Dissertation Type: enterprise

University of BRISTOL

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.

Friday 21st August, 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, Friday 21st August, 2020

Contents

List of Figures

List of Tables

Executive Summary

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

DRL requires for 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 combinations of inputs to represent the current state of the 'game', from which the trader can learn which actions to take 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 Advantage Actor-Critic, Policy Gradient and Deep Deterministic Policy Gradient (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 [?] and Furlan-Falcao's Bristol Stock Gym [?] 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 challenging task as the mapping between the current state of the system and the action the agent takes in such an environment is a problem all financial institutions are aiming to tackle in the most sophisticated ways to gain an overall competitive advantage. The results of this thesis are not a success story, but it details the journey of iterating and analysing results to create a profitable trader using the techniques outlined above and the vast challenges that were presented.

> My 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 the hypothesis

- I spent 100+ hours researching prior work on the field and learning about Deep Reinforcement Learning techniques.
- I wrote a total of ? lines of source code, comprising of the merger of capabilities of BSE into Furlan-Falcao's BristolStockGym, several DRL models, Variational Auto-Encoder and Regression Models.
- Spent 100+ hours training the various models and running experiments on Blue Crystal 4 [?]

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
	$\vdots$	
$\mathcal{H}(x)$	:	the Hamming weight of x
$\mathbb{F}_q$	:	a finite field with q elements
x_i	:	the i-th bit of some binary sequence x, st. $x_i \in \{0,1\}$

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 set by 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 a lower one, while for sell limit orders, the order will be executed only at the limit price or a higher one [**?**]. A collection of orders of this kind formulates the Limit-Order-Book (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 own a stock of their own, but instead buys and sells to secure the best price for their client. A proprietary trader (a "prop" trader) on the other hand buys and sells stocks 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 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 and MGD, a modification of GD by Steven Gjerstad & John Dickhaut) 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.

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 prodcers and utility-maximising consumers in competitive markets arrive at an equilibrium price[?] . 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, where the prices would converge to the theoretical 'best' price of an asset, despite a small number of traders used.

These sets of experiments were extended by Gode and Sunder [?] in the 1990's. Crucially however, the human traders that were the core of Smith's experiments were replaced in Gode & Sunder's work 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 whilst monitoring their allocative efficiency. They concluded that the ZI-U traders were essentially useless in that they did not 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 1996 [?] who hypothesised that the ZI-C traders would fail to equilibrate under a set of market conditions and then empirically demonstrated that they did indeed fail to equilibrate. 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 [?]. 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 [?] ran numerous experiments where they would pit algorithms like ZIP, MGD (a modified GD) against humans as well as 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.

Despite the promising results and the wide press received from the IBM publications, relatively little experimentation has been conducted of this type. Meanwhile, Tesauro and Bredin published an improved version GD called GD-Extended (GDX) in 2002, and Vytelingum published the Adaptive-Aggressiveness Algorithm in 2006 [?]. 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/offer more likely to be accepted [?]. GDX uses real-time dynamic programming to formulate agent bidding strategies in a broad class of auctions[?].

1.2.2 Developments in Artificial Intelligence

Despite the 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. It has been labelled the 'Deep Learning Revolution' by Terry Sejnowski [?].

Similarly, Reinforcement Learning has been an idea coined for a long time[?], however it was applied to very limited and relatively simple problems.

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 [?] 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, who 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 extended 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 in order 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 big financial institutions relatively late adoption of surging technologies, mainly driven by small FinTech companies eating from the established companies' profits with their innovative tech solutions to an otherwise very traditional industry. MX reports [?] that as an example, a breakdown of JP Morgan's revenue indicates that roughly 15% of JP Morgan's interest net income ($9b) stems from their trading of assets and investing securities, which highlights their interest in optimising the entire process of trading.

The IBM experiment described in section **??** 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[?] 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. An objective perspective from an individual that is uninvolved in trading suddenly becomes hard to translate when there is a stake involved whilst trading, because of an animalistic sense of survival that kicks in that 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. This has resulted in 35% of the U.S's equities being managed by automatised processes [?].

On the flip side, despite the removal of human error, Torsten Slok, chief international economist at Deutsche Bank, has named it the number 1 risk to markets in 2019[?]. 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 ?? 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.

AlphaGo Zero surpassed its predecessor AlphaGo in an unprecedented manner because the knowledge of the AI is not bounded by human knowledge[?]. 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 in 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.

Whilst AlphaGo Zero as an example is a 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 [?] also alluded to the excitement of applying DRL techniques to this domain, but after proving unsuccessful, 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) and interface to train DRL 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 which 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 because that is a challenge within itself, to decide 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 hyperparameters and model layer choices. This is a common problem within machine learning, and despite resources being available as well as exemplar material, the choices made w.r.t to these factors have to be bespoke 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, an example is in the OpenAI gym environment 'Lunar Landing' game, in which the agent has to land the spacecraft safely. A well intentioned reward system may include punishing the agent with negative rewards for landing the spacecraft incorrectly, however, 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

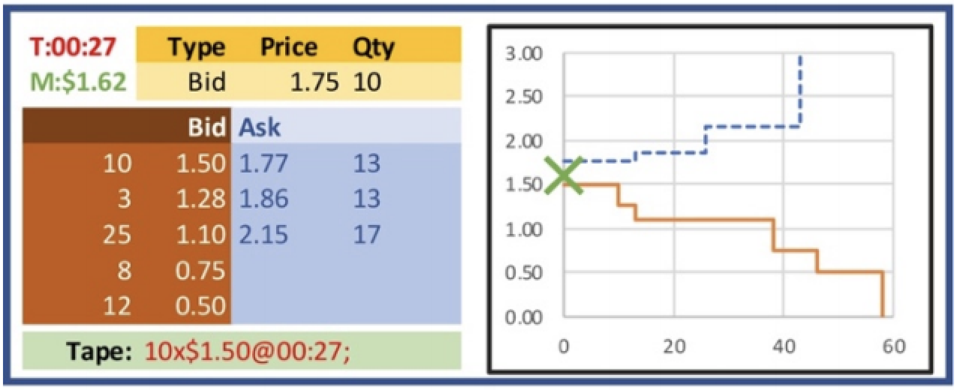
Chapter 2

Technical Background

2.1 The Core Problem

2.1.1 The Limit Order Book

The Limit Order Book (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, ergo the bid book is on one side, and the ask book is on the other.



T:00:27	Type	Price	Qty
M:$1.62	Bid	1.75	10

	Bid	Ask	
10	1.50	1.77	13
3	1.28	1.86	13
25	1.10	2.15	17
8	0.75		
12	0.50		

Tape: 10x$1.50@00:27;

3.00
2.50
2.00
1.50
1.00
0.50
0.00
0
20
40
60

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

The left hand side of Figure 2.1 is a mockup of a graphical representation of BSE, whilst in its original form, is 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 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 **??** 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:

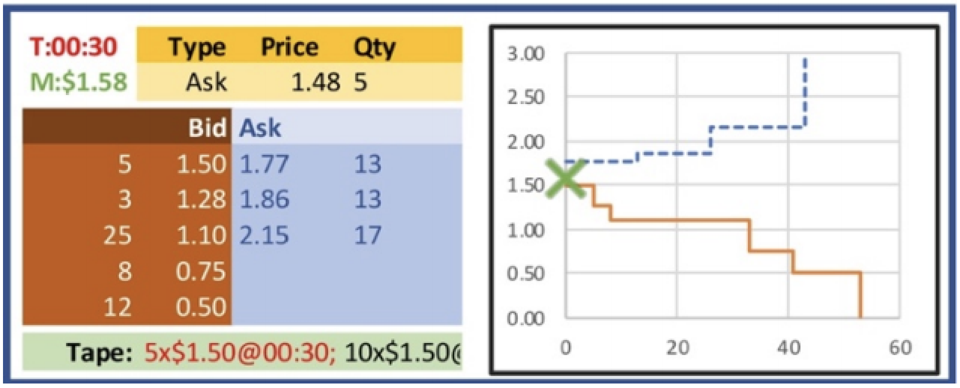
$$\frac{BestBidQty * BestAskPrice + BestAskQty * BestBidPrice}{BestBidQty + BestAskQty} = Microprice \tag{2.1}$$

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

The right hand side of Figure 2.1 represent 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 the quantities that are available 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.



T:00:30	Type	Price	Qty
M:\$1.58	Ask	1.48	5

	Bid	Ask	
5	1.50	1.77	13
3	1.28	1.86	13
25	1.10	2.15	17
8	0.75		
12	0.50		

Tape: 5x\$1.50@00:30; 10x\$1.50(

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

Figure 2.2 indicates that a new order has been placed to the exchange. Interestingly, this order crosses the spread because the ask price \$1.48 was less than the *best bid* in Figure **??** 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 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 can't be compared, but 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.

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 markets where humans did 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, it 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 if the trader is a seller/buyer respectively. 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 gets 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* unliked 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 demonstrated that they behave human-like in these markets.

As mentioned in Subsection 1.2.1 IBM demonstrated that ZIP outperformed their human counterparts in controlled experiments. So this project uses ZIP as the benchmark to explore if a DRL trader can be implemented that will too, outperform its human counterparts.

2.3 Reinforcement Learning

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

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, to maximise a reward [?]. 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. state all the sources for the derivations in the entire chapter

Markov Decision Process

Most Reinforcement Learning requires a fully observable environment in which Markov Decision Processes (MDP) are a 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', which may 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 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 we accumulate 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[**?**] 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.

Alongside this, the Action-Value function, 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, that maximise the utility until the ‘game’ is finished, the value function alone cannot be used, as it only provides the expected return from a specific state, but information regarding which action to take is not provided. 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 impossible in this problem domain. 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 [**?**] allows the Equation 2.5 to be rearranged in such a way that matches the format of a Bellman equation, which then allows the action-value Equation 2.6 to be rearranged as a Bellman Expectation Equation, subject to a policy π: explain what a bellman equation is

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

Which 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, this provides the maximum possible returns out of every possible traversal thereafter. Crucially, if the agent knows $Q_*(\sigma)$, then it has found the best possible traversal, so it has 'solved' the entire problem. So informally, an MDP is solved if $Q_*(\sigma)$ is solved.

Optimal Policy. The optimal policy, is to best possible way for an agent to behave in an MDP. The previous two definitions define *what* the reward is, but this 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 [?] 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 are 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 this 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 its 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 Subsection 2.3.2. This is done via a method called *Policy Iteration*, where unlike the prediction problem which iterates over the Value Function, *Policy Iteration* iterates over the Action-Value function in Equation 2.9. This is done via 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' in which an individual state may not give the agent the maximum expected return, but the states following it yield greater 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 the fact that because the state space in BSG is very complex, and because the Monte Carlo solution requires a 'reset' back to a specific state σ_t to effectively approximate the expected returns, the solution is not a good fit for the problem at hand.

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, rather than learning during the 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, in which $\rho(\sigma_t)$ is the immediate reward and $\gamma V_\pi(\sigma_t)$ is the discounted future reward for the next step and where β is the generalised *learning rate*. We substitute this *estimated* return (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 was driving a car and **almost** crashes, but then drives off safely: the MC method would not update the value function in a negative sense because the end goal was not negative, whilst 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 a sudden drop or rise in price of a stock, which the trader could have bided or asked for, which may have led to a loss. Whilst in MC the trader would have had to commit the to a bad trade to have learnt from the experience.

To solve the control problem for TD Learning, there are various options that can 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 [?]

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 to 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 **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.

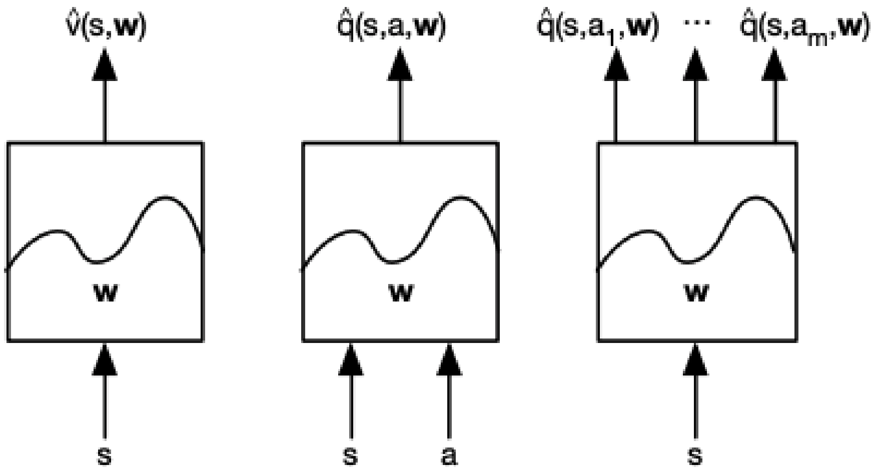


Figure 2.3: Types of value approximators, reproduced from Silver [?]

Figure 2.3 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', and this project explores the usage of neural networks. Since neural networks are function approximators, they a play a crucial role in RL in scenarios where the state space 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 **w**, J(**w**) , where the gradient of J(**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 **w** that minises the mean-squared error (MSE) between the approximated value function $\hat{V}(s, \mathbf{w})$ and true value $V_\pi(\sigma)$ and then use SGD to find local minima as defined in [?]:

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

However, considering the context is reinforcement learning, there is no objective truth for $V_\pi(\sigma)$, 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 precise 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 will 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 paramteres θ. However, the notion of 'best' is ambiguous, and 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 the 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 [?] 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 reintroduced the notion of the action-value function via value function approximation outline in subsection 2.4.4, whilst also aiming to reduce the variance.

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 paramters: w
- Actor - Updates policy parameters θ, in the direction suggested by the critic

The actor is the entity that actually makes the decision in the environment. The critic merely observes the actor, however, it evaluates the actor's decisions. The critic element 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 Discrete vs Continuous Action Spaces

The methodologies described in the latter part of the chapter are versatile because they support problem scenarios with 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. E.g 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 distribution.

Methods which can optimise for continuous action spaces are applicable to many real world scenarios which make the 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. The chapter started with providing the necessary context to understand how reinforcement learning is structured as a 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.

Chapter 3

Project Execution

This chapter intends to describe the entire journey of project. From the birth of the idea of the project, to the implementation and critical analysis of results. Talk about merging chapters 3 and 4 because the nature of project - intertwined

3.1 Origin

In a Bristol University Computer Science fourth year module called *Internet Economics and Financial Technology* Prof Dave Cliff and Dr John Cartlidge conducted intriguing experiments which 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), 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 architecture of this kind of learning is as follows:

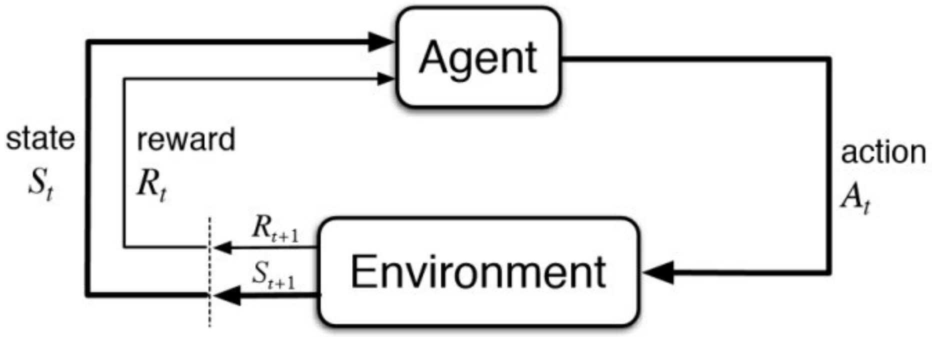


Figure 3.1: Architecture of Reinforcement Learning Environments

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

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 the thesis - called Bristol-Stock-Gym (BSG). BSG is a simulated trading environment that is a refactor of Cliff's Bristol Stock Exchange (BSE) that provides 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 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_
```

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 looks at the observation, takes an action and gives it to the environment to make one *step*. One *step* through an environment allows each trader in the market to react the the current market conditions by placing an order or by updating their 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 crucial details in the environment that are useful to the agent, 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 with the *bids* and *asks*, 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, however, 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 [?]. 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. This theory would be appropriate for a 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 has incorporated this ideology.

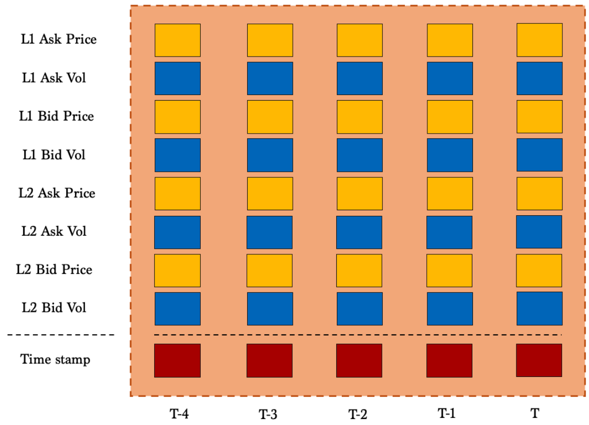


Figure 3.2: Capturing LOB Data

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 of the bids and asks and the current time stamp for the values that are currently on the LOB. It is important to note that the time steps are not necessarily consecutive in singular time steps, they are merely the time steps in which any change occurs to the non-time values 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 of it is constituted by 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 3.2 would take significant time to learn with a matrix of size *9x5*

The architecture of the implemented AE is as follows:

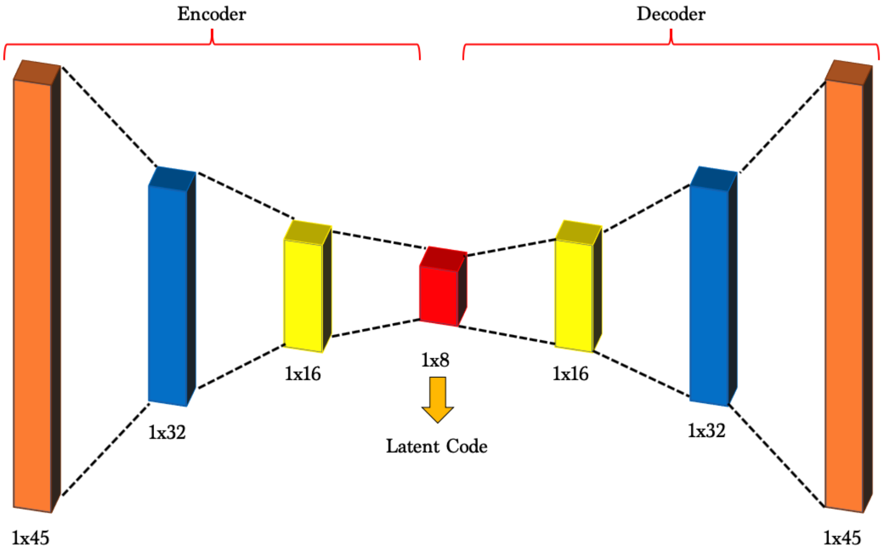


Figure 3.3: Architecture of Auto-Encoder

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 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 model was trained till convergence of test accuracy and saved for later usage. specify accuracy + talk about how the error is minimised between the input and output of decoder

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 profitable trades that occurred in the market session and the other 8 features that were extracted from the AE.

To incorporate more definitive 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 to inform the trader as to 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 having bought 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.

3.2.3 Varying Supply and Demand Schedules

3.2.4 Discretised Strategy

As detailed in Subsection 2.4.6 this context allows varying implementations of DRL techniques that utilise discrete and continuous action spaces. The actions 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 into 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 missing information from this strategy is that the price at which the trader submits a bid or ask order is so far 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 agent decides to place a buy order at price: $midprice_t$, the order may not necessarily lead to a transaction, it will have to be processed by the exchange.

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

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

3.2.5 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 current bid/ask order has not crossed the spread as explained in Subsection 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 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 is a good starting point.

3.2.6 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 Subsection 3.2.4. This implementation is based on the Monte Carlo Policy Gradient explained in Section 2.4.5, which means that the 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 incorporates the vanilla reward system.

3.2.7 Analysis and Evaluation of PG Vanilla

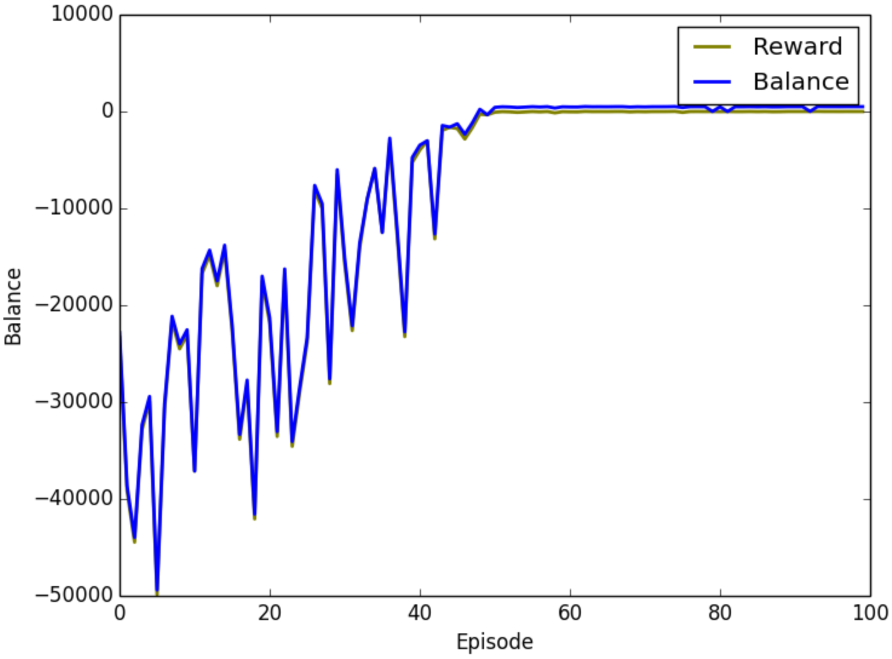


Figure 3.4: Total reward over 100 iterations of the PG Agent in a market evenly populated with ZIC, ZIP and GVWY traders.

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.

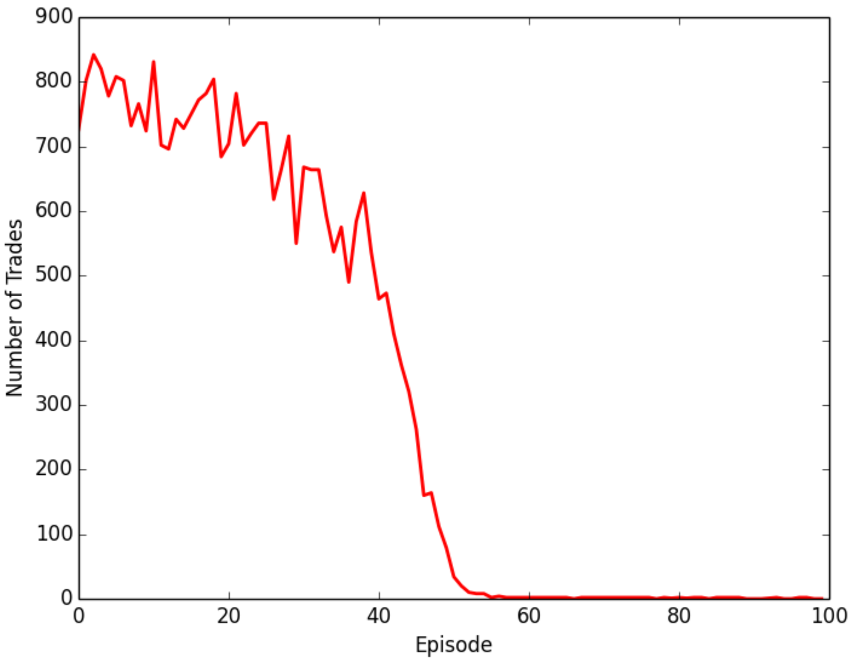


Figure 3.5: A plot of the number transactions the PG Vanilla agent engaged in during each episode

This was a hugely interesting result that was 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.

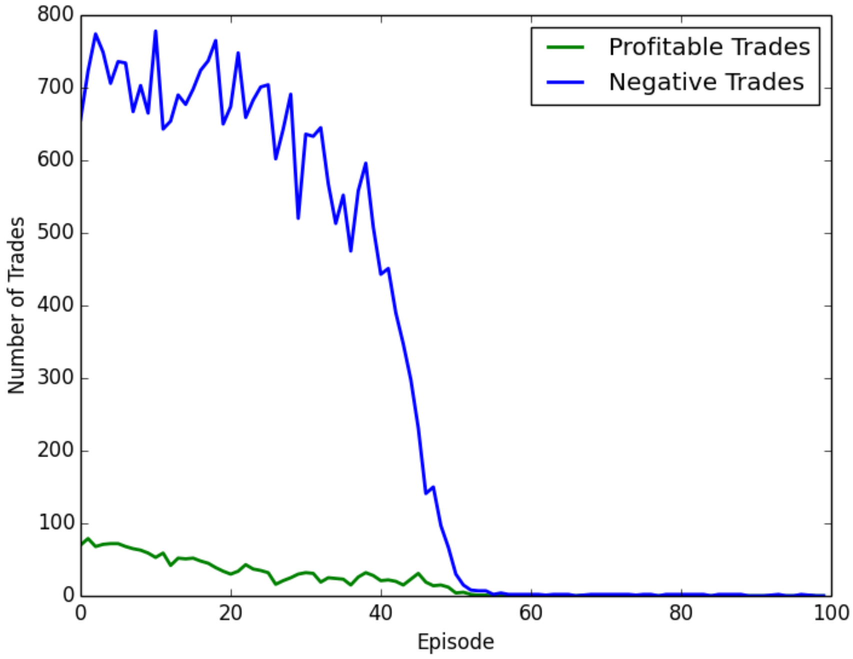


Figure 3.6: A plot of the number of profitable trades and trades that incur a loss for the trader

Figure 3.6 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. However, this result was interesting nonetheless. Since majority of the rewards received by the trader had been negative, it seemingly has learnt to increase its total reward by not trading at all. This is the 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 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 be unrealistic - as it lacked guidance for the trader to incentivise it to take part in good trades. 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, the chosen reward system being 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.

Add the average loss per bad trade and average profit from good trades

3.2.8 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’[**?**]. 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 Subsection 3.2.5, 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. 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) = 10 :`if (Action == SELL and trader.position == BOUGHT and current_midprice >price bought )`
5. R(t) = 10 :`if (Action == BUY and trader.position == SOLD and current_midprice <price sold )`
6. R(t) = -10 :`if (Action == SELL and trader.position == BOUGHT and current_midprice <price bought )`
7. R(t) = -10 :`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 2. is designed to amplify the rewards of profitable trades. The motivation behind this was, considering the trader does not engage in many profitable trades relative to bad trades, to heavily reward the actions that lead to profitable trades. The same logic was applied to punishing the trader for bad trades , however a smaller multiplicative factor was used; this was done 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 Subsection 3.2.9, thus weighting positive rewards more 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, 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 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 get data for this:

3.2.9 Analysis of Optimised Reward System

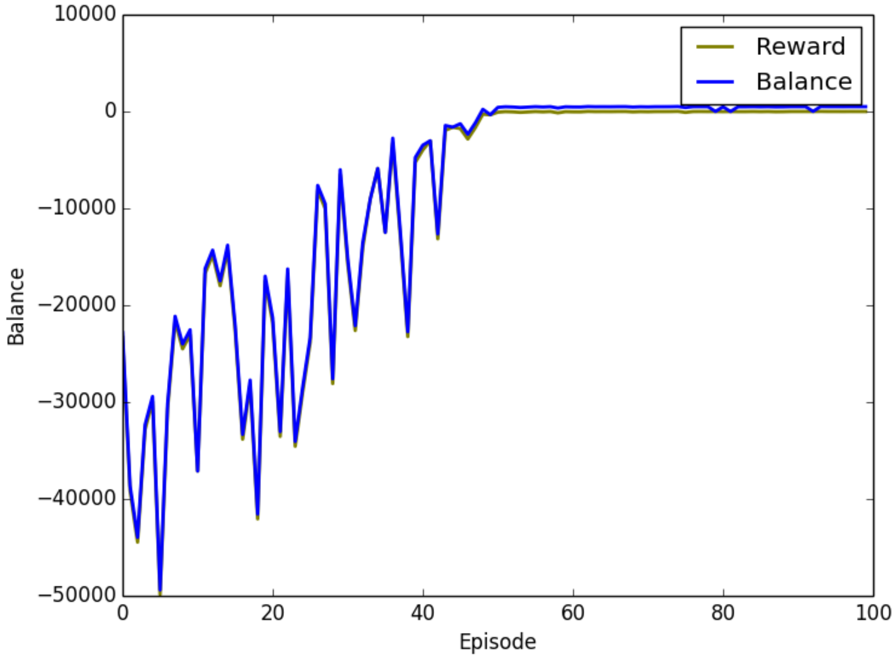


Figure 3.7: Total reward over 100 iterations of the PG Agent with an optimised reward system

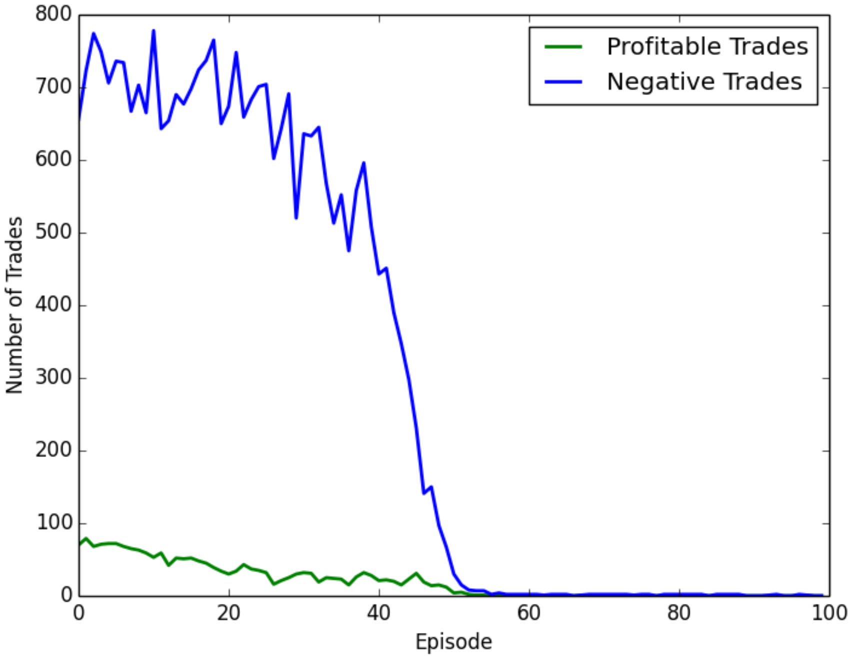


Figure 3.8: A plot of the number of profitable trades and trades that incur a loss for PG-Optimised

As seen in Figure 3.7, 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, however there are interesting key differences in these sets of results.

3.2.10 Continuous Actions

3.2.11 DDPG

3.2.12 Optimising Reward Functions

3.2.13 Sparsity in rewards

3.2.14 Simpler state space

Chapter 4

Critical Evaluation

Chapter 5

Conclusion

Appendix A

An Example Appendix

Content which is not central to, but may enhance the dissertation can be included in one or more appendices; examples include, but are not limited to

- lengthy mathematical proofs, numerical or graphical results which are summarised in the main body,
- sample or example calculations, and
- results of user studies or questionnaires.

Note that in line with most research conferences, the marking panel is not obliged to read such appendices.