Dissertation Type: enterprise

DEPARTMENT OF COMPUTER SCIENCE

The Deeply Reinforced Trader

Ashwinder Khurana

A dissertation submitted to the University of Bristol in accordance with the requirements of the degree of Master of Engineering in the Faculty of Engineering.

Tuesday 1st September, 2020

Declaration

This dissertation is submitted to the University of Bristol in accordance with the requirements of the degree of MEng in the Faculty of Engineering. It has not been submitted for any other degree or diploma of any examining body. Except where specifically acknowledged, it is all the work of the Author.

Ashwinder Khurana, Tuesday 1st September, 2020

Contents

List of Figures

Executive Summary

One of the primary goals of artificial intelligence is to solve complex tasks that usually consist of highly dimensional, unprocessed and noisy input in practical environments. Namely, recent advances in Deep Reinforcement Learning (DRL) has resulted in the "Deep Q Network" [25], which is capable of human level performance on several Atari video games - using unprocessed pixels for input. A similar "gamification" of the process of proprietary trading in a continuous double auction with a LOB, can be an application of DRL.

DRL requires an agent to interact with an environment and to derive actions from the state of the environment at each timestep. The environment in this case can be a market session, in which several traders interact and utilise the LOB to communicate orders to buy or sell an asset, and the agent that interacts with the environment is the proprietary trader, which aims to seek profit for itself rather than on the behalf of a client. Information available from the LOB can be used as a combination of inputs to represent the current state of the 'game', from which the trader can learn which actions to take via a vast number of interactions with the market during the learning process. This is done by using a policy which determines what action to take depending on the current state of the LOB and a reward system which provides rewards depending on the action.
Furlan-Falcao (2019) had used a Policy Gradient agent (PG Agent) in conjunction with Cliff's (1997) ZIP to solve the sales trader problem. In comparison, this thesis attempts to solely use state-of-the-art DRL techniques to solve the proprietary trader problem, namely via models such as: Policy Gradient and Actor-Critic models like DDPG.

Whilst most trading algorithms are trained and analysed with data from public exchanges, collected from sites such as finance.yahoo.com, this thesis focuses on using Cliff's Bristol Stock Exchange [5] and Furlan-Falcao's Bristol Stock Gym [12] to simulate market sessions of robot traders to train and compare the effectiveness of the agent against other traditional machine learning based algorithms.

This is a hugely interesting task as the mapping between the current state of the system and the action the agent takes in such an environment is an abstraction of the overall problem all financial institutions are aiming to tackle in the most sophisticated ways to gain an overall competitive advantage. This makes the motivation and application of this theory entirely relevant to large scale, real-world industry problems.
The results of this thesis are not a success story, but it details the journey of iterating and analysing results in an attempt to create a profitable trader using the techniques outlined above and the vast challenges that were presented.

> The research hypothesis is that the latest developments in machine learning, more specifically, a trader that uses Deep Reinforcement Learning techniques alone provide a tool set to create a proprietary trader that outperforms humans.

This project explores and aims to answer the hypothesis under the limitations of time, manpower and resources of a Masters project. Below is a summary of the work carried out over the span of the project.

- I spent 100+ hours researching prior work on the field and learning about Deep Reinforcement Learning techniques.
- I wrote a total of 2000+ lines of source code, comprising of the merger of capabilities of BSE into Furlan-Falcao's BristolStockGym, exploration of several DRL models, and a Variational Auto-Encoder.

- Spent 100+ hours training the various models and running experiments on Blue Crystal 4 [14], with extensive analysis on all the results.

Supporting Technologies

- All the codebase is written using Python version 3.6.7 and the Python Package Installer PIP tool in its 19.0.3 version.
- Several Python libraries were used extensively throughout the project, namely: PyTorch(1.3.0), NumPy(1.18.1), Matplotlib(3.1.3), Sklearn(0.22)
- I used Furlan-Falcao's version of Cliff's Bristol Stock Exchange that is refactored to be compatible for Deep Reinforcement Learning analogous to OpenAI's Gym

Notation and Acronyms

All of the acronyms used in this document are explained on their first usage and no more. However, several acronyms are used across multiple chapters throughout the thesis, they are as follows:

Acronyms

AI	:	Artificial Intelligence
BSE	:	Bristol Stock Exchange
BSG	:	Bristol Stock Gym
LOB	:	Limit Order Book
CDA	:	Continuous Double Auction
MDP	:	Markov Decision Process
DRL	:	Deep Reinforcement Learning
PG	:	Policy Gradient
DDPG	:	Deep Deterministic Policy Gradient
ZI-C	:	Zero Intelligence - Constrained
ZIP	:	Zero Intelligence Plus
DRT	:	Deeply Reinforced Trader
AA	:	Adaptive Aggressiveness
AE	:	Auto Encoder
TD	:	Temporal Difference
TDL	:	Temporal Difference Learning
SGD	:	Stochastic Gradient Descent
SGA	:	Stochastic Gradient Ascent
MC	:	Monte-Carlo
MSE	:	Mean Squared Error
DQN	:	Deep Q-Network
NN	:	Neural Network
PG	:	Policy Gradient

Acknowledgements

An optional section, of at most 1 **page**

It is common practice (although totally optional) to acknowledge any third-party advice, contribution or influence you have found useful during your work. Examples include support from friends or family, the input of your Supervisor and/or Advisor, external organisations or persons who have supplied resources of some kind (e.g., funding, advice or time), and so on.

Chapter 1

Contextual Background

1.1 Introduction

The underlying context behind this thesis is the comprehension, analysis and exploitation of the mechanisms that underpin the real-world financial markets - from a Computer Scientist's perspective. The infrastructure and mechanism of interest are major stock exchanges. Stock exchanges are where individuals or institutions come to a centralised market to buy and sell stocks, where the price of these stocks are derived from the supply and demand in the market as the buyers and sellers place their orders on the exchange. There are various types of orders that can be used to trade a stock, however, this thesis focuses on a specific type of order, the 'Limit Order'. A limit order is a type of order to purchase or sell a security at a specified price or better. For buy limit orders, the order will be executed only at the limit price or lower, while for sell limit orders, the order will be executed only at the limit price or higher[19]. A collection of orders of this kind formulates the LOB, which is a book, maintained by the exchange, that records the outstanding limit orders made by traders.

The role of a sales trader is to trade on behalf of a client, which means that they do not trade a commodity of their own, but instead buy and sell the commodity to secure the best price for their client. A proprietary trader (a "prop" trader) on the other hand buys and sells commodities on their own behalf with their own capital with the aim of seeking profitable returns. Both types of traders have to constantly adapt to the market conditions and react through the prices they quote for their orders.

Originally, the job of trading was fulfilled by humans who reacted to the market conditions using their intuition and their expert knowledge of the financial markets, however the job was hit with a wave of automation after 2001, when IBM researchers published results that indicated that two algorithms (ZIP by Dave Cliff[8] and MGD, a modification of GD by Steven Gjerstad & John Dickhaut [15]) consistently outperformed humans in a experimental laboratory version of financial markets.

Since then, there has been relatively little research done in this field within *academia*, and subsequent solutions have built upon the traditional heuristic alongside 'old school' machine learning approaches. Considering the explosion of Deep Learning as a field in this last decade, this thesis attempts to explore the application of these state-of-the-art techniques, in particular Deep Reinforcement Learning, to the problem of the adaptive prop trader.

1.2 History of Research

1.2.1 Experimental Economics to Algorithmic Trading

Analysis of the sales trader problem can be seen as early as the 1960s in Vernon Smith's experiments at Purdue University, which led to the birth of the field of Experimental Economics and a Nobel Prize for Smith [35].

Smith's motivation behind these experiments was to find empirical evidence for the credibility of basic microeconomic theory of Continuous Double Auctions (CDAs) to generate competitive equilibria - which is the condition in which profit-maximising producers and utility-maximising consumers in com-

petitive markets arrive at an equilibrium price[21]. He formalised this experiment structure as an oral double auction in which he would a split a group of humans up into two subgroups: Buyers and Sellers. He gave each human a private limit price, where buyers would not buy above that price and sellers would not sell below that price. Prices quoted by the buyers are known as the bid prices and the prices quoted by the sellers are known as the ask prices. The buyers and sellers would interact within small time based sessions, otherwise known as trading sessions.

After running repeated experiments of the trading sessions, Smith empirically demonstrated that the prices quoted by the traders demonstrated rapid equilibration i.e the prices would converge to the theoretical 'best' price of an asset, despite the sessions consisting of a small number of traders.

These sets of experiments were extended by Gode and Sunder [38] in the 1990's. Crucially, however, the human traders that were the core of Smith's experiments were replaced by Gode & Sunder with computer based algorithmic traders. Gode and Sunder introduced two types of *zero-intelligence* algorithmic traders for CDA markets: Zero Intelligence - Unconstrained (ZI-U) and Zero Intelligence - Constrained (ZI-C). ZI-U traders are traders that generated a uniformly random bid/ask price, whilst ZI-C traders were traders that also generated a random price, however it was constrained between the private limit price of the trader and the lowest/highest possible price w.r.t to whether it was a buyer or seller - therefore it could never sell at a loss.

The purpose of Gode and Sunder replicating the experiments by Smith with their adaptation from humans to computers was to figure out if the allocative efficiency (the optimal distribution of goods and services, taking into account consumer's preferences) of a market was down to the intelligence of the traders or the organisation of the market. They ran several experiments with five different types of market structures and monitored their allocative efficiency. They concluded that the ZI-U traders were essentially useless because they failed to gravitate to an equilibrium price, however, the surprising result was that the ZI-C traders were surprisingly human-like as the prices quoted by these traders eventually did gravitate to the equilibrium price over the course of each trading session. This allowed them to conclude that most of the 'intelligence' was in the market structure rather than the traders.

This conclusion was critiqued by Cliff in 1997 [6] who hypothesised that the ZI-C traders would fail to equilibrate under a set of market conditions and then analytically and empirically demonstrated that this was the case. He also developed a new algorithmic trader named zero-intelligence plus (ZIP), which quoted prices based on a profit margin the algorithm adjusts using a learning rule (Widrow-Hoff with momentum). This algorithm succeeded in the market conditions in which ZI-C failed in.

Around the same time, in 1998 Gjerstad and Dickhaut developed the GD algorithm [15]. The core of this algorithm is that the traders compute a belief function that an order will be accepted by the opposing trader types (buyer vs seller) based on the information extracted from the market. This belief function seeks to maximise the trader's expected gain over the course of the session.

Prior to 2001, these algorithms had solely been pitted against other algorithms, however, in 2001, Tesauro and Das alongside colleagues Bredin and Kephant from IBM [10] ran numerous experiments where they pit algorithms like ZIP and MGD (a modified GD) against humans, as well as conducting the traditional agent vs agent trading contests. The results of these experiments were groundbreaking as they demonstrated that ZIP and MGD were dominant in the agent vs agent contents, but more crucially their performance surpassed that of human traders in the agent vs human contests.

Despite the promising results and the wide press received from the IBM publications, relatively little experimentation has since been conducted to compare performances of humans vs agents. Meanwhile, Tesauro and Bredin published an improved version GD called GD-Extended (GDX) in 2002, and Vytelingum published the Adaptive-Aggressiveness Algorithm (AA) in 2006 [39]. AA uses past history of previous trades to estimate the market equilibrium, and uses this to determine whether the current order is intramarginal/extramarginal. Based on these metrics in conjunction with its current level of 'aggressiveness', an AA agent places an order, where a more 'aggressive' order is a bid/ask that is more likely to be accepted [7]. GDX uses real-time dynamic programming to formulate agent bidding strategies in a broad class of auctions[7].

1.2.2 Developments in Artificial Intelligence

Despite the longstanding theory behind Deep Learning, it has been an explosive field in the last decade, with notable achievements including: enabling Google translate to become significantly more accurate, furthering NLP research massively, and a key factor behind the driverless car. This explosivity has been labelled the 'Deep Learning Revolution' by Terry Sejnowski [28].

Similarly, Reinforcement Learning has been an idea coined for a long time, with a strong presence in the learning via trial and error in animal learning psychology. Richard S. Sutton, considered one of the founding fathers of modern computational RL, made several contributions to the field by introducing computational learning paradigms [27].

The two ideas in isolation have had their respective achievements, however, recently there have been unprecedented achievements in the field of artificial intelligence with the combination of the two fields predictably called Deep Reinforcement Learning. Some notable achievements including Robotics (end-to-end systems) [20] that maps raw input pixels to actions from a robot and AlphaZero from Google-owned startup DeepMind which is a Deep Reinforcement Learning successor to their own AlphaGo, which beat the world champion at the board game 'Go'.

1.3 Motivations

The motivations behind this project are three fold: contributing to computational finance as described in Section 1.3.1; exploring the issues from a computer science perspective as detailed in Section 1.3.2; and extending recently published previous work, as discussed in Section 1.3.3.

1.3.1 Finance perspective

The first motivation is in the fact that any project that invents or improves upon current solutions within this domain is solving an extremely complex problem in the world of Finance.

Huge institutions such as banks and hedge funds are engaged in a constant arms race with one another to see who can create the most robust, sophisticated and profitable algorithms to enact their trades, thus an arms race for keeping their clients and staying in business. In doing so, these institutions are hiring the most able computer scientists, and quantitative developers to create optimal solutions in house to deliver this competitive edge against the rest of the market.

The heavy investment in these jobs and the development of trading algorithms comes from the relatively late adoption of surging technologies from big financial institutions, arguably driven by small FinTech firms eating from the established companies' profits with their innovative tech solutions to an otherwise very traditional industry. MX reports [37] present that as an example, a breakdown of JP Morgan's revenue indicates that roughly 15% of JP Morgan's net income ($9b) stems from their trading of assets and investing securities, which justifies their heavy interest and investment to optimise the entire process of trading.

The IBM experiment described in Section 1.2.1 also indicated that the ZIP and MGD algorithms outperformed their human counterparts in the sales trader job, albeit in a much more simplified environment, the claim still holds true for real world applications where algorithmic approaches outperform the previously hired traditional human traders.

There are several reasons for this, as computers are able to outperform a trader's intuition and can analyse large sets of data very quickly to produce fast and appropriate action. Another aspect is more so to do with human psychology. Research indicates [18] that trading and money management is akin to human survival, which directly affects the part of the brain called the amygdala. This is an evolutionary knee-jerk reaction that triggers pain or pleasure. A successful trade creates a huge sense of euphoria, but a bad trade equally creates a sense of pain, regret and frustration. This can lead to inappropriate action taken from the human trader, for example revenge trading. The objective perspective from an

individual that is uninvolved in trading suddenly becomes hard to maintain when there is a stake involved whilst trading, because of an animalistic sense of survival that kicks in which leaves very little room for subjectivity. This aspect of human psychology makes it more desirable for firms to trust algorithms that are not prone to this downfall and because they can work around the clock without a wage. This has resulted in 35% of the U.S's equities being managed by automatised processes [4].

On the flip side, despite the removal of human error, Torsten Slok, chief international economist at Deutsche Bank, has named it the number one risk to markets in 2019[4]. This is down to the potential of concentrated sell offs causing a downward spiral of markets. This is a credible argument made and raises questions for the future of AI within finance and its integration within society as whole.

Despite this, the reliance on state of the art technology and research to propel financial services into the fifth industrial revolution is undeniable which presents an opportunity for curious computer scientists and academics to tackle these complex problems.

1.3.2 Computer science perspective

As mentioned in Section 1.2.2, DRL has recently scaled to previously unsolvable problems. This is down to the ability of DRL to learn from raw sensors or input images as input, without the intervention of human, in an unsupervised machine learning manner.

Perhaps one of the most prominent examples of the impressive nature of DRL is demonstrated by AlphaGo Zero - an AI that can play the board game 'Go'. AlphaGo Zero surpassed its predecessor AlphaGo (an AI that beat that beat the world champion in the game) in an unprecedented manner because the knowledge of the AI is not bounded by human knowledge [32]. This is achieved by an initialised neural network that has no knowledge about the game of Go. The AI then plays games against itself via the combination of its network plus a powerful search algorithm. As more iterations of the games are completed, the network is tuned and better predictions are made. This updates the neural network and combined with the search algorithm to create a more 'powerful' version of itself, a new AlphaGo Zero. This process repeats to continuously improve its performance. AlphaGo on the other hand, used a more traditional approach to learn how to play the game, because it "watched" many different exemplar games and learnt based in a supervised machine learning manner.

AlphaGo Zero differs from its predecessor in many ways. The most apparent difference is that rather than the hand-engineered feature extraction process that AlphaGo used, AlphaGo Zero used the raw images of the black and white stones from the Go board as its input. Also, AlphaGo used a more traditional approach to learn how to play the game, because it "watched" many different exemplar games and learnt based in a supervised machine learning manner, whilst AlphaGo Zero learnt by playing itself. As mentioned earlier, AlphaGo Zero beat the world champion at Go. This was an unprecedented achievement for DRL as Go is classified as an EXPTIME complete problem, which DRL was able to solve to an unseen standard.

AlphaGo Zero is the holy grail within the DRL field, the potential for this process to applied to varying problem is one of the key motivations behind trying to apply the technique to this problem domain.

1.3.3 Furthering previous work

Furlan-Falcao's thesis [13] also alluded to the excitement of applying DRL techniques to this domain, but after proving unsuccessful in applying DRL in isolation to the problem domain, opted to direct his attention to combining DRL techniques with existing more traditional techniques.

Furthermore, Furlan-Falcao built an excellent framework BSG (Bristol Stock Gym) which acts as an interface to train DRL models within the problem definition, as it is a framework that is refactored from Cliff's Bristol Stock Exchange to simulate trade experiments as well training the network, whilst iterating

through episodes of the 'game' which are analogous to market sessions in BSE.

Whilst Furlan-Falcao's thesis provides a very good overview of his journey exploring DRL, this project focuses on applying several more techniques within the field to see if they alone can solve the problem of a prop trader

1.4 Central Challenges

The challenges in this project are vast which build on promising results achieved by Furlan-Falcao that serve as preliminary results for this project.

One of the main challenges is to extract the key information available from the LOB at every time step. The word 'key' is ambiguous here and must be defined which presents a huge challenge within itself. The complexity is in deciding what is useful information for the networks to use as input.

Another challenge is to design a network architecture that is fit to tackle the prop trader problem, including all its hyper-parameters and model layer choices. This is a common problem within machine learning, and despite vast resources available to implement established solutions tackle this problem, the choices made w.r.t to these factors have to be bespoke and tailored to the data available from BSG, which would take a considerable amount of time experimenting and training.

Research indicates that one of the most complex aspects of DRL is to create the optimal reward function so that the DRL agent learns the desired behaviour. The difficulty of this challenge is highlighted in the *OpenAI* gym environment 'Lunar Landing' game, in which the agent has to learn to land a spacecraft safely. A well intentioned reward system may include punishing the agent with negative rewards for landing the spacecraft incorrectly, however, this can result in the agent not wanting to land the spacecraft at all, which is not desirable behaviour. This raises in question the reward function itself as well as the choices behind the magnitude of reward for each action.

1.5 Conclusion

The high-level objective of this project is to use state-of-art Deep Reinforcement Learning techniques to develop an adaptive proprietary trader. More specifically, the concrete aims are:

1. To develop a concrete understanding of Deep Reinforcement Learning techniques from scratch as they are not a core part of the university's machine learning curriculum.
2. To develop a feature extraction mechanism to extract the most crucial information available from the public market session at every time step.
3. Experiment and explore different DRL models with a varying set of hyper-parameters and layer choices
4. Most importantly, develop the optimal reward function for the agent to exhibit the most desirable behaviour

Chapter 2

Technical Background

2.1 The Core Problem

2.1.1 The Limit Order Book

The LOB is a standardised view of all relevant/key market data within a Continuous Double Auction (CDA), and LOBs serve as the backbone for most modern electronic exchanges within financial markets. It summarises all the live limit orders that are currently outstanding on the exchange with bid orders – offers to buy an asset, and ask orders – offers to sell an asset, both displayed on either side of the LOB. Each side is called a book, resulting in the LOB displaying the bid book on one side, and the ask book on the other.

Figure 2.1: A graphical representation of a Limit Order Book, reproduced from Cliff (2018)

The left hand side of Figure 2.1 is a mockup of a graphical representation of the LOB in BSE, whilst in its original form, is encapsulated in a python dictionary data structure. As seen in the figure, the bid book lists all the bid orders with their prices and the respective quantities available at that price as a key/value pair in descending order to represent all the outstanding orders from the buyers. Conversely, the ask book lists the price-quantity pair for all the ask orders in ascending order. In this order, the highest bid price is at the top of the bid book alongside the lowest ask price: the two prices are called the *best bid* and the *best ask* respectively.

This graphical representation visually allows for easy feature extraction for some key elements that may not explicitly stated in the figure.

- The *Time Stamp* for the current time in the trading session, shown post-facing the 'T' in the top left corner
- The *Bid-Ask Spread* which is the difference between the *best ask* and the *best bid*
- The *Midprice* is the arithmetic mean of the *best ask* and the *best bid*

- The *Tape* shows a record of all the executed trades and cancellations of orders
- The *Microprice* is a cross volume weighted average of the *best ask* and *best bid*, prefaced by the green 'M' below the time stamp
- The latest limit order sent to the exchange by a trader

The *midprice* and *microprice* are values that are used to approximate the value of the asset and attempt to summarise the market. In this example snapshot Figure 2.1 the *best ask* is \$1.77 and the *best bid* is \$1.50 so the midprice is (\$1.77 + \$1.50)/2 = \$1.66. The *microprice* is a more intricate calculation because it is a cross volume weighted average:

$$\frac{BestBidQty * BestAskPrice + BestAskQty * BestBidPrice}{BestBidQty + BestAskQty} = Microprice \tag{2.1}$$

$$\frac{5 * \$1.77 + 13 * \$1.50}{5 + 18} = \$1.58 \tag{2.2}$$

The right hand side of Figure 2.1 represents the supply and demand curves calculated from the ask and bid books respectively, where the orange line is the demand curve and the blue line is the supply curve. Following the orange or blue step functions from left to right, at each step, the height represents the price and the width represents the available quantity at the price point. Lastly, the green cross represents the microprice.

A trade occurs when a bid order price is greater than the *best ask* or when an ask order price is less than the *best bid* on the LOB, this is known as *crossing the spread*. This process is quantity invariant, that is if the latest order price crosses the spread, but the requested quantity exceeds the quantity available at that price, all available quantities are sold crossed price, and the unfulfilled quantities are then placed on the LOB as an outstanding order.

As seen in Figure 2.1, the supply and demand curves do not cross, which therefore represents the fact that no trader is willing to trade at the current prices given in the exchange. The supply and demand curves not crossing means that the LOB is currently at equilibrium, as the best bid is not greater than the best ask.

Figure 2.2: A graphical representation of a Limit Order Book, reproduced from Cliff (2018)

Figure 2.2 indicates that a new order has been placed on the exchange. Interestingly, this order crosses the spread because the ask price \$1.48 was less than the *best bid* in Figure 2.1 so therefore a transaction occurs. Incidentally, the quantity requested was less than the quantity available at \$1.50 so the entire order is fulfilled, and the quantity remaining is updated from 10 to 5. For arguments sake, if the quantity requested was 20 rather than 5, all the quantities that are available at \$1.50 (10) would be used to fulfil the new order and then the LOB would have a new *best bid* of \$1.48 with a qty of 10.

2.1.2 Proprietary trader's Objective

The agents that are taking part in the 'gamification' of the trading session in BSE and BSG are sales traders. The aim of the sales trader is to maximise their consumer surplus relative to their client's order. As mentioned in the previous chapter, the sales trader does not hold a stock of their own, they trade on behalf of their *client*, who provides them with a limit price that they cannot buy or sell above or below respectively. The goal of the sales trader is to maximise the difference between the limit price and the trade price, which is the consumer surplus. In the real world, the sales trader makes commission based on the consumer surplus and the client pockets the rest.

The proprietary trader's objective is that to maximise their own capital by buying and selling stocks for themselves. Whilst initially it may seem like the objective of the prop trader is different to the sales trader and therefore cannot be compared in analytical experiments, the proprietary trader's ability to maximise the difference between the price they buy and sell a stock at is analogous to a sales trader maximising the difference between its limit price and its trade price. This rephrasing of the problem allows this project to directly compare the traditional machine learning approaches to the prop trader developed in this project.

This project will aim to use DRL to maximise the prop trader's objective in increasing its capital in a given BSE/BSG market session.

2.2 Traditional Sales Trader Solutions

Chapter 1 outlined a brief history of various automated trading systems that contributed to the evolution of this field. It is important to gain a deeper understanding of the implementations of some of those trading systems as they serve as the foundations from which this project is built upon, and they are systems that are extensively used to evaluate how effective the system developed in this project is.

Zero Intelligence - Unconstrained (ZI-U) - Gode & Sunder 1993

In 1993 Gode & Sunder introduced a series of *zero-intelligence* algorithmic traders designed for CDA markets. These were trading systems with very low level of intelligence that were placed into CDA markets to observe experimental trading agents in markets where humans do very well in terms of allocative efficiency. The first of which is called Zero-Intelligence - Unconstrained (ZI-U). This is a trader that is not constrained in terms of a budget, but rather quotes a price selected from a uniform random distribution between the minimum market price and the maximum market price, irrespective of the client's limit price. Unsurprisingly, the ZI-U trader was not an effective solution to tackle the sales trader problem.

Zero Intelligence - Contstrained (ZI-C) - Gode & Sunder 1993

Gode & Sunder proposed another algorithm called Zero-Intelligence - Constrained (ZI-C). ZI-C is similar to ZI-U in that it quotes a price from a uniform random distribution. However, instead of the distribution ranging from the market maximum and market minimum price, the distribution ranges from the client's limit price to the maximum/minimum market price- w.r.t if the trader is a seller/buyer. ZI-C is surprisingly human-like in many market conditions as alluded to in 1, however is easily beaten by more sophisticated algorithms.

Zero Intelligence Plus - Cliff 1997

Alongside Cliff's 1997 critique of Gode & Sunder's 1993 paper, he developed the Zero Intelligence Plus algorithmic trader. This trader has a 'profit margin' which is adjusted according to a learning rule. To elaborate, the price P, at which the that the trader submits a buy/ask order, is a product of the limit price L plus some margin M: $L(1.0 + M)$.

It is important to make the distinction that the ZIP trader is *adaptive* unlike ZI-U and ZI-C. The margin M is constantly adjusted depending on the behaviour of the other traders in the market. If the ZIP trader is a seller, and the other sellers are accepting bids *below* P, the ZIP trader decreases its

margin (however constrained to be >0). If the other sellers are accepting bids *above* P, then the ZIP trader increases its margin. The buyers behave inversely to this. The rule that dictates how to adjust M is the Widrow-Hoff learning rule.

Cliff demonstrated that ZIP traders succeed in market conditions where ZI-C had failed, and showed that ZIP traders succeed in those markets.

As mentioned in Section 1.2.1, IBM demonstrated that ZIP outperformed their human counterparts in controlled experiments. Therefore, this project used ZIP as the benchmark to explore if a DRL trader can be implemented which will also outperform its human counterparts.

2.3 Reinforcement Learning

This project transforms the current prop trader problem to one that is well suited to a DRL paradigm. This will be the main avenue of solutions this project explores and thus, it is paramount to provide the technical background for reinforcement learning.

The rest of this section explains the mathematical foundations behind the theory that drives RL-based solutions, with the equations and derivations acquired from Sources [1], [16], and [30], unless stated otherwise.

2.3.1 What is Reinforcement Learning?

Reinforcement Learning is the training of machine learning models (agents) to take actions that interact with a complex environment, to maximise a reward [26]. More formally, the agent learns via interactions with an environment, a policy $\pi(\alpha|\sigma)$ which is the probability to take a particular action α, after observing a state σ that maximises its reward.

A simple architecture diagram describing the flow of RL algorithms is as follows:

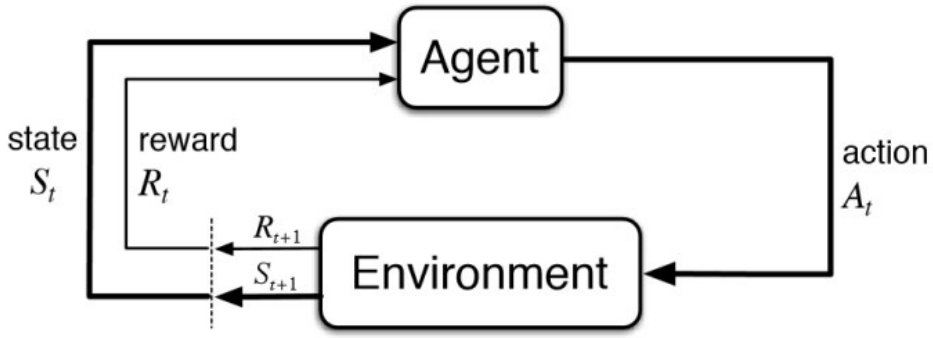

Figure 2.3: Architecture of Reinforcement Learning Environments

Where:

- Environment: is the world that the agent operates
- State S_t: The current obserevation of the environment
- Reward R_t: The feedback from the environment
- Policy: The method to map the state to the agent's action
- Action A_t: The agent's action after observing the state

Markov Decision Process

Most Reinforcement Learning algorithms require a fully observable environment. A Markov Decision Process (MDP) is a mathematical formalisation of this environment and the problem definition as a whole, where almost all RL problems can be formalised as.

A MDP is a 5-tuple which has components that are defined as follows:

- Σ: The set of all states, where a state σ_t is defined as an observation from an environment at time stamp t
- A: The set of all actions possible, where an action α is any action available to the the agent in state σ_t at time stamp t
- Φ: The set of all transition probabilities, where ϕ_{α_t} is the probability that action α_t in state σ_t at time stamp t will lead to state σ' in time stamp $t+1$. $\Phi(\sigma_t, \sigma') = Pr(\sigma(t+1) = \sigma' | \sigma_t, \alpha_t)$. This is a stochastic process because the same state-action transition may not lead to the same state'. This may be due to random factors in the environment or interactions with the other agents, or an inaccurate interpretation of the current state from the agent, which is suitable for this project's environment.
- $\rho_{\alpha_t}(\sigma_t)$: The immediate reward after the transition from state σ_t to an arbitrary state σ'
- $\gamma \in [0, 1]$: This is a discount factor representing the difference in importance between present and future rewards. Depending on the value of γ, it will encode the idea that a reward that is received in the present is more desirable/valuable than the same reward received at any point in the future.

Making MDP a 5-tuple:$(\Sigma, A, \Phi, \rho, \gamma)$.

The goal of a MDP is to optimise a policy $\pi(\alpha|\sigma)$, which defines the probability that the agent should take an action α given an observed state σ, maximising a cumulative reward function of random rewards, i.e the expected discounted sum of rewards over an infinite time horizon. So, an agent during every time step in its existence, tries to maximise the expected return:

$$U(t) = \sum_{t=0}^{\infty} \gamma^t \rho(\sigma_t) \tag{2.3}$$

Equation 2.3 establishes a measure of value for a given *sequence of states*, however, this measure needs to be used to define the value for a *specific state*. This measure ideally represents both the immediate reward, and a measure of the agent 'heading in the right direction', which in this context means to make a profitable trade, to accumulate future rewards.

To summarise so far, the *utility* of a state is the expected reward received in the current state, plus the rewards the agent accumulates on the journey through future states, following a given policy. This definition is framed as an *expectation* because the environment that is specified in this domain is stochastic, so future states are not deterministic. This notion of *utility* is encapsulated in what is called the value function.

$$V_\pi(\sigma) = E_\pi[U(t) | \sigma_t = \sigma] \tag{2.4}$$

This equation can be rearranged with the derivation [16] to give:

$$V_\pi(\sigma) = E_\pi[\rho(\sigma_t) + \gamma(V(\sigma_{t+1})) | \sigma_t = \sigma] \tag{2.5}$$

This now defines a function for calculating the utility of a *specific state* as input, and the function returns a value for entering the given state. This is used in place of the sum of rewards of an infinite time horizon.

In addition to this, the Action-Value function, which is an an estimate of the value given an action, is as follows:

$$Q_\pi(\sigma, \alpha) = E_\pi[U(t)|\sigma_t = \sigma, A_t, = \alpha] \tag{2.6}$$

Considering the fact that the agent is interested in the optimal policy, and the optimal policy is the sequence of actions which maximise the utility until the 'game' is finished, the value function therefore cannot be used in isolation. This is because it only provides the expected return from a specific state and excludes information on which action to take. If the value function is to be used alone, the agent would have to simulate all possible actions to determine which action takes the agent to the state with the highest utility, which is practically impossible in this problem domain because of the continuous state space. The Action-Value function connects the expected returns with actions, which provides the agent with information on what action to take to maximise its utility.

The derivation in [16] allows the Equation 2.5 to be rearranged in such a way that matches the format of a Bellman equation, which by extension allows the action-value Equation 2.6 to be rearranged as a Bellman Expectation Equation, subject to a policy π:

$$V_\pi(s) = E_\pi[R_t + \gamma V_\pi(S_{t+1}|S_t = s] \tag{2.7}$$

The Bellman equation is a condition that is necessary for optimality, historically associated with the optimisation method *dynamic programming* [11]. The equation denotes the 'value' of a holistic problem at a particular point in time, as the payoff of some initial choices and the 'value' of the remaining decision problem, that appear as a result from those initial choices. As is the nature of *dynamic programming* based solutions, large-scale problems are broken down into a sequence of simpler subproblems - which is Bellman's "principle of optimality"[17].

This allows the Value Equation 2.5, in finite terms, to be expressed as:

$$V_\pi(s) = \sum_{\alpha \in A} \pi(\alpha|\sigma) \sum_{\sigma'} \sum_{r} p(\sigma', r|\sigma, a)\{r + \gamma V_\pi(\sigma')\} \tag{2.8}$$

As well as the Action-Value function Equation 2.6, expressed as:

$$Q_\pi(s) = \sum_{\alpha \in A} \pi(\alpha|\sigma) \sum_{\sigma'} \sum_{r} p(\sigma', r|\sigma, a)\{r + \gamma Q_\pi(\sigma')\} \tag{2.9}$$

2.3.2 Optimality

In an ideal scenario, the agent wants to find the optimal solution to the MDP problem. So, the definition of 'optimal' is as follows:

Optimal Value Function. The optimal state-value function $V_*(\sigma)$ is the maximum value function over all policies: $V_*(\sigma) = \max_\pi V_\pi(\sigma)$. In essence, there are all kinds of policies that the agent can follow in the Markov chain, but this means that the agent wants to traverse the problem in such a way to maximise the expectation of rewards from the system.

Optimal Action-Value Function. The optimal state-value function $Q_*(\sigma)$ is the maximum action-value function over all policies $Q_*(\sigma) = \max_\pi Q_\pi(\sigma)$. This means that if the agent commits to a particular action, the action provides the maximum possible return out of every possible traversal thereafter. Critically, if the agent knows $Q_*(\sigma)$, then it has found the best possible traversal and therefore, it has 'solved' the entire problem. So informally, an MDP is solved if $Q_*(\sigma)$ is solved.

Optimal Policy. The optimal policy, is the best possible way for an agent to behave in an MDP. The previous two definitions define *what* the reward is, but the policy defines *how* to achieve the reward. To define what makes one policy better than another, a partial ordering is created over policy space. $\pi \geq \pi' \; if \; V_\pi(\sigma) \geq V_{\pi'}(\sigma), \forall \sigma.$

The theorem [31] summarises that the optimal policy achieves the optimal value function as well as the optimal state-value function.

2.3.3 Challenges to Find Optimal Solutions

The Bellman Optimality Equation is typically used to solve for trivial MDP problems. However there are key challenges that prevent it to be a solution in practice.

- Bellman Optimality Equation is non-linear - as the Expectations are intractable
- No closed form solution (in general)

To deal with these challenges, there are very clever solutions that will be covered in the next section.

2.4 Solving Reinforcement Learning

Model-Free

There are numerous methods designed, developed and tested to solve the MDP problem, such as dynamic programming. However, solutions that are similar in nature require a well defined model and environment. In most practices, the MDP agent does not know the environment and its intricacies, which must be figured out by the agent itself. *Model-Free* solutions give up the assumption that the model is well defined for the agent, which is more ideal in practice.

Model Free Prediction

This is an 'evaluation' task, where given a policy π the agent evaluates it's Value function $V_\pi(\sigma)$ as mentioned in Equation 2.8, which can be done recursively through via the Bellman equation until convergence.

Model Free Control

Control is about the method used to improve on a policy π to find the optimal policy π_*, as outlined in Section 2.3.2. This is achieved via a method called *Policy Iteration*. Unlike the prediction problem which iterates over the Value Function, *Policy Iteration* iterates over the Action-Value function in Equation 2.9. More concretely, this process works by iterating over every $Q_\pi(\sigma, \alpha)$ value and greedily taking the action with the maximum expected return. However, this solution in particular ignores the concept of 'delayed gratification' which would be a scenario where a greedy action for an individual state may not give the agent the maximum expected return, but the states following it yield greater overall reward, which would be ignored with this greedy solution.

In general, solving the Bellman Equation is to recursively solve for Equation 2.8 and Equation 2.9 which is very inefficient as mentioned below Equation 2.6, so other solutions are explained below.

2.4.1 Model Free Prediction: Monte Carlo

The Monte Carlo (MC) method is one of the simpler and intuitive solutions to the reinforcement learning problems.

- MC methods learn directly from experience
- MC methods have no prior knowledge of MDP transitions
- Uses the simplest possible idea: Using sample means from episodic experience

Goal: Given a policy π, learn V_π from episodes of experience: $\sigma_1, \alpha_1, r_1, ..., \sigma_k$ Recall that the value function uses the expected return, however, Monte Carlo uses the empirical sample means as an approximation for the expected return:

$$V_\pi(\sigma) = E_\pi[U(t)|\sigma_t = \sigma] \approx \frac{1}{N}\sum_{i=1}^{N} U_{i,\sigma} \tag{2.10}$$

2.4.2 Model Free Control: Monte Carlo

To solve the control problem, a greedy policy function is used w.r.t to the current value function.

$$\pi_*(\sigma) = arg\max_{\alpha} Q_\pi(\sigma, \alpha) \tag{2.11}$$

The major problem with this approach in this project's problem domain is that the solution may not be a good fit for the problem at hand. This is because the state space in BSG is quite complex and as well as this, the Monte Carlo solution requires a 'reset' back to a specific state (σ_t to effectively approximate the expected returns.

2.4.3 Model Free: Temporal Difference Learning - TD(0)

Temporal Difference Learning (TDL), is an *online*-policy method in which the agent learns by updating its prediction at every time step throughout an episode. This differs to the MC solution as MC relies on episodic experiences as a whole to learn from, rather than learning during an actual episode.

In mathematical notation, this means that the value function $V_\pi(\sigma)$ is updated toward the *estimated* return $\rho(\sigma_t) + \gamma V_\pi(\sigma_t)$:

$$V_\pi(\sigma) \leftarrow V_\pi(\sigma) + \beta(\rho(\sigma_t) + \gamma V_\pi(\sigma_t) - V_\pi(\sigma_{t+1})) \tag{2.12}$$

Where the estimated return is analogous to the Bellman Equation 2.8, $\rho(\sigma_t)$ is the immediate reward, $\gamma V_\pi(\sigma_t)$ is the discounted future reward for the next step and where β is the generalised *learning rate*. The *estimated* return is substituted (otherwise known as the *TD Target*) in for the *real* return unlike in MC methods, which results in the *TD error* of:

$$\delta_t = \rho(\sigma_t) + \gamma V_\pi(\sigma_t) - V_\pi(\sigma_{t+1}) \tag{2.13}$$

Intuitively, this differs with MC methods because in an example where the agent is driving a car and **almost** crashes, but then drives off safely, the MC method would not update the value function in a negative sense. This is because the end goal was not negative. However, with TD learning, the *online* learning policy allows the agent to update the value function for the previous time steps leading up to the incident to change future behaviour. This is desirable behaviour for the prop trader problem as it means that the agent could potentially learn the states that lead up to a sudden drop or rise in price of a stock, which the trader could submit a bid or ask for, which may have led to a loss. In comparison, the MC trader would have to commit to a bad trade to have learnt from the experience.

To solve the control problem for TD Learning, there are various options that can be applied to this kind of learning that follows the 'Generalised Policy Iteration' structure, with the exploration vs exploitation problem trade-off being achieved with the $\epsilon - greedy$ method. They are: *SARSA, SARSAmax, Q-Learning, Expected SARSA*. The following equations have been obtained from [23]

SARSA

This method uses every element of the quintuple of events $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$, inspiring the name "SARSA".

$$Q(\sigma_t, \alpha_t) \leftarrow Q(\sigma_t, \alpha_t) + \alpha[\rho(\sigma_t) + \gamma Q(\sigma_{t+1}, \alpha_{t+1}) - Q(\sigma_t, \alpha_t)] \tag{2.14}$$

This is an *on-policy* method which means that the agent will learn by using the experience sampled by the same policy.

SARSAmax/Q-Learning

This is a *off-policy* TD algorithm. This means that the learned action-value function, Q, directly approximates Q_*, the optimal action-value function. This occurs independent of the policy that is being followed.

$$Q(\sigma_t, \alpha_t) \leftarrow Q(\sigma_t, \alpha_t) + \alpha[\rho(\sigma_t) + \gamma \max_{\alpha} Q(\sigma_{t+1}, A) - Q(\sigma_t, \alpha_t)] \tag{2.15}$$

Expected SARSA

This is an *on-policy* TD control algorithm. It is similar to Q-learning, with a slight variance. Instead of the maximum over next state–action pairs, the algorithm uses the expected value, taking into account how likely each action is under the policy being used. Given the next state, σ_{t+1}, this algorithm moves deterministically in the same direction as SARSA moves in expectation.

$$Q(\sigma_t, \alpha_t) \leftarrow Q(\sigma_t, \alpha_t) + \alpha[\rho(\sigma_t) + \gamma E_\pi[Q(\sigma_{t+1}, \alpha_{t+1})|\sigma_{t+1}] - Q(\sigma_t, \alpha_t)] \tag{2.16}$$

All three control solutions for TD learning follow the same *Policy iteration* procedures as outlined earlier in this section.

2.4.4 Value Function Approximation

In an ideal scenario, the discussed methods so far (MC and TD) would work for all reinforcement learning problems, however, there is a critical hurdle that these solutions do not account for in real world problems. The solutions discussed thus far solve the reinforcement learning problems using *tabular methods* i.e treating $V_\pi(\sigma)$ and $Q_\pi(\sigma, \alpha)$ as dictionaries, which is simply not a feasible solution for problems with a large state space. As examples, Backgammon has 10^{20} states, and the game AlphaGo Zero solved, 'Go', has 10^{70} states. Backgammon is considered a small game, which provides some added perspective as to the scale of problems that reinforcement learning can solve.

This problem is tackled using *Value function approximation*:

$$\hat{V}(\sigma, \mathbf{w}) \approx V_\pi(\sigma) \tag{2.17}$$

$$\hat{Q}(\sigma, \alpha, \mathbf{w}) \approx Q_\pi(\sigma, \mathbf{w}) \tag{2.18}$$

These functions are parametric function 'approximators', where $\mathbf{w}$ are the weights. So now, the parametric regression allows the approximator to generalise the previously denoted dictionaries as a general fitted function, which allows the agent to query the function for unseen/infinite state spaces.

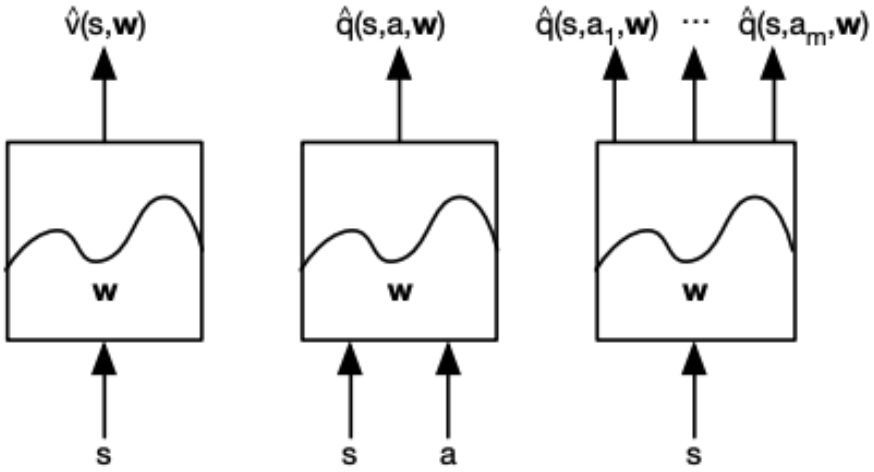

Figure 2.4: Types of value approximators, reproduced from Silver [29]

Figure 2.4 shows the architectures of the varying types of approximators. The leftmost architecture represents a *Value function* approximator, where a 'black-box' function spits out the 'value' of being in state *s*. The middle architecture showcases the *action-value* approximator, where again, a 'black box' function takes in the state the agent is in, and the action it is considering, and it spits out how good that action is. Lastly, the rightmost architecture represents an architecture that will give the agent the action-value for all possible actions in the given state *s*.

The natural step to take at this point is to choose the type of 'black-box' that would achieve this approximation. This project explores the usage of neural networks. Since neural networks are function approximators, they play a crucial role in RL for scenarios where the state or action spaces are too large to be completely known. They are used in the RL context to approximate the *value function* or *action-value functions* by optimising weights along gradients that promise less error.

The methodology to optimise these weights is Stochastic Gradient Descent(SGD). To perform SGD, a differentiable function is required with input parameter $\mathbf{w}$, J($\mathbf{w}$) , where the gradient of J($\mathbf{w}$) is defined as:

$$\nabla_{\mathbf{w}} J(\mathbf{w}) = \begin{pmatrix} \frac{\partial J(\mathbf{w})}{\partial \mathbf{w}_1} \\ \vdots \\ \frac{\partial J(\mathbf{w})}{\partial \mathbf{w}_N} \end{pmatrix} \tag{2.19}$$

In a supervised learning example, the goal is to find parameter vector $\mathbf{w}$ that minimises the mean-squared error (MSE) between the approximated value function $\hat{V}(s, \mathbf{w})$ and true value $V_\pi(\sigma)$, to then use SGD to find local minima as defined in Silver's [29].

$$J(\mathbf{w}) = E_\pi[(V_\pi(\sigma) - \hat{V(\sigma, \mathbf{w})})^2 \tag{2.20}$$

However, considering the context is reinforcement learning, there is no objective truth for $V_\pi(\sigma)$. Instead, there are only rewards to indicate to the agent whether or not the actions are good or bad, which are acquired through experience. Luckily, the previous subsections covered precisely this. The objective function can be defined with the help of the TD/MC method defined earlier as:

- For MC the 'objective truth' becomes the return U_t: $\Delta\mathbf{w} = \beta(U_t - \hat{V}(\sigma_t, \mathbf{w}))\nabla_{\mathbf{w}}\hat{V}(\sigma_t, \mathbf{w}$
- For TD the 'objective truth' becomes the TD target U_t: $\Delta\mathbf{w} = \beta(\rho(\sigma_{t+1}) + \gamma\hat{V}(\sigma_{t+1}, \mathbf{w}) - \hat{V}(\sigma_t, \mathbf{w}))\nabla_{\mathbf{w}}\hat{V}(\sigma_t, \mathbf{w})$

2.4.5 Policy Gradient

The last subsection explained the use of value approximation functions in Equations 2.17 and 2.18, to pick actions greedily with $\epsilon - greedy$ methods to acquire the max Q_π during policy evaluation - summed up as policy iteration.

A more natural and direct approach may be to directly parameterise the *policy* rather than the value functions - This is known as Policy-Based Reinforcement Learning. So now the policy can be manipulated directly, by controlling and learning the parameters to affect the distribution by which actions are picked: so the policy can be directly modelled as:

$$\pi_\theta(\sigma, \alpha) = P[\alpha|\sigma, \theta] \tag{2.21}$$

The goal for policy based reinforcement learning is that for a given policy $\pi_\theta(\sigma, \alpha)$ with paramters θ, find the best set of parameters θ. However, the notion of 'best' is ambiguous, which is why an objective function needs to be defined. There are numerous kinds of functions depending on the type of environment, however, considering that a market session is an episodic environment, which has the notion of a *start-state*, the most suitable objective function is as follows:

$$J_1(\theta) = V^{\pi_\theta}(\sigma_1) = E_{\pi_\theta}[v_1] \tag{2.22}$$

This objective function encodes the notion that if the agent always starts in some start state σ_1 or if the agent has a distribution over σ_1, how does the agent maximise the total end reward.

To find the θ that maximises the objective function $J(\theta)$ to find the optimal policy π_*, the project considers gradient *ascent*.

Policy gradient algorithms search for a local maximum for the objective function $J(\theta)$ by ascending along gradient of the policy, w.r.t to the parameters θ

$$\Delta\theta = \beta \nabla_\theta J(\theta) \tag{2.23}$$

where $\nabla_\theta J(\theta)$ is the **Policy Gradient** and β is the learning rate

$$\nabla_\theta J(\theta) = \begin{pmatrix} \frac{\partial J(\theta)}{\partial \theta_1} \\ \vdots \\ \frac{\partial J(\theta)}{\partial \theta_N} \end{pmatrix} \tag{2.24}$$

Monte Carlo Policy Gradient

To compute the policy gradient analytically by exploiting the likelihood ratio trick:

$$\begin{aligned} \nabla_\theta \pi_\theta(\sigma, \alpha) &= \pi_\theta(\sigma, \alpha) \frac{\nabla_\theta \pi_\theta(\sigma, \alpha}{\pi_\theta(\sigma, \alpha)} \\ &= \pi_\theta(\sigma, \alpha) \nabla_\theta \log_{\pi_\theta}(\sigma, \alpha) \end{aligned} \tag{2.25}$$

Which allows the equation to represent a familiar term in statistics and machine learning, the score function [9] which allows the gradient ascent to maximise the log likelihood to tell it how to adjust the gradients.

For **discrete** actions, a *Softmax* Policy is used, where the actions are weighted using a linear combination of features $\phi(\sigma, \alpha)^T \theta$, and then to convert this linear combination to a probability function, these are exponentiated:

$$\pi_\theta(\sigma, \alpha) \propto e^{\phi(\sigma,\alpha)^T \theta} \tag{2.26}$$

To make the score function:

$$\nabla_\theta \log_{\pi_\theta}(\sigma, \alpha) = \phi(\sigma, \alpha) - E_{\pi_\theta}[\phi(\sigma, \cdot)] \tag{2.27}$$

Intuitively this means that the feature for the action that the agent actually took, minus the average feature for all the actions the agent might have taken. So, the score function summarises how much more of the given feature it has more than usual, and if it gets a more reward, then adjust the policy to do it more.

Actor Critic Policy Gradient

An alternative method is also explored in this project. In the MC Policy Gradient method, the notion of the action-value functions defined earlier in this chapter are completely ignored, and focuses on the policy itself. In addition, the MC policy gradient method also has a high amount of variance. Contrastingly, the Actor-Critic method reintroduces the notion of the action-value function via value function approximation outlined in Section 2.4.4, whilst also aiming to reduce the variance of MC methods.

The main component in this method is that a *Critic* is used to explicitly estimate the action-value function, rather than using the returns of the episode experience.

$$Q_w(\sigma, \alpha) \approx Q^{\pi_\theta}(\sigma, \alpha) \tag{2.28}$$

The Actor-Critic algorithms maintain two sets of parameters:

- Critic - Updates the action-value function parameters: w
- Actor - Updates policy parameters θ, in the direction suggested by the critic

The actor is the entity that decides what action to take in the environment. The critic merely observes the actor, however, it evaluates the actor's decisions. The critic as an entity is introduced in order to reduce the high variance experienced from using MC Policy Gradient methods. This methodology is unique in the sense that it combines the previously explained Value-based methods, as well as the newly defined Policy-based methods. Unsurprisingly, this method follows the overarching structure of policy gradient methods, however, they are defined as an *approximate* policy gradient:

$$\begin{aligned} \nabla_\theta J(\theta) &\approx E_{\pi_\theta}[\nabla_\theta log_{\pi_\theta}(\sigma, \alpha) Q_w(\sigma, \alpha)] \\ \Delta\theta &= \beta \nabla_\theta log_{\pi_\theta}(\sigma, \alpha) Q_w(\sigma, \alpha) \end{aligned} \tag{2.29}$$

The main idea here is that the actor adjusts the policy parameters in the direction that, according to the critic, will get more reward. So, the true action-value function is replaced by the critic's approximation of this function using a neural network.

The natural question is how the critic is estimating the action value function, luckily this was covered in subsections 2.4.1 and 2.4.3, because this is a policy evaluation problem.

2.4.6 Deep Deterministic Policy Gradient - (DDPG)

The idea behind DDPG was first introduced in 2015 [22], in which the technique is explained and applied to a series of games that require a continuous action space, delivering excellent results.

DDPG retains several techniques from a Deep Q-Network (DQN) based solution - which itself is a technique based on the *SARSAmax/Q-Learning* technique outlined in Section 2.4.3, but for larger state spaces. In addition, DDPG makes use of the actor-critic architecture outlined in 2.4.5. However, considering these problems work on stochastic policies, DDPG is an efficient gradient computation for deterministic policies.

DDPG builds on the ideas that drive DQN, such as utilising a *replay buffer*, and having a target Q-network, however, with crucial differences. A comparison of the architectures of DQN and DDPG is given below:

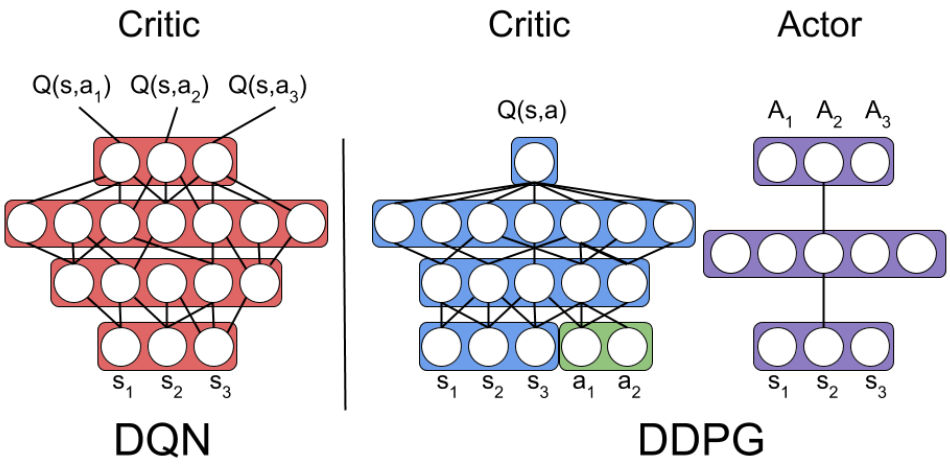

Figure 2.5: Architecture Comparison: DQN vs DDPG, borrowed from [24]

On the left hand side of Figure 2.5, it shows that a DQN critic network takes in a *state* as input, and provides the *Q-Value* for every state-action pair for that given state. However, this would require an infinite amount of neurons for the output in continuous environments, which is not ideal.

DDPG tackles this by having a critic network that takes in a state-action pair and provides the *Q-Value* with a singular neuron as output. As mentioned, the critic takes in a state-action pair, where the action is acquired from the *actor* network, which takes the state as an input and outputs an action. The actor and critic are trained to converge the networks to the optimal actor and critic.

Training the Critic

To train the critic, the TD-error is minimised:

$$\delta_t = \rho(\sigma_t) + \gamma Q_(\sigma_{t+1}, \pi(\sigma_{t+1})|\theta) - Q(\sigma_t, \alpha_t|\theta) \tag{2.30}$$

In doing so, a minibatch of N samples of stored transitions $\{\sigma_t, \alpha_t, \rho, \sigma_{t+1}\}$ and a target network Q' to compute $y_t = \rho(\sigma_t) = \gamma Q'(\sigma_{t+1}, \pi(\sigma_{t+1}|\theta')$

Which is then used to update the network parameters θ by minimising the loss function:

$$L = \frac{1}{N}\sum_i (y_i - Q(\sigma_i, \alpha_i|\theta))^2 \tag{2.31}$$

This represents the difference between the desired value from the target network and the Q-network.

Training the Actor

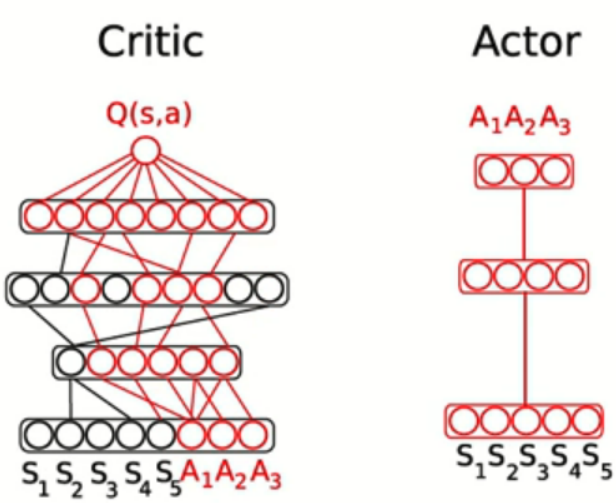

Figure 2.6: Training Actor with DDPG

The idea is based on the deterministic policy gradient theorem proposed by Silver [33]:

$$\nabla_{\mu}\pi(\sigma,\alpha) = E_{\sigma}[\nabla_{\alpha}Q(\sigma,\alpha|\theta)\nabla_{\mu}\pi(\sigma,\mu)] \tag{2.32}$$

The intuition behind the equation is as follows. Assuming that the agent is in a particular state σ and the actor network is proposing a particular action α (as shown in Figure 2.6), the critic evaluates this action. Now, to the improve the action taken by the actor, it proposes a slightly different action and is critiqued by the critic in the form of a higher or lower Q value.

Therefore, taking the gradient w.r.t to *actions* $\nabla_{\alpha}Q(\sigma,\alpha|\theta)$ to backpropagate over the weights to produce an error signal, that will update the actor network's weights. Taking the gradient with respect to the action is perfectly possible as weight's and input's play a symmetry role in NNs.

Exploration

As mentioned previously, the actor learns by slightly changing its action, to observe a change in the Q-value of the action from the critic. This is achieved by adding random noise to the action using an *Ornstein-Uhlenbenk* (correlated) noise process, where the noise at the current time step, is related to the noise at the previous time step. This noise process aims to tackle the age old problem of exploration vs exploitation.

It is important to note that convergence on this technique as a whole is considered 'fragile' because if the training process is unlucky at the start with minimal convergence, the actor will stay poor, and therefore the critic will stay poor, which will have the ripple effect in the time taken to train as it may take a significant amount of time to converge to a good policy.

2.4.7 Discrete vs Continuous Action Spaces

The methodologies described in the latter part of the chapter are versatile because they support problem scenarios which have discrete and continuous action spaces. In the scenario where the action space is discrete, the solutions allow for a stochastic probability distribution between all the possible actions. For example, if there are five possible actions that the agent can take throughout the 'game', the probability of these actions will change as the agent learns from experience, and these actions are sampled from the updated distribution.

Methods which can optimise for continuous action spaces are applicable to many real world scenarios which make the implementations of the policy gradient methods desirable. Now, instead of a distribution over a finite set of actions, the agent can now optimise for actions which involve real numbers.

Both types of action spaces are applicable to solutions to solve the prop trader problem, and both are explored throughout the project.

2.5 Summary

This chapter aimed to summarise the difficult technical challenges of the prop trader problem as well as the plethora of techniques that are traditionally used to solve reinforcement problems. Additionally, the chapter started by providing the necessary context to understand how reinforcement learning is formally defined as a MDP problem, and then the consequent sections built up the necessary knowledge to understand the main solutions outlined in the latter sections of the chapter. The project will aim to implement these techniques and evaluate their effectiveness in this problem domain.

Chapter 3

Project Execution

This chapter intends to describe the entire journey of this project - from the birth of the idea of the project, to the implementation and critical analysis of results.

The convention for a master's thesis is to separate the project execution and the critical evaluation of the implementation to form their own chapters. However, given the iterative nature of the project, the project execution heavily intertwined with the analysis of results, which by extension dictate the next iteration of the implementation of the trader. Hence, the decision was taken to merge the conventional chapters three and four into one chapter.

3.1 Origin

In a Bristol University Computer Science fourth year module called *Internet Economics and Financial Technology*, lecturers conducted Smith's market experiments [35] and used students on the course to demonstrate that human traders in a continuous double auction gravitate to a market equilibrium price when trading under a time constraint. Furthermore, the course outlined outstanding algorithms (ZIP, MGD, AA, refer back to Section 1.2), that were developed and outperformed their human counterparts in this very experiment - this is where the motivation and intrigue behind the project was born.

Considering the fact that these algorithms were sales traders who traded with a limit price given by their clients: what if there was an algorithm designed for a proprietary trader that similarly outperformed humans at this task?

3.2 The Deeply Reinforced Trader

As a reminder, RL (and by extension, DRL) is a type of machine learning technique that allows an agent to learn in an interactive environment, via trial and error using feedback from its own actions and experiences. The overall process process is outlined in Section 2.3.1.

Given the architecture of RL algorithms, the project aimed to produce all aspects of the RL paradigm with carefully selected features explained in the rest of this section.

3.2.1 Training Environment

Considering this architecture and problem definition, this project required a framework that processes a trading environment in this exact manner. Luckily, Furlan-Falcao outlined a framework developed as part of his thesis - called Bristol-Stock-Gym (BSG). BSG is a simulated trading environment that is a refactor of Cliff's Bristol Stock Exchange (BSE), specifically built to provide a framework to train RL models in the context of BSE. This framework is analogous to the frameworks that *OpenAI*'s Python library *Gym* provides to train RL models in this manner.

```
for i in range(Episodes):
        environment = Environment(traders_spec, order_sched,time_step = time_step, max_time =
            end_time, min_price = 1, max_price = end_time, replenish_orders = True)
        done = False
        observation = environment.reset()
        while not done:
            action = trader_strategy(state)
            observation_, reward, done, info, balance = environment.step(action)
            trader.store_rewards(reward)
            observation = observation_
            trader.learn()
```

Listing 3.1: Main loop to step through trading environment

To elaborate on the implementation above, the environment is initialised via the following steps: Initialising an empty `Exchange`, populating the market with traders (specified by a `traders_spec` which defines what types of traders to populate the market with), resetting the market session time to `1` and getting the initial observation. During an *epsiode*, the trader makes an observation, takes an action to interact with the environment, and records the reward acquired for entering a new state. This process spans over a single time step, which when repeated, makes up an entire episode. One step through an environment allows each trader in the market to react the current market conditions by placing an existing order or by updating it's internal variables. If a trade occurs, both parties are notified, and then this trade is published in the lob. This process is repeated till the max time `1000` is reached, which signifies the end of one *Episode*.

This project utilises Furlan-Falcao's BSG heavily in this manner, whilst training several different types of agents with different *Trading Strategies* shown in the code snippet above.

3.2.2 Designing the State

The state in an RL environment is an observation made by the agent which essentially is a snapshot at time t of details in the environment that are useful to inform the agent of the state of the current environment. So the design of this state is crucial to the success of the trader's interpretation of the current environment, to best inform its decisions to maximise its reward.

In BSG, the environment provides an observation which is as follows:

```
def get_public_lob(self, time):
        public_data = {}
        public_data['time'] = time
        public_data['bids'] = self.bids.lob_anon
        public_data['asks'] = self.asks.lob_anon
        public_data['tape'] = self.tape
        return public_data
```

Listing 3.2: BSG Environment Observation: LOB

The data provided by this observation, and their significance, are outlined in section 2.1.1. This observation includes the current *Asks* and *Bids* in the market, the current time, and the tape - which captures all the trades that have taken place and all the cancelled orders by the traders in the market.

The problem with this observation from the environment is that it is of a variable size. At time $t = 1$ the *tape* would be of length 0, similarly to the *bid* and *ask* books. However, given an arbitrary time $t = k$, the size of these data structures can vary quite heavily. The reason as to why this is a problem is that most Deep Learning model architectures require a fixed input size, so the default observation data provided is not appropriate to use. It is noteworthy, that ideally the entirety of this observation can be used as an effective input as it is the maximum amount of information the agent could use. Though, in a practical sense, this is unmanageable as the data increases linearly with time, which would make the agents state space too large.

Latent Representation: Auto Encoder

Research indicates that the local spatial structure in limit order books are key components in predictive analysis about the future behaviour of markets [34]. The paper explores and demonstrates how data in a LOB beyond the best bid and the best ask (which are directly used to calculate the midprice and microprice) can be used to regress to future prices with significant accuracy. This is down to the lower levels of the LOB indicating the varying levels of aggressiveness in the traders since they are more prone to cancellations and updates. These events could be informative to a successful trader in BSG as it may be an excellent way to for the trader to predict buy and sell opportunities. Thus, the design of the state space incorporated this ideology.

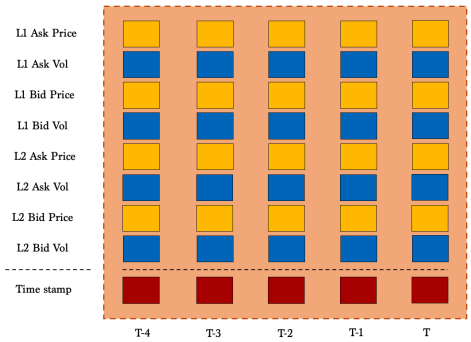

Figure 3.1: Capturing LOB Data

To capture trends in the LOB, the five latest changes to selected details in the LOB are stored as a matrix. The matrix is formed of five, *1x9* vectors which contain the price and volume of level one and level two bid and asks, and the current time stamp for the values that are currently on the LOB as shown in Figure 3.1. It is important to note that the time steps are not necessarily consecutive, they are merely the latest five time steps in which any change occurs to the values (excluding the time stamp value in the vector) in the LOB.
However, similarly to the case of using the entirety of the observation as the state space, this is a complex and high dimensional state space which will increase learning time significantly for the trader's models.

To deal with this high dimensional state space, this state space heavily reduced via an Auto-Encoder (AE).

An AE is a neural network (NN) that attempts to learn to copy its input to its output. It manages this task because it is constituted of two main parts: an encoder that maps the given input to a smaller latent code, and a decoder that maps this latent code to a reconstruction of the original input. The role of the AE is to find an abstract and low-dimensional representation of the input to allow for easier learning and to compress and extract the crucial information out of an otherwise large state space. This is crucial in this problem scenario as the trends that would be apparent in the matrix outlined in Figure 3.1 would take significant time to learn with a matrix of size *9x5*

The architecture of the implemented AE is as follows:

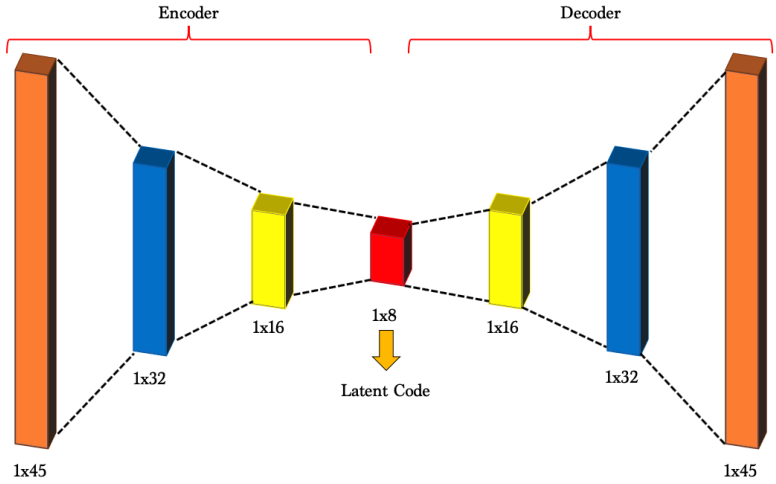

Figure 3.2: Architecture of Auto-Encoder

The *5x9* input matrix is flattened to be an input into a *linear layer* which is then compressed via a series of *tanh* activation functions and linear layers - which were the constituents of the encoder. The compressed latent code is a vector of size *1x8*, which is then decompressed via a series of *tanh* activation functions and linear layers, into the AE model's approximation of the input - which made up the decoder. The latent vector served as the final input used for the DRL trader.

The model was trained on inputs collated from twenty market sessions which amounted to a dataset of size 1,032,290 of stacked *5x9* matrices, which was split with a ratio of 7:3 to form the training and test data set respectively. The training process for the AE works to minimise the loss function of a *mean squared error* between the output of the decoder and the original input. Stochastic Gradient *descent* was used to optimise the weights in the model to minimise the loss function. The model was then trained till convergence to a test loss of less than 0.00112 and saved for later usage in future market sessions.

The latent state representation was then concatenated to a vector consisting of the latest 8 trades that occurred in the market session. The design decision behind this is so that the trader can learn the relationships between the exemplar profitable trades that occurred in the market session and the other eight features that were extracted from the AE.

To incorporate more definitive and informative features about the current state of the LOB, two more features were added to the state space. The first feature was the midprice of the market at the current time stamp t. This is selected to inform the trader of what the best estimate of the price of the commodity is, based on the orders placed on the LOB. The second additional feature was the current *position* of the trader. The *position* of the trader is encoded as a normalised value between -1 and 1. This value represents if the trader has an open position. To elaborate, if the trader has entered a trading position by buying a commodity at price x, the current position is encoded as $[0.x]$ (the position would be encoded as $[-0.x]$ if the trader entered a trading position by selling a commodity at price x). The motivation behind adding these two values is for the trader to learn the connection between its current position and the appropriate action to take, for example if it has just bought a commodity, it now needs to sell it, and vice versa.

The final form of the state space becoming a 18-tuple input vector as follows:

$$\begin{aligned}[AE_1, AE_2, AE_3, AE_3, AE_4, AE_5, AE_6, AE_7, AE_8,\\ Trade_1, Trade_2, Trade_3, Trade_4, Trade_5, Trade_6, Trade_7, Trade_8,\\ midprice_t, position_t]\end{aligned} \tag{3.1}$$

Where AE_{1-8} are the data points provided by the latent state representation *1x8* vector from the AE.

This latent representation of the LOB tackled with the issues outlined earlier regarding the varying sizes of input. The 18-tuple also represents a concise and dense vector of information that the trader can utilise to inform its trading decisions.

3.2.3 Discretised Strategy

As detailed in Section 2.4.7 the trading context allows for varying implementations of DRL techniques that utilise discrete and continuous action spaces. The discrete and continuous action spaces can be manipulated in the design of a `Trader_strategy` function, which will convert the output of a neural network to an action compatible with BSG.

The first approach taken was to discretise the action space to reduce the entire trading process of the agent down to a selection of three actions based on the observed state space.

The actions are as follows:

- Action 0: NULL
- Action 1: Place a bid order
- Action 2: Place an ask order

The discrete action space simplifies the trading process and reduces it down to *entering* and *exiting* trading positions. The problem to solve for the trader then is to time the entries and exits so that they result in profitable trades. The missing information from this strategy is that the price at which the trader submits a bid or ask order is undefined. To solve this issue, if the trader decides to place a bid/ask order (via Actions 1/2 respectively), the current $midprice_t$ (midprice at time stamp t) of the commodity is used as the price of the order. It is important to note that if the trader decides to place a buy order at price: $midprice_t$, the order may not necessarily lead to a transaction, as it will have to be processed by the exchange and matched with another trader's order.

Initially, Action 0 may seem useless. However, it is a crucial action as it will allow the trader to bide its time and wait for the perfect circumstances to submit orders to the LOB and can allow the trader to wait for the prices on the LOB to favour a profitable trading scenario.

This turns the problem akin to a classification task, as the agent would ideally learn what state correlates to a buy/sell opportunity effectively to perform profitable trades.

3.2.4 Vanilla Reward

One of the most crucial aspects in DRL is to formulate an effective reward system that enables the trader to learn the desirable behaviour. This is not a trivial task as it is one of the central challenges described in Section 1.4. A perfectly reasonable design in the reward system may not yield the expected behaviour from the trader, so it is important to find the optimal system that allows the trader to learn to behave appropriately. This is analogous to supervised learning in which a network is fed data alongside truth labels; in this context, the trader must learn these labels by itself with the help of a well tuned reward system.

The first reward system changed the reward r as follows:

$$R(t) = \Delta TradePrice_{t=time\_of\_sell, t=time\_of\_buy} \tag{3.2}$$

This is simply the difference in the price at which the trader sold the commodity at and the price at which the trader bought the commodity at.

- $R(t) = 0$: This will be the most common reward acquired by the trader per time step, as it implies that the trader has not engaged in any trade. This may be down to the fact that the most recent bid/ask order has not crossed the spread as explained in Section 2.1.1 or that no order has been submitted by the trader. An edge case scenario of the trader receiving this reward is if a trade has taken place, but the trader has 'broke even' which means that the trader bought and sold the commodity at the same price, meaning no profit or loss.
- $R(t) > 0$: This reward means that the trader has made a profitable trade. Ideally the trader constantly acquires this reward throughout a trading session. Trivially, a profitable trade is rewarded with positive reward, however, the vanilla reward is the most basic form of reward as it is a *1 to 1 mapping* between the profit acquired and the reward received.

- $R(t) < 0$: This reward indicates that the trader engaged in a trade that incurred a loss. Considering the ZI-C and ZIP algorithms have the hard coded constraint of not trading at a loss, this reward aims to replicate this behaviour by penalising bad trades.

This is denoted as a *Vanilla Reward System* as this is the simplest form of reward system for a trader, which acts as a good starting point for the project.

3.2.5 The PG Vanilla Agent

In Furlan-Falcao's thesis, a Policy Gradient (PG) agent was implemented which provided interesting results. However, the thesis' implementation of the PG Agent explored a continuous action space to train the model to price asset at every time step. This project's implementation of the PG agent will interact with the environment via the discrete action space as defined in Section 3.2.3. This implementation is based on the Monte Carlo Policy Gradient explained in Section 2.4.5, which means that an agent learns after completing an entire 'episode' and using that experience to adjust the model parameters via gradient *ascent*.
This model is denoted as *PG Vanilla* as it incorporated the vanilla reward system.

3.2.6 Analysis and Evaluation of PG Vanilla

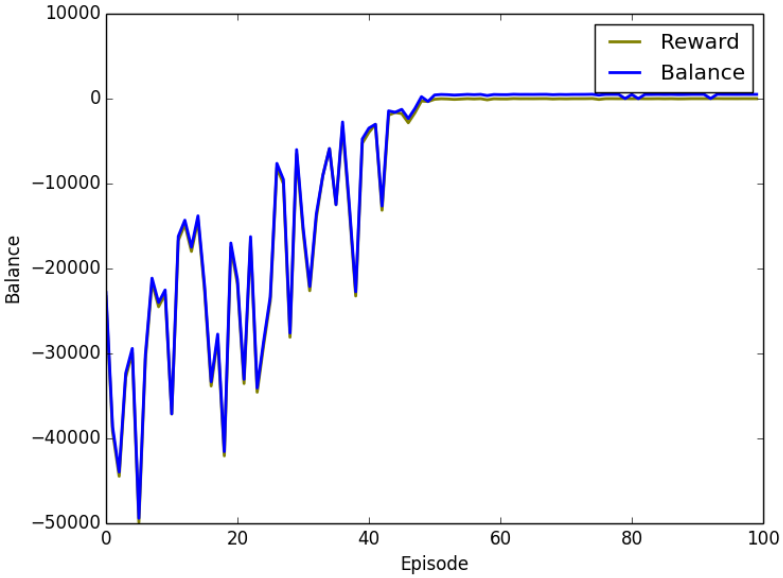

Figure 3.3: Total reward over 100 iterations of the PG-Vanilla trader in a market evenly populated with ZIC, ZIP and GVWY traders.

The graph appeared to indicate promising results as the trader's balance steadily increased from from a trough of approximately -£50,000 in episode five, where the trader constantly engaged in trades that result in a loss, to episode fifty-five which indicated that the trader breaks even. The plotted blue line represents the balance. This is parallel to the line that plots the reward, however there is a constant difference of £500 because that is the balance that the trader starts off with.

However, upon inspection, the steady increase in balance was not down to the trader engaging in better trades, but down to the fact that the trader engaging in less trades.

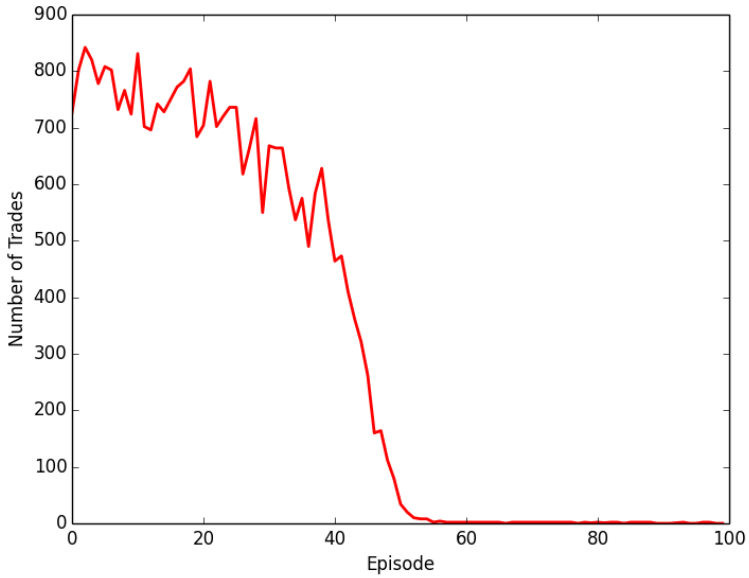

Figure 3.4: A plot of the number transactions the PG Vanilla agent engaged in during each episode

This was an anticlimactic result but served to be informative about the learnt behaviour of the trader. The trader began the learning process by exploring the action space randomly choosing discrete actions: BUY/SELL/NULL. This resulted in the trader engaging in a large number of trades, with a peak of 842 trades in episode two to a fast decline of an average of 0 trades after episode fifty.

To perform a more complete analysis of the results, it was crucial to figure out if the trader improved the rate at which it traded profitably, out of the trades that it was involved in. This was an important step because it provided an oversight as to whether the trader, in conjunction with learning to trade less, was learning to trade profitably too.

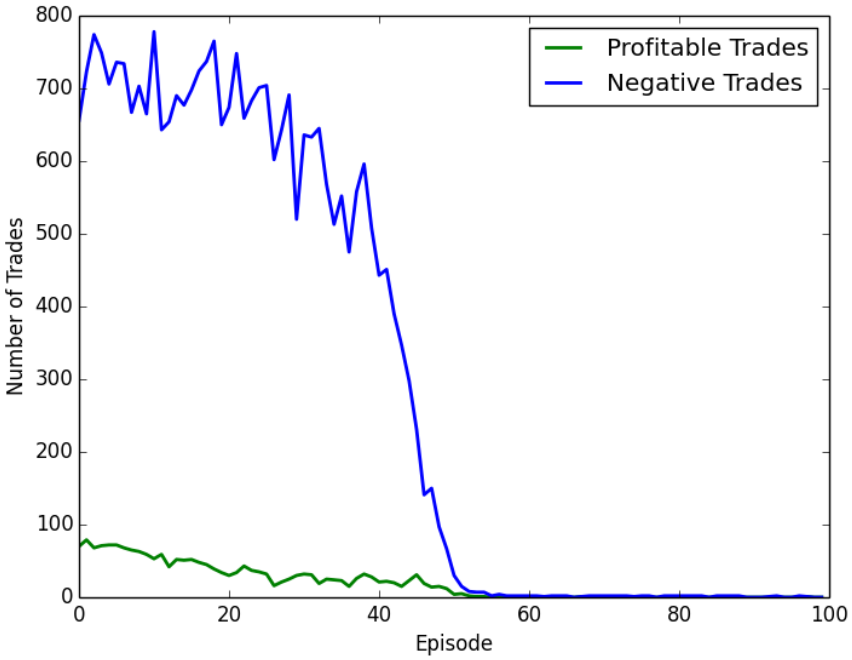

Figure 3.5: A plot of the number of profitable trades and trades that incur a loss for the trader

Figure 3.5 suggests that the trader did not improve in its ability to make a profitable trade. The hard coded property of ZI-C and ZIP traders to not trade at a loss was not learnt by the trader in its experience over 100 iterations. Therefore, the rate at which the trader traded profitably did not increase relative to the number of trades. This result was interesting nonetheless. Since majority of the rewards received by the trader had been negative, it seemingly had learnt to increase its total reward by not trading at all. This is exactly like the case of the Lunar landing analogy given in Section 1.4 in which the agent learnt to avoid a landing penalty by not landing at all. All in all, this is not desirable behaviour as the trader should ideally learn to trade *profitably* and *frequently*.

It is also interesting to note that the average loss per bad trade did not steadily improve as the trader took part in less trades. The average loss per negative trade was £86.83 in episode five, despite engaging in roughly half the number of trades, the average loss per trade was £84.12. This is an indication that the trader was not learning to engage in better quality trades, and thus was truly just learning to engage in less trades to avoid losses.

It was important at this stage to evaluate and scrutinise the implementation to figure out which aspect of the MDP was causing this behaviour. The most obvious factor for the reasons behind this behaviour is the reward system. Expecting the system to learn what a good trade is via a simple 'Vanilla' profit and loss based reward system seemed to be unrealistic - as it lacked guidance and incentives for the trader to learn what good trades are. Other factors include: hyper-parameters of the models, the model architectures, and the state-space design. Ideally, all these factors would be explored to improve the model, however, given time constraints, and because the chosen reward system was rather simplistic and that reward systems play a crucial factor in the behaviour of a DRL-based trader, it was natural to explore this factor further with experimentation.

3.2.7 Optimising The Reward Function

Whilst creating reward systems, the intended behaviour of the agent in training may not be realised, which Bonsai denotes as the 'cobra effect'[3]. To elaborate, the article mentions that within the realm of RL, the agents behave as they are incentivised which may not align with the programmers intentions.

After analysing the current reward system outlined in Section 3.2.4, it became apparent that it does not guide the trader to learn what a good trade looks like before the eventual P&L reward after buying and selling the commodity. This would mean that the trader would have to engage in two trades before eventually finding out whether it was a good or bad trade. In hindsight, considering that the trader constantly engaged in bad trades, it was no surprise that the trader stopped engaging in trade so that it minimises its loss rather than to maximise its gain.

To tackle this issue, a new reward system was devised as follows:

For notation's sake, the Vanilla Reward System was renamed.

$$U(t) = \Delta TradePrice_{t=time\_of\_sell, t=time\_of\_buy} \tag{3.3}$$

The new reward $R(t)$ system became:

1. R(t) = -10 : `if there is no order sent to the exchange`
2. R(t) = 10*U(t) : `if U(t) > 0`
3. R(t) = 2*U(t) : `if U(t) < 0`
4. R(t) = 40 :`if (Action == SELL and trader.position == BOUGHT and current_midprice >price bought )`
5. R(t) = 40 :`if (Action == BUY and trader.position == SOLD and current_midprice <price sold )`
6. R(t) = -40 :`if (Action == SELL and trader.position == BOUGHT and current_midprice <price bought )`

7. R(t) = -40 :`if (Action == BUY and trader.position == SOLD and current_midprice >price sold )`

The motivation behind a reward system of this kind was to provide more detail and guidance for the trader to learn what constitutes of making a good trade.

Considering that the PG Vanilla agent eventually ceased to engage in any trading activity, reward 1. was devised to prevent the trader from remaining inactive in the trading session. This acts as a punishment for not submitting orders to the market.

Reward no. 2 is designed to amplify the rewards of profitable trades. The motivation behind this was to place heavy importance and reward on the actions that lead to profitable trades - considering the ratio of good to bad trades. The same logic was applied to punishing the trader for bad trades, however, a smaller multiplicative factor was used. This was chosen deliberately as the magnitude of the average loss for a bad trade was greater than the magnitude of the average profit for good trades as outlined in Section 3.2.8, thus resulting in a greater weighting for positive rewards than negative rewards.

The last four features in the reward system make up the guidance element talked about previously. Since the trader's willingness in its actions to buy or sell may not necessarily translate into a trade (therefore no reward), this reward system rewards the trader based on its intention/action, regardless of whether a transaction had taken place. If an order, comprised of the current midprice and a bid/ask type, is sent to the exchange that favours the trader to make a profitable trade, it is positively rewarded. Conversely, it is punished if the action chosen leads to an order submitted to the exchange which would lead to a loss. This idea to reward the *intention* of the trader in conjunction with the P&L was chosen to increase the probability in the trader engaging in better quality trades and to learn the hard coded constraints in the ZI-C and ZIP algorithms of not trading at a loss.

The results of training the model using this reward system was as follows:

3.2.8 Analysis of Optimised Reward System

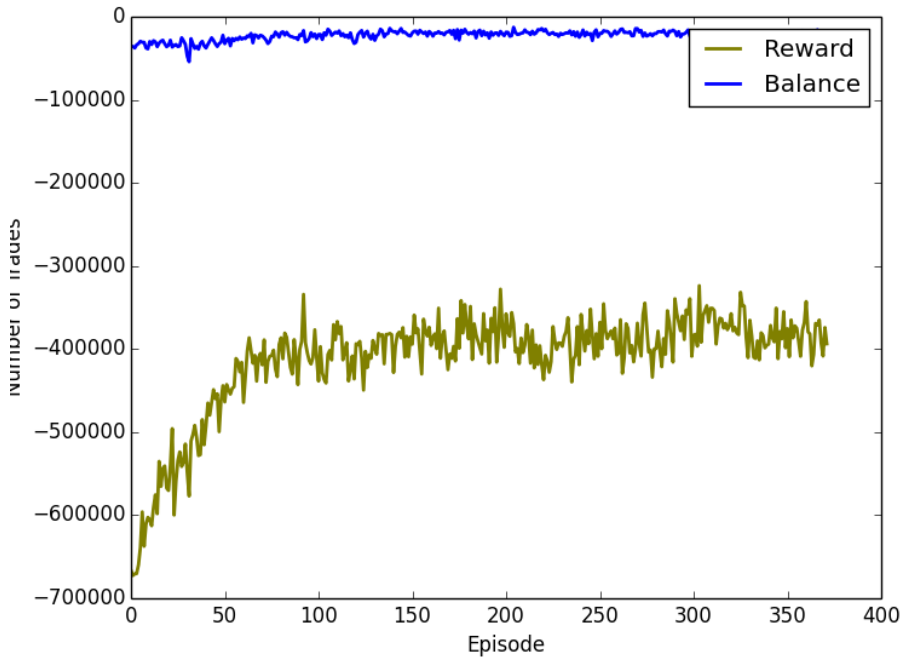

Figure 3.6: Total reward and balance over 373 iterations of the PG Agent with an optimised reward system

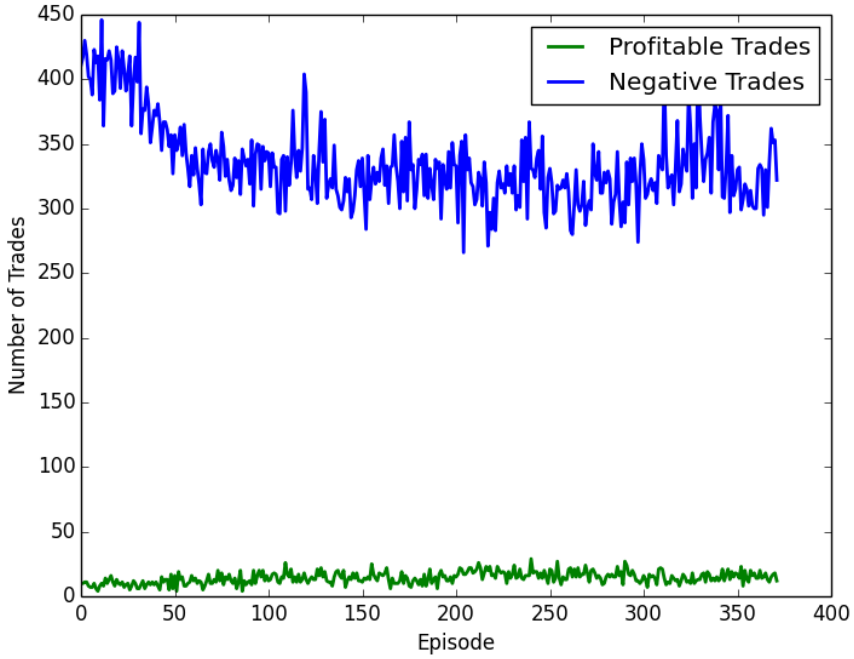

Figure 3.7: A plot of the number of profitable trades and trades that incur a loss for PG-Optimised

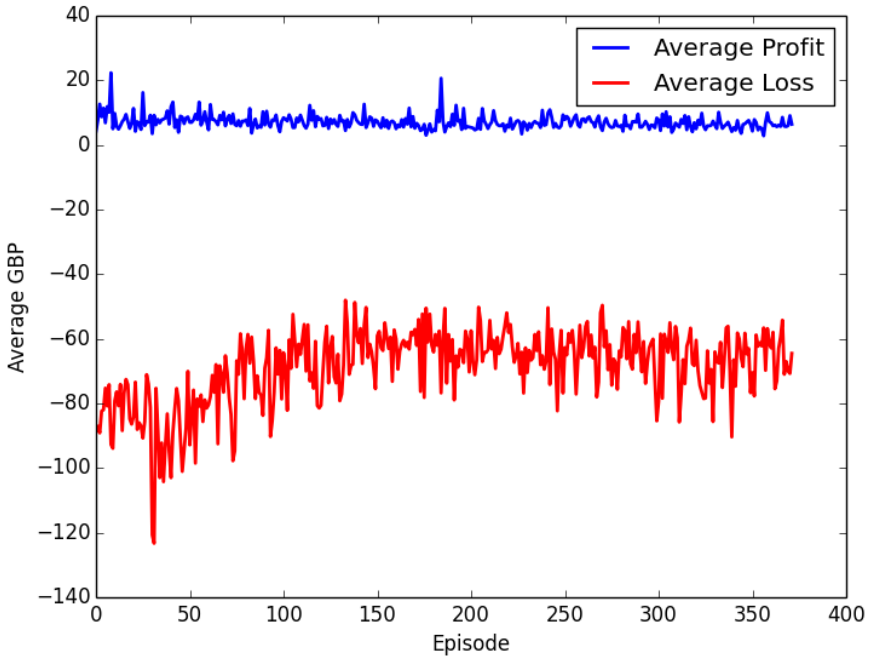

Figure 3.8: A plot of the average profit and average loss of the PG-Optimised trader's trades over 373 iterations

As seen in Figure 3.6, the change in the structure of the reward system did not yield positive results with respect to the balance of the trader at the end of each trading session. Though, there are interesting key differences in these sets of results.

The most noticeable difference between PG-Vanilla and PG-Optimised is that the latter took more

iterations of BSG market sessions to reach a convergence of performance. PG-Vanilla took roughly *50* iterations to reach its terminal performance, whilst PG-Optimised took approximately 300 iterations. This is the clearest impact of the optimised reward system as there were more rewards for the the trader to feed from.

Figure 3.6 indicates a correlation between the improvement in the trader's balance and the reward acquired by the trader over the training period. The trough of the average balance is present in the first 50 iterations at -£33971, and a peak average of -£21987 of the final 50 iterations. Again, this highlights the impact of the reward system that was devised because it allowed the trader to learn how to trader better.

In comparison to PG-Vanilla where the trader ceased to engage in trades, the terminal behaviour of PG-Optimised is that the trader was improving the quality of its bad trades and increased the frequency of its profitable trades. Figures 3.7 and 3.8 provide a clear insight for this behaviour. The overall increase in balance is driven by the trader's ability to decrease the losses incurred from negative trades as well as decreasing the frequency of negative trades that the trader. This is seen as the average loss for a negative trade was approximately -£105 in episode 42 which increased to an average loss of -£50 in episode 265.

In comparison, the average profit from profitable trades did not seem to increase during training whilst the *frequency* of the profitable trades does, however this was rather insignificant in comparison to the decrease in negative trades. The rising frequency of profitable trades is a very promising result as it is the opposite behaviour demonstrated by the PG-Vanilla trader as the frequency of the profitable trades decreased over time during its training process. This indicated that there was an underlying positive learning process that had taken place with regards to spotting trading opportunities for the PG-Optimised trader, which provided empirical evidence that the reward system had improved the trader - despite not being reflected in a profitable trader's balance.

Furthermore, the rate of increase of profitable trades is heavily outweighed by the rate of decrease in negative trades. This comes as a surprise, as the optimised reward system defined in Section 3.2.7 attempted to weight the rewards received from a profitable trade equally to the reward received from a negative trade - taking into account the average profit/loss - but this was not effective in the training process.

To further examine potential reasons behind the behaviour analysed, the experiment's were re-run under observation.

The observation revealed some interesting patterns of behaviour from the trader. Considering that the reward system incorporating a punishment for a lack of trading, the trader had learnt to 'game' the system by avoiding this punishment by frequently submitting quotes at midprices that had very little probability of being matched by other traders in the exchange. It also became apparent that the weakness of the strategy was that the order prices submitted by the trader were constrained to be exactly the midprice. Since the midprice is not a theoretical available trading price, significant amount of time steps passed without the trader being able to engage in a trade in favourable market conditions, since these orders did not cross the spread - and whenever the orders were matched, the difference in time horizon meant that the market was consistently out of favour to the traders actions. Additionally, because BSG consists of a relatively small amount of traders (60 in the current experiments), the LOB did not always have a large amount of depth in its bid/ask books. During the observation of the trading experiments, it became noticeable that the midprice consistently dropped to the market minimum which had rippling consequences on the learnt behaviour of the trader. The midprice consistently dropped to the market minimum due to the LOB's bid/ask books lack of depth. This meant that beyond the L1 bid/ask, the next best orders, were the extremes in the market, so when the L1 bid/ask were lifted, the midprice dropped drastically, which meant that the trader consistently bought the commodity at the market minimum - resulting in a huge loss. Largely down to the lack of depth within the bid/ask books, it meant that the midprice strategy was unsustainable.

Considering that this behaviour was common within a trading session, it became a plausible assumption that constraining the trader to submit orders at the midprice may not be the optimal trading strategy. This meant that the trader should have the freedom to regress to a price, therefore requiring a continuous

action space. This is precisely what was explored.

3.2.9 Continuous Actions

Section 3.2.3 describes the strategy of limiting the trader to a discrete action space of BUY/SELL/NULL. The purpose of the strategy was to simplify the learning task for the trader. However, after thorough analysis in the previous subsection, it was concluded that the midprice-based strategy indicated that it may not have been the most effective method to engage in profitable trades.

This presented a more radical next iteration in the project, as it was a re-approach for the design of a solution for the DRL trader problem. As mentioned in Section 2.4.7, this context allowed for a continuous action based solution given that traders in BSG trade at prices between a market maximum and a market minimum, which are *1000* and *1* respectively.

The benefit of a solution of this kind could be that the trader gets complete freedom to decide on a price that it trades a commodity at, which could potentially solve the problems of being restricted to trading at the midprice - which the market might not favour. However, this freedom may also pose as a problem as it could significantly increase the training time of the agent as there are more actions to consider for the state space.

To implement a solution that could deal with a continuous action space, a *Deep Deterministic Policy Gradient* based agent was implemented.

3.2.10 The DDPG Agent

The DDPG agent implemented follows the theory outlined in Section 2.4.6 which combines TD learning and the Actor-Critic paradigm to learn the policy via PG based methods.

There are crucial details with regards to the implementation of this technique in practice and in this context, which are as follows:

Multiple Continuous Actions

As specified in the problem definition of the prop trader, the trader has the ability to buy and sell a commodity within a given market session, in comparison to the implementations of the traditional sales trader algorithms outlined in Section 2.2, which have to be configured as buyers or sellers at the start of a trading experiment.

Given the decision to opt for a continuous action based solution to dictate the prices at which the trader trades at, this means that there are two actions to be taken on the traders behalf: the previously designed BUY/SELL/NONE actions, and the price to submit an order for.

To incorporate this into the design of the implementation. The actor model comprises of a neural network with two outputs, which crucially are subject to two separate activation functions.

```
def forward(self, state):
        x = self.fc1(state)
        x = self.bn1(x)
        x = F.relu(x)
        x = self.fc2(x)
        x = self.bn2(x)
        x = F.relu(x)
        x = self.mu(x)
        action = x[0]
        price = x[1]
        action = torch.tanh(action)
        price = torch.sigmoid(price)
    return action, price
```

Listing 3.3: Forward pass for the Actor Model

The two separate activation functions were crucial as they are chosen to appropriately distinguish the actions that can be taken by the trader. The BUY/SELL/NULL, is achieved with the $action = tanh(x)$ function which are then rounded to the integers: -1, 0, 1. These are then mapped to the actions specified in 3.2.3, The price at which the trader submits an order at is achieved with the $price = 1000 * sigmoid(x)$ function which restricts the output to be between *1000* and *1* which align with the maximum and minimum price of BSG.

Reward Function

The reward function used for this method was exactly the same as the reward system outlined in Section 3.2.7, however instead of utilising the midprice to reward the trader, the reward is based on the price quoted by the trader.

Similarly to the PG Vanilla Agent, this model was trained via stochastic gradient *ascent* SGA, however with *TDL* which learns by updating its prediction at every time step throughout a trading session.

3.2.11 Analysis and Evaluation of DDPG Agent

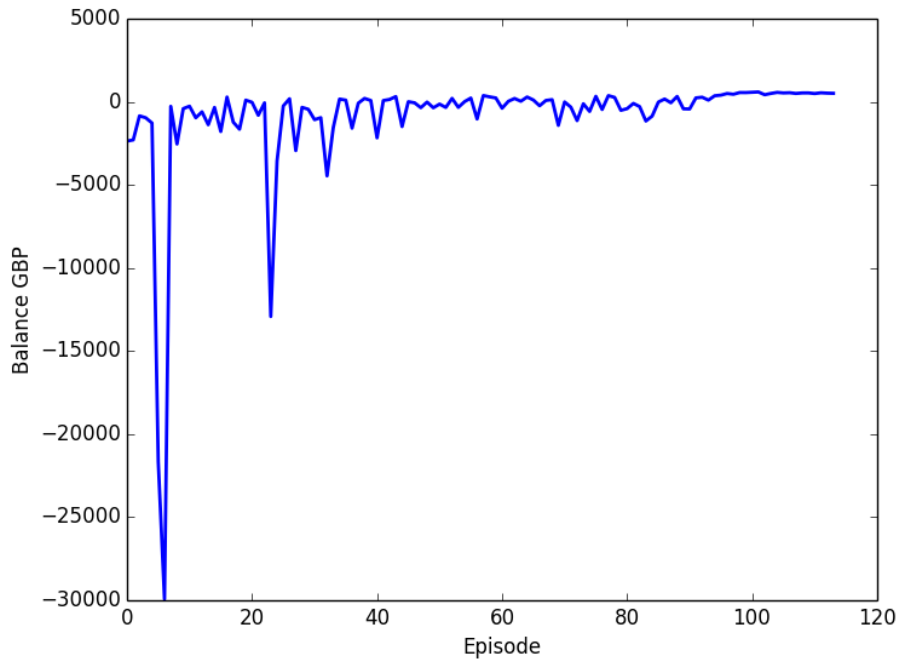

Figure 3.9: Total balance over 114 iterations of the DDPG trader with an optimised reward system

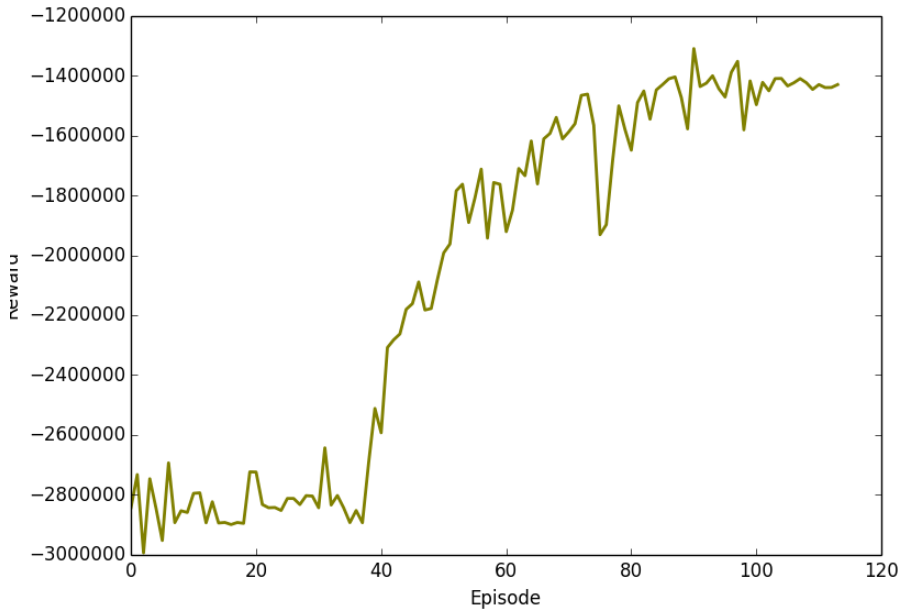

Figure 3.10: Total reward over 114 iterations of the DDPG trader with an optimised reward system

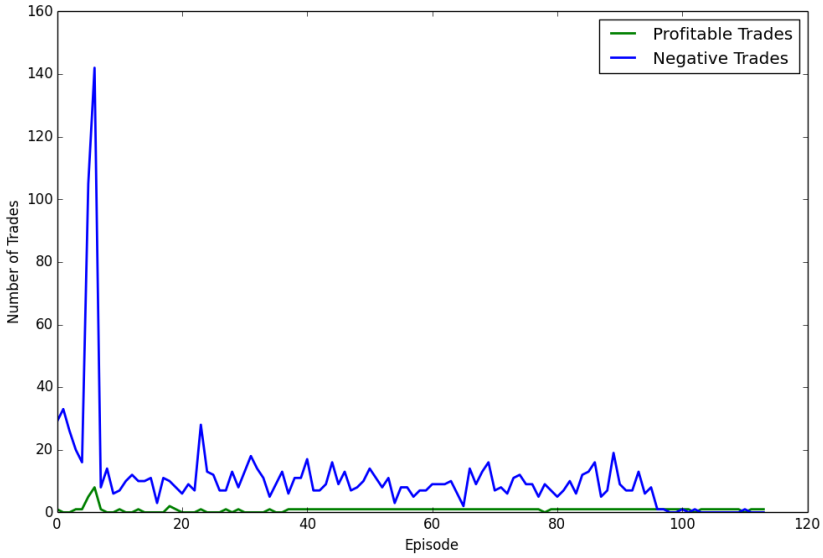

Figure 3.11: A plot of the profitable vs negative trades for the DDPG trader over 114 iterations

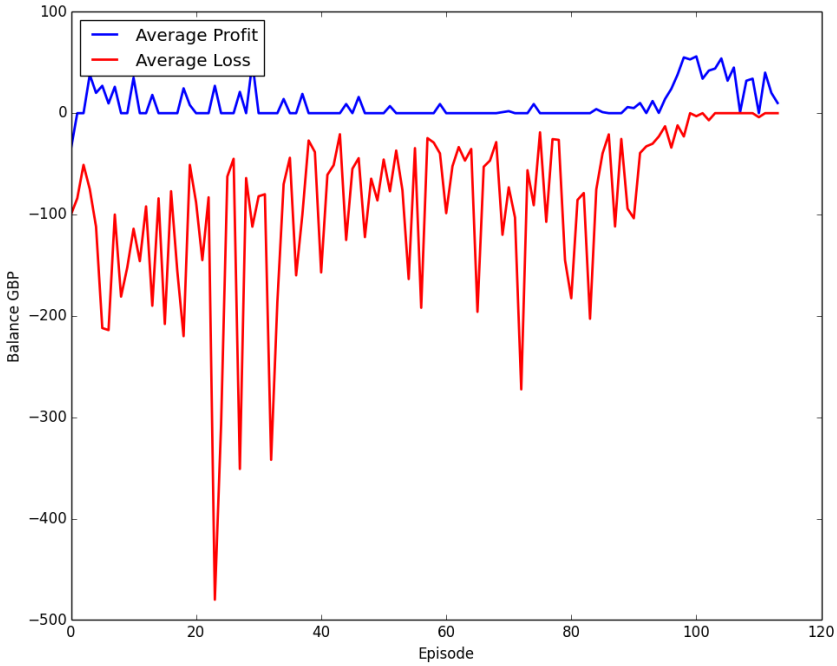

Figure 3.12: A plot of the average profit and average losses incurred from the DDPG trader's trades over 114 iterations

As seen in the tail end of Figure 3.13, the trader seemed to learn the constraints that were hard coded into ZI-C and ZIP. Despite the trader engaging in essentially minimal trades, beyond episode 95, 80% of these trades were profitable trades. This profitable trading behaviour was also observed during training, as every time the trader began to submit a series of bid orders, it would start from a bid price of 1 and would steadily increase the price to engage in a viable transaction, and vice versa from 1000 when submitting ask prices. This was great news as it meant that the trader was guided by the reward system to learn what a profitable trade looks like. This is also correlated with an increase in reward, albeit it remained a very large negative value, which further reinforces the versatility of the reward system across multiple algorithms.

The problem again lies with the fact that the trader took the word 'a' in "learn what a profitable trade looks like" too literally, as it consistently engaged in one or two trades in the market sessions. This meant that the trader was not able to exceed the performance of the ZIP traders which dwarfed the DDPG trader's final balance and consistently yielded profit from a large quantity of trades.

Figure 3.11 clearly indicates that the DDPG trader struggled to even make more than a single profitable trade consistently in an entire market session from the outset of the training process. This meant that the trader had very minimal positive rewards stored in its replay buffer from which it samples to learns from. Also, in comparison to the PG-Optimised trader, the DPPG trader trades substantially less than any of the PG traders. This comes as no surprise as the trader explores the continuous action space with added noise, which leads to majority of the orders that were submitted to have a very low probability of being matched in the market, as they are unlikely to have crossed the spread.

Similarly to the PG-Optimised trader, Figure 3.12 shows the average loss per trade decreasing over the course of training, and the average profit per trade remaining relatively flat and unimproved. Considering that DDPG is an off-policy technique and relies heavily on learning from the replay buffer of mini-batches of stored transitions between state-action pairs, the assumption that the 'experience' that the trader learns from, did not include many positive rewards, seemed a reasonable point of examination to improve on this model.

After conducting analysis on the batches of samples from the replay buffer, it was confirmed that there

was a real sparsity of positive rewards in the replay buffer. Of the batches of 64 arrays consisting of a 4-tuple: $\{\sigma_t, \alpha_t, \rho, \sigma_{t+1}\}$, 98% of the rewards within the arrays consist of a reward of `-10` which is from the punishment of not submitting any order as defined in Section 3.2.7. This equated to $\approx 58k$ time steps out of a possible 60k time steps resulting in a `-10` reward, which is not ideal. This analysis laid grounds for a strong hypothesis that the absence of positive rewards in the replay buffer could have resulted in the trader seemingly not being capable of recognising multiple trading opportunities in a given market session.

The most apparent way to tackle this issue would be to design a more detailed Reward system with an increased amount of positive rewards to guide the trader. However, this would have required too much training time to define new reward rules and optimise the magnitudes of rewards for varying scenarios. The solution that was opted for is outlined in the next subsection.

3.2.12 Sparsity in rewards

A commonly known limitation of traditional Model-Free RL algorithms is that they are sample inefficient [36]. This means that they require a significant amount of samples of *varying experiences* to learn something useful. Due to the analysis in the previous subsection, this potentially was the case in this context, so optimisations the following optimisations were made to attempt to tackle this issue.

According to a Medium blog by Blagojevic [2], sampling randomly from an experience replay buffer is an effective means to get a variety of experiences. However, the blog outlines that this may not be an effective method in many games that RL solutions are applied to, as many games have sparse rewards which result in 99% of the experience results in a 0 reward. This was also analysed to be true for this 'game' with the reward system implemented. To tackle this, Blagojevic suggested that sampling experiences which contain greater a magnitude of reward more favourably in comparison to the experiences that have less reward, would allow a RL agent extract useful information from its experiences with increased efficiency.

```
def sample_buffer(self,batch_size):
        max_mem = min(self.mem_cntr, self.mem_size)
        p = np.array([self.unusual_sample_factor ** i for i in range(max_mem)])
        p = p / sum(p)
        batch = np.random.choice(max_mem, batch_size, p = p)
        states = self.state_memory[batch]
        new_states = self.new_state_memory[batch]
        actions = self.action_memory[batch]
        rewards = self.reward_memory[batch]
        terminal_memory = self.terminal_memory[batch]
        return states,actions,rewards,new_states,terminal_memory

def sort_buffer(self):
        self.state_memory,
            self.action_memory,self.reward_memory,self.new_state_memory,self.terminal_memory = \
           self.sort_lists_by([self.state_memory,
               self.action_memory,self.reward_memory,self.new_state_memory,self.terminal_memory],
                           key_list = 2)

        self.state_memory = np.array(self.state_memory)
        self.action_memory = np.array(self.action_memory)
        self.reward_memory = np.array(self.reward_memory)
        self.new_state_memory = np.array(self.new_state_memory)
        self.terminal_memory = np.array(self.terminal_memory)

def sort_lists_by(self,lists, key_list=0, desc=False):
        return zip(*sorted(zip(*lists), reverse=desc,
                key=lambda x: abs(x[key_list])))
```

Listing 3.4: Sampling experiences with a probability factor

The code snippet 3.4 applied the theory outlined by Blagojevic. It works by first sorting the lists that store the state-action transitions into descending order w.r.t to the reward acquired in the experience. A

probability is then attached to each 4-tuple which according to an 'unusual sample factor' which is set to 0.85^i where i is the position of the tuple in the list of stored experiences. The effect of this is significant as the exponential decrease in probability attached to each experience in descending reward, becomes the probability at which the trader samples the experience to backpropagate over.

This would mean that rather than sampling all experiences with a uniform random distribution, experiences where the trader made a profit (or a significant loss) would be at the forefront of the sorted list and therefore would be selected with greater probability to be sampled. Since these experiences were sparse, the *unusual sample factor* was chosen to be 0.85 to provide a steep decline in probability.

In an ideal case, this tweak would allow the trader to learn from more meaningful experiences that eventually lead it to learn how to trade more optimally.

Analysis

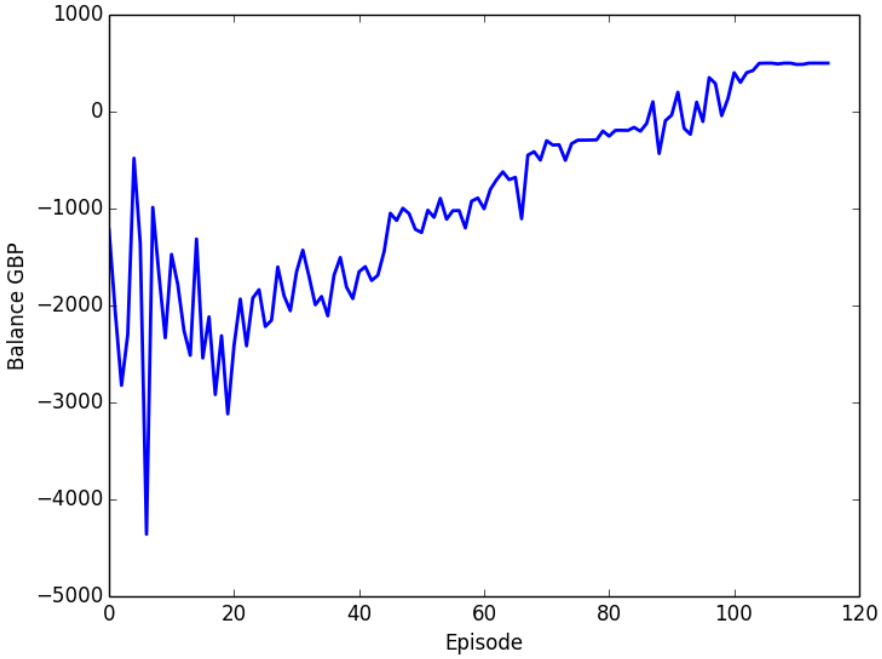

Figure 3.13: Total balance over 116 iterations of the DDPG-Sparsity trader with an optimised reward system

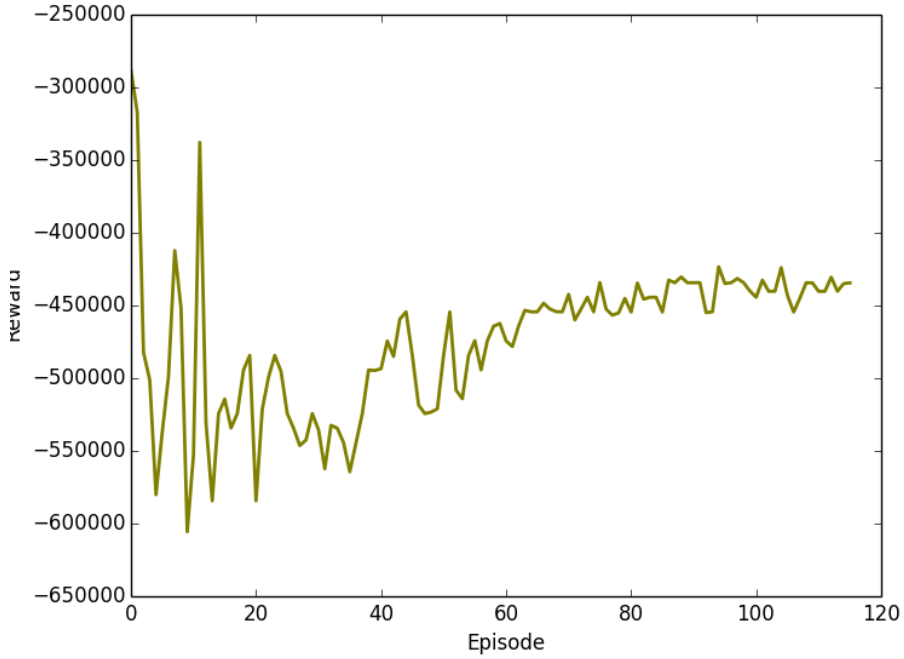

Figure 3.14: Total reward over 116 iterations of the DDPG-Sparsity trader with an optimised reward system

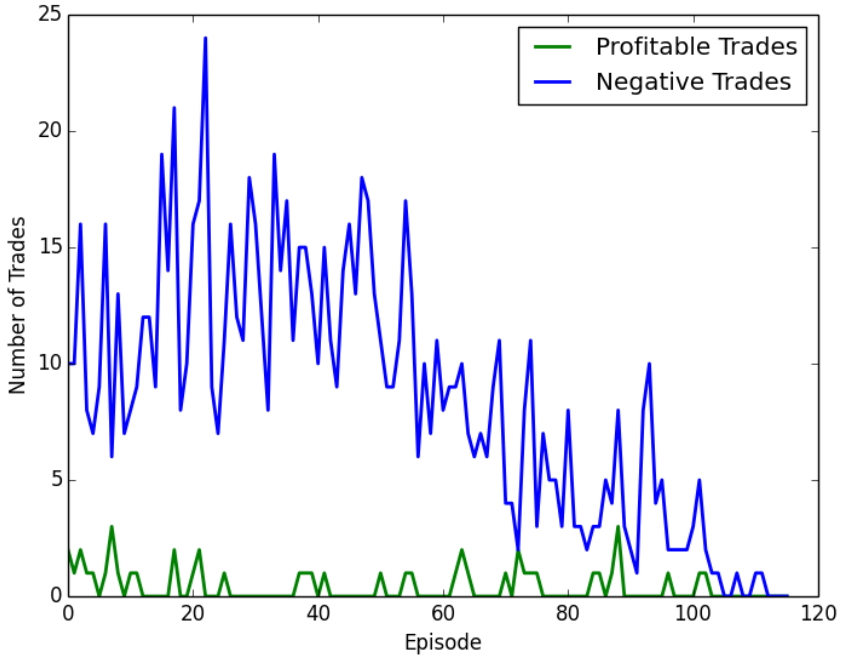

Figure 3.15: A plot of the profitable vs negative trades for the DDPG-Sparsity trader over 116 iterations

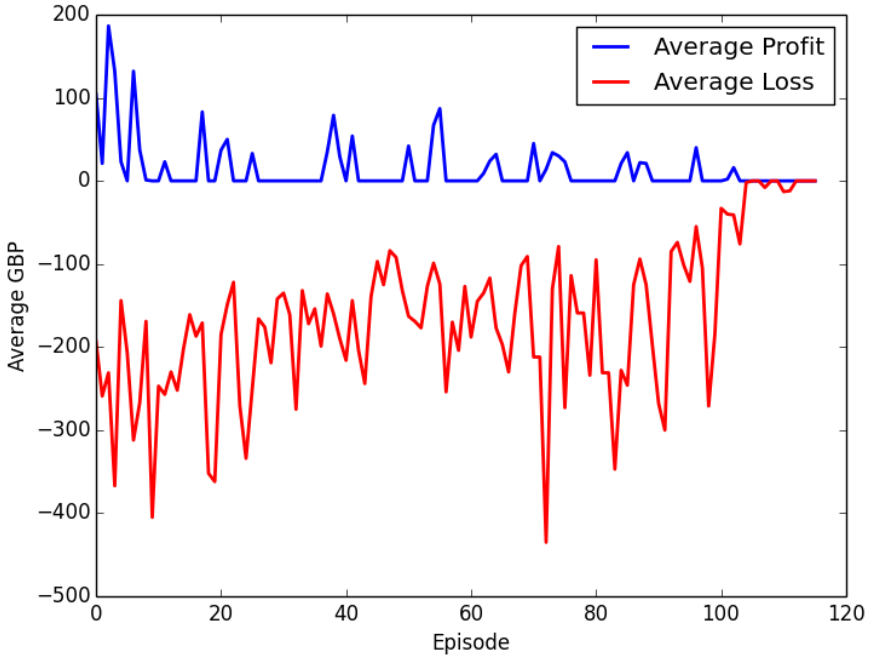

Figure 3.16: A plot of the average profit and average losses incurred from the DDPG-Sparsity trader's trades over 116 iterations

Disappointingly, the promising results produced by the previously built DDPG trader were not built upon by the *DDPG-Sparsity* trading agent. Similarly to the DDPG trader, the DDPG-Sparsity trader consistently failed to make a relatively large volume profitable trades to learn from, and the number of negative trades was similar to the DDPG trader, except with much more variance from one episode to the other. Crucially, the end result of the training process is similar to that of the PG-Vanilla trader, which was that the trader eventually ceased to trade.

The results were much more difficult to reason about, as the analysed weakness in the DDPG trader pointed to a solution that the DDPG-Sparsity ideally could deal with. After much examination, it was hypothesised that the implementation of the *unusual sample factor* on one hand dealt with the inefficient sampling of the replay buffer in the DDPG trader, however, counteracted the impact of the punishment for a lack of trading. As explained in Section 3.2.7, a punishment was added for a lack of trading by the trader. Interestingly, the magnitude of this punishment regularly happened to be smaller than the magnitude largest rewards/punishments acquired by the trader. Therefore, the experiences that were sampled were mostly of negative trading experiences, and not the punishment acquired by the trader for not submitting orders to the exchange.

The overall effect of this is that the trader does not learn from the punishments for a lack of trading, and therefore, follows the same trajectory of behaviour demonstrated by PG-Vanilla to cease all trading activity to minimise the loss acquired. This hypothesis is also supported by the fact that the ± 40 reward acquired by the trader for well intentioned bid/ask orders were also lower in magnitude than the rewards/punishments of the positive and negative trades. So therefore, the sample experiences did not draw from these experiences either.

It is important to note that this algorithm is hugely time consuming. As an example, the 116 market sessions took roughly 192 hours (equates to 99 minutes per market session) to run on a 2.3 GHz Core i5 Macbook Pro. The time complexity is greatly increased due to the introduction of the `sort_buffer()` function introduced in 3.4. Due to the nature of a TD learning algorithm, the trader updates the network parameters after every time step, and therefore samples from the replay buffer at every time step. Before the sampling takes place, the list of experiences are sorted in descending order w.r.t to the reward. This is the time consuming step as roughly 60,000 items in each list are being sorted per time step. To somewhat speed this process up, the sampling process was designed to take place every two time steps.

Naturally, the first point of exploration would be to tune the magnitudes in the reward system to best utilise the replay buffer, however, this form of experimentation could not be afforded due to the time constraints of the project.

Chapter 4

Conclusion

The focus of the chapter is not to serve as an in-depth analysis and evaluation of results - as this was done extensively in the previous chapter - but it is to summarise the high level challenges, and insights acquired whilst exploring solutions to this problem domain with DRL.

4.1 Overview

The project was ambitious in its proposal as well as its execution, and posed an excellent challenge that was entirely suited to a Master's project. Given the relatively little contribution - within academia- to this field as explained in Section 1.1, it was well poised for an interesting mesh between a brown-field and green-field project.

The project is inter-disciplinary, as it is a marriage of Computer Science and Economics/Finance to produce a very interesting field of algorithmic trading. As described in Sections 1.3.1 and 1.3.2 there has been outstanding progress within the respective fields, in particular Google's AlphaGo Zero had demonstrated to learn strategies that were not even known to humans, and this was an excellent selling point and a motivator behind applying RL within the field of algorithmic trading.

The inter-disciplinary nature of the project meant learning a large volume of teaching material within a very limited time period. Given a very limited knowledge of reinforcement learning coupled with a basic awareness of algorithmic trading, a significant amount of time during the project was used in building the foundational knowledge and then learning the state-of-the-art algorithms within the field - which was a huge challenge in its own right.

As mentioned in Section 1.3.3, Furlan-Falcao had luckily built a framework for RL algorithms to interact with BSE in a refactored version called BSG, which was a crucial solution to an initial hurdle. He also demonstrated promising results by combining heuristic solutions with reinforcement learning algorithms to produce ZIP-RL as ZIP-RL was shown to outperform humans by outperforming ZIP in BSE's market simulation. This laid the crucial groundwork and motivation to attempt to extend this work by producing an algorithmic trader that was solely based on RL algorithms and by experimenting with a variety of RL techniques, the project aimed to achieve this.

4.2 Project Status

This project was executed under a strenuous time with the COVID-19 pandemic bringing difficult times with this project being no exception.

Ultimately, the results analysed within this project were not a success story, as the there were numerous challenges faced in practically implementing an RL based solution to tackle the prop trader problem. In spite of this, there were some interesting results and behaviours that were analysed and explained in Section 3.2. It is important to emphasise that the conclusions drawn in the analysis of each result were not necessarily factual, but rather educated assumptions based on extensive analysis of the results.

The *PG Vanilla* trader provided the least promising results as it became an inactive trader in a market session where ZIP and ZI-C traders by far greater results in comparison. PG Vanilla highlighted the importance of a robust and complete reward system that would guide the trader during the training process. It had learnt not to trade at all in order to avoid consistently trading at a loss thus avoiding negative rewards. There was an absence of what the necessary conditions of a good trade look like, which lead to the development of the PG-Optimised trader.

PG-Optimised was named due to the focus on optimising the reward function for the trader. The aim of the new reward system was to provide the trader with more feedback w.r.t to the actions chosen after a state observation. Analysis indicated that the vanilla reward system was far too simplistic, thus the optimised reward function was designed to attempt to prevent the lack of trading activity demonstrated by the *PG Vanilla* trader. In addition, some incentives were added to reward the trader if the trader's intentions exemplified desirable behaviours, which was equally matched with punishments for incorrect behaviours. Disappointingly, this did not create a more lucrative trader. However, there were some interesting realisations and details revealed in the analysis, which suggested that the trader was positively learning from the new reward system, and that the discrete trading strategy may not have been optimal for this context.

The results from the PG-optimised motivated a radical change in approach which was application of a different DRL algorithm called DDPG. A shift was made from the discrete action strategy, to an implementation that allowed the trader to regress to a trading price for itself, which became the continuous action strategy and was called the *DDPG Trader*. This implementation provided some very promising results as the results indicated that for the first time, the trader would complete a market session in profitable position. However, the lingering result remained that the frequency in the amount of trades was still a persistent problem, averaging only a couple of trades per session. It was nonetheless an achievement to build a profitable trader. Though, its performance still did not compare to traditional established algorithms.

The DDPG-Sparsity trader was developed with the analysis of the DDPG Trader indicating that the training of the trader was stunted due to the inefficient sampling of the replay buffer. To tackle this, DDPG-Optimised was developed with a tweak in the implementation which would theoretically increase the sampling efficiency to optimise the training process. The results for this trader were similar to that of PG-Vanilla, where the trader would cease to trade. This came as a surprise and was very challenging to analyse and reason about to explain the behaviour of the agent. This algorithm had by far the most time complexity due to the sorting of the replay buffer, which is a major challenge to deal with when designing an algorithm of this ilk.

Overall, the project details the journey of creating a DRL trader to attempt the answer the hypothesis outlined in the executive summary. Unfortunately, the results were not successful in answering the hypothesis, however the project aims to outline the vast number of challenges faced along the way and concrete implementations to tackle the challenges - which may serve as a valuable contribution to those who wish to pursue this field of work.

4.3 Future Work

As mentioned earlier in this chapter, this project was heavily affected by the impact of the pandemic, which in turn affected the avenues of exploration that could have been implemented in the execution phase of the project.

A huge challenge with applying DRL based training methods, is that the training time to reach convergence takes a significant amount of time, which is a challenge in its own right for Master's level project, due to the time constraints. This directly impacts the ability to explore techniques thoroughly, as a trade-off decision must be made in deciding how much time is spent on trying to make an algorithm work.

Additionally, due to nature of the results, it is inconclusive as to whether a DRL based approach can outperform humans at the task of prop trading. Therefore, there are multiple avenues that could be further explored to test if they are fit for this problem domain.
The subsections below will aim to outline some of the unexplored options that could have potentially

provided more desirable results.

4.3.1 Exploration of State Space Design

The design of the state space is a crucial factor in the success of DRL models. Extensive research was conducted to figure out an effective design to extract crucial information from the lob that would constitute as a 'good' and informative observation for the trader. Hence, an AE was used to compress a large amount of data that would inform the trader about the recent history of the LOB. In hindsight, variations of this state-space should have been explored, such as smaller dimensional inputs with relevant information about the position of the trader and information of the LOB - without AE compression. The experiments conducted could be replicated with various state-spaces to find the optimal state-space.

4.3.2 Regression Models

In the design process of the trading strategies of the implementations detailed in Chapter 3, the reward system was designed to incentivise the trader to capitalise on good trading opportunities depending on the state. In order to make the prediction of a good trading opportunity more accurate, regression models could be used to predict the future prices of the midprice, such as a Long-Short Term Memory (LSTM) recurrent NN (RNN). This could be utilised to inform the trader as to whether the price would increase or decrease. This regression could have been used as an input for the DRL model, to allow the trader to make better trading decisions.

4.3.3 Increase explorative power

As mentioned previously, the training time for DRL methods are significant. In hindsight, any work carried out to explore the application of DRL algorithms would prioritise trying to decrease this time taken by using more efficient techniques to explore the state-action space for DRL traders. *AC3* is a powerful asynchronous method to spawn multiple training agents and multiple environments to speed up the process of learning using multi-processing. The multiple agents can combine their learning after exploring the state-action space within their own environment. This would be an excellent tool to use as it would have given a quicker indication as to whether or not a technique seems to be effective or not.

4.4 Last Words

Despite this project not yielding a successful outcome to answer the hypothesis, the project presents a detailed overview of the challenges presented during the entire process of applying DRL techniques to this problem domain. Crucially, the project is not conclusively claiming that DRL cannot yield successful results in this field, but aims to serve as a valuable contribution to those who attempt similar projects, with the hopes that the lessons learnt from this project enable curious individuals to reach further heights.

DRL is an exciting field and it was a pleasure to contribute to the development of this ever expanding area of research.

Bibliography

[1] Mohammed Ashraf. Reinforcement learning demystified: Markov decision processes (part 1), 2018.

[2] Branko Blagojevic. Reinforcement learning with sparse rewards, 2017. A Medium research blog. Accessed August 2020.

[3] Bonsai. Deep reinforcement learning models: Tips & tricks for writing reward functions, 2017. A Medium research blog. Accessed August 2020.

[4] Katharina Buchholz. Computers manage more stock than humans do, 2019.

[5] Dave Cliff. Bristol stock exchange. Available at https://github.com/davecliff/BristolStockExchange. Accessed May 2020.

[6] Dave Cliff. More than zero intelligence needed for continuous double-auction trading.

[7] Dave Cliff. Economic agents and market-based systems ii, 2019.

[8] Dave Cliff and Janet Bruten. Minimal-intelligence agents for bargaining behaviors in market-based environments, 1997.

[9] Marko Cotra. Deep learning basics:the score function and cross entropy, 2017. A research blog. Accessed August 2020.

[10] Rajarshi Das, James E Hanson, Jeffrey O Kephart, and Gerald J Tesauro. Agent-human interactions in the continuous double auction, 2001.

[11] Avinash K. Dixit. Optimization in economic theory: Second edtion, 1990. Pages 163-166. Accessed August 2020.

[12] Alvaro Furlan-Falcao. Bristol stock gym. Available at https://github.com/ElectronAlchemist/Bristol-Stock-Gym/. Accessed July 2020.

[13] Alvaro Furlan-Falcao. Using reinforcement learning to train adaptive traders in a limit-order-book financial market, 2019.

[14] Caroline Gardiner. Blue crystal phase 4. Bristol Advanced Computing Research. Accessed August 2020.

[15] Steven Gjerstad and John Dickhaut. Price formation in double auctions, 1998.

[16] Jeremy Jordan. Planning in a stochastic environment, 2017.

[17] Donald E. Kirk. Optimal control theory: An introduction. Pages 55-61. Accessed August 2020.

[18] Gavin Koh. Why is trading so hard?, 2017.

[19] Michael J. Kramer. Limit order. Limit Order Investopedia Definition. Accessed August 2020.

[20] Sergey Levine, Chelsea Finn, Trevor Darrell, and Pieter Abbeel. End-to-end training of deep visuomotor policies, 2015.

[21] Daniel Liberto. Competitive equilibrium. Competitive Equilibrium Investopedia Definition. Accessed August 2020.

[22] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, David Silver Yuval Tassa, and Daan Wierstra. Continuous control with deep reinforcement learning. Research paper. Accessed August 2020.

[23] Shubha Manikarnike. Temporal difference methods in rl, 2020. A research blog discussing TD methods. Accessed August 2020.

[24] Jon Michaux. Off-policy actor-critic algorithms. A github post. Available at https://jmichaux.github.io/week4b/. Accessed August 2020.

[25] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning, 2013.

[26] Błażej Osiński and Konrad Budek. What is reinforcement learning? the complete guide, 2018.

[27] Gregory Piatetsky. Exclusive: Interview with rich sutton, the father of reinforcement learning.

[28] Terry Sejnowski. The deep learning revolution, 2018.

[29] David Silver. Value function approximation. A lecture series by David Silver at UCL. Accessed August 2020.

[30] David Silver. David silver ucl course on rl, 2015.

[31] David Silver. Markov decision process, 2020. A lecture series by David Silver at UCL. Accessed August 2020.

[32] David Silver and Demis Hassabis. Alphago zero: Starting from scratch, 2017.

[33] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic policy gradient algorithms. A paper produced in association with DeepMind. Accessed August 2020.

[34] Justin A. Sirignano. Deep learning for limit order books, 2018. A paper produced in the University of Illinois at Urbana-Champaign. Accessed at 2020.

[35] Vernon L. Smith. An experimental study of competitive market behavior.

[36] Stable-baselines. Reinforcement learning tips and tricks. Documentation for stable-baselines. Accessed August 2020.

[37] Money Summit. How the four biggest us banks generate income and revenue, 2020.

[38] Shyam Sunder and Dan Gode. Allocative efficiency of markets with zero-intelligence traders: Market as a partial substitute for individual rationality, 1993.

[39] Perukrishnen Vytelingum. The structure and behaviour of the continuous double auction, 2006.