering the constraints imposed by the stock market, such as liquidity and transaction costs constraints. More specifically, we design an environment that simulates the real-world trading process by augmenting the state (observation) representation with ten different technical indicators and sentiment analysis scores of news releases along with other state components. We then solve the formulated POMDP problem using the Twin Delayed Deep Deterministic Policy Gradient (TD3) algorithm, which can learn policies in high-dimensional and continuous action spaces like those typically found in the stock market environment. From the point of view of stock market forecasting and the intelligent decision-making mechanism, this thesis demonstrates the superiority of deep reinforcement learning in financial markets over other types of machine learning such as supervised learning and proves its credibility and advantages of strategic decision-making.

Another objective of the thesis is to provide a clear and concise introduction to the Reinforcement Learning field, with an easy-to-understand guide combined with illustrative examples and pseudocodes.

1.3 Overview

The thesis is structured in 5 main chapters that we briefly describe as follow:

- Chapter 2 introduces the basic concepts and mathematical formalism of Reinforcement learning along with the most famous algorithms used in the field.
- Chapter 3 provides a survey of the related work and research in Reinforcement Learning in general and in the applications of Reinforcement Learning in the financial market in particular.
- Chapter 4 discusses how the problem is being formulated as Markov Decision Process and shows the approaches and methods we followed to implement the environment and the agent.
- In Chapter 5 we evaluate our approach and compare to different benchmarks. We discuss and show the results through illustrative plots and tables.
- Finally in Chapter 6 we summarize and conclude the results and present some immediate extensions of our work to enhance it possibly.
- The Environment, agent and all experimental code is available on GitHub: `https://github.com/taylankabbani/StockDRL.git`

CHAPTER II

INTRODUCTION TO REINFORCEMENT LEARNING

The primary goal of AI is to yield fully autonomous systems capable of interacting with their environments to mimic the learning process of humans. The way we learn how to drive a car or how to make an optimal decision in a specific situation is through a trial and error sequence ensued from many past interactions with our environment. These past interactions only provide us with an awareness of how our environment might respond to our actions. Therefore we seek to influence the outcome of the surrounding environment by controlling our actions; this is the foundational idea underlying nearly all theories of learning and intelligence. *Reinforcement learning* (RL), a sub-field of *Machine learning*, is capable of delivering such agents that can learn from their environments through interaction. In RL, the agent's environment is designed by setting a bunch of rules and rewards for possible actions, so the agent will learn what to do (map situations into actions) by trying different actions in different situations with an optimal objective which is maximizing rewards.
On the other hand, the other sub-fields of Machine learning [7], i.e., *Supervised and Unsupervised learning*, are merely statistic models which do not involve any intelligent systems, for instance, a *Neural Network* is nothing more than a sophisticated version of *Logistic regression* which maps input to output by finding better representations of the historical data.

Unlike Supervised learning, RL does not learn on historical labeled data provided by a knowledgeable external supervisor but instead learns from past experiences generated by the agent's interactions with its environment. Hence RL can learn to make

decisions that have never been thought of or tried before by humans, and it is capable of learning in real-time interacting with the world where the system dynamics are unknown. A practical example is a self-driving car agent; by letting the car drive for hundreds of thousands of hours (or we can let the car drive more than the whole of humanity has ever driven) in a real-world simulation environment, we might find new and interesting ways of driving a car and do better than people do, as in *Alpha Go*[1] which is one of the most astonishing events related to RL, where an agent of RL found a new way to play the game of *Go*[2] and defeated the world's best player.

This chapter represents our research in the RL field to provide the reader with an easy-to-understand guide with the generous use of illustrative examples. At the beginning of the chapter, we explain the key concepts used, such as value function, policy, reward,.etc. Furthermore, how the RL problem is formalized. After building the foundations of RL, we discuss the different methodologies and algorithms used to solve the RL problem by classifying these algorithms based on their shared concepts and ideas. It is worth mentioning that it is quite challenging to encompass all methods and algorithms used in the field, and it is even more challenging to classify and build a taxonomy of algorithms in the modern RL space because the modularity of algorithms is not well-represented by a tree structure [8]. That is being said, the most crucial broad line separating RL methods is that between *model-free* and *model-based*(general name of Dynamic Programming) algorithms. Model-based methods rely on *planning* as their main component, while model-free methods primarily rely on *learning*.

[1]Computer program that plays the board game Go. It was developed by DeepMind Technologies, which Google later acquired.

[2]A board game originated in China 5000 years ago with simple rules but with 10^{170} possible board configurations, and it is considered much more complex than Chess.

2.1 Finite Markov Decision Processes in Reinforcement Learning

In essence, *Markov Decision Processes* [9] (MDP) is used to model stochastic processes containing random variables, transitioning from one state to another depending on certain assumptions and definite probabilistic rules. *The Markov Property* states that the calculated probability of a random process transitioning to the next possible state is only dependent on the current state and time. Therefore, for a system having this property, the future can be predicted through the present only with no need for the past[10].

MDPs are a perfect mathematical framework to describe the reinforcement learning problem. In this framework, researchers call the learner or decision maker the *agent* and the surrounding which the agent interacts with (comprising everything outside the agent) the *environment*. The learning process ensues from the agent-environment interaction in MDP, at each time step $t \in \{1, 2, 3, ..., T\}$ the agent receives some representation (information) of its current state from the environment $s_t \in \mathcal{S}$, and on that basis selects an action $a_t \in \mathcal{A}$ to perform. One step later, due to its action, the agent finds itself in a new state, and the environment returns a reward $R_{t+1} \in \mathcal{R}$ to the agent as feedback of its action's quality [1].

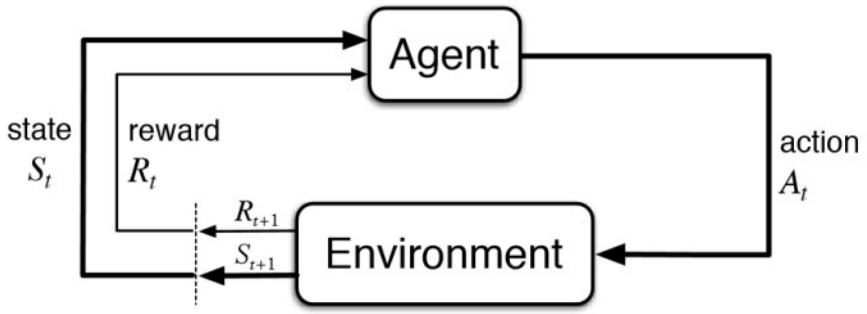


Figure 1: The agent–environment interaction in a MDP

The sequences of interactions between the agent and the environment gives the

following trajectory $S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, ...$ As we are dealing with *finite* MDP, the sets of states, actions, and rewards all have a finite number of elements, and therefore, the random variables R_t and S_t have a well-defined discrete probability distributions based on the previous action and state. That is, the probability of the agent to receive reward $\mathbf{r}$ and transition to state $\mathbf{s}'$ given that it took action $\mathbf{a}$ in state $\mathbf{s}$ is:

$$\mathcal{P}(s', r|s, a) = \mathbf{Pr}\{S_t = s', R_t = r \,|S_{t-1} = s, A_{t-1} = a\} \tag{2.1}$$

$$\forall s, s' \in \mathcal{S}, r \in \mathcal{R}, a \in \mathcal{A}$$

The function $\mathcal{P}$ is an ordinary deterministic function of four arguments defining the *dynamics* of the MDP, and satisfy the following:

i. $\mathcal{P} : \mathcal{S} \times \mathcal{R} \times \mathcal{S} \times \mathcal{A} \longrightarrow [0, 1]$

ii. $\sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} \mathcal{P}(s', r|s, a) = 1$ for all $s, s' \in \mathcal{S}, a \in \mathcal{A}$

Of course, the representations of states and actions vary greatly from task to task, and the choices of these representations have critical effects on the agent's performance.

2.2 The Objective of Reinforcement Learning

In section 2.1 we have discussed the framework representation of the RL problem, and we have shown that after each action made by the agent, the environment sends a simple number $R_t \in \mathbb{R}$ which we call *reward*. This reward signal is used by the agent to evaluate its actions and to form the primary basis for altering the policy (more about policy in section 2.3) to a better one. i.e., if the policy yielded an action with low reward, the agent will alter the policy in the future.

We define the *objective* (goal) of RL to maximize the cumulative reward it receives in the long run instead of the immediate reward. This idea is represented as what is called The Reward Hypothesis:

> *All goals and purposes can be described by the maximization of expected value of cumulative rewards*

To formalize the above hypothesis, we introduce $\mathbb{G}_t$ which represents the cumulative return after time t, and we seek to maximize the *expected cumulative return* as we are dealing with future returns:

$$\mathsf{E}[\mathbb{G}_t] = \mathsf{E}[R_{t+1} + R_{t+2} + R_{t+3} + ... + R_T] \tag{2.2}$$

In the above reward equation (2.2) the term R_T denotes the reward received at the terminal state T, means that the aforementioned approach of calculating cumulative reward is only valid when the problem at hand ends in a terminal state T. Consequently, one can distinguish between two types of problems in RL:

- *Episodic task*: a task which breaks naturally into sub-sequence tasks that all end in the *Terminal* state (T). An example of such a task is the chess game, where every game is an independent episode, starting from the same set and ends by winning or losing.

- *Continuing task*: task that goes on continually without limit (no terminal state), like an ongoing process-control task.

In continuous tasks, $T = \infty$, which makes the return that we are trying to maximize also infinity. Therefore a *discounting factor gamma* ($0 \leq \gamma \leq 1$) is introduced to equation (2.2) to turn $\mathbb{G}_t$ to a finite sequence as the following:

$$\begin{aligned} \mathbb{G}_t &= R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... + \gamma^{k-1} R_{t+k} + ... \\ &= \sum_{0}^{\infty} \gamma^k R_{t+k+1} \end{aligned} \tag{2.3}$$

Although equation (2.3) still a sum of an infinite number of terms, it is still finite if the reward is nonzero and constant. For example, if the maximum reward an agent

can receive is R_{max} we have:

$$\begin{aligned}\mathbb{G}_t = \sum_{0}^{\infty} \gamma^k R_{t+k+1} &\leq \sum_{0}^{\infty} \gamma^k R_{max} \\ &\leq R_{max} \sum_{0}^{\infty} \gamma^k \\ &\leq R_{max} \times \frac{1}{1-\gamma} \Rightarrow \text{which is finite}\end{aligned}$$

We can express $\mathbb{G}_t$ in equation 2.3 recursively as:

$$\mathbb{G}_t = R_{t+1} + \gamma \mathbb{G}_{t+1} \tag{2.4}$$

When the discounting factor (γ) is close to zero, we describe the agent as *short-sighted* (myopic) agent as it gives more weight to the immediate rewards received from the environment. On the contrary, for $\gamma \approx 1$ the discount factor will make the agent strive for a long-term reward, we describe it as *far-sighted* agent.

2.3 Policy and Value Function

Value functions are being used by almost all RL methods to estimate how good (in terms of expected return) it is for the agent to be in a given state or to perform an action in a given state. This evaluation is being made based on the future expected sum of rewards. Accordingly, value functions are determined with respect to the future actions the agent will take. We call a particular way of acting a *Policy* [1]. Formally, we define policy (π) as an ordinary function which maps from environment's states to probabilities of selecting each possible action, then $\pi(a|s)$ is the probability of taking action $A_t = a$ giving that the agent is in state s and following policy π. We define two types of value functions that can be estimated from experience:

1. *State-value function* [11]: The value function of a state s under a policy π, denoted $\mathcal{V}_\pi(s)$, is the expected return when starting in s and following π thereafter if we consider the discounted average reward in equation 2.3, we conclude the

following expression:

$$\mathcal{V}_\pi(s) \doteq E_\pi[\mathbb{G}_t|S_t = s] = E_\pi[\sum_{k=0}^{\tau-t} \gamma^k R_{t+k+1}|S_t = s], \forall s \in \mathcal{S} \tag{2.5}$$

where $\tau < \infty$ if the task is episodic, and $\tau = \infty$ if it is continuing.

2. *Action-value function*: In a similar way, the action-value function is defined as the value of taking action a in state s under a policy π, denoted $q_\pi(s, a)$, as the expected return starting from s, taking the action a, and thereafter following policy π:

$$q_\pi(s, a) \doteq E_\pi[\mathbb{G}_t|S_t = s, A_t = a] = E_\pi[\sum_{k=0}^{\tau-t} \gamma^k R_{t+k+1}|S_t = s, A_t = a] \tag{2.6}$$

where $\tau < \infty$ if the task is episodic, and $\tau = \infty$ if it is continuing.

Bellman Equations

Bellman equations named after *Richard Bellman* [12], are the fundamental property of value functions used in *dynamic programming* as well as in reinforcement learning to solve MDPs, and they are essential to understand how many RL algorithms work. In essence, the Bellman equation expresses a relationship between the value of a state and the values of its successor states.

To better understand how to derive the Bellman equation for value functions, consider the backup diagram in Fig.2 that shows state s (open circles) in MDP and its three possible actions $\mathcal{A}$ (solid circles). Starting from the root node at the top (s), the agent could follow any of the three actions to end up in a state s' and receive reward r. For choosing one action $a \in \mathcal{A}$ the expected return would be the discounted expected reward from next state plus the reward expected from action a as in equation (2.4):

$$\pi(a|s)P(s', r|s, a)[r + \gamma E[\mathbb{G}_{t+1}|S = s']]$$

So the value function of state s would be the sum over all possibilities of expected

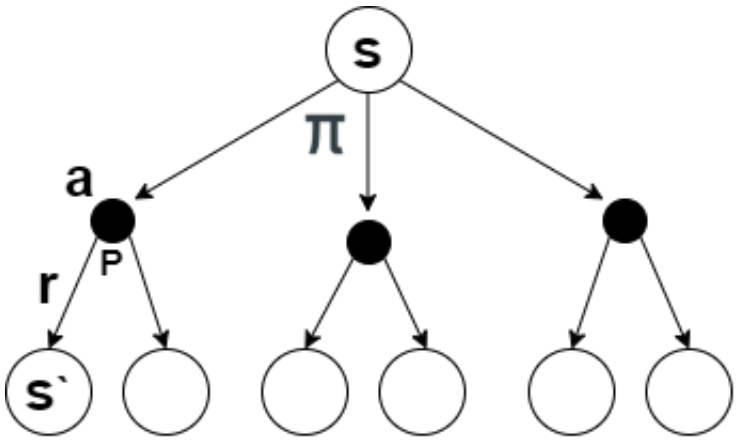


Figure 2: Backup diagram for value function ($\mathcal{V}_\pi$)

returns, weighting each by its probability of occurring following policy π:

$$\begin{aligned}\mathcal{V}_\pi(s) &\doteq E_\pi[\mathbb{G}_t|S_t = s] = E_\pi[R_{t+1} + \gamma\mathbb{G}_{t+1}|S_t = s] \\ &= \sum_a \pi(a|s) \sum_{s'} \sum_r P(s', r|s, a)[r + \gamma E[\mathbb{G}_{t+1}|S = s']] \\ \mathcal{V}_\pi(s) &\doteq \sum_a \pi(a|s) \sum_{s'} \sum_r P(s', r|s, a)[r + \gamma\mathcal{V}_\pi(s')], \forall s \in \mathcal{S} \end{aligned} \tag{2.7}$$

In a similar way we define the action-value function as:

$$\begin{aligned} q_\pi(s, a) &\doteq E_\pi[\mathbb{G}_t|S_t = s, A_t = a] \\ &= \sum_{s'} \sum_r P(s', r|s, a)[r + \gamma E[\mathbb{G}_{t+1}|S = s']] \end{aligned}$$

where:

$$\gamma E_\pi[\mathbb{G}_{t+1}|S = s'] = \gamma \sum_{a'} \pi(a'|s') E_\pi[\mathbb{G}_{t+1}|S_{t+1} = s', A_{t+1} = s']$$

$$q_\pi(s, a) = \sum_{s'} \sum_r P(s', r|s, a)[r + \gamma \sum_{a'} \pi(a'|s') q_\pi(s', a')] \tag{2.8}$$

Optimal Policy and Value functions

As explained in section 2.2, the goal of reinforcement learning is to maximize the expected return, this goal is accomplished by finding a policy π_* better than or equal to all other policies (π'), we call this policy *the optimal policy*. In terms of mathematical

optimization, we can consider finding the optimal policy as finding the global/local optima. For a policy π to be better than a policy π', the value function for each state following π must be equal or larger than those value functions following π':

$$\pi \geq \pi' \text{ if and only if } \mathcal{V}_\pi(s) \geq \mathcal{V}_{\pi'}(s), \forall s \in \mathcal{S}$$

From Bellman equations (equations 2.7 and 2.8) we can derive what is called *The Bellman Optimality Equations*. Intuitively, the Bellman optimality equation expresses the fact that the value of a state under an optimal policy must equal the expected return for the best action from that state [1]

- *optimal state-value function:*

$$\begin{aligned}\mathcal{V}_*(s) &= E_{\pi*}[\mathbb{G}_t|S_t = s] = \max_\pi \mathcal{V}_\pi(s) \\ &= \sum_a \pi_*(a|s) \sum_{s'} \sum_r P(s', r|s, a)[r + \gamma \mathcal{V}_*(s')] \\ \mathcal{V}_*(s) &= \max_a \sum_{s'} \sum_r P(s', r|s, a)[r + \gamma \mathcal{V}_*(s')] \end{aligned} \tag{2.9}$$

- Similarly, we define *optimal action-value function:*

$$q_*(s, a) = \max_\pi q_\pi(s, a) = \sum_{s'} \sum_r P(s', r|s, a)[r + \gamma \max_{a'} q_*(s', a')] \tag{2.10}$$

- And finally, we define the *optimal policy equation* as:

$$\pi_*(s) = \arg\max_a \sum_{s'} \sum_r P(s', r|s, a)[r + \gamma \mathcal{V}_*(s')] \tag{2.11}$$

Notice that if the probability P of the MDP is known, the system can be solved using nonlinear solvers to find $\mathcal{V}_*$ thus find π_* but usually, P is unknown for the optimal policy.

2.4 Generalized Policy Iteration

Generalize Policy Iteration (GPI) is a very important principle in RL, where almost all RL methods are well described as GPI. This principle refers to the interaction of two processes:

i. *policy-evaluation*: Giving a policy π computes the value function and makes it consistent for the current policy. The main goal of this step is to collect information (value function) under the given policy to determine how good it is.

ii. *policy-improvement*: Giving the computed value functions for policy π in the first step, this step helps find a better policy π' by choosing greedy actions with respect to the value functions computed for π in each state, i.e., taking other possible actions in particular states that give better rewards than the action the current policy proposes. This step gives a strictly better policy unless the original policy π is already optimal.

These two steps can be implemented in various ways (like *asynchronous* methods, *value-iteration*,...), but the main concept is that they keep alternating with the value function always being driven toward the true value function for the policy (policy-evaluation) and the policy always being improved with respect to the value function (policy-improvement) until processes stabilize and do not cause changes to the function values or policies, i.e., the optimal policy π_* has been found and its value function $\mathcal{V}_*$ is computed (Fig.3).

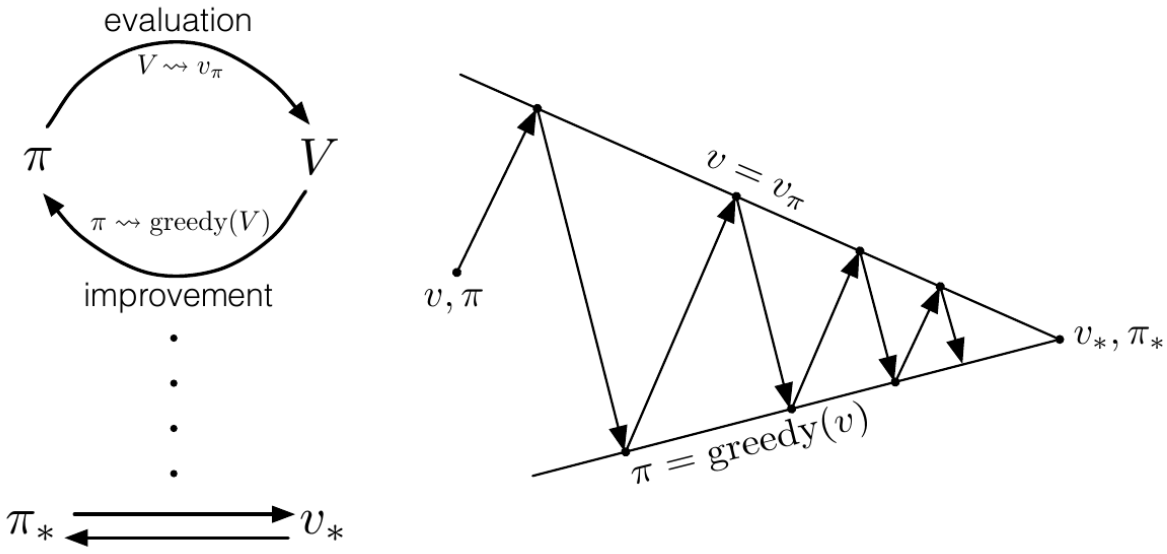


Figure 3: Generalized Policy Iteration[1]

2.5 Model-Based Algorithms (Dynamic Programming)

The *model* of the environment is anything used by the agent to predict the response of the environment to its actions. The model stores knowledge about the transition and reward dynamics of the environment so it can predict the next state and reward giving it the current action and state. In addition, the model can be used to generate samples of the environment(like *distribution models*). The main advantage of having a model is that it gives the agent the ability to forecast the future by thinking and planning its actions. However, having a model of the environment is not always possible, and learning a model is an arduous task and error-prone. In model-based methods, the model may be known as in *Dynamic programming* and *Heuristic search* or learned as in the *Dyna-Q* algorithm. Since having a model of the environment is necessary for model-based algorithms, their utilization in RL is limited.

When a perfect model of the environment (the complete probability distributions of all possible transitions in finite-MDP) is available, *Dynamic Programming* [13](DP)

algorithms can be used to compute the optimal policies of the system. Since DP algorithms expect a complete description of the model, which is hard to ensure for many systems because they tend to be computationally expensive, their utility is limited in reinforcement learning. However, their computational mechanisms are still theoretically important, as many other model-free algorithms utilize. As mentioned earlier, DP algorithms assume full model description and finite MDP, that is, state and action sets, $\mathcal{S}$ and $\mathcal{A}$ are finite and can be stored in tables, and the dynamics are given by a set of probabilities for all states, actions, and rewards $P(s', a|s, r)$.

In essence, DP algorithms turn Bellman equations (see section 2.3) into update rules for improving approximations of the desired value function. This improvement is reached through the two steps of the GPI mechanism (policy evaluation and policy improvement) described in section 2.4. Regarding the way the GPI steps are being applied, we distinguish between two possible DP methods; the first applies the GPI mechanism steps separately and consecutively, we call this method *Policy Iteration*, whereas the second method represents a tight integration of policy evaluation and improvement.

Policy Iteration [14]

Similar to the initial solution in heuristic search, we provide the Policy Iteration algorithm with a random policy or a policy that we believe is good or near-optimal to be the initial starting point. The first step is to calculate the value function of the given policy π iteratively using equation 2.7 until a predefined-threshold of accuracy is reached, after computing the value function $\mathcal{V}_\pi$ we can then implement the second step to find a better policy π' by choosing greedy actions with the highest reward as in equation 2.11. These two steps are repeated until no improvement occurs (optimal policy has been computed). The pseudo-code is illustrated in Algorithm.1.

$$\pi_0 \longrightarrow \mathcal{V}_{\pi_0} \Longrightarrow \pi_1 \longrightarrow \mathcal{V}_{\pi_1} \Longrightarrow \pi_2 \longrightarrow \ldots \Longrightarrow \pi_* \longrightarrow \mathcal{V}_{\pi_*}$$

Where $\longrightarrow$ and $\Longrightarrow$ indicate evaluation step and improvement step respectively.

Algorithm 1: Policy Iteration for estimating $\pi \approx \pi_*$ [14]

Input: A small threshold $\theta > 0$ to determine the estimation accuracy

1. **Initialization**

$\mathcal{V}_{(s)} \in \mathbb{R}$ and $\pi_{(s)} \in \mathcal{A}$ arbitrarily for all $s \in \mathcal{S}$ except for $\mathcal{V}_{terminal} = 0$ if the task is episodic

2. **Policy Evaluation**

repeat

  $\Delta = 0$

  foreach $s \in \mathcal{S}$ **do**

    $V \longleftarrow \mathcal{V}_s$

    $\mathcal{V}_\pi(s) \longleftarrow \sum_a \pi(a|s) \sum_{s'} \sum_r P(s', r|s, a)[r + \gamma \mathcal{V}_\pi(s')]$

    $\Delta \longleftarrow \max(\Delta, |V - \mathcal{V}_{(s)}|)$

until $\Delta < \theta$

3. **Policy Improvement**

policy_stable $\longleftarrow$ $True$

foreach $s \in \mathcal{S}$ **do**

  old_policy $\longleftarrow$ $\pi_{(s)}$

  $\pi_(s) = \arg\max_a \sum_{s'} \sum_r P(s', r|s, a)[r + \gamma \mathcal{V}_(s')]$

  if *old_policy* $\neq \pi_{(s)}$ **then**

    policy_stable $\longleftarrow$ $False$

if *policy_stable is True* **then**

  return $\mathcal{V} \approx \mathcal{V}_*, \pi \approx \pi_*$

else

  Go to 2. **Policy Evaluation**

Value Iteration

One drawback of the preceding algorithm is the computation time. Each iteration involves policy evaluation, which may require many iterations to convergence to the real function values. Value iteration algorithm combines policy evaluation step and policy improvement step in one simple update function, so it would be unnecessary to wait for full convergence, where policy evaluation is stopped after one sweep, then greedy action is chosen, then another sweep of evaluation, and so on. As in Algorithm.2.

Algorithm 2: Value Iteration for estimating $\pi \approx \pi_*$

Input: A small threshold $\theta > 0$ to determine the estimation accuracy

1. **Initialization**
$\mathcal{V}_{(s)} \in \mathbb{R}$ and $\pi_{(s)} \in \mathcal{A}$ arbitrarily for all $s \in \mathcal{S}$ except for $\mathcal{V}_{terminal} = 0$ if the task is episodic

2. **repeat**
   $\Delta = 0$
   foreach $s \in \mathcal{S}$ **do**
      $V \longleftarrow \mathcal{V}_s$
      $\mathcal{V}_\pi(s) \longleftarrow \max_a \sum_{s'} \sum_r P(s', r|s, a)[r + \gamma \mathcal{V}_\pi(s')]$
      $\Delta \longleftarrow \max(\Delta, |V - \mathcal{V}_{(s)}|)$

until $\Delta < \theta$

return *A deterministic policy* $\pi \approx \pi_*$ *such that* $\pi_{(s)} = \arg\max_a \sum_{s'} \sum_r P(s', r|s, a)[r + \gamma \mathcal{V}_{(s')}]$

2.6 Model-Free Algorithm

Although a model of the environment is still required to generate samples in *model-free* methods, unlike model-based methods, do not need complete knowledge of the model (environment) dynamics neither the transition probability distributions, primarily generating experience sampled from the desired probability distributions is much easier than obtaining the transition probability distributions. Generally, reinforcement learning refers to these model-free algorithms, and they are more popular and have been more extensively developed and tested than model-based methods [8].

Model-free RL algorithms are classified based on how to represent and train the agent into three main approaches:

- **Critic-only approach**

  Methods in this family learn to estimate the value function (State-value function or action-value function) by using Bellman optimality equations as objective functions (see section 2.3). We distinguish between two different ways the agent learns the value function of the system. The first way is by representing

value functions as arrays or tables and updating these values with more accurate ones after each iteration as the agent has a better understanding of the environment through its interaction with it, we call these methods *Tabular Solution Methods*. These methods often find exact solutions, i.e., they often find exactly the optimal value function and the optimal policy; however, one drawback of these Tabular methods is that the state and action spaces must be small enough to be stored in tables, the major steps of Tabular methods are shown in Fig.4. we discuss two main methods of Tabular solution methods, Monte Carlo methods in section 2.6.1, and Temporal-Difference Learning methods in section 2.6.2.

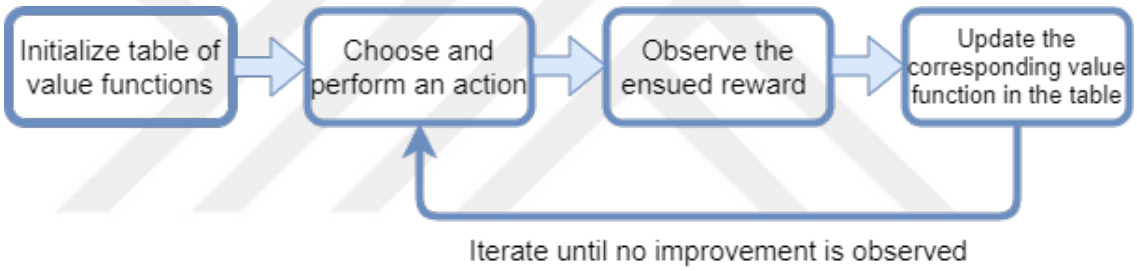


Figure 4: Tabular Algorithms Steps

The second possible way in critic-only approach called *Approximate Solution Methods*, as the name suggests, these methods do not find exact solutions as Tabular methods, but they find approximate solutions. These methods tend to generalize better than Tabular methods but have lower discrimination (see Fig.5), and they are capable of learning the value function for systems with enormous state and action spaces. Approximate methods achieve this generalization by combining RL with *supervised learning* algorithms. These Approximate solution methods are discussed in section 2.6.3.

- **Actor-only approach** (section 2.6.4)

  These methods take a different approach, rather than using value function to

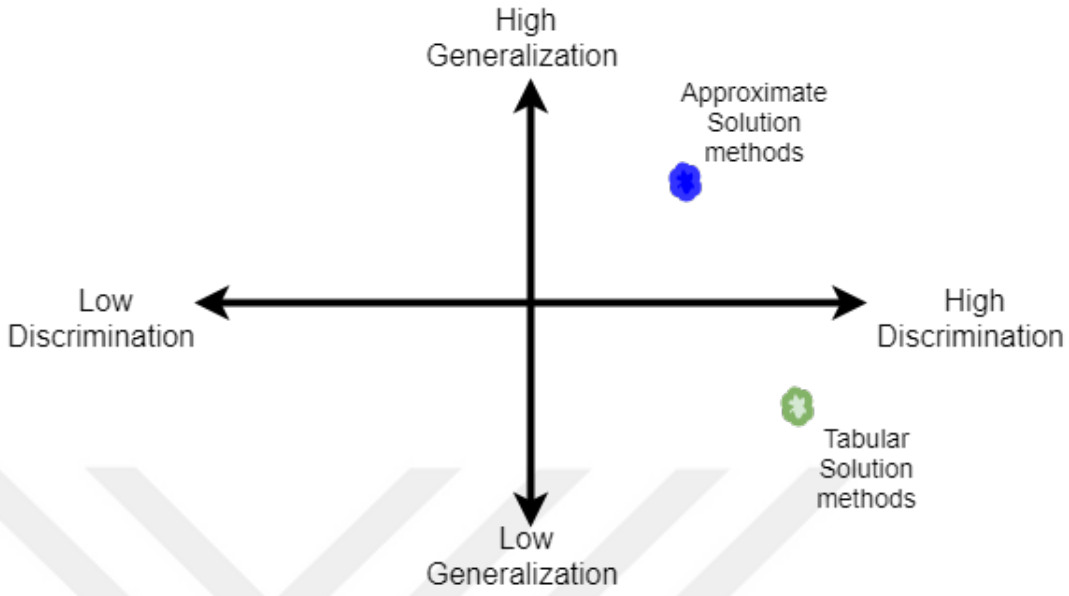


Figure 5: Categorizing value function methods based on Generalization & Discrimination

evaluate every action and find the optimal policy, they learn a policy which directly maps states to actions.

- **Actor-critic approach** (section 2.6.5)

  The actor-critic approach combines the actor-only with the critic-only approach.

2.6.1 Monte Carlo Methods

Monte Carlo(MC) method is a model-free, tabular solution method to solve the reinforcement learning problem based on averaging sample returns acquired from the agent experience (interaction with its environment). Typically, MC methods are applied to episodic tasks, where policy is updated *only at the end of an episode*. The same in DP, MC follows the GPI (see section 2.4) mechanism by alternating between policy evaluation and policy improvement. Because the model is not available, estimating action values are more useful than state values in MC. One main problem that arises with sample averaging methods such as MC is *exploration* since MC starts with a policy π and choose actions greedily, the agent will always end up visiting the

same states and will never be able to observe the return of actions in the states that never been selected by the policy. One way to maintain exploration is the assumption of *exploring starts*, where we specify that episodes start in a state-action pair (s_0, a_0) and that every pair has a non-zero probability of being selected as the starting pair, i.e., randomly start in state s_0 and choose random action a_0. This guarantees that all state-action pairs will be visited an infinite number of times within the limit of an infinite number of episodes (Algorithm.3). The Exploring Start approach is some-

Algorithm 3: Monte Carlo ES (Exploring Starts), for estimating $\pi \approx \pi_*$

1. **Initialization**
$\pi(s) \in \mathcal{A}(s)$ arbitrarily, $\forall s \in \mathcal{S}$
$Q(s,a) \in \mathbb{R}$ arbitrarily $\forall s \in \mathcal{S}, \forall a \in \mathcal{A}(s)$
$Returns(s,a) \longleftarrow$empty list, $\forall s \in \mathcal{S}, \forall a \in \mathcal{A}(s)$

2. **repeat** for each episode
Randomly choose $s_0 \in \mathcal{S}, a_0 \in \mathcal{A}(s_0)$such that all pairs have probability>0
Generate an episode starting from (s_0, a_0), following π:
$s_0, a_0, R_1, s_1, a_1, R_2, \cdots, s_{T-1}, a_{T-1}, R_T$
G $\longleftarrow 0$
foreach *step of episode,* $t = T-1, t-2, \cdots, 0$ **do**
$G_t \longleftarrow \gamma G_{t+1} + R_{t+1}$
Append G to $Returns(s_t, a_t)$
$Q(s_t, a_t) \longleftarrow average(Returns(s_t, a_t))$
$\pi(s_t) \longleftarrow \arg\max_a Q(s_t, a)$

until *forever*

times useful, but it cannot be relied upon in general, wherein real problems can be problematic since it may be difficult to sample an initial state-action pair for many problems randomly. For instance, randomly sampling state-action initial pairs for a self-driven car means putting the car in many different configurations to sample all possible state-action pairs. A better way of exploring in MC is the *ϵ-soft policy algorithm*. This algorithm continue to visit all state-action pairs indefinitely by choosing action that has maximal estimated action with probability $1 - \epsilon + \frac{\epsilon}{|\mathcal{A}(s)|}$ where $\mathcal{A}(s)$ is the number of actions in state s, which means that non-greedy actions also can be selected with probability $\frac{\epsilon}{|\mathcal{A}(s)|}$ (Algorithm.4).

Algorithm 4: Monte Carlo ϵ-soft policy, for estimating $\pi \approx \pi_*$

Input: A small $\epsilon > 0$ which determines the exploration degree

1. **Initialization**

$\pi \longleftarrow$ an arbitrarily ϵ-soft policy
$Q(s,a) \in \mathbb{R}$ arbitrarily $\forall s \in \mathcal{S}, \forall a \in \mathcal{A}(s)$
$Returns(s,a) \longleftarrow$empty list, $\forall s \in \mathcal{S}, \forall a \in \mathcal{A}(s)$

2. **repeat** for each episode

Generate an episode following π:
$s_0, a_0, R_1, s_1, a_1, R_2, \cdots, s_{T-1}, a_{T-1}, R_T$
G $\longleftarrow 0$
foreach *step of episode,* $t = T-1, t-2, \cdots, 0$ **do**
$G_t \longleftarrow \gamma G_{t+1} + R_{t+1}$
Append G to $Returns(s_t, a_t)$
$Q(s_t, a_t) \longleftarrow average(Returns(s_t, a_t))$
$A_* \longleftarrow \arg\max_a(s_t, a)$ /* With ties broken arbitrarily */
forall $a \in \mathcal{A}(s_t)$ **do**

$$\pi(a|s_t) = \begin{cases} 1 - \epsilon + \frac{\epsilon}{|\mathcal{A}(s)|} & \text{if } a = A_* \\ \frac{\epsilon}{|\mathcal{A}(s)|} & \text{if } a \neq A_* \end{cases}$$

until *forever*

2.6.2 Temporal-Difference Learning Methods

> "If one had to identify one idea as central and novel to RL, it would undoubtedly be temporal-difference (TD) learning" [1]

In MC methods, values are updated at the end of the episode since G_t is unknown until the end of the episode (the agent might revisit the state); this can be problematic, especially if the episode takes a long time to end or if the policy used leads the agent to be stuck in a place where it can never reach the terminal state. *Temporal difference* (TD) methods like MC methods can learn directly from raw experience without a model of the environment. Like DP methods, they bootstrap, i.e., TD methods only need to wait until the next time step to update values, at a time t+1 they immediately form a target and make a useful update using the observed reward R_{t+1} and the estimate $q(s_{t+1}, a_{t+1})$. Another advantage of TD over MC methods is that they are implemented in a fully incremental fashion rather than storing all

rewards in an array or list and then perform the computation (averaging) which requires huge memory and computation requirements when rewards grow huge over time, this incremental way of computing the average rewards in n steps is shown bellow:

$$
\begin{aligned}
Q_{n+1} &= \frac{\text{Sum of rewards when action a is taken prior to t}}{\text{Number of times action a is taken prior to t}} \\
&= \frac{\sum_{i=1}^{n} R_i}{n} = \frac{1}{n}(R_n + \frac{(n-1)}{(n-1)}\sum_{i=1}^{n-1} R_i) = \frac{1}{n}(R_n + (n-1)Q_n) \\
&= \frac{1}{n}(R_n + nQ_n - Q_n) = Q_n + \frac{1}{n}[R_n - Q_n] \\
Q_{n+1} &= Q_n + \alpha[R_n - Q_n]
\end{aligned}
$$

$$NewEstimate = OldEstimate + StepSize[Target - OldEstimate] \tag{2.12}$$

Where α is the step size parameter to determine how much weight (importance) we give to the past rewards. In a similar way, we can measure the difference between the estimate action value $Q(s,a)$ and the better estimate $R_{t+1} + \gamma Q(s_{t+1}, a_{t+1})$:

$$Q(s_t, a_t) \longleftarrow Q(s_t, a_t) + \alpha[R_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)] \tag{2.13}$$

As in all tabular methods, we follow the Generalized Policy Iteration (GPI) mechanism, where the update is done after every transition from a non-terminal state, if the state is terminal we set $Q(s_{t+1}, a_{t+1})$ to zero. An example of such a TD algorithm is *SARSA*, the name comes from the fact that SARSA uses every element of the quintuple of events $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$. It's also known as The *On-Policy TD algorithm* to indicate that the algorithm uses the behavior policy π to estimate action-value function $q(s,a)$ and at the same time changes π toward greediness with respect to q_π. The pseudo code of SARSA is shown in Algorithm.5

Algorithm 5: SARSA (on-policy TD control) for estimating $Q \approx q_*$ [1]

Input: Step size $\alpha \in (0, 1]$, small $\epsilon > 0$

1. **Initialization**
$Q(s, a) \in \mathbb{R}$ arbitrarily $\forall s \in \mathcal{S}, \forall a \in \mathcal{A}(s)$ except that Q(terminal, .) $= 0$

2. **repeat** for each episode:
   - Initialize **s**
   - Choose action **a** from **s** using policy derived from Q (e.g., ϵ-greedy)
   - **foreach** *state in the episode until* ***s*** *is terminal* **do**
     - Take action **a**, observe R, $\mathbf{s}_{t+1}$
     - Choose $\mathbf{a}_{t+1}$ from $\mathbf{s}_{t+1}$ using policy derived from Q (e.g., ϵ-greedy)
     - $Q(s_t, a_t) \longleftarrow Q(s_t, a_t) + \alpha[R + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]$
     - $\mathbf{s} \longleftarrow \mathbf{s}_{t+1}$
     - $\mathbf{a} \longleftarrow \mathbf{a}_{t+1}$

until *forever*

Off-Policy Temporal-Difference Learning (Q-Learning)

One of the early breakthroughs in RL was the development of *Off-Policy TD methods*. The off-policy methods attempt to solve the dilemma of exploration/exploitation trade-off, where most of the methods discussed so far seek to learn action values conditional on *subsequent optimal behavior*, but in order to explore all actions (to find the optimal one), they need to behave non-optimally. Off-policy TD methods use two policies instead to solve the problem:

1. *Target Policy*: The policy which is learned to become the optimal policy, usually denoted by "π", $\pi(a|s)$ and mostly it's a deterministic policy like Greedy policy. The value function that the agent is learning is based on this Target policy.

2. *Behavior Policy*: The policy that the agent uses to select actions (generate behavior), usually denoted by "b", $b(a|s)$ and mostly it's a stochastic policy like Random policy and ϵ-greedy policy. It's used as an exploratory policy.

In order to use episodes from b to estimate values for π , we require that every action taken under π is also taken, at least occasionally, under b. That is the behavior policy must always cover the target policy: $\pi(a|s)$ where $b(a|s) > 0$, this is called the

assumption of *coverage* [1].

One problem that arises with using two different policies is that we are trying to estimate the expected values under one policy distribution (target policy) given samples generated from another distribution (behavior policy), this can be solved by weighting returns according to the relative probability of their trajectories occurring under the target and behavior policies ratio called the *importance-sampling ratio*, $\rho(s_i) = \frac{\pi(s_i)}{b(s_i)}$.

$$E_\pi[G_t] = E_b[G_t\rho(s)] \approx \frac{1}{n}\sum_{i=1}^{n} R_i\rho(s_i) \text{ Where } R_i \text{ generated from b not from } \pi$$

One of the most famous algorithms of off-policy TD methods is the *Q-Learning* algorithm. This algorithm learns to approximate action-value function Q(s,a) using the optimal action-value function (choose the greedy action), regardless of how the agent has chosen to explore the environment, i.e., the behavior policy (b) determines which state-action pair (s_t, a_t) to visit and update, where the target policy (π) performs the update by choosing the action-state pair (s_{t+1}, a_{t+1}) with the maximum reward (Algorithm.6). The update function of Q-learning is defined as:

$$Q(s_t, a_t) \longleftarrow Q(s_t, a_t) + \alpha[R + \gamma \max_{\mathbf{a}} Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)] \tag{2.14}$$

Expected SARSA

Instead of using the next action-value to update the estimates of the action-value function as in SARSA (Equation 2.13) or the max action-value as in Q-learning (Equation 2.14), *Expected SARSA* explicitly compute the expectation of all possible action values following that policy:

$$Q(s_t, a_t) \longleftarrow Q(s_t, a_t) + \alpha[R + \gamma \sum_{a'} \pi(a'|s_{t+1})Q(s_{t+1}, a') - Q(s_t, a_t)] \tag{2.15}$$

Because of the use of the expected values, Expected SARSA has a more stable update target than SARSA (less variance that SARSA) but it's more expensive as the number if actions increases.

Algorithm 6: Q-learning (off-policy TD) for estimating $\pi \approx \pi_*$[1]

Input: Step size $\alpha \in (0, 1]$, small $\epsilon > 0$

1. **Initialization**
$Q(s, a) \in \mathbb{R}$ arbitrarily $\forall s \in \mathcal{S}, \forall a \in \mathcal{A}(s)$ except that Q(terminal, .) $= 0$

2. **repeat** for each episode:
   Initialize **s**
   Choose action **a** from **s** using policy derived from Q (e.g., ϵ-greedy or Random)
   foreach *state in the episode until* ***s*** *is terminal* **do**
      Take action **a**, observe R, $\mathbf{s}_{t+1}$
      Choose $\mathbf{a}_{t+1}$ from $\mathbf{s}_{t+1}$ using policy derived from Q (e.g., ϵ-greedy)
      $Q(s_t, a_t) \longleftarrow Q(s_t, a_t) + \alpha[R + \gamma \max_{\mathbf{a}} Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]$
      $\mathbf{s} \longleftarrow \mathbf{s}_{t+1}$

until *forever*

2.6.3 Approximate Solution Methods (Deep RL)

Most of the RL methods presented so far stored value functions in a table-like structure to perform updates on these values after each iteration; this is considered a colossal limitation when we are trying to apply such methods to a problem with enormous state space, which makes storing and computing the value functions almost impossible. For example, if we train an agent to play a video game where states are presented to the agent as images, we would have a tremendous number of possible images (larger than the number of atoms in the universe). In such an ample state space, almost every state will be encountered once or never, and we cannot find an optimal policy or optimal value function even in infinite time, and data [1].

Approximate Solution Methods solve the problem of large state space by generalizing from previously encountered states to the unseen ones using what is called *Function approximation*, which is an instance of *Supervised Learning*. Here, the approximate value function is represented not as a table but as a parameterized function form with a vector of *weights*, $W \in \mathbb{R}$, for example, the approximate state-value of s given weight vector W is $\hat{V}(s, w) \approx \mathcal{V}_\pi(s)$, where $\hat{V}$ might have been computed by

decision tree algorithm or by a multi-layer *artificial neural network* or perhaps it's a simple linear approximate function ($\hat{V}(s,w) = w_1X + w_2Y$). Typically the number of weights (dimensionality of W) is much less than the number of states. Function approximation uses the experience generated using a known policy π to find the weight vectors W, which maps the input state (or state-action pair) to the approximate value function, similar to a supervised learning algorithm.

The way states are represented in terms of features is critical in approximate solution methods. Each state s has a corresponding real-valued vector $\mathbf{X}(s)$ (same length with the weight vector W) of features determined according to the problem to be solved.

It is simply impossible to estimate all states' true value functions with approximate solution methods as tabular methods do. Therefore we need to a use prediction error called *Prediction Objective* $\bar{VE}$ that will be minimized by *Gradient Decent* methods to find the best set of features W:

$$\begin{aligned} \bar{VE} &= \sum_{s\in\mathcal{S}} \mu(s)[\mathcal{V}_\pi(s) - \hat{V}(s,w)]^2 \\ \text{Where: } \mu(s) &\geq 0, \sum_s \mu(s) = 1 \end{aligned} \tag{2.16}$$

Where $\mu(s)$ is a state distribution to represent how much is essential to minimize the error in each state, that is because the update of one value function of a state affects the values of many other states, i.e., making one state's estimate more accurate means making another estimate less accurate.

In equation 2.16, for supervised learning problem $\mathcal{V}(s)$ is the true label of the instance and does not change (training is stable), but with RL problem, $\mathcal{V}_\pi(s)$ is the observed label (value function) by the agent following policy π, hence, is continuously changing with each iteration as the agent continue to explore.

Similar to equation 2.12 we define the update function to minimize error on the observed return G_t, following a policy π, by adjusting the weight vector W after each return observed by a small amount in the direction that would reduce the error on

that example (also known as *Stochastic Gradient Descent*)

$$W_{t+1} = W_t + \alpha[G_t - \hat{V}(s_t, w_t)]\nabla(s_t, w_t) \tag{2.17}$$

Algorithm.7 represents an approximate solution method to estimate a given policy π (find value function) using Monte Carlo method to generate return G_t.

Algorithm 7: Gradient Monte Carlo algorithm for estimating $\hat{V} \approx \mathcal{V}_\pi$[1]

Input:
Step size $\alpha \in (0, 1]$
The policy π to be evaluated
A differentiable function $\hat{V} : \mathcal{S} \times \mathbb{R} \longrightarrow \mathbb{R}$

1. **Initialization**
Value-function weights $W \in \mathbb{R}$ arbitrarily (e.g., W=0)

2. **repeat** for each episode:
   Generate an episode
   $s_0, a_0, R_1, s_1, a_1, R_2, ..., s_{T-1}, a_{T-1}, R_T$ following policy π
   foreach *step in episode, t= 0, 1, 2,..., T-1* **do**
      $W = W + \alpha[G_t - \hat{V}(s_t, w)]\nabla\hat{V}(s_t, w)$

until *forever*

Deep Reinforcement Learning

Deep Reinforcement Learning (DRL) is an approximate solution method, where RL defines the *objective* , and *Deep Learning* (DL) defines the *mechanism*. A neural network (like CNN) is used as a function approximator to approximate optimal policy (π_*) and/or optimal value functions ($\mathcal{V}_*/Q_*$) based on one of the RL algorithm families to represent the loss function (TD, MC, etc.).

One of the most famous algorithms in DRL is the *Deep Q-learning* (DQL) algorithm [15] which has been designed to outperform the human level of play in Atari game; this algorithm is built upon the famous TD-gammon algorithm [16] which used temporal-difference learning method (see section 2.6.2) to approximate the state-value function using a multi-layer perceptron with one hidden layer.

DQL algorithm uses the off-policy TD control algorithm (Q-learning) described

in section 2.6.2 but with using neural networks instead of table structure, Fig.6 illustrates the difference between the two algorithms. The DQL algorithm utilize a

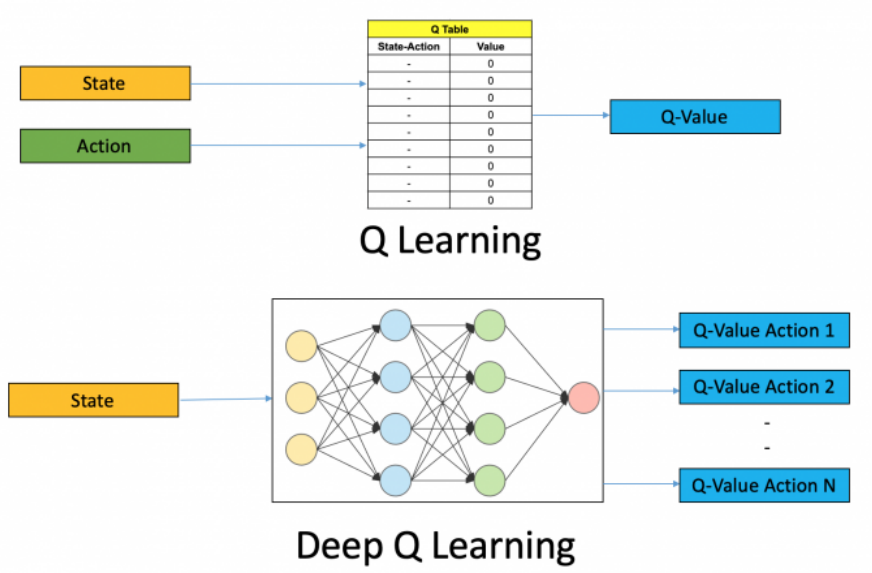


Figure 6: Comparison between DQL and QL algorithms

technique known as *experience replay*[17] to solve the strong correlations between the samples when acquired consecutively from the environment, experience replace technique forms a kind of a buffer(data-set $\mathcal{D}$) that stores the agent's experiences at each time-step, $e_t = (s_t, a_t, r_t, s_{t+1})$ pooled from different episodes and by randomly sampling experience from $\mathcal{D}$ to preform the Q-value updates, the algorithm reduces variance of these updates (Algorithm.8).

2.6.4 Policy Gradient Methods (Actor-only)

All methods discussed so far relied on the Generalized Policy Iteration(GPI) (section 2.4) framework to learn approximate action values to infer a good policy, *Policy Gradient Methods* estimate the gradient of the objective which is maximizing rewards with respect to the policy parameters and adjust the parameters based on this estimate, we use θ instead of W to represent the policy parameter vector. The parameterized policy function will take state and action as an input and will return the probability

Algorithm 8: Deep Q-learning with Experience Replay

Input:
Step size $\alpha \in (0, 1]$, small $\epsilon > 0$
A differentiable action-value function (e.g., CNN) $\hat{q} : \mathcal{S} \times \mathcal{A} \times \mathbb{R} \longrightarrow \mathbb{R}$

1. **Initialization**
Action-value function weights $W \in \mathbb{R}$ arbitrarily (e.g., W=0)
Replay memory $\mathcal{D}$ to capacity N

2. **repeat** for each episode:
    Initialize **s**
    foreach *step in episode, t=1, 2, 3,.., T* **do**
        Choose action **a** from **s** using ϵ-greedy policy
        Take action **a**, observe R, $\mathbf{s}_{t+1}$
        Store transition (s_t, a_t, r_t, s_{t+1}) in $\mathcal{D}$
        Randomly select a transition (s_t, a_t, r_t, s_{t+1}) from $\mathcal{D}$
        if $\boldsymbol{s}_T$ *(Terminal state)* **then**
            $W_t = W_t + \alpha[r_t - \hat{q}(s_t, a_t, W_t)]\nabla\hat{q}(s_t, a_t, W_t)$
            Go to next episode
        else
            $W_{t+1} = W_t + \alpha[r_t + \gamma \max_{a'} \hat{q}(s_{t+1}, a', W_t) - \hat{q}(s_t, a_t, W_t)]\nabla\hat{q}(s_t, a_t, W_t)$

until *forever*

of taking that action in that state, Fig.7 shows the difference between approximate solution methods (section 2.6.3), which return the value of the action a taken in state s ($\hat{q}(s, a, w) \in \mathbb{R}$) as an output and the policy gradient methods which return the probability of taking action a in state s ($\pi(a|s, \theta) > 0$ and $\sum_{a \in \mathcal{A}} \pi(a|s, \theta) = 1$) as an output.

In section 2.2, we showed that the reward in RL can take two forms, the first form is discounted when dealing with continuous tasks and the other one is for episodic tasks. Here, another way to represent rewards for continuous tasks will be utilized, it's called the *Average Reward*, which is simply calculated by taking the differences between reward and the average reward of the policy at each time:

$$G_t = R_{t+1} - r(\pi) + R_{t+2} - r(\pi) + ... = \sum_{t=0}^{\infty} R_t - r(\pi) \tag{2.18}$$

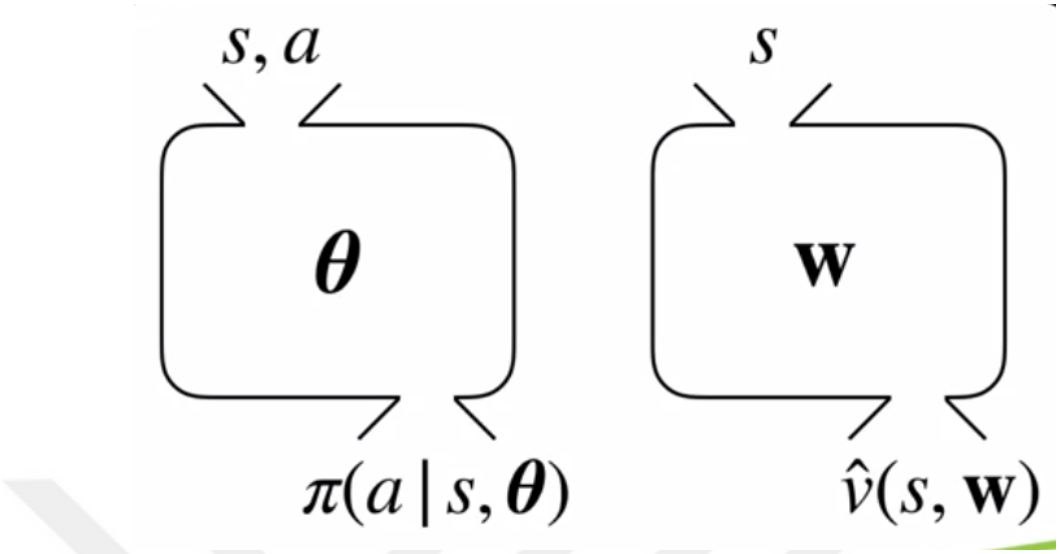


Figure 7: Approximate solution methods versus Policy gradient methods

We can express the average reward achieved by a policy π using the distribution $\mu(s)$ which represents the fraction of time the agent spent in state s under that policy as:

$$r(\pi) = \sum_{s} \mu(s) \sum_{a} \pi(a|s,\theta) \sum_{s',r} P(s',r|s,a)r \tag{2.19}$$

And the expected reward following policy π would be:

$$\mathbb{J}(\theta) = E_{\pi}[r(\pi)] \tag{2.20}$$

The goal in policy gradient methods is to find a policy which maximize the average reward in equation 2.19, this is achieved by estimating the gradient of the objective with respect to the policy parameters θ and adjust the parameters based on this estimate:

$$\begin{aligned}\theta_{t+1} &= \theta_t + \alpha \nabla \mathbb{J}(\theta_t) \\ &= \theta_t + \alpha \nabla \sum_{s} \mu(s) \sum_{a} \pi(a|s_t,\theta_t) \sum_{s',r} P(s',r|s_t,a)r\end{aligned}$$

However, $\mu(s)$ depends on a long-term interaction between the policy and the environment which makes calculating $\nabla\mu(s)$ a challenging task, luckily, the *Gradient*

Policy Theorem gives a simple formulation as (the proof of the theorem in [18]):

$$\begin{aligned}\theta_{t+1} &= \theta_t + \alpha \sum_s \mu(s) \sum_a \nabla\pi(a|s_t, \theta_t) G_t \\ &= \theta_t + \alpha \sum_a \nabla\pi(a|s_t, \theta_t) G_t\end{aligned} \tag{2.21}$$

We can further make the update equation above more compact by using the natural logarithm of π, where calculating the gradient of logarithm is always easier [19]:

$$\theta_{t+1} = \theta_t + \alpha \nabla \ln \pi(a_t|s_t, \theta_t) G_t \tag{2.22}$$

Algorithm.9 represents a policy gradient method using Monte Carlo algorithm to generate experience.

Algorithm 9: Monte-Carlo Policy-Gradient Control (episodic) for π_* [1]

Input:

Step size $\alpha \in (0, 1]$

A differentiable policy parameterization $\pi(a|s, \theta)$

1. **Initialization**

   Policy parameter $\theta \in \mathbb{R}$ (e.g., to 0)

2. **repeat** for each episode:

   Generate an episode

   $s_0, a_0, R_1, s_1, a_1, R_2, ..., s_{T-1}, a_{T-1}, R_T$ following policy $\pi(.|., \theta)$

   foreach *step in episode, t= 0, 1, 2,..., T-1* **do**

   $G \longleftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$

   $\theta = \theta + \alpha \nabla \ln \pi(a_t|s_t, \theta_t) \gamma^t G$

   until *forever*

2.6.5 Actor-critic approach

As the name suggests, The actor-critic RL approach combines two components, the actor and the critic:

- *Actor*: The actor selects the agent's actions at each time step, and it's responsible for forming the policy followed by the agent. As outlined in section 2.6.4 at each time step t, the actor takes state s_t and action a_t to return the probability $\pi(a|s,\theta)$ and then choose the action with the highest probability, this accomplished as in illustrated in equation 2.22:

$$\theta_{t+1} = \theta_t + \alpha \nabla \ln \pi(a_t|s_t,\theta_t) G_t$$

- *Critic*: The critic's role is to evaluate the action taken by the actor. So the critic approximates the value function using the current state s_t and the actor's action a_t as input, i.e., calculating a ''quality score" for the action given by the actor. Usually, the critic is one of the approximate function methods discussed in section 2.6.3.

The actor-critic approach is gradually adjusting the policy parameters θ of the actor to take actions that maximize the total reward predicted by the critic, so the first step in this optimization problem is to calculate the error between the predicted and actual reward as follows:

- In equation 2.22, G_t is the total expected reward predicted by the critic and can be expressed in an discounted way using a function approximation $\hat{V}(s,w)$ as outlined in section 2.6.3:

$$G_t = R_{t+1} + \gamma \hat{V}(s_{t+1}, w)$$

- In a similar way, we could write in terms of average reward:

$$G_t = R_{t+1} - \bar{R} + \hat{V}(s_{t+1}, w)$$

- And the critic calculates the prediction error as:

$$\delta = R_{t+1} + \gamma \hat{V}(s_{t+1}, w) - \hat{V}(s_t, w) \tag{2.23}$$

This is called the TD error, which needs to be minimized by adjusting the weights (W) of the value function approximator. The value function estimation of the current state $\hat{V}(s_t, w)$ is added as a baseline to make the learning faster.

After executing the action, we use the TD error to estimate how good the action was compared to the expected reward for that state. If TD error is positive, that means the action resulted in a higher value than expected, and taking this action more often will lead to better policy, and the opposite is true if the TD error is negative. The parameter θ of the actor is being adjusted in the way of maximizing the total future reward from equations 2.22 and 2.23:

$$\theta_{t+1} = \theta_t + \alpha \nabla \ln \pi(a_t|s_t, \theta_t)(R_{t+1} + \gamma \hat{V}(s_{t+1}, w) - \hat{V}(s_t, w)) \tag{2.24}$$

Fig.8 shows the whole process of actor-critic approach, where two updates are performed in alternating manner.

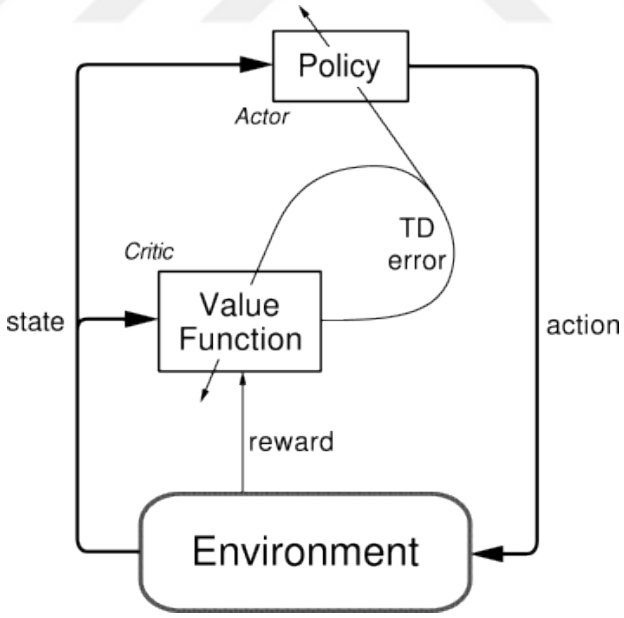


Figure 8: The interaction between actor and critic

An example of such an actor-critic algorithm is the *Deep Deterministic Policy Gradient* (DDPG) algorithm [20], which combines the DQL algorithm (Algorithm.8)

with the policy gradient approach to learn a deterministic policy, the algorithm is classified as model-free, off-policy actor-critic algorithm. (Algorithm.10)

Algorithm 10: Deep Deterministic Policy Gradient

1. **Initialization**

Critic network $Q(s, a|w)$ and actor $\pi(s|\theta)$, randomly, with weights W and θ.
Target network Q' and π' with weights $W' \longleftarrow W, \theta' \longleftarrow \theta$
Replay buffer $\mathcal{D}$

2. **repeat** for each episode:

Initialize a random process N for action exploration
Receive initial observation state **s**
foreach *step in episode, t=1, 2, 3,.., T* **do**
Choose action $\mathbf{a}_t = \pi(s_t|\theta) + N_t$ according to the current policy and exploration noise
Take action $\mathbf{a}_t$, observe r_t and $\mathbf{s}_{t+1}$
Store transition (s_t, a_t, r_t, s_{t+1}) in $\mathcal{D}$
Randomly select a minibatch of N transition (s_t, a_t, r_t, s_{t+1}) from $\mathcal{D}$
Set $y_i = r_i + \gamma Q'(s_{i+1}, \pi'(s_{i+1}|\theta')|W')$
Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|W))^2$
Update the actor policy using the sampled policy gradient:

$$\nabla J \approx \frac{1}{N} \sum_i \nabla Q(s, a|W) \nabla \pi(s|\theta)$$

Update the target networks:

$$W' \longleftarrow \alpha W + (1 - \alpha) W'$$

$$\pi' \longleftarrow \alpha \pi + (1 - \alpha) \pi'$$

Where $\alpha << 1$

until *forever*

CHAPTER III

RELATED WORK

The integration of Deep Learning (DL) with Reinforcement Learning has revolutionized the field of AI by delivering Deep Reinforcement Learning (DRL) algorithms capable of scaling to previously intractable problems. *Mnih et al., (2013)* [15] is considered the father of DRL, where he trained an agent of Deep Q-network (DQN) to play Atari game, where pixels of the game screen was the input data (state), and the directions of the joystick were actions. He proved that DRL had outperformed all existing algorithms in 2015 [21]. After that, many researchers build worked on improving the DQN algorithm. To solve the deviation problem in DQN, *Van Hasselt et al. (2015)* [22] proposed to use two networks instead, one Q-network to choose the action and the other to evaluate the action taken; they called it Double-DQN. *Lillicrap et al., (2018)* [23] build on the top on Double-DQN, an algorithm based on the deterministic policy gradient (DDPG) that can operate over continuous action spaces. The Twin Delayed Deep Deterministic Policy Gradient (TD3) algorithm was proposed by *Fujimoto et al., (2019)* [24] tackled the problem of the approximation error in DDPG.

Supervised learning is the most common approach used in almost all the studies that apply Machine learning in any area of finance. A vast number of research papers has been published to predict the future price of an asset [25],[26],[27],[28]. *Vargas et al., (2018)* [29] proposed a deep learning approach to predict the trend direction of a stock price utilizing financial news titles and technical indicators as input features to a hybrid model combining Convolutional Neural Network (CNN) and Recurrent Neural Networks (RNN). The prediction of their hybrid model is then fed to a trading

agent that buys or sells stock on the next day. The study demonstrated the significant role of financial news in stabilizing the predictions.

Many other studies investigated the other part of the trading problem by applying heuristics approaches to solve the Portfolio Optimization Problem(POP). *Chang et al., (2000)* [30] used three different metaheuristic approaches (Genetic Algorithm, Simulated Annealing, and Tabu Search) to solve a multi-objective portfolio optimization problem with cardinality and quantity constraints. They reported that no individual heuristic was found to be dominating across five data sets. *Schaerf et al. (2002)* [31] explored the use of Tabu Search (TS) on single-objective constrained POP. Also, they proposed new algorithms that combine different neighborhood relations. They implemented several token-ring procedures to diversify and intensify the search space and concluded that TS is the most promising heuristics to solve POP.

The application of Reinforcement Learning in finance has gained much interest over the last few years due to its ability to tackle the issues usually Supervised learning suffers from. *Bertoluzzo and Corazza (2012)* [32] investigated the performance of different RL algorithms in day-trading on one selected Italian stock. More specifically they compared the performance of Q-learning and Kernel-based reinforcement learning, concluding that Q-learning performance outperformed Kernal-based RL. In a subsequent study (2014) [33], they explored the effect of different reward functions such as Sharpe ratio, average log return, and OVER ratio on the performance of Q-learning in trading six selected Italian stocks reporting that lagged return reward function has the best performance. To enhance existing trading strategies *Tan et al. (2011)* [34] developed an RL-based trading system by employing Adaptive Network Fuzzy Inference System (ANFIS) to predict inflection points in the cyclic movements of stock prices, i.e., to detect entering and exiting points. RL is used here as a trading agent to execute the buy/sell actions as closely as possible to the inflection points.

When back-tested on five selected US stocks, they reported an outperforming performance by 50 percentage points than the market performance. Other studies that also applied RL to enhance and fine-tune existing trading strategies are *Kitchin, (1923)* [35], *Sarlan (2001)* [36]. Instead of approximating a value function (critic-only), *Deng et al., (2017)* [37] made one of the first attempts on combining Deep Learning with Recurrent Reinforcement Learning to directly approximate a policy function, this approach is called "deep recurrent reinforcement learning" (DRRL). In their approach, first, the DL part extracts 45 useful features from the market to be used as state representation in the environment. Secondly, they use a Recurrent Neural Network (RNN) as a trading agent to interact with deep-generated state features and make decisions.

To investigate the potential advantage of Actor-Critic methods in solving the day trading problem, *Conegundes and Pereira* (2020) [38] used Deep Deterministic Policy Gradient (DDPG) algorithm to solve the asset allocation problem. Considering liquidity, latency, slippage, and transaction cost constraints, they back-tested their approach on different datasets from the Brazilian Stock Exchange. They showed that their approach successfully obtained 311% cumulative return in three years with an annual average maximum drawdown of around 19%.

One of the most important studies in the field of RL in finance and had inspired this thesis is a study by *Yang et al., (2020)* [39]. Their research paper used a continuous action space to model the trading of 30 stocks, representing the state as a 181-dimensional vector. This state vector includes; available balance, close price, Moving Average Convergence Divergence. (MACD), Relative Strength Index (RSI), Commodity Channel Index (CCI), and Average Directional Index (ADX). They tested and reported the performance of three different critic-actor based algorithms, Advantage Actor-Critic (A2C), Deep Deterministic Policy Gradient (DDPG), and they used an Ensemble method of these three algorithms in order to adjust to different market

situations and select the best performing agent to trade based on the Sharpe ratio. Their ensemble strategy outperformed the three individual algorithms, the Dow Jones Industrial Average and min-variance portfolio allocation the method in terms of Sharpe ratio by balancing risk and return under transaction costs.

CHAPTER IV

METHODS AND IMPLEMENTATION

4.1 Problem Definition

As explained in chapter 2, to train a deep RL agent, a well formulation of the stock trading problem is required. Here, the stock trading problem is being modeled as *Markov Decision Process* (MDP), which can be formulated by describing its State Space, Action Space, and Reward Function. The MDP model of the problem is referred to as the ***Environment***. The proposed environment (MDP model) is built to carefully mimic the real world trading process and it can be characterized as:

- *Stochastic environment*: we can not always determine the next state of the trading environment from the current state by performing a buy, hold, or sell action because the nature of the trading markets is random and can not be entirely determined by an agent.

- *Partially Observable Environment*: no matter how complete the state representation is, it is impossible to obtain a real state space of the financial world with all features impacting the market movement, mainly that the environment (market movement) includes other agents (traders) which we can not observe. *Partially Observable MDP* (POMDP) is a generalization of MDP with a simple changing of terminology, where unlike the policy function in MDP, which maps the underlying states to the actions, POMDP's policy is a mapping from the observations (or belief states) to the actions [40].

- *Continuous Environment*: the problem at hand is considered continuous as there is an infinite number of states and actions that can be performed.

- *Sequential Environment*: the current agent's actions could affect the future actions and states; therefore, the problem is described as sequential rather than episodic.

- *Dynamic Environment*: the stock market (environment) is time-dependent, and it keeps changing while the agent perceives the state or whether it took action or not.

- *Single Agent Environment:* there is only one single agent that interacts and explores the environment in the proposed approach.

4.1.1 State Space

The state-space in the proposed environment is designed in a way to support multiple and single stock trading by representing the state as (1+ 13 x $\mathcal{N}$)-dimensional vector where $\mathcal{N}$ is the set of assets we consider in the market. Hence the state space increases linearly with the number of assets available to be traded.

There are two main parts of the state representation; the first part is the *Position State* $\in \mathbb{R}_{+}^{1+\mathcal{N}}$, which holds the current cash balance and shares owned of each asset in the portfolio, and the second part of the state is the *Market Signals* $\in \mathbb{R}^{12\times\mathcal{N}}$, which contains the necessary market features for each asset as a tuple, these features are the required information provided to the agent to make predictions of the market movement. The first type of information is based on the hypothesis of *technical analysis* [41] which states that the future behavior of financial markets is conditioned on its past. Hence, technical indicators are being used in the state space to help the agent interpret market behavior. The second type of information is based on what is called *fundamental analysis* [42] which studies everything from the overall economy and industry conditions to news releases, therefor a Natural Language Processing (NLP) approach is used to measure the overall sentiment from the news releases and integrate it with the state representation. The state (observation) vector at each time

step is provided to the agent as follows:

$$\mathbf{S_t} = [[\mathbf{b_t}, \mathbf{h_t}], [\{(\mathbf{C_t^i}, \mathbf{SS_t^i}, \mathbf{T_t^i}) | \mathbf{i} \in \mathcal{N}\}]]$$

Each component of the state space is defined as follows:

- $\mathcal{N} \in \mathbb{Z}_+^{\mathcal{N}}$: Number of assets in the portfolio.
- $\mathbf{b_t} \in \mathbb{R}_+$: The available cash balance in the portfolio at time step t.
- $\mathbf{h_t} = \{h_t^i | i \in \mathcal{N}\} = \{h_t^0, h_t^1, ..., h_t^{\mathcal{N}}\} \in \mathbb{Z}_+^{\mathcal{N}}$: The number of shares owned for each asset i in $\mathcal{N}$ at time step t.
- $\mathbf{C_t^i} \in \mathbb{R}_+^{\mathcal{N}}$: The close price of asset i in $\mathcal{N}$ at time step t.
- $\mathbf{SS_t^i} \in (-1, 0, 1)$: An integer 1, 0 or -1 to indicate the sentiment of the news related to stock i at time step t.
- $\mathbf{T_t^i}$: The 10 different *Technical Indicators* vector for asset i in the portfolio at time step t using the past prices of the asset in a specified look-back window $\mathcal{W} = 14$ (most common window is 14 or 9).

To demonstrate the state space, let's assume that we have 3 different assets ($\mathcal{N} = 3$) in the trading environment and an initial capital of 1000$ to be invested, the state vector would be a 40-dimensional vector and the *initial state(s_0)* given by the environment would be:

$$s_0 = [[1000, 0, 0, 0][(p_0^1, SS_0^1, T_0^1), (p_0^2, SS_0^2, T_0^2), (p_0^3, SS_0^3, T_0^3)]]$$

4.1.2 Action Space

The majority of the studies that applied RL in finance and particularly in trading markets utilized discrete action spaces to present the agent's decisions regarding the re-allocations of the portfolio's position, i.e., the agent's decision would be either hold,

buy or sell a fixed number of shares. In this study, a continuous action space approach is adopted to give the trading agent the ability to gradually adjust the portfolio's positions with each time step (dynamically re-allocate investments). From this perspective, the designed agent in this study receives the state s_t at each time step t as input and sends back action in the range between 1 and -1 inclusive, $a_t \in [-1, 1]$, the action then is re-scaled using a constrain K_{max} which represents the maximum allocation (buy/sell shares), transforming a_t to an integer $K \in [-K_{max}, ..., -1, 0, 1, ..., K_{max}]$ which stands for the number of shares to be executed, resulting in decreasing, increasing or holding of the current position of the corresponding asset [43]. There are two essential conditions regarding the action execution in our approach:

- If the current capital (cash) in the portfolio is insufficient to execute the buy action, the action will be partially executed with what the current capital can buy of the requested stock.
- If the number of shares for a specific asset (h_t^i) in the portfolio is less than the number of shares to be sold ($a_t^i \in \mathbb{Z}^-$), the agent will sell all the remaining shares for this asset.

We can mathematically express the action space as the following:

$$
\begin{aligned}
A_t = \{a_t^i | i \in \mathcal{N}\} = \{a_t^0, a_t^1, ..., a_t^{\mathcal{N}}\} \\
S.t. \\
a_t^i \in \mathbb{Z}^{\mathcal{N}} \\
-K_{max} \leq a_t^i \leq K_{max},\ \forall i \in \mathcal{N} \\
a_t^i = h_t^i \ \ if\ |a_t^i| > h_t^i\,,\ \forall a_t \in \mathbb{Z}^-
\end{aligned}
\tag{4.25}
$$

Where:

* $\mathcal{N}$: assets in the portfolio.
* A_t: the action vector sent by the agent to the environment.
* a_t^i: the action (number of shares) to buy/sell for asset i at time step t.
* K_{max}: the maximum number of shares the agent can re-allocate of an individual asset at each time step t.

* h_t^i: the portfolio position (number of shares) of asset i at time step t.

As in this study we are dealing with multi-stock trading problem, the action space depends on the number of assets available in the portfolio $\mathcal{N}$ and it's given as $(2 \times K_{max} + 1)^{\mathcal{N}}$, hence the action space increases exponentially by increasing $\mathcal{N}$.

4.1.3 Reward Function

An appropriate design of the reward function is essential in the RL problem to provide accurate feedback to the agent on the quality of its action. Therefore, we need to accurately represent the goal of our problem to the agent to guarantee a smooth convergence to the local optimum solution.[44], [38]. This study aims to design a trading strategy that maximizes the investment return at a target time T_f; we denote the final investment return as G. In this study, the immediate reward $r(s, a, s')$ received by the agent after each action is the difference between the portfolio value $\mathcal{V}_t$ calculated at the end of period t and the value at the end of previous period $t-1$.

$$r(s, a, s') = \mathcal{V}_t - \mathcal{V}_{t-1} \tag{4.26}$$

Where the portfolio value $\mathcal{V}$ at each time step is calculated as:

$$\mathcal{V}_t = b_t + h_t.C_t \tag{4.27}$$

Where:

* b_t: the available cash balance in the portfolio at time step t.

* $h_t = \{h_t^i | i \in \mathcal{N}\}$: the position vector (number of shares of each asset) at time step step t.

* $C_t = \{C_t^i | i \in \mathcal{N}\}$: the closing price of each asset in the portfolio at time step t.

To better simulate the real-world trading process in the stock market, transaction costs (i.e., commission fees) are incorporated into the immediate reward ($r(s, a, s')$)

calculation. The transition cost can be represented in many different ways in real life, and it varies from one broker to another. In this study, we set the commission as a fixed percentage of the total closed deal amount, where d_{buy} represents the commission percentage when buying is performed and d_{sell} is the commission percentage for selling (see section.4.5.2 for the numeric values):

$$d_t = \{d_t^i | i \in \mathcal{N}\} = [d_t^0, d_t^1, ..., d_t^{\mathcal{N}}]$$

$$where : d_t^i = \begin{cases} d_{buy}, & \text{if } a_t^i > 0 \\ 0, & \text{if } a_t^i = 0 \\ d_{sell}, & \text{if } a_t^i < 0 \end{cases}$$

The commission vector d_t is incorporated into the immediate reward function by excluding the commission amount paid from the portfolio value calculated in equation 4.27) so the agent would avoid excessive trading that results in a high commission rate and, therefore, a negative reward:

$$\mathcal{V}_t = b_t + h_t.C_t - h_t.(C_{t-1} \circ d_t) \tag{4.28}$$

In the above equation, the amount paid for the commission is calculated by taking the Hadamard product of the commission vector d_t and the closing price of the previous period C_{t-1}, that is because the action of buying/selling occurred in the previous state. Therefore commission should be calculated using the closing prices in that state.

And the final goal of maximizing the investment return is boiled down to maximizing the individual returns after each action, where:

$$G = r_{t+1} + r_{t+2} + ...+$$

4.1.4 Environment constraints and assumptions

The following constraints and assumptions were imposed on the MDP environment for two main reasons. First, to idealize and simplify the complex financial market systems (e.g., via liquidity assumption) without losing the nature of the problem. The second reason is to make the model closer to a real-world situation.

I. **Non-Negative Balance Constraint**: The cash balance in any state is not allowed to be negative, $b_t > 0$. Therefore, the actions should not result in a negative cash balance, to achieve that, the environment prioritize the execution of sell actions ($a_t < 0$) in the action vector A_t (equation 4.25) to guarantee cash liquidity in the portfolio so buy actions ($a_t > 0$) would be fulfilled afterward. If the buy action still results in a negative balance (i.e., not enough cash to fulfill the action), it is fulfilled partially with what remains in the portfolio's cash balance.

II. **Short-Selling Constraint**: Short selling is prohibited in the designed environment, all portfolio's positions must be strictly non-negative:

$$\mathbf{h_t} = \{h_t^i | i \in \mathcal{N}\} = \{h_t^0, h_t^1, ..., h_t^{\mathcal{N}}\} \in \mathbb{Z}_+^{\mathcal{N}}$$

III. **Zero Slippage Assumption**: When the market volatility is high, slippage occurs between the price at which the trade was ordered and the price at which it has completed [45]. In this study, the market liquidity is assumed high enough to complete the trade at the same price when it was ordered [46]. This assumption is valid primarily in a real-world trading environment when trading in big stock markets.

IV. **Zero Market Impact**: In financial markets, a market participant has an impact on the market when it buys or sells an asset which causes the price change. The impact provoked by the agent in this study is assumed to have no impact

on the market when it performs its actions. This assumption is primarily valid even in real-life trading when the market volume is big enough to make the individual investment insignificant [46].

To remove any confusion, we explain the dynamic of the MDP environment designed in the study through the following example:

Environment configuration:

- Initial Capital (b_0) = 1000$
- Stocks to consider ($\mathcal{N}$) = $[AAPL, AMZN, FB]$
- Buy commission (d_{buy}) = 0.005%
- Sell commission (d_{sell}) = 0.001%
- Action upper bound (K_{max}) = 100
- Time-step (t) = day
- look-back window period ($\mathcal{W}$) = 14t (14 days)

Initial state $S_{t_0} = [[1000, 0, 0, 0], [(20.04, SS, T)_{AAPL}, (218.11, SS, T)_{AMZN}, (34.03, SS, T)_{FB}]$

Action generated by the agent A_{t_0} = [+20, +30, +10]

Next state $S_{t_1} = [[21.88, 20, 2, 4], [(19.9, SS, T)_{AAPL}, (215.33, SS, T)_{AMZN}, (31, SS, T)_{FB}]$

Notice that the number of shares bough for AMZN and FB are less than the generated action (+30 and +10) that is because of insufficient cash (capital) after fulfilling the first action (+20), therefore the action is partially fulfilled (2 and 4).

Immediate reward $r(S_{t_0}, A_{t_0}, S_{t_1}) = \mathcal{V}_{t_1} - \mathcal{V}_{t_0}$ = -25

Again,

Action generated by the agent A_{t_1} = [-10, -20, 0]

Next state $S_{t_2} = [[650.83, 10, 0, 4], [(20.3, SS, T)_{AAPL}, (217.3, SS, T)_{AMZN}, (32, SS, T)_{FB}]$

Immediate reward $r(S_{t_1}, A_{t_1}, S_{t_2}) = \mathcal{V}_{t_2} - \mathcal{V}_{t_1} = 8$

Repeat until final state t_f

4.2 Technical Indicators

We used the 10 most famous indicators used by technical traders when trading in the stock market [41] with a look-back window $\mathcal{W} = 14$ we describe them as follows:

I. **Relative Strength Index (RSI)** $\in \mathbb{R}_+^{\mathcal{N}}$: a momentum indicator to measure the magnitude of recent price changes to identify overbought or oversold conditions in the stock price. In this environment, RSI is calculated using closing prices in a given look-back window by taking the ratio of the average gain of up periods to the average loss of down periods during the look-back window:

$$RSI_{\mathcal{W}} = 100 - \frac{100}{1 + \frac{\text{Average of gains}}{\text{Average of loss}}}$$

The RSI can have a reading from 0 to 100 inclusively. When RSI is above the centerline, it contributes more to the buy action of the agent, but when it's below it contributes to the sell action [47].

II. **Simple Moving Average (SMA)** $\in \mathbb{R}_+^{\mathcal{N}}$:an important indicator to identify current price trends and the potential for a change in an established trend. SMA is calculated by taking the average of closing prices in a given look-back window $\mathcal{W}$.

$$SMA_{\mathcal{W}} = \frac{\sum_{i=0}^{i=\mathcal{W}} ClosingPrice_i}{\mathcal{W}}$$

III. **Exponential Moving Average (EMA)** $\in \mathbb{R}_+^{\mathcal{N}}$:Like SMA, EMA is a technical indicator used to spot current trends over time. However, EMA is considered an improved version of SMA by giving more weight to the recent prices considering old price history less relevant; therefore, it responds more quickly to price changes than SMA. This indicator is being calculated as follows:

$$EMA_t = \text{close price}_t \times Multiplier + EMA_{t-1} \times (1 - Multiplier)$$

SMA for the given $\mathcal{W}$ is calculated as the initial EMA_{t0} value, and the weighted multiplier is calculated using a look-back window as:

$$Multiplier = \frac{2}{1 + \mathcal{W}}$$

IV. **Stochastic Oscillator (%K)** $\in \mathbb{R}_{+}^{\mathcal{N}}$: it's a momentum indicator comparing the closing price of the stock to a range of its prices in a look-back window period $\mathcal{W}$. %K indicator is used to generate overbought and oversold trading signals to be utilized by the agent, and it's bounded in between 0-100 range:

$$\%K = 100 \times \frac{\text{close price} - \text{lowest low over } \mathcal{W}}{\text{highest high over } \mathcal{W} - \text{lowest low over } \mathcal{W}}$$

V. **Moving Average Convergence/Divergence (MACD)** $\in \mathbb{R}^{\mathcal{N}}$: is one of the most used momentum indicators to identify the relationship between two moving averages of the stock price, and it helps the agent to understand whether the bullish or bearish movement in the price is strengthening or weakening [47]. MACD is calculated by subtracting a long-period EMA from a short-period one. In this study, the short-period is considered the look-back window $\mathcal{W}$ and the long period is calculated as $(\mathcal{W} \times 2) + 1$

$$MACD = EMA_{(long\_period)} - EMA_{(Short\_period)}$$

VI. **Accumulation/Distribution Oscillator (A/D)** $\in \mathbb{R}^{\mathcal{N}}$: a volume-based cumulative momentum indicator helps the agent assess whether the stock is being accumulated (bought) or distributed (sold) by measuring the divergences between the volume flow and the stock price. For example, if the price of a certain stock is moving up, but the A/D indicator is falling, the agent would get the insight that buying the stock might not be a good idea because there is not enough volume to support the rising price. The A/D is calculated in the study as follows:

$$Multiplier = \frac{closeprice - lowprice}{highprice - lowprice}$$

$$\text{money flow volume} = Multiplier \times volume$$

$$A/D = A/D_{prev} + \text{money flow volume}$$

VII. **On-Balance Volume Indicator (OBV)** $\in \mathbb{R}^{\mathcal{N}}$: another volume-based momentum indicator that uses volume flow to predict the changes in stock price [48]:

$$OBV = OBV_{prev} + \begin{cases} Volume, & \text{if close price} > \text{close price}_{prev} \\ 0, & \text{if close price} = \text{close price}_{prev} \\ -Volume, & \text{if close price} < \text{close price}_{prev} \end{cases}$$

VIII. **Price Rate Of Change (ROC)** $\in \mathbb{R}^{\mathcal{N}}$: a momentum-based indicator that measures the speed of stock price changes over the look-back window $\mathcal{W}$. A rising ROC above zero would indicate an uptrend, while a falling ROC below zero indicates a downtrend:

$$ROC = 100 * \frac{\text{close price}_t - \text{close price}_{t-\mathcal{W}}}{\text{close price}_{t-\mathcal{W}}}$$

IX. **William's %R** $\in \mathbb{R}_{+}^{\mathcal{N}}$: also known as Williams Percent Range, is a momentum indicator used to spot entry and exit points in the market by comparing the closing price of the stock to the high-low range of prices in the look-back window ($\mathcal{W}$):

$$Williams\%R = -100 \times \frac{\text{highest high over } \mathcal{W} - \text{closing price}}{\text{highest high over } \mathcal{W} - \text{lowest low over } \mathcal{W}}$$

X. **Disparity Index** $\in \mathbb{R}_{+}^{\mathcal{N}}$: its value is a percentage that indicates the relative position of the current closing price of the stock to a selected moving average. In this study, the selected moving average is the EMA of the look-back window ($\mathcal{W}$):

$$Disparity = \frac{\text{closing price} - EMA(\mathcal{W})}{EMA(\mathcal{W}) * 100}$$

All technical indicators are calculated at the end of the time step, including the data from the current time step, i.e., if we are using a daily period, technical indicators are being calculated at the end of the trading day t including the prices of the current day t and given to the agent.

4.3 Sentiment Analysis Score

The supply and demand fluctuations in the stock market are highly sensitive to the news of the moment due to the impact of mass media on the investor's behavior. Hence many traders and investors consider the news reports in their stock-picking strategy. In our proposed approach, we believe that incorporating the general news sentence towards the asset being considered in the observation (state) definition will help the agent learn a better trading strategy. In Ding et al. [49] study, they showed that news headlines are more useful in forecasting than using the full news article content. Therefore, we only consider news headlines as our input to calculate the sentiment score. We describe the process of calculating a sentiment score for each asset in the portfolio at time step t (day) as the following:

- The first step is to extract the target asset's news at day t. This step is essential as accurate extraction of events (news) directly related to the target asset directly influences the sentiment score. We use a rule-based matching approach to search for the asset name, stock symbol, or other keywords in the headline news (ex. Microsoft or MSFT, tech,..) released on day t.

- Bidirectional Encoder Representations from Transformers (BERT) [50] has achieved state-of-the-art performance in almost all language processing tasks, and the most critical advantage is that the pre-trained BERT model can be fine-tuned with just one additional output layer to adapt to a more specif task. We use a fine-tuned BERT model to calculate each news headline's sentiment probability (Positive, Negative, or Neutral). FinBERT model [51] which is a pre-trained NLP model to analyze sentiment specifically for financial text, where its base BERT model is trained on books and Wikipedia pages to capture general features of the language and then fine-tuned on *Financial PhraseBank* by Malo et al. [52] to adapt to the financial domain. We feed the news headline to the model to get the sentiment probability

of the input being positive, negative, and neutral; bellow is a sample of the model's output:

Date	Headline	Positive	Negative	Neutral
2010-08-31	A Cynical, Contrarian View on Microsoft	0.05	0.76	0.189
2010-08-31	SalesForce faces the Google/Microsoft Music	0.03	0.035	0.93
2010-10-29	Microsoft: Analysts' New Targets, Estimates	0.05	0.022	0.925
2012-02-01	MSFT Rises 5% on Earnings	0.93	0.025	0.045
2012-02-01	Is Microsoft Infiltrating Apple's World?	0.02	0.75	0.22

- We take the average of the asset's news sentiment probabilities for each day and assign 1 if the positive probability is higher than the negative probability and -1 otherwise. We ignore the neutral probability as we believe that if an asset has been mentioned on the news, it will impact the asset price (positively or negatively). If the asset has no news on a given day we assign 0 to the sentiment score, the above example becomes:

Date	Positive	Negative	Sentiment Score
2010-08-31	0.04	0.39	-1
2010-10-29	0.05	0.022	1
2012-02-01	0.95	0.775	1

- We repeat this process for each asset considered in the portfolio and align the sentiment score of the same day with the rest of the features.

4.4 The Learning Algorithm (TD3)

As we have discussed in chapter 2.6.3, not all algorithms in RL can handle continuous action spaces. *Actor-Critic* based algorithms were successfully capable of of solving the continuous action space utilizing function approximation and policy gradient methods (see

chapter 2.6.5). One of the most famous actor-critic, off-policy algorithms is the Deep Deterministic Policy Gradient algorithm (DDPG) (Algorithm 10). However, despite the great performance DDPG achieved in continuous control problems such as robotics and autonomous driving, it has a major drawback similar to many RL algorithms, which is the overestimation of action values ($\max_{\mathbf{a}} Q(s_{t+1}, a_{t+1})$) as a result of function approximation error. This overestimation bias is unavoidable in RL as we are using estimates instead of ground truth in the learning process, which is further amplified by the Temporal Difference (TD) nature, which uses these estimations in the value function updates to end up into a local optimum (sub-optimal policy) as result of the accumulated errors.

In this study, as our problem has a continuous space of actions, we use Twin Delayed Deep Deterministic Policy Gradient (TD3) [24] algorithm, which is a direct successor of DDPG but with improvements to tackle the aforementioned overestimation bias drawback. TD3 is capable of reducing the overestimation bias, thus reducing the accumulation of errors in the learning process by introducing three main components to DDPG:

- **Clipped Double Critic Networks:**The first component added is a novel clipped variant of Double Q-learning [22] to replace the single critic. Using two different and separate critic networks to make an independent estimate of the value function can be used to make unbiased estimates of the actions selected using the opposite value estimate. TD3 uses a clipped double Q-learning instead of the traditional one used in Double Q-learning where it takes the smallest value of the two critic networks estimates, that is, if we use the traditional Double Q-learning in actor-critic methods, the policy and target networks are updated so slowly that they make similar estimates and offered slight improvement.

- **Delayed Updates:** The second component is added to solve the residual error accumulation formed due to the learning process without a fixed target (estimates instead). In critic-actor methods, this accumulation of errors is amplified caused by the interaction between the policy (actor) and value (critic) networks, where the

policy gradient is maximized over the value estimate. Delaying the policy network update, i.e., updating it less frequently than the value network, allows the value network to stabilize before it can be used to update the policy gradient. This results in a lower variance of estimates and, therefore, better policy.

- **Target Policy Smoothing Regularization:** The final component is applying a regularization strategy to the target policy by adding a small random noise and averaging over mini-batches. This is important to reduce the variance of the target values when updating the critic, which causes by overfitting spikes in the value estimate.

Algorithm 11: Twin Delayed Deep Deterministic Policy Gradient (TD3) [24]

---

1. **Initialization**

Critic networks $Q(s, a|w_1)$, $Q(s, a|w_2)$ and actor $\pi(s|\theta)$, randomly, with weights W_1, W_2 and θ.
Target networks Q'_1, Q'_2 and π' with weights $W'_1 \longleftarrow W_1, W'_2 \longleftarrow W_2, \theta' \longleftarrow \theta$
Replay buffer $\mathcal{D}$

2. **foreach** *t=1 to T* **do**

Initialize a random process N for action exploration
Select action with exploration noise $a \sim \pi(s|\theta) + \epsilon, \epsilon \sim \mathcal{N}(0, \sigma)$
Observe reward r and next state s'
Store transition tuple (s, a, r, s') in $\mathcal{D}$
Sample mini-batch of N transitions (s, a, r, s') from $\mathcal{D}$
$\tilde{a} \leftarrow \pi(s'|\theta) + \epsilon, \epsilon \sim clip(\mathcal{N}(0, \tilde{\sigma}), -c, c)$
$y \leftarrow r + \gamma \min_{i=1,2} Q(s', \tilde{a}|w_i)$
Update critics $W_i \leftarrow \arg\min_{W_i} N^{-1} \sum (y - Q_{W_i}(s, a))^2$
if *t mode d* **then**

Update θ by the deterministic policy gradient:
$\nabla_\theta J(\theta) = N^{-1} \sum \nabla_a Q_{W_1}(s, a)|_{a=\pi_\theta(s)} \nabla_\theta \pi_\theta(s)$
Update target networks:
$W'_i \leftarrow \tau W_i + (1 - \tau) W'_i$
$\theta' \leftarrow \tau\theta + (1 - \tau)\theta'$

4.5 Details of Implementation

The study is implemented using Python3 [53], with Pandas [54], Numpy [55], Tensorflow [56] and other public source libraries. We employ OpenAi gym [57] framework to implement the proposed environment. The Twin Delayed DDPG (TD3) algorithm is implemented using StableBaselines3 library [58] which provides a set of reliable implementations of reinforcement learning algorithms in PyTorch [59]. The agent is trained on Intel(R) Core(TM) i7-6500U CPU @ 2.50GHz, 8.00 GB RAM and 64-bit operating system.

4.5.1 Data Description and Preprocessing

As our proposed approach is designed to generalize any financial data, we can apply various datasets to train, evaluate and test the model. In this work, to test our approach and compare to different benchmarks (chapter.5), we use Yahoo Finance [60] to retrieve historical market daily prices. The retrieved historical data consists of 7 columns; *Date, Volume, Open, Close, Adjusted Close, High* and *Low* prices. A sample of data is presented as follows:

Date	Open	High	Low	Close	Adj Close	Volume
2019-01-2	38.72	39.71	38.55	39.480	38.43	148158800
2019-01-03	35.99	36.43	35.5	35.54	35.61	365248800
2019-01-04	36.13	37.13	35.95	38.06	36.088	234428400
2019-01-07	37.17	37.20	36.47	36.98	36.00	219111200

To prepare each dataset to be used by the model, we first perform timestamps processing by using the trading calendar (exchange-calendars package [61]) to check if the market was open between the given dates to the agent and exclude weekends and holidays from the dataset so the agent will not face gaps in the trading process. Further processing

of the dataset is required to ensure that all financial assets (stocks) considered in the portfolio have an equal length of historical data points, where some stocks have been recorded for decades, other newly listed stocks are only a few months. This time-dimension alignment of stocks' historical data will prevent the bias action of the agent towards the stock with more data. For example, if the considered portfolio consists of company **A**'s stock which went public in 2007, and company **B**'s stock, which has been public since 1980, and the date given to the agent to start trading were in 2005, the dataset will include data points starting from 2007 where both stocks, **A** and **B**, have equal historical data points. Once we have the timestamps processed, we use **Close, High, Low** prices and **Volume** at each timestamp to calculate the technical indicators of each asset using their mathematical expressions explained in section 4.2 with a look-back window ($\mathcal{W}$) equals to 14 days.

To obtain comprehensive and accurate financial news, we combined headline news from *Benzinga, Seeking Alpha, Zacks* and other financial news websites [62], and crawled historical news headlines from *Reddit worldNews Channel* [63]. The final dataset consists of 3,288,724 news headlines ranging between 2009-2020, which we utilized to calculate the sentiment score of a given asset at each period (day) as dissuaded in section 4.3.

4.5.2 Experimental settings

In a real-life stock trading environment, traders usually follow three major trading styles; day trading, swing trading, or long-term trading. Although our model can support different trading styles, in this study, we will be focusing on long-term trading style as it is considered the most used one among the three styles mentioned above. Consequently, the environment follows a daily time-frame approach meaning that the environment will send the current observation (market status) at the end of the trading day to the agent, and it will execute the agent's decisions (buy/sell) on the opening of the next trading day taking into considerations the assumptions we have discussed in section 4.1.4.

Normalizing the action space is essential for the learning process of RL algorithms as they rely on a Gaussian distribution (initially centered at 0 with std 1) for continuous actions. To normalize our continuous action space, we set upper bound equals n (depends on the experiment) for the number of shares the agent can execute on one stock in a single trade, i.e., the agent can not sell/buy more than n shares of the same stock on the same day. This parameter plays a vital role in diversifying the portfolio. If n is small and we are considering many stocks to trade, the agent will learn to diversify the capital as much as possible to make profits. Additionally, the commission is a hyperparameter and has a significant impact on the agent's performance since higher commissions discourage the agent from trading, and we observe an opposite in the case of low commissions [6]. In this study, we set the commission as a fixed percentage from the total closed deal, where buying commission equals 0.005 and 0.001 for selling.

Another important key point that has a significant impact on the agent's performance is the scaling of the vector observations, the different components (price, technical indicators, sentiment score) of our state representation have different physical units, which makes it difficult for the agent's neural network to learn effectively and generalize across the environment. One of the most effective methods to scale the observation vector is the *batch normalization* [64], which uses mini-batches from the samples to have unit mean and variance and maintains a running moving average of the mean and variance to use for normalization during testing and validation, we utilize this method to normalize all values in our state representation.

4.5.3 Environment Rendering

. To visually assess the performance and decision made by the agent when trading assets, we developed a dashboard that shows the current portfolio's value along with the date and the open positions for each asset. Fig. 9 shows a screenshot of the dashboard when the agent is trading.

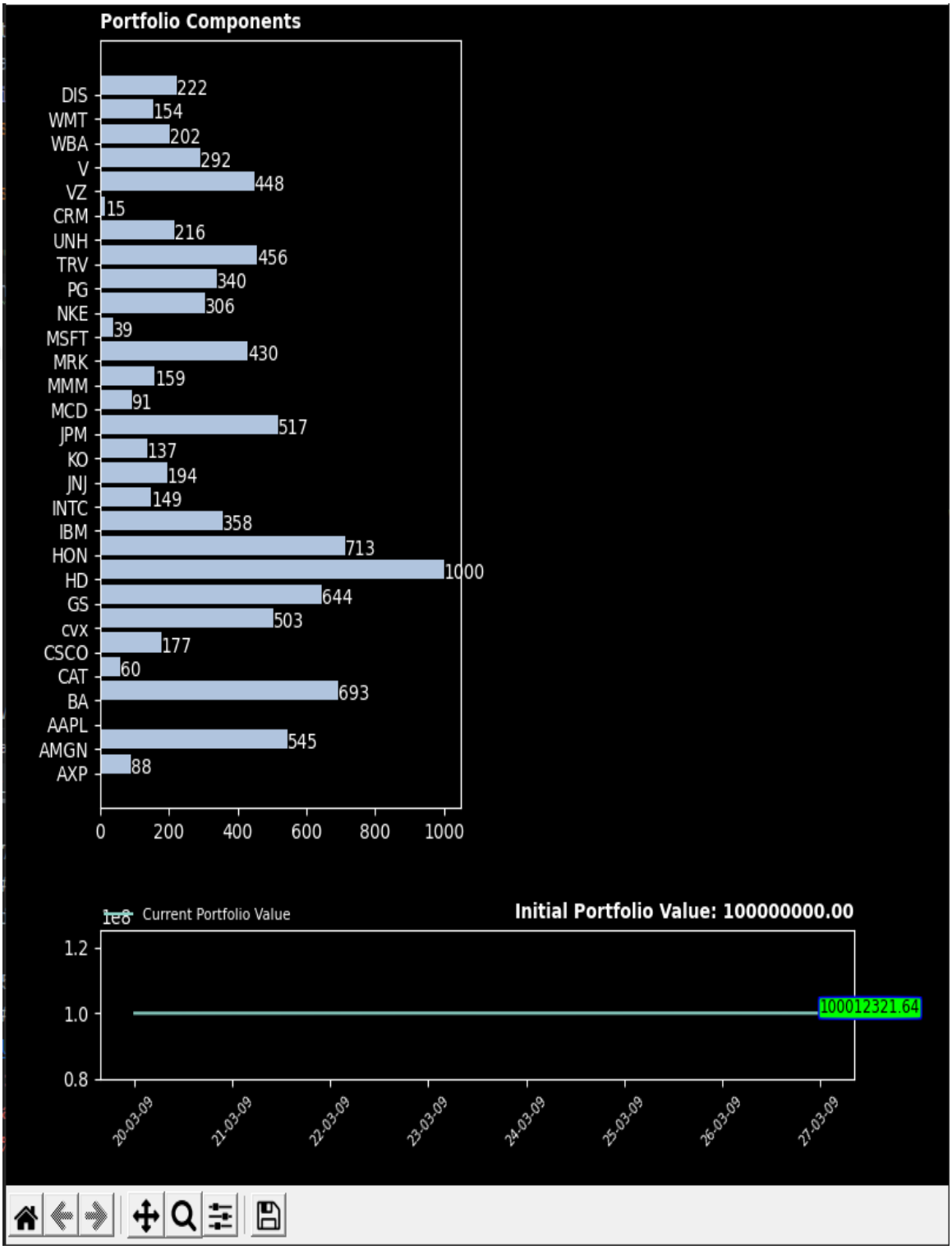


Figure 9: Environment Rendering

CHAPTER V

RESULTS AND DISCUSSION

In this chapter, we show the evaluation of our proposed approach by performing back-testing, which is the process used by traders and analysts to assert the viability of a trading strategy by testing it on historical data. We conduct two different back-testing experiments. The purpose of the first experiment (section 5.1) is to validate the superiority of the continuous action space to solve the trading problem over the discrete action space and to demonstrate how each component of the state representation in our approach contributes to the learning process of the agent. The second (section 5.2) experiment is conducted to validate the robustness of our model on an ample space of actions and states by considering 10 different assets in the portfolio also to evaluate the performance on an unseen market data to check the agent's ability of generalization.

5.1 First Experiment

In the first experiment we conduct three evaluations, each with exactly same configurations like the number of assets in the portfolio, initial capital, commission rates, etc. but with different components of the environment's state representation. We start by a baseline with only the close price as market signal feature (section 5.1.1), then we add technical indicators in the second evaluation (section 5.1.2), and finally we evaluate with adding sentiment analysis scores (section 5.1.3). In Table. 1 we summarize the three evaluations results of the experiment. The experiment's hyperparameters and settings are in Appendix.A and the learning curves of the agent in each evaluation are reported in Appendix.B

Due to the stochasticity in the learning process, the experiment results may change at each run depending on different factors such as the actions the agent randomly starts

with and use to explore or the random weight initialization. As suggested in [65] to ensure fairness and the reliability of our results, we average multiple runs over different random seeds to have an insight into the population distribution of the algorithm performance on an environment. In this experiment's evaluations, we report and highlight results across several independent runs. While the recommended number of trials to evaluate an RL algorithm is still an open question in the field, we reported the mean and standard error across 5 trials (runs), which is the suggested number in many studies [65]. Unless otherwise mentioned in the benchmark settings, we use default settings regarding the environment's parameters (e.g., commission rate and upper bound for actions).

We use two metrics to evaluate our results: the first metric is the *cumulative sum of reward*, i.e., the total profits at the end of the trading episode. The second metric is the annualized *Sharpe ratio* [66] that combines the return and the risk together to return the average of the risk free return by the portfolio's deviation:

$$\textit{Sharpe Ratio} = \frac{\text{return of portfolio} - \text{risk-free rate}}{\text{standard deviation of the portfolio's excess return}}$$

In general, a Sharpe ratio above 1.0 is considered to be *"good"* by investors because this suggests that the portfolio is offering excess returns relative to its volatility. A Sharpe Ratio higher than 2.0 is rated as *"very good"* where a ratio above 3.0 is considered *"excellent"*. Whereas a ratio under 1.0 is considered sub-optimal. Further, we observe the total amount of commission in an episode which gives an idea of the agent's trading behavior.

5.1.1 Evaluation on Baseline Environment

To evaluate the continuous action approach (see section 4.1.2) in our model, we test it by solving the problem with only the close price of the assets as a market signal. Hence the state representation in this baseline environment consists of only the *position state* and the close price of the asset at t (C_t) as a market signal, i.e., the agent will solely make its trading decision based on merely the closing price of the stock as a market feature (see

section 4.1.1 for the full description of state representation in our approach). We consider the results here as the baseline results that we use and other benchmarks to evaluate the usefulness of the addition of technical indicators and sentiment scores to the state representation in the following evaluations.

In Kaur's paper [67], the approach proposed is similar to ours. However, the paper follows a discrete action space where the agent can choose to buy, sell or hold action (i.e., discrete action space) of a fixed number of shares on each time step for a portfolio of two assets, namely; Qualcomm (QCOM) and Microsoft (MSFT). We back-test our approach on the same 5-years daily historical stock data (data between 2011-2016) used in their study with the same amount of initial capital ($10,000) and compared the annual Sharpe ratio. They reported a Sharpe ratio equal to 0.85 using only the closing price in the state representation. It is worth mentioning that the commission rate used in their study was not mentioned, so we set ours to default as discussed in 4.5.2.

We perform 5 experiment trials, each with 200 epochs (episodes) for the same hyper-parameter configuration, only varying the random seed across trials (See the experiment's hyperparameters in Appendix.A.1). Fig.10 shows the average return (sum of rewards) at each trading episode and the standard error across the 5 runs. As can be observed, the agent's performance increases with more experience it gains with the number of epochs to successfully achieve 33960$ average return (profits) with standard error equals to ± 4473\$. From the commission spent by the agent, we can conclude that the agent was successfully able to find a balanced trading strategy by balancing between trading and holding positions. Finally, the average annual Sharpe ratio of our approach on the baseline environment was 1.43 with a standard error of ± 0.13. This is significantly higher than the reported Sharpe ratio 0.85 in [67] benchmark, which indicates the advantage of continuous action space over the discrete space. See Appendix.B.1 for the learning curves.

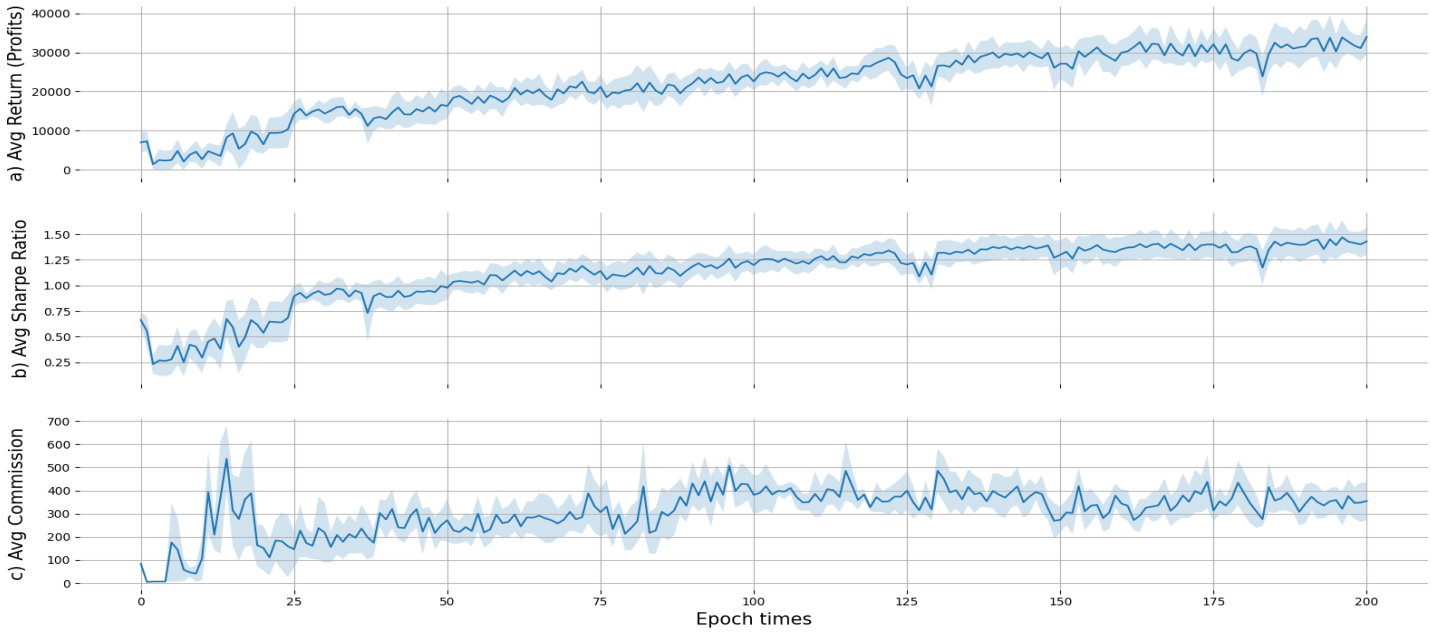


Figure 10: TD3 agent performance metrics on Baseline environment using the same hyperparameter configurations averaged over 5 different random seeds. **a)** Average return (Profits in dollars) at the end of each episode. **b)** The average annual Sharpe ratio at the end of each episode. **c)** The average amount of commission spent at the end of each episode

5.1.2 Evaluation on WithTechIndicators Environment

In this section, we show how introducing 10 technical indicators described in chapter 4.2 to the state representation improves the performance of the agent by providing it with a more in-depth knowledge of its environment. We refer to this environment with technical indicators and close price in the state representation as *WithTechIndicators* environment.

Using the same configurations used in baseline environment evaluation in section.5.1.1 we augment the state with technical indicators and run 5 independent experiments to report the average of return, Sharpe ratio, and commission. The results in Fig.11 demonstrate that augmenting the environment with technical indicators has brought more useful information to the agent to make better decisions. The agent successfully achieved 89782$

average return (profits) with ± 18980\$ standard error, and an average Sharpe ratio equals 2.75 with a standard error ± 0.43. We can also notice that the average amount of commission is almost two times the amount spent in the baseline environment, which means that the agent was significantly more active in buying/selling stocks and closed more successful deals. In addition, our approach outperformed the benchmark [67] reported Sharpe ratio of 1.4, where they have used market technical indicators and neural network to find trend information and predict the direction of the trend, 1 if an upward trend is observed and 0 otherwise, this prediction is then incorporated with the state representation.

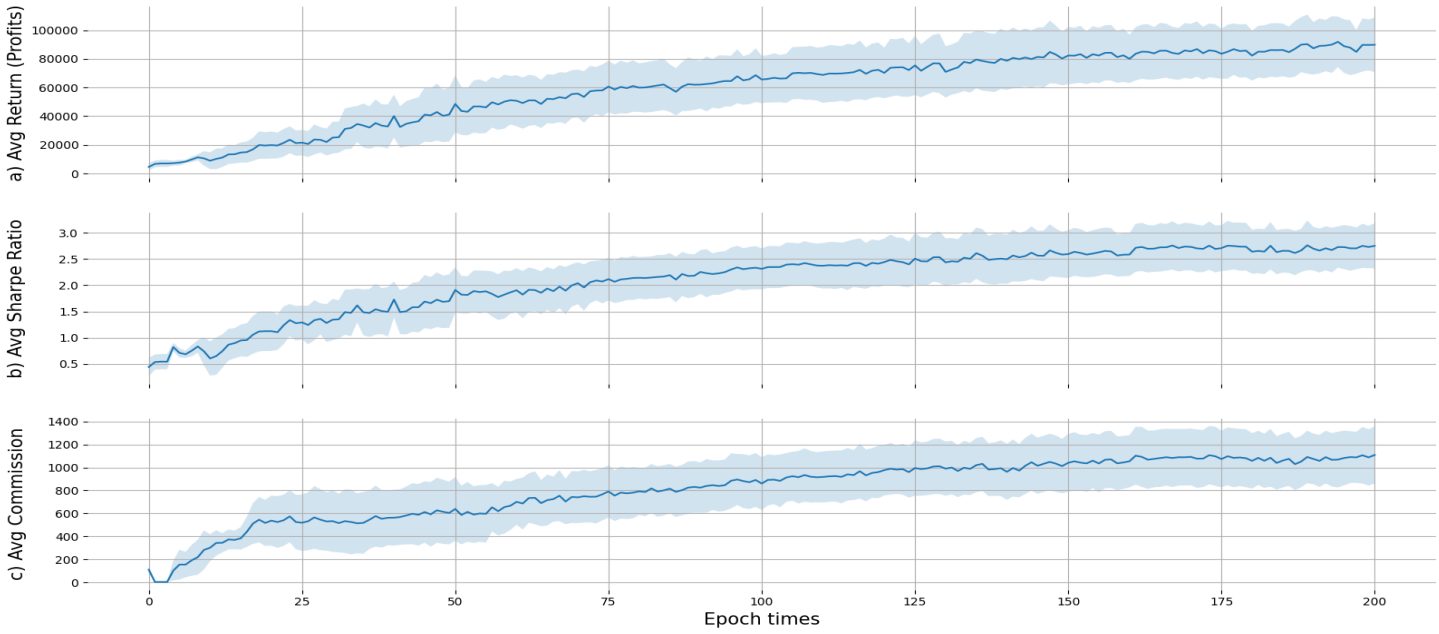


Figure 11: TD3 agent performance metrics on WithTechIndicators Environment using the same hyperparameter configurations averaged over 5 different random seeds. **a)** Average return (Profits in dollars) at the end of each episode. **b)** The average annual Sharpe ratio at the end of each episode. **c)** The average amount of commission spent at the end of each episode

5.1.3 Evaluation on WithSentiments Environment

We include the sentiment scores of news headlines for each asset in the state representation and repeat the experiment with same configurations, similar to sections 5.1.1 and 5.1.2. We refer to this environment with sentiment analysis scores, technical indicators and close price in the state representation as *WithSentiments* environment.

The plot showing the results in Fig.12 demonstrates that augmenting the state with sentiment analysis along with technical indicators has improved the agent performance.

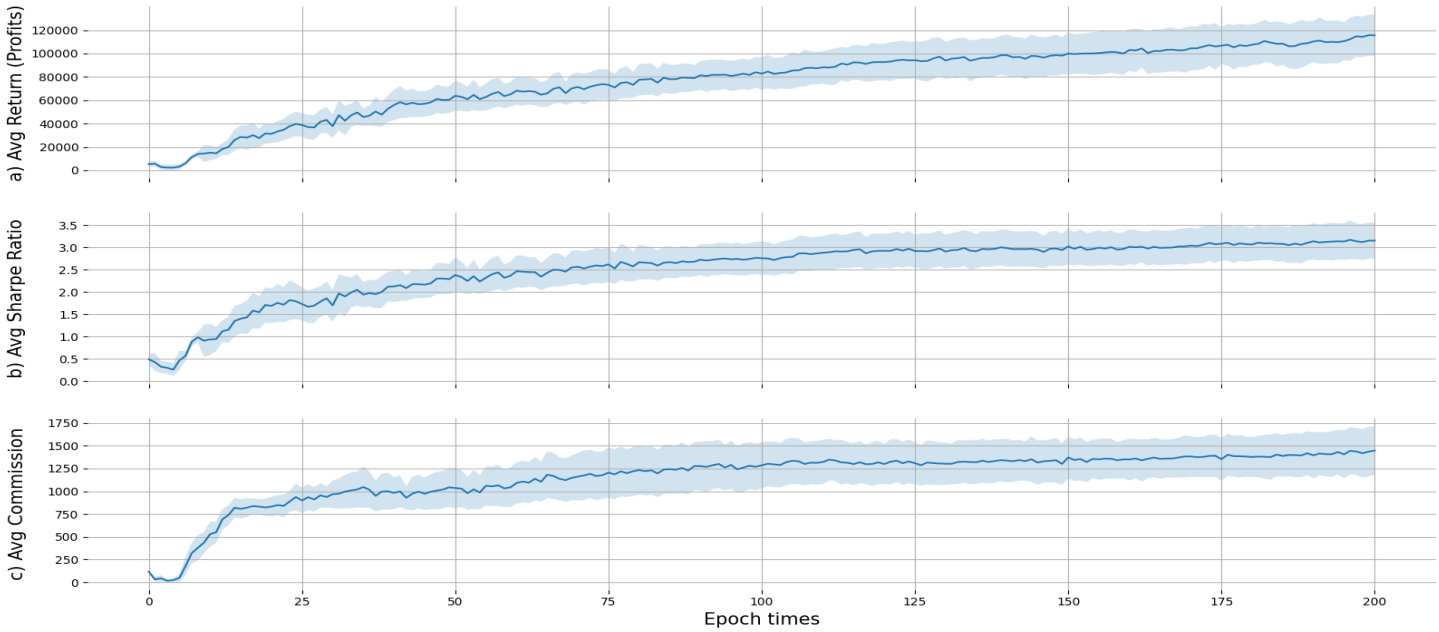


Figure 12: TD3 agent performance metrics on WithSentiments Environment using the same hyperparameter configurations averaged over 5 different random seeds. **a)** Average return (Profits in dollars) at the end of each episode. **b)** The average annual Sharpe ratio at the end of each episode. **c)** The average amount of commission spent at the end of each episode

The total average return profits increased to 115591$ with standard error equals to

± 17721\$ across the five runs. Sharpe ratio increased to 3.14 and ± 0.40 standard error. The average amount of commission approximately equals the amount spent in the environment with only technical indicators (WithTechIndicators environment), which means that the agent performed almost the same number of trades but with a better decision (policy). In the benchmark [67] study, they also reported an increase in the agent performance when adding sentiment scores to the state with a Sharpe ratio equal to 2.4.

- **Experiment's Summary**

  In all plots of the three evaluations, we notice that the policy improves over time, as the agent accumulates more reward, and thus the Sharp ratio increases. The slope is almost flat towards the end, indicating the policy has stabilized to the local optimum. As the stock trading problem has never been solved, we do not have a specified reward or Sharpe ratio threshold at which it's considered solved.

Evaluation Environment	Baseline	WithTechIndicators	WithSentiments
Accumulated Return	33960\$ $\pm$4473	89782\$ $\pm$18980	115591\$ $\pm$17721
Sharpe Ratio	1.43 $\pm$0.13	2.75 $\pm$0.43	3.14 $\pm$0.4
Commission	355\$ $\pm$83	1109\$ $\pm$248	1447\$ $\pm$268
Sharpe Ration benchmark	0.85	1.4	2.4

Table 1: THE PERFORMANCE EVALUATION COMPARISON BETWEEN THREE DIFFERENT EVALUATIONS AND BENCHMARK

5.2 Second Experiment

In the second experiment, we evaluate our approach on a wider action and state spaces by considering 10 assets to trade, AAPL, MSFT, QCOM, IBM, RTX, PG, GS, NKE, DIS, and AXP. Our back-testing consists of historical daily data from 01/01/2010 to 01/01/2018 with initial capital of 100000\$ for performance evaluation (Fig.13). We split the dataset into two periods; the first period is to train the agent, the second is used to test the agent's performance on unseen data. We notice that for our model to generalize better, we had to impose regularization by normalizing the observation space using *Batch*

Figure 13: Training and test data splits

Normalization. This technique uses mini-batches from samples to have unit mean and variance. It maintains a running moving average of the mean and variance to normalize the observation vector during testing. We further normalized the rewards received by the agent as it makes the gradient steeper for better rewards. We also set the look-back window to 20 ($\mathcal{W} = 20$). We added action noise to encourage exploration during training to force the agent to try different actions and explore its environment more effectively, leading to higher rewards and more elegant behaviors.

Our approach successfully archived a 2.68 Sharpe ratio which considered *"very good"* and 110308$ as total profits (Rewards) on the test data. We let the agent keep learning on the test set since this will help to better adapt to the market dynamics.

- **Disabling Sell Action:**

  To investigate whether the profits made on the test data (between 2016 and 2018) are a matter of the standard increase in the stocks and the market growth in general (primarily that tech stocks are known for their excellent performance in the past years and that Dow Jones represents has one of the best companies in an economically stable country) or are made due to the decision made by the agent, we disable the sell action and only let the agent buy and hold during the trading episode. As a result the agent allocated the capital as follows:

Stock	Shares
AAPL	751
AXP	398
MSFT	398
IBM	200
DIS	199

and hold on to this position until the end of the episode. The Sharpe ratio has decreased to 2.00 with 66949$ as total profits (Rewards), which indicates that the decision made by the agent had a positive effect on the return and it was not merely due to the natural growth of the market.

- **Risk-Free Rate:**

  When calculating the Sharpe ratio, subtracting the risk-free return from the mean return allows us to isolate better the profits associated with risk-taking activities. In all evaluations so far, we assumed that the risk-free return is 0 because nowadays, since the interest rates are so low, it is commonly assumed that the risk-free rate is zero. To better reflect the risk associated with profits, we evaluate again on the test data but with using the yield of *Treasury Securities* rate instead.

  As we are calculating the Sharpe ratio of a 3-year portfolio, we use a 3-year Treasury note (T-note) rate that equals 2.46% at the end of 2018 (according to the U.S. Department Of The Treasury). After introducing the risk-free return, the Sharpe ratio has decreased slightly to 2.639, given that the return percentage of the portfolio is 98.8% with standard deviation of 36.5%.

CHAPTER VI

CONCLUSION AND FUTURE WORK

This work presented a Deep Reinforcement Learning approach that combines technical indicators with sentiment analysis to find an optimal trading policy for assets in the stock market. Results show that the addition of technical indicators and sentiment scores of the news headlines to the state representation has significantly improved the agent's performance and the superiority of using a continuous action space over a discrete one to solve the trading problem. We also explored the potential of using an Actor-Critic algorithm (TD3) to solve the portfolio allocation problem. Our approach achieved an annual Sharpe ratio of 2.68 on test data, which is considered "Good" by investors. The approach can be improved in future work by having more computational power to run more experiences and better evaluate the approach. Our environment, agent, and learning process possess many hyperparameters that must be tuned. It will be interesting to see the model's performance with better-tuned parameters, which requires high computation power. In addition, we believe that training an NLP algorithm to process the financial news content instead of only the headline may positively affect the agent performance.

APPENDIX A

EXPERIMENTS HYPERPARAMETERS

A.1 First Experiment setup and configurations

- **Environment Configuration**
  - Date: '2011-01-01' - '2016-01-01'
  - Stocks: 'QCOM', 'MSFT'
  - Env type: 'baseline', 'WithTechIndicators', 'WithSentiments'
  - Capital: 10000
  - Action Bound: 100
  - Buy Commission: 0.0005
  - Sell Commission: 0.0001
- **TD3 configuration**
  - Action Noise: None
  - Batch Size: 100
  - Buffer Size: 1000000
  - Gamma: 0.99
  - Gradient Steps: -1
  - Learning Rate: 0.001
  - Learning Starts: 100
  - Policy: MlpPolicy
  - Policy Delay: 2
  - Target Noise Clip: 0.5
  - Target Policy Noise: 0.2
  - Tau: 0.005
  - Train Frequency: 1 episode
- **Learning Configuration**
  - Observation Space Normalization: Epsilon: 1.0e-09, Gamma: 0.99, Bound: 100
  - Reward Normalization: None
  - Number of episodes: 200
  - Seed Set: [1, 2, 3, 4, 5]

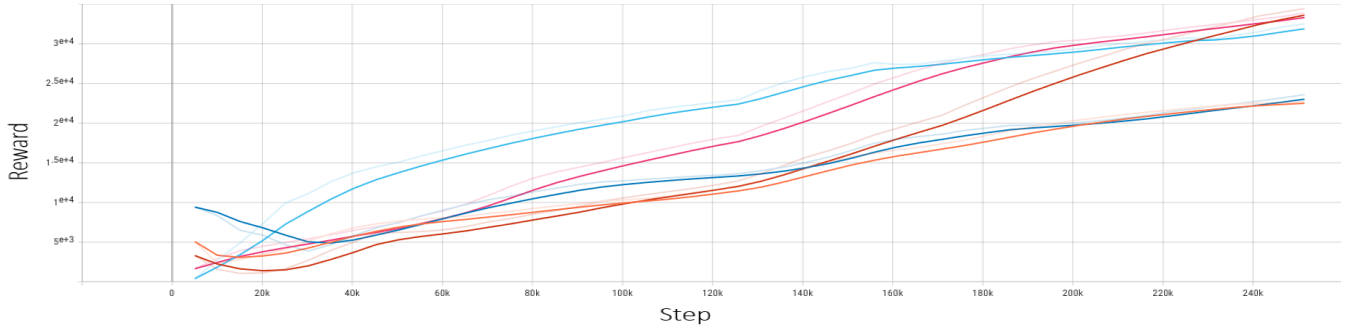
A.2 Second Experiment setup and configurations

- **Environment Configuration**
  - Date: '2010-01-01' - '2018-01-01'
  - Stocks: 'AAPL', 'MSFT', 'QCOM', 'IBM', 'RTX', 'PG', 'GS', 'NKE', 'DIS', 'AXP'
  - Env type: 'WithSentiments'
  - Capital: 1000000
  - Action Bound: 200
  - Buy Commission: 0.0005
  - Sell Commission: 0.0005
- **TD3 configuration**
  - Action Noise: NormalActionNoise
  - Batch Size: 100
  - Buffer Size: 1000000
  - Gamma: 0.85
  - Gradient Steps: -1
  - Learning Rate: 0.0001
  - Learning Starts: 100
  - Policy: MlpPolicy
  - Policy Delay: 2
  - Target Noise Clip: 0.5
  - Target Policy Noise: 0.3
  - Tau: 0.0001
  - Train Frequency: 1 episode
- **Learning Configuration**
  - Observation Space Normalization: Epsilon: 1.0e-09, Gamma: 0.9, Bound: 1000
  - Reward Normalization: Epsilon: 1.0e-09, Gamma: 0.9, Bound: 1000
  - Number of episodes: 200
  - Seed Set: 3

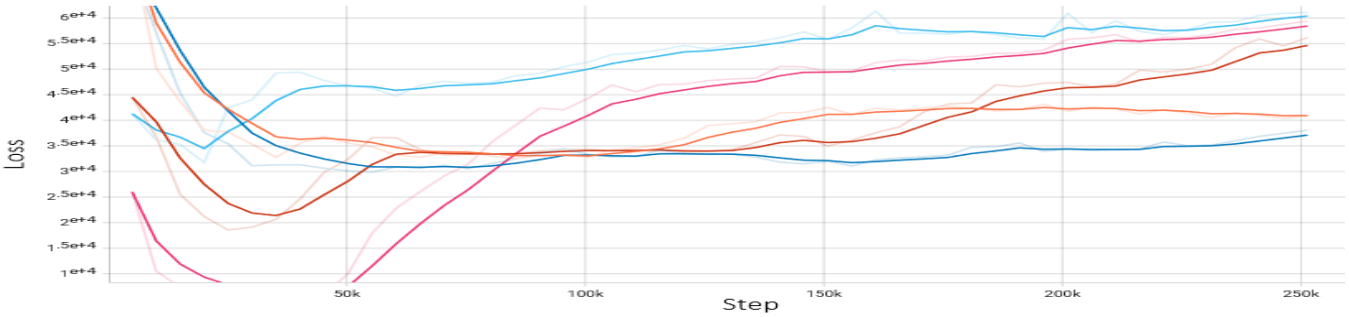
APPENDIX B

LEARNING CURVES

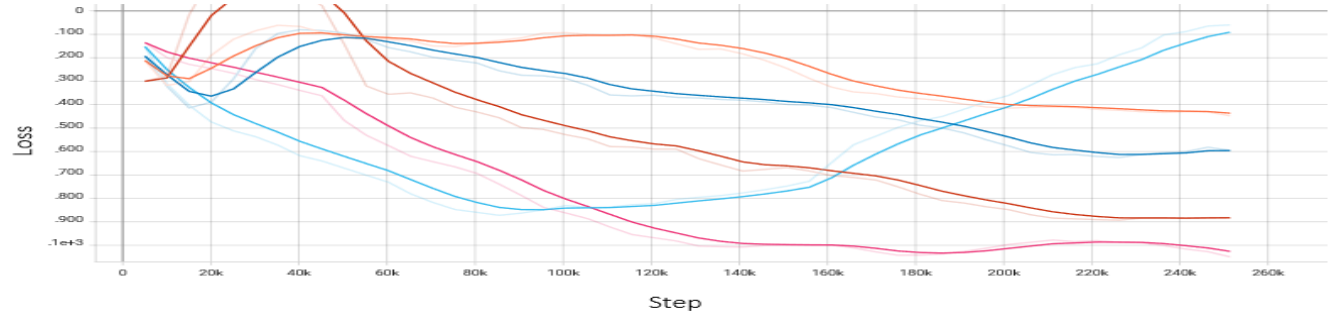
B.1 Baseline Environment



(a)



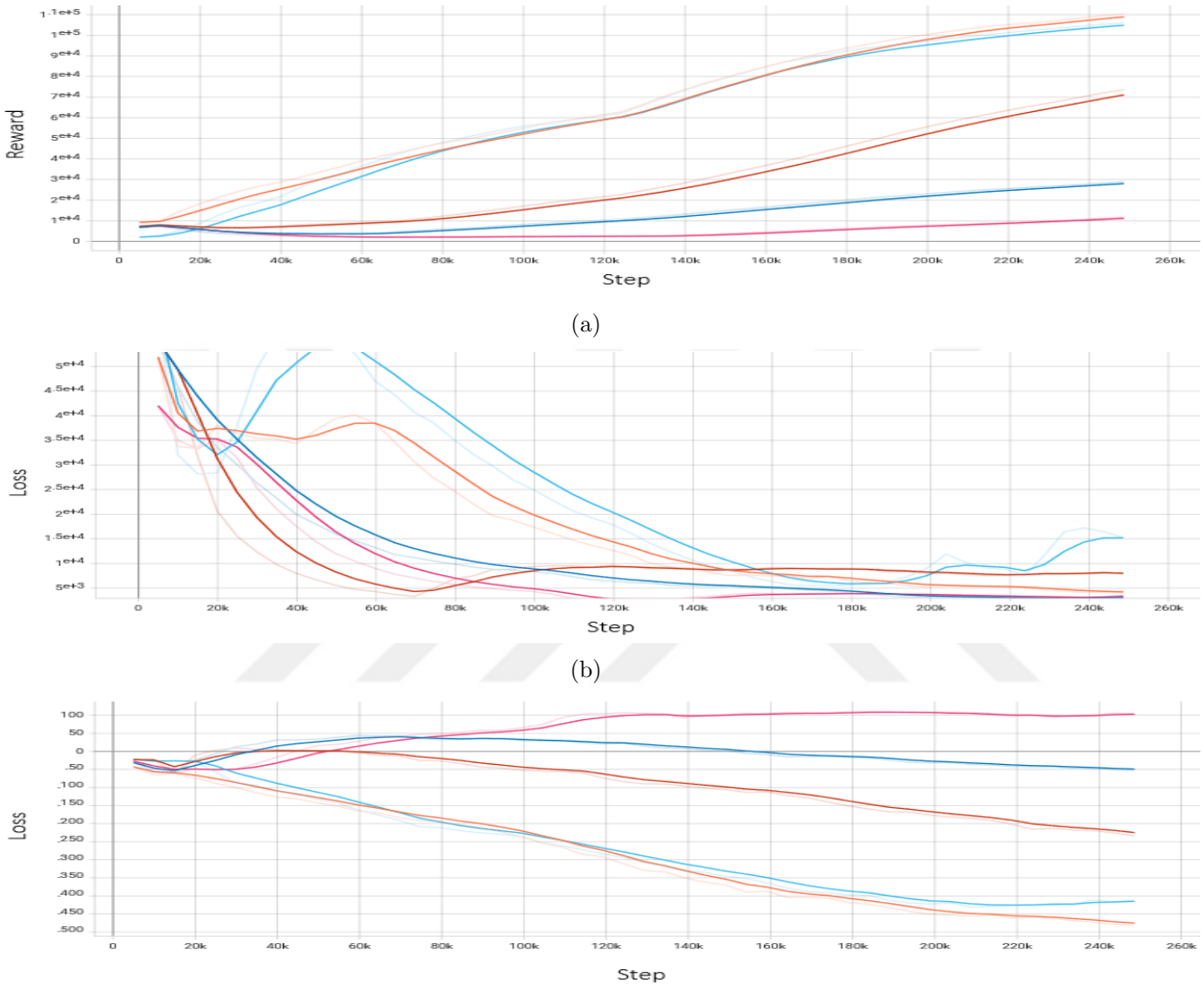
(b)



(c)

Figure 14: Learning curves of TD3 agent on Baseline Environment. Curves are smoothed uniformly for visual clarity. a) Average Reward b)Train Critic Loss, c) Train Actor Loss

B.2 WithTechIndicators Environment



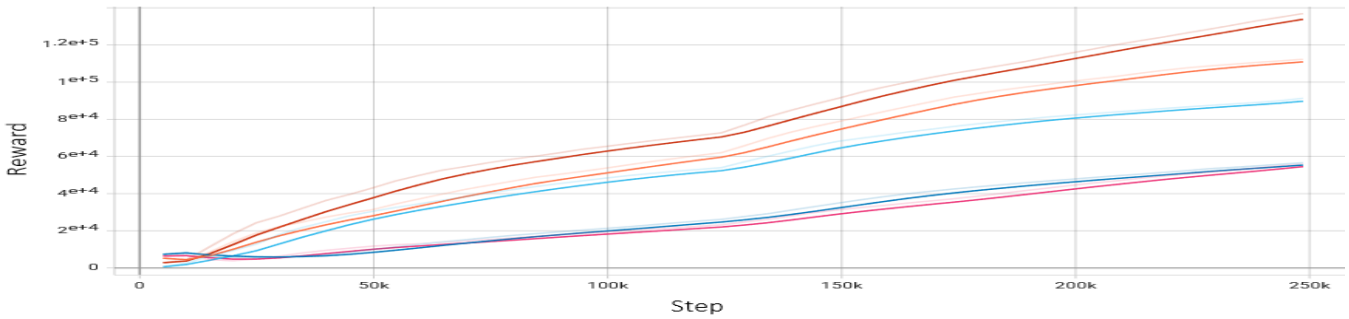
(a)

(b)

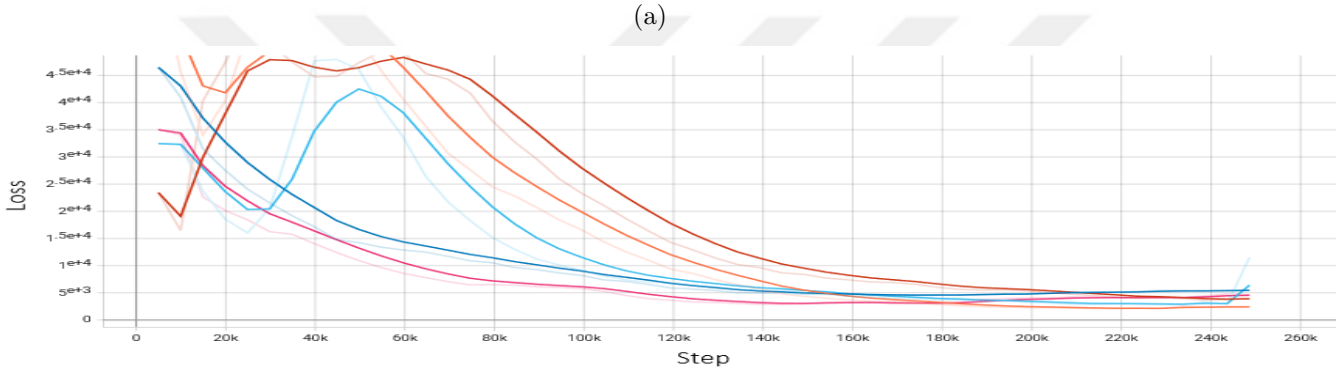
(c)

Figure 15: Learning curves of TD3 agent on WithTechIndicators Environment. Curves are smoothed uniformly for visual clarity. a) Average Reward b)Train Critic Loss, c) Train Actor Loss

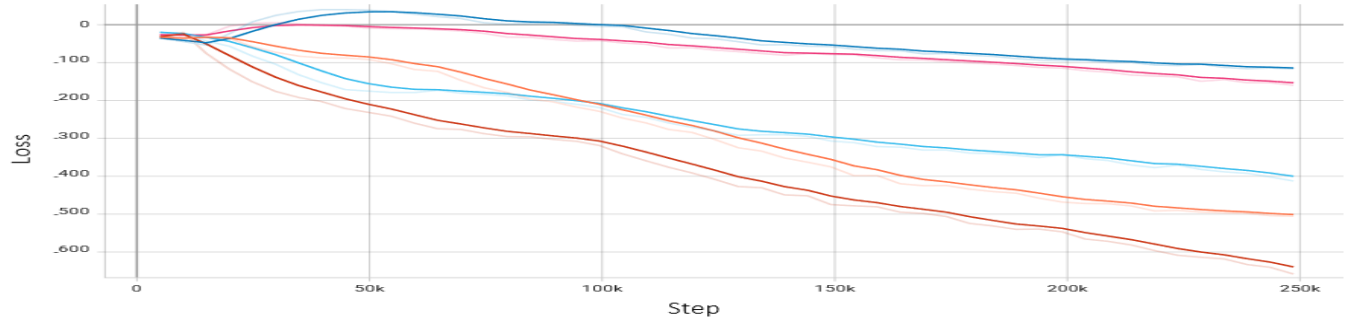
B.3 WithSentiments Environment



(a)



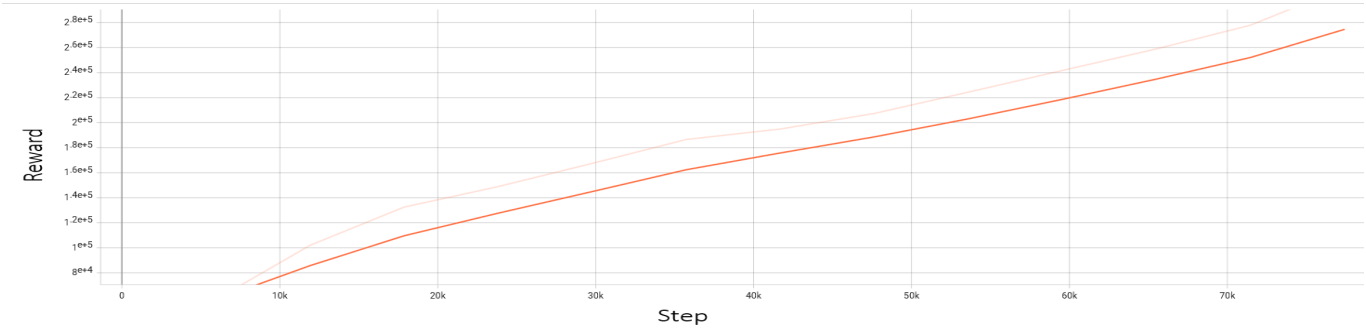
(b)



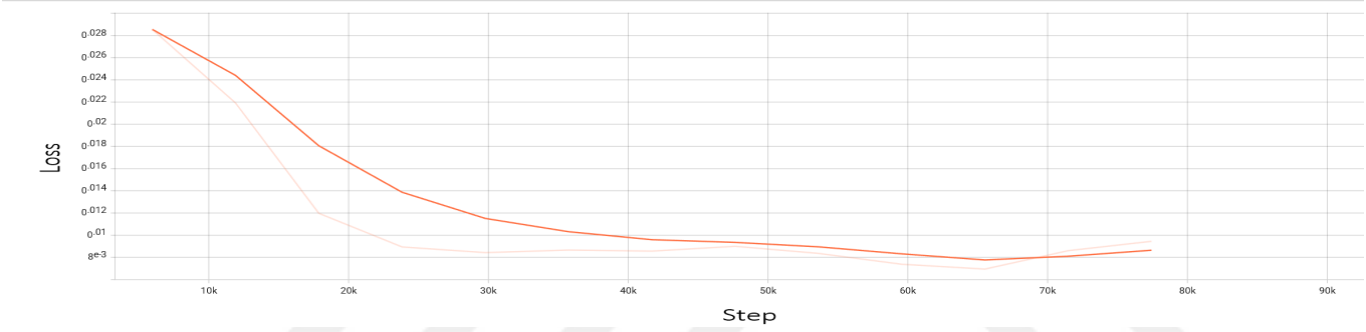
(c)

Figure 16: Learning curves of TD3 agent on WithSentiments Environment. Curves are smoothed uniformly for visual clarity. a) Average Reward b)Train Critic Loss, c) Train Actor Loss

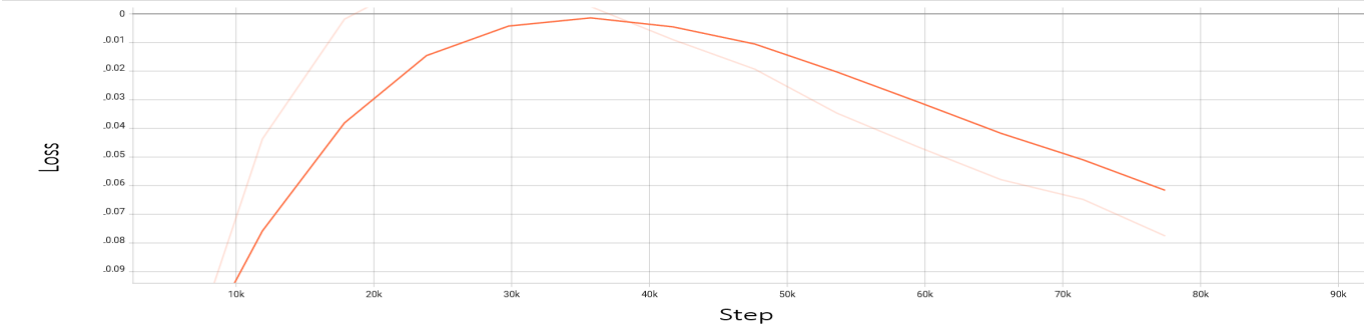
B.4 Second Experiment



(a)



(b)



(c)

Figure 17: Learning curves of TD3 agent on WithSentiments Environment during training phase. Curves are smoothed uniformly for visual clarity. a) Average Reward b)Train Critic Loss, c) Train Actor Loss

Bibliography

[1] R. S. Sutton, F. Bach, and A. G. Barto, *Reinforcement Learning: An Introduction*. MIT Press Ltd, 2018.

[2] M. M. L. de Prado, "The 10 reasons most machine learning funds fail," *WGSRN: Data Collection & Empirical Methods (Topic)*, 2018.

[3] H. Markowitz, "Portfolio selection*," *The Journal of Finance*, vol. 7, no. 1, pp. 77–91, 1952.

[4] T. L. Meng and M. Khushi, "Reinforcement learning in financial markets," *Data*, vol. 4, no. 3, 2019.

[5] K. Zarkias, N. Passalis, A. Tsantekidis, and A. Tefas, "Deep reinforcement learning for financial trading using price trailing," in *2019 IEEE International Conference on Acoustics, Speech, and Signal Processing, ICASSP 2019 - Proceedings*, pp. 3067–3071, IEEE, 2019.

[6] S. Chakraborty, "Capturing financial markets to apply deep reinforcement learning," 2019.

[7] S. Parsons, "Introduction to machine learning by, mit press," *The Knowledge Engineering Review*, vol. 20, no. 4, p. 432–433, 2005.

[8] P. A. Josh Achiam, "Part 2: Kinds of rl algorithms," 2018.

[9] O. Alagoz, H. Hsu, A. J. Schaefer, and M. S. Roberts, "Markov decision processes: A tool for sequential decision making under uncertainty," *Medical Decision Making*, vol. 30, no. 4, p. 474–483, 2009.

[10] M. L. Puterman, *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2010.

[11] M. Wiering and M. v. Otterlo, *Reinforcement learning: state-of-the-art*. Springer, 2012.

[12] R. E. Bellman, *Dynamic programming*. Princeton University Press, 2010.

[13] D. P. Bertsekas, *Dynamic programming and optimal control*. Athena Scientific, 2018.

[14] R. A. Howard, *Dynamic programming and markov processes*. M.I.T. Press, 1969.

[15] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," 2013.

[16] G. Tesauro, "Temporal difference learning and td-gammon," *Commun. ACM*, 1995.

[17] L.-J. Lin, "Scaling up reinforcement learning for robot control," *Machine Learning Proceedings 1993*, p. 182–189, 1993.

[18] R. S. Sutton, D. McAllester, S. Singh, and Y. Mansour, "Policy gradient methods for reinforcement learning with function approximation," in *Proceedings of the 12th International Conference on Neural Information Processing Systems*, NIPS'99, p. 1057–1063, MIT Press, 1999.

[19] P. A. Josh Achiam, "Part 3: Intro to policy optimization," 2018.

[20] A. Akhmetzyanov, R. Yagfarov, S. Gafurov, M. Ostanin, and A. Klimchik, "Continuous control in deep reinforcement learning with direct policy derivation from q network," in *Human Interaction, Emerging Technologies and Future Applications II*, (Cham), pp. 168–174, Springer International Publishing, 2020.

[21] V. Mnih, K. Kavukcuoglu, D. Silver, A. Rusu, J. Veness, M. Bellemare, A. Graves, M. Riedmiller, A. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, pp. 529–33, 02 2015.

[22] H. van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," *CoRR*, vol. abs/1509.06461, 2015.

[23] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," 2019.

[24] S. Fujimoto, H. van Hoof, and D. Meger, "Addressing function approximation error in actor-critic methods," *CoRR*, vol. abs/1802.09477, 2018.

[25] J. Patel, S. Shah, P. Thakkar, and K. Kotecha, "Predicting stock and stock price index movement using trend deterministic data preparation and machine learning techniques," *Expert Systems with Applications*, vol. 42, no. 1, pp. 259–268, 2015.

[26] A. Tsantekidis, N. Passalis, A. Tefas, J. Kanniainen, M. Gabbouj, and A. Iosifidis, "Forecasting stock prices from the limit order book using convolutional neural networks," in *2017 IEEE 19th Conference on Business Informatics (CBI)*, vol. 01, pp. 7–12, 2017.

[27] A. Ntakaris, J. Kanniainen, M. Gabbouj, and A. Iosifidis, "Mid-price prediction based on machine learning methods with technical and quantitative indicators," *PLOS ONE*, vol. 15, pp. 1–39, 06 2020.

[28] Y. Hao and Q. Gao, "Predicting the trend of stock market index using the hybrid neural network based on multiple time scale feature learning," *Applied Sciences*, vol. 10, no. 11, 2020.

[29] M. R. Vargas, C. E. M. dos Anjos, G. L. G. Bichara, and A. G. Evsukoff, "Deep leaming for stock market prediction using technical indicators and financial news articles," in *2018 International Joint Conference on Neural Networks (IJCNN)*, pp. 1–8, 2018.

[30] T.-J. Chang, N. Meade, J. Beasley, and Y. Sharaiha, "Heuristics for cardinality constrained portfolio optimisation," *Computers & Operations Research*, vol. 27, no. 13, pp. 1271–1302, 2000.

[31] A. Schaerf, "Local search techniques for constrained portfolio selection problems," *Computational Economics*, vol. 20, pp. 177–90, 12 2002.

[32] F. Bertoluzzo and M. Corazza, "Testing different reinforcement learning configurations for financial trading: Introduction and applications," *Procedia Economics and Finance*, vol. 3, pp. 68–77, 2012. International Conference Emerging Markets Queries in Finance and Business, Petru Maior University of Tîrgu-Mures, ROMANIA, October 24th - 27th, 2012.

[33] M. Corazza and F. Bertoluzzo, "Q-learning-based financial trading systems with applications," Working Papers 2014:15, Department of Economics, University of Venice "Ca' Foscari", 2014.

[34] Z. Tan, C. Quek, and P. Y. Cheng, "Stock trading with cycles: A financial application of anfis and reinforcement learning," *Expert Systems with Applications*, vol. 38, no. 5, pp. 4741–4755, 2011.

[35] J. Kitchin, "Cycles and trends in economic factors," *The Review of Economics and Statistics*, vol. 5, no. 1, pp. 10–16, 1923.

[36] H. Sarlan, "Cyclical aspects of business cycle turning points," *International Journal of Forecasting*, vol. 17, no. 3, pp. 369–382, 2001.

[37] Y. Deng, F. Bao, Y. Kong, Z. Ren, and Q. Dai, "Deep direct reinforcement learning for financial signal representation and trading," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 28, no. 3, pp. 653–664, 2017.

[38] L. Conegundes and A. C. M. Pereira, "Beating the stock market with a deep reinforcement learning day trading system," in *2020 International Joint Conference on Neural Networks (IJCNN)*, pp. 1–8, 2020.

[39] H. Yang, X.-Y. Liu, S. Zhong, and A. Walid, "Deep reinforcement learning for automated stock trading: An ensemble strategy," *SSRN Electronic Journal*, 01 2020.

[40] Wikipedia, "Partially observable Markov decision process — Wikipedia, the free encyclopedia," 2021. [Online; accessed 22-September-2021].

[41] C. Kirkpatrick and J. R. Dahlquist, "Technical analysis: The complete resource for financial market technicians," 2006.

[42] J. R. Nofsinger, "The impact of public information on investors," *Journal of Banking & Finance*, vol. 25, no. 7, pp. 1339–1366, 2001.

[43] Z. Xiong, X.-Y. Liu, S. Zhong, H. Yang, and A. Walid, "Practical deep reinforcement learning approach for stock trading," 2018.

[44] A. Y. Ng, D. Harada, and S. Russell, "Policy invariance under reward transformations: Theory and application to reward shaping," in *In Proceedings of the Sixteenth International Conference on Machine Learning*, pp. 278–287, Morgan Kaufmann, 1999.

[45] "Investopedia – slippage definition." `https://www.investopedia.com/terms/s/slippage.asp`. [Online; accessed 02-October-2021].

[46] Z. Jiang, D. Xu, and J. Liang, "A deep reinforcement learning framework for the financial portfolio management problem," 2017.

[47] T. T.-L. Chong, W.-K. Ng, and V. K.-S. Liew, "Revisiting the performance of macd and rsi oscillators," *Journal of Risk and Financial Management*, vol. 7, no. 1, pp. 1–12, 2014.

[48] J. Granville, *Granville's New Key to Stock Market Profits*. Papamoa Press, 2018.

[49] X. Ding, Y. Zhang, T. Liu, and J. Duan, "Using structured events to predict stock price movement: An empirical investigation," in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, (Doha, Qatar), pp. 1415–1425, Association for Computational Linguistics, Oct. 2014.

[50] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," 2019.

[51] D. Araci, "Finbert: Financial sentiment analysis with pre-trained language models," 2019.

[52] P. Malo, A. Sinha, P. J. Korhonen, J. Wallenius, and P. Takala, "Good debt or bad debt: Detecting semantic orientations in economic texts," *Journal of the Association for Information Science and Technology*, vol. 65, 2014.

[53] Python Core Team, *Python: A dynamic, open source programming language*. Python Software Foundation, 2019. Python version 3.7.

[54] W. McKinney, "Data structures for statistical computing in python," in *Proceedings of the 9th Python in Science Conference* (S. van der Walt and J. Millman, eds.), pp. 51 – 56, 2010.

[55] C. R. Harris and M. *et al.*, "Array programming with NumPy," *Nature*, vol. 585, no. 7825, pp. 357–362, 2020.

[56] M. Abadi and A. A. *et al.*, "TensorFlow: Large-scale machine learning on heterogeneous systems." https://www.tensorflow.org/, 2015. Software available from tensorflow.org.

[57] G. Brockman and V. C. *et al.*, "Openai gym," 2016.

[58] A. Raffin, A. Hill, and E. *et al.*, "Stable baselines3." `https://github.com/DLR-RM/stable-baselines3`, 2019.

[59] A. Paszke and G. *et al.*, "Automatic differentiation in PyTorch," in *NIPS Autodiff Workshop*, 2017.

[60] "Yahoo finance." https://finance.yahoo.com/.

[61] G. Manoim, "exchange-calendars." `https://pypi.org/project/exchange-calendars/`.

[62] "Kaggle – daily financial news for 6000+ stocks." `https://www.kaggle.com/miguelaenlle/massive-stock-news-analysis-db-for-nlpbacktests`. [Online; accessed 15-November-2021].

[63] "Kaggle – sun, j. (2016, august). daily news for stock market prediction." `https://www.kaggle.com/aaron7sun/stocknews`. [Online; accessed 15-November-2021].

[64] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," 2015.

[65] P. Henderson, R. Islam, P. Bachman, J. Pineau, D. Precup, and D. Meger, "Deep reinforcement learning that matters," 2019.

[66] W. F. Sharpe, "The sharpe ratio," *The Journal of Portfolio Management*, vol. 21, no. 1, pp. 49–58, 1994.

[67] S. Kau, "Algorithmic trading using reinforcement learning augmented with hidden markov model. working paper, stanford university.," 2017.

VITA

Taylan Kabbani graduated from Yildiz Technical University in 2018 with a B.Sc. degree in Bio and Biomedical Engineering. During his undergraduate studies, he joined the 1. Temel Onkoloji Sempozyumu conference in Dokuz Eylül University and has been awarded the third prize for his research paper under the name of CLASSIFICATION OF TUMOR GENE EXPRESSION USING ARTIFICIAL NEURAL NETWORKS. in 2019, he joined the Master program in Özyeğin University with a full scholarship under the supervision of Prof. Ekrem Duman. Since graduation, he has worked in multiple roles as a Data Scientist and AI Engineer, and he is currently working as a Software Engineer in Huawei.