Dissertation Type: Research

DEPARTMENT OF COMPUTER SCIENCE

Using Reinforcement Learning to Train Adaptive Traders in a Limit-Order-Book Financial Market

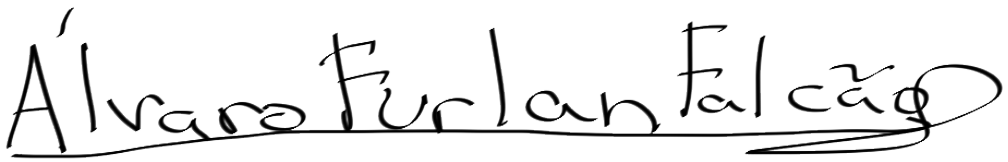
Alvaro Furlan Falcao

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 10th May, 2019

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.



Alvaro Furlan Falcao, Friday 10th May, 2019

Contents

1 Contextual Background **1**

1.1 Introduction . . . 1
1.2 Motivation . . . 1
1.3 Previous Work . . . 3
1.4 Central Challenges . . . 4
1.5 Conclusion . . . 5

2 Technical Background **7**

2.1 The Problem at Hand . . . 7
2.2 Traditional Trading Strategies . . . 9
2.3 Reinforcement Learning . . . 11
2.4 Summary . . . 18

3 Project Execution **19**

3.1 Origin . . . 19
3.2 Research and Learning . . . 19
3.3 The Bristol Stock Gym . . . 20
3.4 Agent Implementation . . . 23

4 Conclusion **35**

4.1 Challenge . . . 35
4.2 Outcome and Deliverables . . . 35
4.3 Future Work . . . 36
4.4 Last Words . . . 37

Executive Summary

Everyone is talking about Artificial Intelligence. As its achievements pile up and more possibilities open up, it is important for us to improve our understanding of it and to develop the tools necessary to fully exploit its potential.

There are few fields where AI has made such a big impact as in finance. Credit decision-making, automated trading or risk management are areas that have completely changed in the last years. However, the recent changes are but a taster of what is to come. Finance and AI still have a long path ahead to walk together.

This project aims to explore how the latest developments in AI – most specifically, Deep Reinforcement Learning – can be used in to improve established strategies in a specific financial problem: that of sales trading. Through that, I aim to make a small contribution in the study of AI and its impact in the financial services. In more detail:

> My research hypothesis is that the combination of the latest developments in Artificial Intelligence – specifically Deep Reinforcement Learning – with "traditional" heuristic approaches to automated trading should let us create agents for the sales trader problem comparably better than humans and the heuristic approaches alone.

This project explores that hypothesis and aims to answer it within the limited constraints in time, manpower and resources of a Masters project. To do so, I have done the following during the last few months:

- I spent 120 hours reviewing and learning techniques in machine learning, reinforcement learning and deep learning.
- I wrote the Bristol Stock Gym (BSG), a reinforcement learning library, using Python. This library implements a functional Limit-Order-Book financial exchange to be used as an environment for agent training. Its interface is designed to be most similar to industry and academia standards for similar libraries such as OpenAI Gym.
- I trained three different automated trading agents using BSG in the sales trader problem. Two of them are new algorithms that combine traditional heuristic automated trading strategies with state-of-the-art reinforcement learning techniques in a novel way.
- I compared those algorithms to traditional strategies, with a special focus in comparing the ZIP algorithm with a version that modifies it to use reinforcement learning instead.

If what I have done seems interesting to you, I encourage you to read ahead.

Supporting Technologies

This section presents a list of third-party resources and technologies used in this project:

- All the codebase is written using Python version 3.6.7 and the Python Package Installer PIP tool in its 19.0.3 version.
- I used the Python libraries Numpy(1.16.2), Matplotlib(3.0.3), Scipy(1.2.1), Scikit-learn(0.20.3), Tensorflow(1.13.1) and OpenAI Gym(0.12.1) and their dependencies in different sections of the project.
- I based my implementation of the Bristol Stock Gym (BSG) simulator / environment on that of Cliff's Bristol Stock Exchange (BSE), specially the low-level classes `Order`, `OrderBookHalf` and `Exchange`.
- I based my implementation for some algorithms in modified versions of snippets of code from an online MOOC by The Lazy Programmer Inc. [12]. This is mentioned again in the relevant sections.

Notation and Acronyms

While most notation and acronyms are only used explained and used once, some are ever-present across multiple sections and even chapters. They are as follows:

Acronyms

AI	:	Artificial Intelligence
BSE	:	Bristol Stock Exchange
BSG	:	Bristol Stock Gym
CDA	:	Continuous Double Auction
GOFAI	:	Good Old-Fashioned AI
LOB	:	Limit-Order-Book
MC	:	Monte Carlo
MDP	:	Markov Decision Process
TD	:	Temporal Difference
PG(A)	:	Policy Gradient (Agent)
RL	:	Reinforcement Learning
ZI-C	:	Zero Intelligence - Constrained
ZIP	:	Zero Intelligence Plus
ZIP-RL	:	Zero Intelligence Plus with Reinforcement Learning
ZI-U	:	Zero Intelligence - Unconstrained

Notation

$G(t)$	:	expected return at time t
$V(s)$	:	value function of state s
$Q(s, a)$	:	action-value function of action a at state s
α	:	learning rate
π	:	policy function
θ	:	model function (what it models is usually anotated in underscore, e.g. θ_π)

Acknowledgements

To my parents. Through hard work and effort they have given me the opportunity to make a difference in the world. Please take care. I love you.

Chapter 1

Contextual Background

1.1 Introduction

The Limit-Order-Book (LOB) market mechanism underlies many real-world financial markets such as major stock exchanges. Its rules and structure can be a powerful platform on which computer scientists can design, develop and analyze trading algorithms that can then be of use in real life contexts.

A good example is that of the role of the *sales trader*, an agent who does not hold stock of their own but instead buys or sells their client's stock – securing the best price for their client, for a commission. Importantly, a sales trader has to be *adaptive*, meaning it has to constantly observe and adapt to the state of the market to try to get a better price. Originally a job done by humans, sales trading was automatised during the 90's and early 2000's, with IBM showing in 2001 [7] that algorithmic traders could surpass human traders on the task.

Since then there has been relatively little research done in this problem. Subsequent solutions still used and improved upon heuristic algorithmic approaches and "old-school" Machine Learning techniques.

During the last decade new tools such as Deep Learning, Generative Adversarial Networks, Reinforcement Learning, and Genetic Programming have been applied to similar problems – explaining the explosion of research in investment algorithms. However, it is about time that we go full circle and start applying those new tools in the development of adaptive sales traders.

1.2 Motivation

1.2.1 A Computer Scientist's Perspective

In 2016, the Google-owned startup DeepMind made headlines after AlphaGo, an AI they developed, beat Lee Sedol, a human world champion, in a best-of-five Go match [18]. Go is an abstract strategy board game that was thought to be years from being "resolved" by a computer program.

However, AlphaGo (and its successors, AlphaGo Zero [20] and AlphaZero [19]) sent shock-waves across the field of AI research not only because they beat the most optimistic predictions by years, but also due to how they did it. While AlphaGo learnt by "watching" a large number of human games – a widespread approach known as supervised learning, AlphaGo Zero learnt completely on its own, playing games only

against itself – a paradigm known as Reinforcement Learning [23] that, while known, they combined with Deep Learning to form what they named *Deep Reinforcement Learning*.

AlphaGo Zero completely surpassed its predecessor, but its impact was larger than that. Being trained without any human input, many of its strategies were completely unknown to its human opponents, devising completely new lines of play that thousands of years of human research and mastery hadn't even thought possible. AlphaGo Zero changed the game forever, and nowadays professional Go players need AI assistants to train as much as they need human assistants.

The final iteration of this AI, AlphaZero, went on to generalize this approach to many other abstract strategy board games such as chess and shogi, becoming a master in all of them.

We live in an exciting time, where new advances in Computer Science allow us to glimpse better futures. Most AI experts agree that will change the world we live in. The controversial question is: Will that be for the better, or the worse?

While many believe AI heralds an age where humanity will achieve Utopia, others see in it a threat to our very own existence (see Harari [11] or Tegmark [24] for an in-depth analysis of this debate). The AI debate is very much open, with some even calling it "the most important question of our age" [24]. What we know for sure is that such question will not be answered without research into the possibilities and implications of AI in broader contexts. And one of those is, of course, Finance.

1.2.2 A Financial Perspective

Finance can be described as the science of allocating ("investing") resources (both good, "assets"; and bad, "liabilities") under conditions of uncertainty. While the same definition has usually been used to describe microeconomics, with finance being constrained to the allocation of money, scholars have argued that the past few decades have witnessed an unparalleled expansion of financialization (the role of finance in everyday business or life) [14]. Finance is becoming of paramount importance when analyzing and improving the well-being of our species and society.

The main reason for that is, of course, for its close relation with financial economics and money management. Asset pricing and budgeting are essential skills that individuals, families, corporations and even governments need to succeed. The right financial policies in all levels will see a society (and its members) prosper and live healthier, happier, more fulfilled lives.

On the other hand, finance is not limited to money-related problems, but can be used in any problem involving resource allocation. Prominently, market economics have become an useful metaphor to solve engineering problems dealing with the allocation of scarce resources. The technique known as Market-Based Control (MBC) [2] or Market-Based Resource Allocation (MBRA) provides us with fast, robust, distributed control for dynamically varying problems ranging from distributing air-conditioning loads within corporate facilities [27] to automated management of "smart" electric grids [16].

Therefore, it is no wonder that Computer Science has often been used to solve financial problems. It has been incredibly successful, and it is our duty as computer scientists to keep improving, always aiming higher.

1.3 Previous Work

1.3.1 From Experimental Economics to Algorithmic Trading

The sales trader problem can be traced to the famous experiments pioneered by Vernon Smith [21] in the 1960's, which served as both the origin of Experimental Economics and the reason why Smith was awarded the Nobel Prize.

On his experiments, Smith studied human market-trading behaviours under laboratory-style conditions. He split a small number of human subjects into "buyer" and "seller" groups, and gave each one of them a private limit price: buyers could not buy above that limit price, while sellers could not sell below it. Those two groups could interact within market mechanisms: buyers had the option to quote bid prices and sellers to quote offer prices.

Finally, this was repeated multiple times simulating trading days or sessions. Smith demonstrated that the price the traders quoted would rapidly converge towards the theoretical "best" price of the asset (known as the *equilibrium price*) with a very small number of traders needed.

This experiments were replicated by Gode and Sunder [10] in the 1990's, with an important extension: they replaced human traders with computer algorithmic traders. They introduced *zero-intelligence* algorithmic traders: unconstrained (ZI-U) traders that generate random bid/offer quote prices and constrained (ZI-C) traders that also generate random quote prices, but are constrained by their limit price to never trade at a loss.

Their objective was to find out how much of the allocative efficiency of a market is due to the intelligence of the traders, and how much of it is due to the organisation of the market itself. And their results spoke for themselves: while ZI-U traders were useless, the results from markets populated with ZI-C traders were surprisingly human-like, and they concluded that most of the "intelligence" of a market is within the market mechanisms and not the traders.

In 1996/7, Cliff [4] wrote a critique of Gode and Sunder, predicting conditions where ZI-C trader would fail to equilibrate and then empirically demonstrated those predicted failures. He developed a new algorithmic trader he called zero-intelligence plus (ZIP), which quotes prices based on a 'profit margin' it adjusts using a learning rule.

That was done in parallel to Gjerstad and Dickhaut [9], who developed what has since been known as the GD trading algorithm. It computes a belief function using data from recent market activity. That belief function represents the belief that a quote at a certain price will be accepted. That belief is then used to maximize the trader's expected gain.

In 2001, Tesauro and Das [25] from IBM ran a series of experiments in which they pitted different trading algorithms like ZIP and a modified GD (which they referred to as "MGD") against humans and between themselves in agent-vs-agent trading contests. They demonstrated that ZIP and MGD surpassed other trading strategies and, importantly, also surpassed human traders. Those findings were widely reported in the press of the time.

Since then, little has been done in this style of experimentation. Tesauro and Bredin published a new, improved version of GD they called GD-Extended (GDX) in 2002, and Vytelingum published the Adaptive-Aggressiveness Algorithm in 2006 [26]. Since then, AA has been considered the best in the published literature, and most efforts have been focused on improving it.

1.3.2 Recent Developments in Artificial Intelligence

While neural networks have been around for a long time, the victory of Krizhevsky *et al.* [15] at the October 2012 ImageNet competition started what many experts have considered the "deep learning revolution", with deep learning becoming a mainstream technique and being used to solve a wide range of problems – either on its own, or combined with other machine learning techniques.

On a similar manner, Reinforcement Learning (RL) has also been around for a long time [22], but limited to simple problems where it could prove more effective than other approaches.

However, during the last few years RL has had some notorious developments – especially when paired with neural networks. Techniques like Q-Learning offer possibilities waiting to be exploited, and those who succeed, e.g. the previously mentioned AlphaZero, are surpassing the expectations on the current limits of Artificial Intelligence.

1.3.3 In the Intersection between GOFAI and Intuition

While during the course of my project different approaches have been taken, I want to highlight the interesting insight provided by AlphaGo Zero: combining (Deep) Reinforcement Learning with GOFAI can yield surprisingly good results. That insight has informed many of my following decisions and shaped my approach.

In this case, I have considered the different algorithmic trading strategies developed during the 1990's and 2000's to be "GOFAI", and combined them with the AI techniques developed during the last few years. This, however, does not solve all many of the challenges inherent both in Machine Learning and, most specifically, Deep Learning and Reinforcement Learning.

1.4 Central Challenges

The main challenge for this project is to devise a Deep Reinforcement Learning architecture fit for the sales trader problem. This includes problems common in Machine Learning – including an analysis on different assumptions, hyper-parameters and feature extraction models – but also unique to Reinforcement Learning – such as transforming observations into states, devising a good reward function or an effective Exploration vs. Exploitation strategy.

This challenge will have to be tackled within time and computational resources constraints, making each decision taken during design and development important. In addition, there is a major challenge to overcome: an environment fit for training an AI in this problem does not exist, and so this project requires the development of a training environment prior to training the actual traders. Thankfully, there exists an already built LOB financial market in the Bristol Stock Exchange (BSE, see [5] and [3]), providing a starting point for the whole environment.

1.5 Conclusion

In conclusion, the high-level objective of this project is to use state-of-the-art deep reinforcement learning to develop adaptive algorithmic traders for the the sales trader problem. This will require:

1. An understanding of "traditional" machine learning approaches to the sales trader problem developed during the 1990's and early 2000's.
2. To learn both "traditional" reinforcement learning – as it is not part of the core machine learning teaching material in the University of Bristol – and its latest advancements when combined with deep neural networks.
3. To develop an environment where to train the agent, with fully working implementations of a Limit-Order-Book financial market, other algorithmic trader implementations and Vernon Smith-style experiment logic.
4. To design a working architecture for the agent, including choices in models and hyper-parameters – common in machine learning – but also in the reward function and state modelling – specific to reinforcement learning.

Chapter 2

Technical Background

2.1 The Problem at Hand

2.1.1 The Limit Order Book

A Limit Order Book is the data structure which serves as the pillar of modern electronic exchanges for Continuous Double Auction financial markets – that is, markets where an asset is continuously bought and sold between agents.

In its essence, an LOB is just a list of the price and quantity of all assets currently available for trade, both *bids* – offers to buy the asset – and *asks* – offers to sell the asset. In addition, it also offers some additional information about the current *trading session* – a single period of time of uninterrupted trading – a trader might find useful. In more detail, an LOB contains the following:

- The time within the current trading session.
- The *midprice* of the asset, the closest to a single price-tag for the asset, described as the unweighted average between the best bid and the best ask.
- Information on the latest executed offer.

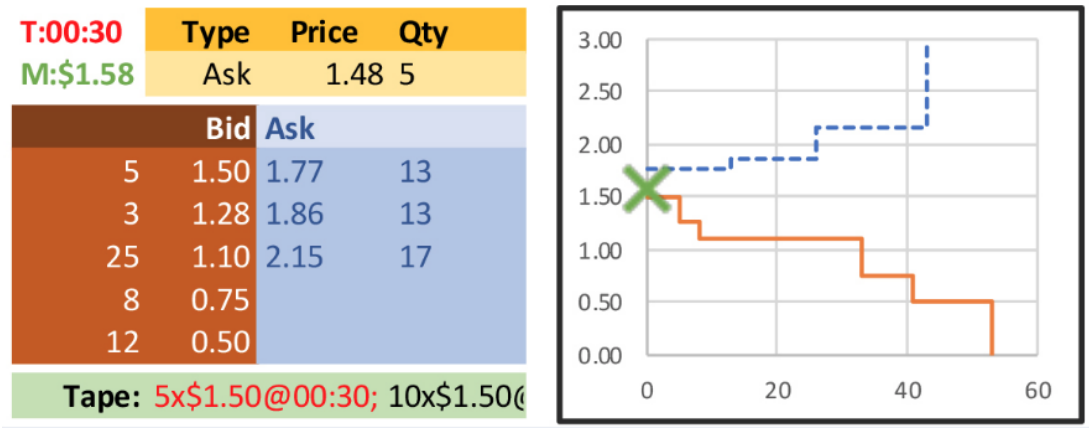


Figure 2.1: Visual representation of a Limit Order Book. Figure copied from [6]

- An ordered list (from higher to lower price) of all bids currently in the exchange and their price and quantity.
- An ordered list (from lower to higher price) of all asks currently in the exchange and their price and quantity.
- A time-stamped *tape* containing all trades executed within the current session.

2.1.2 "Game" Rules

The agents trained in this project take the role of a *sales trader*. As mentioned in the previous chapter, a sales trader does not hold stock of their own, but instead buys or sells the stock of someone else, their *client*.

Thea client provides the sales trader with orders for them to execute. Each order is comprised of a type (either bid or ask), a quantity and a *limit price* – the maximum price the agent can buy the asset or the minimum price they can sell it, depending on its type. The objective of the sales trader is to carry those orders, selling higher or buying lower than the limit price, for a profit. In the real world, that profit is then split according to some rule between the asset holder and the sales trader.

At any point in time within the trading session, the sales trader may be provided with a new order to execute. If the order is from the same client and for the same asset, that usually requires the agent to cancel their current quote in order to place a new one in the LOB. An agent is able to trade more than once per trading session.

2.1.3 Comparing Strategies

In order to objectively analyze the different strategies implemented in this project, it is important to understand how they are distributed within the experiment and compared to each other. For the sales trader problem, Tesauro and Das [25] introduced three different types of tests, classified with respect to their trader distributions. These will be used throught the project:

- Homogeneous population test: all the traders are follow a single strategy. This test is used for validating implementations and to produce baseline results for future reference. In this project, they were used during the development of the environment (BSG) to validate the inner workings of the exchange and the implementation of the non-reinforcement-learning algorithms. More details of this are given in the next chapter.
- One-in-many tests: these are used to explore vulnerability to defection/invasion and how a new strategy would do in an already-established market. In them, a single trader follows a different strategy than the others, in what would otherwise be a homogeneous exchange. These tests are the ones that I carried out for the main results of the project, with a single "AI" trader in a market populated with "traditional" traders. The logic behind the choice was twofold: it better explores how good an AI trader would do on its initial stages, and it can be carried out within the single-agent limit of the environment I developed – again, more on this in the next chapter.
- Balanced-group tests: these compare two groups of traders with two different algorithms, in equal proportions and with equal orders given to them. While interesting as they produce a "fairer" comparison than one-in-many tests, they would require both a much more developed environment allowing for multiple AI agents and enough computational resources to support those AIs – putting them outside the limits imposed by the resources available to the project.

2.2 Traditional Trading Strategies

The previous chapter presented a summary of different automated trading strategies of importance for this problem. A deeper understanding of some of them is important, as they serve as the foundation upon which the implementations of this project are built, and they are the ones those new implementations will be compared with.

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

In 1993 Gode & Sunder [10] introduced what they called *zero-intelligence* algorithmic traders for CDA markets. The first one of them, Zero Intelligence - Unconstrained (ZI-U) simply quotes random prices within the range permitted by the electronic exchange.

```
import random
# Classes needed to generate/deal with orders:
from Order import OType, Order
# Trader base class to be extended:
from Trader import Trader

class ZIU(Trader):

    def action(self, player_action, time):
        # If the trader has no pending trade orders, do nothing
        if self.order == None:
            return None
        # Return an order with a random price
        price = random.randint(
            self.exchange_rules['minprice'],
            self.exchange_rules['maxprice']
        )
        order = Order(self.tid, self.order.otype, price, ...)
        return order
```

Listing 2.1: Zero Intelligence - Unconstrained (ZI-U) – Gode & Sunder 1993

As one might imagine, ZI-U traders are plain useless. As they do not take into account their limit price, they very often for a loss.

2.2.2 Zero Intelligence - Constrained (ZI-C) – Gode & Sunder 1993

The other algorithm Gode & Sunder [10] presented was Zero-Intelligence - Constrained (ZI-C). Just like ZI-U, ZI-C quotes random prices. Unlike ZI-U however, ZI-C is constrained by the limit price, never trading at a loss.

```
import random
# Classes needed to generate/deal with orders:
from Order import OType, Order
# Trader base class to be extended:
from Trader import Trader
```

```
class ZIC(Trader):

    def action(self, player_action, time):
        # If the trader has no pending trade orders, do nothing
        if self.order == None:
            return None
        # Quote a random price, never make a loss
        if self.order.otype == OType.BID:
            price = random.randint(
                self.exchange_rules['minprice'],
                self.order.price
            )
        elif self.order.otype == OType.ASK:
            price = random.randint(
                self.order.price,
                self.exchange_rules['maxprice']
            )
        order = Order(self.tid, self.order.otype, ...)
        return order
```

Listing 2.2: Zero Intelligence - Constrained (ZI-C) – Gode & Sunder 1993

ZI-C is surprisingly human-like in many scenarios, making some amount of profit on the long run. However, it is easily beaten by better strategies.

2.2.3 Zero Intelligence Plus – Cliff 1997

In 1997 Cliff [4] wrote a critique of Gode & Sunder in which he demonstrated both analytically and empirically that the results in [10] were artefactual, and that there would be market situations where ZI-C traders would not reach a market equilibrium and thus would not exhibit 'human-like" market dynamics as mentioned before.

In addition, Cliff developed the Zero Intelligence Plus (ZIP) trader. This algorithmic trader has a "profit margin" that gets adjusted using a heuristic learning rule. In a short summary, the price the trader asks P is the limit price L plus some margin M i.e. $P = (1.0 + M)L$.

If the trader is selling, and other sellers are accepting bids below P or making offers below P, then the trader decreases the margin (but it stays constrained, that is, $M > 0$). If trades are happening at prices above P, the the trader increases M, even if they do not have anything to sell. The buyers do the inverse of this. The amount the profit margin is increased/decreased is determined by the Widrow-Hoff with momentum learning rule.

It is very important to note that, opposite to the previous zero-intelligence traders, ZIP is *adaptive* – it adjusts its behaviour according to the actions taken by the other traders in the exchange.

Cliff demonstrated that ZIP traders succeed in markets where ZI-C traders had failed, and that they actually behaved human-like.

Most importantly, in 2001 a group of IBM researchers (Das et al. [7]) ran a series of laboratory experiments where they compared human subjects with algorithmic traders. They demonstrated that ZIP traders "consistently obtain significantly larger gains from trade than their human counterparts". They concluded that "the successful demonstration of machine superiority in the CDA and other common auctions could

have a much more direct and powerful impact – one that might be measured in the billions of dollars annually".

That paper started a race to develop the best algorithmic trader that spanned the earlier 2000's, and continues to this day in AI projects like mine. Beating ZIP can be considered a worthy objective for an AI trader, as it also implies the trader is better than humans.

2.3 Reinforcement Learning

To understand this project, it is also important to know the basics of reinforcement learning: how the problem is defined and how it is solved using a range of approaches, from simple algorithms such as Random Search to more complicated ones such as Q-Learning or even Deep Q-Learning.

2.3.1 Defining Reinforcement Learning

Reinforcement learning is a field within machine learning concerned with training *agents* to take *actions* within an *environment* to maximize a *reward*. In more detail, we are trying to define the *policy* $\pi(a|s)$: the probability an agent should take a certain action a given an observed state s.

Markov Decission Processes

Given an environment and a state, a Markov Decission Process(MDP) is a collection of five elements:

- The set of all states, where a state s is defined as everything measured from the environment
- The set of all actions, where an action a is defined as anything an agent can do while in a state
- The set of all rewards, where a reward r is a value the agent obtains after doing an action
- The state transition probabilities for each pair of states, where each one is the probability that given a state s and an action a the environment evolves to state s'. Not all environments are deterministic: the same action in the same state might not yield the same future state. This might be due to randomness sources in the environment, the actions of other agents or imperfections in the agent's knowledge of the state.
- The discount factor γ, which encodes the notion that the same reward obtained now is more valuable that the same reward obtained at any moment in the future.

At every step of its existence, the agent tries to maximize its expected return:

$$G(t) = \sum_{t=0}^{\infty} \gamma^{\tau} R(t + \tau + 1) \tag{2.1}$$

Because rewards are probabilistic, and returns are sums of rewards, they are also probabilistic. We can define the expected value – an estimate of the return from this point onward. We call this the Value function:

$$V_{\pi}(s) = E_{\pi}[G(t)|s_t = s] \tag{2.2}$$

We can also define the Action-Value Function, an estimate of the return given we take an action from this point:

$$Q_\pi(s,a) = E_\pi[G(t)|s_t = s, A_t = a] \tag{2.3}$$

Q is better for choosing an action given a state, as it is indexed by the action. To use V to choose an action, first we would have to take every action, then see what the new state s' is to determine the best action. In real problems it is often not possible to "reset" the environment to try every possible action.

We can solve the value function recursively using a Bellman equation [1]:

$$V_\pi(s) = \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a)\{r + \gamma V_\pi(s')\} \tag{2.4}$$

However, two problems arise when trying to solve it:

The Prediction Problem

Given a policy π, find the value function for that policy

We can "evaluate" a policy by solving Equation 2.4. This can be done iteratively by solving the right-hand side and assigning it to the left hand-side until they converge – which they should, as the equation tells us both sides should be equal.

However, doing so is inefficient, and no "experience" from the environment is learnt whatsoever.

The Control Problem

Given a state, choose the action that maximizes the expected return

Choosing an action given the value function might sound easy: just iteratively compute the Action-Value Function (Equation 2.3) for every possible action given the current state, and take the maximum. Do so until convergence. This is known as *Policy Iteration*.

However, evaluating the policy (to be able to compute $Q_\pi(s,a)$ has already been mentioned to be inefficient without further improvements, and this is too. Together, both problems make this approach extremely inefficient.

2.3.2 Solving Reinforcement Learning

Monte Carlo Methods

A Monte Carlo (MC) method can be used to learn from experience. This is done by approximating the expected values with sample means:

$$V_\pi(s) = E_\pi[G(t)|s_t = s] \approx \frac{1}{N} \sum_{i=1}^{N} G_{i,s} \tag{2.5}$$

The returns for each state are stored during the episode. At the end of the episode the average of those returns is then used to evaluate the value function. This solves the prediction problem, with the only caveat that it forces the task to be episodic. Luckily most tasks can be divided in episodes, e.g. a game of chess or a trading session.

The control problem can be solved using policy iteration as explained before. Note that we still use $Q(s, a)$ instead of $V(s)$, as using the value function is not practical without full control of the environment and the ability to "reset" it to any point – just as explained when 2.3 was introduced.

In addition note that, because we are waiting to the end of an episode, the agent might get stuck in a loop and never end – this can be solved by detecting such loops, forcing the end of the episode and assigning a large negative reward to the action that started the loop. Also note that non-visited states remain unexplored – this can be solved by having a different random start state each episode, or by using the *epsilon-greedy* technique: have a (small) random probability ϵ of doing a random action.

Temporal Difference Learning

While in Monte Carlo we only learn at the end of each episode, temporal difference learning (specifically, the one known as TD(0)) is an algorithm unique to reinforcement learning that lets the agent learn at each time-step within an episode. The following derivation has been obtained from [23] and [12].

Instead of using the sample return G (2.1), we can use $r + \gamma V(s')$, which estimates the return from this point onward. r is still the sample obtained from playing the episode.

In the value function, We can compute the current sample mean from the previous sample mean:

$$V_t(s) = V_{t-1}(s) + \frac{1}{t}[G_t - V_{t-1}(s)] = \frac{1}{t}\sum_{t'=1}^{r} G_{t'} \tag{2.6}$$

This is, in fact, gradient descent – where the *learning rate* equals one over t. If we generalise it using α:

$$V(s) = V(s) + \alpha[G - V(s)] \tag{2.7}$$

$$V(s_t) = V(s_t) + \alpha[r + \gamma V(s_{t+1}) - V(s_t)] \tag{2.8}$$

Being able to evaluate the value function at each time-step, this solves the prediction problem. For the control problem, we can use SARSA [17] or Q-Learning.

SARSA is basically policy iteration applied to TD learning:

$$Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma Q(s', a') - Q(s, a)] \tag{2.9}$$

$$a' = \text{argmax}_a / Q(s', a) / \tag{2.10}$$

Q-Learning is similar but it does not matter which action is taken next: the target remains the same since it is a max of all Q.

$$Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \text{max}_{a'} Q(s', a') - Q(s, a)] \tag{2.11}$$

Approximation Methods

Up to this point we have reviewed reinforcement learning using tabular methods, i.e. treating $V(s)$ and $Q(s, a)$ as dictionaries. However, this will not work for the large or infinite amounts of states that most problems have.

Supervised learning can be used to approximate those functions, converting each state into a feature vector and then treating the return as the target.

$$\text{Input} : x = \varphi(s), \text{Target} : G \tag{2.12}$$

Since the reward and the return are real numbers we are performing regression. Therefore, the appropriate loss function is squared error.

$$E = \sum_{n=1}^{N} (G_n - \hat{V}(\varphi(s_n); \theta))^2 \tag{2.13}$$

To be able to perform gradient descent, the model needs to be differentiable.

$$\hat{V}(\varphi(s_n); \theta) = \theta^T \varphi(s_n) \tag{2.14}$$

Where θ is the model parameter, therefore what we need to update:

$$\theta = \theta - \alpha \frac{\delta E}{\delta \theta} = \theta + \alpha(G - \hat{V})\varphi(s) \tag{2.15}$$

2.3.3 Advanced Reinforcement Learning

Temporal Difference Lambda

Temporal Difference Lambda, better known as TD(λ) is a technique that finds an "in-between" approach between Monte Carlo and TD(0). Once again, this derivation comes from [23] and [12].

In TD Lambda we define the λ-return, a return made of the sum of all weighted expected returns, were the weights decrease geometrically and sum to 1. We denote those weights by λ.

$$G_\lambda(t) = (1 - \lambda) \sum_{n=1}^{\infty} \lambda^{n-1} G^{(n)}(t) \tag{2.16}$$

Assume an episode ends at some point in time, defined by T, and any other step after that is the full Monte Carlo return G(t). Doing some math wizardry we can reach that:

$$G_\lambda(t) = (1 - \lambda) \sum_{n=1}^{T-t-1} \lambda^{n-1} G^{(n)}(t) + \lambda^{T-t-1} G(t) \tag{2.17}$$

When $\lambda = 0$, we get the TD(0) return since $0^0 = 1$. When $\lambda = 1$ we get the full return just as in Monte Carlo. Any λ in between gives a combination of individual returns with geometrically decreasing weights.

However, this problem is not computationally practicable. To approximate it the notion of eligibility traces is introduced. An eligibility trace is a vector the same size as the parameter vector θ (see Equation

2.15) that keeps track of old gradients. It is made of the sum of the current gradient and the λ-weighted previous gradient. You can think of λ as telling us "how much" of the past to keep.

$$e_0 = 0, e_t = \nabla_\theta V(s_t) + \gamma\lambda e_{t-1} \tag{2.18}$$

And we update θ as follows:

$$\theta_{t+1} = \theta_t + \alpha\delta_t e_t \tag{2.19}$$

Policy Gradient Methods

Up until this point, to solve the control problem we have parameterized the value function while iteratively improving the policy – policy iteration. This was done implicitly by taking the max during policy evaluation, and it converges. However, we could model the policy too, and try to find the optimal policy π^*. This is known as Policy Gradient.

First, we can model a score for each action j:

$$score_j = f(a_j, s, \theta) (= \varphi(s)^T\theta_j \text{ if linear}) \tag{2.20}$$

And then use the $softmax$ function to turn this action scores into probabilities:

$$\pi(a_j|s) = \frac{exp(score_j)}{\sum_{j'} exp(score_j)} \tag{2.21}$$

Next we need an objective towards which we optimize the policy, and it has to be differentiable so that we can use gradient descent to reach it. We call that objective the *performance*, defined as the expected return of the entire episode.

$$\text{Performance} : \eta(\theta_P) = V_\pi(s_0) \tag{2.22}$$

$$\text{Policy} : \pi(a|s, \theta_P) = f(s; \theta_P) \tag{2.23}$$

$$\text{Value Function Approximation} : \hat{V}_\pi(s) = f(s; \theta_V) \tag{2.24}$$

Where θ_P and θ_V are the parameters of the policy and the value function, respectively. Then, thanks to the Policy Gradient Theorem and some more math wizardry (see [23] for the Theorem, proof of it and the full derivation):

$$\nabla\eta(\theta_P) = E[G\nabla_{\theta_P} log\pi(a|s, \theta_P)] \tag{2.25}$$

That is, the gradient of the performance is the expected value of the return from an episode times the log of the gradient of the parameterized policy. Then we can optimize that using either batch or stochastic gradient ascent. Note how this is a Monte Carlo kind of method (the agent only "learns" at the end of each episode).

Policy Gradient can be improved by adding a $V(s)$ as a "baseline" to G. We call $G - V(s)$ the *advantage*. Using this advantage instead of the return G has been shown [23] to significantly speed up the agent's learning. V is updated using gradient descent, as previously.

Actor-Critic Policy Gradient

Policy Gradient can be made into a Temporal Difference kind of method, where the agent "learns" at each individual time-step. This is known as the *Actor-Critic Method*. The "actor" in the name is the policy, while the "critic" is the TD error, which in time depends on the value estimate. To update those values every time-step, we substitute G with the single time-step estimate of G:

$$PolicyModel : \theta_{P,t+1} = \theta_{P,t} + \alpha(r_{t+1} + \gamma V(s_{t+1}) - V(s_t))\nabla log\pi(a_t|s_t) \tag{2.26}$$

$$ValueModel : \theta_{V,t+1} = \theta_{V,t} + \alpha(r_{t+1} + \gamma V(s_{t+1}) - V(s_t))\nabla V(s_t) \tag{2.27}$$

Why Use Policy Gradient

The Policy Gradient Methods, both MC and TD, have a couple of characteristics that it is important to understand. This is because they will come in handy once we start working in our actual problem, the adaptive sales trader.

These methods yield a probabilistic policy. And while this has been true since we introduced ϵ-greedy earlier in this section, these methods are much more expressive: instead of giving us a preferred action and a small random chance of doing some other action instead, now each action has a probability of being taken.

When is this specially important? When the states themselves are stochastic (that is, they have sources of randomness), because given a state the best action in that state might not always be the same one. Instead, each action will have a random chance of being the best one. This happens, for example, when the observation does not give us all the information in the environment. Most importantly, we have a huge source of randomness in the problem we want to solve: we do not know which action will be taken at any point in time by each other trader in the exchange. It is hugely important to have a tool that lets us take that uncertainty into account.

Continuous Action Spaces

On the previously discussed policy gradient methods, the output of the policy model was a discrete probability distribution on the action space. As a result, it would return something like "Action A has a 60% probability of being taken, action B a 30% probability, action C an 8% probability and action D a 2% probability".

However, we might not have a fixed discrete number of actions (like, A, B, C and D) but a continuum of values – for example, any integer between an electronic exchange's minimum price and its maximum price, as we have in the sales trader problem. That is known as a *continuous action space*.

The solution to this is to change the model of the policy from a discrete probability function to a continuous distribution: the Gaussian distribution springs to mind as it tends to be the most appropriate for things like this. That is, the policy becomes:

$$\pi(a|s) = \mathcal{N}(a; \mu, \nu) = \frac{1}{2\pi\nu} e^{-\frac{1}{2\nu}(a-\mu)^2} \tag{2.28}$$

Where μ denotes the mean and ν the variance of the normal distribution, obtained from the feature vector $\varphi(s)$ as follows:

$$\mu(s, \theta_\mu) = \theta_\mu^T \varphi(s) \tag{2.29}$$

$$\nu(s, \theta_\nu) = \exp(\theta_\nu^T \varphi(s)) \tag{2.30}$$

The exponential function ensures that the variance stays positive even when it is optimized using gradient descent, which is unbounded. A solution is to use the softplus function, where $\text{softplus}(x) = \ln(1+\exp(x))$. This function also ensures the values stays positive while having a behaviour most similar to that of the Rectifier Linear Unit (ReLU) function of common usage in Deep Learning.

$$\nu(s, \theta_\nu) = \text{softplus}(\theta_\nu^T \varphi(s)) \tag{2.31}$$

Note that the models are still updated just as before, using gradient descent/ascent optimization.

2.3.4 Deep Reinforcement Learning

As the complexity of the problem to be solved increases, the strategies previously described become worse and worse at providing results. Combining Deep Learning with Reinforcement Learning lets us deal with that increased complexity – the result is often termed *Deep Reinforcement Learning*.

Deep Q-Learning

Consider this Q-Learning model (which was already theoretically defined in Equation 2.11)

$$\theta = \theta + \alpha(r + \gamma(\max_{a'} Q(s', a')) - Q(s, a))\nabla_\theta Q(s, a) \tag{2.32}$$

As $Q(s, a)$ gets more and more complex, we can approximate this function using a deep neural network. Actually, we approximate the feature expansion of s, and make each output node correspond to a different action a. This network is known as a *Deep Q-Network*.

Learning: Experience Replay

When implementing an Experience Replay model, we train using (s, a, r, s') 4-tuples. We do not do Stochastic Gradient Descent with respect to the (most recent) TD Error, but instead we sample a random mini-batch from a First-In-First-Out buffer and do gradient descent with respect to it (the buffer is filled with random experiences at the start of an episode, also resulting in random actions). This works better than learning from the beginning to the end of an episode – like in the default Q update.

The Semi-Gradient Problem and the Dual Networks Solution

Recall that Q-Learning is a Temporal Difference method, and thus it uses the value function as its target return. The Q-approximation itself is used as the target for TD Error : this means that its target is not a true gradient to perform, but a "semi-gradient", which causes instability while training the deep neural network in the deep Q-Learning model.

The solution to this problem is to introduce a second Deep Q-Network, the *target network*. This network is responsible for creating the target for the TD error used by the original network. While a copy of that original network, it is not updated as often – this helps stabilize the targets.

2.4 Summary

This chapter has introduced in depth the sales trader problem, which I will be attempting to solve; traditional algorithmic trading strategies, that I will be trying to improve; and reinforcement learning theory that I will be using to implement RL models – those will then be combined with the traditional strategies, hopefully being better sales traders than those strategies.

Chapter 3

Project Execution

This chapter discusses the main stages the project went through, from the origins of its idea, through the learning, design, implementation and experimentation phases.

3.1 Origin

As it has been mentioned multiple times in this document, Artificial Intelligence has been making waves as a tool to radically transform many problems traditionally solved using other Machine Learning techniques, and tackling new ones previously unsolvable.

Therefore, it is not surprising that I was reminded of it as I learnt about algorithmic traders in one of my *Internet Economics and Financial Technology* lectures. If we could design good automated sales traders could be done by the turn of the century, what can we do with our currently technology?

After that lecture, I approached Prof. Dave Cliff with the idea to explore exactly that question. He was intrigued, and this project was born.

3.2 Research and Learning

Once I decided to tackle the sales trader problem using Reinforcement Learning, the next order of business was to learn as much as possible about it.

I started by doing a general, week-long review of the "pillars" of Machine Learning: theory on linear and logistic regression, an introduction to reinforcement learning [12] and a review of deep learning. I also went through the core Python machine learning libraries that form the Numpy stack, including Numpy, Pandas, Matplotlib and SciPy [13]. In addition, I learnt some more Tensorflow additionally from what I had learnt on my *Applied Deep Learning* unit (specially how to use interactive sessions) and learnt the basics of the OpenAI Gym library of reinforcement learning environments.

What then followed was a review of reinforcement learning following some of its most significant techniques. I started reviewing Markov Decission Processes – how they can be described and the problems found when solving them. Then I studied how to solve them: starting with Monte Carlo Methods and Temporal Difference Learning and finishing in Deep Q-Learning. I implemented solutions – based on

similar implementations in [12] – for the OpenAI Gym *CartPole* and *MountainCar* environments for each of the following algorithms:

- Random Search
- TD (Temporal Difference) Lambda
- Policy Gradient Method
- Deep Q-Learning

3.3 The Bristol Stock Gym

Once I started working on the core of the project – that of designing an automated agent for the sales trader problem – the most pressing question became that of which environment to use.

My initial idea was to use the Bristol Stock Exchange (BSE) [5] to train and test the agents. BSE simulates an LOB financial market and makes it easy to set up different experiments. However, BSE was designed with the purpose of running large batches of automated trading with the objective of generating and analyzing data, and not online real-time training of reinforcement learning agents. For that reason, I found its interface to be lacking some of the functionality needed for my specific usage.

In response, I decided to develop my own reinforcement learning environment. I based its logic on the LOB implementation of BSE, but modified parts of its logic and completely re-wrote its interface. I decided to call this new implementation the Bristol Stock Gym (BSG).

3.3.1 The BSG Interface

I based the interface of BSG in that of the OpenAI Gym – in fact, the interfaces are so similar that I believe making BSG into a gym-compatible module is fairly trivial. This is important, as Gym can be considered the "standard" reinforcement learning library for both industry and academia, and its standardized environments attempted as challenges (with calls for papers, leaderboards, etc.) by machine learning researchers and enthusiasts all over the world.

As an example, this is how the CartPole environment is used in Gym:

```
# Import OpenAI Gym:
import gym

# Set up environment:
env = gym.make('CartPole-v0') # Optional arguments can be added here
# Initialise parameters:
params = np.random.random(4)*2-1

# Define function to play one episode
def play_one_episode(env, params):
    # Reset environment, obtain initial observation:
    observation = env.reset()
    # Initialise done flag ant time counter:
    done = False
    t = 0
```

```
    # Play episode:
    while not done and t < 10000:
        # Increment time counter:
        t += 1
        # Choose action:
        action = get_action(observation, params)
        # Carry out action, obtain feedback:
        observation, reward, done, info = env.step(action)
        if done:
            break

    return t

# Run function
play_one_episode(env, params)
```

Listing 3.1: Setting up the CartPole environment in OpenAI Gym and running an episode

And this is how the same thing is done in BSG:

```
# Import BSG:
from BristolStockGym import Environment

# Set up environment:
env = Environment([...]) # Optional arguments can be added here
# Initialise parameters:
params = np.random.random(4)*2-1

# Define function to play one episode
def play_one_session(env, params):
    # Reset environment, obtain initial observation:
    observation = env.reset()
    # Initialise done flag and reward variable:
    done = False
    session_reward = 0

    # Play episode:
    while not done:
        # Choose action:
        state = generate_state(observation)
        price = get_price(state, params)
        order = generate_order(observation, price)
        # Carry out action, obtain feedback:
        observation, reward, done, info = env.step(order)
        # Store reward
        session_reward += reward

    # Return reward
    return session_reward
```

```
# Run function
play_one_episode(env, params)
```

Listing 3.2: Setting up the environment in BSG and running a trading session

As can be observed, the core interface is exactly the same: the environment is first created (with the option of taking arguments to customize it). Then, before it can be used it must be `reset()` – which returns the initial observation s_0. Then the `step()` method is called. It takes the action, runs a single time-step of the environment and returns a 4-tuple of the new observation, the reward, a done flag and a string with additional information.

While there are some helper methods and other functionality usually available in Gym – such as helper methods or the possibility of rendering video representations of the environment as the agent trains – the core functionality is still there, at it works pretty much the same as the stadard used in Gym.

3.3.2 Implementation Details

The core of the library is divided between five different modules:

- BristolStockGym : The highest-level module of the library, it defines the environment and the methods to interact with it – both their interface and the logic behind them.
- Exchange : Implements the Limit-Order-Book data structure and the methods necessary to use it as a stock exchange: how to process and delete orders and how to release a public version of it to be used by the traders.
- OrderBookHalf : Implements an ordered list of market orders that forms one "half" of the Limit-Order-Book, in addition to methods to add, remove and publish that list.
- Order : Implements the structure of an order and the enumerator OType (an order is either of type BID or of type ASK).
- Trader : Implements the core elements any trader should have, including its attributes and methods to process incoming orders from a "client" and to interact with the stock exchange. This class is then extended by other library scripts to implement a series of "traditional" automated traders such as ZIU, ZIC or ZIP.

3.3.3 Distribution

The library has been developed with the intention of making it public and open-source and can be accessed and downloaded from its GitHub repository [8] – note that no contributions will be accepted until this project has been marked. My intention is for it to be further developed until it becomes a more universal tool to train sales traders, and then make it into an OpenAI Gym module.

The plan includes publishing the library in the Python Package Index so that it can be easily installed using PIP, the Python Package Installer.

I have contacted people within the field – for example, machine learning instructors for Massive Open Online Courses such as [12] – with the intention of having them try it and provide feedback once it goes public. However, at the time of writing this document nothing has been set in stone.

3.3.4 Further work

There is so much more that could be done to improve the current version of BSG.

For starters the library is lacking functionality from BSE that I deemed unnecessary – or, at least, not useful enough to warrant the time an effort needed to implement it. That includes variations on the way orders are assigned to the traders and what those orders are, or a way of choosing experiment outputs into data files for posterior analysis, as that is not a focus of BSG in the way it is for BSE.

Secondly, BSG lacks functionality to train multiple reinforcement learning agents at the same time – to have AIs compete with AIs, instead of AIs competing with traditional strategies. While the lack of time was a factor here, another one was that currently OpenAI Gym lacks a clear interface for multi-agent environments in the way it does for single-agent environments. However, currently there are talks in the library repository discussing exactly this subject, and it would be an important thing to add once those talks are over and some standard starts to appear.

Finally, I believe there is a need for better verification techniques. This is a problem also present in BSE, but it is very difficult to prove that the results produced are 100% correct. Most specifically, the library lacks unit testing for the different components and integration testing as a whole. I do not believe that is a big problem for the current version, as it is slim and each individual component was manually tested (the code can be found in the `main()` function of each component). However, it is of utmost importance to automate the process if the library were to scale up as expected.

3.4 Agent Implementation

In this section I am going to talk about what is probably the most important part of the project: implementing the AI sales trader. Using BSG, I tried a series of different reinforcement learning algorithms and had them trade against traditional strategies. In the chronological order I tried, implemented and tested them:

3.4.1 Random Search

The Algorithm

Random Search is probably the simplest algorithm that can be considered a form of reinforcement learning. The algorithm works as follows: each iteration it randomly generates a parameter vector, and then "plays" a hundred trading sessions with it. At each time-step within the session, the random parameters are multiplied with the observed state to regress into a price. Then the agent modifies the price its bidding / asking for the asset accordingly. After the hundred sessions the average reward obtained is returned and, if its better than the previous best, the current parameters become the new "best" parameters.

At the end of the training period the best parameters found are set as the ones used by the agent in the testing phase. Note that this algorithm therefore lacks two of the most important reasons that make reinforcement learning so interesting in problems like the one we are trying to solve: firstly it does not do true online learning, but instead has separate training and testing phases; secondly, it is not adaptive: if the market conditions change after the training phase, the agent will start to greatly under-perform.

The algorithm is defined in the `random_search.py` file, with the core functionality being in the `random_search()` function:

```
# Random search algorithm
# Searches through N parameter vectors,
# each randomly selected from an uniform destribution between -1 and 1
def random_search(env, N = 100):
    session_averages = []
    best = -100000
    params = None
    for t in range(N):
        print("Attempt", t)
        new_params = np.random.random(4)*2-1
        print("Params:", new_params)
        avg_reward = play_multiple_sessions(env, 100, new_params)
        session_averages.append(avg_reward)

        if avg_reward > best:
            print("Set␣of␣params␣improved!")
            params = new_params
            best = avg_reward

    return session_averages, params
```

Listing 3.3: random_search.py - random_search() function

Designing the State

Defining the state is one of the most important design questions that need to be answered when implementing a reinforcement learning algorithm.

I could not use the full observation returned by BSG for the state: for starters, the Limit-Order-Book can be very large or infinite in size, and its size varies; while the state I needed had to be finite and of fixed size. Therefore, I decided to define my own state as a 4-tuple of two elements related to the "orders" given to the agent – the order type and limit price – and two elements related to the LOB – the best bid price and the best ask price. Then I created a function that transforms the observation given by BSG – published LOB and the order the agent has to execute – into that 4-tuple:

```
# Takes a BSG observation and returns a RS state
# A basic RS state is a 4-tuple
# - The limit price on the current order
# - The order type (-1 if BID, 1 if ASK)
# - The best bid
# - The best ask
def generate_state(observation):
    if observation['trader'].order == None:
        state = np.array([0, 0, 0, 0])
        return state

    order_price = observation['trader'].order.price

    order_type = 0
    if observation['trader'].order.otype == OType.BID:
```

```
        order_type = -1
    elif observation['trader'].order.otype == OType.ASK:
        order_type = 1

    best_bid = observation['trader'].exchange_rules['minprice']
    if len(observation['lob']['bids']) > 0:
        best_bid = observation['lob']['bids'][0][0]

    best_ask = observation['trader'].exchange_rules['maxprice']
    if len(observation['lob']['asks']) > 0:
        best_ask = observation['lob']['asks'][0][0]

    state = np.array([order_price, order_type, best_bid, best_ask])
    return state
```

Listing 3.4: random_search.py - generate_state() function

Choosing a Reward Function

Another very important question to answer when implementing reinforcement learning is that of the reward function: how should an agent's performance be measured?

The trivial answer is that of using the balance for that time-step, which is one of the returns of BSG's `step()` method. Let's think of the different scenarios, and what that implies:

- $r > 0$: This means that the trader has made a profitable trade. Obviously, this should be positively rewarded, but a question can be made as if it has to be a 1 to 1 reward. Maybe we could multiply the reward by 10, making good deals much more rewarding that simple "decent" deals. For now, I've left it as it is.
- $r = 0$: This is probably the most common reward returned by BSG, as it means the agent has not performed a trade this time-step. Again, the question of leaving it at 0 is not as trivial as it sounds: making it into a small negative reward would propmt the agent to trade more often, even at a small loss, while making it into a small positive number would make the agent trade less often (even when it could make a small benefit from a trade) and only trade with good deals.
- $r < 0$: This means that the trader made a trade, but lost money on it. This is penalized, but maybe we would be interested in the trader really interiorizing that trading for a loss is not a good idea – and to learn on its own the constraint hard-coded in the ZI-C and ZIP algorithms. This can be achieved, for example, by multiplying the loss by a large number, say, 100 or 1000. Again, for this specific implementation I will leave it as it is.

Results

The results of this Random Search implementation are very promising, and the agent already beat ZI-U traders easily. The agent was able to greatly increase its average rewards from an average -106284 to 8706 after a hundred training sessions, as can be seen in Figure 3.1, where it peaks at around the iteration 55. That best result is retained after training and used for testing. In testing, that translated to the agent making an average of £137.2 per three-minute session (quite consistently, with them rarely losing money) while the ZI-U traders were making an average £11.46 (with wildly varying results, as expected from a random trader). That means the agent is making ten times the profit!

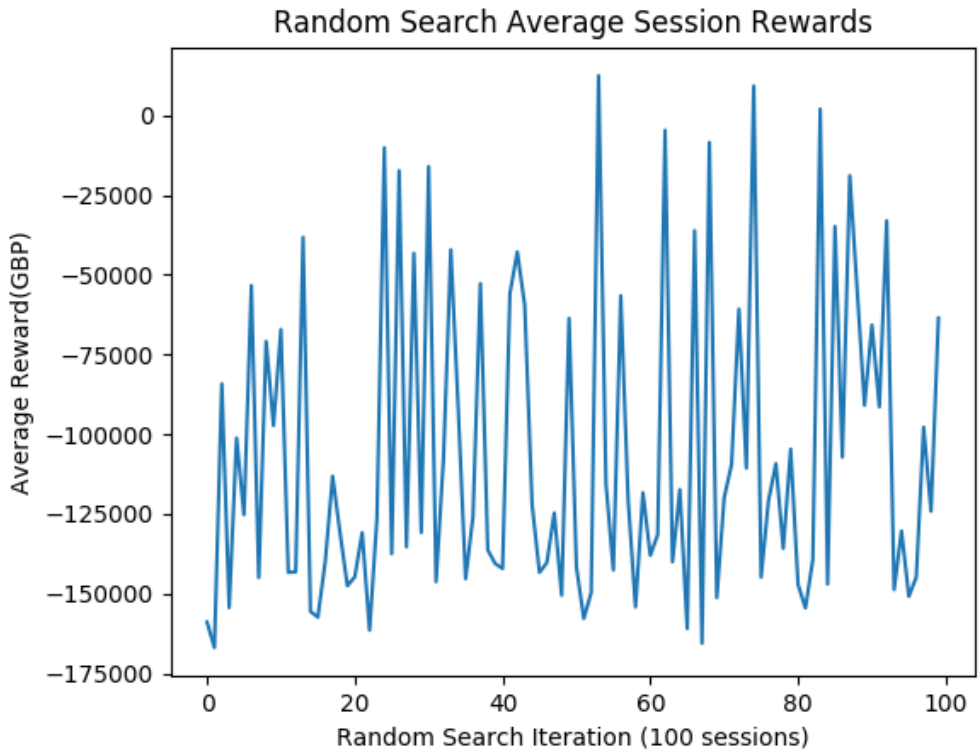


Figure 3.1: Average reward over 100 iterations of Random Search in a one-in-many experiment where the market has been populated by ZI-U traders. After training finishes, the best result is kept and used in testing, while the rest are discarded.

Intuitively, this means that "random with memory of best during training" – as one could think of the Random Search agent – beats "pure random" agents. We are already doing better than random. Nice!

Even more interesting are the results of this trader in a market populated by ZI-C traders. They are, obviously, worse, but the agent was still able to make major improvements. It went from an average reward of -55238 – smaller, as the ZI-C traders would often simply not trade with the agent, reducing losses – to a much better -245. This is shown in Figure 3.2, where the best parameters where found around iteration 60.

This is very important (and exciting!): the agent is learning to constrain itself to rarely trade at a loss, without the concepts of "money", "trading" or, specially, "loss" being coded into it. After training the agent was losing an average £27.25 per three-minute session while the ZI-C traders were making £7.35 per session. While it might seem like a bad result, it is much better than random: for comparison, in the same test (a one-in-many test in a market populated by ZI-C traders) a ZI-U trader was losing an astonishing £25074.5 average per session.

I believe these results are quite impressive, and show the promise reinforcement learning has for developing automated trading agents.

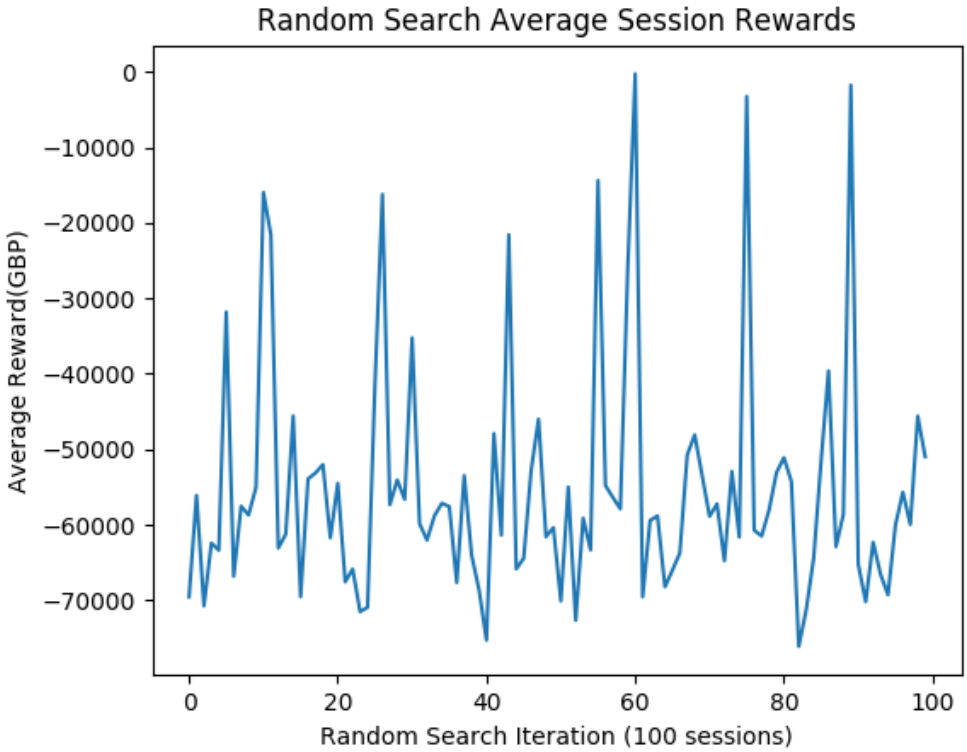


Figure 3.2: Average reward over 100 iterations of Random Search in a one-in-many experiment where the market has been populated by ZI-C traders. After training finishes, the best result is kept and used for testing, while the rest are discarded.

3.4.2 Policy Gradient Agent (PG Agent)

After the good promise shown by the results of Random Search, I finally could start working on a real reinforcement learning model.

My first idea was to train the model to price the asset at every time step. That gives us a continuous action space (from the minimum price of the exchange to its maximum price). Basically, this means that our agent will be solving a regression problem: given an observation of a Limit-Order-Book and an order to buy/sell with a limit price, regress to a price to buy/sell the item in the exchange. As the simplest method I know that can handle continuous action spaces, I decided to use the policy gradient method for continuous action spaces.

Designing the State and the Reward

While I kept the state design from the previous algorithm, firstly I decided to improve it by trying a feature extraction technique provided in the scikit-learn machine learning Python library: RBF Networks. The idea of adding an RBF Network between the input and the Policy Gradient model originated from [12], where it was successfully used for a similar problem. However, due to the way I designed the state, the input would not work well after being scaled – a necessary condition for it to be transformed before being fed into the model.

In the end, I decided to keep the same exact state as used in Random Search. The same is true for the reward function: while the trivial solution can surely be improved, I decided not to spend too much time in it as I considered this approach to be a "stepping stone" in the path to designing and implementing better agents.

Results

Figure 3.3 shows the average rewards of a PG agent in a market populated by ZI-C traders. While it might seem like a small number of iterations, the agent's performance greatly improves over the first fifty iterations (approximately) before stabilising for the rest of the test, and therefore I deemed it unnecessary to extend the training period – while in the graph it might seem like it is improving again, the general tendency is for the average reward to go up and down in a manner reminiscent of a sine wave.

Surprisingly for me, the results showed its performance is worse than that of the random search agent: after those 200 iterations of experience it was still loosing an average £1001.8 per session, while the ZI-C agents were making an average £48.77 – were the profits came mostly from trades with the PG agent.

On the other hand, this trader presents the advantages that the random search algorithm did not have. Firstly, it does true online learning: the agent keeps learning as it trades for more and more sessions. Secondly, it is adaptive: if the market conditions change, the trader will adapt to the new conditions without the need to modify the algorithm or even training it again. In other words: if a market shock were to happen – say, a huge increase in the offer, leading to the value of the stock being traded going down due to the law of supply and demand – the random search algorithm would start losing large amounts of money (being even worse than random) while with this strategy the balance would stay mostly the same – after a period of adaption.

If only that balance was not a loss. Luckily, there is more that can be done to improve the agent. For example, combining it with a traditional strategy.

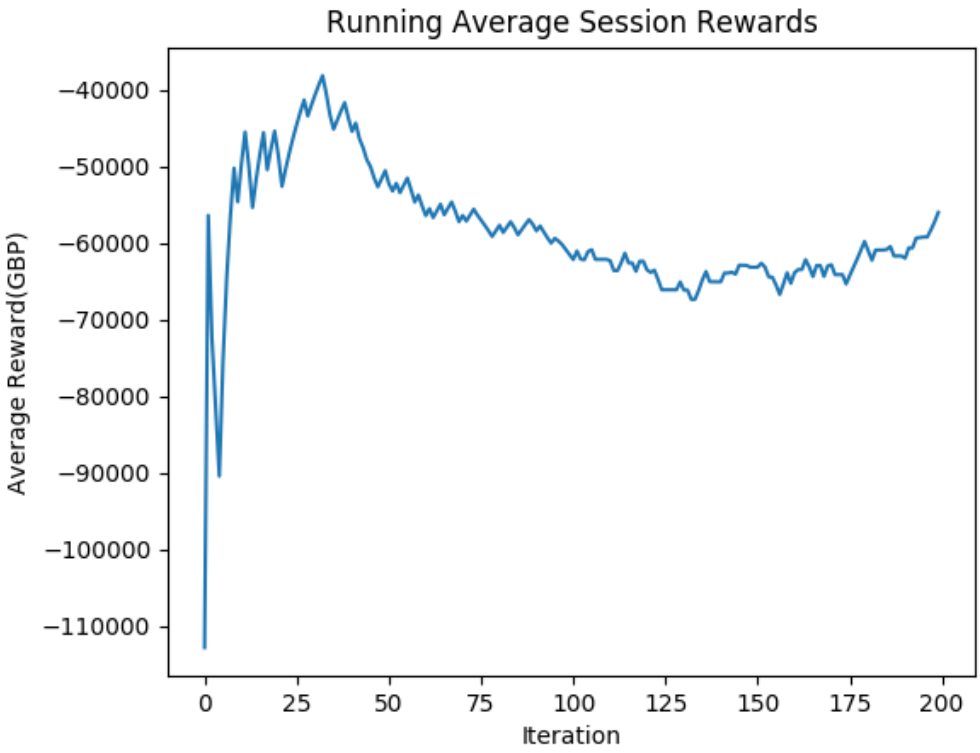


Figure 3.3: Average rewards for 200 iterations of a Policy Gradient agent in a one-in-many test in a market populated by ZI-C traders.

3.4.3 Zero Intelligence Plus using Reinforcement Learning (ZIP-RL)

After I failed to create a model that would successfully regress to the price, I decided to go back and try to solve my hypothesis: to combine algorithmic agents ("Good Old Fashioned AI") with reinforcement learning providing "intuition". In this case, I took Cliff's Zero Intelligence Plus algorithm [4] and modified it. Remember that ZIP has a "profit margin" M that it updates every time step using a heuristic learning rule (Widrow-Hoff with momentum). I decided to replace that rule with a continuous distribution model for the policy. I decided to call this variation Zero Intelligence Plus using Reinforcement Learning, or ZIP-RL to simplify.

Designing the State and the Reward

As in the previous section, I have kept the state to be a 4-tuple – the best bid, the best ask, the order type and the order limit price – and the reward function to be the raw balance.

While those work well, a possible modification for the state could be to include "time" data in the input. That would mean augmenting the dimensions of the input e.g. the state could be a 6-tuple of the best bid, the previous best bid, the best ask, the previous best ask, the order type and the order limit price. However, ZIP-RL already reached the target I had in mind without those modifications, and I left them to be explored in further work.

Other Implementation Details

In order to replicate the results presented in Figures 3.4 and 3.5 – which will be analyzed in the next subsections – there are some implementation details I want to go through.

Firstly, the architecture of the models. For both I used a "shallow" neural network with just an input and an output layer, with no hidden layers between them. The input layer is 4-dimensional, as those are the dimensions of our state.

The policy neural network has two output nodes – one regresses to a mean, the other to a standard deviation. Those values are then used to sample from a normal distribution, the output of which becomes the final output of the model – in this case, the margin the trading agent will try to make. The value neural network has one output node, which regresses to an approximation of V(s).

Secondly, the training. I used the ADAM optimizer, with the policy model having a learning rate of $1e-4$ and the value model having a learning rate of $1e-3$. In the policy model we are trying to minimize the cost (see Equation 2.26), while in the value model we are minimizing the TD Error (see Equation 2.27), with one slight difference: I take the whole of π and not its logarithm, as that gave me better results.

Results versus ZI-C

For the first experiment I ran a one-in-many test pitting a ZIP-RL agent against ZI-C agents. The results are shown in Figure 3.4. The modification, that at this stage I considered to be no more than an experiment, substantially improved the results with respect to the PG agent. After the initial training phase, the ZIP-RL was making an average £5.80 per three-minute session versus the £5.39 of ZI-C. It was a close margin, so I ran the experiment some more times. There were sessions were ZIP-RL would do worse than ZI-C, even losing small amounts of money, implying that the agent has not fully learnt not to trade at a loss. However, on average ZIP-RL was still coming on top.

Therefore, the next stage was to populate the market with the more advanced ZIP traders, and to compare how ZIP-RL fares against its older brother.

Results versus ZIP

The second experiment was running multiple one-in-many tests, where a ZIP-RL agent would trade in a market populated by ZIP traders. Beating ZIP is a most important milestone: as explained in Chapter 2, ZIP demonstrably beats humans in the sales trading test. Therefore, an algorithm/AI that beats ZIP can also be considered to perform better than a human being.

The running average session rewards are shown in Figure 3.5. The algorithm tends to take around 50 training sessions until it stabilises, but even then tends to be unstable: there are sessions where it averages a high reward – thus, we can expect it also made good amounts of profit – but others where it does not – and its profit was lower than that of ZIP.

However, the truth appears when we analyze the data numerically: while ZIP traders were making an average £1.72 per three-minute session, ZIP-RL agents were making £2.52 per session. Very surprisingly, they did better against ZIP than they did against ZI-C.

The good news aside, I believe it is important to analyze the reasons why this might be. Firstly, the experiment rules (symmetry, the rule replenishment schedule and the limit orders all being around the equilibrium price) favoured the simpler ZI-C. In addition, my assumption here is that ZIP-RL learned to "game" the experiment rules. The ZIP traders would rarely make offers that cross the spread or hit a

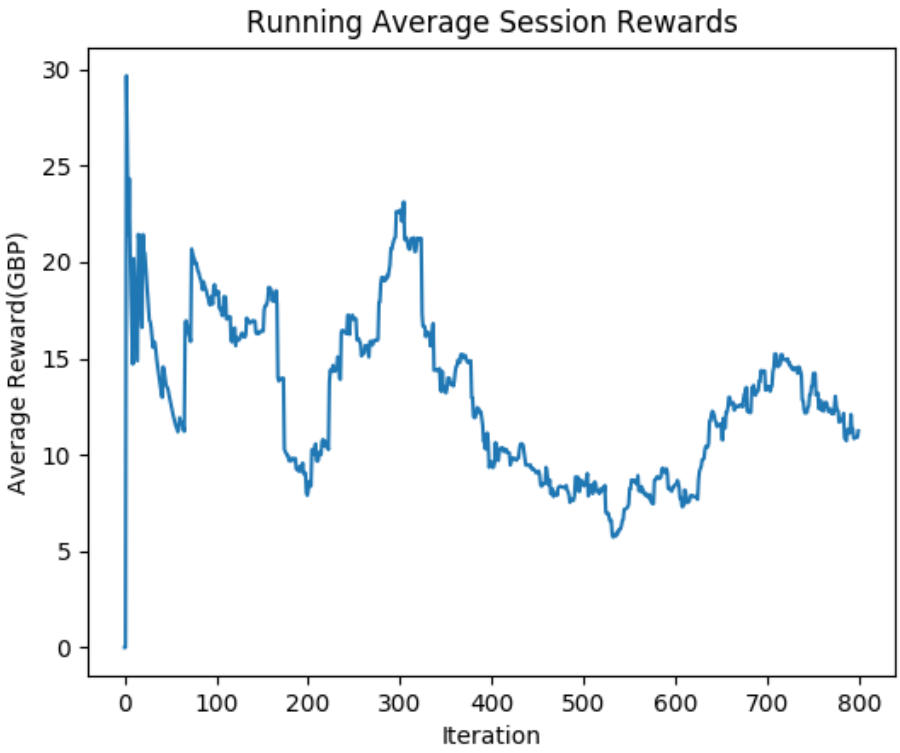


Figure 3.4: Average rewards for 800 iterations of a ZIP-RL agent in a one-in-many test in a market populated by ZI-C traders.

bit/lift an ask, a condition compounded by the short duration of the experiments. However, the ZIP-RL trader tended to want lower margins and, thus, made more trades over time. And while often the ZIP trader would get the most out of a deal, the fact that many of the traders populating the market made no deals (and thus had a balance of 0 at the end of each session) brought their average down lower than that of the ZIP-RL agent.

Long-Term Behaviour and Stability Analysis

Looking at Figures 3.4 and 3.5, one might wonder about the long-term evolution of ZIP-RL. Does it improve over time, or maybe get worse? To answer that question, I have run two long-term (spanning 10000 trading sessions) one-in-many tests: one in a market populated by ZI-C traders (Figure 3.6) and one in a market populated by ZIP traders (Figure 3.7).

The results support my claim that the reinforcement learning agents tend to have a fast learning phase right at the start and "stabilize" quite fast – after 50 iterations approximately. After that, they tend to vary but only within a range, with its ups and downs somehow resembling a high-frequency sine wave.

Additionally, even after the initial 50 iterations we can also see more variation ("instability") in the test versus ZIP than in the one versus ZI-C. I believe this is additional evidence to my analysis of the results presented in the previous section. While ZIP-LR consistently outperforms ZI-C, it can sometimes be outperformed by ZIP – even if it does outperform ZIP on average.

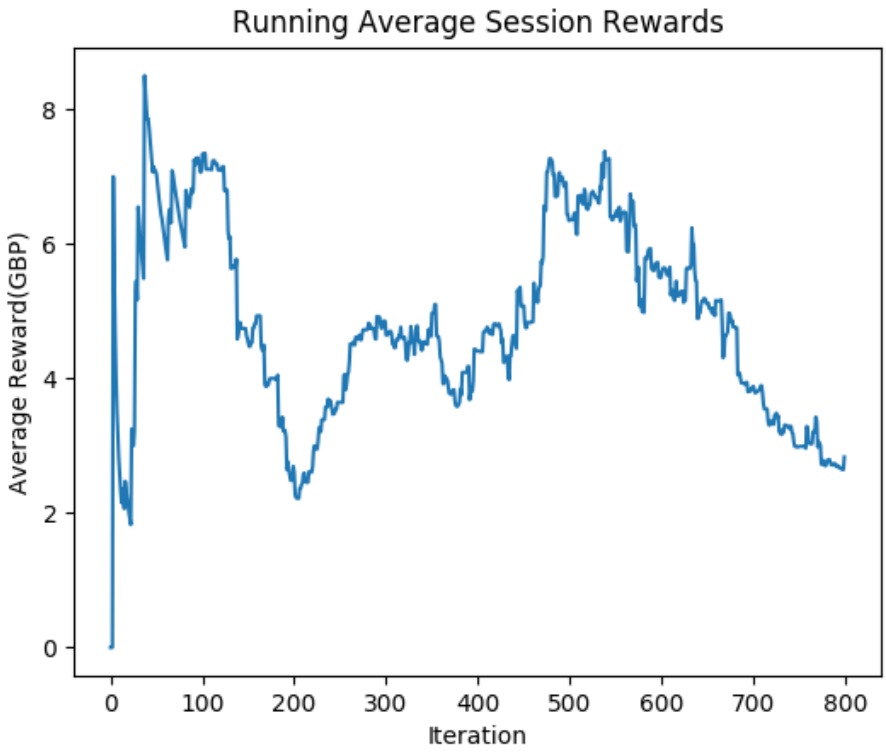


Figure 3.5: Average rewards for 800 iterations of a ZIP-RL agent in a one-in-many test in a market populated by ZIP traders.

Run Time Analysis

Separate to the algorithmic analysis, these tests can also provide us with some other interesting data: run time. In an Intel i7-6700K running at 4.0Ghz (4 cores, 8 threads) – the processor of a two-year-old high-end personal gaming computer – each set of 10000 three-minute trading sessions took around forty minutes to run. This translated in an average per 100 sessions of 23.05 seconds. And now that data supports that the agent has finished training after its first 50 to 100 iterations, I can confidently say that ZIP-RL takes from 11.5 to 23 seconds to train in a high-end personal computer.

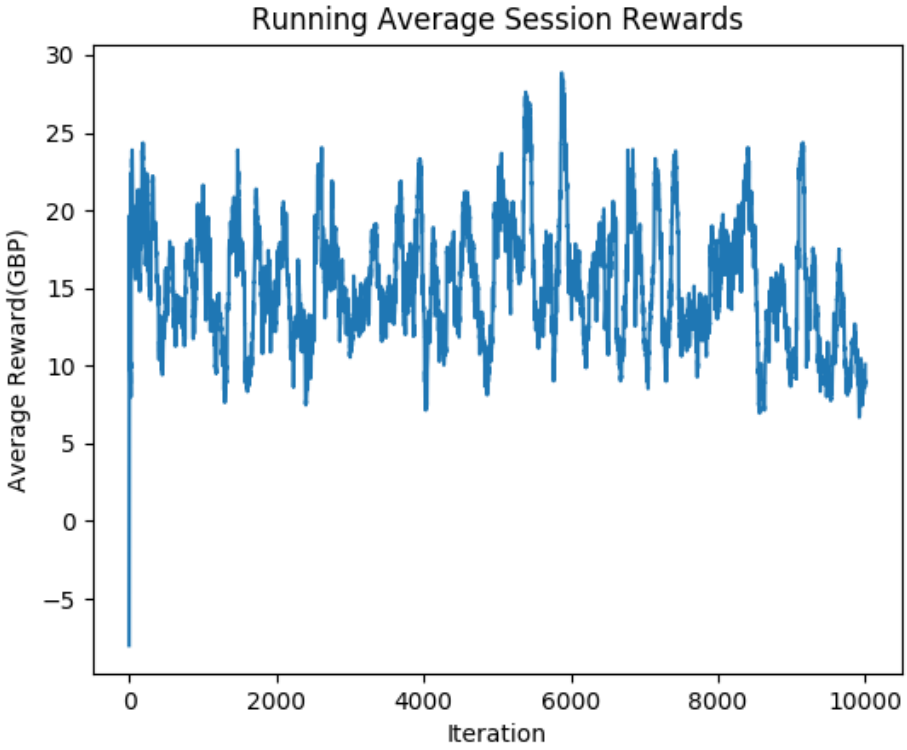


Figure 3.6: Average rewards for 10000 iterations of a ZIP-RL agent in a one-in-many test in a market populated by ZI-C traders.

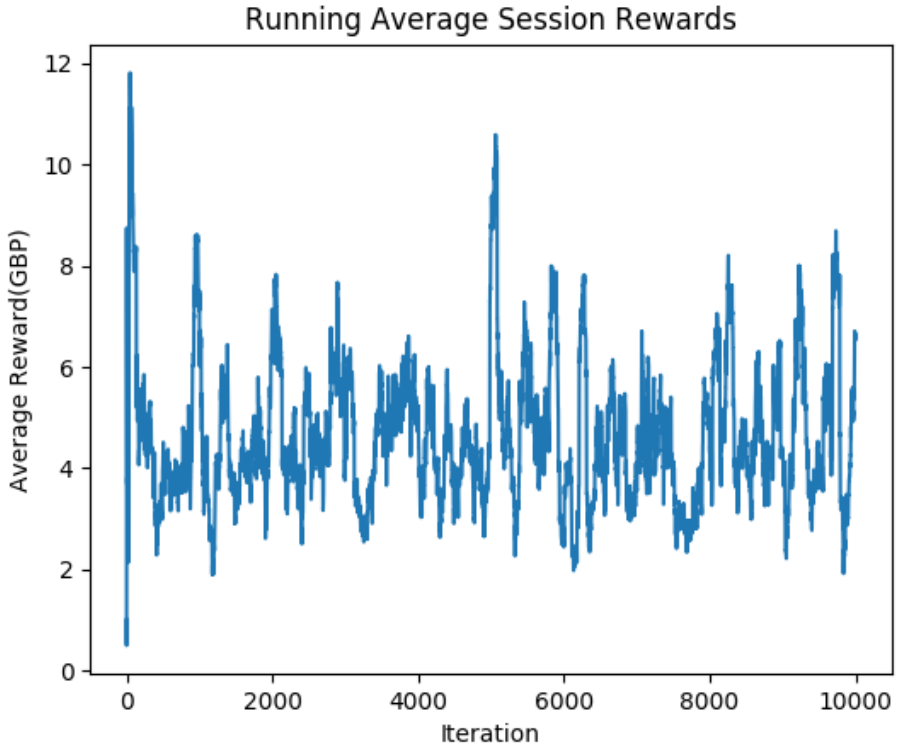


Figure 3.7: Average rewards for 10000 iterations of a ZIP-RL agent in a one-in-many test in a market populated by ZIP traders.

Chapter 4

Conclusion

In this chapter I intend to evaluate the project from a holistic perspective. The focus is not in the specific results obtained – those were analyzed in the previous chapter – but in the high-level challenges, what was achieved and the deliverables produced.

4.1 Challenge

This project was ambitious and posed a more than adequate challenge for a Masters project. The project is inter-disciplinary, with a combination of Computer Science (mainly machine learning for the AIs and software development for the Bristol Stock Gym) and Economics/Finance. That combination also made the work presented interesting from the start. As I described in the first chapter, AI is poised to change the world, and its impact in the financial markets is worth of study for both computer scientists and financial analysts.

That wide scope of the project meant doing large volumes of work within the limited time: at the start of the project I had little knowledge of reinforcement learning (much less state-of-the-art) and only knew the basics of algorithmic trading. But that was not the only obstacle I had to overcome during the early stages of the project: I faced the lack of a tool purposely built for training reinforcement learning agents, and thus had to develop one myself.

4.2 Outcome and Deliverables

Deliverable 1 : The Bristol Stock Gym

The Bristol Stock Gym became not only an essential tool for developing the algorithms present in this document, but also a good piece of software on its own right. While I have talked about it extensively (from design to implementation, its interface and usage) here I want to discuss it as the first deliverable of this project.

Firstly, I want to remind the reader that BSG was built on top of Cliff's Bristol Stock Exchange, which he uses to teach and research algorithmic trading. Interestingly, one of the main differences between BSE and similar libraries is that BSE does not use pre-existing market data to test its traders, but instead

simulates a Limit-Order-Book exchange where the traders get real-time feedback of their actions in the market. BSG was built upon it and retains the same idea: while there are libraries to train reinforcement learning agents in algorithmic training, they tend to either use sets of historical trading data, or connect themselves to the APIs of trading platforms for real-time data – obviating the fact that an extra trader would change how the market evolves from its very first action. Interestingly, some of those libraries even use the OpenAI Gym API or similar interfaces, as BSG does. As time passes, it is becoming more and more of a standard to develop environments that way.

Secondly, I developed the Bristol Stock Gym with the intent of distributing it as an open-source library (which has already been done, the library can be found at [8]), and easing its installation by publishing it in the Python Package Index. I plan on sharing it in the circles where experts and amateurs in reinforcement learning move – such as forums, blogging platforms (e.g. Medium), content sharing platforms (e.g. YouTube) and MOOC platforms (such as in the forums of machine learning/reinforcement learning courses in Coursera, FutureLearn and Udemy). With the right push behind it, I honestly do not believe it will die after this project has been submitted.

Deliverable 2 : A New Approach to Developing Algorithmic Traders

The second deliverable of this project is the answer to its research hypothesis. Recall:

> My research hypothesis is that the combination of the latest developments in Artificial Intelligence – specifically Deep Reinforcement Learning – with that of "traditional" heuristic approaches to automated trading should let us create agents for the sales trader problem comparably better than humans and the heuristic approaches alone.

Can the combination of reinforcement learning with traditional heuristic approaches to algorithmic trading beat humans and the heuristic traders alone? It seems so.

In the previous chapter I showed how ZIP-RL, an algorithm based on Cliff's ZIP but that regresses to a benefit margin using a policy gradient method (implemented with a "shallow" neural network) could beat ZIP. We can reasonably assume that, because ZIP was shown to be better than humans, ZIP-RL also is.

However, it is important to note that the original research hypothesis mentioned exploring deep reinforcement learning – that could not be done due to time and resource constraints. In addition, I would have liked to test this hypothesis in more situations – a more varied array of experiments, varying the market conditions. Most importantly, I would have liked to run balanced-group tests between ZIP and ZIP-RL. While this was impossible due to the lack of a multi-agent training environment, it gives us clues on what future work on the subject could expand upon.

4.3 Future Work

Future work originating from that I reported here could come in two different flavours: from improving what I have done and from the new projects and ideas that can be built upon it. While I have hinted about what could be done in the previous sections and chapters, in summary:

Firstly, the Bristol Stock Gym library should be improved with integration and automated unit testing. This would help validate the data produced by it, and make development possible as the software expands and scales up. Features for that scale-up go from simple auxiliary methods common in gym libraries – such as a method to sample from the action space – to advanced features like real-time video rendering

of the exchange (i.e. Chapter 2, Figure 2.1). A difficult to implement feature that should take priority is, however, that of multi-agent training.

With respect to the results presented in Chapter 3, I believe they are promising and sound. However, I would have liked if I had time to validate them as explained in the previous section in order to have additional assurances that those results are not artefactual.

But BSG remains a good tool, and the idea to combine algorithmic traders with reinforcement learning is even more compelling to explore now that it has shown promise. On the one hand, it would be interesting to explore how later-developed algorithms such as Vytelingum's 2006 Adaptive-Aggressiveness [26] could be augmented using reinforcement learning. On the other hand, it is only a matter of time until someone decides to combine a heuristic algorithm with deep reinforcement learning. While harder to develop and more resource-consuming to train, its capacity to deal with the complexity of a CDA market would probably produce very good results.

4.4 Last Words

I believe this was a significant, challenging project and that despite its difficulties I managed to produce a good software tool and to give an answer to the motivating research hypothesis – at least broadly, while lacking all the depth I would have wished for. I just hope this was a small stepping stone in expanding our understanding of AI and its possible future within the context of the financial markets, and that other people find BSG and my insights of use in their own projects.

Bibliography

[1] Richard Ernest Bellman. *Dynamic Programming*. Dover Publications, Inc., New York, NY, USA, 2003.

[2] S.H. Clearwater. *Market-based Control: A Paradigm for Distributed Resource Allocation*. World Scientific, 1996.

[3] Dave Cliff. Bristol stock exchange. Available at https://github.com/davecliff/BristolStockExchange, Accessed May 8th 2019.

[4] Dave Cliff. Minimal-intelligence agents for bargaining behaviors in market-based environments. *Hewlett-Packard Labs Technical Reports*, 1997.

[5] Dave Cliff. Bse: A minimal simulation of a limit-order-book stock exchange. In Michael Affenzeller, Agostino Bruzzone, Emilio Jimenez, Francesco Longo, Yuri Merkuryev, and Miquel Angel Piera, editors, *30th European Modeling and Simulation Symposium (EMSS 2018)*, pages 194–203, Italy, 10 2018. DIME University of Genoa.

[6] Dave Cliff. An open-source limit-order-book exchange for teaching and research. In *2018 IEEE Symposium Series on Computational Intelligence (SSCI)*, pages 1853–1860. IEEE, 2018.

[7] Rajarshi Das, James E Hanson, Jeffrey O Kephart, and Gerald Tesauro. Agent-human interactions in the continuous double auction. In *International joint conference on artificial intelligence*, volume 17, pages 1169–1178. Lawrence Erlbaum Associates Ltd, 2001.

[8] Alvaro Furlan Falcao. Bristol stock gym. Available at https://github.com/ElectronAlchemist/Bristol-Stock-Gym, Accessed May 9th 2019.

[9] Steven Gjerstad and John Dickhaut. Price formation in double auctions. *Games and economic behavior*, 22(1):1–29, 1998.

[10] Dhananjay K Gode and Shyam Sunder. Allocative efficiency of markets with zero-intelligence traders: Market as a partial substitute for individual rationality. *Journal of political economy*, 101(1):119–137, 1993.

[11] Y.N. Harari. *Homo Deus: A Brief History of Tomorrow*. Random House, 2016.

[12] Lazy Programmer Inc. Advanced ai: Deep reinforcement learning in python. Accessed May 8th 2019.

[13] Lazy Programmer Inc. Deep learning prerequisites: The numpy stack in python. Accessed May 8th 2019.

[14] Adam Hayes @ Investopedia. Finance definition. Accessed May 8th 2019.

[15] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors,

Advances in Neural Information Processing Systems 25, pages 1097–1105. Curran Associates, Inc., 2012.

[16] Sarvapali D. Ramchurn, Perukrishnen Vytelingum, Alex Rogers, and Nicholas R. Jennings. Putting the 'smarts' into the smart grid: A grand challenge for artificial intelligence. *Commun. ACM*, 55(4):86–97, April 2012.

[17] Gavin A Rummery and Mahesan Niranjan. *On-line Q-learning using connectionist systems*, volume 37. University of Cambridge, Department of Engineering Cambridge, England, 1994.

[18] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484, 2016.

[19] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018.

[20] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *Nature*, 550(7676):354, 2017.

[21] Vernon L Smith. An experimental study of competitive market behavior. *Journal of political economy*, 70(2):111–137, 1962.

[22] R.S. Sutton. *Reinforcement Learning*. The Springer International Series in Engineering and Computer Science. Springer US, 1992.

[23] R.S. Sutton and A.G. Barto. *Reinforcement Learning: An Introduction*. Adaptive Computation and Machine Learning series. MIT Press, 2018.

[24] Max Tegmark. *Life 3.0: Being Human in the Age of Artificial Intelligence*. Knopf Publishing Group, 2017.

[25] Gerald Tesauro and Rajarshi Das. High-performance bidding agents for the continuous double auction. In *Proceedings of the 3rd ACM Conference on Electronic Commerce*, EC '01, pages 206–209, New York, NY, USA, 2001. ACM.

[26] Perukrishnen Vytelingum. *The structure and behaviour of the Continuous Double Auction*. PhD thesis, University of Southampton, 2006.

[27] Yao Yao, Yizhi Cheng, and Peichao Zhang. Market-based control of air-conditioning loads with switching constraints for providing ancillary services. In *2018 IEEE Power & Energy Society General Meeting (PESGM)*, pages 1–5. IEEE, 2018.